UiO : Institutt for informatikk
Det matematisk-naturvitenskapelige fakultet

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction

Mikkel Næss

Masteroppgave våren 2013

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction

Mikkel Næss

Abstract

Today, several soccer teams make use of the capabilities provided by analysis systems to further improve the overall team and player performance. As more and more analysis systems are developed, using these types of tools have become common and the advantages these systems provides can be the difference between winning or loosing a match.

To offer an analysis system that can capture team data, annotate important events and deliver real-time video we present the idea of Bagadus. With all of the these goals fulfilled, it is able to provide all the capabilities needed for team development. The system combines player tracking sensors, a easy-to-use annotation system and cameras to capture video of the games. The goal of Bagadus is to be fully automatic and real-time, but currently there are several issues that needs to be solved before one can take full advantage of the system.

In this thesis, we present an improved version of the previously existing Bagdus video capture component. By designing a pipeline for panorama creation, we are able to deliver a panorama video in real-time consisting of frames from four individual cameras. We will also look further into the visual output of the pipeline and how we can improve the image quality by applying a method for color correction.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisors Pål Halvorsen, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam and Carsten Griwodz for great help and motivation. I would also like to thank them for feedback, guidance and ideas for the Bagadus system. Kai-Even Nilsen have also been of great help when installing the system at our testbed in Tromsø.

I also want to aknowledge and thank Espen Oldeide Helgedalsrud, Marius Tennøe, Henrik Kjus Alstad and Simen Saegrov for their companionship, our interesting discussions and their work on the Bagadus system.

Finally i wish to thank my family and friends for their support.

Oslo, april 28, 2013
Mikkel Næss

Chapter 1

Introduction

1.1 Background

Today, there exists several system that delivers analytical tools for sport analysis. In team sports like soccer, these systems are both used to investigate overall game statistics as well as individual player performance. Often, these systems are expensive, demand a large amount of manual labor and can only deliver a limited amount of information in real-time.

One of the systems that is used for analytical purpose is Interplay sports [4]. By capturing the game from different camera angles and by later using manual labor, they analyze the video for events and general player performance. ProZone [5] is a system that takes another approach. By providing video analysis software, they are ableto identify interesting situations in the video based on object positioning, general player movement and more. This technology minimizes the manual labor, but the system may not be able to annotate all the events during a game. STATS SportVU Tracking Technology [6] takes a similar approach as it finds the player position with the help of video. The data is then used to generate game and player statistics. A general issue with these type of systems is that they often use a limited set of preprogrammed events. This can in many situations be very helpful, but coaches may want to capture a variety of events not detected by the system. Therefore, systems like the ones presented above will often create events that are false positives, i.e., not of interest for the system user.

ZXY sport Tracking [7] is a different system that have the capability of delivering data shortly after it was recorded. Taking a different approach compared to the other systems described, ZXY gathers its information with the help of player sensors and radio antennas. Creating a coordinate system from the field makes it possible to collect game information about the players and store it in a database. Although this is gathered and processed during the game and the information can be delivered only seconds after an event happened, the system lacks the capability of presenting the statistics with video.

As presented above, most of the systems today can only deliver and process a limited amount of information and most of them lack the ability to present the data in real-time. The systems offering video solutions fail to generate and deliver analytical data in real-time, while the ones that do, lack the ability to combine the data with video.

In the recent years, using video as a analytical subsystem has become a more and more common approach and generating a panorama image from a sequence of images is a well known topic in image and video processing. Many of these systems (e.g., [8–12]) have however, a different definition of real-time then the one we will use in this thesis. They often claim to process and deliver a panorama in real-time, but the definition of real-time is often dependent on the application and its area of use. When creating a panorama image, real-time is often described as "within a couple of seconds". On the other hand, when the application delivers a panorama video stitched together by different camera streams, the definition is often dependent on the frame rate the cameras can deliver, e.g., if the cameras reads a frame each 30 milliseconds (ms), the application should be able to produce a panorama frame at the same rate.

A system that delivers a panorama video within our definition of real-time is Camargus [13]. With high-end expensive hardware equipment, using 16 cameras (in an array) they are able to create a panorama video of the entire football field in high-definition (HD). Other systems, like Haynes [14] and The Omnicam system from the Fascinate project [15, 16] also produces high resolution panoramas, but as with Camargus, the systems are dependent on expensive and specialized hardware and as far as we know, they do not combine the panorama video with other analytical tools.

Immersive Cockpit [17] and the application presented in [18] manage to deliver panoramas in real-time, but with visual limitations. Immersive Cockpit aims to fulfill another goal then our application, and therefore the quality of the outputted panorama does not meet our requirements. Although the system in [18] is similar to ours, if they where to fulfill our real-time goal, they would only be able to generate low resolution panoramas from two camera frames.

An important part in generating visually pleasing panoramas, is the task of minimizing, or removing if possible, the color differences in the individual images. The task, often referred to as color correction, is a well known problem in visual applications. By minimizing the color and luminance differences in a image sequence, the method can under the right conditions completely remove the seam created when stitching the individual image frames together.

There currently exists many algorithms that presents solutions to the problem (e.g., [19], [20]). The system presented in [21] is one of them. By using their method, one is able to correct for color differences on a six image sequence within 5.5 seconds. An application that color corrects panoramas on hand held devices is described in [22]. Their method is fast and does not require accurate pixel to pixel mapping. The algorithm is also simple making it an interesting approach for minimizing color differences in a image sequence.

Although there exists systems that generates panoramas in real-time, systems that can deliver analytical data of important game events and application used for annotating game events, there exists no system as we know of today, that combines these features into one fully automatic analysis system. Bagadus, as presented in this thesis and in [23–25], aims to accomplish all these goals. By combining different subsystems, Bagadus aims to create a panorama video, that in combination with information gathered by ZXY, gives the users all player and team information they require in order to enhance their performance. The prototype of Bagadus is presented in [23, 26] and shows how the different components work together. However, the system is currently not able to generate and process all the information in real-time.

1.2 Problem Definition

Todays systems for generating panoramas either does not fulfill our real-time goal, or makes use of expensive and specialized hardware. To have a system that better fits our needs, we need to improve the performance of the Bagadus video capture component, and more specifically its panorama stitching pipeline.

In this thesis, we look further into the issues regarding panorama stitching by trying to improve the already implemented Bagadus pipeline. The goal is to create a real-time panorama pipeline with a visually pleasing output. To improve the quality of the end panorama, we will in this thesis investigate the opportunities for color correction as a method for removing color and luminance differences between the individual camera frames.

To improve overall pipeline performance, we will look into alternative program architectures and how we can divide the workload between different hardware components to ease the overall panorama pipeline workload. As an important part of the pipeline design, we will also see how to parallelize and divide the image operations into several sub modules. The goal is to create a real-time pipeline that can create a panorama consisting of frames from four individual camera streams. Lastly, we want to be able to record and write the four camera streams to disk at the same time as the panorama is generated.

By looking into different image and video algorithms, we investigate solutions for improving the visual output of the panorama pipeline. This will include different color correction methods as well as other algorithms aimed to improve the visual quality of stitched images. To see how the color correction fits our scenario, we will need to investigate how it works under different lighting conditions and if the camera setup and scenario at our testbed causes problems for the method. The goal is to have an output that is visually pleasing and can remove most of the visual artifacts caused by the panorama stitching.

1.3 Limitations

When creating a program that creates video from cameras in real-time, there are high requirements for hardware, both on the computer, and as well as the cameras and other equipment that is included in the process of creating and delivering video. In this project we have worked on commodity HD cameras, that compared to similar system is of low cost. This puts limitations on the project regarding the visual quality of the videos generated.

As there is no current implementation that connects the different Bagadus subcomponents together, we have only been able to test the interactions with the system in an off line environment. With the panorama pipeline we are also putting a lot of stress on the computers, meaning that for future scaling of the system, new hardware will probably be required.

The color correction suffers limitation do to the lack of blending in our panorama pipeline. For this reason the algorithm does not completely remove the visual artifacts in the pipeline output. Due to the real-time requirement of the pipeline, processing time and visual result is a trade off that needs to be taken into consideration.

The biggest limitations of today lies in the cameras, their lenses and setup. Fortunately, this is not the main scope of this thesis as the primary focus will be on processing speed and creating a stable, easy to use system for panorama video creation. Camera calibration is also not within the scope of this theses, therefore we do not have optimal camera calibration values.

1.4 Research Method

In this thesis, we have designed, implemented and evaluated an improvement of the Bagadus system. The latest version is deployed at Alfheim stadium in Tromsø. The research method utilized equals the *Design* paradigm, described by the ACM Task Force on the Core of Computer Science [27].

1.5 Main Contributions

We have shown with our work and research that creating a analytical system that is fully automated is possible. By creating panorama video in real-time, a goal described in section 1.2, we can deliver a video that capture the whole field, giving the system users a better overview of the different events, and improving the team performance as a whole.

By having a decent camera setup and some fundamental system components intact, we where able to get a system up and running. By creating a system for generating real-time panorama video, we have shown that with a good system design and the use of a Graphic Processing Unit (GPU), you can create good quality panorama video with limited hardware. Adding color correction to the program as mentioned in section 1.2 have made it possible to improve the visual quality of the panorama. We have also implemented a algorithm for dynamic stitching that further improves the visual quality of the output.

In our prototype, as well as with later versions of the system we have shown that by combining an event system, a tracking system and a video creation program, we are able to provide analytical data in a short amount of time. We have also shown with our research that the video capture component works well in a real life scenario and that the results we have come up with can be of great importance for improving team sport analysis.

When working on the Bagadus system and creating a real-time panorama pipeline, we have been able to submit and publish a poster for the NVIDIA GPU Technology Conferences 2013 [28] as well as contributed to a paper delivered for evaluation to ACM Multimedia Conference. Furthermore, the system has has attracted a lot of attention and several other soccer clubs has shown interest in Bagadus.

1.6 Outline

In this thesis, we present Bagadus, a real-time sports analysis system, and in particular, one of its most important subcomponents, a program for creating real-time panorama video using frames from four different video streams.

In the second chapter, we give a description of Bagadus as a whole, exploring and explaining the different components and the interaction between them. The hardware used by the system is presented as well as tools and libraries needed by the system. We present the Bagadus prototype and go further into details about the creation of an offline version of the system video capture component.

In chapter 3, we will move on to the main topic of this thesis, the real-time panorama pipeline creation. Its design and implementation details are presented and discussed in the first part of the chapter. Furthermore, we talk about the stepwise system progression and the tools needed to get the pipeline in real-time. At the end of the chapter, we present research and timing results that shows how the pipeline performs under different hardware specifications.

We have had a special focus on the color correction component of the real-time pipeline to improve the visual quality of the pipeline. Color correction as a module is therefore presented in chapter 4. We describe the fundamentals of the algorithm and why we have implemented the method in our pipeline. As with chapter 3, we present the design and implementation details of the algorithm used in the panorama creation pipeline. The last parts of the chapter contains discussions regarding modifications of the algorithm, as well as test result presenting both visual output and computational performance for some parts of the algorithm.

In the last chapter, we conclude by giving an overall view of the system in its current state and talk about future work in the project.

Chapter 2

Bagadus - A Fully Automatic Sports Analysis System

2.1 Motivation

Most sports analysis systems today require a large amount of manual work and is therefore also time consuming. As a result, the analytical data is not available until hours after it was recorded, meaning that the user cannot make use of the system capabilities until after the session has ended. This in turn leads to a scenario where the application user may not be able to take full advantage of the possibilities such a system has to offer. To provide these capabilities, we present Bagadus [23–25], a fully automatic sports analysis system that combines different subsystems into a analysis tool that can deliver both real-time video and real-time player/team analytics.

In this chapter we will take a closer look at the basic system ideas and how we create a fully automatic analysis system. We address the problems of video capture and synchronization between different system components. We also take a quick look into libraries and tools used to create a panorama image from four individual camera streams, and present the different subsystems used in Bagadus. Lastly, we look into the first prototype, its possibilities and limitations.

2.2 Bagadus - The Basic Idea

Most sport analysis systems today requires manual labor and lack the capability of gathering, processing and delivering the information in real-time. This limits the usability of the systems making it impossible to retrieve and analyze the data during a session. These issues however, can be solved by creating a component that gathers the information from different analysis tools and automates the interaction between them.

The idea of Bagadus is to create a fully automatic sports analysis system that can offer player statistics, annotated events and video in real-time. By integrating different subcomponents, Bagadus is able to remove the manual labor often needed for analyzing games and player performance. Making the system fully automatic also leads to a analysis tool that can offer its capabilities after and during the recording session.

The system is divided into different subcomponents. By using a sensor system gathering player information in real-time and an easy-to-use annotation systems, we can extract important analytical data without the use of manual labor. By using four cameras for recording, Bagadus is able to connect player positioning with video from four different angles. In addition the system offers a panorama view of the entire soccer field.

Figure 2.1 shows the setup and interaction between the different Bagadus components. By offering the capabilities of the subsystems in one application the user can easily replay annotated events and receive important analytical information. As can be seen from the figure, the different components works independently of each other and is later combined to offer the user all their capabilities as one system.

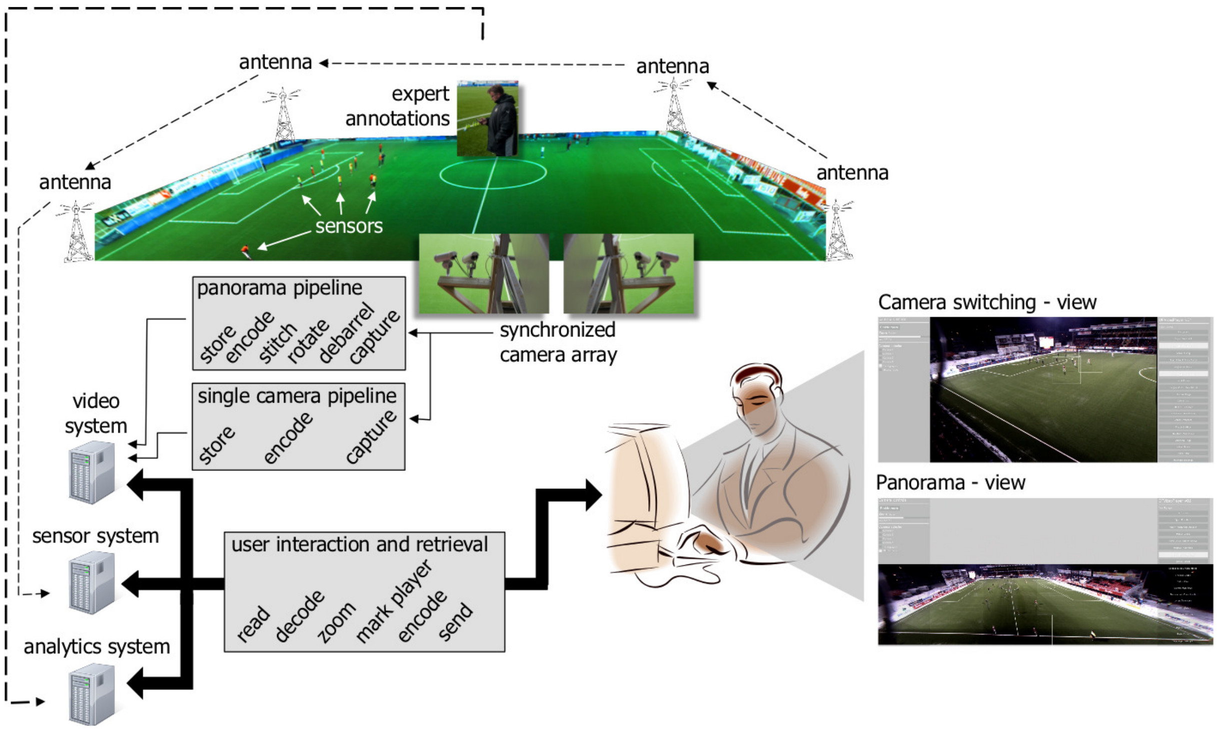


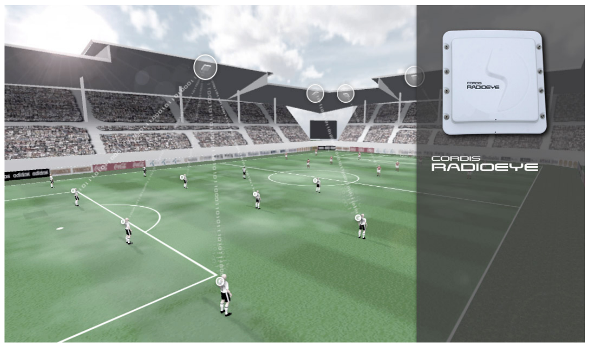
Figure 2.1: Bagadus component interaction.

2.3 ZXY Sport Tracking - Analytical Subsystem

ZXY sport tracking (ZXY) [7] is a system used for sport tracking and analysis. With the help of radio antennas and a belt strapped to each player, the system is able to gather information about each individual on the field. ZXY is currently installed at our testbed in Alfheim and offers player data like position, speed, hart rate and more. The system stores and updates data at 20 samples per second, giving player position accuracy of plus/minus one meter. Figure 2.2 shows two important components for the ZXY system, one of several radio antennas and the belts worn by the players.

2.3.1 ZXY Coordinate System

In order to gather the player statistics, the entire football field is mapped to a coordinate system where each coordinate represents a square meter on the field. In other words, the bottom left



(a) The radio antennas located on the stadiums where the system is implemented.

(b) The belt worn by the players. Transmits signal to the radio antennas.

