UNIVERSITY OF OSLO
Department of Informatics

Towards Real-Time Depth Estimation in Large Spaces — A Soccer Case Study

Masters thesis

Henrik Kjus Alstad

Towards Real-Time Depth Estimation in Large Spaces — A Soccer Case Study

Henrik Kjus Alstad

Abstract

Many professional soccer sports clubs analyze their games either manually or using existing analytics tools. However, the existing software often requires much manual work and spending time looking through video tapes. Researchers and students from Simula Research Laboratory, The University of Oslo, and The University of Tromsø have created a real-time analysis system prototype called Bagadus. It is aimed at being automatic, easy to use and integrated with both annotations, recording software, video processing and sensor-networks. The latter is providing heartbeat, speed, position and various other statistics from the players.

Bagadus capture video footage from multiple angles, and delivers a stitched panorama video. A natural extension of a system using many cameras from different angles, is a free-view functionality. Free-view and virtual cameras means that we reconstruct an image from an imaginary virtual camera. This image is reconstructed using data from nearby real cameras. To reconstruct the scene as seen from the virtual camera, the depth of the pixels from the nearby real cameras must be known.

In this thesis, we show look at how we can improve the Bagadus system by camera calibration and stereo vision for depth map generation needed by the virtual camera application. We show how cameras can easily be installed and the system configured for the free-view, and how depth maps can be generated in real-time. Finally, we discuss see how results from this work can be applied to the Bagadus system.

Contents

List of Figures

Acknowledgments

I would like to thank my supervisor, professor Pål Halvorsen, and professor Carsten Griwodz for guidance, proofreading and helpful suggestions. I would like to thank Ph.D. candidates Vamsidhar Reddy Gaddam and Håkon Kvale Stensland for proofreading my thesis and providing helpful suggestions and encouragement. I would like to thank Luis Alvarez for explaining their paper [16] in more detail over a series of email exchange. Thanks to Marius Tennøe, Mikkel Næss and Espen Oldeide Helgedagsrud and Simen Sægrov who have also worked on the Bagadus system, for great discussions. Thanks to my family for their support.

Chapter 1

Introduction

1.1 Background

Many professional sports clubs analyze their games either manually with pen and paper or using existing software analytics tools, or a combination of the two. The ability of the coach to analyze the performance of the team, such as what the players do correct and how they can improve their performance, is important for the success. In this thesis, we look at soccer as a case study, but with modifications to our pitch model and such, other games such as ice hockey, basketball, handball etc, can also be considered.

To reduce the time and effort the coaching crew has to spend on reviewing games, a step in the right direction is using a computer system which can automate some of this work. However, the existing software often still require much manual work, such as looking through large video recordings of the matches.

As described by [32,47], numerous software systems already exist today. Interplay sports [6] have been used since 1994, but require coaches or trained personnel to manually look through the video material and note who makes a pass, who has the ball and so on.

ProZone [8] and STATS SportVU Tracking Technology [9] are systems that automates some of the manual work by automatic tracking of players. However, these systems lack analytics tools, and the STATS SportVU Tracking Technology uses only visual information to generate the statistics such as player tracking and velocity information. Tracking all players based on visual information is computationally expensive and sometimes relies on pure estimation in cases when the tracked objects or players can not be seen because they are hidden behind other objects or players.

Our system will be integrated with the ZXY Sports tracking [12], which offers a sensor-system where every player carries a small sensor that transmits data such as the position of the player to a nearby server. Thus, our system will track players much more accurately and less computationally intensive compared to the systems using only visual information. ZXY also provide our system with information not obtainable by cameras covering the pitch, such as heartbeat statistics. It samples these data at a resolution of 20Hz, which we can present with charts, 3D graphics and animations.

Camargus [2] is a system that delivers good video analysis, but lacks such statistics. By positioning 16 cameras in a camera rack from a fixed position, it allows relatively smooth transitions from camera to camera. Virtual cameras in general, can be chosen at any 3D point and orientation, and allows the user to move a virtual camera around in the scene. This can be useful

for the coach, if he want to see things from different angles, but it also offer a totally new way for the consumers to watch soccer matches.

In summary, there exist several systems today, both with respect to features and performance. However, to our knowledge there are no existing systems that fully integrate all the mentioned features. We have created a real-time analysis system prototype called Bagadus [32, 46] in cooperation with researchers from Simula Research Laboratory, The University of Oslo, The University of Tromsø, and ZXY Sports tracking [12]. The system is also integrated with Muithu [37] to provide optional manual input from the coach. For instance, using Muithu, the coach can tag events such as an attack using a mobile device. The event tagged by the coach will then automatically be synchronized with the video system and be ready to be played back at halftime. The system is aimed at being automatic, easy to use and integrated with both annotations, recording software, video processing and sensor-networks, with the latter providing heartbeat, speed, position, and similar statistics from the players.

Our system will have digital zoom, the ability of tracking players based on sensor readings, panorama view and single camera views covering all parts of the pitch, event annotations, player statistics, free-view and more. The prototype is now deployed at Alfheim Stadium (Tromsø IL, Norway). While most of the components in the system are already working, the free-view / virtual camera functionality is a work in progress.

1.2 Problem Definition

As seen, Camargus can offer a free-view, which we currently do not have in our prototype. But from investigation, we see that the Camargus system uses 16 cameras mounted in succession side by side vertically and horizontally in a small camera rack. Not only is this very costly, but it also narrows your free-view enormously. For instance, imagine a player standing in the middle of the pitch, and the Camargus rig is capturing the pitch only with cameras in the camera rig in front of the player. The system can not possibly know exactly how the scene looks behind the player if no camera has any field of view in the area. Thus, rotating 180 degrees to get a view of the stadium from the opposite side results in poor visual results as the system lacks data that is covered behind the players.

Papadakis et al. [44] have investigated a more dynamic free-view or virtual camera, with cameras further apart covering larger distances and angles, using 3D reconstruction. This involves using depth maps to try to estimate the third dimension, and recreating the scene in 3D space. Their results look good visually, however, according to their measurements, it takes between 5 to 10 minutes to process just 1 frame, using a NVIDIA GTX 280 graphics card and an Intel Core2 Quad CPU. For our system to be real-time, it must be able to do this 30 times each second. We will therefore measure our algorithms and investigate if we can do the steps fast enough, by using less computationally demanding algorithms at the expense of accuracy.

In this thesis, we will present an overview of the Bagadus system, depth map generation and the initialization and setup needed, with focus on real-time results suitable for a free-view system applied for a soccer scenario.

Although our current prototype is working, it is far from dynamic at the moment. Some modules such as the panoramic view generation and ZXY tracking is coded in a way that only works with the current setup. Even slightly moving a camera from its original position will cause the system to produce wrong results, and a new calibration, coding and recompiling

of the system will be necessary. As such, we will also apply several utilities discovered and implemented for the initialization of the free-view application, to the original system to aid in easy setup and a more dynamic and mature system.

1.3 Limitations

To create a working good quality free-view application that runs in real-time and is easy to install, is very complicated and will likely be too much for a single master's thesis. We will focus on the initialization and setup of the system, and depth generation suitable for a free-view application.

Banz et al. [17] implemented a less computationally depth estimation technique based on stereo vision, which performs at 27 frames per second (FPS) for 1024×768 pixel images on an Nvidia Tesla C2050 GPU. GPU processing like how Banz et al. implemented could definitely help the system handle larger images, and more expensive and more recent cards might perform at real-time, since Banz et al. were close to real-time with 27FPS.

We will only consider CPU optimization in this thesis, as the implementation of Banz et al. performed at less than our real-time threshold, despite running on an expensive GPU card, so achieving better results will likely be more time consuming and more expensive than what a masters thesis can cover.

We will present all the individual components of the pipeline and show that they can run real-time, but the implementation of a pipeline where all the modules are integrated, is also out of scope due to time limits. Although we mention free-view systems and offer some thoughts on it for the future, the main focus of this thesis is the depth map generation and initialization of the system.

1.4 Research Method

Our research method for this paper will be the design paradigm method of the ACM classification [25]. We will create a design and implementation of a prototype which then is tested and evaluated. The system will be tested both in our lab environment and on an actual soccer pitch; Alfheim Stadium in Tromsø.

1.5 Main Contributions

In this thesis, we give an overview of the Bagadus system and investigate real-time depth map generation suitable for a soccer free-view application, although it can be applied to other sports such as ice hockey, basketball and handball as well.

We will present a pipeline for generation of depth maps with an automatic and a manual approach for calibrating and initializing the pipeline. The individual parts of the pipeline will be implemented, and we offer some thoughts on how to make it real-time, but we will not integrate all the components into a working real-time pipeline.

Since the initialization and calibration for the depth estimation closely resemble the needs for the calibration for the existing Bagadus system, we will also apply our solutions to the

Bagadus system to help initialize its tracking system and panoramic view stitching. The calibration and applications for the existing prototype are tested at our system deployed at Alfheim stadium.

The calibration is made less cumbersome, by reading parameters from a configuration file and saving the output to a file in the XML format, allowing other applications such as a free-view or the existing Bagadus system, to read them easily.

The system is tested on data sets captured in the lab for optimal lighting conditions, positioning and ease of calibration, as well as being tested on a data set captured at Alfheim Stadium to test the system in a realistic setting with respect to lighting conditions and the long distance between the cameras and the players on the pitch. We will time each individual component to determine its real-time capabilities and efficiency, in addition to judging visual results mostly based on visual inspection. Since we are using our down data sets is it difficult to give numerical performance measurements with respect to visual quality and correctness. However, we will use existing depth estimations that are well explored.

Because the disparity decreases over distances, we compensate for this by increasing the distance between the cameras.

By splitting the original image into sub-images centered around the players, we show that the individual depth maps is calculated at 32ms, within our real-time constraint, for sub-images of sizes up to roughly 442×331 pixels. We show that the players depth can be estimated with an accuracy of roughly few meters. To increase the accuracy of the depth estimation a lens that gives more pixels per square meter or a higher resolution camera is required.

1.6 Outline

In chapter 2, we present an overview of the current prototype deployed at Alfheim stadium, its main functionality, and the overview of each of its subsystems such as player tracking, panoramic view, video capture, analytics subsystem etc. We also cover the most recent advances, such as the real-time pipeline. We discuss the free-view functionality, why it is needed, and outline the first steps for an implementation.

In chapter 3, we investigate different methods for calibrating and initializing the system to enable depth map generation. Solutions that can easily be applied to solve initialization and calibration for the original Bagadus prototype will also be covered here.

In chapter 4, we give an overview of what depth maps are, and how to calculate them, with the focus on stereo vision. Based on the results of [28] we investigate how the most promising depth estimation algorithms we have available with respect to speed and accuracy. We will apply these algorithms on our own data sets captured in the lab for optimal calibration, with our own calibrated system to verify the correctness and performance of our system.

In chapter 5, we investigate possible ways of optimization to make the depth map generation real-time for our free-view purposes. We will apply the algorithms to soccer related data sets to test our system in the field.

Finally, in chapter 6, we discuss our main achievements and future work.

Chapter 2

The Bagadus System

In this chapter, we will give a brief overview of the Bagadus system and the prototype deployed at Alfheim Stadium. In particular we will focus on the areas where we can apply initialization and methods learned from the work with setting up the depth estimation system later on.

2.1 The Basic Idea

Sports analytics have become more and more important with the recent years. Unfortunately, many of the existing systems require much manual effort to integrate the information from all the different systems like expert annotations and video system. A full analysis might not be finished until several hours after the game is finished. Bagadus [32, 47] is developed as part of the iAD-project [5], and aims at integrating video, analytics, tracking, and annotation systems. Also, the processing of the data should be done in a real-time fashion. Because of both limited funding, the challenge of it, and the potential usefulness of a low cost system, we will not be using very expensive hardware. Our system aims at offering the following functionality:

- Video recording
- Tracking of players
- Panorama and single camera views
- Digital zoom
- Automatically show video footage from a selected annotated event
- Other features are in development, such as a free-view functionality

As seen in figure 2.1, the Bagadus system is integrated with the ZXY Sports Tracking system, explained in detail in section 2.5, which we name simply ZXY in this thesis. ZXY got sensors on all the players transmitting location and other information, and inputs the information from the sensor-belts into a dedicated database.

Bagadus is also integrated with the Muithu system, explained in section 2.2, which is a novel notational analysis system, allowing the coach to manually tag events in classes such as attack or defense, and the players involved. The data from Muithu is being sent to a separate analytics

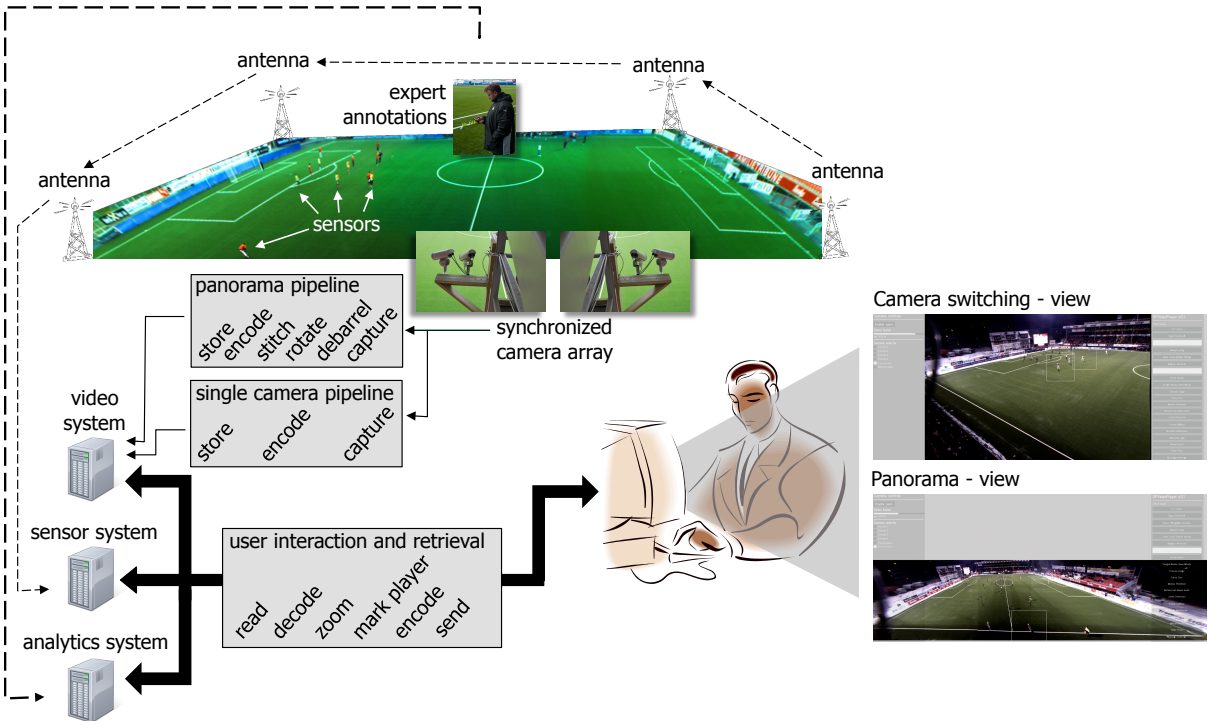


Figure 2.1: Overall architecture [32]

system, and the timestamps, type of event and involved players can be queried by the Bagadus system.

We use four cameras with wide angle lenses, covering the entire pitch, including areas of overlap to allow seamless stitching or transition between cameras. We generate both a panorama video of the whole pitch, and the separate four views. The video is also processed in a pipeline fashion and saved to disk.

A front-end system can then read data from all of the three systems, and create a real-time analytics system with the features previously described. The idea is then that the coach can use the system, with the annotations/events being ready to play with the click of a button, eliminating the need to manually search through the video. The two screenshots to the right in figure 2.1 shows a prototype front-end graphical user interface (GUI), running on Linux.

According to [14], television and cinemas have used a frame rate of 24 and 30 frames per second (FPS) for a long time. Our system delivers 30 FPS, on par with common television and cinema standards. This means we have $\frac{1000ms}{30frames} = 33.33ms$ to process one frame. Real-time in the context of our system means processing the data associated with a set of frames from the cameras within the $33.33ms$ real-time constraint. However, we do accept an initial delay of approximately 2 seconds, so that our system can have a latency of some seconds after the real world events, due to buffers and pipeline components being initialized. Imagine pouring water through an empty water pipe. Initially, it will be some delay, but once the pipe is flooded, it will arrive at a constant rate.

2.2 Analytics Subsystem

Muithu [37] is a system allowing coaches to tag events, such as when a player is making an attack, a good defense, and so on. The coach tags these events by clicking on the involved players icons on an app on his smart phone, and the system saves it in a database. It is developed by the University of Tromsø, but integrated with the Bagadus system through our front-end system. When the coach wants to show the corresponding video material, all he has to do is to click on the event-button showing up in the Bagadus user interface, and it will automatically play the video material, tracking the involved players inside a tracking box and showing the correct camera footage.

2.3 Video Capture

In our current setup at Alfheim stadium, we use four Basler acA1300 - 30gc cameras [1], seen in figure 2.2. To cover the entire pitch with 4 cameras, including some overlap for stitching the images to a panorama, we use Kowa 3.5mm wide angel lenses. The cameras have a max resolution of 1289×964 pixels, a frame rate of 30FPS, and delivers images in the format YUV4:2:2P. In YUV4:2:2P [14], Y is the luma, and UV is the chroma components. 4:2:2 means we use chroma subsampling, sampling the two chroma components less often than the luma component. The standard considers the number 4 to represent the full sampling, while 2:2 means we sample the two chroma values at half resolution than what they used to be. This means that we loose some chroma resolution. However, the human eye is less sensitive to chroma than luma, in fact, the human eye can not even see the full 4:4:4 resolution chroma. The gain by using 4:2:2 is that we reduce size of the images, and therefore the bandwidth usage between the cameras and the computer, by a factor of $\frac{4*height*width+4*height*width+4*height*width}{4*height*width+2*height*width+2*height*width} = \frac{2}{3}$. For the interested reader, see [47] for a more in-depth explanation.

Figure 2.2: The Basler ACE camera [1].

One of the selling points of the system is that it should require as little manual work as possible. As such, we do not want to require the operator to setup or control the cameras during the game, it should be enough to press a start and stop button. We offer this functionality by covering the entire pitch with the 4 cameras, so that manual control or setup of the cameras is unnecessary once the cameras are correctly deployed. They should ideally be positioned with equal spread along the width of the pitch, to cover the most ground, and maintain the same

angles. However, because our trigger boxes (explained in section 2.3.1) have a limited cable length, we had to use the camera setup as seen in figure 2.3.

The cameras are currently connected in pairs to two computers. The computers have the following specifications: “Intel Core i7-2600 @ 3.4 GHz, 8 GB system memory and NVIDIA GTX 460 graphic card”. As suggested in previous work [47], we later managed capture footage with all four cameras connected to a single computer of the same specification without frame loss at 30FPS using an Intel Server Adapter i350-T4 with 4 network interfaces.

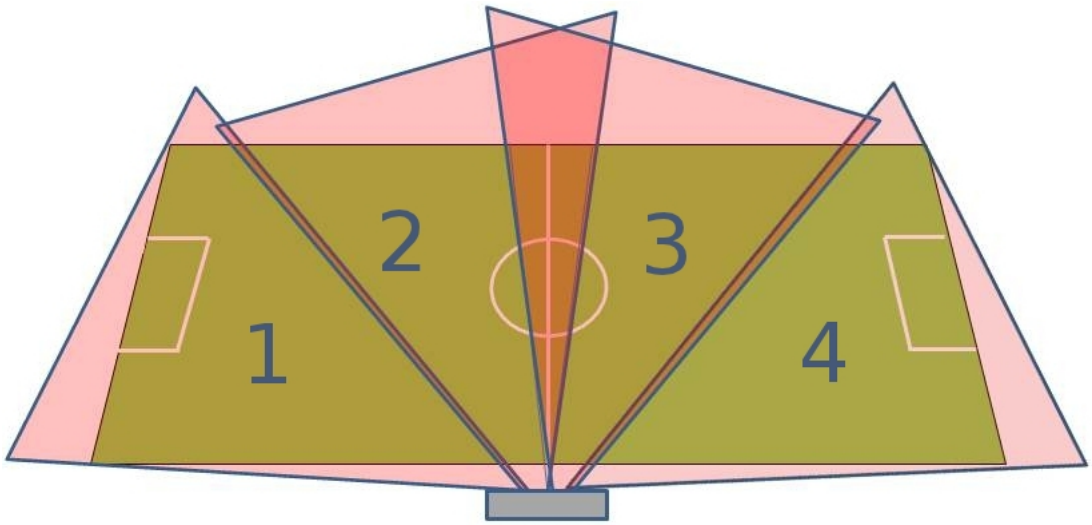


Figure 2.3: Our current camera setup [47].

2.3.1 Synchronization

Making a panorama image out of several independent images captured at different viewpoints is an important goal of the Bagadus system, giving an overview of the entire pitch in a single high resolution image, eliminating the need to change the active camera view depending on which part of the pitch should be visible. It is imperative that our cameras capture the images at the exact same time. Without such synchronization, the different images do not correspond to the same point in time. As players are often moving very fast, even a minor difference in the capture time might introduce visual abnormalities like a player appearing at two locations at once, if he is running in the overlapping area of the two cameras. As such, we need a shutter level synchronization on the cameras. This is achieved by using an external synchronization device we will refer to as the “trigger box”, developed at Simula Research Laboratory. The trigger box is connected to all four cameras, sending a signal arriving at the same time at all the four cameras signaling when to capture an image. The trigger box allows for adjusting the wanted rate of signals being fired each second, directly corresponding to our wanted FPS. The trigger boxes can also be wired together, supporting more than just four cameras. The old setup did as mentioned make use of two separate machines to capture data, as one machine was not sufficient to calculate everything in real-time, due to limited hardware. Even though the cameras have been synchronized, the frames arrive from the cameras without any timestamp data. Knowing when a frame was captured is useful in synchronizing with the ZXY database, which is a separate system operated by the ZXY Sport Tacking company. Because the system

initially lacked a reliable Internet connection, utilizing the Network Time Protocol (NTP) [26] was not considered. The synchronization of the two computers was done by using the "TimeCodeServer" [47]. The TimeCodeServers implementation is based on the well known Berkley algorithm described in [31]. However, we later got a fast Internet connection to the stadium, and we thus started using the NTP instead, as it was less error prone and easier to use than the TimeCodeServer.

2.3.2 Software

The cameras come with their own C++ SDK called "Pylon SDK". The SDK allows controlling a vast range of camera functions such as auto exposure, white balance, frame rate capture, resolution, and more. The auto exposure is of special interest. The exposure time control for

Figure 2.4: Images of Greenwich, taken at different exposure times [27].

how long time the camera allow light into the senor. As seen in figure 2.4 we see that a longer exposure times increase the amount of light in the image. The lighting conditions in Tromsø vary greatly, from dark winter days to bright summer days. The auto exposure could therefore help maintain a somewhat constant illumination.

The Pylon drivers and SDK also comes with a handful of executable applications such as a configuration tool for the camera IP (ZeroConf and DHCP), and a "Pylon viewer app" that allows us to capture an image or show the live feed from the detected cameras. However, for capturing movies or using the data in any other way, we needed to use the Pylon SDK and create our own recorder. This was done previously by [47]. The recorder takes as input parameters the URI of the camera to record with, when to start, for how long to capture video, wanted FPS, resolution and so on. The output is the frames, compressed and encoded in the H.264 encoding, in 3 second segment files. The 3 second segmented files allow our front-end application, that displays the results, to easily seek in the video.

2.3.3 Compression

With a resolution of 1294×964 pixels, and three 8bit channels, every frame requires approximately 3.74MB. However, since we are using YUV4:2:2P with chroma subsampling, every frame will then require approximately 2.5MB. Considering we have four cameras, and a frame rate of 30FPS, we will produce $2.5 \times 4 \times 30 \approxeq 300MB/s$. For a game of 90 minutes, which is the length of a standard soccer game, it would require 1620GB. In addition we would like to save the panorama image, which is roughly the same size as the 4 cameras combined (a bit less, due to cropping and overlapping regions). To require more than 3TB of disk space for one single match, we need an elaborate storage system for anything more than storing a few games, increasing both hardware costs and ease of use. Even if we downsample to YUV4:2:0 instead of YUV4:2:2, we loose even more chroma resolution, but reduce the data size by $\frac{1}{2}$, requiring

1.2TB for the four individual cameras, and a bit less for the panorama. It is still too much, so we employ both image and video compression when saving the results to disk. To reduce the bandwidth and disk usage, we encode the data as H.264, using the x264 library [11]. In short, the H.264 video compression codec takes advantage of the many types of redundancy in a video. Every frame contains too much chroma for the eye to see as mentioned (psychovisual redundancy), and this can be subsampled. Pixels are often very similar to their neighbors, so by using the difference from the last pixel instead of the value it self we get many similar values, which will in turn be run length encoded. Then, there are redundancies in between the frames themselves, as frames are often very similar when capturing at a rate of 30FPS. H.264 uses some frames stored as an independent image, called intra coded frame (i-frame). Then, there are predicted and bi-directional frames, which depend on the differences between the frames, meaning it needs data from the previous and future frames.

When watching a video, using the seek function and for a brief moment see just black gibberish, it is because we are processing p-frames and bi-directional frames. Without having the original i-frame for the sequence, the relative differences in the p-frames and bi-directional frames have no meaning. Once you hit the first i-frame, the quality looks fine again.

For a more detailed explanation of H.264, see [47]. Using H.264, storing 30 frames takes roughly 3.23MB using lossy compression, compared to raw YUV4:2:2 which require approximately 112MB.

2.3.4 The Northlight Library

The codebase for our project was based upon the Verdione project [10], a large project focusing on real-time video processing. We use many different libraries. Many of them will be covered later, but examples are x264, the Basler SDK [1], and OpenCV [7]. The Northlight library [47], contains functions for converting in between different representations that the different libraries uses, and was aimed to integrate the different libraries. We mainly use it for converting between image formats, x264 wrappers, and the Basler SDK. In particular, our recorder software makes use of the Basler camera function wrappers provided by Northlight.

An alternative to the YUV representation is the red, green and blue (RGB) representation, where every pixel is represented by the three additive primary colors. As mentioned, the cameras deliver YUV4:2:2P, where one sample of chroma represents two pixels. For this reason, and the fact that it is easier to manipulate RGB values, we convert YUV4:2:2 to RGBA at the beginning of the pipeline, right after reading the data from the camera. The reason for not going for RGBA from the camera it self is that the network bandwidth between the camera and our computer is limiting. At the moment, Northlight does not support conversion between YUV4:2:2P and RGBA, so the way it is solved now is by converting from YUV4:2:2P to YUV4:2:0P, and then from YUV4:2:0P to RGBA. The extra conversion step is fast and does not cause any problems in the current system. However it consumes some of the CPU resources, and some chroma accuracy is lost for no reason. In the future, it would be beneficial to develop or make use of a library supporting YUV4:2:2P directly to RGBA.

2.4 Image Undistortion

Because we use inexpensive and wide angle lenses, the image becomes distorted. In figure 2.5, we see how straight lines get bent in the image. What is happening is that the magnification decreases the further away from the optical axis you go. This particular distortion is called barrel distortion, because straight lines gets bent like how straight lines at a barrel do. Because

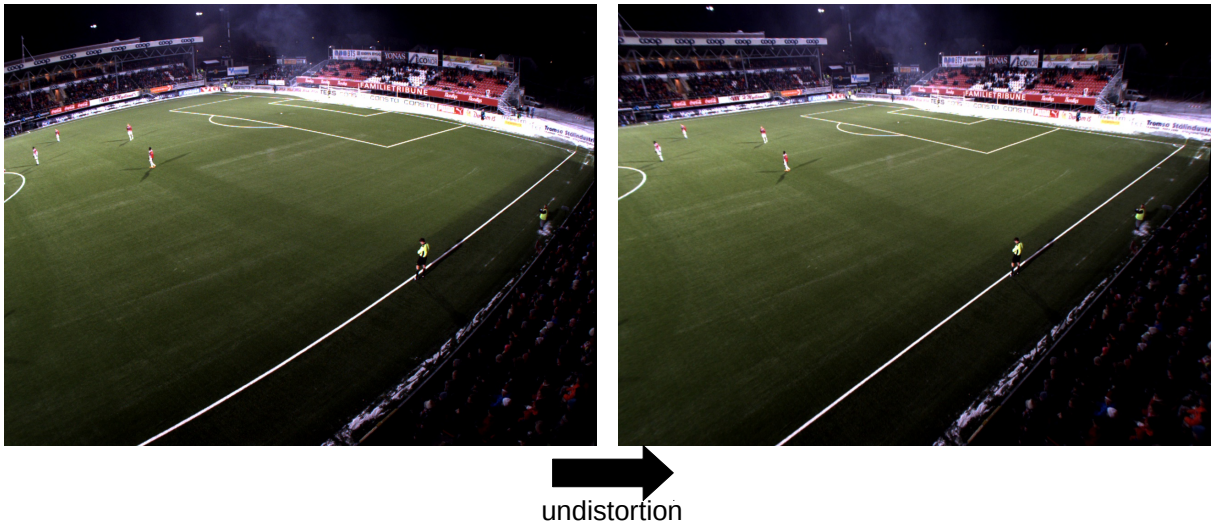


Figure 2.5: Fixing barrel distortion. Note the distorted lines in the left image

wide angle lenses use this effect to their advantage for capturing a wider view, our images will suffer heavily from this kind of distortion. However, the distortions can be corrected using functions provided by OpenCV. For a more detailed explanation, see chapter 3.

2.5 Tracking Subsystem

The Bagadus system is able to track players, using the ZXY sensor tracking system. Tracking in the context of our system means to locate where on the pitch a player is located, which cameras includes this position in their field of view, and which pixel corresponds to the player coordinate. This allows us to automatically zoom and follow a player close up with digital zoom.

ZXY is in use by professional Norwegian soccer clubs like Tromsø Idrettslag (TIL) and Rosenborg Ballklub (RBK). Each player carries a belt around the waist with a small sensor chip, which does not affect the performance of the players in any negative way. It uses wireless communication, and conforms to the IEEE 802.11 standard. Receivers are then mounted at different locations in the stadium, tracking the sensors sending the positions to a ZXY database, as illustrated in figure 2.6. Besides positional data, it also registers data such as heart rate and acceleration, but the positions are the most interesting for us.

The ZXY sensor tracking system use a coordinate system with the origin located in a pitch corner, in the case of our prototype at Alfheim stadium, the origin is located at the lower left corner as observed from the camera rig, which we will call the ZXY coordinate system. To obtain the ZXY coordinates of a player, we make queries to a database system storing the data being sent from the sensors on the players.

The prototype uses an early version of the ZXY sensor system [12], which delivers tracking information at a rate of 20Hz. This is less than our video sampling, which means we sometimes reuse the last received ZXY data. However, judging by the visual quality, this problem is barely noticeable, if at all. However, according to the ZXY web page, they are now able to deliver a resolution of 40Hz, so in any case this is no longer a problem, if the system were to be installed at another stadium, we could choose to install the ZXY system capable of 40Hz instead.

Figure 2.6: ZXY Sport tracking technology [12].

2.5.1 Data and Queries

The data is stored in a relational database, optimized for high import and export.This database is separate from the Bagadus video system, but is integrated by allowing us to login to it and do normal SQL queries for the data we want. ZXY use the Sybase SQL Anywhere as their database management system(DBMS), which offers a C interface, which we use for minimum overhead. The database contains a table mapping the player names of each team to the ID of a belt. This table must be manually maintained. To get the positions of a belt/player within a time period, one can use a SQL query such as this:

Listing 2.1: Get all positions of player with id $tagId$ and in between the time $intervalStart$ and $intervalEnd$

```
SELECT timestamp, x_pos, y_pos
FROM zxy_oversample
WHERE tag_id= tagId
AND DATEFORMAT(timestamp, 'yyyy-mm-dd hh:mm:ss.sss') >=
    DATEFORMAT('" + intervalStart + "', 'yyyy-mm-dd hh:mm:ss.
    sss')
```

```
AND DATEFORMAT(timestamp, 'yyyy-mm-dd hh:mm:ss.sss') <=
    DATEFORMAT('" + intervalEnd + "', 'yyyy-mm-dd hh:mm:ss.sss
    ')
ORDER BY timestamp;
```

Other interesting queries could be to look for all cases where the x_pos and y_pos of any of the sensor belts are within the 18-yard box of the pitch, See figure 2.7. By such a query we could fully automate interesting events happening. Other examples are corners, time periods when the players are clustered or very far apart. But for the system overall, the most interesting data is the positions them selves, as they can be used to track the players.

2.5.2 Coordinate System

The x_pos says where the player is along the x-axis, which is defined to go along the line of the longest of the pitch dimensions. The y_pos is then the axis that goes parallel to the shortest of the pitch dimensions. One of the corners are chosen as the (0,0) origin. The ZXY coordinates are in other words a Cartesian coordinate system of the pitch, seen from a birds eye perspective, given in meters. If the pitch is 105m wide, and 68m long, then the center would be at ZXY coordinate x_pos=52.5 and y_pos=34. Since the ZXY uses the actual sizes and metrics

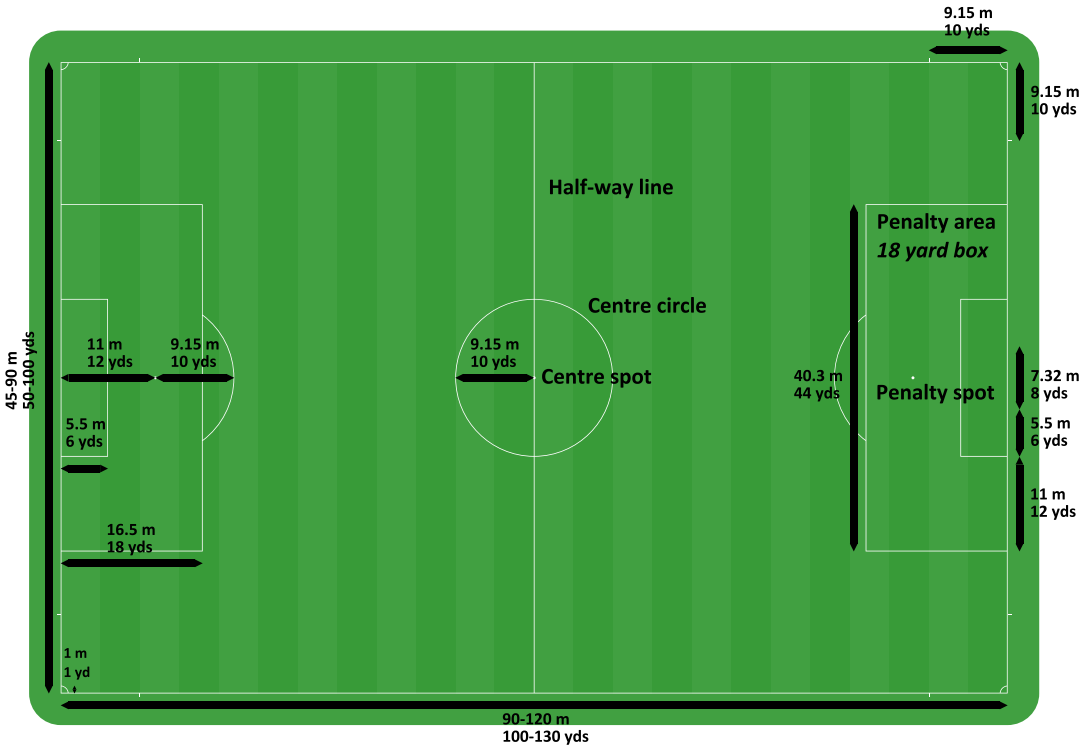


Figure 2.7: The meterics of a soccer pitch [23].

of the pitch as its coordinate system, we can easily assign certain points in the pitch, to ZXY coordinates, such as the corners, and points where the white lines intersect. For instance, the lower left corner as seen from the cameras in Alfheim stadium is the origin (0,0).

2.5.3 Synchronization

As previously mentioned, NTP was not an option for synchronization in the beginning of the project. Our initial way to solve it was manually synchronizing timestamps from ZXY to our own video frame timestamps, typically by finding the time difference between the ZXY timestamp for the start of the match, and when we see it starting in the video. This is a cumbersome, error prone and manual process, which we fixed later on by coordinating with ZXY to use the same NTP server, allowing us to assume our own video frame timestamps equals the timestamps from ZXY, given sufficient synchronization on a local area network (LAN).

2.5.4 ZXY to Camera Coordinates

The biggest challenge we had in tracking was mapping the ZXY coordinate system to the camera coordinates. Each camera will need its own individual mapping. If we were to move the cameras, they would need a recalibration. Luckily, once the system is installed, the cameras can stay stationary, since our current setup covers all of the pitch. The ZXY coordinate system can be viewed as a plane, by applying a projection onto a cardinal axis. [30], as seen in figure 2.8. A projection is simply a dimension reducing operation, like we are projecting a 3D space onto a 2D space. In our case, we do this simply by omitting the Z dimension. For instance, the ZXY coordinate (20,50,0) becomes just (20,50). This is fine, as we are assuming the pitch to be a plane, with all points of the pitch located at Z=0.

Figure 2.8: Projection of ZXY 3D onto a 2D plane

The camera coordinate system also be seen as a plane, with the number of pixels being the metrics, and x and y being the two dimensions in the captured image, as seen in figure 2.9.

Thus, to track a player by finding roughly which pixel in the captured image that corresponds to a ZXY player location, we define a function $transformZXY(x_z, y_z)$, such that if (x_z, y_z) is a location in the ZXY plane, and (x_c, y_c) is the corresponding point in camera space, then $transformZXY((x_z, y_z)) = (x_c, y_c)$, this is illustrated in figure 2.10.

To find the *transformZXY* function, we use the open source computer vision library OpenCV [7]. OpenCV contains a functions aimed at solving computer vision tasks, including mathematical calculations. The function maps a plane to another plane. OpenCV has functions to calculate a homography [33], which is a transformation that maps straight lines to straight lines (see figure 2.11), described by a 3x3 floating point matrix. The homography is the concept of transforming the image, while the 3×3 matrix represents the transformation in a compact way. In this thesis

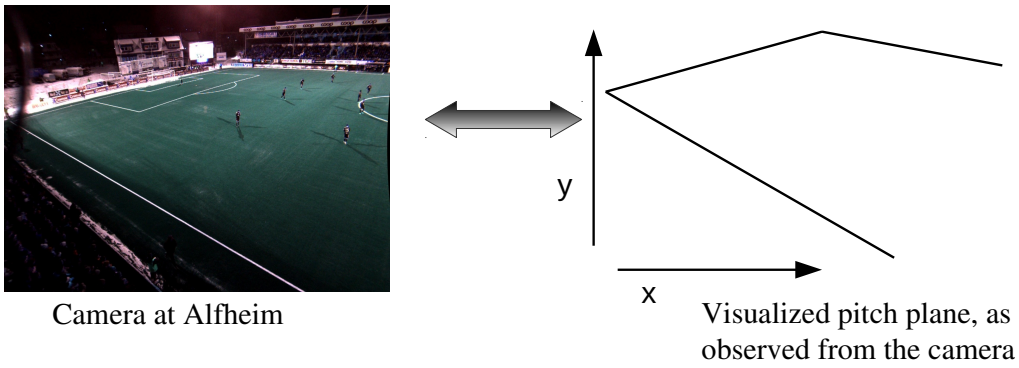


Figure 2.9: Visualization of soccer pitch plane seen from cameras

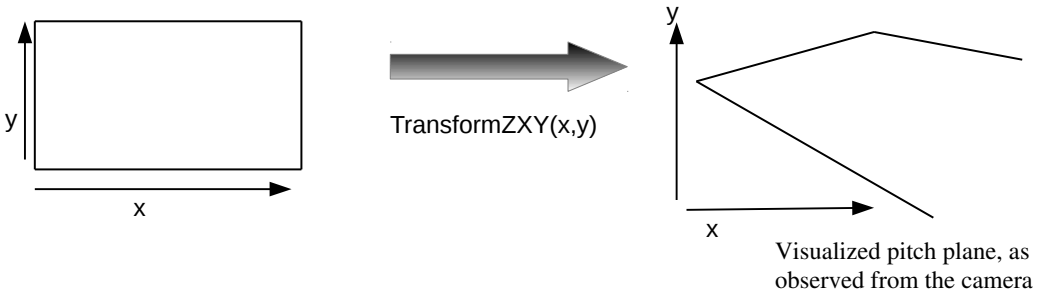


Figure 2.10: Transforming from ZXY plane to camera plane

we will from now on just refer to the matrix it self as a homography matrix or just a homography. It is also called perspective transform, because it changes the perspective of the image. In our case, we want to change the perspective from a top-down birds eye perspective, to the perspective of the camera. Equation 2.1 shows the components in a homography.

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \tag{2.1}$$

The homography is defined such that if the matrix representing it is H, and a desired transformation function is h, then$h(x) = Hx$.

If we assume for a while that we have already computed the 3x3 homography H, then a pixel/point in the original image can be mapped to a pixel/point in the target image as seen with the new perspective, given the following function [7]:

$$dst(x, y) = src\left(\frac{H_{11}x + H_{12}y + H_{13}}{H_{31}x + H_{32}y + H_{33}}, \frac{H_{21}x + H_{22}y + H_{23}}{H_{31}x + H_{32}y + H_{33}}\right) \tag{2.2}$$

This is in fact a matrix multiplication, since we are working with homogeneous coordinates [33]. If we define a 2D point (x,y) to be equal to a point (x,y,1), and further, any 2D point (x,y) is equal to a point (kx, ky, k), then we can translate from $(kx, ky, k) \rightarrow (x, y, 1)$ by dividing by k: $(kx, ky, k) \rightarrow (x/k, y/k, k/k)$. All our points are originally defined to be on the form $(x, y, 1)$. But after the matrix multiplication, the third matrix component is $(H_{31}x + H_{32}y + H33 * 1)$ which is why we divide by this sum. As such, a homography is a 3x3 matrix such that given a

collection of points x_p from the original image, and a set of corresponding points x'_p from the other image, a 3×3 matrix H will be defined such that $x_p H = x'_p$. This has several uses. One is that we can multiply 2D points with 3x3 matrices, allowing translation, skewing and rotation. Another use, is that we can represent points at infinity, by $(x, y, 0)$.

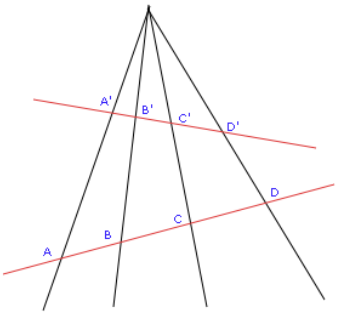


Figure 2.11: Points A,B,C,D are mapped to A', B', C', D' by a homography [53].

To change the perspective of the entire image, we need to do this to every pixel in the original image. If every pixel in the source image is mapped to the destination image using this function, we say that we have *warped* the image. Note that when warping images, we also need to do interpolation, which will be explained in section 2.6. In this specific case, we only need to transform the ZXY coordinate to the camera coordinate, by applying equation 2.2. Our function $transformZXY$ would thus take the x,y of the ZXY coordinates as input, and apply equation 2.2, using a calculated homography H. The output will be the corresponding point/pixel in the image taken by the camera. We can then for example draw tracking boxes around the players in the video by applying the transform function on the ZXY-point for the players we want to track, for every frame, and painting a rectangle around the transformed points. Because the cameras do not move, we only need to calculate the homography H once for each camera. This can be done as a calibration step, offline before the game, with no real-time constraints. However, since every camera shows the pitch from a different angle and position, we need a different homography for each individual camera. The average time to perform these calculations on our machines are less than 0.5ms, far below our real-time constraint.

2.5.5 Finding the Homographies

To find the homographies, we use the OpenCV function *findHomography*. As mentioned earlier, this step can be done offline (not during a match) as an initialization step, and it only needs to be done once, unless the camera is moved. It takes an array of selected 2D points in the original plane, and a list of the corresponding 2D points in the target plane. From these points it calculates the homography that will transform the original plane points, to the target plane points. The (3,3) component of the homography is a scale parameter, much like the last component of the homogeneous coordinates, thus the homography has eight numbers that can vary that we need to find, another way to say this is that it has eight degrees of freedom. Every point has two constraints (x and y), so that four points is enough to estimate the homography. For the interested reader, the details on how the algorithm works is described by OpenCVs documentation and source code [7] and [33]. The original plane will in our case be the ZXY plane, and the target plane is the plane as seen from the cameras.

Since we know the locations of all the intersections of the white lines in the pitch, based on standard soccer metrics (see figure 2.7), and that Alfheim is 105m wide and 68m long, we can calculate all the known ZXY / real-world birds eye intersections of the white lines in the pitch. To find the corresponding points in an image captured by a camera, we zoomed in at maximum with an image editor like Gimp, to find the corresponding pixel coordinate of the intersection. Figure 2.12 shows an example of corresponding points.

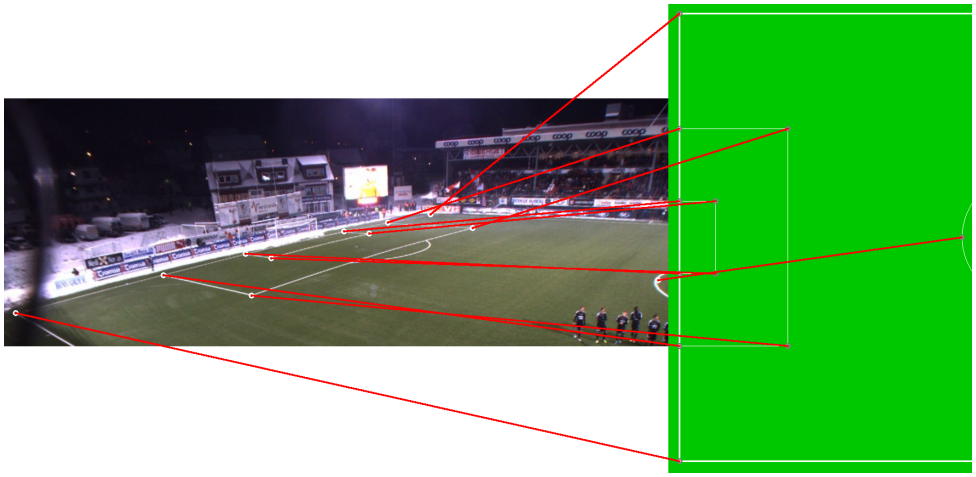


Figure 2.12: ZXY points mapped to image points [47].

2.5.6 Finding the Correct Camera

When we track a player, we would like our system to figure out in which camera the player is visible, and automatically show footage from this camera. In addition, if the player moves out of the cameras field of view and into another, we would like the system to automatically switch to the new camera. To find out in which cameras a given player is visible, we transform the ZXY coordinate of the player with our function and the homography H of that camera. We then observe the transformed point is outside of the image boundaries, meaning x must be within [0,imageWidth] and y must be within [0, imageHeight]. If either the x or y value is outside ranges, it means the player is not visible in the given camera.

2.6 Panorama Creation

To create a panoramic view of the entire pitch, we need to *stitch* the photographic images from each of the four cameras together to a single panoramic view. To stitch the images means the process of combining multiple images into a segmented panorama or high resolution image. Since we are working with video, we have to do this for every set of matching frames from the four cameras, meaning we have less than 33.33ms to do the stitching, in order to do it in real-time. Stitching involves “image registration” and “compositing” [7]. “Image registration” is to locate overlapping regions between the images, and find common feature points within these areas. “Composition” is to warp the images as described in section 2.5, based on the common features points found, find the seam masks and blend the seams. However, note that the warping of the images can be done with more than a homography.

(a) Planar stitching [47]

(b) Spherical stitching [47]

(c) Cylindrical stitching [47]

Figure 2.13: Different stitching methods

Figure 2.13a shows an example of a panoramic view of Alfheim stadium, created using OpenCVs Automatic stitcher functionality, using planar projection. Figure 2.13b and figure 2.13c shows spherical and cylindrical projections respectively. These are created using map projections, where the image plane is wrapped around a real-world 3D object like a cylinder or sphere. While these methods yield quite visually appealing results, they perform far from our real-time constraint of 33.33ms, as seen in table 2.1. One reason for this is that the OpenCV stitcher performs functionality such as blending pixels of the overlapping image together, finding seam masks, color correction of the images and more.

Homography stitching. Thus we chose a more manual stitching method, based essentially upon the same idea as the mapping from ZXY to camera space, as in section 2.5, only now we are warping images to fit a common image plane, simply selected to be one of our camera perspectives. In figure 2.14, we see how we stitch the images together by warping them to fit the second camera from the left, by first warping them and then padding the images until they

Algorithm	Image resolution	Execution time (ms per image)
OpenCV Planar projection	12040x3051	14103
OpenCV Cylindrical projection	9275x1803	4900
OpenCV Spherical projection	2371x1803	1746
Homography stitch	7000x960	974

Table 2.1: Stitching characteristics [32].

overlap. We do not use blending of the overlapping pixels, instead we do a simple static cut bottom to top, at a given width of the image. Like with the ZXY→camera homography, we only need to find it once, unless the camera is moved out of position, same goes for finding where to cut the images (deciding when to stop using the leftmost image, and when to start displaying the rightmost image in the area of overlap). Thus, the operations that have to be executed in real-time are only the warping and a series of memcopy operations. In table 2.1, you can see that our homography stitch is executed in $974ms$, which shows potential to be real-time with further optimizations.

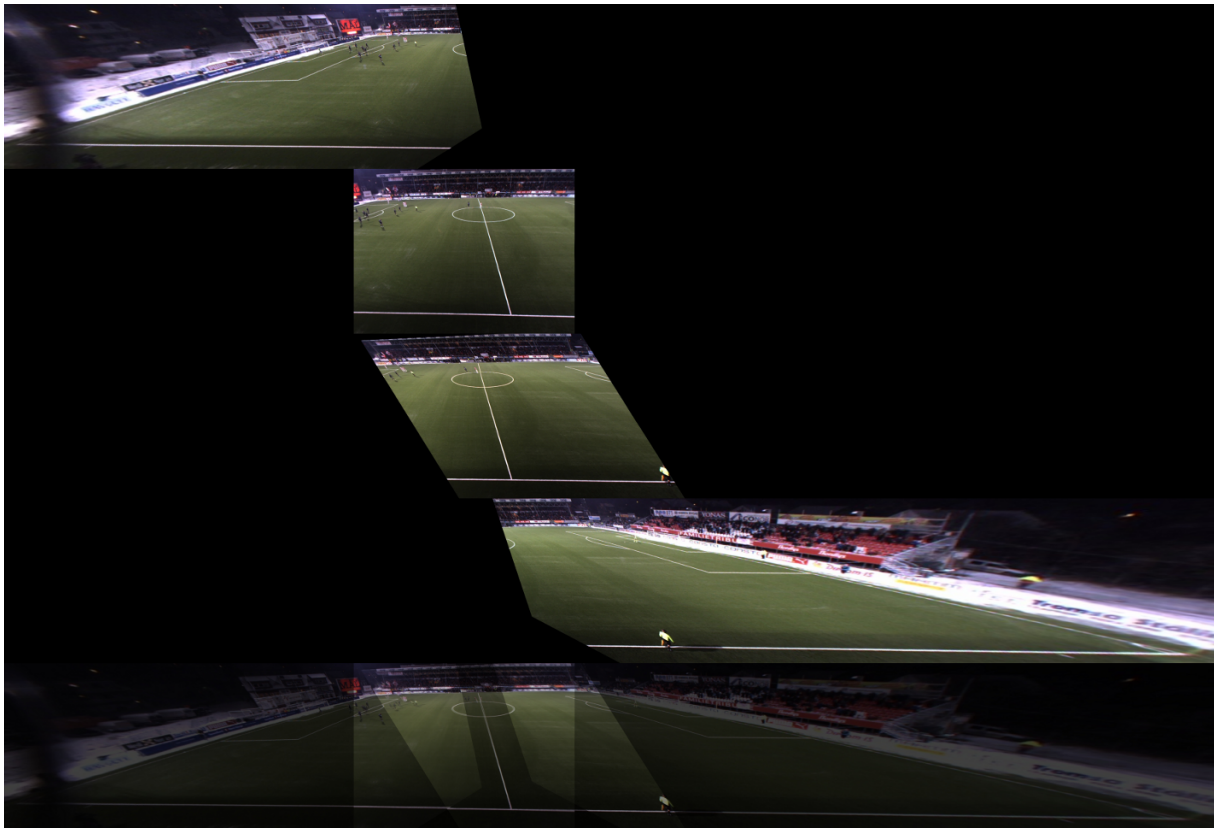


Figure 2.14: Homography stitching [47].

To find the common points for the homographies, we use the exact same method as in section 2.5, opening the images in an image editor like Gimp, zooming in and manually locating corresponding feature points, and using them to create the homography with OpenCV. As for the seam, it is found by aligning the images in an image editor and testing where to cut the

images along the x-axis by trial and error. In chapter 3, we will present a more automatic approach.

2.7 A Front-End Demo

A demo application that integrates the features discussed has been implemented, and is featured in [47]. It plays the video footage captured at a frame rate of 30FPS, both footage from the single cameras and a stitched panorama video, allowing the user to manually select which camera to show footage from, or to show the panorama video.

It supports tracking of players, drawing a white box around each player. Everything is done real-time in the demo, except the panorama video stitching, which was done offline and stored for the demo purposes. However, in the current system, panorama video stitching has been made a real-time process.

You can zoom using the current tracked player as the center of focus, or track multiple players at the same time. It also features automatic switching if the player runs outside of the current camera, as well as automatically forwarding or rewinding to the time of events tagged by the coach, tracking the players involved in the event.

A screenshot of the demo can be seen in figure 2.15, and a youtube video of the demo can be found at [1].

Figure 2.15: An interactive demo, integrating ZXY, Muithu and our video system [47].

[1] http://www.youtube.com/watch?v=1zsgvjQkL1E

2.8 The new Bagadus Video Pipeline

While the Bagadus prototype had functioning components, they did not run within our real-time constraint, when applied in sequence, on a single CPU core. [34, 43, 49] have been working on pipelining the individual steps, so that they can run in parallel. In addition, not all of the subsystems, such as the panorama creation, were fast enough, even when timed individually. To offload the CPU, and increase the performance, the panorama warping and stitching were implemented to run on the GPU.

Figure 2.16, describes the new Bagadus video pipeline, with the modules listed along the columns, and the rows representing different frames(and thus also time) , with a new frame arriving every 33.33ms. Every 33.33ms, the frame in every module is passed on to the next module, where every module runs in parallel on different cores, utilizing both the GPU and CPU. When adding all the sequential modules, we get far more than our real-time constraint of 33.33ms, but because we are pipelining, and running the modules in parallel, the system stays real-time. After 11 frames have been processed, the all the modules are processing a frame, and a new frame is written to disk every 33.33ms, by the *panoramaWriter* module.

Input frame number (time)	Cam-Reader	Converter	Debarreler	SingleCam-Writer + Uploader	BGS	Warper	Color-Corrector	Stitcher	Yuv-Converter	Downloader	Panorama-Writer
n	n										
n + 1	n + 1	n									
n + 2	n + 2	n + 1	n								
n + 3	n + 3	n + 2	n + 1	n							
n + 4	n + 4	n + 3	n + 2	n + 1	n						
n + 5	n + 5	n + 4	n + 3	n + 2	n + 1	n					
n + 6	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n				
n + 7	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n			
n + 8	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n		
n + 9	n + 9	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n	
n + 10	n + 10	n + 9	n + 8	n + 7	n + 6	n + 5	n + 4	n + 3	n + 2	n + 1	n

Figure 2.16: The steps needed for storing video are pipelined. The blue CPU bar is the old prototype, and the green GPU bar is the new pipeline

2.8.1 The Components

The components of the video pipeline are mostly the ones we have discussed earlier in this chapter, with a few additions and modifications. The following is a short list describing the modules of the list:

Cam-reader (CPU)
Accepts synchronized frames from the cameras, and stores them in a small input-buffer

Converter (CPU)
Converts the images to RGB, as explained in section 2.3.4.

Debarreler (CPU)
Removes the geometrical distortion from the lens, as explained in section 2.4.

SingleCam Writer + Uploader (CPU)
Writes the frames from each individual camera to disk, encoded using H.264. It also uploads each frame to the GPU for further processing.

BGS (GPU)
This module performs background subtraction, using differences in between video frames to separate the objects in the scene, into background and foreground. The foreground represents moving objects, typical objects of interests such as the players. The foreground is separated from the background with a mask. For more information, [49], and our usage of the BGS in chapter 5.

Warper (GPU)
Warps the images onto a common plane, so that they are aligned for panorama stitching. This part was optimized by [34, 43], which implemented a stitcher integrated in the Bagadus system, running real-time utilizing the Graphics processing unit (GPU), through the NVIDIA Performance Primitives library, as seen in figure 2.17.

Color corrector (GPU)
[43] implemented a color correction application, looking at the differences of intensity between two overlapping areas of image pairs, and adjusting the global image intensities so that the images have a similar intensity. This makes the end result of stitching and free-views more visually appealing, and the stereo correspondence algorithms look for corresponding intensities.

Stitcher (GPU) The old prototype used static cuts in the images, sometimes cutting through players, which results in distortions since the warping is not perfect. [34] implemented a dynamic stitcher, searching for a path through the image which does not hit a player, using both color information, and the foreground masks from the BGS, to find a more optimal seam.

Yuv-converter (GPU) [43] implemented an RGB-to-YUV converter on the GPU, because of the large size of the panorama images.

Downloader (CPU) Downloads the processed video frame from the GPU back to system memory.

PanoramaWriter (CPU) Writes the images back to disk, using encoded with H.264.

2.8.2 The Interface

In the earlier prototype, the components of the video system were not integrated. To record a match, we had to start a recorder for every camera, which saved the raw footage to disk. The footage would later be processed manually by an application for removing the barrel distortion, and then through another application, which warped and stitched the frames into a panorama video. This requires much input from the user, and assumes the user to posses quite technical skills.

To increase the process more user friendly, the new video pipeline offers a web-based user interface, for starting and stopping recordings, pictured in figure 2.18. Using the web interface,

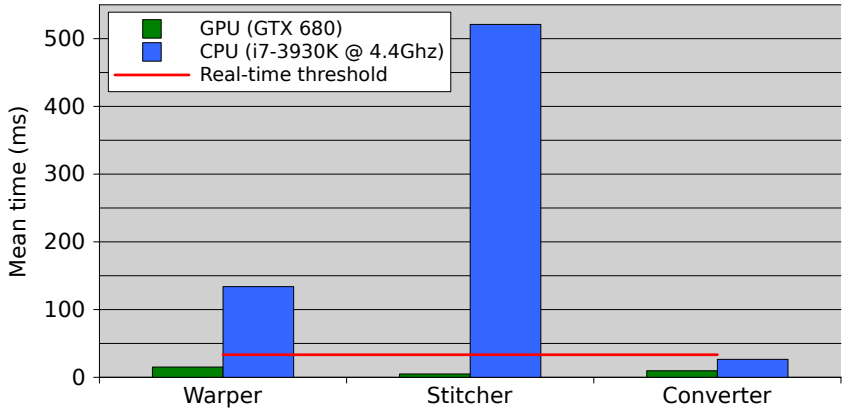


Figure 2.17: The new pipeline: The results from the new stitcher show a clear improvement, compared to the old CPU based pipeline

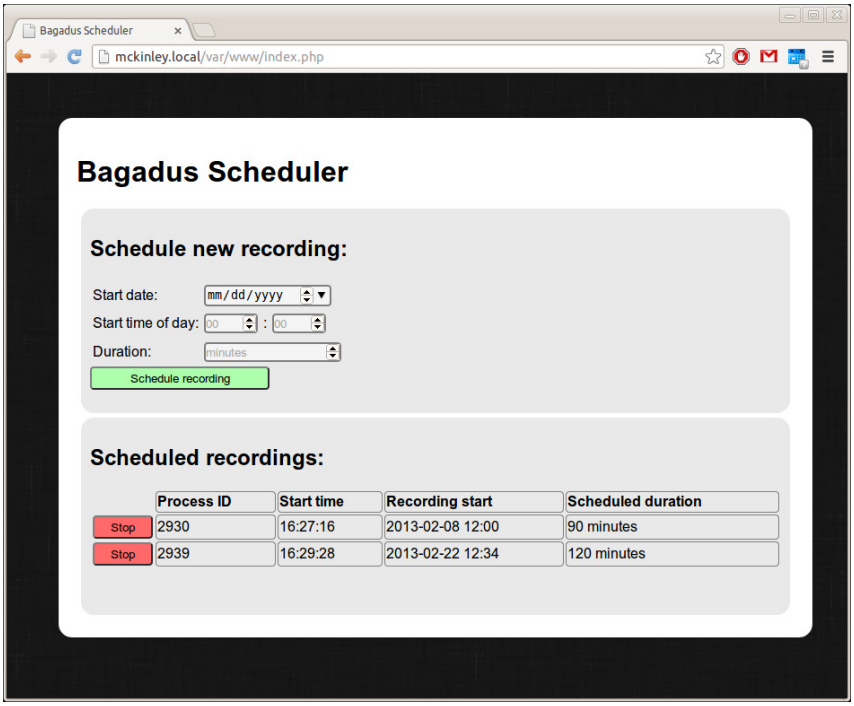


Figure 2.18: A web-based user interface for starting and stopping recordings

the user can schedule recordings to start at a chosen time, or recordings can be canceled. Since the pipeline now integrates all the modules, the user only need to interact with the web interface, and the pipeline takes care of the rest, all within real-time.