Figure 2.2: ZXY system components used to track individuals during a game or training session.

corner is represented as (0,0) and the upper right corner as (105,64). This makes it possible to see the direction and position of the player at all times.

2.3.2 Player Tracking

Tracking objects in videos is a large field of research [29], but the current solutions are often time consuming and expensive duo to the large amount of data to be processed. Tracking players in team sports further complicates the task as you need separate the players from each other and other moving objects.

One of the key features of Bagadus is its ability to easily track players in video streams. By taking advantage of the capabilities of ZXY, Bagadus can track the players with the help of a homography mapping. In computer vision, if two images have the same planar surface in space, they are related by a homography mapping/projective transformation [30]. The homography maps the straight lines from one plane to the straight lines in another plane, in our case from ZXY coordinates to image pixels and image pixels to ZXY coordinates. This projective transformation makes it possible to locate a point in the ZXY coordinate system and find the corresponding pixel in the image frame.

To find the homography, represented as a 3×3 transformation matrix, we need to find common points between the the two planes. An illustration of this can be seen in figure 2.3. By using a figure that resembles the football field and the ZXY-coordinate system, we can find points in the coordinate system that matches points in the image frame. Here, this is illustrated by the red lines drawn between the ZXY-coordinate figure and the view from the leftmost camera.

When enough points have been mapped, we use OpenCV (described in section 2.7.2), a open source image library, to find the homography. OpenCV finds the transformation matrix with the help of the corresponding points between the two planes Now the resulting homography matrix (seen in equation 2.1) can be applied t.o equation 2.2, making it possible to find all the point in the corresponding planes. This homography makes it possible to track a player in real time as it relates all pixels in the image frame to a point on the field/coordinate system. For more about ZXY and player to pixel mapping, see [23].

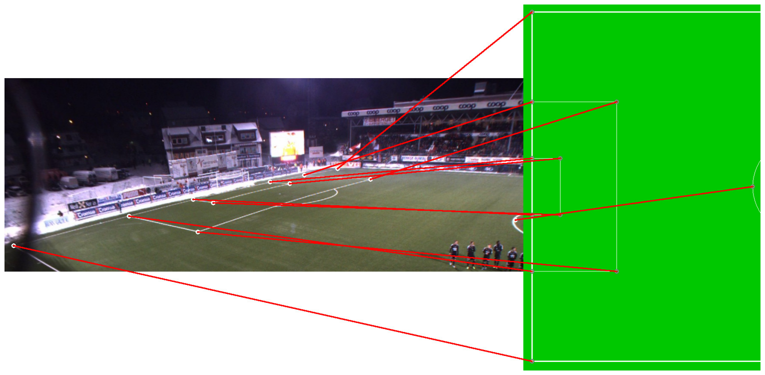


Figure 2.3: Mapping points between the camera frame and a figure of the pitch.

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

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

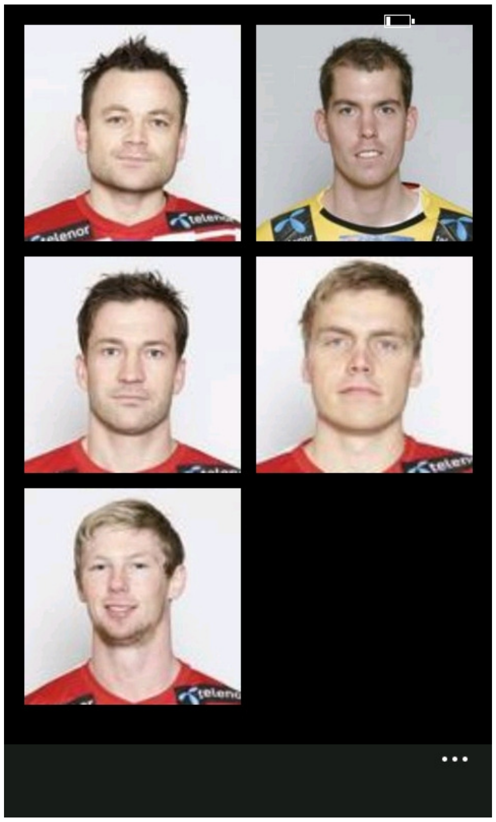
2.4 Muithu - Event Annotation Subsystem

When marking important game events, one common and often used method involves writing with pen and paper, and later move it to a computer that integrates the information with analytical data and/or video streams. To simplify the task of annotating interesting game situations, we have integrated our system with Muithu [1]. Muithu is a system that allows the coaches to annotate the events during a game with the help of a hand held device. After creation, the events are stored in a database for later analysis.

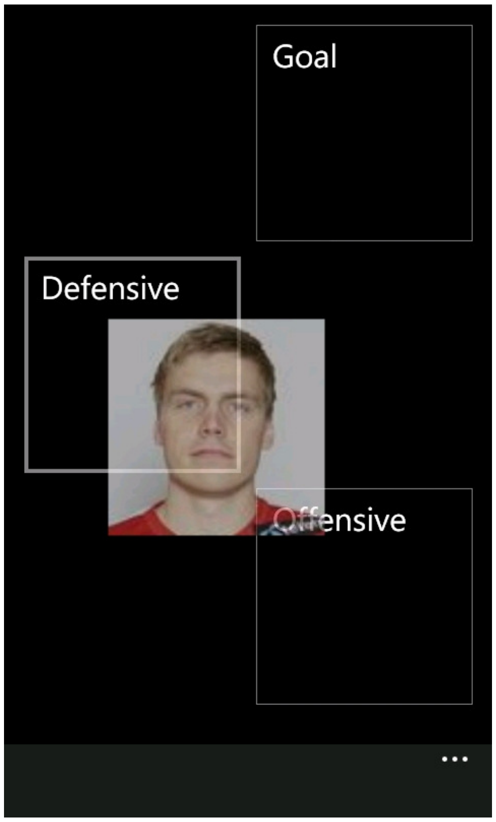
For fast and correct annotation, Muithu have a easy-to-use interface implemented on a Windows 7.5 cellular phone. The task of annotating an event is a two step process. First, you select the player involved, in figure 2.4(a) represented with a tile, and then you select the type of event. These events can be individual to each player and are most likely based on player position and skill level. When using the system in a training session, events are typically different for each player and corresponds to his or hers specific training goals. In a game, the types are simplified to offensive and defensive events do to the situation enhancing the importance of quick and easy annotation. Typical game events can be seen in figure 2.4. As the images shows you mark an event by first selecting a player, then dragging the tile to report the event type.

Event annotation happens in retrospect, i.e., to see if an event is actually interesting, the annotator lets the event play out. If the situation is of interest, the event is marked. Muithu stores the event by getting the event end-time and finds the start time by using a pre-defined time interval set by the system, a concept called ”hindsight recording”.

The features described above makes the system easy and highly reliably and the retrospect annotation makes sure that the number of false positives stays at a minimum.



(a) Example of how the the user interface of Muithu looks,where you select the player involved in the event



(b) Examples of how the the user interface of Muithu looks, here you see typical event types.

Figure 2.4: Muithu user interface [1]

2.5 Video Capture

The video capture component is the last piece of the Bagadus system. By using four cameras, the system can capture both game and training sessions from different angles that in combination covers the entire soccer field. In the current setup, we have placed the cameras in a cluster on one side of the field where two and two cameras are placed close together and the middle cameras are placed about four to five meters apart.

As of today, we use "Basler acA1300 - 30gc" cameras [31] with Kowa 3.5mm lenses, giving them a 68 degrees wide angle of view [24]. The wide angle of view lenses makes it possible to generate a panorama image of the entire football field using just four frames. The cameras are low/moderate cost industrial cameras giving us 1280×960 pixel frames at a rate of 30 frames per second (fps). Figure 2.5 shows the setup and camera overlap, giving us a view of the entire field.

2.5.1 Camera Options

The cameras from Basler offer several settings for optimizing the visual quality of the frames. The option that is of most interest to us is auto exposure, a camera mode that make the decisions regarding aperture setting, shutter speed etc. In the Bagadus project the mode is needed do to the troublesome light conditions at our testbed in Tromsœ. Due to long summer nights and dark winter days there is a lot of variety in the light intensity. This means that we cannot use the same exposure setting when recording games or training sessions.

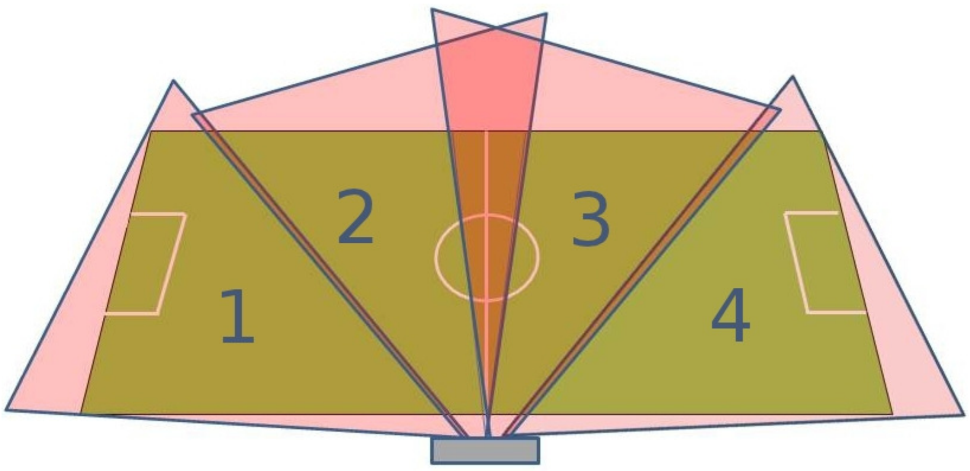


Figure 2.5: An illustration of the camera setup for the initial prototype. In the real-life setup the overlapping region between the middle cameras are much larger than depicted in the illustration.

2.5.2 Color Spaces

The Basler cameras use the Red Green Blue (RGB) color space, while capturing and presenting the frames in YUV.

Red Green Blue

In the RGB color model, each pixel is presented as a triplet, where the light intensity is measured in each of the components Red, Green and Blue. The values ranges from 0-255, where (0,0,0) is black and (255,255,255) is white [32].

YUV

The Basler cameras capture frames in the colorspace YUV, with the pixel format YUV422. Y'CbCr identifies a color by its brightness and color difference (Y is the luminance, while Cb, Cr represent the chrominance levels), and takes advantage of the fact that the human vision cannot comprehend all the information represented in RGB [32]. By using Y'CbCr instead of RGB we are able to minimize the data and still have good visual results. (For the rest of this thesis, we will use YUV when talking about Y'CbCr).

YUV can be represented in both packet and planar formats. Packet means that all values, Y,U and V, are interspersed. Planar means that the values instead are grouped together, which in turn can cause greater compression. In this thesis, we use three different YUV formats; YUV444, YUV422 and YUV420p. The different representations and the space required are shown in table 2.1and how they are stored in memory and compressed is explained below:

- YUV444 is stored as packed in memory, without any compression.

- YUV422 has a Y sample for every pixel, U and V are sampled to every second pixel, stored as packet format.
- YUV420p has a Y sample for every pixel, U and V are sampled to a two by two block in the frame. As can be seen in figure 2.6 the format is stored as planar.

YUV format	Subsampling ratio
YUV444	3 bytes per pixel
YUV422	4 bytes per 2 pixel
YUV420p	6 bytes per 4 pixels

Table 2.1: YUV format subsampling. Shows the different YUV compressions used in the system implementation.

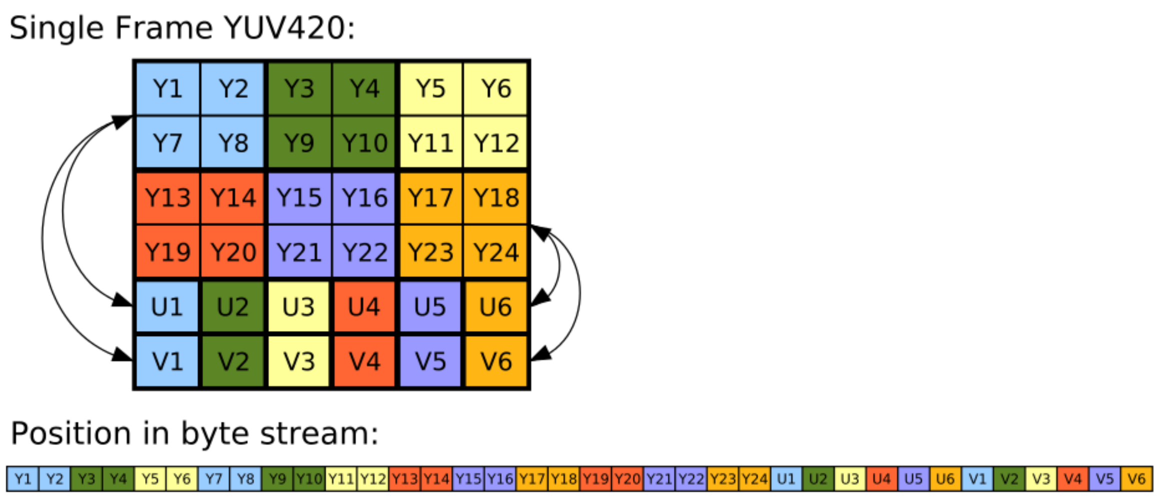


Figure 2.6: Example of YUV 420p [2]

2.5.3 Northlight

Northlight is a library developed at Simula Research Laboratory and is a part of the Verdione project [33]. It works as an interface towards common image and video libraries like x264 [34] and ffmpeg [35], and handles conversion, video recording and other related tasks.

Northlight is an important part of the Bagadus project as it is used for capturing video,conversion between different color formats and video encoding. For more on information regarding Northlight, please refer to the thesis of Simen Sægrov [23].

2.5.4 System Synchronization

For Bagadus to work as a whole, the system requires synchronization between the different subcomponents. If the systems do not work with synchronized clocks, we will not be be able to connect the correct player statistics from ZXY with video streams, or the video streams with Muithu, the event annotation system. We also needed to synchronize the different cameras and their frames so we can create a panorama video.

Camera Synchronization

For the first Bagadus prototype, each camera was controlled by its own computer. This caused two different synchronization problems; frame time stamp synchronization as well as camera shutter synchronization. As described in the section about camera options (2.5.1), we use auto exposure due to the varying light conditions at our testbed. Auto exposure controls the camera shutters, which in turn can lead to the cameras shutters closing at different times. To solve this, as well as the general issue where the machines controlling the cameras may themselves not be synchronized, we controlled the cameras and their frame rate with a device developed at Simula. The electronic device, called a trigger box, is able to synchronize all the cameras and their shutter speed so that all the frames where taken at the exact same time.

Subcomponent Synchronization

Synchronizing the different Bagadus components also means that we can synchronize the frame time stamps. In the prototype, the machines where not connected to the Internet, making it not possible to synchronize the components with the help of a Network Time Protocol (NTP) server.

With the help of Northlight, the problem was solved by using a time synchronization server called TimeCodeServer. The server is Open Sound Control (OSC) based, a protocol for communication between multimedia devices and its goal is to establish consensus between all clients. To accomplish this, server synchronizes a media clock by working as a slight modification of the Berkley algorithm [36]. For a more detailed explanation of TimeCodeServer and its algorithm, see [23].

The TimeCodeServer has later been deprecated in our system due to all the cameras now running on a single computer and that the machines are now connected to the Internet, making it possible to use a NTP server for synchronization.

2.6 Encoding And Storing

Saving video data for an entire game forces us to encode the videos. If using YUV420p, reading from disk will be to slow, making it impossible for the system to be real-time. The encoding also makes it possible to store more data for later analysis.

To find a suitable video format some tests needed to be done concerning read times as well as storage space. Some of the results can be seen in table 2.2. As can be seen in the table, RAW YUV420p, a subsampling using 6 bytes per four pixels is still to slow, forcing us to use another format. Needing a format that is faster to read and takes up less space, we chose to store our files in H.264.

Data location	Storage format	Storage space	Read and encode 3 second segment (ms)
Local	H264	179.69	2150.2
Local	YUV	2872.4	8150.9
Remote	H264	179.69	2872.4
Remote	YUV	910.98	13396.4

Table 2.2: Uncompressed YUV vs compressed H.264. Storage and processing tradeoffs for 4-image stitch [24].

H.264 (Advanced Video Coding) [34] is a video compression standard that was completed in 2003. As of today H.264 is one the most used format for High Definition video encoding, recording and compression. Below is a list of different frame types used to compress the data. The frame types are used when encoding H.264 and is based on a trade off between encoding times, decoding complexity and read-times.

- I-frame is the only type of frame that is totally independent. I-frames take more space, but is fast to encode and decode, but takes up more space.
- The P-frame contains only the differences between the previous frame and the one it represents. It takes up little space, but is dependent on its previous frame in order to be decoded.
- The B-frame is dependent on both preceding frames in order to be decoded [23].

Since one of the goals with Bagadus is fast generation of events, we will need to give all the video files timestamps, a type of metadata that is not supported in H264. To solve the problem, we store the videos in 3 second chunks and add the time stamp of the first frame in the filename. As a result there is no problem finding the start frame of an event knowing that each 3 second chunk contains 90 frames (30 fps). In Bagadus we have used the open source library x264 for encoding and decoding to H.264.

2.7 Bagadus - Initial System Prototype

In this section we give an introduction to the prototype of Bagadus, its implementation and address the issues and areas of future work. The prototype presented works on a subset of the ZXY database, and a set of predefined Muithu generated events, showing how the different components work together. Being the case that this is meant as a Bagadus proof-of-concept, the system version works off-line and is not able to process and deliver the needed data in real-time.

Due to the differences in the sample rate between the ZXY-sensors and the camera frame rate, resulting in a 2:3 relationship, we reuse the previous ZXY-information every third frame to synchronize the components.

2.7.1 Bagadus Demo

The interface of the demo can be seen in figure 2.7. The picture shows some of the possibilities the system provides, like selecting cameras, zooming and more. At the left side of the demo there are two lists. The top one lets you select one or more players that you can track. The names marked in dark grey are the players currently being followed. When tracking players in the system, we mark them by drawing a square around the players. Below the player tracking list, we have listed the different game events annotated. By pressing a given event, a pre-generated video of the event will start playing.

The top bar of the demo gives you options like zooming and camera selection. The camera selection lets you choose camera angle, including a panorama view or if you want to track certain players. When tracking players, the demo uses a option called ”smart camera selector”. The option chooses the best camera angle dependent on the players you track and where they are located on the field. The demo also gives the user a zooming option, making it possible to see a situation in more detail. The zooming option is so-called digital zooming that crops the image down to the specified area, but with the same aspect ratio as before.

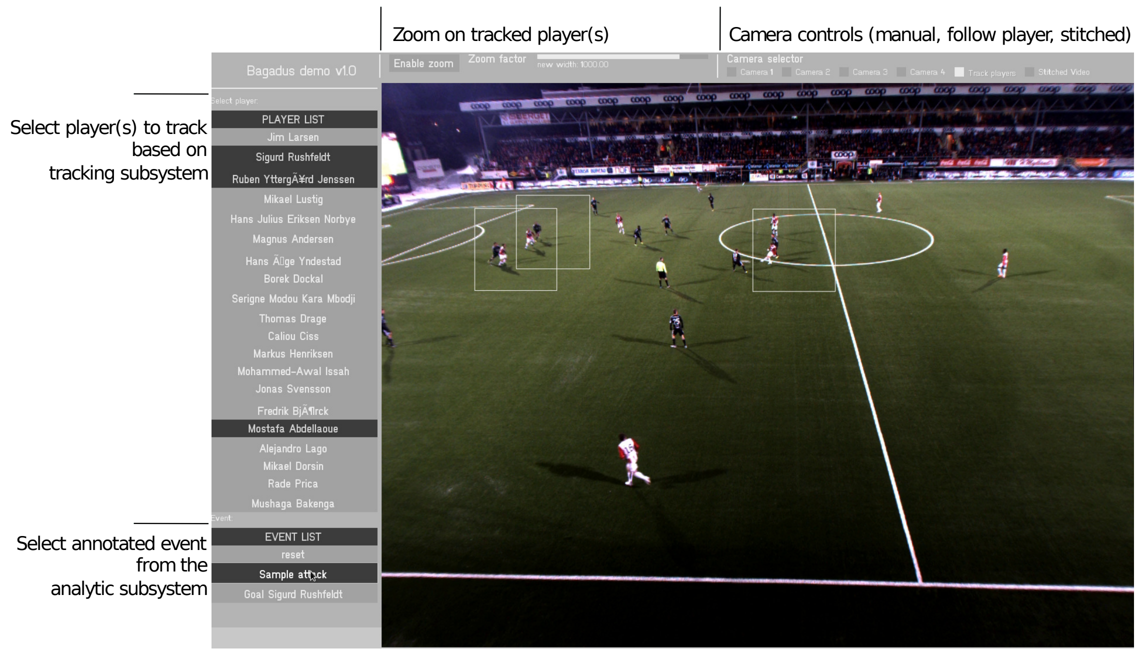


Figure 2.7: The Bagadus user interface.

2.7.2 Open Source Computer Vision Library

Open source computer vision library (OpenCV) [37] is a library that includes algorithms for image processing and computer vision. It offers a wide range of algorithms that can deliver its result within a short amount of time. In Bagadus, the library is used for its interpolation algorithms, method for removing barrel distortion and in some places for general image representation. More information regarding these issues can be found in [23].

Image Interpolation

Image interpolation happens every time you resize or remap an image. Image resizing involves changing the number of pixels in the image, while remapping is the task of relocating the pixels to a new location. To accomplish these task, we apply a interpolation methods to the image data. By estimating unknown points using already known data, the algorithms tries to find the best approximation for the pixels color value and intensity.

There are several interpolation methods implemented in OpenCV, one of them being Nearest neighbor interpolation (NN). NN treats all pixels equally and uses its closest adjacent pixels to interpolate. This low number of pixels makes the algorithm fast, but limits the visual end result. Throughout this thesis, NN is used as the interpolation method do to its performance.

Camera calibration

OpenCV have implementations of several different calibration functions. In the prototype, we only consider photometric camera calibration, i.e.,finding the true camera parameters.

For reasons described in section 2.5 we need wide angle lenses on our cameras, which in return gives us a distorted image. For removing these errors, we will use methods offered by OpenCV. How we make use of the algorithms and why is discussed later in the thesis under the section about debarreling 2.7.3.

2.7.3 Video Panorama Creation - Demo Version

To achieve the Bagadus goal of delivering panorama video, a stitching pipeline was created in the first prototype. The implementation is an offline version using all four cameras to generate a video consisting of the entire field. As the improvement of this pipeline is one of the main aspects of this thesis, we will explain the pipeline steps, how they are implemented and their performance.

Color space conversion

The conversion from RGB to YUV and YUV to RGB are both done by calling functions in the Northlight library. Using ffmpeg, Nortlight is able to perform fast conversion between the two color spaces.

The operations in the panorama creation does not force us to use RGB, but the format was chosen as it is slightly simpler to work with compared to YUV, and the implementation of computer vision algorithms can be somewhat simplified in the later stages of the project. For this reason, we convert as the first and last step in the pipeline.

As seen in table 2.3, the conversion from YUV420p to RGB is finished within our threshold with good margin. The reverse conversion, RGB to YUV420p, have a maximum value higher than our threshold and a mean value that manages to stay just below 33 ms. As the prototype pipeline does not execute anything in parallel, the conversion can use all the resources it needs and therefore manages to execute within 33 ms.

Image Debarreling

As a general rule of thumb, using lenses with a wide angle of view requires removal of barrel distortion, a distortion giving the impression that the image frame is wrapped around a barrel, i.e. the center of the image is magnified more than the perimeter. Do to the visual result and the video player tracking explained in section 2.3, this effect needs to be removed from the images. Here, we will only give a brief explanation on how to remove barrel distortion and since image debarreling is a complex process and topics involving camera calibration is not in the scope of this thesis, we ask the reader to look into barrel distortion in [23] if interested in more details.

When implementing the system, we used OpenCV to calibrate the cameras. The debarreling can be divided into three basic steps.

- OpenCV's algorithm for solving distortions starts with a initialization/configuration step. By using a chess board pattern, you take a series of pictures from different angels and give them as input to the OpenCV algorithm. As output you get so-called distortion coefficients represented in a 5x1 matrix. This needs only to be done once.
- The next step involves a setup phase for the final remapping where the calculation for each pixel is done. This needs to be done for every frame.
- As a final step, the actual remapping is done by using a interpolation algorithm. In the prototype the algorithm is as previously mentioned Nearest Neighbor Interpolation.

The debarreling uses only one thread, i.e., the frames are debarreled in a sequence. Debarrel performance can be seen in table 2.3. The mean value is only at 17.879 milliseconds indicating that the debarrel implementation is computationally fast enough if we where to parallelize the execution of the different panorama creation steps.

Figure 2.8 shows what the barrel distortion algorithm aims to accomplish. By remapping the pixels the algorithm is able to straighten out the lines suffering from barrel distortion. Since the lenses are different from each other, we need apply the procedure to all the cameras using their camera matrix and distortion coefficients.

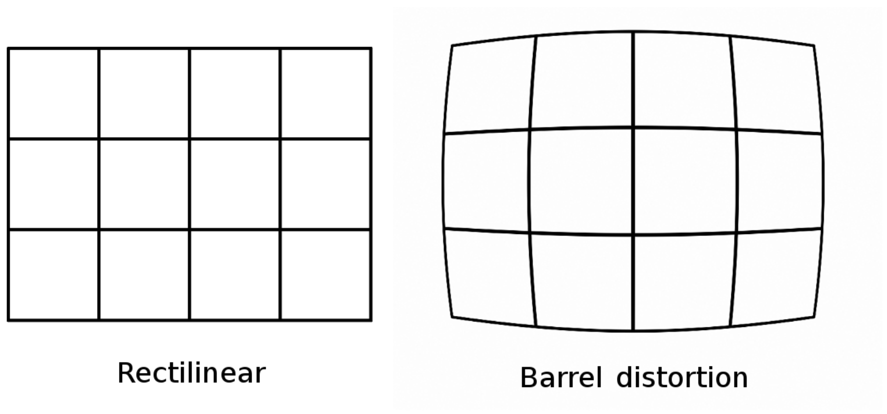


Figure 2.8: Barrel distortion

Warping step

To be able to stitch the four camera frames together we need to align the images to fit the same plane. We accomplish the task by warping the images, a method used for moving pixels to another location without disrupting its colors [38]. The warping does not only help for image alignment, but also simplifies the panorama pixel to ZXY coordinate system mapping used for player tracking.

We start the image alignment procedure by first selecting a ”head” camera, in the prototype this is the middle left camera. The camera will be the basis for the realignment of the other frames. This means that when doing the plane transformation, the other cameras will changes their alignment in order to fit the head camera. To do this alignment, we need to find the homography between the middle left camera and the other cameras frames. The homography, as described in section 2.3 is found by matching and mapping points between the main plane and the other images. In the first prototype this is done manually at system setup and typical points include lines, goal posts, corners and more. The resulting homography/transformation matrix is similar to the one in figure 2.1. These matrices are then applied in order to warp the images. The warping, just like debarreling makes use of the Nearest Neighbor interpolation and is implemented with the help of functions offered by OpenCV.

The resulting warped images can be seen in figure 2.9. The image shows the warped frames and how they are aligned and fitted on top of each other to create the final panorama.

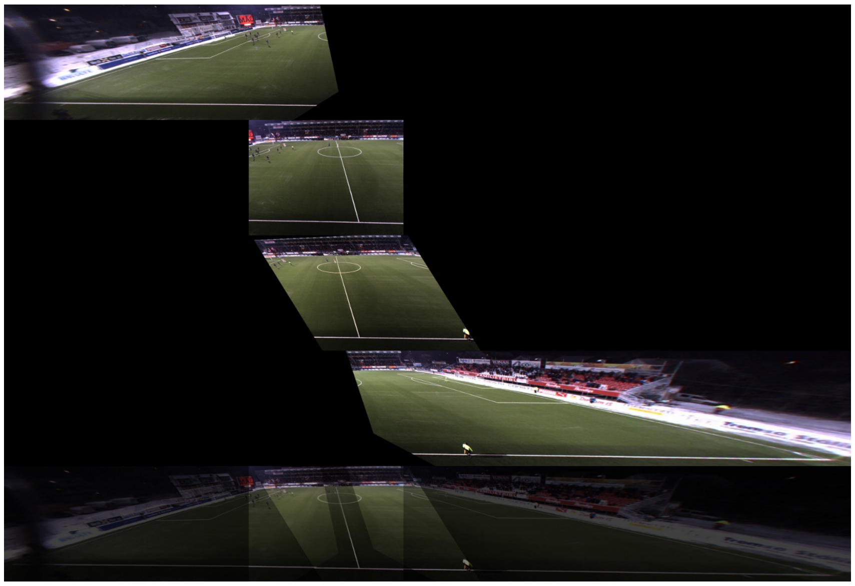


Figure 2.9: Warping result

Stitching step

To create the final output, we need to stitch the images into a panorama that covers the entire field. Being the case that all images are aligned to the same plane, the stitching is quite simple. To find suitable positions for the seams separating the frames, we need to have the overlapping regions between the adjacent images. These overlapping regions are in the prototype found and set at system setup. By using the information from the overlapping regions we find suitable offsets that decide where the seams will go. When the positions for the seams have been found, the images can be stitched together by simple memory operations. In the prototype these seams are static, a solution that is not optimal since the seams create visual artifacts in the end panorama. An example of these artifacts can be seen in image 2.10. Here we can see that when players are in the region of the stitch, they get cut in half do to effects caused by warping and poor image alignment.

After the stitching is finished the final panorama is cropped to remove dark regions surrounding the image. For mapping the ZXY data with the panorama a new transformation matrix need to be applied being the case that three out of four images have been warped and the pixels are no longer located at the same position in the image.

The Stitcher and the Warper results are combined under the title ”Stitch” in table 2.3. These steps are the main bottlenecks for the first pipeline prototype. Using a mean time of 1005.11 ms to finish warping and stitching is far from what we aim to accomplish with Bagadus. The visual output of the first prototype is seen in figure 2.11. Here, we see the static seams and the effects the current camera setup and image alignment have on the end result.

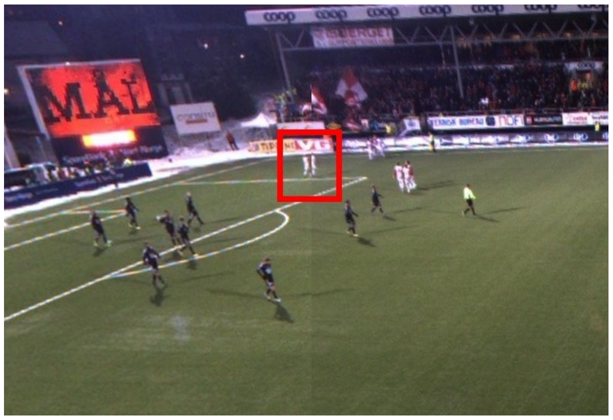


Figure 2.10: Artifact caused by player crossing stitch seam

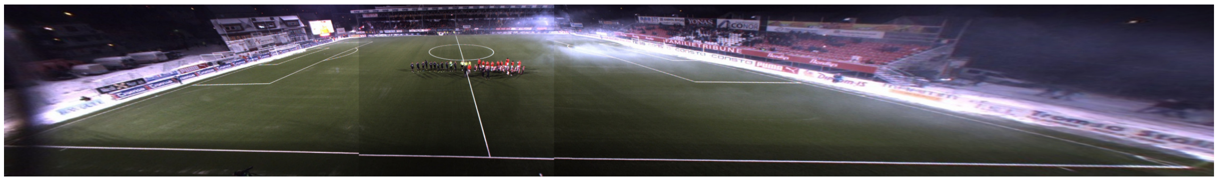


Figure 2.11: Stitched Panorama

Prototype Performance

As shown in table 2.3, the panorama generation in the prototype is nowhere near real-time. The system only manage to deliver one panorama frame per second. Both the conversion steps manages to stay within the threshold, but the conversion from RGB to YUV is just within our time limit. As of now, not one of the operations presented above executes its code in parallel, thus the pipeline can not deliver frames at a high rate. Parallel execution of the different program steps might be a good solution to improve the panorama creation performance. However, this will also lead to a bigger workload on the CPU. This will in turn give less resources to the different pipeline steps and might be enough to cause the conversion to go above the Bagadus threshold.

The stitching time represents as mentioned, the time it takes to warp the images and stitch them into the final panorama. In the prototype this represents the main bottleneck and was something that needed to be improved in order to produce real-time panorama video.

Module	Min	Max	Mean
Reading YUV420p	46.566	573.159	85.419
Convertion to RGB	5.157	10.891	6.365
Debarrel	17.338	20.469	17.879
Stitch	988.76	1041.61	1005.11
Convertion to YUV	26.759	34.892	31.505
Writing YUV420p	0.001	0.015	0.001
Total Time	1092.3	1660.06	1146.29

Table 2.3: First prototype results (in ms), creating a panorama from four frames. Running on Test machine 2 (TM2 in table A.2)

2.8 Summary

In this chapter, we have described the idea and opportunities of Bagadus, a fully automatic sport analysis system. The system, consisting of three subcomponents, removes the manual labor often needed in analysis systems, and lets the coaches themselves mark interesting events that later can be presented in video streams.

We have looked into the different subcomponents ZXY, Muithu and the video capture component. ZXY [7], a system for monitoring the overall game statistics as well as individual player performance, is an important component of Bagadus. By using sensors, ZXY is able to track players and store important game information regarding the team as a whole and the individual players. For easy annotation, we have integrated Muithu [1], a system that delivers a easy-to-use interface that lets you tag interesting game events based on the players and tactical aspects involved. The video capture component records video from four different angles and also offers a video of the entire field in a panorama.

We have presented the current setup of the system, its drawbacks and looked into the issues regarding component and camera synchronization. We have also briefly presented some of the tools needed when creating the system, the open source libraries Northlight and OpenCV. At the end of the chapter, we discussed the Bagadus prototype, an offline version showing some of the capabilities the system can offer.

The demo seen in figure 2.7 demonstrates the Bagadus proof of concept, but as mentioned before, the component synchronization is done offline and the panorama pipeline is not able to deliver video in real-time as seen in table 2.3. The panorama also displays some visual artifacts do to the lack of color correction, dynamic stitching and more. Therefore, we will in the next chapter present an improvement of the Bagadus panorama pipeline that is able to create better visual output and deliver panorama frames in real-time.

Chapter 3

Real-time Panorama Video

3.1 Motivation

The goal of Bagadus, being a fully automatic system requires that all data; player statistics as well as game videos can be accessed shortly after it was recorded. As seen in the previous chapter, the creation of a panorama video was far from real-time. The implementation of the prototype was also lacking the ability to create a good visual output.