2.9 Future Development: Free-View

Although it is out of scope for this thesis, the goal for the Bagadus system is to at some point in the future be able to offer a *free-view* or *virtual camera* functionality [44] [28]. We will use the terms free-view and virtual camera interchangeably. A videos viewpoint is usually fixed, usually at the position and heading of the single camera capturing the footage. In free-view, we

use several cameras to capture the scene from different positions and angles, allowing the user to change the position and orientation of the viewpoint. Further, the viewpoint is not restricted to one of the physical cameras in the scene. The goal of the free-view is to allow the scene to be seen a arbitrary point and angle in 3D-space, not necessarily a physical camera, but somewhere between two physical cameras. Thus, we will recreate the scene as seen from a *virtual camera* using information from the real cameras. Figure 2.19 shows four cameras and an example

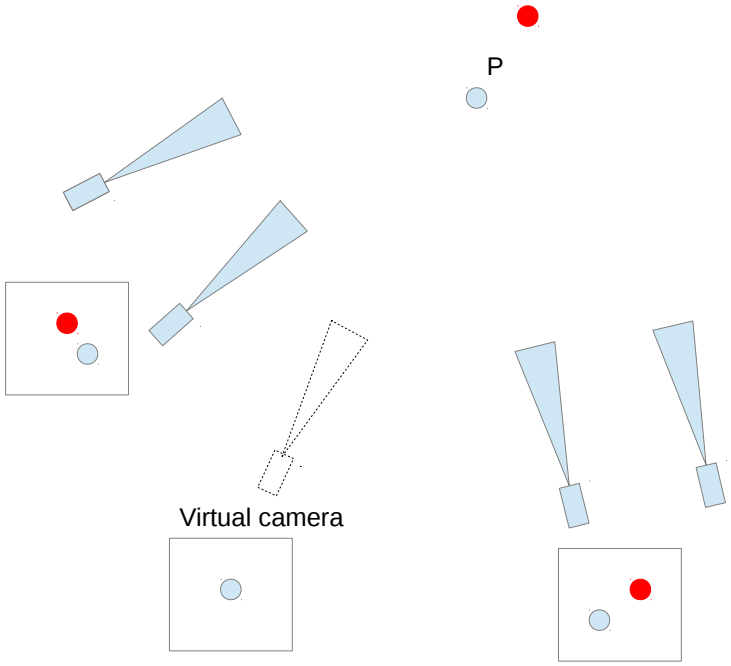


Figure 2.19: A free-view demonstrated by a virtual camera

virtual camera view, as well as the results of two of the cameras capturing the scene, in addition to how the scene will be rendered, as seen by the virtual camera. Two of the real cameras can see both of the red and blue objects, while the image computed by the virtual camera shows only the blue, because the red object is hidden behind the blue one from the view of the virtual camera.

The reconstruction of the scene can be done by transferring the depth-maps of the nearby real cameras to the free-view camera, and then backward mapping the pixels from the virtual camera to the real ones, by the depth of those pixels. Further operations needed for a high quality free-view are testing the depth-value of the pixels mapped to the virtual camera and only show the lowest depth values so that the nearest ones are shown.

Also, some post-processing of the result will be required, since the virtual camera relies on data from other cameras in other positions, small "holes" in the view might appear due to the fact that the virtual camera might include small areas which is not visible for the two real cameras. This step is known as hole-filling. The objects being projected using the depth maps might also look artificial along their edges, which can be remedied by for instance blurring the edges and the background.

The generation of depth maps is a very computationally expensive operation, and estimating such a map for a 1294×964 pixel image within our real-time constraint is not trivial to do with today's hardware. For our definition of real-time, see section 2.3.

When dealing with free-view in soccer stadiums, we can make the task of depth estimation and the synthesis of the free-view image more feasible by taking into advantage that apart from the players, the scene has very few depth planes. The main focus is on the pitch it self, which we can assume is a flat plane. Thus, we can create a 3D-model of the stadium, using a flat plane with a texture model on top of it to illustrate the grass, as suggested by Papadakis et al [44]. The background and sides of the stadium are not important for the game, but for aesthetic reasons, we can approximate them by considering them to be flat walls, wrapping a texture on them, captured by the real cameras. Thus, only computing the depth of the players remains. Our focus for the depth estimation in this thesis, is thus obtaining a depth map of the players, in real-time.

2.9.1 Motivation

With a free-view function, it is possible to radically change the way spectators watch soccer games. Instead of having a fixed viewpoint, the spectators them selves could control a virtual camera, offering a much more immersive experience. A virtual camera could be placed in locations where a physical camera could not have been, like in between the goal posts, or it could be following a specific player, or even the ball. It can also be used by the coach, to get a better view of a given situation, by seamlessly rotating around the scene, in either a paused or slow motion state.

2.9.2 A Depth Map Pipeline

There are different methods for estimating the depth. With some assumptions, it can be done from a single image (monocular video) [38]. This method assumes the scene to be a plane, and estimate the depth by comparing the height of the pixel, and the real-world height of the camera. This method is briefly covered in section 4.3.

The depth can be also be estimated using two cameras observing the same scene from slightly different positions. This technique is called stereo matching, and is the one we will use, as it gives a more accurate depth map by combining information from two cameras instead of just one. The basic idea is to observe the same objects of interest from two different viewpoints, then compare the difference of the pixels positions in the two images, to determine the depth. For a detailed explanation of stereo matching, see chapter 4.

Figure 2.20 shows our pipeline for depth map estimation. Note that the first step, the camera calibration is done offline, meaning it does not have to be real-time, and we only have to do it once for each camera setup. If we change lenses or move any of the cameras in relation to each other, then a new calibration will have to be done. Since we are mounting the cameras at a far distance with wide angle lenses, we assume that moving the cameras will not be done often, especially not when the system is running. During the camera calibration step we calculate important camera characteristics and create maps and matrices, used for removing lens distortion and warping the images to a common plane as needed by the stereo correspondence algorithms.

The online processes have to process frames within our real-time constraint at 33.33ms. We will only measure their performance individually, as they can be pipelined in the same way as the Bagadus prototype. In our implementation the rectify component of the pipeline is a simple call to the OpenCV function $remap$ and symbolize that we apply the calibration parameters estimated in the offline step. The color correction component should ideally be included, but

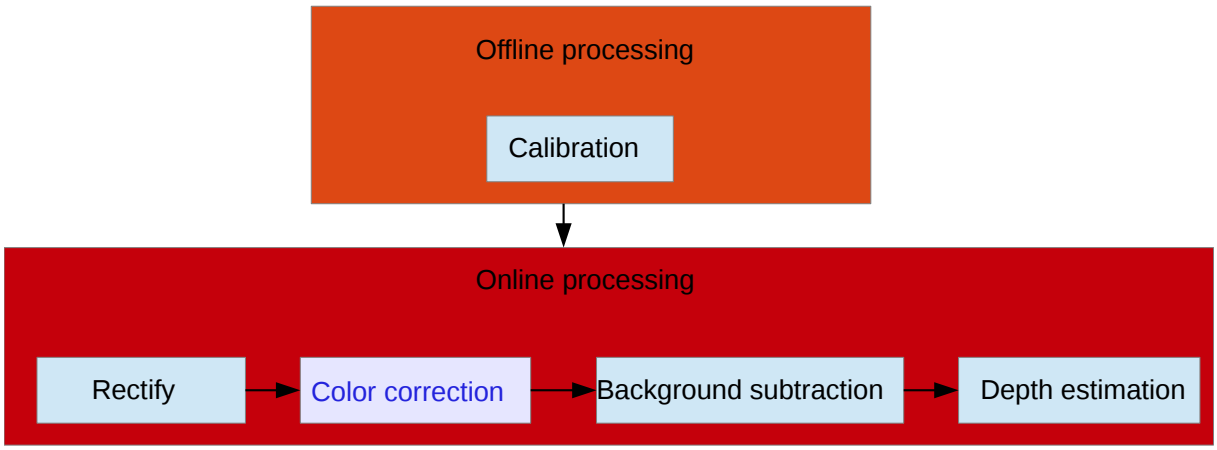


Figure 2.20: A pipeline for real-time depth map estimation

in our implementation it is not. [43] implemented a color correction application, looking at the differences of intensity between two overlapping areas of image pairs, and adjusting the global image intensities so that the images have a similar intensity. This makes the end result of stitching and free-views more visually appealing, and the stereo correspondence algorithms look for corresponding intensities. Due to the late completion of the color corrector, it was not integrated into the pipeline, however it is shown to run real-time, and can be directly applied to our images.

The background subtraction is implemented by [49], and separates the moving parts from the background of the movie. In our case, the moving parts will be the players. Since the stereo correspondence algorithms are computationally intensive and shown not to run real-time for large images [44] such as ours, we will use the background subtracter to only calculate the depth for the areas around the players, and use a simpler depth model for the pitch.

The depth estimation component will run a stereo correspondence algorithm to estimate the depth of the pixels. It is important to note that the image pairs from the stereo cameras are synchronized at the camera shutter level, so that they are captured at the exact same time. This is accomplished by the trigger box, as described earlier in section 2.3.

2.9.3 Related works

Kjetil Endal [28] created a pipeline for a free-view application, however he concluded that he probably needed a better rectification and some kind of hole-filling in the depth-maps as a post-processing step, to get good quality depth maps. He tested various stereo-vision algorithms for depth computation on a system running an Intel Core i5-450M 2.4 GHz dual core processor. However, his data set consisted of 320×240 pixels images, far less than our dimensions. The algorithms he tested were Block Matching (BM) and Semi-Global Block Matching (SGBM). The BM performed at an average of 45FPS, and SGBM at an average of 13FPS, and he concluded that BM included too much noise and had several holes. SGBM gave much more accurate results, but did not run real-time. We will test these algorithms for our data set, and see if optimizations and post processing can make it run real-time and with acceptable quality.

The Camargus system has functionality for a free-view, but they use 16 cameras mounted close to each other on a rig, and it is uncertain if their system generate depth maps at all. A long

term goal for Bagadus is to present a cheaper more flexible free-view solution using generation of depth maps.

Papadakis et al. [44] presented a system capable of calculating depth maps, and creating high quality synthesis images from a virtual camera. It is also coined at soccer games, and they use a similar image resolution as we are: 1920×1080 pixels. Their solution is far from running real-time though; one single frame took between 5 to 10 minutes of processing on a single CPU core, running on a system with an Intel Core2 Quad CPU and a NVIDIA GTX280 graphics card. We need to spend no more than 33.33ms for each frame, to satisfy our definition of real-time. They state that half of the intern processes are real-time: separating the players and the background, synthesis, and filtering. The components that did not run in less than 33 ms real-time were the depth-estimation and in-painting. The in-painting / hole-filling could be optimized by preparing a search-tree of texture to use, instead of building a new tree every frame. The majority of the time however, was spent on computing the depth maps. They were segmenting the players, limiting the depth map calculations to the players, but they were using a computationally expensive algorithm (graph cuts) for computing the depth maps. We will try to employ a less computationally intensive algorithm and see if we can still get good quality depth maps, within our real-time constraint.

2.10 Summary

We have seen how the Bagadus system records video using four cameras covering the entire pitch, and how it stitches the frames from each individual camera into a panorama video. We have seen how to track players using the ZXY sensor system and how the coach can annotate events and save the data into a database through the Muithu system, and how this can be integrated into a playback application instantly showing the desired events with the click of a button.

But as we have seen, the Bagadus system is depending on much manual work and hardcoded parameters for initialization of the system, especially in regards to the ZXY and panoramic warping homographies. Further, the system lacks a free-view functionality, which depends on having video frames with their corresponding depth maps. We have looked at some existing systems and previous work for depth map generation, such as Papadakis et al. [44], Camargus [2], and [28]. Camargus needed many cameras, and could only support a limited change in viewpoint. [44] were far from real-time, requiring between 5 to 10 minutes per frame. [28] investigated less computationally expensive algorithms, but the tests were performed at low resolution images, the most promising method (SGBM), had an average of 13 FPS, and it was not tested in a large open space. Thus, we investigate the possibility of providing depth map generation in real-time, and the first step in our depth map pipeline, is the camera calibration.

Chapter 3

Camera Calibration

An important part of the depth map pipeline discussed in section 2.9, and for the current Bagadus pipeline as it exists today, is the calculation of various homographies, the position of the cameras, and the mappings used for correcting geometrical distortions. In this chapter, we will investigate methods for calculating these, as part of the *camera calibration*[1]. Several of the algorithms involved in depth estimation assume images with no optical distortions, and the cameras must have the same lens and point in the exact same heading. However, this is near impossible to achieve in practice, so we will have to perform camera calibration and correct for the misalignments, optical distortions and so on in software.

3.1 The Theory

3.1.1 The Pinhole Model and Intrinsics

There are several camera models, but the simplest of them is the pinhole model [33]. If we consider a camera without any lens to bend the light, we end up with a small "hole" where the light is allowed to pass through, and is captured on a plane further back in the device, illustrated in figure 3.1.

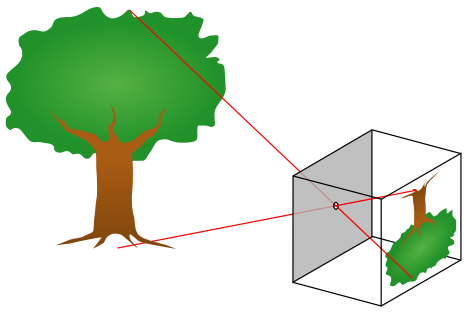


Figure 3.1: The camera pinhole model [45].

The plane at the back of the device where the image is projected, is called the image plane, or focal plane. The point letting in the light is called the camera center, and can be seen as the

[1]While the term camera calibration is usually reserved for calculating the intrinsics, we will also cover the extrinsics, and homography calculations

point where we place the aperture, which lets light in. In practice when constructing cameras, it is natural to put the image plane in the back of the device, behind the camera center. For the purpose of the theoretical model, we can move the image plane in front of center of projection, and still pretend that the same rays of light hits the plane, as if it were in the back. The effect of this is that the image is not flipped horizontally, nor vertically, anymore. If we consider **C** to be the center of projection, in figure 3.2, we see the principal point **p** is located along the perpendicular axis Z. The principal point marks where along the perpendicular axis Z, that the image plane goes, illustrated by the bold black line from **p**. The distance between **C** and **p**, is the *focal length*.

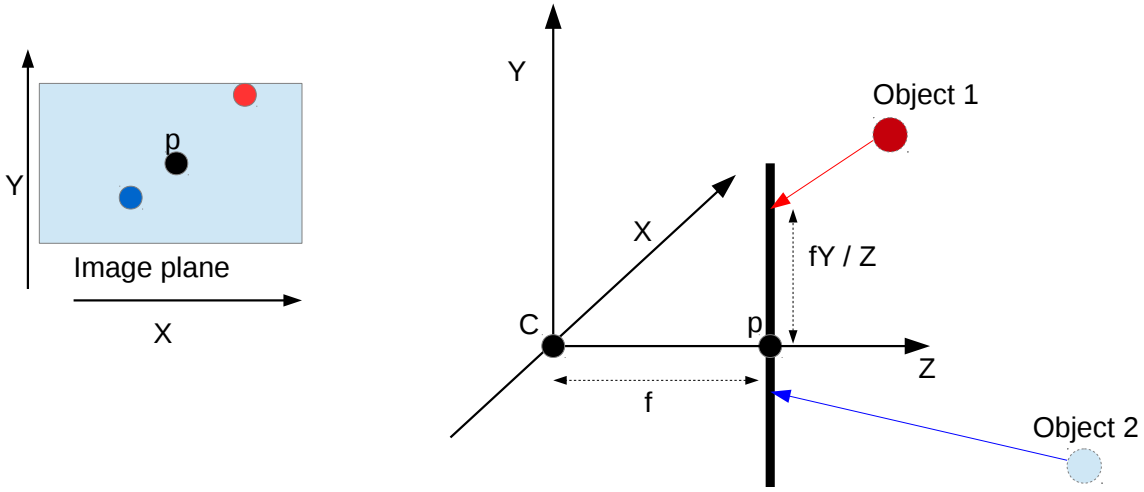


Figure 3.2: Pinhole model detailed

The focal length, will directly affect where on the image plane a ray will intersect. In general, the point (x, y, z) will be mapped to $(\frac{fx}{z}, \frac{fy}{z}, f)$. For the purpose of determining the points in the image plane, we do not need the last coordinate, as the image plane is parallel to f. However, this assumes that the origin of the image plane is the same as the principal point, which is not always true. To correct for this, we need to assume that the principal point can be moved to some other location in the image plane. If we let (p_x, p_y) be the position of the principal point in the image plane, then the point (x, y, z) will be mapped to $(\frac{fx}{z}+p_x, \frac{fy}{z}+p_y, f)$. This can thus be expressed by matrix multiplication as in equation 3.1.

$$\begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \left(\frac{fx}{z} + p_x, \frac{fy}{z} + p_y, 1 \right) \tag{3.1}$$

This matrix represent a mapping from real-world coordinates to the image plane, and is called the *Camera matrix*, representing the position of the principal point, and the focal length. This assumes that the image plane will have the same scale for each axis, but this not always the case. For instance, a camera might have less pixels per unit in height, than in width. This is corrected for, by replacing f in the camera matrix with $f * pixelsPerUnit$ in each x and y axis, denoted f_x and f_y. Thus, our final camera matrix looks like:

$$C = \begin{bmatrix} f_x & 0 & p_x & 0 \\ 0 & f_y & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We have assumed that we are using a pinhole model. If we have a small aperture, then very little light hits the image plane, and we require long exposure times. If we increase the aperture size, we can reduce the exposure time, but multiple rays of light from the same object point will hit different spots in the image plane, resulting in a blurry image. Using a lens, we can bend the light to give a more correct focus. In practice, cameras use a lens, and do not act exactly like the pinhole model. However, lenses are never exactly correct, and deviate from the "perfect" pinhole model, creating *aberrations*, especially so for wide angle and inexpensive lenses. Several types of aberrations exist, such as chromatic aberrations, but we will only cover geometrical distortions. The result of geometrical distortions, is that straight lines in the real world, is not straight in the image plane. The most commonly encountered distortions are radial distortions, because the lens bend the light in such a way, that straight lines appear to have a circular form. Figure 3.3, shows an example of a radial distortion.

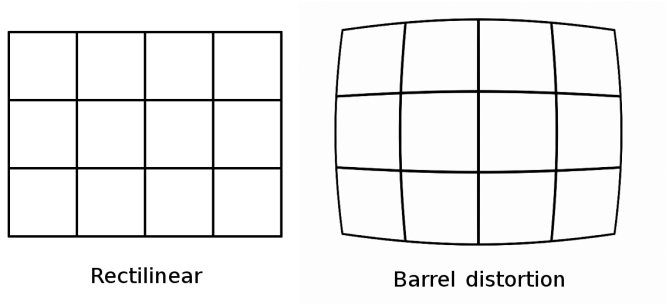


Figure 3.3: Barrel distortion [47]

There are other kinds of geometrical distortions than radial distortions, such as tangential distortion, which is a distortion which does not affect the image in a symmetric way. The radial and tangential distortions can be described as:

$$x = x^{'}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + 2p_1 x^{'} y^{'} + p_2(r^2 + 2x^{'2}) \tag{3.2}$$

$$y = y^{'}(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) + p_1(r^2 + 2y^{'2}) + 2p_2 x^{'} y^{'} \tag{3.3}$$

Where r is the distance to the distortion center, k_2, k_4 and k_6 are variables describing the radial distortion, and p_1 and p_2 describe the tangential distortion. (x, u) is the undistorted point and $(x^{'}, y^{'})$ is the distorted point.

Intrinsics

We have now described the camera matrix, and the distortion coefficients, which maps real-world coordinates to the image plane. These are called the *intrinsics* of the camera, and are usually constant for each individual camera and lens. Exceptions are cameras where you can swap the lens, or adjust the focal length. By calculating approximations for the camera matrix and distortion coefficients, we can correct the tangential, and radial distortions.

3.1.2 Extrinsics

We have looked at how intrinsics determine how the camera maps the world points to the image plane, through the pinhole model, and how they are "internal" to each individual camera. We also have the *extrinsics*, which is the rotation and translation needed to align the axes of the external, real-world coordinate space to the camera space. The translation can be expressed by a translation vector of three elements $T = (x, y, z)$, which is how much to move along each cardinal axis, and a rotation vector $R = (r_x, r_y, r_z)$, expressed as an Euler angle, specifying how much to rotate the heading of the camera along first the x axis, then the y axis, and then the z axis. For instance, if we were to move the real-world coordinate system with the points of (0,0,1), (1,1,1) and (5,5,1) to the 2D image coordinates of (3,3,1), (4,4,1) and (8,8,1), we see that the extrinsics would be $R = (-3, -3, 0)$, and $T = (0, 0, 0)$. The rotation and translation of a camera is also called the *camera pose*, or just the *pose*.

It is worth noting that the extrinsics values depend largely on what we choose as the real-world coordinate system. We can choose whatever coordinate system, and origin we want. Further, the extrinsics change every time any of the coordinate systems changes, which means if we either move the cameras or move the "world", then the extrinsics will have to be re-calculated.

One way to calculate the extrinsics, is to undistort the image using the intrinsics, so that the straight lines appear as straight in the image, then find the homography based on common points, as described in section 2.5. Since the homography translates between two planes, if one of those planes are the real-world coordinate system we are interested in, then the homography and the intrinsics, is enough to extract the extrinsics. The basic idea is to equate the columns of the manipulated matrices of extrinsics and intrinsics [16].

3.1.3 Epipolar Lines and Image Rectification

In chapter 4, we will see that depth estimation algorithms based on stereo vision, need images of the same scene, taken by two cameras at different positions. Further, the stereo correspondence algorithms desire that a feature point detected at pixel $p_1 = (x_1, y_1)$ in the first image, should be found at $p_2 = (x_2, y_1)$ in the second image, to limit the search space for matching pixels along the epipolar lines. The stereo vision algorithms are explained in detail in chapter 4. In other words, the pixels in both images can vary along the x-axis, but they must be on the same image scanline. Achieving this effect by perfect real-world camera alignment is extremely difficult, and software correction is used instead. To do this we need to make use of *epipolar geometry* [33], which is the intrinsic projective geometry between two views. What kind of objects are in the scene, and how far away the objects are, does not change the epipolar geometry, it is only dependent on the intrinsics, and extrinsics of the cameras.

Figure 3.4, shows a point X observed by two cameras, with center of projection O_L and O_R. X_L is the point where the ray from X hits the left image plane, and similarly with X_R for the right view. Notice that the point X, O_L and O_R together defines a plane, which is called the epipolar plane. The intersection of the epipolar plane and the image planes defines a line in the two image planes, which is called the *epipolar lines*. Thus, the epipolar lines change if any of our three points defining the epiplane change. Points that lie in the plane will be visible in the epipolar lines. Though, the epipolar lines do not need to be parallel to any of the cardinal axes in the image planes.

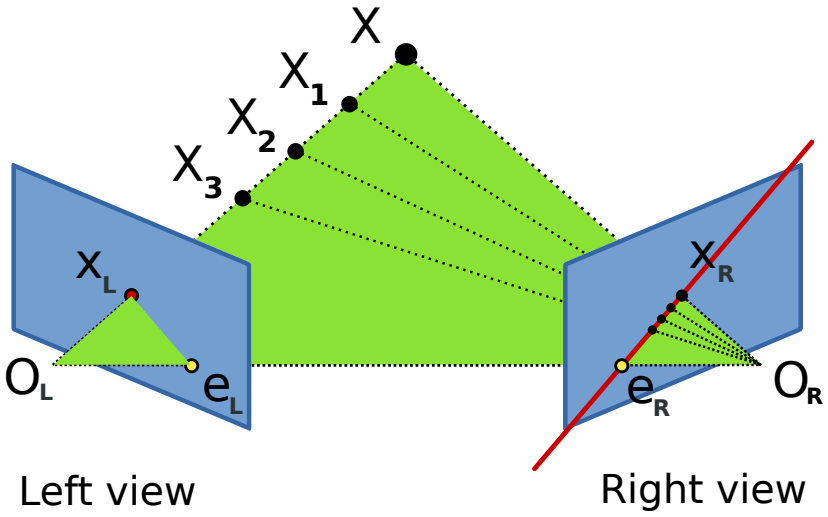


Figure 3.4: The epipolar lines e_R - x_R and e_L - x_L [42]

Image rectification

Assuming the images have already been corrected for geometrical distortions from the lens, a linear transformation can transform the images by rotating, skewing, and scaling the images so that the epipolar lines are parallel to the scanlines (horizontal axis). This is the same as transforming the two images onto a common plane. The rectification relies on the epipolar

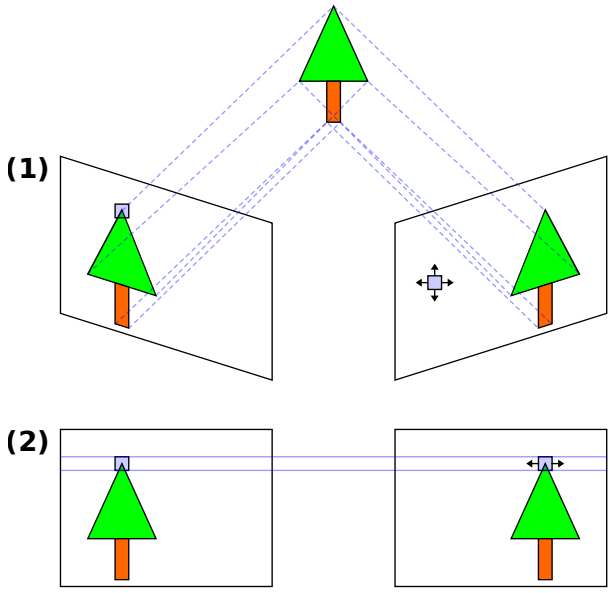


Figure 3.5: Rectification of two images observing the same object from different view points [51]

geometry between the cameras, described by the 3×3 *fundemental matrix* F. If x and x' are two corresponding homogeneous points in the image planes of two views of the same scene, then F is defined such that $x'^T Fx = 0$, where Fx describes an epipolar line, on which x

lies. The *essential matrix* E, is also a 3×3 matrix, describing the normalized coordinates of F. Thus, to rectify the images, one must estimate either F, or E. Figure 3.5, shows two cameras observing the same object from different angles, and the result of rectifying the images, so that the epipolar lines becomes horizontal.

3.2 Implementation

We will now discuss how we implemented the calibration system, and made use of the theory. We will use the OpenCV library for much of the mathematical functionality.

3.2.1 Intrinsics

To correct the distortions in the image, so that straight lines remain straight, we need to estimate the camera matrix and distortion coefficients. Finding the distortion coefficients and camera matrix, involves approximation based on sets of corresponding 3D real-world points, and 2D image points, where each point correspondence makes up for two degrees of freedom.

OpenCV provides functions for doing this easily, using a calibration pattern that provides recognizable points. In our implementation, we use a chessboard, where the black and white squares makes it easy to find corners where black squares meet white squares.

The real-world coordinates are somewhat imaginary, where we simply say that the Z coordinate are always 0, since the chessboard is a plane, and we define the origin to be the upper left of the chessboard. The first corner will then be at (1,1). Since we know that the chessboard is a plane, and that we are dealing with squares, the distance between two corners is the same along both the x, and y axis. Thus, we can increase our location in the coordinate space by a unit of 1, for every new corner.

To support for multiple kinds of chessboard patterns, the OpenCV function takes the number of corners along the x, and y axis, as parameter, and generates a two-dimensional array containing the real-world 3D points assuming Z=0. In figure 3.6, we see an example coordinate system, with the number of corners along the x axis as nine, and the number of corners along the y axis as six. The real-world 3D points will remain unchanged, no matter how the chessboard is rotated or positioned, because translating or rotating the chessboard also translate and rotate the chessboard local coordinate system by the same amount.

Finding the corresponding points in the image involves object recognition, locating the chessboard, and then locating the internal chessboard corners. This is done by the OpenCV function *findChessboardCorners(image,patternSize,corners)*, where $image$ is the input image, $patternSize$ is a tuple describing the $x \times y$ corner chessboard, and $corners$ is the output two dimensional array of image points, row for row, left to right. To improve the accuracy the points detected, we can call the function *cornerSubPix* to get more accurate results. *cornerSubPix* use the gradients of the chessboard squares to determine more accurately where the corner is. An example of chessboard corners located by the algorithm, is seen in figure 3.7.

Now, we have two arrays of corresponding points. By calling the function *calibrateCamera*, OpenCV approximates the camera matrix and distortion coefficients, by using the corresponding points to solve for the degrees of freedom in the camera matrix C, and the distortion coefficients, of equation 3.1, 3.2 and 3.3. Although one picture is enough to estimate the intrinsics, the points from the chessboard will not cover all parts of the image plane, which might have

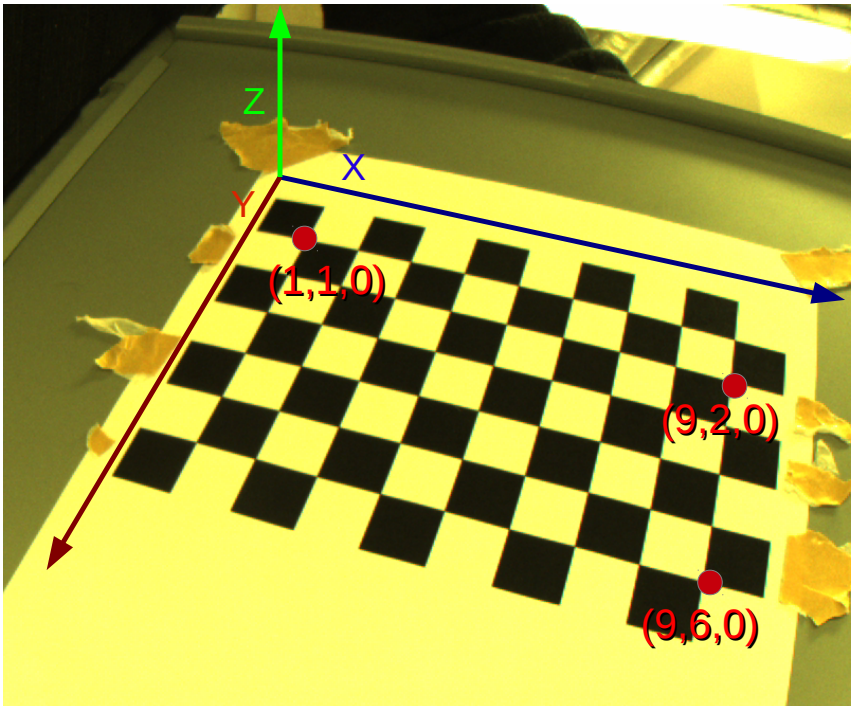


Figure 3.6: The 3D real-world coordinate system of the chessboard.

local distortions, and the approximation is likely not accurate enough. By taking multiple views of the chessboards, in different angles and positions, all sets of points are considered, by use of the Levenberg-Marquardt optimization [33]. This algorithm estimates the set of coefficients, so that the difference between 2D points and the reprojected 3D points, is minimized. In our case the minimum function is the difference between the 2D-points and the 3D-points, reprojected into camera-space using the camera matrix. Remember that $X = Cx$, where X is the image point, and x is a 3D real-world point.

Every camera has slightly different intrinsics, even cameras and lenses of the same model. Although the differences are not large, for maximum accuracy, we need to calculate the intrinsics for every individual camera. Figure 3.8, shows two images taken by two different cameras, corrected using the same intrinsics, while they have the same camera and lense models. We see that even though the models are the same, there is still differences, as one of the images appears to be corrected, while the other have quite visible distortions. In the Bagadus prototype, we used the same intrinsics for all the cameras, and hardcoded them. A free-view application needs many cameras, and future experiments with different lenses are likely needed to find an optimal setup. Therefore, we edited the original OpenCV code to read the number of cameras to calibrate, and the name of each of the directories containing input pictures, from an XML file. We also save the intrinsics in the XML format, in a file named the same as the camera name, read from the application configuration file. In this way we can handle lots of different cameras, without loosing track of which is which, and Bagadus can easily stop using hardcoded values, by reading stored the XML files containing the individual camera intrinsics.

Since the intrinsics do not change unless we change lenses, adjust zoom, or focal length, we only need to do this step once. It can be done in a very controllable environment, like in a computer lab. Once all the images of the chessboard are captured, the camera can be mounted at the soccer stadium.

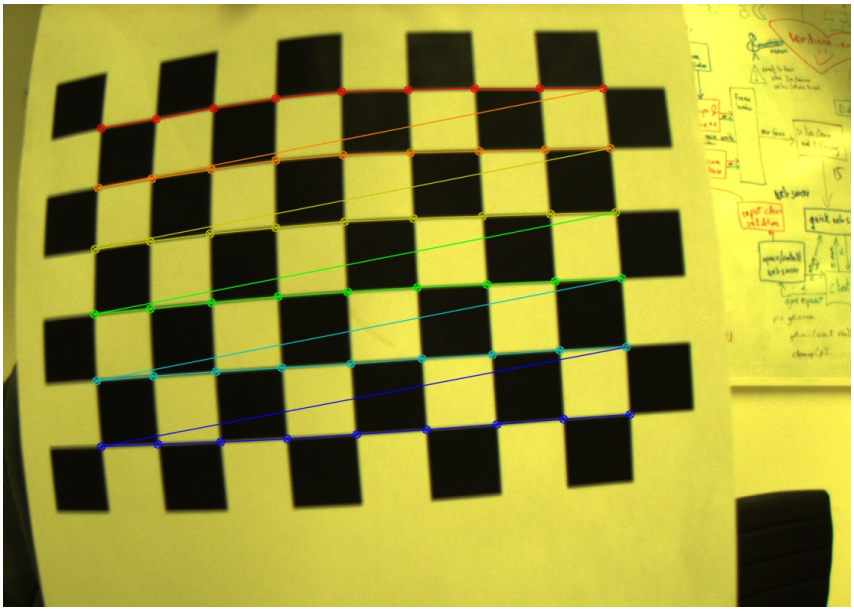


Figure 3.7: The detected corners of the chessboard.

3.2.2 Undistortion

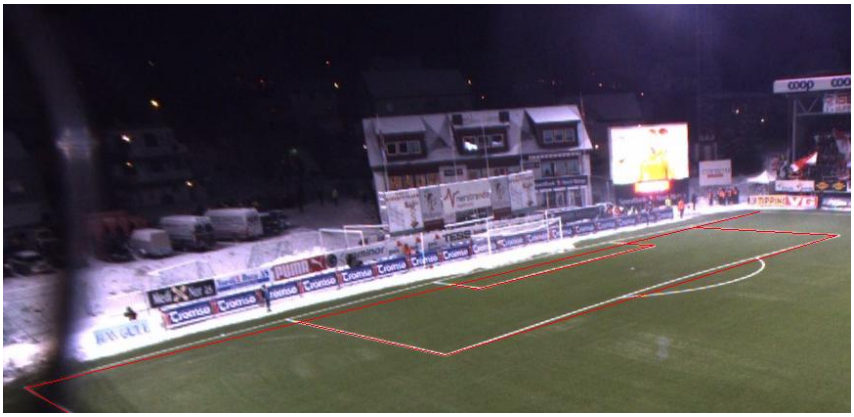
When we have the intrinsics, we can undistort, or correct for the distortions in the image. This is done by calculating where the undistorted pixels would be, using the intrinsics. The corresponding undistorted coordinates is stored in two two-dimensional arrays of the same size as the original image, one array for the x axis, and one for the y axis. Just as the intrinsics only need to be calculated only once, so can the undistortion mapping arrays. By calculating them only once, we can simply use the maps to find the undistorted pixel location, instead of running all the computations for every frame. The function *initUndistortRectifyMap*, generates the maps, based on the intrinsics and equations explained earlier in section 3.1.1.

To apply the map on a frame, we call *remap*. The function is defined such that:

$$dst(x, y) = src(mapx(x, y), mapy(x, y))$$

Because some of the coordinates in the maps are floating points, and the pixels are only integers, we need to do some kind of *interpolation*. One option is to simply round to the nearest integer. This is called *nearest neighbor interpolation*, and is the fastest one. Other methods look at the values of neighboring pixels, and fit the value based on their values, often using a polynomial function. These other methods have more smooth and accurate results, but are more computationally demanding. Table 3.1, shows the performance of the different interpolation methods available in OpenCV.

Since we can not use more than 33.3ms of processing pr frame, due to our real-time constraint, we need to stick to the nearest neighbor, or bilinear interpolation. The bilinear interpolation fits data from the 4×4 neighbor area. Figure 2.5, shows an example of the pitch before and after the remapping.



(a) Camera1 undistorted

(b) Camera2 undistorted

Figure 3.8: Images captured by different cameras, corrected with the same intrinsics. As seen in figure 3.8a, there are some errors which are not present in 3.8b

3.2.3 Line Based Extrinsics

The extrinsics, or the homography to map the images onto a common plane, is needed for rectification of the images, to make the epipolar lines horizontal. The extrinsics might also be useful for the free-view application it self. For instance if you move a virtual camera around in world-coordinates, it makes sense to use the cameras closest to the virtual position, for the extraction of pixels.

A line based approach

The extrinsics change each time one of the coordinate systems is moved, meaning when you move the cameras, the extrinsics must be recalculated. This means it can likely not be done in the lab, it must be done when the cameras are mounted in the right position at the soccer stadium. To make installation of the system easy, we tried to make the process of finding the extrinsics fully automatic, using a similar approach as Alvarez et al. [16].

The outline of the approach is the following:

1. Generate a reference model of the pitch

Algorithm	Min	Max	Mean
Nearest neighbor	4.1	5.1	4.2
Bilinear	7.4	7.8	7.4
Bicubic	47.9	55.3	48.3
Lanczos	240.1	245.4	240.7

Table 3.1: Speed comparison between the different interpolation methods (per frame). [32]

2. Detect lines of the pitch, as seen from the camera

3. Estimate the homography that maps the reference model to camera coordinate space, based on corresponding lines, instead of points

4. Extract the pose, from the homography

In chapter 2.5, we discussed how to find a homography using corresponding points. For this automatic approach, we will be using lines. In figure 3.9, we see an example of a camera which does not have enough line intersections in its view, meaning we can not find four known pitch points. It can identify three points along the center line, and one at the 16m box. No more line intersections / corners are visible in the picture, but to estimate a homography which maps between planes, the points need to identify a plane. When three out of four points lies on the same line, then there is not enough information to estimate a plane. By identifying lines, and representing them on the standard form $Ax + By + C = 0$, we can use the lines to form the plane instead of the points.

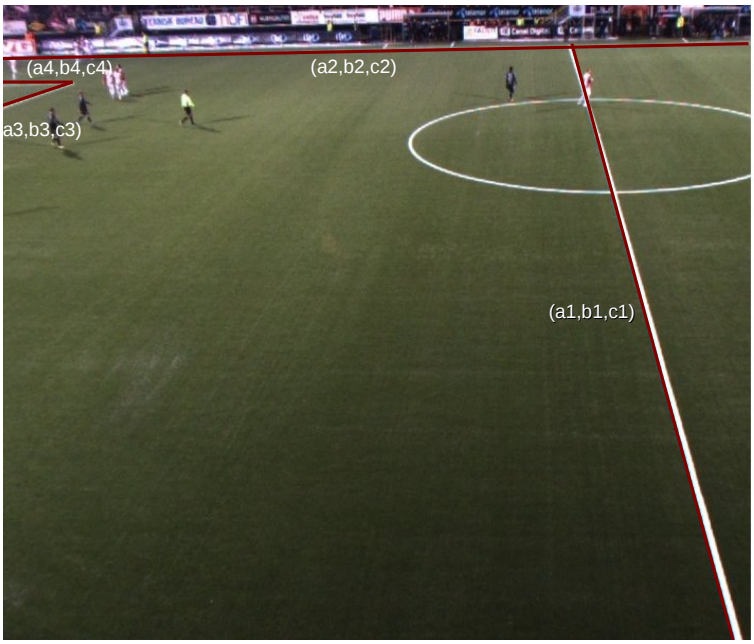


Figure 3.9: Examples of detectable lines in the pitch highlighted manually

The reference model

First, we need to create the reference model, that is a real-world model of the pitch. As seen in figure 2.7, the pitch dimensions can be anywhere from 45m×90m to 90m×120m. However, some of the line intersections are fixed size (like the 16m box), and the rest have relative positions, which makes it possible to calculate the entire model based only on the width and height, as parameters. Since the height and width of the pitch varies from pitch to pitch, we read the height and width from a configuration file.

Since the number of lines on a pitch does not vary, the lines are hardcoded at the moment, but adjusted by the width and height read from the configuration file. Every line is first represented as two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, in which the line pass through. For instance, the leftmost goal line (vertical line), will have the points $p_1 = (0, 0)$ and $p_2 = (0, pitchHeight)$. The middle line will be have $p_1 = (pitchWidth/2, 0)$, and $p_2 = (pitchWidth/2, pitchHeight)$. The 16m boxes are a bit more difficult, as they are fixed size, and centered. The line closest to the pitch center of the 16m box, parallel to the goal line, have the points $p_1 = (16.5, \frac{pitchHeight-40.3}{2})$ and $p_2 = (16.5, \frac{pitchHeight+40.3}{2})$. The rest of the lines are computed in a similar fashion. Finally, they are converted to the standard form by the following equation:

$$\begin{aligned} A &= (y_1 - y_2) \\ B &= (x_2 - x_1) \\ C &= (x_1 y_2 - x_2 y_1) \end{aligned} \tag{3.4}$$

Unlike the slope-intercept form($y = mx + b$), the standard form has the ability to represent vertical lines. If we tried to represent the goal lines with the slope-intercept form, the closest we would get is $y = 0$ for both, while in the standard form they are represented uniquely as $(A = h, B = 0, C = 0)$, and $(A = h, B = 0, C = -pitchWidth \times pitchHeight)$.

Line detection

The next step, is to automatically detect the lines in the pitch, as seen from the camera. OpenCV got the $HoughLines$ and $HoughLinesP$ methods for identifying the lines. However, the functions only work on binary image input. It treat everything as either a zero or a one, and it tries to find lines consisting of consecutive pixels of the value one.

Since most of the picture has some kind of color value, running the function right away would result in an extreme number of lines going everywhere. To only consider lines along recognizable object borders, like the white lines, we first run an edge detection algorithm on the images. The OpenCV implements this functionality in the function named $Canny$, using the canny edge detection algorithm [22]. The function basically works by calculating the image gradient for each point, and then links the gradient points to edges, if they are between two certain threshold values. Apart from the input and output arguments, the $Canny$ function has the following parameters in OpenCV:

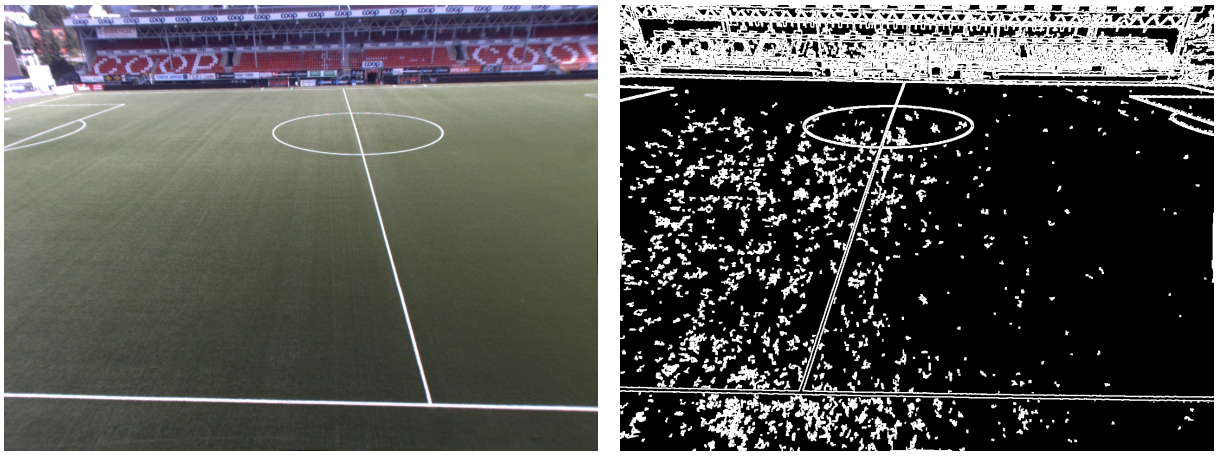
threshold1 Min threshold of what gradient to consider part of an edge. Smaller values allows for edges expand, even if the gradient values are small. Too low value risks identifying too much noise. We chose a *value of 40*. Further discussion of these threshold values follows after the function description.

threshold2 Max threshold, in addition to specifying the minimum starting value of an gradient, to be considered an edge. So to be considered an edge, it must start with at least the threshold2 value, but edges being added can have values as low as minimum threshold1. Too high value risks identifying too little, and too low risks finding too many edges. We chose a *value of 110*.

apertureSize The size of the sobel kernel, which is used to compute the gradients. A size k means a $k \times k$ kernel will be used, since a gradient must be calculated over some number of pixels. A gradient (change of intensity and the direction of the change) makes no sense if we are just considering one pixel. Larger kernels blurs the results, and might fail on small objects, but will be more robust for naturally blurred edges, like the edge of a rainbow. Since the white lines in the pitch will remain fairly sharp and at times small, if far from the camera, we left the kernel size at the *default value of 3*.

L2gradient Specifies if the method should use a more accurate, but more computationally intensive equation, to calculate the gradient. Since we only have to do this operation once, and have no real-time constraint, we chose to set this parameter to *true*.

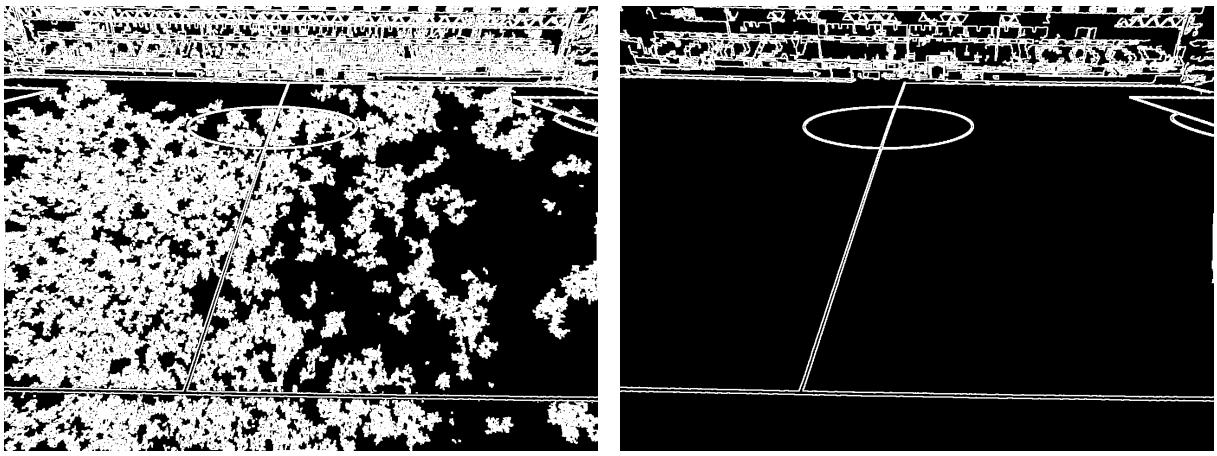
In figure 3.10, we see the input image, and the output image of the Canny algorithm. Note that we have some false positives, or noise, in the pitch, and many edges are found in the background, outside of the pitch. However as long as the edges do not form straight lines, it does not matter, as we will apply the mentioned $HoughLinesP$ function on the image as well, to extract only the edges that form straight lines.



(a) Input image (b) Result of canny

Figure 3.10: The result of a canny operation

In figure 3.11, we see the result when threshold1 is set too low, and threshold2 is set too high. Although we have set threshold1 quite low already, we see that setting it to even lower values like 10, just produce more noise stemming from already detected edges. We also see that, if we set threshold2 too high, there is no noise, but we have failed to detect all the edges along the white lines in the pitch. We chose our threshold1 and threshold2 to give us a bit too much, rather than too little, as we can filter the edges with other methods to reduce the number of detected lines. As seen in figure 3.10, some of the lines are not consecutive, especially some of the skewed lines. We found that by blurring the image with a 2×2 kernel, using the OpenCVs



(a) Canny with threshold1=10 (b) Canny with threshold2=400

Figure 3.11: Tweaking canny parameters

$boxfilter$ function, we could make every white line thicker in every direction, thus increasing the chances of detecting lines.

The next step is to detect the lines, based on the already detected edges, using the $HoughLinesP$ or $HoughLines$ function in OpenCV. The main difference between the two functions is that $HoughLinesP$ is probabilistic, which means it is implemented to select points at random, and the points it assigns to a line, can not be assigned to another line. Therefore, $HoughLinesP$ is not deterministic, and does not give the exact same result, for the same input. The functions work on lines represented in the polar coordinate system. Thus, a line can be expressed as two variables r describing the distance from origin, and θ describing the angle based on the x-axis:

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right) \tag{3.5}$$

Rewriting with respect to r we get:

$$r = x * \cos\theta + y * \sin\theta \tag{3.6}$$

By assuming the points x and y to be constant, and plotting the values when θ and r varies, and we are plotting all the kinds of lines that pass through he given point (x, y), the result is a sinusoid. By doing this multiple times, letting the points (x, y) be the edge-pixels in the input image, we get several such sinusoids, and the sinusoids that intersect at the same point, in the θ - r plot lies on the same line in the input image. Thus, the HoughLinesP detects lines by finding such intersections in the graphs, where the edge points (x, y) in the input image, is put into the line equation 3.6. Figure 3.12, shows the family of lines passing through the points $(8, 6)$, $(9, 4)$ and $(12, 3)$. For instance we see that the line with $r = 10$, and $\theta = 0.5$, pass through the points $(8, 6)$ and $(9, 4)$.

The OpenCV function $HoughLinesP$, got the following arguments:

image The 8bit input image, which we set to the output from the $canny$ function

lines The output vector, returning all the lines found on the form of its two end points $p_0 = (x_0, y_0)$, $p_1 = (x_1, y_1)$. Both points are concatenated into a 4-element vector

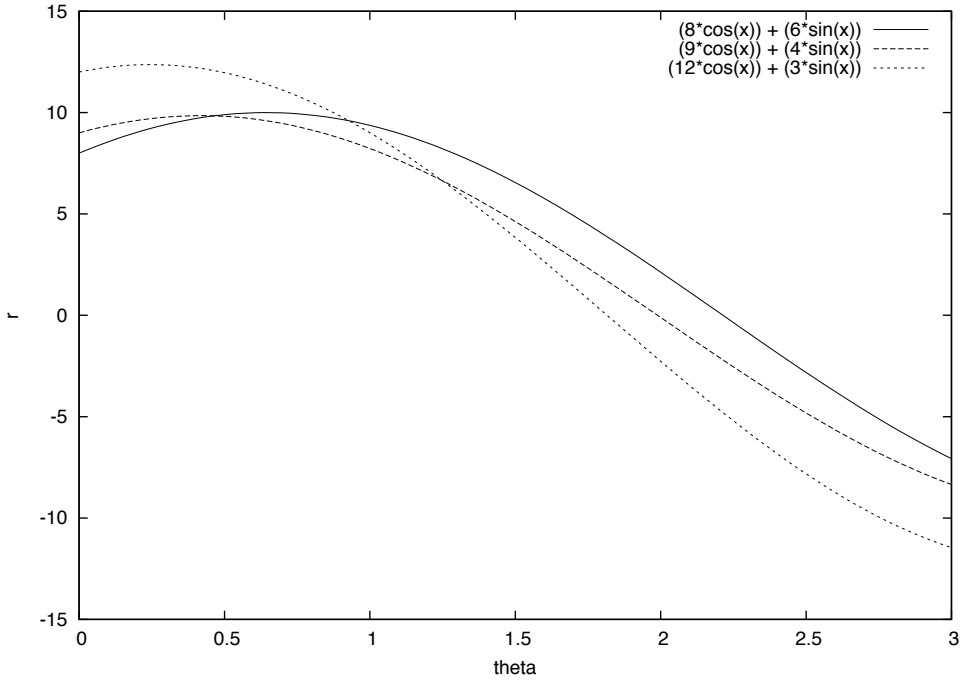


Figure 3.12: Plot of lines passing through the points $(8, 6)$, $(9, 4)$ and $(12, 3)$

rho The resolution of r in pixels. We chose to set it to 1, the highest resolution possible, as speed is not really an issue at the calibration stage. This specifies how much to increase r when checking for intersecting graphs. Higher resolution (lower number), yields more accurate results.

theta The resolution of the θ in radians. We chose to set it to one degree$(CV_PI/180)$.

threshold The number of line intersections to be considered a match. We found a value of 130 to almost always identify all the lines, and at the same time not detect too much noise.

minLineLength The minimum line distance in pixels. Lines of a shorter length are rejected. This is somewhat tied to the $threshold$ parameter, but we allow some line-gaps in between each points. We set the minimum length to 50 pixels, to filter out the small insignificant edges, while still identifying all the white lines in the pitch, which will almost always be more than 50 pixels, unless the camera is placed extremely far away or using extreme camera angles.

maxLineGap The maximum amount of distance allowed between two points to still consider it a consecutive line. We found a number of 3 pixels to be sufficient, since the edges of the white lines are almost always consecutive.

Figure 3.13, shows an example of lines detected from Alfheim stadium. Unfortunately, many of the lines detected are lines outside the pitch, for instance, numerous lines are detected

on the stands. Such a large number of lines outside the pitch is identified, that they will likely impact our automatic calibration. Note that the lines in the background are not noise, they are actual lines, and there is not much we can do with the parameters of $HoughLinesP$ to improve the result. We could increase the $minLineLength$ parameter, but that would mean small lines, or partial lines in the pitch would be discarded.

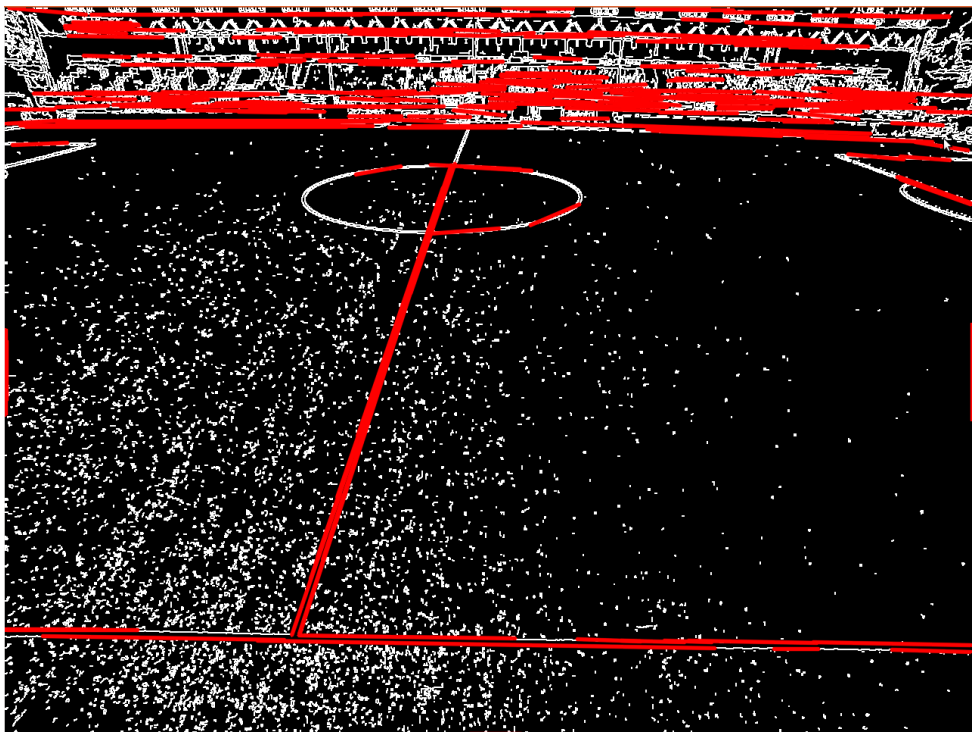


Figure 3.13: All detected lines returned by $HoughLinesP$

To reduce the number of lines, we need to somehow filter the set of detected lines. We do this by filtering them based on color information, as we assume the pitch lines will always be white. However, we use a somewhat lower value than 255, as we try to take into account that there might be shadows, or other lighting conditions, that makes the lines look gray. In addition to checking that the lines them selves are white, we also check that there are some green next to the lines, on either of their sides. We found that checking the end points of the line is enough to limit almost all false matches, but it could also be extended to check all points along the line intersecting the two line end-points.

Note that the filtering as described in algorithm 1, is quite flexible, as it only looks at the sum of the color channels for determining if a pixel is white, so if it would correctly identify the line as white if it passed through a shaded area. Also, when checking if a pixel is green, we only check if a pixels green component is larger than half of the red and green channel combined.

Ideally, we should calculate the normal of the detected line, and check for green pixels along the normals, but the quick solution in algorithm 1 will work well, unless the image resolution is so high that 10 pixels in every cardinal axis direction is no longer enough. However, since the function is working on edges, we are extremely close to the green areas, so this scenario is highly unlikely.

This is done for the two end-points of the line, and only if both points is found to have a white pixel nearby, with green on both sides, do we consider the line to be a pitch line. In

Algorithm 1 Check if a line-point is close to a white pixel, with green side pixels

```
function isPointValid(point, image)
    for x = −2 → 2 do
        for y = −2 → 2 do
            pixel ← image[point.x + x][point.y + y]
            if pixel.red + pixel.green + pixel.green ≥ 450 then
                rightPixel ← image[pixel.x + 10][pixel.y]
                leftPixel ← image[pixel.x − 10][pixel.y]
                topPixel ← image[pixel.x][pixel.y + 10]
                botPixel ← image[pixel.x][pixel.y − 10]
                if right and lefts green component ≥ (red + blue)/2 then return true
                end if
                if top and bots green component ≥ (red + blue)/2 then return true
                end if
            end if
        end for
    end for
     return false
end function
```

figure 3.14, we can see the lines that remains after filtering. Compared with figure 3.13, we see that the lines in the background are gone, without loosing the lines we wanted from the pitch.

Finding the homography based on lines

To find the homography based on corresponding lines as described in 3.1.2, we use the computer vision and numerical algorithm library *gandalf*, for the computation of lines instead of points, as OpenCV does not have any implementations for this task. If we assume that the system somehow knows which lines are corresponding, then we can compute the homography by the function: $gan\_homog33\_fit(aMatch, numLines, m33P)$. The $aMatch$ parameter is an array of structs representing corresponding lines, $numLines$ is the number of corresponding lines (detected lines, and lines from the reference model), and we are using 4 lines. The $m33p$ parameter is the output homography.

To get a sense of its correctness, we can draw the lines of the reference model, as seen in figure 3.15. Then, we warp the drawn model using the computed homography, as described in section 2.5. The result of warping the drawn reference model using the homography, should be an image equal to the detected lines, both in terms of size, location, and orientation. Figure 3.16, shows the original pitch (undistorted), and the result of reference model in figure 3.15, being warped with the estimated homography.

As we can see, the homography is quite accurate, as the warped model is definitely close to the white lines in the original input image. However, if we take a look at the warped image in the lower left corner, we see a part of the middle line. In the input image however, we can not see it. Although the homography warps the image to approximately the right orientation, skew and size, it is not accurate enough for a calibration where every pixel counts. The distance between the observed points after warping, and the true projection (called the *reprojection error*), is a common way to measure the accuracy of the warping, but since our reference model contains

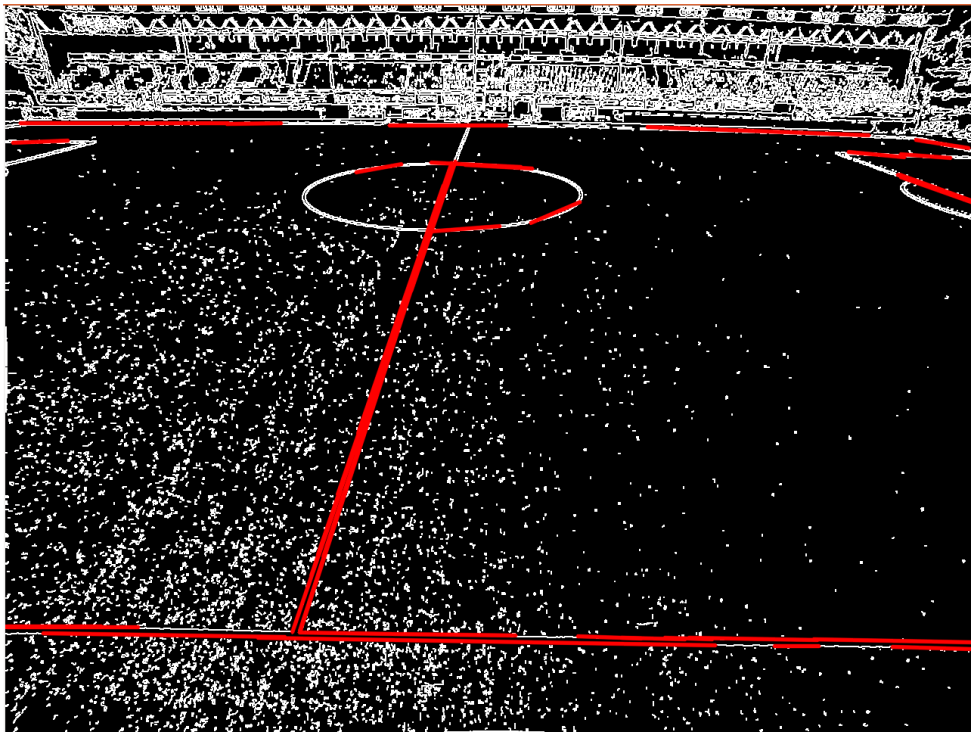


Figure 3.14: Detected lines returned by $HoughLinesP$, filtered by color information

so few points, it makes little sense to use that approach here. But by simple visual inspection, it is evident that the calculated homography is not accurate enough. In addition, the results of the warping go completely off when using certain lines, while other lines work correctly. There might be several reasons for this. If we look at figure 3.14, the line which is the furthest away from the camera is very hard to estimate correctly, so the estimated line might be a bit off. The barrel distortion might not removed entirely. More importantly, Hui Zeng et al. [54], showed that line based homographies can perform worse, and produce less accurate homographies than what point based estimation does, more specifically, when lines goes through, or close to, the origin(0,0), the line based estimation becomes highly unstable. Our reference model has two lines that pass through the origin.

Finding the corresponding lines

We have seen that we can compute a homography using corresponding lines, but that the homography is not accurate enough for our use. We will later look at another way of computing the homography, but for completeness we will briefly cover the last step of the automatic line based homography estimation. We have both estimated lines and reference model lines, but so far we have assumed that we already know which lines corresponds to which. In reality, the computer does not know.