In this chapter, we present a pipeline for generating an improved real-time panorama video. By looking at ideas and previous work on the topic, we have designed and implemented our own program for panorama creation.

3.2 Related Work

In the recent years, several methods for stitching image sequences into a panoramic image have been proposed (e.g., [8–12]). Many of these applications and systems also claim to be able to do the panorama creation in real-time, and the task is even a supported operation for many hand held devices, including cell phones and tablets. However, these applications and systems do not always share the same definition of real-time. Unlike for video panorama systems where the real-time requirement is most often based on the frame rate delivered by the camera, other systems may use a real-time definition of ”withing a couple of seconds”.

A system that generates a panorama, but uses another definition of real-time than our system requires, is presented by Baudisch et al. [39]. The system is also highly dependent on user interaction, making it ill suited for Bagadus. Camargus [13], Omnicam from Facinate [15, 16] and the Content Interface Corporation system described by Haynes [14] all deliver high resolution panorama video within our definition of real-time. However, they all require expensive and specialized hardware and as of now, we also lack insight to the systems. Haynes [14] system also makes use of static stitching which in our scenario leads to a unpleasing visual result.

The system presented by Chen et al. [40] creates panoramas of already existing videos, but as far as we know, as the with other systems (e.g., [39, 41–43]), it does not fulfill our real-time demand. Another system, Immersive Cockpit [17] generates panorama video for tele-immersive applications. The system main goal is to create a video that gives a wide field of view, and therefore the quality of the output is not their first priority. Although they claim to generate panorama video with a frame rate of 25 fps, the visual limitations to the end result is not good enough for Bagadus.

A system that fulfills many of our goal is presented by Adam et al. [18]. By computing stitch maps on the GPU they can deliver frames within our definition of real-time. However, to accomplish the real-time goal, the stitch maps generated by the system only manages to create low resolution panoramas stitched together by two images.

In summary, the systems presented above (e.g. [18, 39, 41–43]) does not fulfill our demands with regard to our visual and real-time requirements. The ones that do (e.g. [13–15]) require expensive and specialized hardware, making them not fit into our goal of creating a low cost panorama pipeline. Therefore, do to the lack of existing systems fulfilling the Bagadus goals, we have implemented our own system that is able to generate a real-time panorama video at a frame rate of 30 fps by distributing its workload between the CPU (host) and GPU (device). The next sections presents the system by describing its design, implementation and performance results.

3.3 Nvidia CUDA

In this section, we will give a short description of CUDA and the CUDA primitives used in the system. For a more detailed description, please refer to the CUDA manual [3].

To be able to accomplish our goal of generating a panorama video that can deliver a new frame each 33 ms, we need to parallelize the computations and algorithms do to the large amount of data each image frame represents. The best way to do this is with the help of Nvidia CUDA and its many libraries. CUDA is platform for parallel computing and programming model developed by NVIDIA and is often used for graphic programing as it is able to run thousands of lightweight threads at the same time.

3.3.1 CUDA Threading

Having a architecture that stresses thread parallelization makes implementing algorithms somewhat different than on the CPU. While CPU threads tends to be heavy, CUDA threads are lightweight.

Kernel execution

A CUDA function is referred to as a kernel and to execute them you need to know how the architecture of the device looks from a programmers perspective. When programming using CUDA you work on so-called grids, blocks and threads. Grids are divided into blocks that are divided into threads. While a grid is organized as a two-dimensional array(gridDim.x $\times$ gridDim.y), the blocks are divided into three-dimensional arrays(blockDim.x $\times$ blockDim.y $\times$ blockDim.z) and can maximum be made up of 512/1024 (dependent on CUDA Compute) threads. By using the blockID and threadID variables you are able to give each thread a unique ID, shown in figure 3.1. This means that each CUDA thread launched can be specified to work on one specific is of interest in the data. To understand how CUDA takes advantage of these capabilities, see figure 3.2. The illustration shows the CUDA architecture where grids are divided into blocks that again is divided into threads.

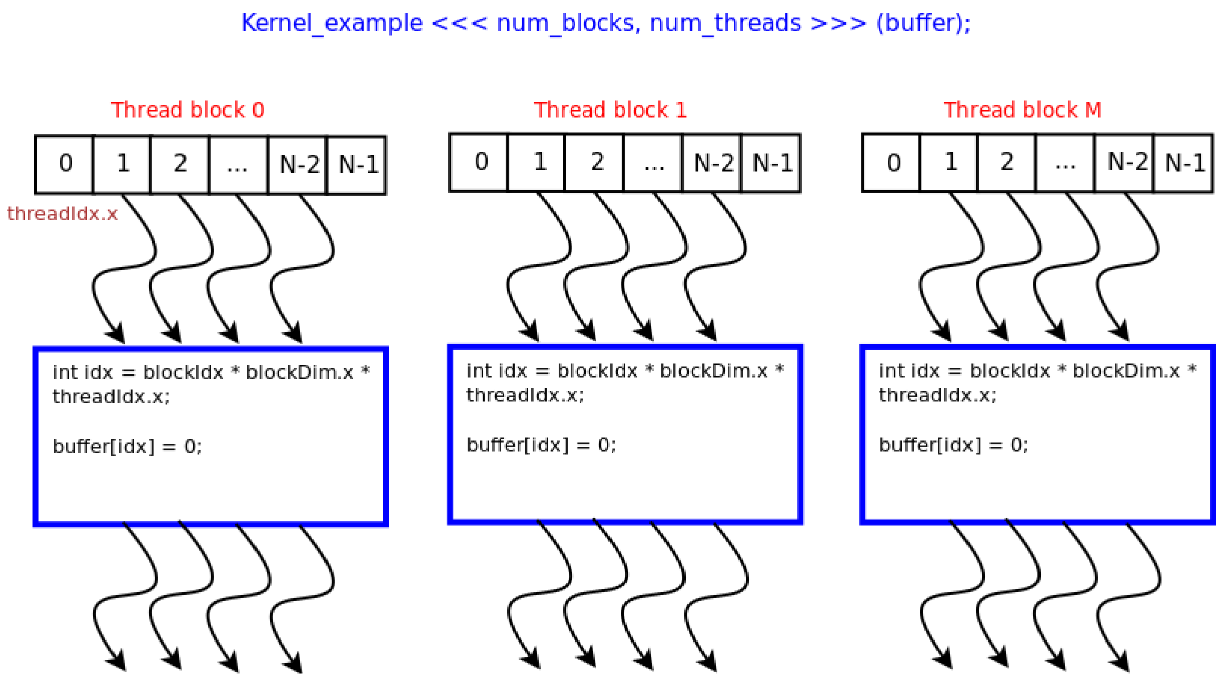

Kernel_example <<< num_blocks, num_threads >>> (buffer);
Thread block 0
Thread block 1
Thread block M
0
1
2
...
N-2
N-1
threadIdx.x
int idx = blockIdx * blockDim.x * threadIdx.x;
buffer[idx] = 0;

Figure 3.1: CUDA thread execution.

To call a CUDA kernel you specify how many threads on how many blocks (blocks $\times$ threads) you want. As an example in image processing you typically let each thread handle/work on a given amount of pixels by finding the optimal number of threads and blocks. Below is an example of a call to a CUDA kernel:

- Kernel_test <<< num_blocks, num_threads >>> (kernel_arguments);

The name of the CUDA kernel in this example is Kernel_test, the num_blocks and num_threads parameters tells us how many blocks and threads we want to have running for executing the kernel with arguments kernel_arguments.

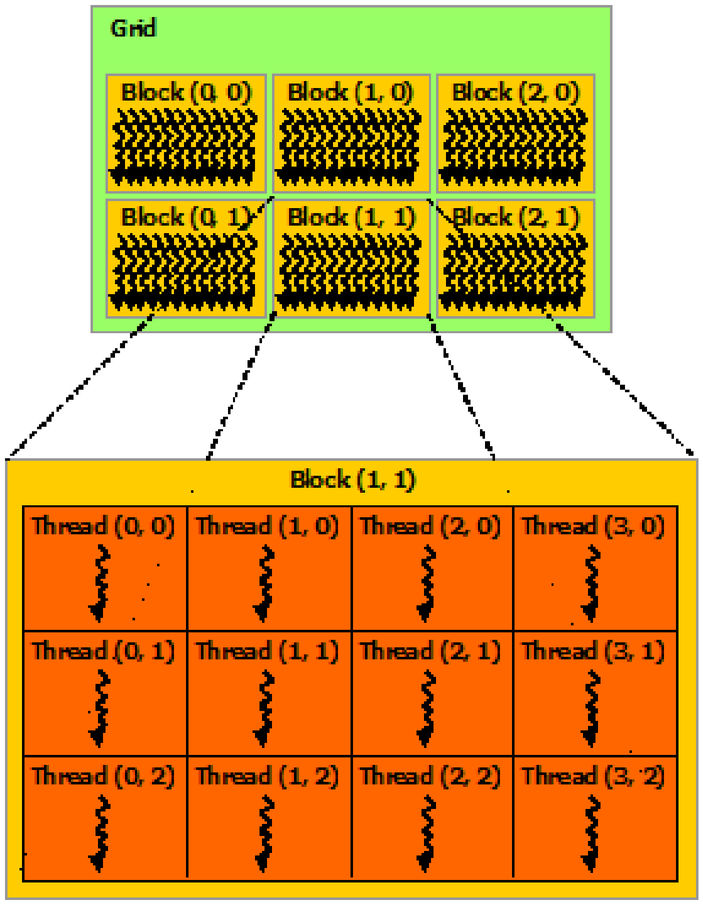


Figure 3.2: CUDA architecture [3].

3.3.2 Memory Management

Before we can make use the device, we need to move the data, i.e. memory buffers and variables to GPU memory.

Device Memory

CUDA offers a number of different ways to allocate memory on device. The ones used in this thesis is mostly cudaMalloc() and nppiMalloc(). cudaMalloc() allocates memory on the device, that later can be accessed by device code. nppiMalloc() is explained in section 3.3.4.

Pinned Host Memory

CUDA offers a mechanism for allocating pinned memory on host, cudaHostAlloc(). When allocating memory as pinned, the buffer becomes page locked. When memory is page-locked, the OS will always keep the page in its physical memory. Knowing this, the GPU can copy data to or from the host with Direct Memory Address(DMA) providing higher transfer throughput between host and device. DMA are copy operations that is not intervened by the CPU meaning that the CPU could simultaneously paging out, or reallocating the buffers in physical memory address. Paged memory should only be used when using the memory in a CUDA memory copy, and should be freed as soon as the operation is done.

3.3.3 CUDA Streams

CUDA streams is similar to threading on the GPU, it is a sequence of operations that executes in a specific order on device. The streams may run concurrently and the operations in these streams may be interleaved. To use CUDA streams and do work concurrently, one needs to use asynchronous memory copies, as well as page-locked (pinned) memory. Operations are added to the stream, and the order they are added will be order they are executed. A stream is created with CudaStreamCreate().

Asynchronous Memory Copies

cudaMemcpyAsync() performs a memory copy in a stream specified by the last function argument. For using Asynchronous memory copies it is required that the memory allocated on host is allocated on pinned memory. To make sure that the memory copies are done one needs to synchronize with the host with a call to cudaStreamSynchronize().

3.3.4 NVIDIA Performance Primitives

Nvidia Performance Primitives(NPP) [44] is a library that focuses on imaging and video processing, and have a wide variety of media functions and algorithms implemented. All the functions in NPP works on data that is located on the device, so if the image data is not already on the GPU, it needs to be moved from host to device.

The NPP library makes us of a line step (pitch) in order to align images so that each row starts on a well aligned address. This is done by adding a number of unused bytes to the end of the row. The pitch is therefore the width of the image plus additional padding. The use of this pitch can be seen in figure 3.4. This padding optimizes memory-access patters. To make use of this line-step the NPP library has its own memory coping function, cudaMemcpy2D() and its own set of allocating functions.

Although NPP delivers a lot of functionality, the library is used mainly for copying data, as well as conversion between different color formats. As for built in algorithms, we make use of the functions for warping an image as explained in section 3.6.8 and for RGBA to YUV444 conversion.

3.4 Bagadus Demo Pipeline Improvements

To find the bottlenecks of the panorama generation, we started our implementation of a real-time panorama pipeline by taking steps to improve the performance of the prototype described in chapter 2.

We started by identifying and improving the part of the program that was the most time consuming. As seen in both table 3.1 and table 2.3 this is the part referred to as Stitch. This part includes both warping the images and stitching them into a final panorama. The other parts, the I/O, the converters and the debarreler are all running within the Bagadus threshold and therefore was not the main focus of this program improvement.

Having located the system bottleneck, we now needed to start improving the performance. The warper in the first prototype is as mentioned in section 2.7.3 implemented with OpenCV, executing on the CPU. To speed up the performance, we moved the warping to the GPU by using the NPP library. This improved the warping to the extend where the only program step not running in real-time was the stitcher. In the first prototype, the stitching copied all image pixels one by one to the final panorama. Being the case that the first Bagadus prototype uses static stitching, we simply improved the performance by copying the frames directly over to the final panorama, i.e. not one and one pixel, but the entire image in one copy operation.

After these improvements, the only program step not running within our 33 ms threshold is the video reader. However, being the case that the videos is stored at YUV420p, storing the files in another format will resolve the problem. The timings from the first pipeline and the improved one can be found in table 3.1. We can see that, except for reading, everything is in real-time.

Even though every program step is within our real-time requirement, we can see that the total time taken to deliver a frame is 132.505 ms. As mentioned in chapter 2, the first Bagadus prototype is not parallelized and therefore these improvements where not enough to fulfill our goal of a program for real-time panorama creation. The program also lacks the capabilities for creating a visually pleasing panorama. For these reasons, we needed a new program with a different design that could parallelize the different program steps and improve the visual result.

Module	Improved prototype	First prototype
Read	83.073	85.416
Convertion to RGB	7.154	6.365
Debarrel	17.928	17.879
Stitch	20.1881	1005.11
Convertion to YUV	26.1542	31.5905
Write	0.001	0.001
Total Time	132.505	1146.29

Table 3.1: Differences in processing time between the vanilla prototype pipeline and the updated prototype (Running on TM2).

3.5 Bagadus Real-time Panorama Pipeline

To generate a panorama video in real-time, we needed to create a new pipeline design that would take advantage of both the CPU and the GPU when executing. For this reason, we created a pipeline that separates its workload into several subcomponents. These subcomponents, or modules as they are referred to in this thesis, each have responsibility for doing its individual image processing step without further interference with the other components.

By letting all the modules execute its work at the same time, the pipeline is able to process as many different image sets (one image from each camera) as there are modules in the program. When the first module in the pipeline is done with a frame set, this frame set is delivered to the next module in the pipeline while the first module takes in a new frame set. The different modules works in parallel to optimize performance and output rate of the pipeline. This is done so that we are able process as much data as possible, while still being able to deliver panorama frames at a high rate.

The pipeline has an initial delay when delivering the panorama video, but after the first frame is written, the frames are delivered at a rate of 33 fps. This means that the modules have a limit of 33 ms to finish its image processing. The modules can do their work on the CPU, the GPU, or on both host and device.

The program is controlled by a main thread named the Controller. The thread controls the entire execution of the pipeline, making sure that each module executes at the correct time and only works on its own data. Figure 3.3 shows the different modules and the flow of the pipeline. The blue color indicates the modules running on the CPU, while green indicates GPU implemented modules. Comparing this figure with the module flow pattern in figure 3.5 shows how the pipeline modules run in parallel to make it real-time, and the general frame flow through the system.

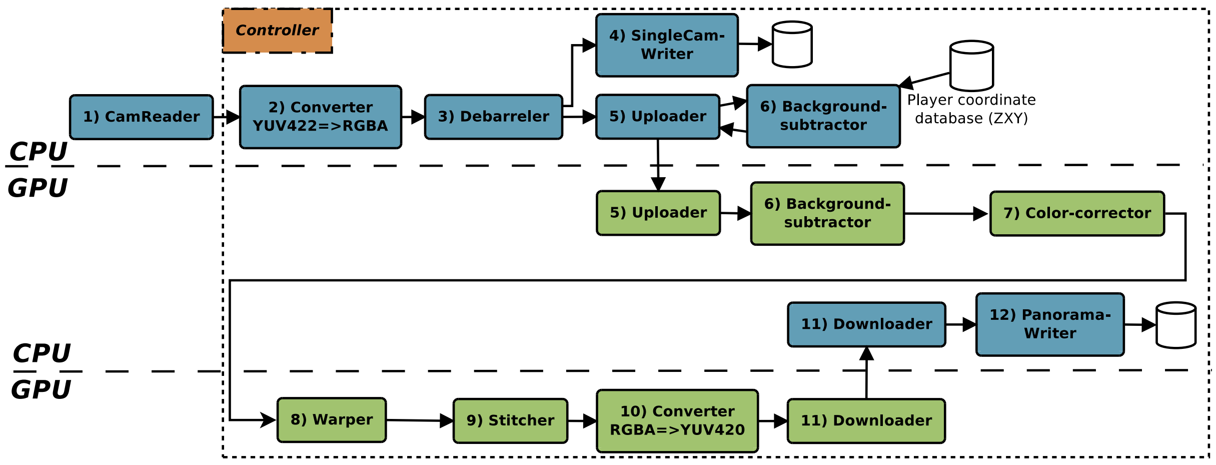


Figure 3.3: Pipeline flow.