The idea is to do like we did when checking the correctness of the warping. We warp the reference model, and check the difference between the warped model, and the detected lines in the image. We do this by going through every pixel in the detected lines, and find the closest

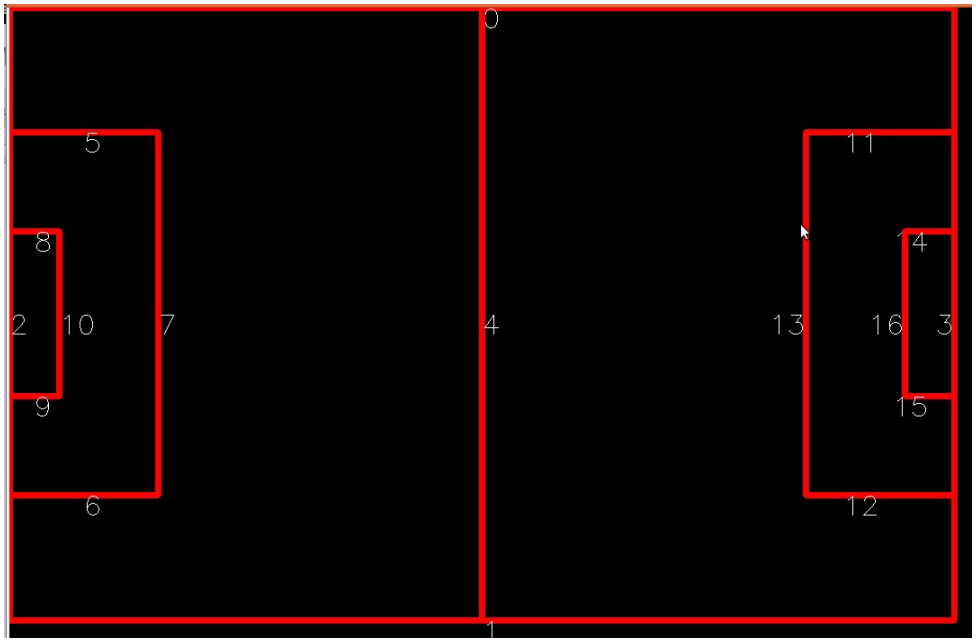


Figure 3.15: Reference model drawn

distance to a pixel which is not zero in the warped model. Thus we have a loss function:

$$f(H) = \sum_{p_i \in P} distance(H(p_i), C)^2$$

Note that we compute the square of the distance, in case of negative distance. Where P, is the set of line pixels in the reference model, C is the set of pixels in the detected lines, and the function $distance$ finds the distance between the pixel-coordinates of the $H(p_i)$ and the $nearestpixel \in C$.

Thus, a first step would be compute the homography H, by selecting every possible combination of four detected lines, and four reference model lines, compute the loss function, and select the minimum value. The number of lines detected in the input image varies, but we have a number of roughly 20 detected lines, because the white lines are sometimes detected as a series of smaller lines. The reference model we use has 17 lines. For simplicity, assume we only have a low number of lines among the detected lines, say 9 lines. The total number of configurations N, needing to be checked would be:

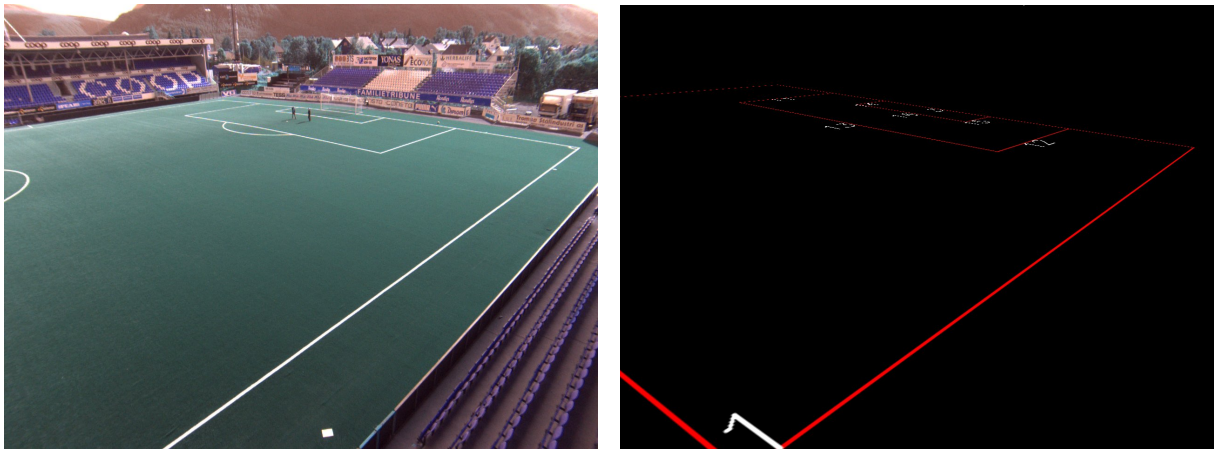
$$N = ({}_{17}P_4) * ({}_9P_4) = 172730880$$

If we assume it only takes 1 millisecond to compute the homography, warp the model, and compute the loss function, then it would still run for:

$$\frac{172730880}{1000ms * 60s * 60m} = 47.98hours$$

However, as suggested by [15], the number of permutations can be drastically reduced, by dividing the lines of the model into horizontal and vertical classes. The model has 10 horizontal lines, and 7 vertical lines. The number of configurations would then become:

$$({}_2P_{10}) * ({}_2P_7) * ({}_4P_9) = 90 * 42 * 3024 = 11430720$$



(a) Original image (b) Warped reference model

Figure 3.16: Result of warping the reference model

If we again assume only 1ms for the combined computations, it will take 3.17 hours. However, our assumptions are way too low.

Operation	Min	Max	Mean
findHomography	0.1	0.3	0.11
warpPerspective	21.6	38.5	21.9
calculateError	1.5	2.3	1.6

Table 3.2: Time spent on operations for each configuration-test (in milliseconds).

Table 3.2, shows the times for the various operations, that have to be performed for each configuration. The homography calculation and warping is fairly well optimized already with OpenCV. Our computation of the loss function has room for improvement, for instance by scaling the two images prior to warping, however, reducing the image size means losing accuracy.

The automatic calibration using line detection has several advantages, like easier setup that requires less human interaction. More importantly, using lines can make our camera setup more flexible, as a line might be observable from a camera, while the line intersections might be out of the view-range.

3.2.4 Point Based Extrinsics

We have seen how the automatic line based approach to estimating the homography was not accurate enough in our case. Also, it took a considerable time to compute, even if we did some optimizations. Since the number of configurations is too large without doing advanced checking to limit both the detected lines, and reference model into lines, we chose to implement a more manual point based estimation.

The way compute the homography is the same as in section 2.5. However, in the Bagadus prototype, we found each corresponding point manually in an image editor for every camera, mapped them to the corresponding points, and hardcoded them in the source code. For the point based extrinsics, we chose to read the number of cameras from an input file, then sequentially

loop through every camera, and compute the homography mapping the real-world-coordinates to the camera. This is done first by displaying an image from the camera, asking the user to input the number of points he want to use, and then use the OpenCVs graphical user interface (GUI) to register the (x, y) position of a mouse click in the window. Every click stores a point to a vector of points, as seen by the camera. Then the user is shown a scaled model of the reference pitch, with a number being displayed next to every line intersection. In the original reference model, one pixel equals one meter, if drawn in its original size. However, this will be too little to be displayed for a user in the GUI, thus we scale every point with a scale factor. As written earlier, at least four points forming a rectangle are needed to estimate a homography, and additional points generally increase the accuracy. After the homography is calculated, we warp the non-scaled reference model using the homography, and superimpose warped reference model onto the input image as seen from the camera, for visual inspection by the user. Figure 3.17, shows

Figure 3.17: Reference model warped and superimposed onto the image

the result of warping the reference model and merging it with the image from the camera, where the red lines and white numbers are the warped reference model image. The more overlap the lines and the reference model has, the better is the accuracy of the homography. If the user is not happy with the accuracy, he is given the option to select new points until the homography is accurate enough. In addition to displaying the image and warped model for visual inspection, we also output the reprojection error of the points. However, note that this value is actually less meaningful than the visual inspection. In a typical homography estimation, we would identify perhaps somewhere between four to eight corresponding points. Assuming we only use four points, the result is that a homography is estimated to fit those four points, and the reprojection error will be zero. While that is true for those four points, the rest of the model might be off. In general, the more points picked, and the more spread the points are, the more meaningful the reprojection error will be. As another measure of accuracy, we compute the loss function

described in 3.2.3. However this is also not that useful, as it might contain larger errors, if some detected lines not part of the white pitch lines are detected in the background. Further, it is dependent on how many lines is detected, which varies.

Since the reference model use the same coordinate system as ZXY, we can find the mapping between a player and the camera using the same homography we calculated in this section.

An automated improvement of accuracy

The points we choose might not be the optimal ones. Since we are using a simple point and click GUI for selecting the points in the image, we might hit point (x_1, y_1), while the most optimal point was for instance $(x_1 + 2, y_1 - 1)$. Picking the slightly wrong pixel points affects the accuracy more heavily if we use a few number of points, and if the points we select are closer to each other. Figure 3.18, shows how a few points located close to each other creates an inaccurate homography if the points chosen, are not accurate enough. The further from the ideal point we deviate, the larger the error becomes, as the angle makes the two lines deviate. The blue circles are the optimal points, and the dashed lines are how the optimal warped line would look. The red circles are points chosen instead of their nearby optimal counterparts. As seen, the further from the collection of points we get, the larger the reprojection error of the horizontal lines becomes.

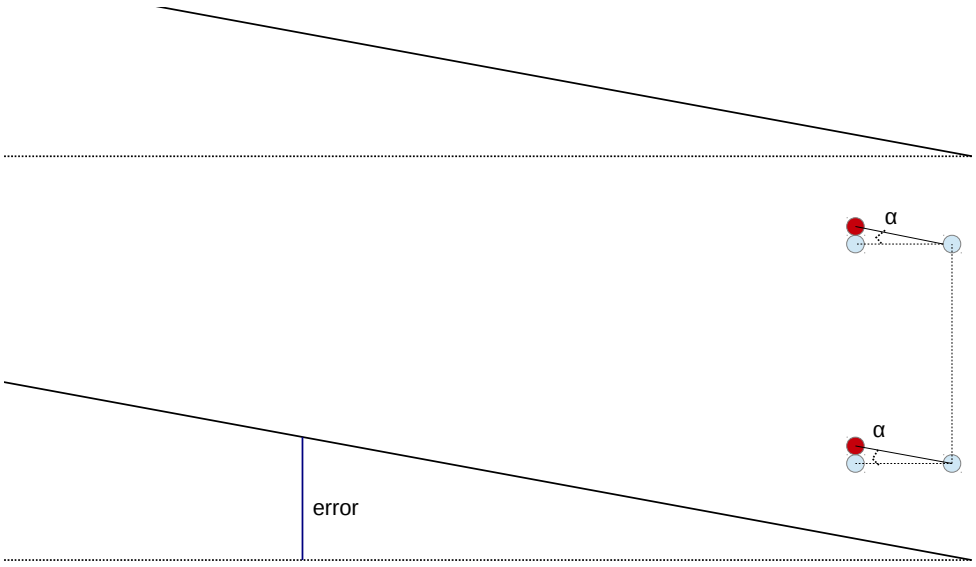


Figure 3.18: The effect of choosing the slightly wrong point coordinates

To improve the accuracy in this situation, we implemented an optional automatic improvement of accuracy, by brute force testing every combination of pixels in a small area around each point/pixel we selected with the point and click GUI. If we want to check a square of width and height k, centered on every selected point, then for each point, we check we multiply the number of point configurations needed to check by k^2, thus if we have n points, the total number of configurations to check would be k^{2n}. Since this number grows very fast as n increase, we keep $n = 4$. This also makes sense, since $n = 4$ is the case when the method is the most vulnerable to inaccurately selected points. To test if a combination of pixels is better than what we already got, we use the same algorithm as we described to find the corresponding lines, in the automatic

line based method for finding a homography. The testing of the configurations of pixels is done by eight nested for loops, like this:

Listing 3.1: Testing every configuration of pixels within a square centered around 4 original points

```
for(int p1x=MIN_PIXEL; p1x <= MAX_PIXEL; p1x++) {
  for(int p1y=MIN_PIXEL; p1y <= MAX_PIXEL; p1y++) {
    for(int p2x=MIN_PIXEL; p2x <= MAX_PIXEL; p2x++) {
      for(int p2y=MIN_PIXEL; p2y <= MAX_PIXEL; p2y++) {
        for(int p3x=MIN_PIXEL; p3x <= MAX_PIXEL; p3x++) {
          for(int p3y=MIN_PIXEL; p3y <= MAX_PIXEL; p3y++) {
            for(int p4x=MIN_PIXEL; p4x <= MAX_PIXEL; p4x++) {
              for(int p4y=MIN_PIXEL; p4y <= MAX_PIXEL; p4y++) {
                vector<Point2d> pointsLocal = chosenPoints;
                pointsLocal[0].x += p1x;
                pointsLocal[0].y += p1y;
                pointsLocal[1].x += p2x;
                pointsLocal[1].y += p2y;
                pointsLocal[2].x += p3x;
                pointsLocal[2].y += p3y;
                pointsLocal[3].x += p4x;
                pointsLocal[3].y += p4y;

                // pointsLocal now have the configuration of
                   the 4 points to check
                // Do the test for the pixels here:
              }
            }
          }
        }
      }
    }
  }
}
```

The reason for keeping the number of points, and the search distance $k = (MAX\_PIXEL - MIN\_PIXEL)$, to small numbers is evident by the number of nested for loops and complexity. Given a constant search space k, we get the big-O notation $T(n) = O(k^{2n})$. Inside the loops, we calculate a temporary homography, warp the scaled reference model using the new temporary homography, and calculate the loss function comparing the warped homography to the detected lines in the camera, described earlier. If the loss function for the given pixel configuration is less than what we already got, we save the points. If the loss function is larger than the error we already have, we discard that configuration. The computation of homographies and warping, is already heavily optimized OpenCV functions. To gain any further optimizations, we would need to utilize the video card.

However, we can speed up the process of identifying the lines by choosing to warp the model to the image as captured by the camera. This means we can do the line detection only once,

outside the nested for loops. We define a function $findNearestPixel()$, for finding the closest pixel $p_{detectedLines}$ to a point $p_{warpedModel}$, which is not zero in the detected lines image. To find the total deviation between the detected line pixels and the pixels of the reference model, the easiest solution would be to loop through all the pixels in the detected lines image, and then run the function $findNearestPixel(p_{detectedLines})$ for every pixel, and sum the result. However, this can be optimized a bit by storing the location of the points we find in the detected lines, in a vector beforehand, so that we can avoid looping through all the zero-pixels in the detected lines image. Also, while finding the sum of the total error, we can immediately discard the configuration once the sum exceeds the error of our current configuration, enabling us to only calculate the error for a subset of the total collection of points found in the detected lines. Algorithm 2, shows how to calculate the loss function for the point configurations: Another step we can do

Algorithm 2 calculate point configuration total error

```
tmpError ← 0
for i = 0 → numberOfPointsDetected do
    tmpError ← tmpError + findNearestPixel(pointCollection[i], searchImage)
    if tmpError ≥ currentError then
      break loop
    end if
end for
```

to optimize the computation of the loss function, is to reduce the number of points to check among in the detected lines. We could check every point from line start, to line end, but it is a bit excessive, and we gain very little accuracy. Bear in mind, that since the values from the loss function is calculated with the same amount of points, namely $numberOfPointsDetected$, it does not matter that much, if we remove some points in the detected lines, since all the lines are compared to the same loss function. It is however, important to note that we can not reduce every line to just two points (the beginning and end of a line). Doing this would mean that a line with a length of 10 pixels would carry the same weight as a line of 100 pixels. So, it is a question of speed versus accuracy, but if we choose the distance in between each point sample to be a bit less than the smallest line we detect(we discard lines less than 100 pixels), then this is no longer the case, although some accuracy with respect to the weight/effect of the detected lines is lost at the very end of every line unless $(lineLength\%sampleFrequency = 0)$ or $sampleFrequency = 1$.

Calculating the distance to the nearest pixel $p_{referenceModel}$ in the reference model, given the pixel $p_{detectedLine}$ (from now on p_r and p_d), can be done by checking every pixel in the warped reference model, check if the location is not zero, calculate the distance

$$\delta = \sqrt{(p_r.x - p_d.x)^2 + (p_r.y - p_d.y)^2}$$

and update it every time δ is less than what we have already found.

This is a quite naive approach, with complexity $O(imageWidth * imageHeight)$. A more efficient approach could be to start looking near the pixel, then progressively expand in all directions, stopping at first match. The worst-case is the same, but the average is better, but depends largely on the computed homography. The algorithm we used in our implementation was a simplified one, that gives some speed, at the expense of accuracy.

Listing 3.2: Calculate the distance to the nearest pixel not zero

```
double optimizedLook(Mat &image, int x, int y) {
  int add = 0;
  int xSize = image.rows;
  int ySize = image.cols;
  double distance = xSize + ySize;
  while(true) {
    if(image.at<char>(min(x+add, xSize-1), y) != 0) {
      distance = min(add, xSize-1);
      break;
    }

    if(image.at<char>(max(0,x-add),y) != 0) {
      distance = add;
      break;
    }

    if(image.at<char>(x,min(ySize-1, y+add)) != 0) {
      distance = min(add, ySize-1);
      break;
    }

    if(image.at<char>(x, max(0,y-add)) != 0) {
      distance = add;
      break;
    }

    if(image.at<char>(min(xSize-1,x+add),min(ySize-1, y+add))
       != 0) {
      distance = sqrt(pow(min(add, xSize-1), 2) + pow(min(ySize
         -1,add),2));
      break;
    }

    if(image.at<char>(min(x+add, xSize-1), max(0,y-add)) != 0)
       {
      distance = sqrt(pow(min(add, xSize-1), 2) + pow(max(0,add
         ),2));
      break;
    }

    if(image.at<char>(max(x-add,0), min(y+add, ySize-1)) != 0)
       {
      distance = sqrt(pow(add, 2) + pow(min(ySize-1,add),2));
      break;
    }
```

```
    if(image.at<char>(max(0,x-add), max(0, y-add)) != 0) {
      distance = sqrt(pow(add, 2) + pow(add,2));
      break;
    }

    add++;

    if((x-add < 0) && (x+add > xSize-1) && (y - add < 0) && (y
       + add > ySize-1)) {
      break;
    }
  }
  return distance;
}
```

By checking every direction from 0 to 360 degrees, with 45 degrees in between each line of checks, we can eliminate a for loop, and replace it with eight if-tests. We do not need to calculate the distance in x and y distance anymore, as the distance is increased only once in each loop, and stays the same for all the if-tests. If we can not find a single matching pixel in the image, due to a homography that is completely off, then the add variable will never be updated, and we return the sum of the width and height of the image instead. This artificial value is made, to avoid the special handling of when we can not find a pixel, as this would introduce more if-tests and unnecessary branching. The reason this algorithm works, is because the reference model consists of parallel and consecutive lines. If we test in eight directions, the real distance will not stray too far from a full search.

Extracting the pose

Now that we have found a homography that is accurate enough, we can extract the camera pose from the homography, as described in [16]. OpenCV has a function for calculating the pose, called $solvePnP$. Its parameters are the two sets of corresponding points, the camera matrix, and the distortion coefficients. The distortion coefficients was set to zero, since we undistorted the image before picking corresponding points, so doing undistortion on them a second time would produce wrong results. It does not extract the pose from the homography directly, although it is possible. It calculates the pose, given our set of corresponding points, which we used to compute the homography, minimizing the reprojection error, and in general does the calculations all over again, but it is not a big problem, as this step is only done once during initialization. And, having used the same corresponding points to calculate the homography before, we know the points will produce a reasonable pose. The paper model depicted in 3.17, is of course not up to scale to the real pitch. Dividing the width of the paper model (in centimeters) by the width of the actual pitch, we get $105/17.5 = 6$. The model printed on the paper is thus a scale of 6 less than the real model. A result of the $solvePnP$ is shown in table 3.3.

The tvec is the translation vector, and the rvec is the rotation vector. If we divide the tvec by the scale $\frac{1}{6}$, we get $x = 0.33$, $y = -5.25$ and $z = 23.18$. Measuring with a ruler, we find 23.18cm to fit perfectly measuring from the origin to the camera, and the same for the y=-5.25, measuring from the origin. However, the $x = 0.33$ is not correct, as the correct value should

Vector	x	y	z
Rvec	-1.059	0.2415	0.1017
Tvec	2.033	-31.5175	139.131

Table 3.3: Pose estimation result

have been roughly 2cm, which might be because we are only using four to nine points for our estimation. Further, there is some problems with the negative numbers, because we are working with points in a plane, we have not really defined which way is up, and which way is down. This makes the $solvePnP$ give a less accurate pose, which will in turn affect our depth estimation.

3.2.5 Homography Based Rectification

Rectification is needed in stereo vision algorithms, where two cameras capture images from the same scene. Since the epipolar lines change, when we change the relative pose between the two cameras, an optimal solution would be to calibrate the cameras after they are mounted at the soccer stadium. OpenCV provides a function for finding the pose by chessboard calibration, but this can be very difficult, as the cameras are typically mounted on platforms, far above the ground. Therefore, getting to them with chessboards are difficult.

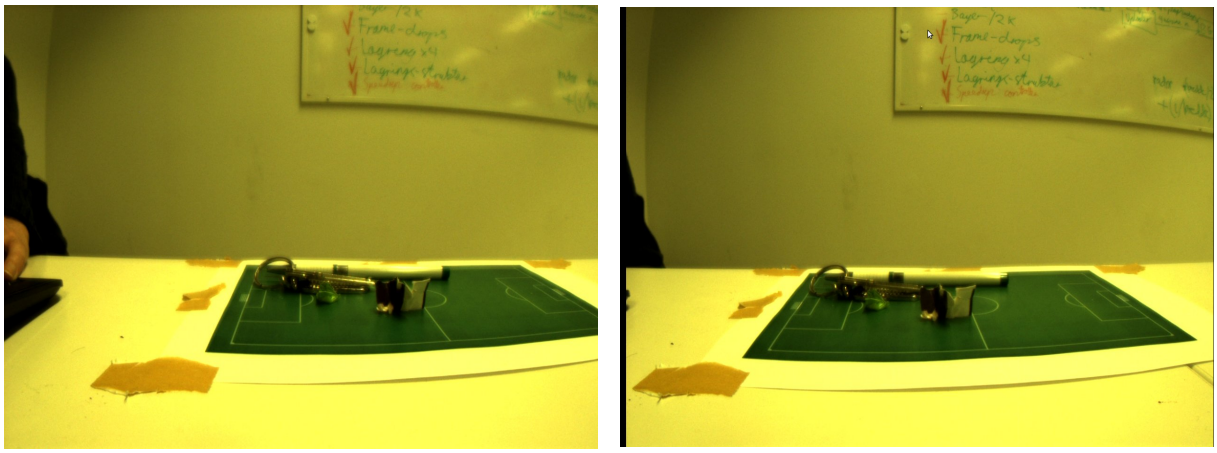
The other function provided by OpenCV is $stereoRectifyUncalibrated$ which estimates two 3×3 homographies, that warps the images from the two cameras such that the epipolar lines become horizontal. The $stereoRectifyUncalibrated$ needs the fundamental matrix F, which we can find using the function $findFundamentalMat$. It takes two arrays of corresponding image points, and the method for computing the fundamental matrix. The choices are between a 7point or 8point algorithm, or using an 8 point RANSAC algorithm.

We chose to use RANSAC, so that outliers have less impact on our estimation. Using RANSAC, the eight points are iteratively selected randomly from our number of points, and F is estimated, a reprojection error is calculated, and the F with the lowest reprojection error is chosen. We set the RANSAC threshold to one, meaning any error greater than 1 pixel is considered an outlier. Once F is calculated, we can pass it to $stereoRectifyUncalibrated$, which will return two homographies H_1, and H_2. The rectification can then be performed by warping the left and right input image, using the two homographies returned by *stereoRectifyUncalibrated*. Note that we are operating on images that have been corrected for distortions, using the chessboard calibration in the lab.

To select which points should be passed as a parameter to *stereoRectifyUncalibrated*, we first tried an implementation based on the GUI point selection, described in section 3.2.4.

As seen in figure 3.19, our two cameras are positioned parallel to the paper model of the soccer pitch, with very little distance in between them. Images from these two cameras are rectified, using a homography based rectification, based on eight to twelve selected points in the pitch. Looking at the input images, it is intuitive that we will not need much rotation to align the images in such a way that the epipolar lines are horizontal. But looking at the result of the rectification depicted in figure 3.20, we see that while most of the illustrated epipolar lines are horizontal, line 1 and 3 starting from the top are off, and you can see the bottom line as well being a bit off. More importantly, both images are rotated significantly, which we see is not necessary, by looking at the input images. We suspect that this is the result of having too few points to put enough constraints on the computation of the homography, as well as inaccurately

picking points manually. This makes sense, as we can see that the typical locations where we selected points, like the intersections of the white lines, is well rectified, while other points and features are not.



(a) Input image1 (b) Input image2

Figure 3.19: Two images taken at the same scene from different views

Figure 3.20: Rectification of images based on picked points

SURF feature detection

To get a drastic increase in the amount of feature points, we can not expect the user to click through them all in the GUI, as it will take an enormous amount of time, and it will be tedious and error prone. We made use of the automatic Speeded Up Robust Features (SURF) [19] feature detector, implemented in OpenCV. It is a blob detector, meaning that it detects areas that contains different properties, such as intensity, or change in intensity. Since it works on

intensity, and not color information, we convert our images to grayscale, using OpenCVs cvtColor method. Next, we detect the feature points in both the images, that is, the areas which stand out from the surrounding area, in terms of intensity, change in intensity etc. Typical features detected are corners, edges, etc. The SURF class constructor, takes the parameter $hessianThreshold$, which specifies the minimum Hessian to allow, which increase or decrease the number of detected features in the images. The lower the number, the higher the number of features detected. While mathematics behind the SURF algorithm is out of scope for this thesis, a more in-depth explanation can be found at [19].

To get a potentially large number of points, we found $hessianThreshold = 100$ to be a good value. Next we use the Fast Approximate Nearest Neighbor Search Library (FLANN) [39] descriptor matcher in OpenCV, to automatically connect the feature points in each image, into corresponding points, based on distance and similarities like intensity values. Figure 3.21, shows two images with several features detected, and their correspondences.

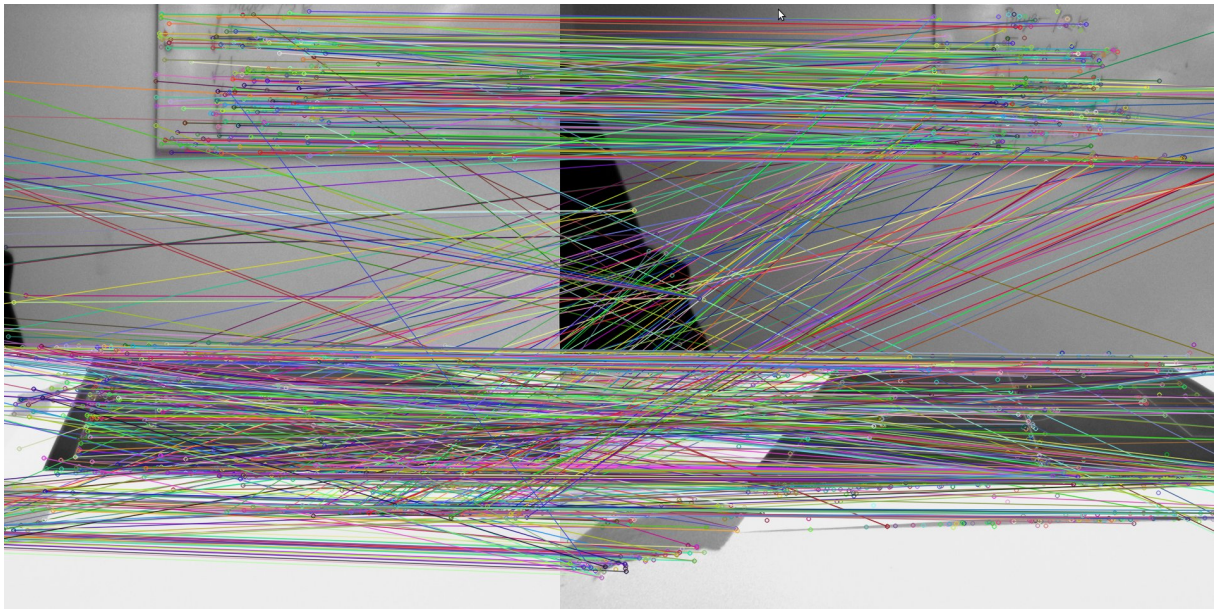


Figure 3.21: Corresponding points detected by SURF

Looking at figure 3.22, the rectification now gives a result we expect, considering that the two cameras are positioned parallel to the pitch, with the same heading, so that only a pure translation separates them. Since the two images are close to being rectified already, many of the false matches and outliers detected by the SURF algorithm can be easily spotted by looking at the diagonal lines, or lines that have a large difference in the y coordinate. We can further remove some of the outliers by manually looping through the corresponding points, check the difference in y-coordinate and discard the pairs with $\Delta y > threshold$. We set our threshold to 30 pixels. But there are still very few points in the actual pitch, which will be the case in a real soccer pitch as well, because much of the pitch is just green grass, and the SURF algorithm can not distinguish any features from the neighboring areas which is also just green grass.

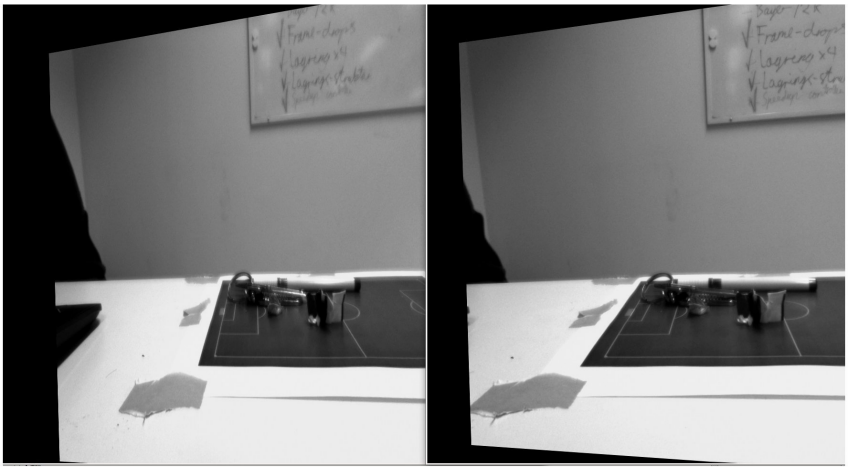


Figure 3.22: The two input images rectified using the homography computed based on the corresponding points found by SURF.

3.2.6 Chessboard Stereo Calibration

Using the automatic detection of chessboard corners, as described in 3.2.1, we can get as many points as we want covering all parts of the image. However, the chessboard calibration we saw earlier, was done for just one image. For calculating the relative pose between the two images, we need to match the chessboard points in the two images to corresponding points. Of course, it is then important that the images we take of the chessboard are synchronized. In our code base, we have a program for recording synchronized video, but not for taking synchronized stereo images.

Stereo grabber

To capture a stereo image using two cameras, capturing the image at the exact same point in time, we created a small stereo grabber application. It uses a slightly modified version of the code for capturing live streams, from the Bagadus pipeline. It automatically detect all cameras connected, ask the user to input which two cameras should be used, and starts a thread for fetching data from each camera. The trigger box makes sure that frames are captured at the same time, once every 30ms. The two threads keep a local frame count, and only put data into the buffer when the old pair of images are processed. If the program processes them too slow, the new frames are discarded. The main program converts each pair of images from YUV to RGB, and use the GUI provided by OpenCV to display each frame in two separate windows, effectively creating a live-video feed from the cameras, so that the user can see how the chessboard is positioned in both the images, right away. By pressing space, synchronized images are saved to disk in a directory specified by the user, with the name "FrameNumber_CameraName.bmp", where *FrameNumber* is a number running from 0 and upwards, so that multiple stereo images can easily be taken by repeatedly pressing space. As with the chessboard calibration for intrinsics, we found the best results when rotating and translating the chessboard for between every image capture. We tried to get all possible angles and positions of the chessboard, and doing at least 20 stereo pair images.

Chessboard calibration

Using the stereo images captured by the stereo grabber, we can calculate the relative pose between the cameras. This is done using a modifed OpenCV example code, making use of the function $findChessboardCorners$, described in 3.2.1, to detect the chessboard corners, thus finding matching points in both stereo images. The image pair file names, chessboard width and height, and intrinsics of the cameras are read from an XML configuration file.

We will be using the function *stereoCalibrate* to calculate the pose, as well as the fundamental and essential matrix. OpenCVs documentation states that *stereoCalibrate* is also capable of calculating the intrinsics, but that such a high degree of freedom and image distortions, could lead to less accurate results. Therefore, we calibrate each camera individually to get the intrinsics first, and then pass the intrinsics to the *stereoCalibrate*.

Besides the stereo images, object points, and output-arguments, it takes the following arguments:

M1, M2 The camera matrices from the two cameras.

D1,D2 The distortion coefficients from the two cameras.

ImageSize We set it to the same as the input-image dimensions.

flags Various flags for how to compute the pose and intrinsics. Since we already have input ready, we found **CV_CALIB_USE_INTRINSIC_GUESS** to give the least reprojection error. It use the intrinsics already provided as a starting out and further improves upon it using the images from the stereo grabber.

It returns the camera pose, expressed by a translation vector T and a rotation vector R. Further, it returns the average reprojection error of all the points in the chessboards. These are passed to the function $stereoRectify$, which along with the camera matrices, distortion coefficients, returns two 3×3 rectification transforms for both cameras, along with the projection matrices. These are written to an XML file, so that other parts of the system can use them.

Rectification

Unlike the rectification when using homographies, $stereoRectify$ allows us to build lookup maps, like those used for removing barrel distortion in section 3.2.2. Using the function *initUndistortRectifyMap* as described earlier, and in addition providing the rectification transformation returned by $stereoRectify$, and its corresponding projection matrix as the target camera matrix, we can combine both the undistortion operation and rectification into one map.

3.3 Applications for the Bagadus Prototype

Some of the methods we have investigated in our calibration and initialization of the depth estimation, can also easily be applied to the Bagadus prototype, to ease its cumbersome system initialization.

3.3.1 Tracking Homography

The common real-world reference system used for all the cameras in section 3.2.4, is the same as the ZXY reference system. This means that we have already found a way to calculate the homography needed by the Bagadus system for mapping between the ZXY player tracking coordinates, and the camera coordinate system. Thus, to find calculate the tracking homography, we use the method described in section 3.2.4. The automatic generation of reference models based on pitch size and width, and the use of a GUI to select points, showing the resulting homography instantly, as well as saving the resulting homography to a file using XML, drastically reduce the time spent to calculate the homography. It also creates a more agile system adaptable to change of camera setup.

3.3.2 The Stitching Matrices

As described in chapter 2.6, the current homographies we use to warp the images from the different cameras to a panorama image, was calculated by manually opening every image in an image editor to find corresponding points, then hard coding them into a small program to calculate the matrices, then those matrices were hardcoded into our prototype.

In section 3.2.4, we introduced a manual, yet faster way of finding the warp matrices for the ZXY and real-world pitch coordinate system, to the camera coordinate system, using homographies. Since we already have much of the code needed to manually select corresponding points in an image, we did some minor modifications to the UI, to allow easy generation of the matrices needed for the panorama view. We already have a function for marking points of interest like corners, edges, small objects and other features, by click on them in the UI. For the generation of stitching matrices, we identify such points twice, once for each camera, as opposed to when finding the ZXY matrix, where we clicked on features in one window, but selected predetermined coordinates based on the pitch-model. Although selecting four points is enough, more points will generally result in better accuracy. As before, the $findHomography$ function is used with the corresponding points, to find the homography. The program reads a list of cameras from a configuration input file, assuming the cameras to be positioned in a line, in the same order as in the list. The ID of which camera to be the "head camera", to which all the other cameras are warped.

However, there are a few problems, which we did not encounter for the pitch coordinate to camera coordinate mapping. Namely, if there are more than two cameras, then each camera must be mapped to a common plane, but unless the two cameras are neighbors, they do not have any common pixels.

In figure 3.23, we set "camera 2" to be the head, meaning the other images will be warped so that their coordinate spaces will fit the coordinate space of the head camera. We do this, by first finding the warp matrices for every neighboring camera, which is straight forward: Use the GUI to register mouse clicks, and show alternate images between the two cameras, allowing the user to click on corresponding points, and use OpenCVs $findHomography$ to estimate the homography. In this way, we find the homographies 1-2, 3-2 and 4-3. Computing the homography 4-2 is a bit trickier, since camera four and two have no overlap in their field of view. However, we know how to get from camera four to three, and from three to two. We can find the combined transformation matrix 4-2, by multiplying 4-3 and 3-2.

Another problem we encountered, was that when warping the points in the images, the points in the left image in each pair, might get warped to negative coordinates, for instance a

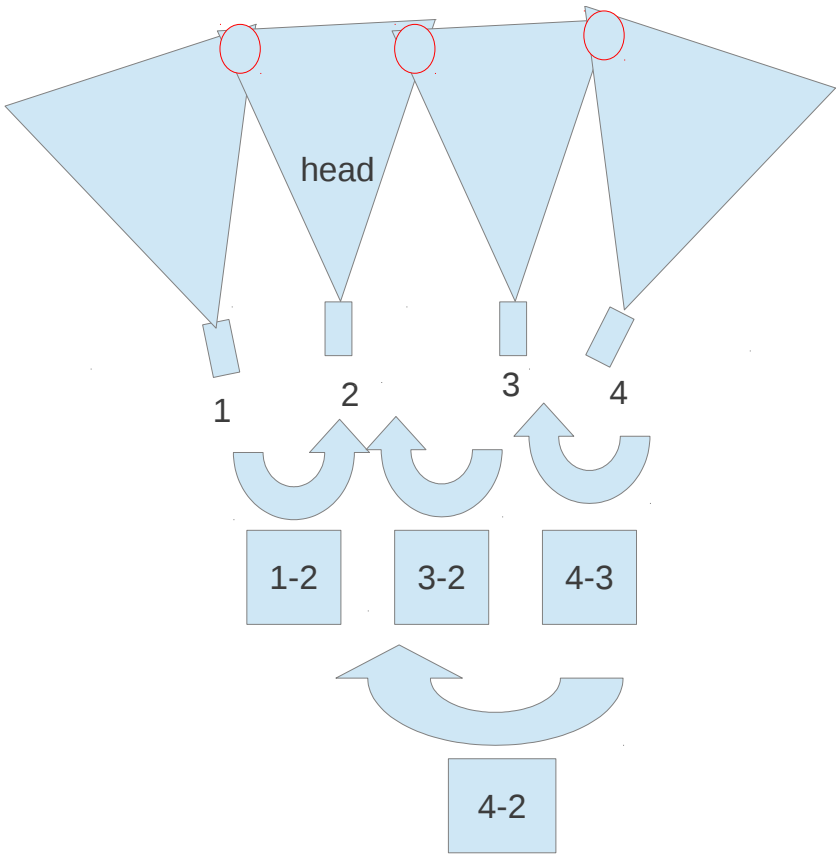


Figure 3.23: Cameras are enumerated from 1 to 4. Each cameras field of view is showed as an example. The arrows symbolize that we try to find a matrix that warps the image from its own coordinate system to the target cameras. The blue squares are the matrices. Camera 2 is set as the head

point like $(5, 50)$ might get transformed to $(-104, 50)$. Since we represent images with $(0, 0)$ being the lowest possible point in both axes, points with negative values are discarded, and the resulting image will look cropped. We can find the resulting minimum x-axis after the warp, by first calculating the homography, and applying the homography at the points $p1 = (0, 0)$, and $p2 = (0, imageHeight)$. Applying the warping matrix is the same as a matrix multiplication, as described in equation equation 2.2, on page 15. If the result of the warping is $p1_w = (x1_{warped}, y1_{warped})$, and $p2_w = (x2_{warped}, y2_{warped})$, then the minimum value a coordinate will get warped to is: $x_{min} = \min(x1_{warped}, x2_{warped})$.

When we computed the warping matrices in the Bagadus prototype, the images was manually padded with pixels, to extend the image width. This was resource demanding and time consuming. Another way of solving the problem of negative warping coordinates, is to add $|x_{min}|$ to every point in the target image, and recalculate the matrix. Thus, we can skip the padding operation, and automatically get the warping matrix, and the expected warping result, by just selecting the corresponding points once.

Like when we find the ZXY matrix, we immediately show the warped result, quickly allowing the user to judge if the results are good enough, or if new points need to be added. By reading the list of cameras from the configuration file, and specifying the head by name, we can

easily calibrate a flexible amount of cameras, should more be added to the system. The warp matrices are written to an XML file. However, we did not compute the areas of overlap, and where to do the cut for the panorama, which must still be found manually. To make the list of matrices complete, so that they can be read in a generic way, we also add the identity matrix I, for the head camera, as the head camera should not be warped, and $IH = H$.

Automation using SURF

Since we already had used SURF to find corresponding points, we investigate using SURF to automate the process of finding the corresponding points, for finding the homographies needed for the panorama warping. However, the cameras used for the panorama have varying angles, as opposed to the stereo cameras, and the SURF algorithm find a large number of false correspondences, as seen in figure 3.24. Figure 3.25, shows the result of warping the left-most image.

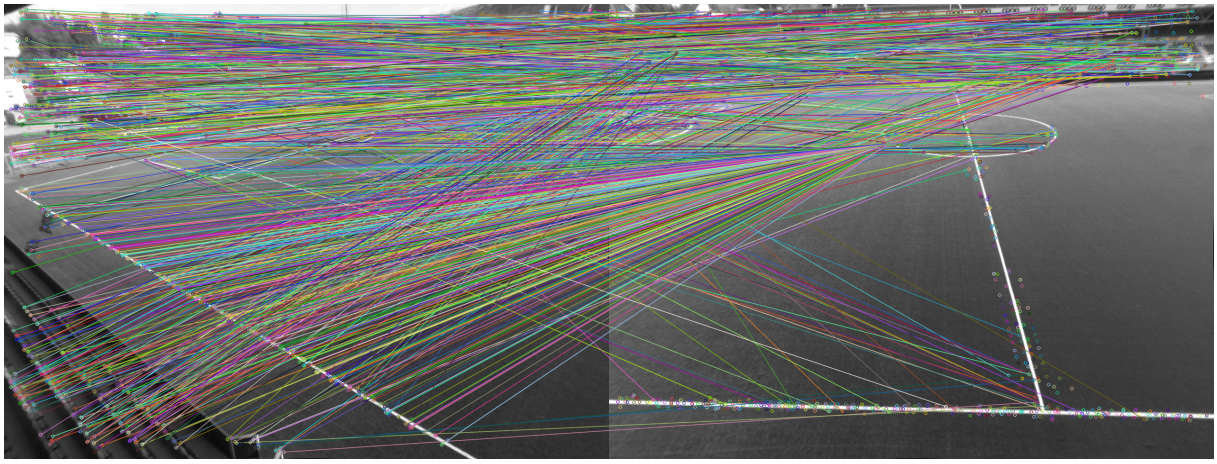


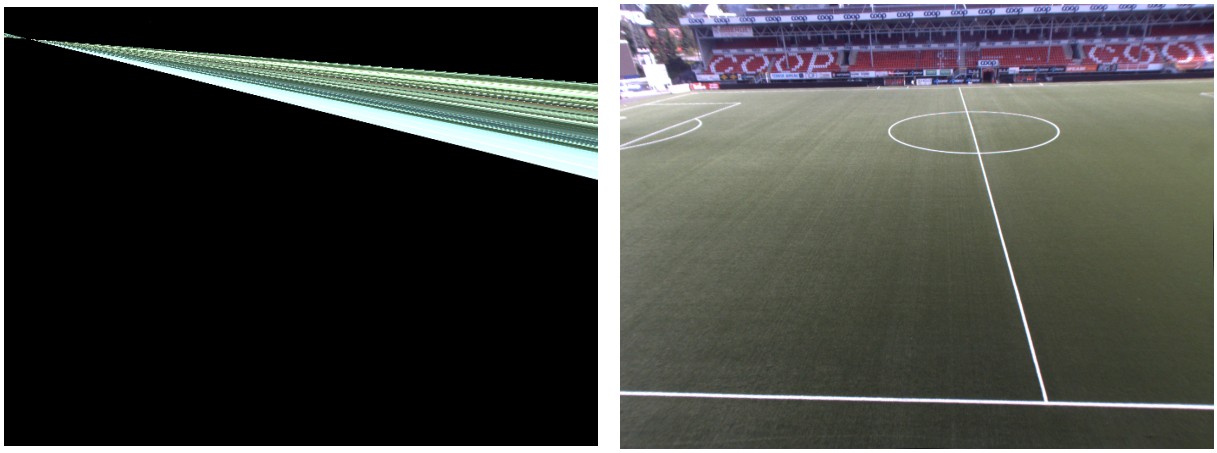
Figure 3.24: Finding corresponding points between two cameras for panorama generation purposes

While the accuracy could be improved by asking the user to manually click in the GUI, to define regions of interest, where the SURF-algorithm should operate, by doing so, the solution would again be non-automatic.

We conclude that the manually selected points result in far more accuracy, at the expense of time spent on the users part. However, since this must be done only once, with no time constraints, the time it takes the user to click on the images is worth the gain in accuracy. Figure 3.26, shows a panorama image of Alfheim stadium, warped using matrices generated by our calibration tool.

3.4 Summary

We have seen how the internals of the camera like the focal length, principal point and lens affect the results when capturing an image with a camera, especially the effects of barrel distortion introduced by the lens. We have calculated the homographies needed for warping and creating the panorama view. Both a manual point selection, and an automatic line detection approach have been presented, for calculating the homography between the real-world coordinate system of the pitch, and the pitch as viewed from a camera.



(a) Warped image

(b) Target image

Figure 3.25: A failed attempt at warping to the target image, based on SURF points. Cropped for display purposes

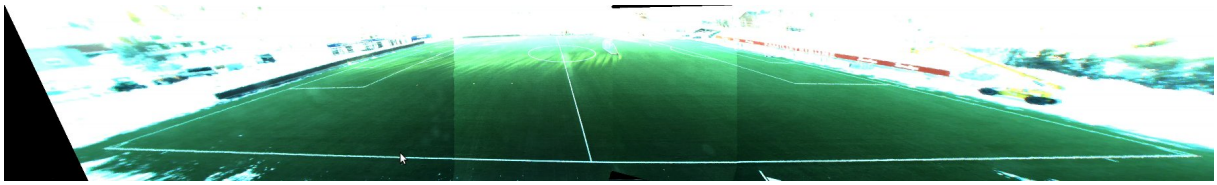


Figure 3.26: Panorama image warped using matrices generated by our calibration tool

We have presented a method for calculating the real-world pose of each individual camera, with respect to the pitch origin $(0, 0)$, using $solvePnP$. The automatic line based automatic method was not accurate enough, in its current implementation. We have presented two methods for rectifying images; one using an automatic approach, where we use the SURF algorithm automatically detect corresponding points, and a manual method where we use a chessboard pattern. We have also used features detected in the images to calculate the homographies needed to map coordinates from ZXY system, to camera coordinates.

We have seen how we can use a chessboard to calibrate each camera, using OpenCVs functions $findChessboardCorners$ and $calibrateCamera$. With the cameras now calibrated, we can perform the depth estimation, and in the next chapter, we look closer into challenges in this area.

Chapter 4

Depth Estimation

With the intrinsics, and either the rectification homographies, or the extrinsics calculated, we can perform depth estimation. In this chapter, we explain what a depth map is, discuss related works, and explain the methods for calculating the depth maps, with focus on stereo vision.

4.1 Depth Maps

An image contains $x \times y$ pixels, and where the rows and columns make the two cardinal axes of the local 2D coordinate system of the image. Figure 4.1 show an example of a 10×5 image.

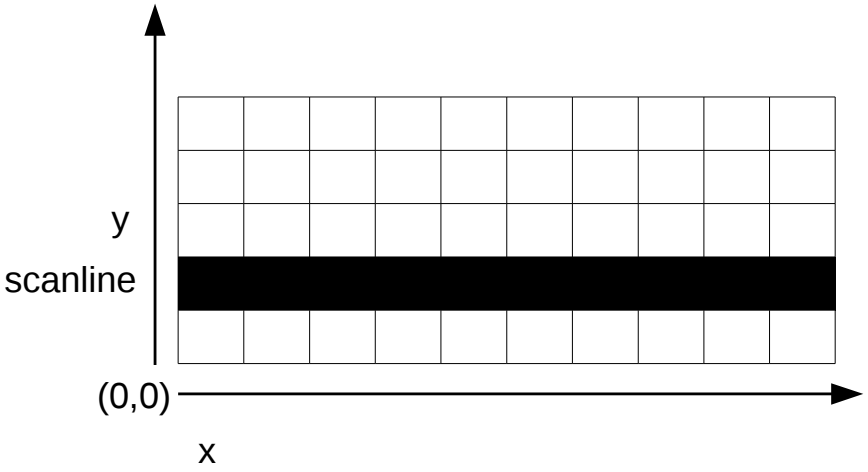
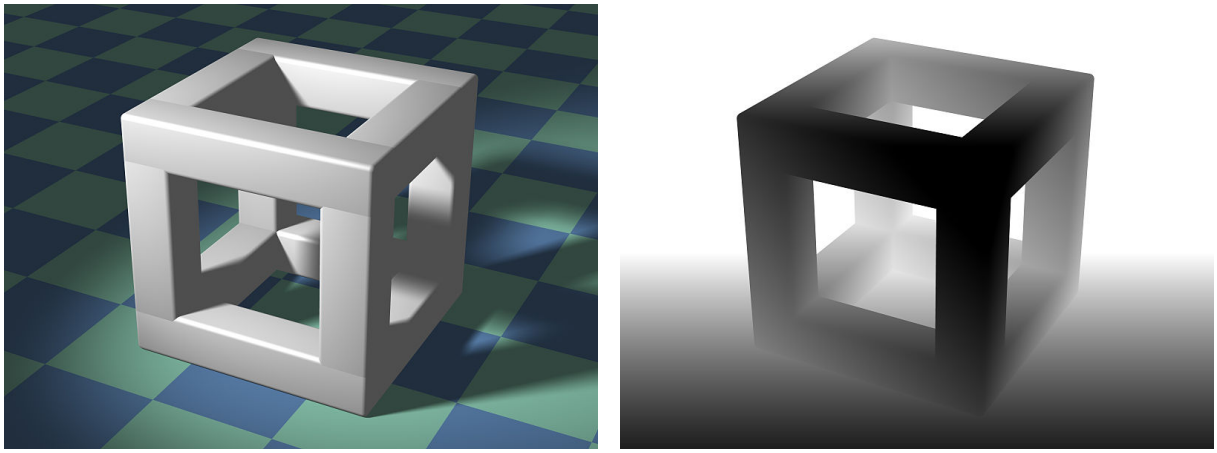


Figure 4.1: A digital raster image

The rows are sometimes referred to scanlines. The number of columns (along the x-line) is sometimes referred to as the width of the image, and the number of rows (along the y-line) as the height of the image. When a camera capture a scene, it projects the 3D world onto a 2D plane, thus we loose a dimension. Objects with real-world coordinates $(x_{world}, y_{world}, z_{world})$ get projected to (x_{camera}, y_{camera}), where the real-world coordinate system is usually different from the camera coordinate system.

A depth map is an array of the same dimensions $x \times y$ as its corresponding $x \times y$ image, where each element in the depth map denote the depth of the corresponding pixel in the image. The depth can also be included in the image as an extra channel besides the colors. Note that

the depth values in the depth map is not equal to the depth of the real-world coordinates, the depth values in the depth maps denote the distance between the point pictured, and the camera. In figure 4.2, we see a depth map of a cube, illustrated as a grayscale image. The closer a value



(a) Original image [4] (b) Corresponding depth map [3]

Figure 4.2: A depth map illustrated as a grayscale image.

is to zero, the closer it is to the camera, and larger values are further away.

4.2 Applications

Depth maps have several applications. Although not the primary focus of this thesis, the free-view application we discussed in chapter 2, is of particular interest for the Bagadus system.

Another application of depth maps is collision detection. There are many collision detection systems for cars on the marked already [41], mostly based on active systems such as laser or radar. By using stereo vision, it is possible to track multiple objects at the same time, and more importantly to determine where they are located and how much space the objects occupy. A pair of cameras could be mounted in front our at the back of a car, and the depth maps could be used to determine when an object is closer than a threshold, in which case it could alarm the driver.

Depth maps can also be used in robotics, where a robot could have a stereo camera pair integrated to function as eyes. Besides capturing visual information from the scene, the robot can use the camera pair to generate depth maps, and use the depth-maps to navigate in its environment [40]. The information can be used for avoiding crashing into objects, simply determine which object is currently the closest and prefer an interaction with this object.

4.3 Related Works

Jen-Shiun Chiang et al. [24] present a robot using stereo vision for the purposes of determining the distance to the soccer ball. Sebastian Dröppelmann et al. [13] present a guide of how to use the OpenCV library to generate depth maps, but does not present any numbers regarding the speed of the operations.

Kazunori Umeda et al. [50] present a stereo matching algorithm based on subtracting the background, thus focusing only on calculating the depth of the moving parts of the image. It is similar to what we try to do, and they do it real-time with an FPS of 37, however they operate on a 320×240 pixel image. Our images have dimensions of 1294×964, meaning we work on images roughly $\tilde{1}6.24$ times larger.

Kakuta et al. [38] use background subtraction to get a mask of only the moving parts, such as a person walking, much like we aim to do. However, they estimate the depth using information only from monocular video sequences, meaning they try to determine the depth using information from images captured by a single camera in a single location, as opposed to our goal of using stereo matching. To be able to estimate the depth from a single camera, they assume the ground to be a flat plane, which is a valid assumption for our case as well, since we are dealing with soccer pitches. The depth is calculated by the equation

$$D = H_{camera} \tan\left(\frac{\pi\left(H_{image} - H_{bottom}\right)}{H_{image}}\right)$$

where H_{image} is the height of the image, H_{camera} is the real-world distance to the ground-plane from the camera, and H_{bottom} is the bottom of the foreground object. Thus, the point where the foreground object touches the plane, in our scenario, the players feet, will be the depth of the entire player. It is a simple and efficient solution, but if a player stands with his legs far apart, only the leg closest to the camera is used for the depth of the entire player, and if two players stand aligned in such a way that they overlap, both players will get the same depth value. This is especially true for situations in close up soccer duels. We will instead implement a stereo correspondence algorithm capable of detecting several depth values within the same object of interest, to get more a more detailed depth map.

Banz et al. [18] implemented a high quality semi global stereo matching algorithm to run on a field programmable gate array (FPGA), and ruled out use of the GPU since both GPU and SIMD-processors had too high power consumption and achieved only 13 FPS on QVGA images, meaning 320×240 pixel images. However our system will not use any specialized hardware, so FPGAs are out of scope. Further, they operated on 640×480 pixel images.

4.4 Stereo Vision

Stereo vision refers to two cameras capturing the scene from different viewpoints, and using information from both cameras to estimate the depth of the pixels. The length of the line that passes through the two image centers is called *the baseline*. The length between the cameras is an approximation of the baseline. We will refer to the two cameras as the left and right camera, as illustrated in figure 4.3. For the best results, the cameras should have the same orientation, and the baseline should be orthogonal to the camera heading, or else the images must be scaled, rotated, and translated so much that both the overlapping area between the cameras is decreased, and the resolution is reduced, when rectifying the images to get horizontal epipolar lines.

4.4.1 Disparity

The basic principle behind stereo correspondence, is to find the corresponding pixels, or in some cases, blocks of pixels, in each of the stereo images. For simplicity, we will from now on refer

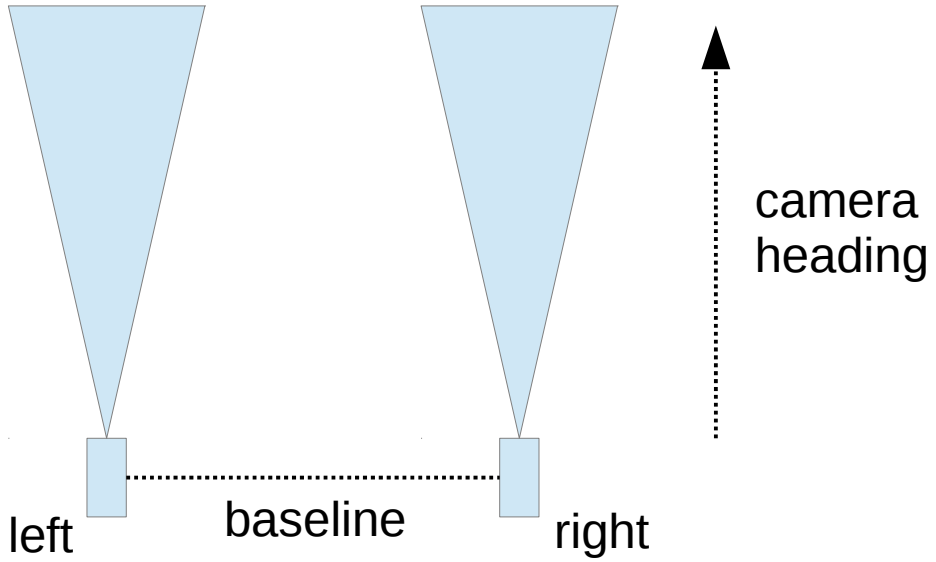


Figure 4.3: A stereo camera pair

to just blocks, where a block with a size of one is a pixel, although it is important to note that far from all stereo algorithms operate on blocks, some only match pixels. For every block in the left image, we search the right image for the corresponding block. The distance between them is related to their depth, and is called *disparity* [21]. In general, the larger the disparity, the closer the block will be to the camera. The vision of the human eye works in a similar fashion, although humans also use visual clues in the scene to estimate depth. If you hold your thumb in front of you, with only your left eye open and then quickly shut the left eye and open the right, your thumb will appear to have moved to the left. Now, if you hold your thumb right next to your eyes, you can see the disparity is much larger than when you hold it as far away as you can. But to find a matching block in the right image can be a complex and time consuming task. Luckily, the search for the matching block can be constrained. From chapter 3, we know that the epipolar lines are horizontal after we have rectified the images, and we know that a corresponding block in the right image lies either at the same location, or somewhere to the left of the position in the left image. Thus, we can restrict the search for a corresponding block to the same scanline, and to the left of the original position. If a block have position (x_1, y) in the left image, then the position in the right image will be (x_2, y), where $x_1 \geq x_2$. The disparity δ would be: $\delta = x_1 - x_2$.

In figure 4.4, we see a row of pixels from the left and right image of a stereo camera pair, where the blue dots represent pixels. The set of matching pixel pairs M is M = {(1,0), (2,1), (6,2), (7,3), (8,4), (10,9), (11,10)}. We can see that the pixels represent three objects in the image, where the middle object is closer to the camera than the two others, as its pixels have a disparity of 4, while the two objects to the left and right have disparities of 1.

4.4.2 Matching

We have seen how we can determine the disparity, by matching a block in the left image with a block in the right. The matching is partially done by comparing color or intensity of the pixels in

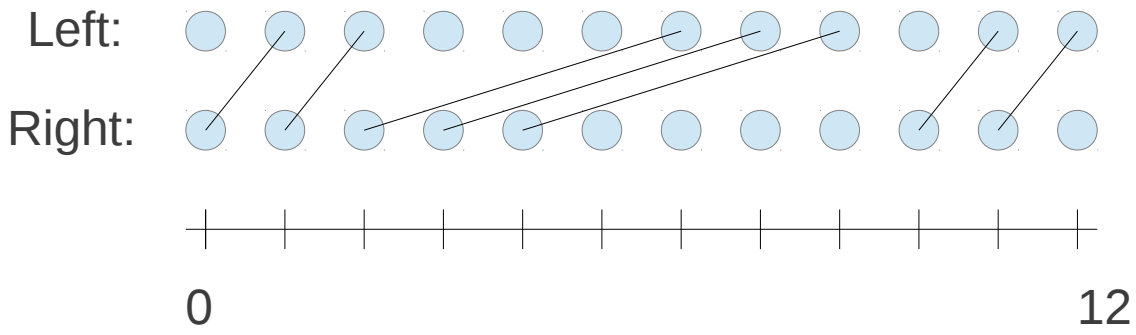


Figure 4.4: Disparity from a left and right stereo camera pair image. The blue dots represents pixels, and the lines connecting them means the pixels match.

each image, where a similar value indicates they might be a corresponding pair. Unfortunately, finding the matching pixels is difficult, even with a limited search space. For instance, if the objects in the scene are reflective, the two cameras might capture a different color or intensity value for the same object. An extreme case of this is a mirror, which will look very different depending on the position of the camera. Different camera settings could also affect the overall intensity values captured by a camera, although the color corrector mentioned in section 2.9.2, could correct for some of these differences.

Figure 4.5, shows a problem that occurs when an object is visible in one camera but not visible in the other camera, because its view is blocked by some object in the scene. This is known as *occlusion*. There is no solution for this problem, as there are simply no corresponding pixels, but the problem must be detected and handled in the best possible way, for instance by trying to estimate the disparity values instead, or by not assigning depth values at all.

Another problem regarding matching blocks, is that when both the left and the right image contain large surfaces without textures, it becomes impossible to match one block to the other. For instance, if both cameras are aimed at a large smooth white table that occupies large parts of the image plane, then it becomes impossible to determine which blocks are corresponding pairs.

4.4.3 Calculating the Depth

The output of the stereo correspondence algorithms is not the depth of the pixels, but the disparities. Although they are not equal, they are correlated, as a large disparity value means a small depth value - the disparity is inversely proportional to the depth. A depth can be calculated from its correlating disparity value, using triangulation [33]:

$$Z = \frac{fT}{d} \tag{4.1}$$

where Z is the depth, f is the focal length and T is the baseline.