3.5.1 Pipeline Encoding

To save storage space and be able to read the videos in real time, we encode our videos to H.264. Keeping the encoding within our pipeline threshold requires use of AUTO number of threads, i.e. number of cores on the machine $\times$ 1.5. We also encode with slicing [45] the image into 4 regions (one thread per region) for optimal performance.

The encoding part itself is not its own module, but placed in the different writer modules, respectively SingleCamWriter and PanoramaWriter decribed in sections 3.6.14 and 3.6.13. Table 3.2 shows the encoding time with different settings for profile and bitrate. We have selected to use the lossyless profile and hifi bitrate as it is a good tradeoff between speed and visual result. When encoding in the pipeline we use the Northlight library described in section 2.5.3.

Table 3.2 shows encoding timings with different number of threads as well as with and without slicing. As the figure shows, the number of threads we are using in the pipeline gives far better results than with the other settings. The key to get the encoding in real-time is the use of slicing and as the table shows, this setting makes the encoding execute approximately 3 times as fast. (For more details on encoding, go to section 2.6).

	No threading	Frame threads: AUTO	Frame threads: AUTO Slice threads: 4
Min	29 ms	29 ms	8 ms
Max	140 ms	115 ms	92 ms
Mean	34 ms	34 ms	10 ms

Table 3.2: Encoding 1000 frames on TM2 (presented in A.2), includes writing to disk.

3.5.2 Pipeline Frame Drops

The pipeline will in some scenarios drop frames. If remaining unsolved, this will be a problem that causes the timings and time stamps to be incorrect when later decoding and viewing the panorama video. To solve this, we have implemented pipeline features that handles these drops. The mechanism makes sure that each seconds of video contains 30 frames.

There are two situations where the frames might be dropped in the program:

1. The frames may be dropped by the cameras or their drivers. This usually happens with the first couple of frames, but may also happen at a later point in the recording. If frames are dropped in the beginning of the recording, i.e. the first ones read are also lost, we replace the frames with empty data. Although visually these frames will be green in the video, this is only the case in the first couple of ms and therefore the solution does not effect the analytical goals of the video capture component.

   When we drop frames later in the recordings, we approach the problem slightly different. By using the frame read just before the one lost, we can simply replace the new frame with the old one, only updating the frame time stamp. As an example, lets say that the camera drivers drops frame 50003, we will then replace the dropped frame with frame number 50002. Since we receive 33 frames per second this will not be visible when playing the video. These frame drops are caught and resolved in the reader module described in section 3.6.3.

2. The frames may be dropped in the pipeline do to OS spikes, interrupts or hardware limitations. When a pipeline module uses more time than our threshold, the frame needs to be dropped. To solve pipeline drops, each module, as well as the controller have implemented frame drop handling. When a module drops a frame all the remaining modules gets notified. The last one to know is the PanoramaWriter. The module does the same as described in point one; it replaces the last frame with the previous one.

The installation of the pipeline at our testbed drops just below 1 percent of the frames processed and the drops are pretty equally divided between the panorama pipeline and the drivers/cameras. For test results with frame drop handling enabled, go to section 3.8.10.

3.5.3 Pipeline Startup

To run the program, it is important that the user supplies the pipeline with parameters indicating when the pipeline should start recording and how long it should run. At startup, the program parses these parameters and starts the recording when specified. Next, the pipeline initiates CUDA and finds the device best suited to run our pipeline.

As a last step, it initializes all the modules used in the pipeline and allocates the memory space needed for the camera frames, this is done approximately 30 seconds before the scheduled recording should start.

3.5.4 Controller

To control the execution of the pipeline, a main thread called the Controller ensures that all the modules executes when they are suppose to. Other important controller tasks include sub-component synchronization, catching errors and handling the image buffers in the pipeline. By letting a single component handle all synchronization and communication between the different modules, we are able to save processing time and simplify the overall program architecture. With the help of lock mechanisms such as barriers the controller is able to remove pipeline race conditions and catching pipeline frame drops.

Implementation Details

Except from initializing the modules and allocating necessary frame buffers, the Controller does the following as long as there is still frames to record:

- Retrieve a new frame set.
- Manage the module buffers:
  - Move the image data from one module to the next. The controller manages the buffers contained in the modules by moving the image data from one module to the next. This is done either with a copy of memory space, or with swapping of pointers. The pointers are swapped if the buffers are of the same type and size, otherwise, we copy between the modules.
- If a module drops a frame, handle it in the controller.
- Start a new module execution where all modules are ready for working on a new frame set.

Figure 3.4 shows how we swap buffers between modules in the pipeline. As we can see by the code, the Uploader and Backgroundsubtracter (described in sections 3.6.6 and 3.6.7) have buffers of same type and size and can therefore use swapping. Since the Backgroundsubtracter does not allocate memory with extra padding as described in section 3.3.4, but warping does, we need to copy the memory from one buffer to the other. Extra padding can be seen in the upper for-loop in the figure. Using 2D memory copies and names like pitch_in indicates that the images are aligned in memory to optimize bus-transactions between host and device.

```
// Copy from BackgroundSubtractor to Warper:
for (unsigned int j = 0; j < numCams; j++) {
  cudaMemcpy2D(pw->frames_in[j], pw->pitch_in[j], pb->d_inputFrames[j],
                           width*ICHANNELS, width*ICHANNELS, height, cudaMemcpyDeviceToDevice);
  cudaMemcpy2D(pw->masks_in[j], pw->pitch_masks_in[j], pb->d_foregroundMasks[j],
                           width, width, height, cudaMemcpyDeviceToDevice);

}

// Copy from Uploader to BackgroundSubtractor:
for (unsigned int j = 0; j < numCams; j++) {
  SWAP(pu->d_inputFrame2[j], pb->d_inputFrames[j], tmpNpp8u);
  SWAP(pu->d_playerPixels2[j], pb->d_playerPixels[j], tmpChar);
}
```

Figure 3.4: The two ways the controller switches image data between modules.

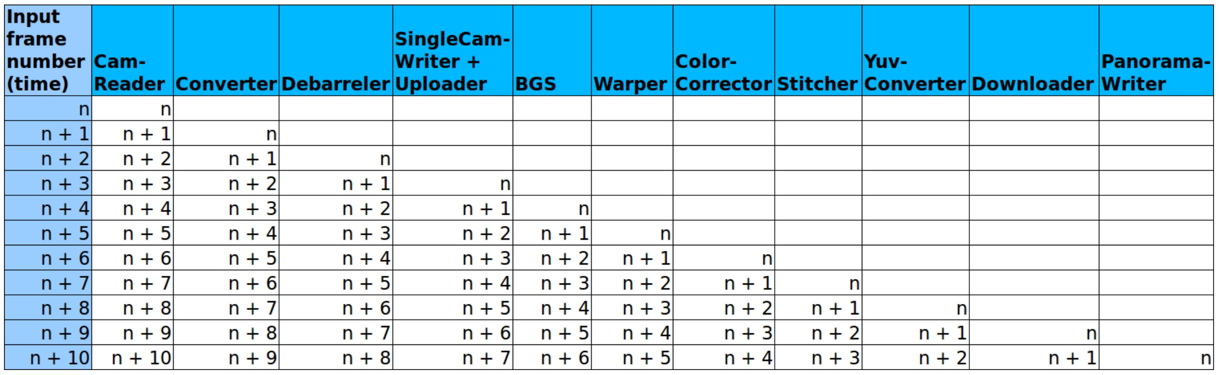
3.6 Pipeline Modules

To process frames in parallel and deliver a new frame every 33 ms, we have implemented our program as a set of modules that is controlled by the main thread (Controller in section 3.5.4). The modules works on individual memory buffers that represent the image data at a given state in the iteration. This solution makes it possible to process large amount of data fast and efficiently. It also makes it easier to change the panorama creation by adding more, or removing modules to create a more visually pleasing result. We will now present the different modules, explain what they do, and evaluate the results.

Before we go into the different modules, some common attributes for the module sections are listed below:

- At the start of the pipeline all module buffers and other data is allocated when initializing the classes.
- Unless stated otherwise, all modules run in a single thread.
- The real-time requirements for the modules are 33 milliseconds, since the cameras deliver 30 fps.
- A frame set is defined as 4 frames, one from each camera stream, all having the same timestamp.
- Each module has input buffers and corresponding output buffers.
- Module performance is presented in section 3.8.
- Results presented in the performance sections are from processing 1000 frames.

Figure 3.5 shows how a frame moves between the different modules. When the frame has passed all the modules the panorama is written to file. When a frame is read, we cannot deliver a panorama containing that frame after 33 ms. This frame will have a delay, but the pipeline will be able to deliver frames, after the first one is delivered, at our 30 fps rate.



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 3.5: Pipeline execution, how a frame moves through the pipeline.

3.6.1 Module Buffers

For keeping the module work to a minimum and optimize synchronization between the modules, most of them have a set of input and output buffers that can be seen in table 3.3. The buffer contains image data either stored as RGBA or as YUV dependent on where the modules are placed in the pipeline. The only modules not using both output and input buffers are the modules reading from the cameras and the ones writing the videos to disk.

3.6.2 Frame Buffer Delay

The ZXY-database stores the information it retrieves with a delay of approximately 2-3 seconds, and querying the database takes between 600-700 ms. Since we want information from the ZXY-database when creating the panorama, we need to have a frame buffer that can temporary store the image data. Given the ZXY delay, we make use of a frame buffer delay to make sure that we are working with ZXY data that is up to date. The buffer is placed between the Debarreler and the Uploader modules (discussed in sections 3.6.5 and 3.6.6) as we do not need the ZXY data when writing the single camera streams. To make sure that we have the ZXY data when we are processing the frames, the buffer contains 150 frames (5 seconds of video). This should the pipeline sufficient time to query the database so we can access the position data of the players being used in some of the modules.

Module	Host (CPU)	Device (GPU)
Reader	In: 4 x raw camera stream Out: 4 x YUV frame	-
Converter	In: 4 x YUV frame Out: 4 x RGBA frame	-
Debarreler	In: 4 x RGBA frames Out: 4 x RGBA frames	-
SingleCamWriter	In: 4 x RGBA frame	-
Uploader	In: 4 x RGBA frame	Out: 2 x 4 x RGBA frame Out: 2 x 4 x bytemap
BGS	-	In: 4 x RGBA frame In: 4 x bytemap Out: 4 x RGBA frame (unmodified) Out: 4 x bytemap
Warper	-	In: 4 x RGBA frame In: 4 x bytemap Out: 4 x warped RGBA frame Out: 4 x warped bytemap
Stitcher	-	In: 4 x warped RGBA frame In: 4 x warped bytemap Out: 1 x stitched RGBA frame
YuvConverter	-	In: 1 x stitched RGBA frame Out: 1 x stitched YUV frame
Downloader	Out: 1 x stitched YUV frame	In: 1 x stitched YUV frame
PanoramaWriter	In: 1 x stitched YUV frame	-

Table 3.3: Overview of the input and output buffers for the different pipeline modules.

3.6.3 CamReader - Reading Frames From The Cameras

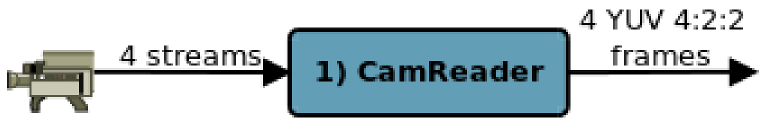


Figure 3.6: Camera reader module.

The CamReader module is responsible for retrieving frames from the four HD cameras. To do this the modules needs to initialize the frames streams and do other camera preparations so that the frames can be read and further processed in the pipeline. As mentioned earlier in the thesis, we receive frames in the color space YUV, more specifically using YUV422 at a rate of 30fps. Even though the cameras support pixel resolution up to 1296×964, the camera drivers does not. Therefore the frame size is at 1280×960 pixels. The module is also responsible for handling frame drops caused by the cameras or drivers. How these are solved is described in section 3.5.2.

Implementation Details

Since the CamReader is the module retrieving the image frames, it only utilizes one set of output buffers. The buffers, one per frame, stores the frames as YUV420 and passes the frames to the next module in the pipeline as seen in figure 3.6. The module uses 5 threads, one that executes the module itself and one for each camera stream.

The module first step is to initialize the cameras. With the help of their URI and video capture code from the Northlight library, we are able to set up streams for the cameras that allows us to get the frames and store them. After the initialization step that makes sure all cameras are running, we start reading from the four camera streams. To synchronize the streams, we give each frame a timestamp as well and keep track of the total number of frames read in the pipeline. Both the timestamp and frame number is later used for both debugging and decoding in other parts of the system.

The threads reads from the camera streams as long as the program is running and there is more video to capture, so there execution are not controlled in the same way as the other modules. Reading frames is done in the following steps while there is still more frames to read:

1. If the camera is up and running
   (a) If we retrieve a new frame within 33 ms, we catch the frame and copy it to the frame buffer so it can be stored and processed in the pipeline.
   (b) If we did not manage to retrieve the frame within 33 ms, we ignore that frame and try to catch the next one.
2. If the camera is not up and running, we check if all cameras are done reading. If that is the case, we close the streams.
3. Next, we give the frame a timestamp and update the frame number counter in the pipeline. Then, we signal the pipeline Controller so it knows that a new frame have been read and can be further processed.
4. Before continuing, the threads wait for each other to finish before the sequence is repeated.

3.6.4 Converter - Converting From YUV422 To RGBA

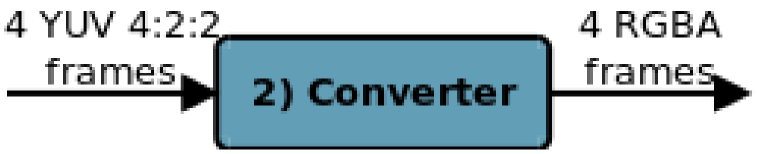


Figure 3.7: Converter module.

Do to reasons explained in chapter two, we convert the image data to the RGB color space, but unlike the prototype, we use RGBA. To maximize the CUDA performance you want to minimize the number of bus-transactions. Choosing to use RGBA over RGB improves the memory alignment, therefore also minimizing bus transactions. There is more data to store, but since the calculations simply ignore the alpha channel, we do not see any noticeable difference in the processing time of a 4-channel image compared to 3-channel image. The conversion itself is done on CPU and is performed before correcting for lens distortion in the pipeline. As with the prototype the conversion is done with Northlight.

Implementation Details

The module, seen in figure 3.7 works on four input buffers and four output buffers. The procedure is performed sequentially, i.e., the module only uses one thread. In order to convert from YUV422, a compressed YUV format to RGBA or RGB, we first remove the ”compression rate” from the existing frame data. This means that conversion between the two color spaces is done in two main steps:

1. Convert from YUV422 to YUV444
2. Convert from YUV444 to RGBA

3.6.5 Debarreler - Correcting For Lens Distortion

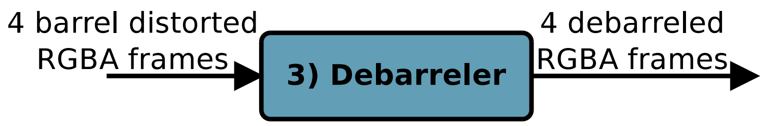
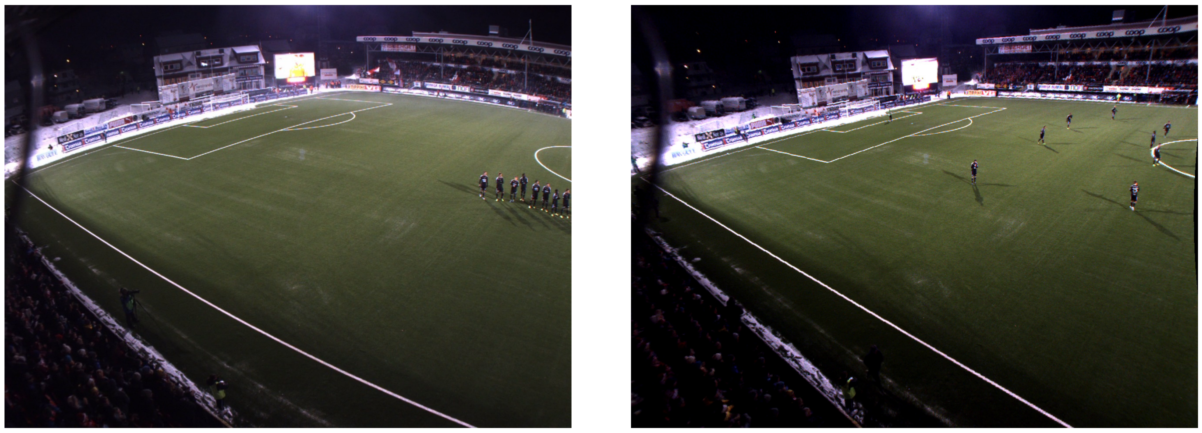


Figure 3.8: Debarreler module.

Correction for barrel distortion as described in section 2.7.3 is a process that is needed when using lenses with a wide angle of view. As mentioned in chapter 2, this is the case for Bagadus. To be able to work on the images and combine them with data provided by ZXY, we need to remove the barrel distortion caused by the camera lenses. For debarreling in the pipeline, we still use the same implementation described in the prototype presented in section 2.7. By applying OpenCV with debarreling coefficients and the individual camera arrays, we are able to debarrel the frames within our real-time requirement. Figure 3.9 shows a frame retrieved from the leftmost camera stream with and without barrel distortion. As we can see, the image to the right is corrected and have straight lines compared to the image on the left.



(a) Image before debarreling. (b) Image after debarreling.