4.4.4 Algorithms

There are several stereo correspondence algorithms. They differ mostly in the way they solve these problems in section 4.4.2. Some well known stereo correspondence algorithms:

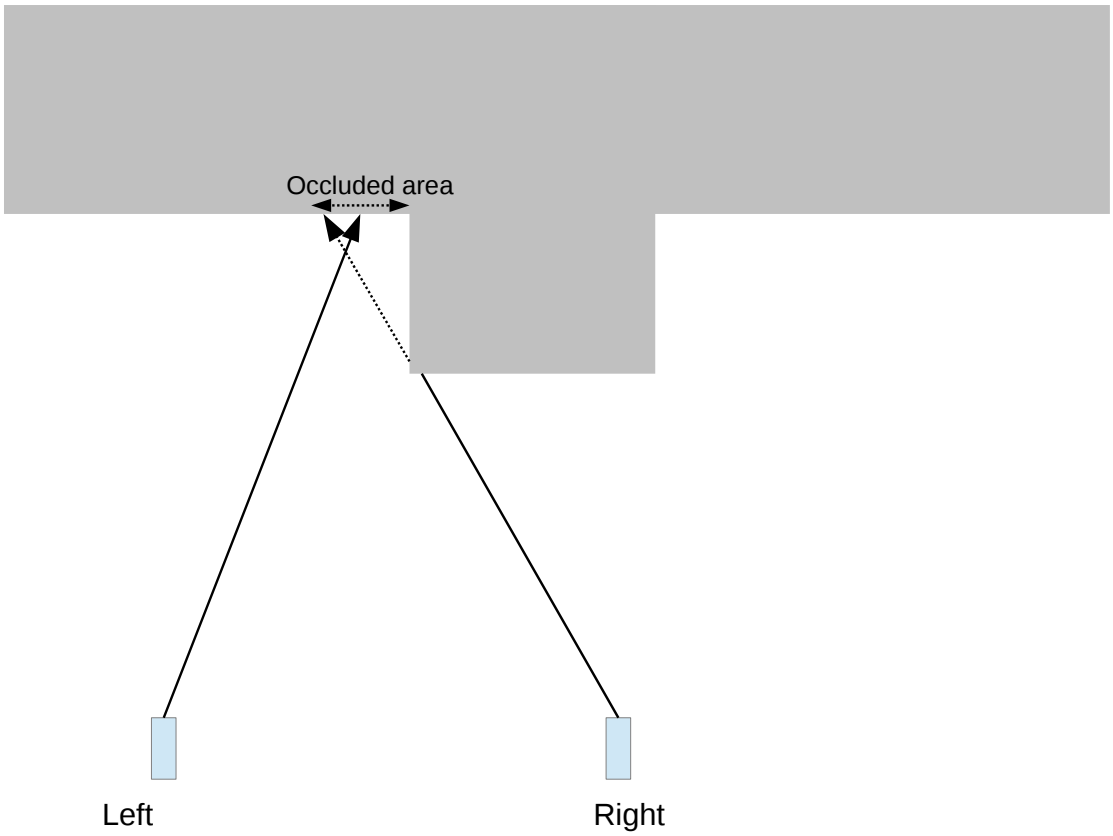


Figure 4.5: Two cameras capturing the same scene from different viewpoints. The building obstructs parts of the view for the right camera

- Block Matching (BM) [48]
- Semi-Global Block Matching(SGBM) [35]
- Belief Propagation (BP) [29]
- Constant Space Belief Propagation [52]
- Graph Cuts [36]

The algorithms we will apply are based on block matching.

4.5 Quality Assessment

To give a quantified number of the accuracy of the algorithms, we have to know the real depth values of the objects, or a disparity map where the disparities are known to be true beforehand, usually because they are hand drawn. These disparity maps are known as *ground truth*.[1] Thus, when testing new algorithms, or implementing new algorithms, these data sets are very useful as we can compare the results of the stereo vision algorithms to the ground truth, and get an

[1]Several such sets with ground truth can be found at `http://vision.middlebury.edu/stereo/data/`

accurate quantified error, like how many percent of the pixels were given a correct disparity value.

Since we do not implement any new algorithms, running these standard data sets will be of limited use for us, as the correctness and accuracy of the existing algorithms are well explored, and readily available in other works. For instance, Kjetil [28] performed several tests against ground truth data sets, and measured both the accuracy and the speed of the algorithms. We can see from his results, that the BM and SGBM algorithms have potential to be real-time, but that BM has a high percentage of disparity errors. Instead, we will look at how some of the more common algorithms perform, applied to our own data sets.

Operation	**FPS**	**average error (%)**
BM on CPU	45	28.4
BM on GPU	43	41.6
SGBM on CPU	13	12.3
BP on GPU	1	10.8
CSBP on GPU	0.15	10.4

Table 4.1: Frame per seconds for a 320×240 image set [28]. Upload and download times for the GPU is included. The average error is calculated by comparing the disparity of running the algorithm, to the ground truth for four different data sets

4.6 Implementation

There are several implementations of the stereo correspondence algorithms, but we will use OpenCVs library. Since this part is done online in our pipeline-model, these operations can not spend more than 33.33ms of processing-time, as we need to deliver 30FPS. We will measure each process that can be pipelined, or parallelized, individually.

4.6.1 PreProcessing

Before we can run the stereo algorithms, the input images from the cameras need some preprocessing, such as removing geometrical distortions, rectification, and so on.

Rectification

If we use the chessboard calibration from chapter 3, the next step is to apply the rectification based on the rectification maps we generated in section 3.2.6, to make the epipolar lines horizontal. Further, the rectification mappings were combined with the maps for correcting lens distortion, allowing both operations to be done using the same mapping. Also, these mappings can be applied to every frame no matter if the scene changes or not, as the rectification mappings are only dependent on the relative pose between the cameras and the intrinsics. To apply the mappings, we use the function $remap$ as described earlier in section 3.2.2, with its performance listed in table 4.2.

If we chose to use the homography based rectification, then the rectification is done by warping the left and right image using $warpPerspective$, with the computed homographies

for the left and right image, respectively. In addition, we have to do the remapping operation using the maps for correcting geometrical distortions, discussed in section 3.2.2. As seen in table 4.2, it takes 48.9ms to warp both left and right image, which is longer than our real-time constraint, however, we can devote 1 core to each left and right image. But, running a remap to correct the geometrical distortion, and using the *warpPerspective* function is slower and creates unnecessary load on the CPU, compared with the rectification method where rectification and geometrical correction is combined into one map.

Grayscale conversion

Before we can apply the stereo correspondence algorithms, we have to convert the images to grayscale. Most stereo correspondence algorithms work on grayscale values, because it is less expensive to compute the weights, as we can compare one channel instead of three, for every pixel. Even algorithms that usually accept color values, simply calculate the matching cost as the mean of the three color channels. To some extent, using grayscale images can even yield more robust matching, as the mean value change less due to lighting conditions and so on [20]. The conversion to grayscale is done by the OpenCV function $cvtColor$, with the processing time listed in table 4.2.

Flipping

As suggested by Kjetil Endal [28], OpenCVs stereo correspondence algorithms do not return any disparity map for the right camera, only the left one. To have OpenCV calculate a disparity map for the right camera as well, we have several options. We can swap the left and right image, and use a negative disparity, but this is not allowed in all the stereo correspondence implementations in OpenCV. A more elegant solution, is to flip the two images around the vertical axis, and pass the flipped right image as the left, and the flipped left image as the right. Thus, the flipping operation becomes part of the preprocessing for generating a disparity map for the right camera. This is done by the function $flip$, and is done well within real-time, as seen in table 4.2. The flip operation is seen in figure 4.6, which shows two images being flipped, with two features marked by red points. We see that the flip operation does not change the disparity, which is 10, both before and after the flip operation.

Operation	**Min**	**Max**	**Mean**
flip	1.2	2.2	1.2
remap (nearest)	9.3	14.0	9.6
remap (linear)	16.2	18.5	16.5
cvtColor	2.7	4.1	2.8
warpPerspective	47.3	50.1	48.9

Table 4.2: Preprocessing operations executed on both left and right image, averages over 100 frames, measured in milliseconds on one core

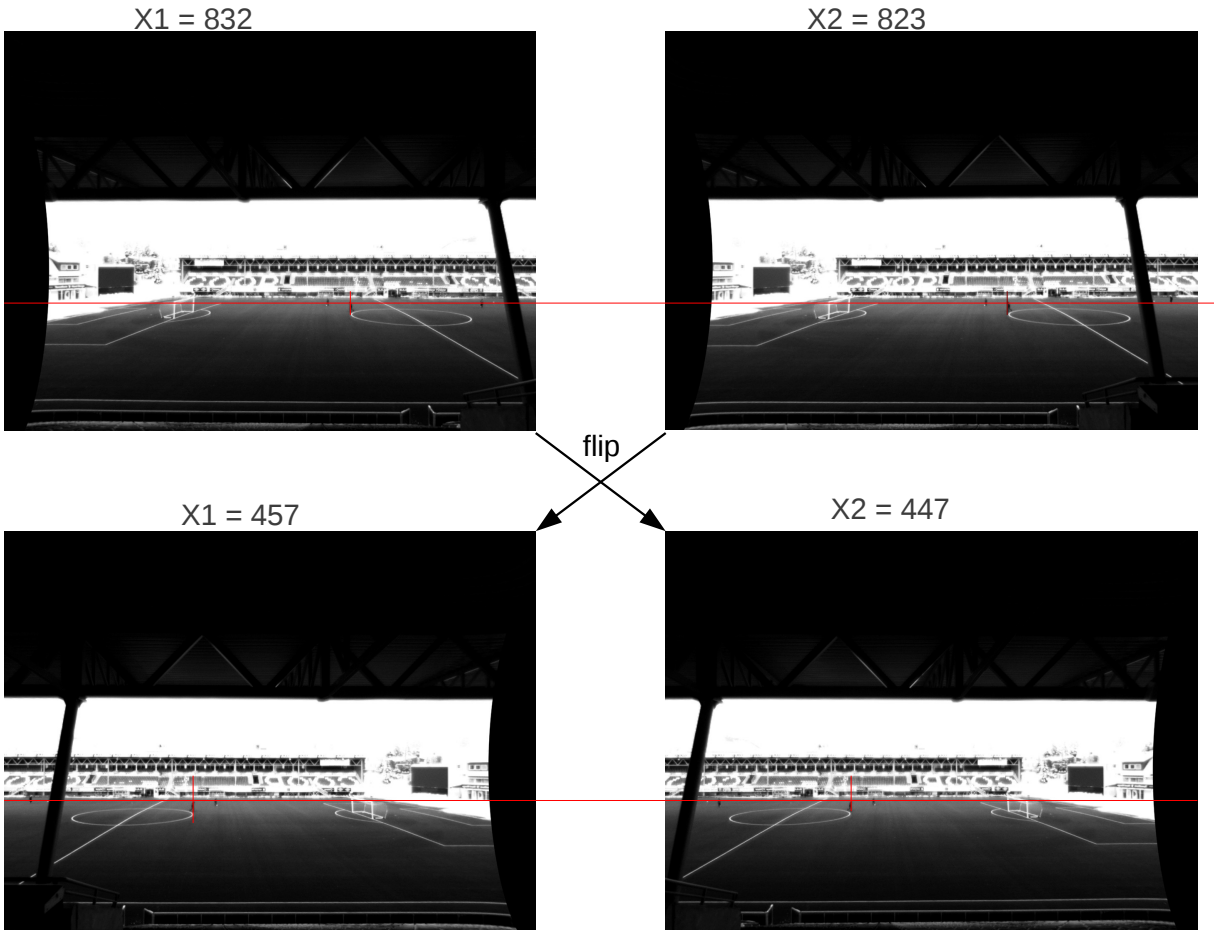


Figure 4.6: Flipping the input images to calculate the disparity for the right image

4.6.2 Disparity Calculation

The data sets available for download with ground truth, are usually already rectified. We will apply the BM and SGBM algorithm at our data set calibrated at our lab, for full control of the cameras and calibration, and a data set taken at Alfheim stadium to test the accuracy of the algorithms in outdoor lighting conditions, and long distances.

Block matching

We run the BM algorithm on our data sets by using the StereoBM class in OpenCV. The BM is a local matching algorithm, taking only into account the matching costs in the block it is working on, and uses a sliding window, with the matching cost being the sum of absolute differences (SAD) of the pixels inside the window, and then match it to the window-costs of the other image.

When initializing an object of the StereoBM class, we must pass some parameters that fine tunes the algorithm. Those are:

preset A flag that can be set to either CV_STEREO_BM_BASIC,CV_STEREO_BM_FISH_EYE or CV_STEREO_BM_NARROW. They are supposed to specify the whole set of pa-

rameters for the algorithm based on the type of lens you are using. We set ours to FISH_EYE_PRESET, since we are using wide angle lenses. When downloading the OpenCV source code and examining the code however, it seems this is not yet implemented properly and the choice for the preset does not matter as of OpenCV version 2.4.2 .

ndisparities The max disparity value. The minimum value is 0 by default, so the search range will be from 0 to ndisparities. As seen in section 4.4.3, the max disparity depends on the baseline and the distance the objects in the scene have to the baseline. In an indoor environment such as our lab, and a camera setup with a baseline of a few centimeters, a value of 80 was enough.

SADWindowSize The SADWindowSize×SadWindowSize size of blocks matched by the algorithm. It should be an odd number, since the center of the block is centered at the current pixel. A larger window-size increases the chances of finding the correct correspondences. Since the sum of absolute differences are computed over a larger area, then the chance are larger that some pixel values with unique values will be inside the window. The downside of a large window size, is that a single disparity is assigned to every block. Thus, large blocks introduce greater errors when they are centered on object edges of different depth, where half of the block will be within the correct object, and half will be outside of it, giving the areas outside wrong disparity. The optimal block size depends largely on the size of the objects in the scene, where small objects should have corresponding small window sizes.

Figure 4.7, shows the rectified images taken from our lab. Figure 4.8a, shows the BM algorithm with SADWindowSize = 7. Its performance is listed in table 4.3. The BM algorithm manages to find correspondences for areas with much unique texture, like the human face, and along object borders. However, it has a difficult time finding correspondences for large objects with a constant color, like the floor, and our clothes. The large, black areas in the disparity map, means the BM algorithm was not able to find any correspondence, and the blocks are not assigned any depth. It could also mean the objects have a distance of "infinity", like the sky, but the floor in our lab is clearly not at infinity. Figure 4.8b, shows the results when using a larger window size, SADWindowSize= 21, where the depth is identified for a larger number of pixels, with less false correspondences on the floor, but at the expense of assigning the areas around the objects with the wrong correspondence values. For instance the areas between our legs are all assigned the same depth value, even though in reality there is a gap into the background in between.

Semi-global block matching

We run the SGBM algorithm on our data sets by using the StereoSGBM class in OpenCV. Unlike BM, SGBM use a global optimization for finding the correct correspondences. The implementation in OpenCV aggregates cost over windows, in the same way as the BM implementation. However, in addition it saves the costs, and applies a global optimization by minimizing an energy function based on the change in disparity:

$$E(D) = \sum_{p} \left(C\left(p, D_p\right) + \sum_{q \in N_p} P_1 T[|D_p - D_q| = 1] + \sum_{q \in N_p} P_2 T[|D_p - D_q| > 1] \right) \quad (4.2)$$

The first term, is the normal matching cost of the blocks compared to the alternatives in the right image, just as in the BM algorithm. The second term adds a small penalty P_1, for every neighboring pixel that have a change of one disparity, permitting small adaptations in disparity, such as slanted objects. The third term adds a larger penalty P_2, for every neighboring pixel that have a larger change of disparity than one, preserving object borders. This function should ideally be applied in 2d, to every pixel. However, to obtain an efficient implementation, they are instead calculated along five to eight paths from the image edges and to the block we are considering, as in the loss function discussed in section 3.2.4.

When initializing an object of the StereoSGBM class, we must pass some parameters that fine tunes the algorithm. It takes the same parameters as the stereo matching class, with the addition of the minimum disparity, and some optional parameters:

SADWindowSize The matched block size, similar to the SADWindowSize described in the block matching function. Unlike the BM algorithm, SGBM allows SADWindowSize to be set to one, meaning pixel-wise matching instead of block matching.

numDisparities This is the max disparity number, as described in the function block matching function.

minDisparity The minimum disparity, as with the maximum disparity, this value depends on the baseline and distance to the objects in the scene. A value of zero is a safe choice, allowing objects at the distance of infinity.

Among optional parameters, we have P_1 and P_2, corresponding to the weights in equation 4.2, giving weights for small and larger disparity changes, respectively. *fullDP*, is a flag being set, if we want to compute equation 4.2 for eight directions instead of just five, increasing computation time and memory usage, turned off by default. A few other parameters controls post-processing and preprocessing, such as excluding small regions with the same disparity, effectively regarding it as noise. We chose to use the default settings, as they gave the best visual result, and the players in our Alfheim data set are very small, sometimes only a few pixels wide, so it is difficult to exclude small regions without affecting the players.

Figure 4.9, shows the results of SGBM in our lab data set, and table 4.3 show its performance.

4.6.3 Post-Processing

After the disparity maps have been calculated, there are a few steps needed in order to calculate a disparity map. An optional step, which we will cover in chapter 5, is filling in gaps in the disparity map where there are no assigned value, for instance with an estimate based on neighboring disparity values. To align the disparity map with the camera pixels, we must remap the disparity map back to the original camera coordinate space, which can be done by applying the same remap function, but using the inverse of the rectification mapping. We saw in section 4.4.3, that the disparity maps are inversely proportional to the depth maps, and that a depth maps can be computed by applying equation 4.1.

4.7 Comparison

Comparing the SGBM and BM algorithms in figure 4.8, and figure 4.9, we see that SGBM is able to match, and assign a depth value to more pixels than BM. However, SGBM is more prone to noise in areas with a large constant value without distinct texture, such as the white floor in our lab.

Operation	Min	Max	Mean	Average FPS
SGBM	336.6	361.1	338.6	2.9
BM	65.1	73.6	65.7	15.2

Table 4.3: Time measured in milliseconds for stereo matching algorithms applied at our 1280×960 pixels data set, using a window size of 11 and a disparity range of 80. Time measured by a sample of 100 frames.

The times spent in milliseconds for the SGBM, and BM algorithms are listed in table 4.3. While the BM algorithm is faster, it produces less accurate results as seen from both table 4.1, and our test data from the lab. Both algorithms are capable of processing several frames per second, with BM 15.2FPS and SGBM 2.9FPS.

4.8 Importance of Rectification

As we have described in section 4.4, the calibration of the system is important for the correctness of the depth map, as the stereo correspondence algorithms limit their search to the epipolar lines, which the rectification should map to the scanlines of the image. If the rectification mapping is wrong, then pixels will not get matched correctly, because the corresponding pixels are mapped to a different scanline. A wrong rectification mapping might also skew, or scale the images, so that the disparity becomes incorrect.

Figure 4.10, shows how the lab data set looks using the homography rectification based on automatic detection of corresponding points, using SIFT, as described in section 3.2.5. We see that rectification have a large impact on the accuracy of the stereo correspondence algorithms, and we will be using chessboard calibration for our rectifications, as the automatic homography based method, are not accurate enough for good quality depth maps, in our current implementation.

4.9 Summary

An overview of some of the more common stereo correspondence algorithms have been presented, with the BM and SGBM algorithms being our main focus, as they have potential to go in real-time. We have presented a list of preprocessing steps needed for the stereo correspondence, among them are converting the images to grayscale, rectification, and optionally, flipping and swapping the left and right image, to calculate the disparity map for the right camera.

We have seen how an incorrectly rectified image greatly affects the accuracy of the stereo correspondence algorithms, with our rectification based on the chessboard being the most reliable.

We have seen how to calculate the disparity for an entire image, using the stereo correspondence algorithms SGBM and BM. We have compared the results of some of the more common stereo correspondence algorithms from both previous works, and using our own rectifications and data sets, concluding that both BM and SGBM have potential to run real-time, with further optimizations. Assuming that the depth maps will be used for the free-view application, we can assume that only the depth of the players is needed. Thus, applying the stereo correspondence algorithms to a real soccer scenario, remains. In the next chapter we will look into testing the system at a soccer scenario, and investigate optimizations.

(a) Left rectified image

(b) Right rectified image

Figure 4.7: Rectified images from our lab

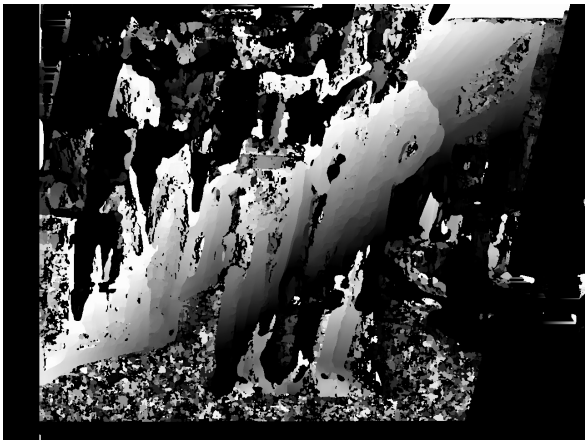
(a) Block matching with SADWindowSize = 7

(b) Block matching with SADWindowSize = 21

Figure 4.8: Block matching results with different window size

Figure 4.9: SGBM with SADWindowSize=21

(a) Correctly rectified



(b) Incorrectly rectified

Figure 4.10: The difference between a correct rectification 4.10a and a wrong rectification 4.10b

Chapter 5

Optimization of the Depth Estimation

In chapter 4, we investigated BM and SGBM, two relatively fast stereo matching algorithms with respect to getting a good quality depth map in real-time. Both algorithms spent more than 33.3ms on the entire image, but they have potential to be real-time. Also, we observe that the SGBM might introduce noise in textureless areas, and that both algorithms fail to assign a disparity value to absolutely all pixels.

The algorithms were tested in our lab to get an optimal lighting and calibration environment. To improve the quality and performance of the disparity, we will also apply background subtraction [49]. To check how the algorithms perform in practice when applied to soccer games, we deployed a test system at Alfheim Stadium.

5.1 Camera Setup

The data set we captured in the lab only had a few meters of distance to the camera, and our baseline was just a couple of centimeters. At Alfheim stadium, the distances can be more than 100 meters, and as we saw from equation 4.1, the further away from the baseline, the lower the disparity will be. It is therefore important to increase the length of the baseline accordingly, to avoid that the disparity will be so low that it is not detectable. For our test data at Alfheim stadium we used a baseline of about 0.7m.

We used the same kind of lens that the current Bagadus pipeline uses, the Kowa 3.5mm wide angel lenses. We could not use the cameras already mounted for the Bagadus system, as they are both difficult to calibrate once mounted on the platform, and their angles with little overlap makes them unfit for stereo vision. Also, because we had limited time to capture the dataset, and limited equipment for mounting cameras at floating platforms etc., we did not spend much time to find optimal mounting points for the cameras. Instead we mounted the cameras at the back of the stands, purely for testing purposes. Thus, our cameras are not in a ideal position, meaning we get a lower resolution in the area of interest, such as the players, compared to how many pixels a player is represented with using the existing camera setup.

For calibration, we used the chessboard method, described earlier in section 3.2.6. However, since we mounted the cameras outside, we also had to perform the chess calibration outside. We used a chessboard pattern printed on a paper, as we did in the lab. However, we discovered several problems with this approach. The biggest challenge was that the paper curled up due to the changes in temperature, wind and most importantly humidity. This makes the chessboard pattern calibration less accurate. To reduce the effect of the wind and humidity, we wrapped the

paper in a clear envelope. However, the humidity still had some effect, and the surface of the clear envelope had a reflective property. As you can see from figure 5.1, we have some curling

Figure 5.1: Chessboard pattern on printed paper wrapped in a clear envelope. Some reflection and curling of the paper introduce problems for the automatic calibration

of the paper, some reflections due to the clear envelope, and the chessboard pattern lacks a wide white border, which is required by the algorithm. The lacking white border was solved by adding it manually in an image editor afterwards, but the reflections and curling could not be undone easily. The chessboard calibration algorithms only recognized 15 out of the 40 image pairs we captured for stereo calibration.

In future experiments, we would suggest either printing the chessboard pattern on a large white laminate surface, with at least a few centimeters of white border on all sides of the chessboard pattern. However, a laminate surface might also have some reflective properties, so the best would be an actual white chessboard in a non-reflective material. Non-reflective laminate might also work, but it is important to use materials that does not distort the colors of the chessboard pattern. Since the baseline is so large, we need to hold the chessboard at a further distance to cover more of the overlapping areas in the two cameras, thus a larger chessboard pattern will increase the accuracy, so we also recommend at least printing in the A3 size.

An example frame pair of the data set captured at Alfheim is pictured in figure 5.2, where the cameras heading is orthogonal to the length of pitch, with the cameras having a baseline of roughly 0.7m. In other words, these are an example our input images from our stereo camera setup. As we observe, the camera depicted in figure 5.2b, is located a bit further to the right, as the vertical support beam to the right in the images, is located further to the left, compared to

the left camera image in figure 5.2a. We also observe that the support beams disparity is larger than the disparity of the players and white lines in the pitch, as the support beam is closer to the camera.

(a) Left input image after rectification

(b) Right input image after rectification

Figure 5.2: The Alfheim data set

5.2 Background Subtraction

Background subtraction is a technique in computer vision used to extract the foreground of an image from its background, where the foreground typically consists of objects of interest, which should be used for further processing. Examples of foregrounds might be humans, cars, animals, etc. The foreground is extracted from the background using a mask of the same dimensions as the input image, where for instance a non-zero value represents foreground and a zero represents background. Some background subtraction algorithms such as Zivkovic [55], also detects shadows, and assign a special value for them.

There are several ways to extract the foreground, some of the more simplistic methods include methods like frame differencing, finding the difference between the values of a given pixel between two frames. If the absolute difference in pixel value is above a threshold, it is considered to be foreground. However, this simple method has several shortcomings. It only works if the background is entirely static, and the foreground is moving constantly.

We will use the work of [49], which has investigated how the Zivkovic background subtraction model [55] can be applied to the Bagadus system, and how to integrate it with the ZXY player tracking system to reduce noise, by only selecting small regions around the players coordinates and, performing the background subtraction on those. It is a more complex background subtraction algorithm, based on the Gaussian Mixture Model [55], however, it runs in real-time and is capable of detecting shadows as well as foreground. The algorithm learns over time, thus it needs to process a few frames before it correctly estimates what is background and what is foreground. Figure 5.3, shows the Zivkovic background subtraction applied at our data set captured at Alfheim stadium.

5.3 Improving the Quality

When applying the SGBM algorithm to figure 5.2, we get much noise in the image, as seen in figure 5.4. We can also see that, not all the blocks are given a disparity value, because the matching algorithm can not find any matching block. Using background subtraction, we can obtain a mask containing only the player-pixels, and extract the depth for the players only. As discussed in section 2.9, we only need the depth of the players, as a more simple 3D model can represent the static planes of the stadium. By extracting only the players, the noise is filtered out, as the noise is typically located in large almost textureless areas, such as the green grass. When extracting the depth values from the masks, we also check if the depth values. If the depth is zero, meaning it has not been assigned any value, we perform a simple gap filling technique, by keeping track of the last valid depth within the mask, and replacing the zero values with the last known depth. See algorithm 3, for the details of the infilling and extraction of pixels. Figure 5.5, shows the disparity map of figure 5.4, after extracting the foreground and performing infilling.

When running algorithm 3, on 100 images, we measured the mean, maximum and minimum times to be 3ms, which far below our real-time constraint of 33.33ms.

(a) Left camera input

(b) Background subtraction results

Figure 5.3: The results of applying the Zivkovic background subtraction

5.4 Towards Real-Time

We have seen in chapter 4, how the stereo correspondence algorithms do not run in real-time. When looking at the stereo correspondence algorithms in OpenCV, we see that they are already optimized for single instruction, multiple data (SIMD) instructions. As discussed in section 1.3, to further optimize the algorithms, we would need to investigate parallelizing them on the GPU, but this is out of scope for this thesis.

5.4.1 The Idea

To increase the performance of the stereo matching, we will instead try to limit the search space for the stereo matching algorithms. Since we are only interested in the depth of the players, we

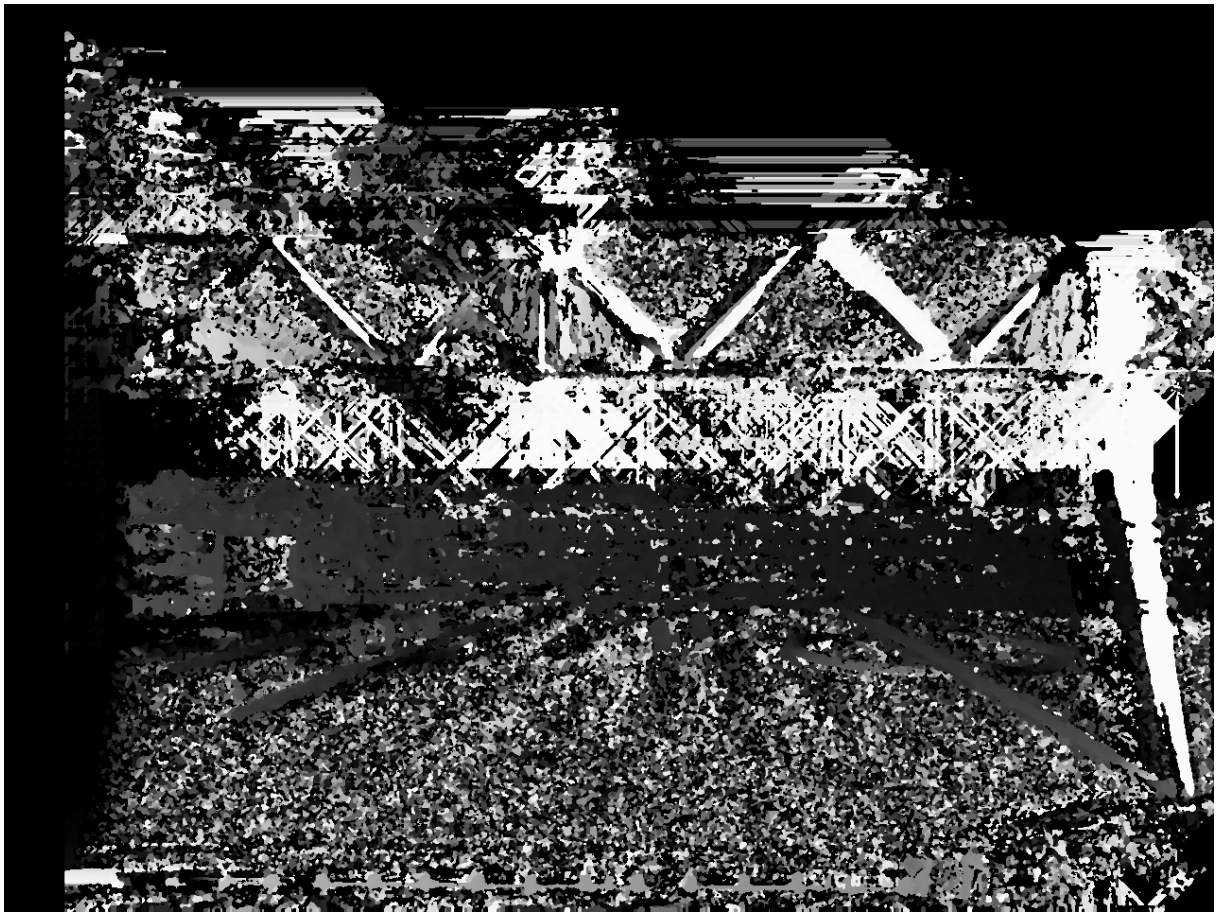


Figure 5.4: SGBM applied at Alfheim stadium.