Figure 3.9: Pipeline debarreling results.

Implementation Details

The module as seen in figure 3.8, takes four image buffers as input, and delivers four image buffers as output. To optimize the performance of the module, each frame works on its own dedicated thread, making the thread count for the module five. The debarreler is implemented on the CPU and works on images represented in RGBA. The homography applied to the OpenCV debarreler is created as a part of the system setup, and as with earlier pipeline versions, we still use Nearest Neighbour as the interpolation method. The debarreling is done with the following steps:

1. Set the values for the camera matrix and debarrel coefficients.
2. Applying the debarreling coefficients and camera matrix to remove barrel distortion in the image with the help of OpenCV.

3.6.6 Uploader - Move Data To The GPU

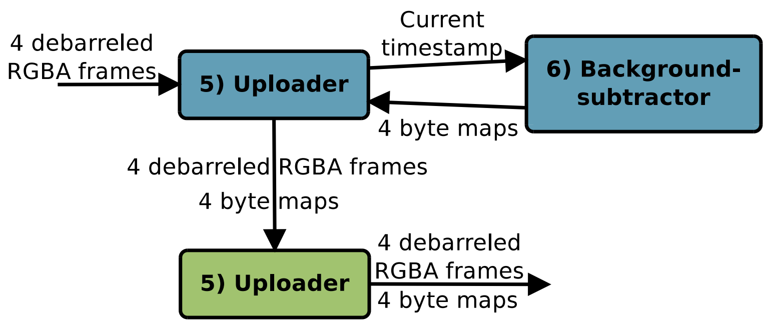


Figure 3.10: Uploader module.

As can be seen from the numbers in table 2.3, the CPU does not fulfill our needs for effective media processing. Now that we are done with removing barrel distortion and ready for the pipeline construction, we can move the frame buffers to the device. When uploading the images, we use double buffering and pinned memory for optimal performance. In addition to uploading the individual frames, the Uploader seen in figure 3.6.6 also has the job of initializing some metadata for the next procedure in the pipeline, the Background subtracter.

Implementation Details

The module takes four image buffers located on the host as input, and delivers four image buffers located on device as output. The Uploader also delivers the Background subtracter metadata as output. For uploading the data to GPU, the modules runs on a single CPU thread and execute the following steps:

- Synchronize the CUDA streams
- For each frame
  - Switch buffers to make use of double buffering

  - Copy image data to pinned memory
- If a Background subtracter is initialized
  - Generate metadata for the Background subtracter
  - Move it to device with asynchronous memory copies.

3.6.7 BackgroundSubtracter - Extracting Moving Objects

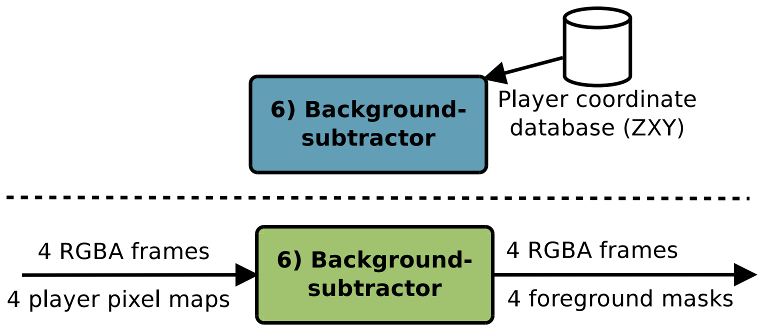


Figure 3.11: Background subtracter module.

The job of the Background subtracter (BGS) is to locate the foreground objects in each individual frame. In our case the objects are football players. The method is often used in surveillance, traffic monitoring and object tracking. In the pipeline the background subtracter is applied to simplify later work on the images since it (hopefully) locates the objects of interest. The module in our system is based on the algorithms presented in [46, 47] and is implemented on the GPU to keep its processing time below 33 ms. Although the BGS algorithm is executed on the device, some of the data used by the code is preprocessed in the Uploader, a CPU module. This preprocessing is done in order to use ZXY tracking information to minimize and simplify the workload of the algorithm. The algorithm output is a set of *masks* consisting of white, grey or black pixels, where white pixels indicates moving objects.

As can be seen in figure 3.11, the module divides its workload into two subcomponents;

- Algorithm initialization and creation of metadata (CPU, done in the Uploader).
- Running the background subtracter algorithm (GPU).

Implementation Details

The module takes four frame as input and gives background subtracter masks and the input frames as output. The masks generated by this module is transfered between the next modules in the pipeline, but only used in the panorama stitcher.

Algorithm Initialization: This part of the background subtracter creates the metadata used to simplify the algorithm with ZXY information. The simplification involves creating byte maps that indicates where players are located in the pitch:

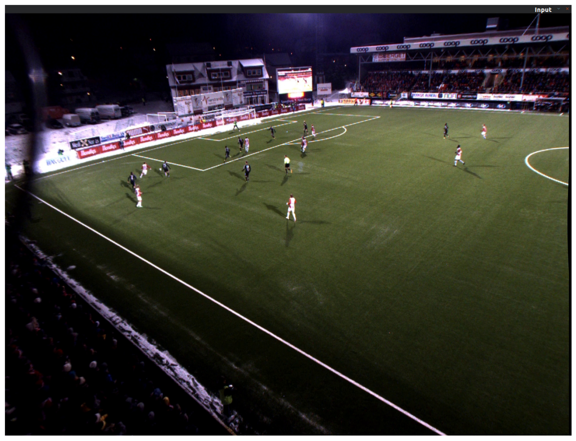
- The algorithm extracts the player positions from the database.

- A byte map of frame size 1280×960 is created.
- with the help of the position data, the pixels where a player will be located is marked by 1. The other parts of the field = 0. To make sure that we are able to capture the player in the pixel lookup map, we also set pixels close to the players to 1.

Background Subtracter:

- For all the pixels in all images, find out if the pixel is background, foreground or shadow.

The figure 3.12 shows the result when applying the BGS on image a frame from camera 1. By feeding the algorithm with player tracking data from ZXY, we are able to remove the noise from the output. See the work of Marius Tennøe [48] for further implementation details.


(a) Input frame, camera one.

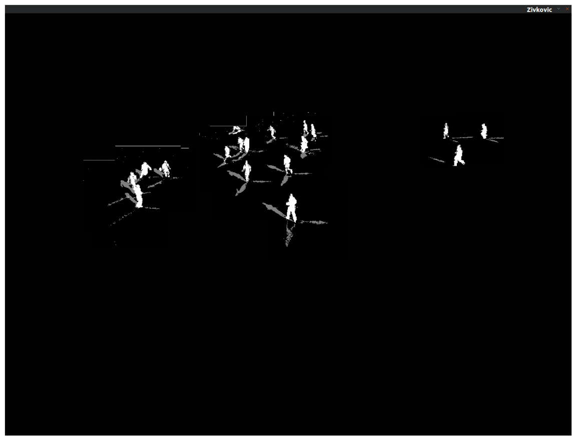

(b) ZXY BGS, camera one.

Figure 3.12: Background subtracter results with ZXY player data.

3.6.8 Warper - Giving All Frames The Same Plane

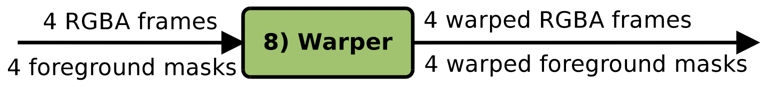
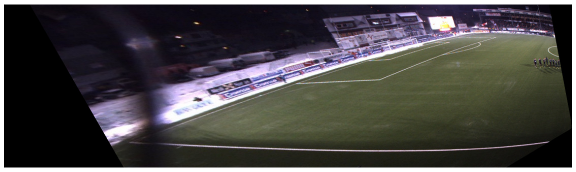


Figure 3.13: Image warping module.

The Warper, illustrated in figure 3.13, is responsible for aligning all images to the same plane. The task is done on the GPU with the help of CUDA NPP library described in section 3.3.4. The module aligns the planes for the image frames and the masks created by the BGS. The module is, except for running on the device, implemented in a similar way to the warper component in the prototype (described in section 2.7.3). In order to simplify the tasks done by the Colorcorrector, the warper is executed after the BGS in the pipeline. The result from warping camera one can be seen in figure 3.13. The resulting image is warped to fit the position of camera two and as we can see that the image is stretched giving us more data to process for the rest of the pipeline. The same stretching effect is also apparent in frames from camera three and four.


(a) Camera one, not warped.

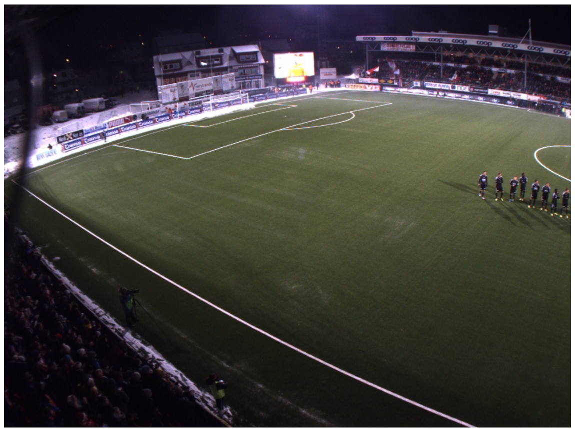

(b) Camera one after warping is applied.

Figure 3.14: Warping camera one.

Implementation Details

The module gets as input four image buffers and four buffers containing the BGS player masks. As output it delivers the warped result of both the frames and the masks. The implementation is simple and except for some initial setup, only involves calls to the NPP library function nppiWarpPerspective().

The Warper does all its work in a sequence, using only one thread. As with the prototype, NN is used as the interpolation method and the transformation matrices are generated at system setup.

3.6.9 ColorCorrecter - Correct Color Differences In The Frames

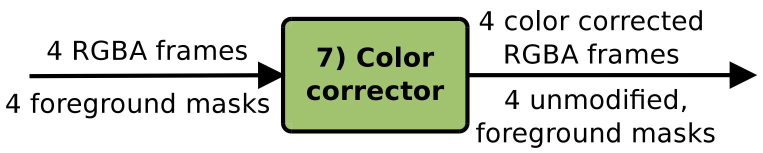


Figure 3.15: Color Correction module

To remove and smoothen the differences in color and luminance values, we have added a color correction module to the pipeline. The module makes use of the overlapping regions between images to create color coefficients that represents the overall color differences in two adjacent images. The current color correction algorithm is a simple method that sequentially remove obvious color mismatches in the images. The algorithm does not remove all the color and luminance differences in the sequence completely and struggles to give a good output in bad lightning conditions. The module is implemented on the GPU to improve its performance.

Implementation Details

The use of input and output buffers can be seen in figure 3.15. As input it gets four image buffers, and the four image masks, working only on the images. As output, the buffer delivers the color corrected image data and the four image masks. The algorithm finds individual color coefficients, one for each color channel for the entire sequence. These are then applied to the image to give the end result. The Colorcorrector uses one thread and does the following step to minimize color and luminance differences in the images:

1. As an initial step, set all color coefficients of the leftmost image to 1.0.
2. For the rest of the sequence
   (a) Find its individual color coefficients by comparing the image overlapping region, with the overlapping region of the image to the left in the sequence.
   (b) Correct the image by applying the color coefficients for each image.

3.6.10 PanoramaStitcher - Generating The Final Panoramic Image

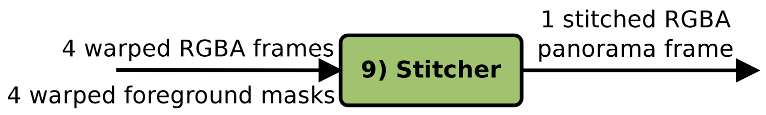


Figure 3.16: Stitching module.

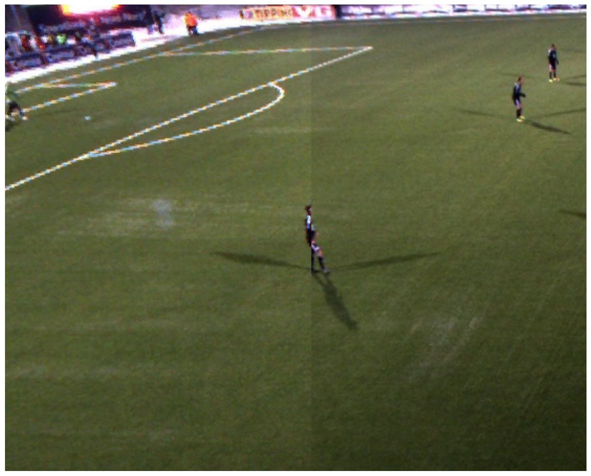
To stitch the frames into a panorama, we have implemented a stitching module that instead of using static cuts that creates artifacts and clearly visible edges, uses a dynamic seam to improve the visual end result. With the help of predefined overlapping image regions the method is able to find a seam between the images that avoids moving objects. For this task, we use the masks generated from the Background subtracter in addition to doing a graph search in the overlapping region. Dijkstra [49], a shortest-path algorithm is used for the graph search, but the implementation is somewhat different and simplified to suit our scenario. For a more elaborate explanation of PanoramaStitcher, see the thesis of Espen Oldeide Helgedalsrud [50].

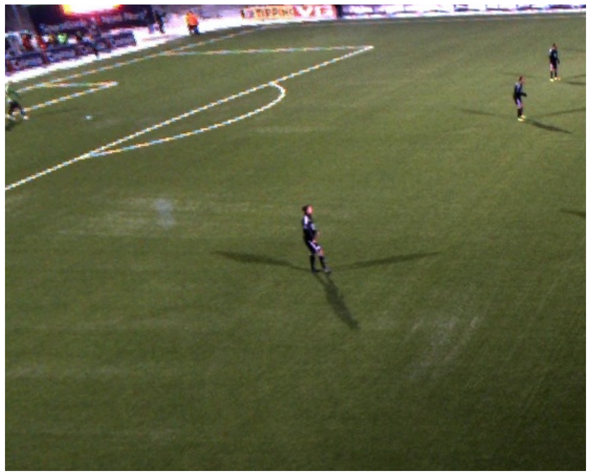
Implementation Details

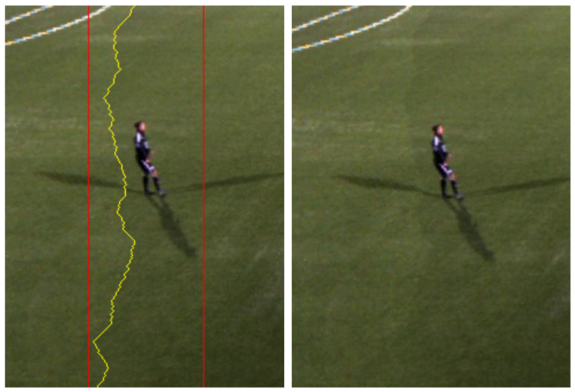
The stitcher gets four frames as input, one from each camera stream, as figure 3.16 illustrates. All the frames are taken at the same time and should have the same time stamp. It delivers one panorama frame as output.

In order to create a stitch seam that avoids the moving objects on the field, we start by defining a region in the overlapping area of the images where the seam will go through. In this predefined area, the pixels is represented as nodes and we perform a graph search that moves from the bottom of the image to the top. To improve the performance of the graph search, we limit the movement of the graph to always be upwards. When searching for the next node, we have three choices, go straight up, go upwards to the left, or upwards to the right.

A weight function is calculated by comparing the absolute color differences between the corresponding pixels in the overlapping regions. The seam is then found by running a simplified version of Dijkstra in the search region to find the path with the smallest cost. The final graphs represents where the seams will run through the panorama. As a final step, we copy the image data from the separate frames into the final panorama. Visual results from the PanoramaStitcher and the PanoramaColorCorrecter be found in figure 3.17. Here, we see the effects of the dynamic stitcher and color correction on old footage from the first Bagadus prototype. The top images illustrates the differences between the old and the new pipeline and as we can see, the player is not cut in half and the stitch seam is not visible. The bottom images shows the different paths chosen by the dynamic stitcher with and without color correction.


(a) The original stitch pipeline in [24] and [23]: a fixed cut stitch with a straight vertical seam, i.e., showing a player getting distorted in the seam.


(b) The new stitch pipeline: a dynamic stitch with color correction, i.e., the system search for a seam omitting dynamic objects (players).


(c) Dynamic stitch with **no** color correction. In the left image, one can see the seam search area between the red lines, and the seam in yellow. In the right image, one clearly see the seam, going outside the player, but there are still color differences.

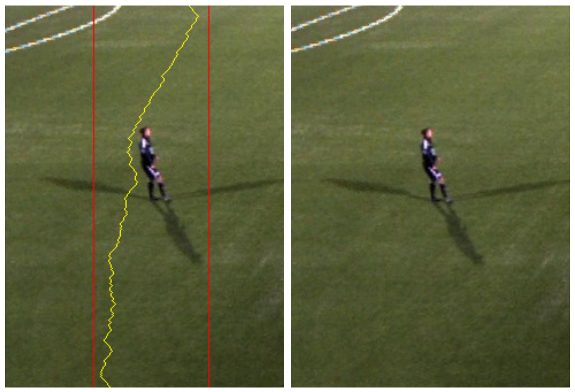

(d) Dynamic stitch **with** color correction. In the left image, one can see the seam search area between the red lines, and the seam in yellow. In the right image, one cannot see the seam, and there are no color differences. (Note that the seam is also slightly different with and without color correction due the change of pixel values when searching for the best seam after color correction).