Algorithm 3 Extracting foreground and infilling

```
function EXTRACTFOREGROUND(foreground, dispMap, imgWidth, imgHeight)
    lastDisp ← 1
    for y = 0 → imgHeight do
        for x = 0 → imgWidth do
            if foreground[x][y] = 0 or dispMap[x][y] = shadow then
                dispMap[x][y] ← 0
            else if dispMap[x][y] = 0 then
                dispMap[x][y] ← lastDisp
            else
                lastDisp ← dispMap[x][y]
            end if
        end for
    end for
end function
```

can only focus on those. We can also return the depth of the ball, but only if it is located near a player. If we want to make sure we always return the depth of the ball, we would need a way to track the ball, such as using SURF and movement estimation.

Figure 5.5: The disparity map after infilling and removing the background

There are two ways to limit the search space for the stereo correspondence functions. One is to modify the algorithms, to make them accept a foreground mask as an input parameter, thus only considering the foreground when searching for matches. The other is to split a larger image into smaller regions of interest, and only consider these.

The former is the more efficient, in the sense that it only considers the foreground masks. However, the SGBM algorithm calculates the depth by aggregating paths through the image, making it difficult to implement such a feature. Also, the entire image still has to be considered, so using this method means we can not parallelize the operation.

The latter is easier to implement, and allow us to parallelize the disparity estimation by looking at each region of interest separately, where such a region can be small sub-images extracted and processed separately. The problems with this approach are finding the regions of interest. Also, the disparity estimation is often noisy at the end of the image, as there are less paths that pass through the border-pixels in the case of the SGBM algorithm, and that the leftmost pixels in the right image will not have any corresponding pixels due to the disparity.

Although the two methods are not mutually exclusive, we will focus on the latter.

5.4.2 Finding the Region of Interest

To find the region of interest, we use the ZXY sensor data from the players. To calculate the homography used for the $ZXY \rightarrow cameraCoordinate$ mapping, we use the approach explained in chapter 3. When we have the homography, the mapping is done by a simple matrix multiplication, as described in section 2.5. Because the ZXY system mounted at Alfheim might have up to 1m of error margin, we set the mapped ZXY coordinates to the center of the bounding box, and set the box width and height to 220 pixels. Further, because the players close to the camera appear larger, while players further away appear smaller, we also scale the size of the bounding box depending on the distance between the player and the camera, which can also be computed given the ZXY coordinate and the location of the camera. Typically the cameras heading will be perpendicular to one of the pitch lines, and the depth scaling will be:

$$scale = playerPosition_y \times shrinkFactor$$

where the width and height of the bounding box is multiplied with $scale$. The shrink_factor is a constant, and depends on the position of the camera. The bounding boxes depicted in this thesis used a shrinkFactor of 0.0021.

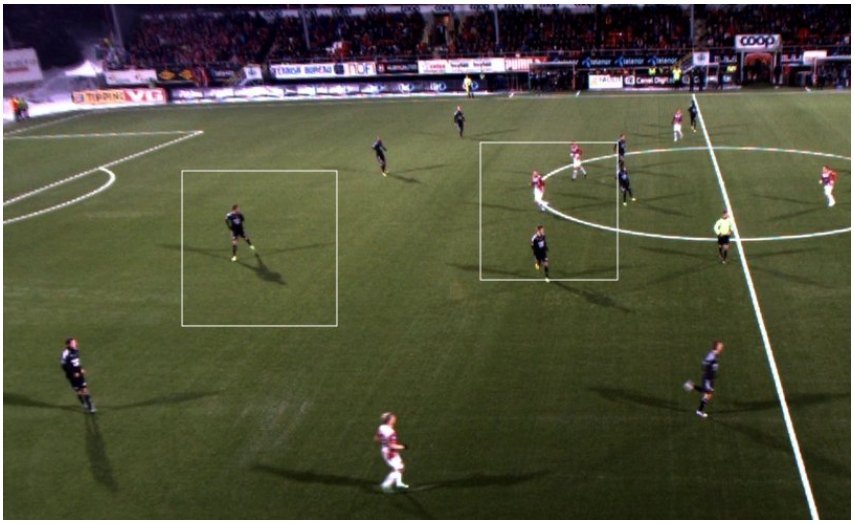


Figure 5.6: Bounding boxes illustrated for two players (not all). Cropped for display purposes

Note how the bounding boxes in figure 5.6 cuts through the neighboring players. To decrease the chances of having a bounding box cut through a player if two players stand near each other, which might result in wrong disparity values along the edges, we implemented an optional algorithm to combine the bounding boxes of players if they get too close to each other, as seen in algorithm 4.

Because the bounding box sizes are dynamic, both due to the scaling, and possible multiple bounding boxes combining into one, the $threshold$ is in reality set to $\frac{1}{5}$ of the box-width, allowing some overlap before they merge:

$$threshold = \max(boxA.width, boxB.width) - min(boxA.width, boxB.width)/5)$$

Because combining two bounding boxes might end up increasing the bounding box dimensions into a third bounding box, we run the *mergeBoxes* function in a loop until it returns false. Figure 5.7 shows all players being tracked, and a bounding box drawn around each cluster of players.

Algorithm 4 Combining bounding boxes

function MERGEBOXES(boxes, threshold)
  $foundBox \leftarrow false$
  for all $boxA \in boxes$ **do**
    for all $boxB \in boxes$ **do**
      if $boxA = boxB$ **then continue**
      end if
      if $|boxA_x - boxB_x| < threshold$ **and** $|boxA_y - boxB_y| < threshold$ **then**
        $boxA_{width} \leftarrow \max(boxA_{width} + boxA_x, boxB_{width} + boxB_x)$
        $boxA_{height} \leftarrow \max(boxA_{height} + boxA_y, boxB_{height} + boxB_y)$
        $boxA_x \leftarrow min(boxA_x, boxB_x)$
        $boxA_y \leftarrow min(boxA_y, boxB_y$
        $boxA_{width} \leftarrow boxA_{width} - boxA_x$
        $boxA_{height} \leftarrow boxA_{height} - boxA_y$
        $boxes \leftarrow boxes - boxB$
        $foundBox \leftarrow true$
      end if
    end for
  end for
  return $foundBox$
end function

Figure 5.7: Boxes combined

5.4.3 Splitting Bounding Boxes

The point of using bounding boxes is to reduce the large single image into several smaller images with less complexity, which can then be calculated in parallel. How large a bounding box can be before its corresponding disparity map is calculated depends on the hardware of the

system, and the algorithms used. In section 5.4.4, we will test the box sizes against our current hardware.

Figure 5.8: Splitting bounding boxes

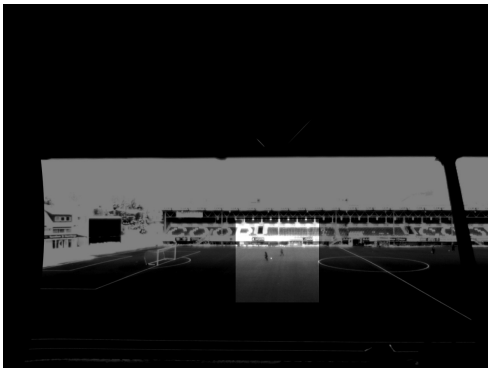
If the players are roughly equally distributed along the pitch, the bounding boxes might combine into a single large one, and we gain little from dividing the image into bounding box clusters. However, this happens quite rarely, and in the cases this happens, we solve it by checking if a bounding box is larger than a threshold (which size depends on how large sub-images we are able to process in real-time). If the bounding box is larger than this threshold, we try to split the bounding box, by checking the corresponding foreground mask from the background subtracter. Figure 5.8, shows an example of bounding boxes that are too large, and split into smaller ones.

The splitting operation is performed by looping through all the pixels in a vertical line, starting at the lower middle of the bounding box. If the pixels along the line hits more than a given threshold of foreground mask pixels (we set it to three pixels, to tolerate some noise), the algorithm moves four pixels to the right and tries the next vertical line. Because this check is expensive, it is only performed on the bounding boxes that are larger than the threshold. When extracting the images from the bounding boxes for separate processing, it is important to copy the exact same rectangle position and dimensions, so that we do not alter the disparity in the sub-image.

5.4.4 Results

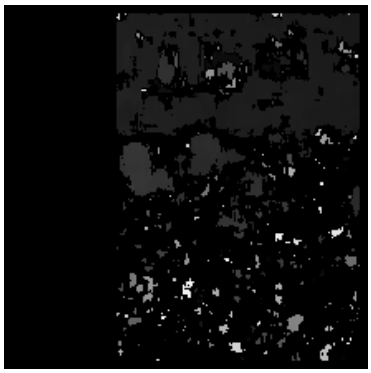
Although we have not implemented a complete depth estimation pipeline which integrates all the individual components, we test them separately. By using a previously recorded game with corresponding ZXY data, we can test the combining and splitting of bounding boxes. Combining and splitting the bounding boxes take a mean time of only 1.2ms, when calculating the bounding boxes for 18 players, seen in figure 5.8.

Figure 5.9, shows the rectified input-images, the sub-images, and the resulting disparity map corresponding to the sub-images. The extracted sub-images are illustrated as the bright



(a) Left input image

(b) Right input image



(c) Resulting disparity map.

Figure 5.9: Calculating the disparity for a sub-image. The disparity map of figure 5.9c is scaled for display purposes.

Operation on image	**Min**	**Max**	**Mean**
SGBM on 180×984	28.3	28.7	28.5
SGBM on 442×331	31.7	35.6	32.0
BM on 960×720	29.8	32.2	30.2

Table 5.1: Speed of stereo correspondence algorithms on sub-images of different size, with time measured in milliseconds

boxes in the middle of the input-images (5.9a and 5.9b. Each such pair of boxes can then be processed in parallel. The resulting disparity map, before infilling and background subtraction is performed, is shown in figure 5.9c. We see that the disparity map correctly identifies the two running players, and the stands in the background, with the objects closer to the camera having a slightly higher disparity than those further back.

With the images split into sub-images, we measure the time it takes to compute the depth map for sub-images of different sizes. Table 5.1, shows the processing time of the stereo correspondence algorithms, applied to different sizes of sub-images, which all performs below our real-time limit. We notice that with our current hardware and stereo correspondence algorithms, we can calculate the disparity for sub-images with sizes up to 442×331 pixels using SGBM, and sizes up to 960×720, using the less accurate BM.

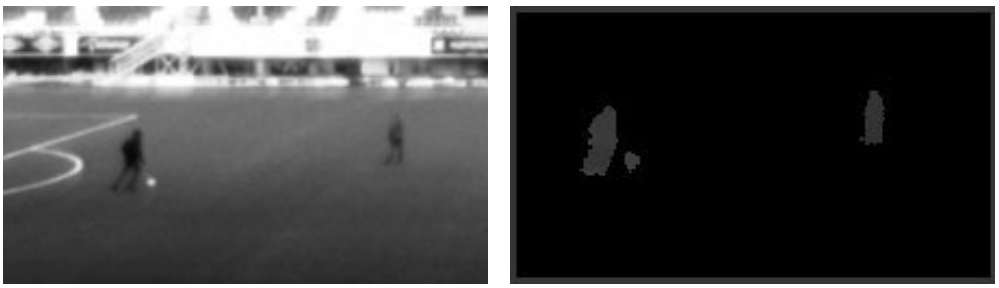
As we see, the computation of the bounding boxes is within our real-time constraint, and can be parallelized and executed in a pipeline. When the computation is finished, the results can

be copied back onto an initially zero filled disparity map, at the same location as from where it was taken in the original rectified input image.

5.5 Correctness

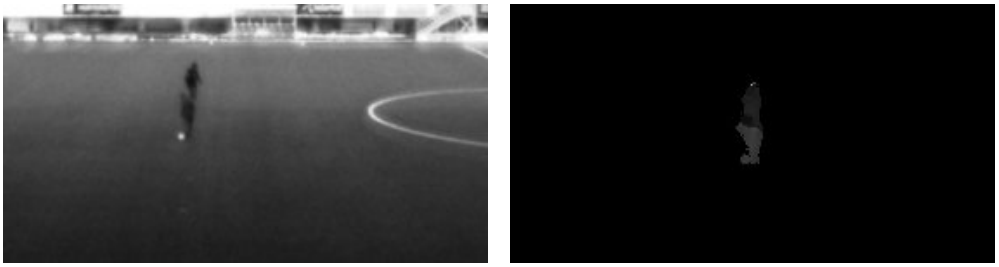
We can see from the image in figure 5.5, that a few noisy pixels are present around the edge of one of the players. This is because the foreground mask from the background subtracter in this case also covered a small part of the pitch, where the stereo vision algorithms fail to find corresponding pairs. However, [49] have demonstrated that it is possible to get a very accurate background subtraction at Alfheim stadium, in his thesis, so the accuracy of the background subtraction can possibly be increased, by using more fine tuned parameters for the background subtracter. Noise can also be filtered out by a smoothness function, or simply by rejecting very high disparity values. But overall there is not much noise in our results.

Figure 5.10, shows two different frames and their disparity estimation, converted to a grayscale image for display purposes. The left player in image 5.10a, has a grayscale value of 58, while the right player has a grayscale value of 50, which is consistent with the leftmost player being closer to the camera. In figure 5.11a, we see that the closest player is given a higher value, but the disparity is not accurate, as the change of disparity happens approximately on the closest players head. However, this happens to all pixels in the area, and is most likely a result of the inaccurate calibration we described in section 5.1.



(a) Input (b) Disparity correctly estimated

Figure 5.10: Disparity comparison of the Alfeim data set



(a) Input (b) Inaccurate disparities

Figure 5.11: Disparity comparison of the Alfeim data set

5.6 Summary

In this chapter, we have applied the stereo matching algorithms from chapter 4, to a data set captured at Alfheim stadium. However, while setting up and calibrating the camera setup at Alfheim stadium, we noticed some flaws with our current calibration equipment, that affected the end result, as only 15 out of 40 calibration images was recognized by the *findChessBoard* algorithm. We thus discussed ways to improve the chessboard pattern calibration, in an outdoor environment, by using laminate.

Since the system is intended for a soccer scenario, we have applied a background subtraction algorithm to the data set, used to filter out the noise in the pitch, and only extract the depth values from the players, as well as infilling depth values where the algorithm fails to find corresponding blocks. Also, we showed that the stereo vision algorithms managed to distinct the depth of two players with a few meters distance, despite the long distance they had to the stereo cameras. For more accuracy, better cameras or lenses are required to increase the number of pixels per square meter.

However, the stereo vision algorithms do not run in real-time. Thus, the ZXY tracking system was used to divide the image into several sub-images, by extracting sub-images from bounding boxes centered around the ZXY-coordinates of the players. The sub-images can be then processed independently by the stereo correspondence algorithms. By combining the bounding boxes that are too close to each other, we avoid sub-images cutting through another player. Using the background subtraction foreground mask, we can also split the bounding boxes that are too large to run in real-time, into smaller bounding boxes. We have also showed that the disparity sub-images can be computed in real-time. Thus, the computation of the sub-images can be done in parallel.

Chapter 6

Conclusion

In this chapter, we summarize our work, present our contributions, and finally look at future improvements that can be done.

6.1 Summary

In this thesis, we have presented an overview of the old Bagadus system, and presented a new version of the system, which runs in real-time on a single machine with commodity hardware, making use of both the GPU and CPU, as described in chapter 1. The Bagadus system lacks a free-view system, which depends on a depth map corresponding to every video frame, to be able to select the correct pixels to render.

The depth maps in turn depends on the cameras to be perfectly aligned, which is close to impossible to achieve. Therefore, it is important to calibrate the cameras, perform undistortion and rectification on the input images, before the depth maps are computed. We investigated using automatic detection of lines to automate the calibration process, and to be able to use the information a line represents, even in the cases when there are no visible line intersections (corners) along that line. We also created a point based approach, where the user must manually click on the points he wants to mark.

For the rectification, we also implemented a chessboard based calibration, including an easy to use application for taking stereo images. We found that only the chessboard based calibration produces a rectification mapping accurate enough for quality disparity estimation.

The calibration of the existing Bagadus system is improved, by applying the methods used for the calibration in the depth estimation pipeline, to the similar problems of the Bagadus system. This includes finding the intrinsics needed for removing geometrical distortions, finding the homography needed to map the ZXY coordinate space to the cameras coordinate spaces, and finding the matrices needed for warping the images from all cameras onto a common plane for panorama stitching.

We also explain the theory of stereo correspondence and depth estimation. Using the mappings calculated during the camera calibration, we can rectify the images to make the epipolar lines horizontal along the image scanlines, as desired by the stereo correspondence algorithms. We compare the performance of some of the more common stereo correspondence algorithms from earlier works, and test SGBM and BM, the two most promising algorithms, on our own data sets to check that the calibration is sufficient. All the independent steps such as rectification, converting to grayscale, etc., are done in real-time.

We have also tested our calibration system and stereo correspondence algorithms on a data set captured at Alfheim stadium, to investigate how our system performs over long distances. Assuming that the depth estimation will be used for free-view purposes, we can focus on obtaining the players, as the rest of the pitch can be modeled as simple planes in a 3D model. The system manage to identify the depth of two players standing a few meters apart, but to get a disparity map with higher depth resolution, a higher resolution camera or a lens giving more pixels per square meter is required.

By applying background subtraction, we can identify the players and extract only their depth, effectively filtering out the noisy parts of the pitch, as well as infilling depth values in the foreground masks to make sure there are no gaps in the disparity map.

Using the ZXY system, we can see how to create bounding boxes around each player, and combine them to avoid the bounding boxes from cutting through a nearby player. By using the foreground mask from the background subtracter, we can split bounding boxes that may be too large for the disparity calculation to perform real-time. We test the BM and SGBM algorithms on the sub-images extracted from the bounding boxes to show how large individual boxes we can be without the processing time exceeding our real-time constraint. Thus, by splitting the image into bounding boxes into multiple such bounding boxes which can be processed in parallel, the disparity calculation is performed within our real-time constraint.

6.2 Main Contributions

As discussed in section 1.2, we wanted to investigate real-time depth map generation for a soccer scenario, using less computationally expensive algorithms, than [44] used for their system. Also, we wanted to improve the initialization of the current Bagadus system, by applying the calibration steps from the depth map pipeline. We approached this problem, by implementing a prototype, and measuring its performance. To make sure the system works in a real soccer scenario over longer distances, we tested the prototype at Alfheim stadium.

The Bagadus system is lacking a free-view functionality, thus, a short overview of the free-view application is presented. For the free-view to be able to render the scene correctly, it needs the frames of the nearby real cameras, with corresponding depth maps. Therefore, a pipeline for depth estimation is presented, and its individual components are implemented and measured, with every single component running in real-time.

Because the depth estimation will be applied to a soccer scenario, it is tested both in the lab, and on a soccer stadium. When testing the system at the soccer stadium, we prove the systems capability to estimate the depth of players in the pitch, with players closer to the camera having a higher disparity than players further back. Because the disparity decreases over distances, we compensate for this by increasing the distance between the cameras. By splitting the original image into sub-images centered around the players, we show that the individual depth maps is calculated at 32ms, within our real-time constraint, for sub-images of sizes up to roughly 442×331 pixels. We show that the players depth can be estimated with an accuracy of roughly few meter. To increase the accuracy of the depth estimation, a lens that gives more pixels per square meter, or a higher resolution camera, is required.

Many of the tasks performed when calibrating the depth estimation pipeline, such as homography the computations, are very similar to what we need for calibrating the existing Bagadus system. Therefore, we implemented an application for calibrating the Bagadus system, includ-

ing calculating the intrinsics for correcting geometrical distortion, the ZXY to camera coordinate system mapping, and the homographies for the panorama stitching.

6.3 Future work

We have taken several data sets with multiple cameras, and the calibration tools were important for setting up the system quickly. The manual calibration approach was robust, but a bit more time consuming. However, for a large scale system with as much as 20 to 50 cameras distributed around the stadium, our manual approach will require a lot of time, thus it can be worth investigating how to improve the automatic line detection perform as well as our manual point based selection.

The automatic line based approach for estimating the homography can perform better, by trying to classify the detected lines into vertical and horizontal lines as well. The inaccuracy introduced because our model have lines passing close to the origin, might be corrected for by translating the model and detected lines, so that the lines do not pass through the origin anymore. Should the automatic detection of lines include too many false positives, the color check algorithm can be extended to checking the colors for all parts of the line instead of just checking the endpoints.

The calibration tools include a point and click GUI to identify matching points in between images, to calculate the homographies needed for warping the images onto a common plane for stiching. However, the stitch seam must be done in an area where the two images overlap, but our application does not calculate the overlap area. At the moment, the overlap area is found by aligning the images in an image editor and manually calculating the overlap areas. It could be made easier by asking the user to manually select the overlapping region by point and click in a similar fashion.

Also, we demonstrated that the SURF based rectification can be unreliable. To improve the automatic homography based rectification based on SURF, a more robust matching method must be applied, and regions of interest can be used to select which areas to search within.

When processing our data set captured at Alfheim, we saw that the calibration images of the chessboard was not sufficient, as only 15 out of 40 images were recognized by the calibration algorithm. A better calibration is needed for a quality depth map for our data set captured at Alfheim stadium. To obtain a reliable chessboard calibration in an outdoor environment, a black and white chessboard in a non-reflective material should be used, and it is vital that there is at least a few centimeters thick white border around the chessboard pattern, or else OpenCVs *findChessboardCorners* can not locate the chessboard. To get a disparity map with higher resolution, a better lens or a higher resolution camera is needed.

The disparity estimation it self can be made real-time using either BM, or SGBM. To offload the CPU and get more efficient disparity estimation, the SGBM and BM methods can be implemented on the GPU, allowing us to tolerate larger bounding boxes (see section 5.4), before attempting to split them. Also, the stereo correspondence algorithms could be modified to only calculate and check the pixels marked as foreground to speed up the estimation. Since the disparity error at the boundaries of the bounding boxes are usually just a few pixels large, and could be smoothed out as a post-processing step. If the accuracy needed by the free-viewer is not too large, the disparity calculation could be performed per bounding box without combining them.

We have implemented and tested the individual parts of the depth estimation pipeline, but the parts are not fully integrated. The individual steps must be pipelined and parallelized using threads, using the same pipelining approach as the new Bagadus video pipeline. In particular, every bounding box should run in its own thread, where every thread share a common, initially empty disparity map, where every thread copies its bounding box to the common disparity map, at the same location as the original bounding box was. The free-view system can then be implemented using the depth maps generated by our pipeline.

Appendix A

Accessing the Source Code

The source code for the Bagadus system, including what is described in this thesis, can be found at `https://bitbucket.org/mpg_code/bagadus`. To retrieve the code, run *git clone git@bitbucket.org:mpg_code/bagadus.git*.

Bibliography

[1] Basler. http://www.baslerweb.com/products/ace.html. Accessed: 05/03/2013.

[2] Camargus - premium stadium video technology inrastructure. http://www.camargus.com. Accessed: 02/03/2013.

[3] Cubic frame stucture and floor depth map. http://en.wikipedia.org/wiki/File:Cubic_Frame_Stucture_and_Floor_Depth_Map.jpg. Accessed: 09/04/2013.

[4] Cubic structure. http://en.wikipedia.org/wiki/File:Cubic_Structure.jpg. Accessed: 09/04/2013.

[5] iad - information access disruptions. http://iad.ndlab.net/. Accessed: 16/04/2013.

[6] Interplay sports. http://www.interplay-sports.com. Accessed: 02/03/2013.

[7] Opencv. http://opencv.org/. Accessed: 09/03/2013.

[8] Prozone. http://www.prozonesports.com. Accessed: 28/02/2013.

[9] Stats technology. http://www.sportvu.com/football.asp. Accessed: 01/03/2013.

[10] Verdione. http://www.verdione.org/. The official website at the time of delivery is down. Google cache: http://webcache.googleusercontent.com/search?q=cache:http://verdione.org/.

[11] x264 video encoder. http://www.videolan.org/developers/x264.html. Accessed: 09/03/2013.

[12] Zxy sport tracking. http://www.zxy.no. Accessed: 28/02/2013.

[13] Stereo vision using the opencv library. http://tjpstereovision.googlecode.com/hg-history/551f9b6e2e9549337e7c26b4bac6a9a69a6c509c/doc/verslag.pdf, June 2010.

[14] F. Albregtsen and G. Skagestein. *Digital representasjon: av tekster, tall, former, lyd, bilder og video*. Unipub, 2007.

[15] Luis Alvarez. Private email correspondence, 2013.

[16] Luis Alvarez, Luis Gomez, Pedro Henriquez, and Luis Mazorra. Automatic camera pose recognition in planar view scenarios. In Luis Alvarez, Marta Mejail, Luis Gomez, and Julio Jacobo, editors, *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, volume 7441 of *Lecture Notes in Computer Science*, pages 406–413. Springer Berlin Heidelberg, 2012.

[17] C. Banz, H. Blume, and P. Pirsch. Real-time semi-global matching disparity estimation on the gpu. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 514–521, 2011.

[18] C. Banz, S. Hesselbarth, H. Flatt, H. Blume, and P. Pirsch. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 93–101, 2010.

[19] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008.

[20] Luca Benedetti, Massimiliano Corsini, Paolo Cignoni, Marco Callieri, and Roberto Scopigno. Color to gray conversions in the context of stereo matching algorithms: An analysis and comparison of current methods and an ad-hoc theoretically-motivated technique for image matching. *Mach. Vision Appl.*, 23(2):327–348, March 2012.

[21] S. Birchfield and C. Tomasi. Depth discontinuities by pixel-to-pixel stereo. In *Computer Vision, 1998. Sixth International Conference on*, pages 1073–1080, 1998.

[22] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, PAMI-8(6):679–698, 1986.

[23] chandler. Football pitch metric and imperial. `http://en.wikipedia.org/wiki/File:Football_pitch_metric_and_imperial.svg`. Accessed: 10/03/2013.

[24] Jen-Shiun Chiang, Chih-Hsien Hsia, Hung-Wei Hsu, and Chun-I Li. Stereo vision-based self-localization system for robocup. In *Fuzzy Systems (FUZZ), 2011 IEEE International Conference on*, pages 2763–2770, 2011.

[25] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[26] J. Martin D. Mills, U. Delaware. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905, RFC Editor, June 2010.

[27] Aram Dulyan. Shutter speed in greenwich. `http://en.wikipedia.org/wiki/File:Shutter_speed_in_Greenwich.jpg`. Accessed: 07/03/2013.

[28] Kjetil Endal. A pipeline for high-quality free-viewpoint video. Master's thesis, University of Oslo, Norway, 2011.

[29] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient belief propagation for early vision. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–261–I–268 Vol.1, 2004.

[30] Ian Parberry Fletcher Dunn. *3D Math Primer for Graphics and Game Development*. Wordware, 2002.

[31] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by tempo in berkeley unix 4.3bsd. *Software Engineering, IEEE Transactions on*, 15(7):847 –853, jul 1989.

[32] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David K. C. Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, and Dag Johansen. Bagadus: An integrated system for arena sports analytics - a soccer case study. In *Proceedings of the International Conference on Multimedia Systems (MMSys)*, pages 48–59, Feb/March 2013.

[33] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.

[34] Espen Oldeide Helgedagsrud. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching. Master's thesis, University of Oslo, Norway, 2013.

[35] H. Hirschmuller. Stereo processing by semiglobal matching and mutual information. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(2):328–341, 2008.

[36] Li Hong and G. Chen. Segment-based stereo matching using graph cuts. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–74–I–81 Vol.1, 2004.

[37] Dag Johansen, Magnus Stenhaug, Roger Bruun Asp Hansen, Agnar Christensen, and Per-Mathias Høgmo. Muithu: Smaller footprint, potentially larger imprint. In *Proceedings of the IEEE International Conference on Digital Information Management (ICDIM)*, pages 205–214, August 2012.

[38] Tetsuya Kakuta, Lu Boun Vinh, Rei Kawakami, Takeshi Oishi, and Katsushi Ikeuchi. Detection of moving objects and cast shadows using a spherical vision camera for outdoor mixed reality. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, VRST '08, pages 219–222, New York, NY, USA, 2008. ACM.

[39] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[40] Don Murray and Jim Little. Using real-time stereo vision for mobile robot navigation. In *Autonomous Robots*, page 2000, 2000.

[41] S. Nedevschi, A. Vatavu, F. Oniga, and M. M Meinecke. Forward collision detection using a stereo vision system. In *Intelligent Computer Communication and Processing, 2008. ICCP 2008. 4th International Conference on*, pages 115–122, 2008.

[42] Arne Nordmann. Epipolar geometry. `http://en.wikipedia.org/wiki/File:Epipolar_geometry.svg`. Accessed: 04/04/2013.

[43] Mikkel Næss. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction. Master's thesis, University of Oslo, Norway, 2013.

[44] N. Papadakis, A. Baeza, I. Rius, X. Armangué, A. Bugeau, O. D'Hondt, P. Gargallo, V. Caselles, and S. Sagàs. Virtual camera synthesis for soccer game replays. In *Visual Media Production (CVMP), 2010 Conference on*, pages 97 –106, nov. 2010.

[45] Pbroks13. Pinhole-camera. `http://en.wikipedia.org/wiki/File:Pinhole-camera.svg`. Accessed: 18/03/2013.

[46] Simen Sægrov, Alexander Eichhorn, Jørgen Emerslund, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Bagadus: An integrated system for soccer analysis (demo). In *Proceedings of the International Conference on Distributed Smart Cameras (ICDSC)*, October 2012.

[47] Simen Sægrov. Bagadus: next generation sport analysis and multimedia platform using camera array and sensor networks. Master's thesis, University of Oslo, Norway, 2012.

[48] Tangfei Tao, Ja Choon Koo, and Hyouk-Ryeol Choi. A fast block matching algorthim for stereo correspondence. In *Cybernetics and Intelligent Systems, 2008 IEEE Conference on*, pages 38–41, 2008.

[49] Marius Tennøe. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction. Master's thesis, University of Oslo, Norway, 2013.

[50] Kazunori Umeda, Yuuki Hashimoto, Tatsuya Nakanishi, Kota Irie, and Kenji Terabayashi. Subtraction stereo: a stereo camera system that focuses on moving regions. pages 723908–723908–11, 2009.

[51] Bart van Andel. Image rectification. `http://en.wikipedia.org/wiki/File:Epipolar_geometry.svg`. Accessed: 04/04/2013.

[52] Qingxiong Yang, Liang Wang, and N. Ahuja. A constant-space belief propagation algorithm for stereo matching. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 1458–1465, 2010.

[53] Ylebru. Birapport et projection. `http://en.wikipedia.org/wiki/File:Birapport_et_projection.png`. Accessed: 14/03/2013.

[54] Hui Zeng, Xiaoming Deng, and Zhanyi Hu. A new normalized method on line-based homography estimation. *Pattern Recogn. Lett.*, 29(9):1236–1244, July 2008.

[55] Zoran Zivkovic and Ferdinand van der Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction. *Pattern Recogn. Lett.*, 27(7):773–780, May 2006.