Figure 3.17: Stitcher comparison - improving the visual quality with dynamic seams and color correction.

3.6.11 YUVConverter - Convert Panorama From RGBA To YUV420

1 stitched RGBA panorama frame
10) YuvConverter
1 stitched YUV 4:2:0 panorama frame

Figure 3.18: GPU converter module.

This process is needed so that we can convert the video frames to H.264 format in the encoder. The operation is done on the GPU since the process is highly parallelize-able and can be done much faster one the device rather then on the host. In the result section you can also see that this process takes to much time for the CPU, and therefore needed to be done with the help of CUDA. The module converts from RGBA format to YUV420p.

Implementation Details

The converter, seen in figure 3.18, takes as input a panorama frame, and gives the converted frame as output. To convert from the frames from RGBA to YUV420, we need to divide the implementation into different two main steps.

1. With the help of NPP, we first covert from RGBA to a YUV format without compression, YUV444. The result is stored on a temporary buffer located on the device.

2. The last part of the conversion is from YUV444, which is a packed format, to YUV420p, a subsampled planar YUV format. For converting, the following steps are applied to the image data:

   - Since Y is represented with the same amount of bytes, we copy these to the beginning of the YUV420p output buffer.

   - For the remaining data; U and V need to be subsampled as described in section 2.5, where you only use 1 out of 4 bytes to represent the two chrominance levels. The pseudo code for moving these last two values in the final image can be seen in algorithm 1. Here, source_buffer is the image sampled as YUV444 and destination_buffer is the image sampled as YUV420p. N is the maximum number of bytes in the source image.

Algorithm 1 Pseudo code for finding correct indexes and setting the U anv V values in the converted image

```
$intidx = blockIdx.x * blockDim.x * threadIdx.x;$
int pos;
**if** idx < N **then**
    **if** $((idx \mod 2) == 0) \quad \&\& \quad (((\frac{idx}{width}) \mod 2) == 0)$ **then**
        $pos = \frac{\frac{idx}{width} * \frac{width}{2}}{2} + \frac{idx \mod width}{2};$
        destination_buffer[pos] = source_buffer[idx];
    **end if**
**end if**
```

3.6.12 Downloader - Move Panorama Back To Host

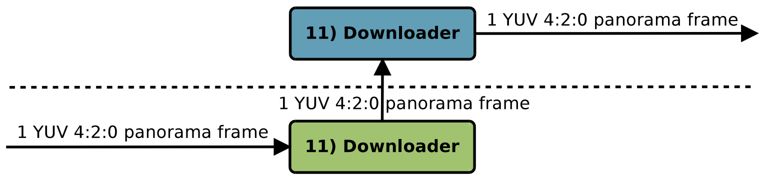

11) Downloader
1 YUV 4:2:0 panorama frame
1 YUV 4:2:0 panorama frame
1 YUV 4:2:0 panorama frame
11) Downloader

Figure 3.19: Download module.

Downloading the final panorama image from device to host is the final stage before writing the frame to file. To keep both the PanoramaWriter and Converter modules in real-time, non of them downloads the data back to the CPU. Since the panorama generation is finished and there is no need to do more work on the GPU, we move the frame buffer back to host.

Implementation details

This component illustrated in figure 3.19 makes use two data buffers. One of them has the data that you want to move back to host, and the other one contains free space on the host where the image will be placed. It is just a simple cudaMemcpy() call.

3.6.13 PanoramaWriter - Writing Panorama To Disk

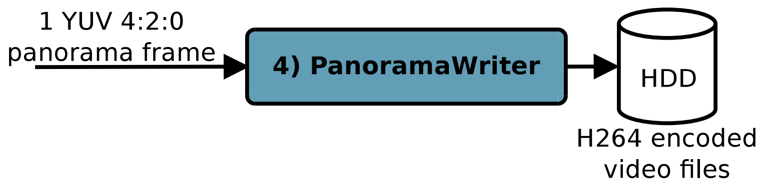


Figure 3.20: Writer module.

This module is the final step of the panorama creation. It writes the panorama frame to disk in 3 second (90 frames) chunks and has the time stamp of the first frame in its filename. The files are written to a folder named after the recording start time. Encoding the frame to H.264 are also a part of the writer. See sections 2.6 and 3.5.1 for an explanation on encoding in general and encoding in the panorama pipeline.

Implementation Details

The module takes one panorama frame as input and writes it to disk as seen in figure 3.20. The algorithm used for writing can be seen in figure 2 and is repeated for every panorama frame.

Algorithm 2 Panorama Writer pseudo code

```
if (NumberofFrames $\mod$ ChunckSize(90)) == 0 then
    if current frame is not the first(0) then
        close the previous file
    end if
    Open new file
    Write H.264 header.
end if
Encode the current frame data
Write the encoded data to disk
```

3.6.14 SingleCamWriter - Writing Frames To Disk

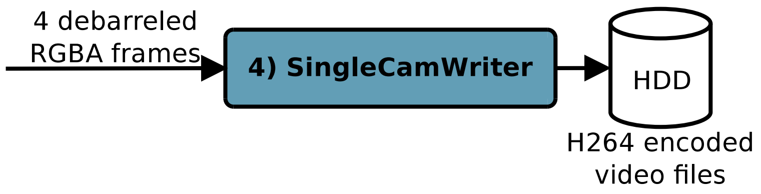


Figure 3.21: Single camera writer module.

This module writes the four single camera videos to file in 3 second chunks. The four videos are stored in individual folders and the module uses four slave thread for writing the files in parallel. The videos are written as soon as the frames have been debarreled. Since the single camera frames are only 1280x960 pixels the CPU is fast enough for converting from RGBA to YUV420. Therefore, we do not need a individual module in the pipeline to convert the camera frames.

The single camera streams are encoded to H.264 before they are written to file and use the same encoding settings as the PanoramaWriter in section 3.5.1.

Implementation Details

The module takes four frames buffers, one from each camera stream and is illustrated in figure 3.21. The implementation details are very similar to the Panorama writer. The only difference is that when writing the different camera frames to file, we also convert the files from RGBA to YUV in this module. This means that to simple steps gets added to the Pseudo code from the Panorama Writer:

- Copy the data from the pipeline buffers to the buffers used by the module. (As for the modules used in the panorama generation this happens in the controller, but since this not directly effect the pipeline in any way, this is done by the module itself).
- Convert from RGBA to YUV420 in order to be able to convert to H.264.
- Do the same as described in the PanoramaWriter pseudo code.

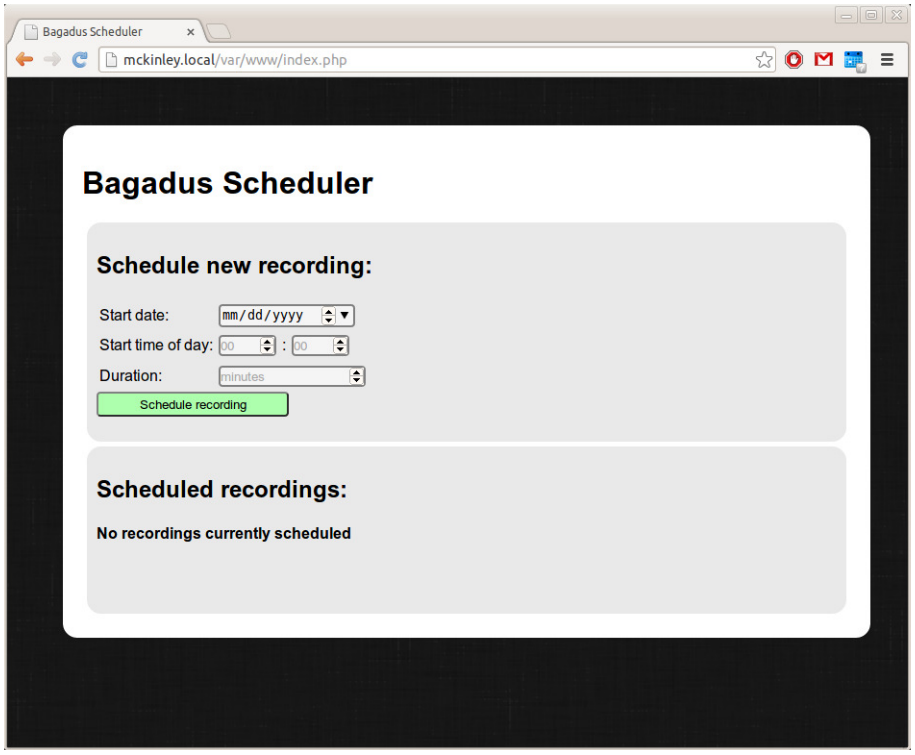
3.7 System Web Interface

For easy use of the system, we have created a simple web interface that allows users to schedule recordings from a Internet browser. The interface allows you to start recordings by specifying session start time and duration. There is also options for stopping ongoing and later scheduled recordings. The interface is simplistic so that it requires no background knowledge from the user in order to start a recording session. When several sessions have been scheduled they are listed at the bottom of the application and the user is free to remove or stop one or more of these recordings.

The interface is shown in figure 3.22. When scheduling a recording you specify the start date, time, and the duration in minutes. When the recording is scheduled, it shows up in a table below in the interface. Here, we list the recordings with the start time as well as the duration. By pressing the red stop bottom on the left side, the recording is removed from the list. When scheduling a recording, you cannot set to record at the same time then another already scheduled recording since we can only have one stream from the same camera at once.

Implementation Details

- When scheduling a recording, the Panorama pipeline is launched with the help of a PHP script. Using the recording time as parameters, the pipeline runs until the recording is done.
- The list showing the scheduled recordings is retrieved using the Unix commands *grep* [51] and *ps* [52].
- The interface runs on a Apache server located on the same machine as the panorama pipeline.
- To stop an ongoing recording, we end the process with the help of the Unix *kill* [53] command.



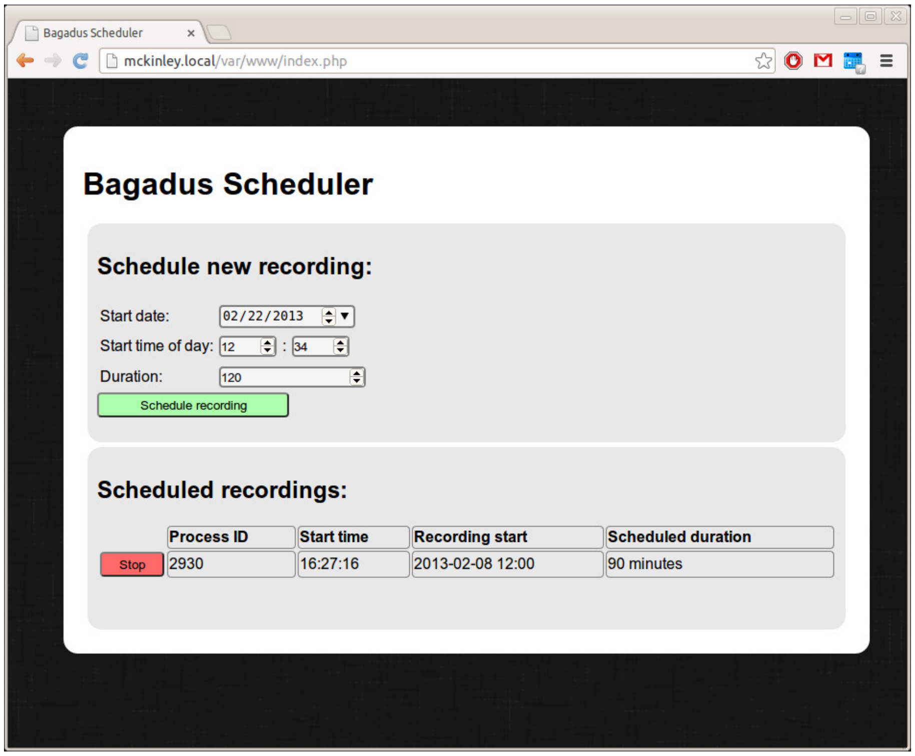
Bagadus Scheduler

Schedule new recording:

Scheduled recordings:

No recordings currently scheduled

(a) Interface when there is no scheduled recordings.



Bagadus Scheduler

Schedule new recording:

Start date: 02/22/2013
Start time of day: 12 : 34
Duration: 120

Scheduled recordings:

	Process ID	Start time	Recording start	Scheduled duration
Stop	2930	16:27:16	2013-02-08 12:00	90 minutes

(b) Interface with scheduled recordings.

Figure 3.22: The Bagadus scheduling interface.

3.8 Pipeline Performance

In order to get an overview of the pipeline performance and how it depends on the different hardware, finds its limitations and bottlenecks, we have executed a series of tests in different hardware environments.

This process is important for further scaling of the system as it can give indications of what needs to be done when adding new modules, using cameras that deliver higher quality images or if we want the system to take on more of the general Bagadus workload. We will in this section of the chapter present our findings and our test results from the different modules and the pipeline as a whole.

3.8.1 Pipeline Visual Results

Although the main reason for improving the first prototype pipeline was to improve the computations and make it deliver frames in real-time, we have also implemented modules that with correct camera settings and a good camera setup will create an improved visual panorama.

A comparison between the output of the first panorama program and the one presented in this chapter can be seen in figure 3.23. As can be seen, the old panorama shows a better output regarding color and luminance levels. This is caused by the time of year and the good lighting conditions at our test bed when the that video was recorded. To show output from the new pipeline with some modifications to the camera settings, we have added three panoramas using different camera settings to figure 3.23.

As can be seen in the images, the dynamic seam avoids players and the color correction manages to remove the seam between the middle frames. Do to image alignment that is not optimal, the camera settings used and different exposure times when taken the footage, the differences between the frames are to the extend where the color correction algorithm is not able to remove all the differences in color and luminance. By improving the camera calibration, its settings and synchronizing the exposure times, we should get a better visual result.

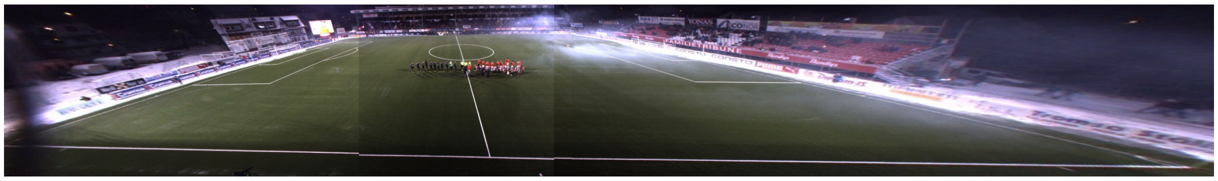
Figure 3.23(b) shows a panorama where the exposure times get to high. In addition, the white balancing is set at startup leading to the panorama output being to bright. Enabling automatic white balancing can be seen in figure 3.23(c). Currently, we are using and experimenting with auto exposure and the results are presented in figure 3.23(d). Improving these camera settings is ongoing work and therefore the panorama output is not in the current state visually impressive. Also note that the output form the old pipeline in figure 3.23(a) were recorded in the fall, while the panoramas from the improved pipeline where taken at early spring, therefore the lighting conditions are quite different.

3.8.2 Test Environments

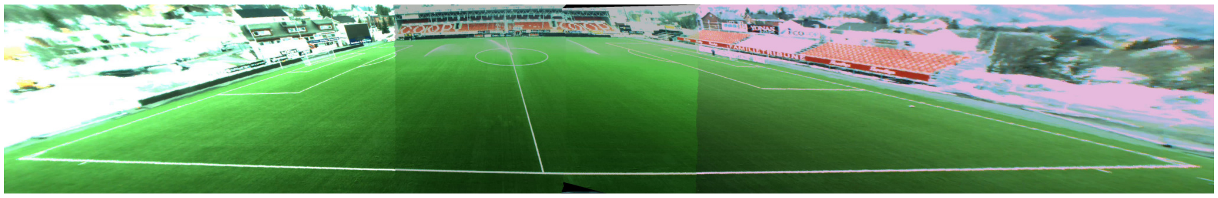
Here follows a presentation of the different hardware specifications used when performing system tests. More details on the computers and the different GPUs can be seen in appendix A.

The computers used in the testing have the following specifications:

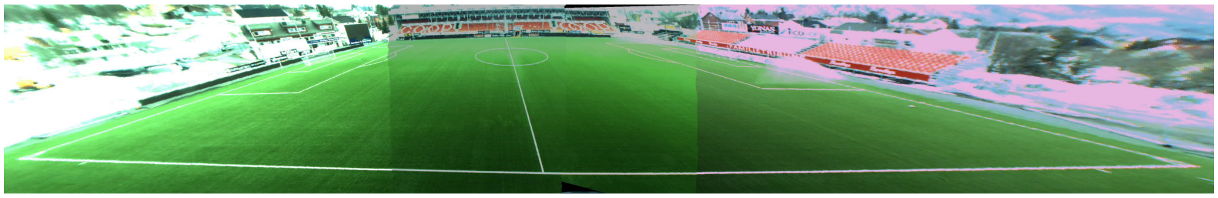
- Test machine 1(TM1 in A.1) have an Intel Core i7-3930K processor (with frequency overclocking) and an nVidia GeForce GTX 680 GPU.



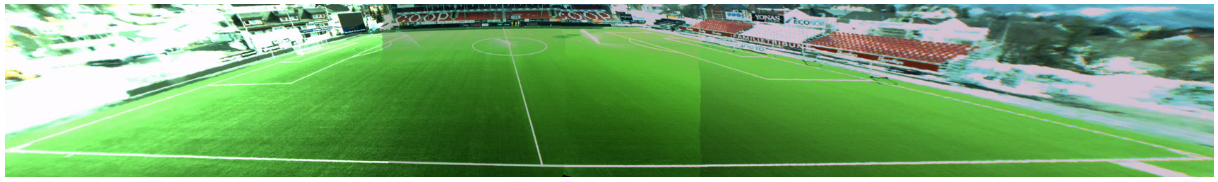
(a) Old pipeline output.



(b) Improved pipeline output, locked exposure times, locked white balance.



(c) Improved pipeline output, locked exposure times, automatic white balance.



(d) Improved pipeline output, experimental auto exposure enabled, locked white balance.

Figure 3.23: Old vs. improved pipeline output. Note that the panorama from the first prototype where recorded with manually set exposure settings.

- Test machine 4(TM4 in A.4) have an i7-3930K Intel processor with 16 cores using a nVidia GeForce GTX 480 GPU.

The GPU specifications can be seen in table 3.4.

GPU	Architecture	CUDA compute	Bus Support	Memory
GTX Titan	Kepler	3.5	PCI Express 3.0	6144MB
GTX 680	Kepler	3.0	PCI Express 3.0	2048MB
GTX 580	Fermi	2.0	PCI-E 2.0 x 16	1536MB
GTX 480	Fermi	2.0	PCI-E 2.0 x 16	1536MB
GTX 280	N/A	1.3	PCI-E 2.0	1GB

Table 3.4: Specs for different GPUs.

Frame drops

When presenting frame drops in a execution there is some differences dependent on the test environment:

Frame drop handling disabled:
: When testing and showing tables indicating the frame drops, these are the number of dropped frames in the pipeline, in addition the pipeline processes 1000 frames. In other words, when the table shows 4 dropped frames in the pipeline and 4 dropped by the cameras, this means that the pipeline processed or tried to process in total 1008 frames.

Frame drop handling enabled:
: When indicating frame drops with the frame drop handling functionality enabled, the pipeline will only read and process 1000 frames regardless of how many frames lost. This means that if you have a similar scenario as the example above, dropping 8 frames means the pipeline processed or tried to process 1000 frames.

3.8.3 Pipeline Demo Improvements

The Bagadus prototype presented a panorama pipeline that was based on OpenCV code and only running on the CPU. As presented above, the pipeline installed in Bagadus today divides its workload between the CPU and GPU. The algorithms have been optimized and simplified to suit our scenario. The diagram in figure 3.24 shows the differences in average between the pipeline used in the prototype and the current installation at Alfheim running on TM1. As we can see, the conversion between YUV422 and RGBA, and the camera calibration (debarreling) is not in the figure. For these modules, the implementations remains the same between the two versions, and therefore the timings also remains the same.

For the image warping, where the current implementation is using the NPP library, we see a improvement in orders of magnitude. The warper running on GPU also does more work, as it warps the images and the masks made by the BGS. The Stitcher is the module that have the biggest changes in implementation, and the best time improvements of all the modules. From just having a static cut, the stitcher now does much more calculations to find a seam that avoids moving objects. (Note that as mentioned earlier in the thesis, the CPU implementation of the static stitcher is not optimized). The Converter is real-time both in the prototype and the new version, but due to additional workload the new pipeline added to the CPU, the timings was no longer beneath our threshold. Therefore, we made a implementation on the GPU that with the help of the NPP library is below our threshold. As the diagram shows, it uses under half the time of the prototype version.

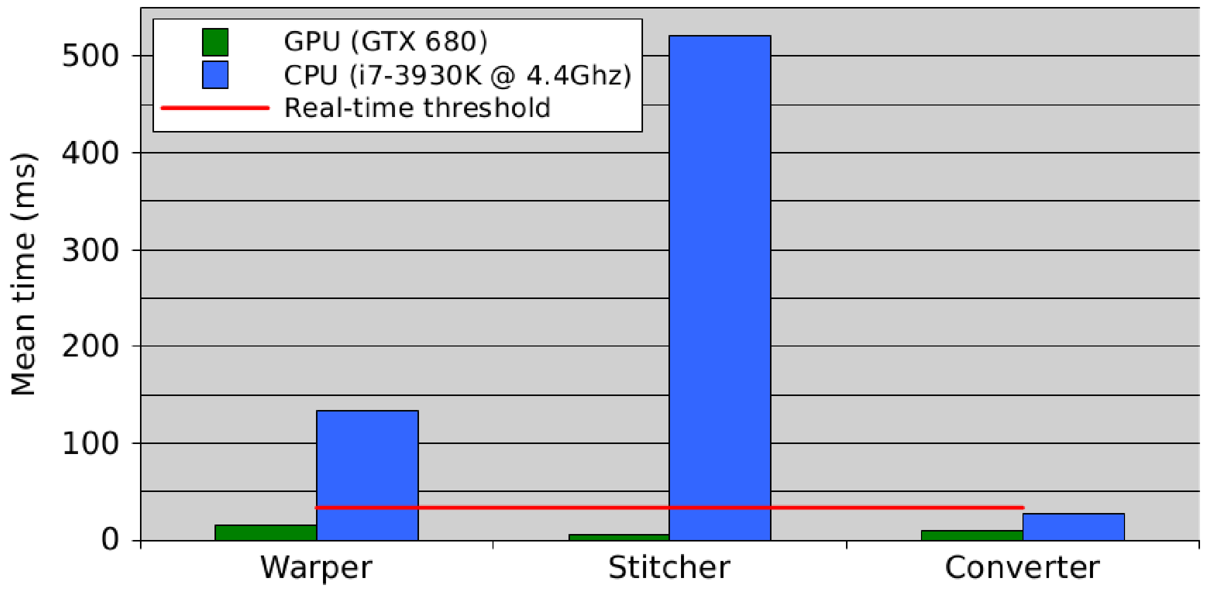


Figure 3.24: Pipeline improvements. A Comparison between the first prototype and the current installation of the system. Timing results in green are results from the old implementation (CPU), while the blue indicates the real-time panorama implementation (GPU).

3.8.4 Comparison Between CPU And GPU

To give an indication of the importance of the GPU in the pipeline, we have compared some of the modules that we have implemented on both host and device running on TM1. As we can see from the histogram in figure 3.25 all the modules benefit greatly by executing its code on the GPU. The BGS shows this well as it is able to process the same amount of data in under half the time it takes on the CPU.

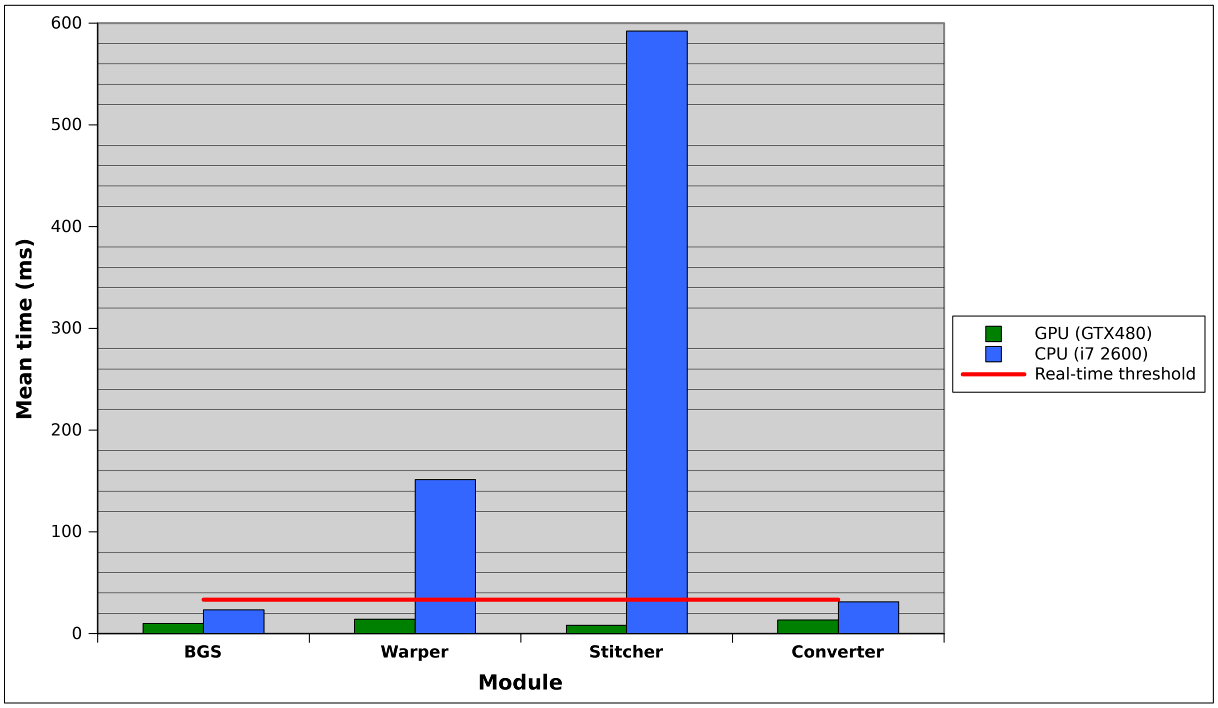


Figure 3.25: Comparison between CPU and GPU implementations. Timings in green are from the real-time panorama pipeline. Blue indicates timings from the first Bagadus prototype (CPU).

3.8.5 Alfheim Installation

Before the pipeline can be implemented as a Bagadus component, it needs to run in real-time, processing and delivering frames every 33 ms. Figure 3.26 shows pipeline timings from the machine installed in Tromso, TM1. Comparing the figure with the illustrations from figures 3.3 and 3.5 tells us that if each module runs in real-time, the pipeline as a whole runs in real-time. This is do to the fact that the modules runs in parallel and therefor takes on a new frame set every 33 ms.

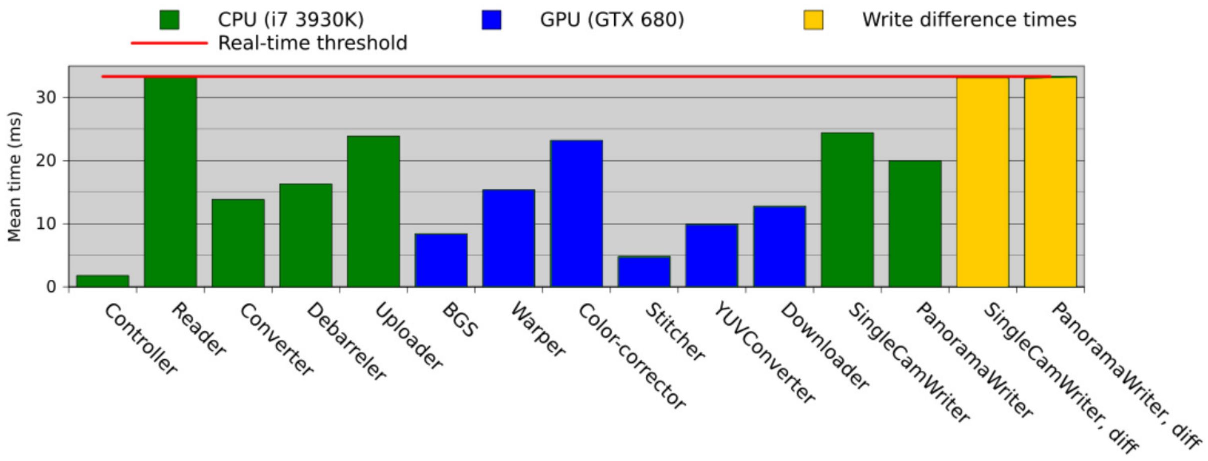


Figure 3.26: Alfheim installation, pipeline execution results.

Modules running on the CPU are marked in green. As the plot shows, the only module taking up time close or equal to the threshold is the Reader. As mentioned before, the Reader will never use less then 33 ms since the camera frame rate is set to 30 fps. The Uploader is, except from the Reader and writers, the module that is most time consuming. This is do to the workload it does for the BGS. The calculation of metadata for the BGS requires more processing power and time than uploading the four frames to the GPU. Note that the Debarreler also is quite time consuming. The current implementation is still using OpenCV, and therefore is not yet optimized. GPU modules are marked in blue and are all processing the frames within our threshold.

Pipeline End-To-End Delay

Each frame has an average pipeline end-to-end delay of 5.33 seconds as long as the pipeline modules is able to process the data within our threshold. The 5.33 seconds is based on the number of modules in the pipeline and the current frame buffer delay. Since each module uses 33 seconds between starting executions, each frames is processed for number of modules (10) $\times$ threshold (33 ms) + 150 frames delay = 5.33 seconds. Being the case that the end-to-end delay of each frame is based on the module count, this also means that increasing the number of modules will mean a similar increase in the end-to-end frame delay. If we where to remove the ZXY dependencies in the pipeline, the initial delay will only be 0.33 seconds, i.e., the time it takes for a frame to be processed by all the modules.

Write Differences

The write differences in yellow tells us the time between when two frames where written to disk, i.e. the difference in time between frame f_i and f_{i+1}, where i represent the frame number. Since the frame rate is 30 fps, the writers will only receive frames every 33 ms, which in turn means that the time is controlled by the frame rate of the cameras, and the writers never will be able to write to disk at a higher rate then the pipeline can read the frames. The pipeline implementation can never speed up the writers, however it can slow them down. If the pipeline frame processing is slow, the writer times will also increase. In other words, if a module in the pipeline takes up more then its allowed time, (and frame drops are not activated) the writer will not be able to process a new frame, and the difference time will increase. This tells us that the writer differences are a good indication on whether the pipeline processes the frames in real-time or not.

The scatter plot in figure 3.27 shows the different frame timings running 1000 frames. As we can see there are small spikes eminent. These are likely caused by OS calls, interrupts or disk access. Note that the panorama writer, indicated in blue, does not start writing its frames before approximately 150 frames have been processed. This is do to the frame buffer described in section 3.6.3. Although some of the timings are above 33 ms, as long as the average time of the write differences is within our threshold, the pipeline can be executed in real-time.

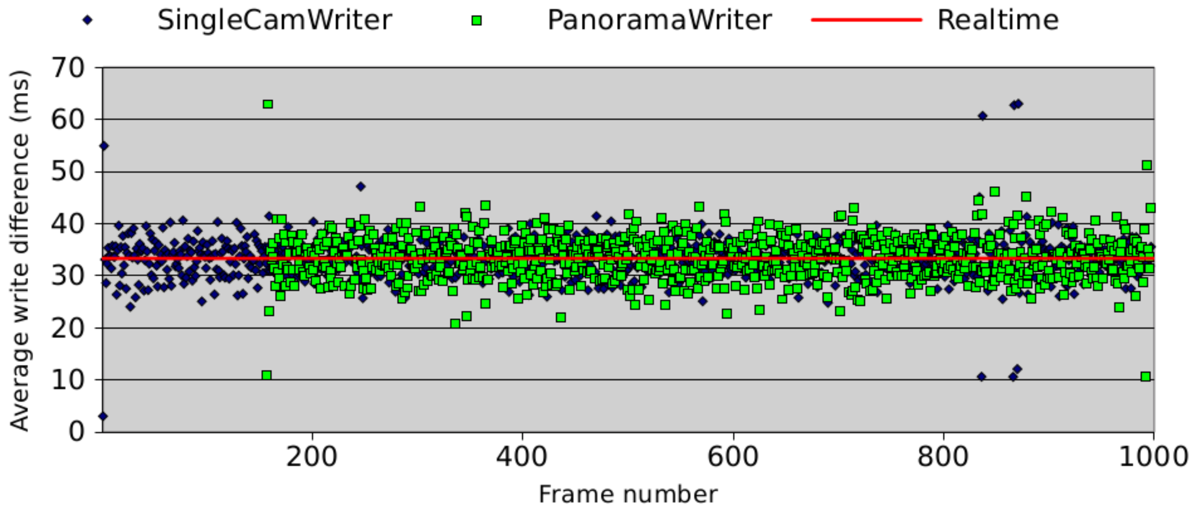


Figure 3.27: Alfheim installation, individual read and write times (per 1000 frames processed).

3.8.6 Different GPU Architectures

Diagram in figure 3.28 shows how the pipeline running on different GPU architectures. The GPU specifications can be found in Appendix B, tables A.5 and A.6. To see how the current pipeline performs with different GPUs, we have tested the pipeline on five architectures having different CUDA compute capability. The tests are executed on TM1, but by using different devices. As can be seen in the diagram, the results are as expected. GTX280(A.5), with compute 1.3, does not manage to process our amount of data within the threshold goal. The main reason is that GTX280 does not support concurrent CUDA kernel execution. The rest of the GPUs, using the either the Fermi [54] or Kepler [55] architecture and having a compute higher or equal to 2.x/3.x, all support concurrent executions of CUDA kernels. GTX480 is the oldest of the Fermi cards, but still manages to process the pipeline workload within our threshold. As can be seen in the diagram, GTX580 and GTX680 timings are close to equal, and only slightly lower than GTX480. GTX Titan gives as expected the best result when comparing the different GPUs. Both the Kepler cards, GTX680 and GTX Titan use PCI Express 3.0 giving 16 GB/s bandwidth in each direction, while GTX480 and GTX580 has an effective bandwidth of 8 GB/s in each direction. This indicates that the card bandwidth is not a bottleneck in the pipeline. To confirm our suspicions regarding bandwidth, we calculate the estimated data uploaded and downloaded in the pipeline per second:

$$Uploader:$$

$$(1280(frameWidth) \times 960(frameHeight))$$

$$\times 4(RGBA) \times 4(numberofframes) \times 30(fps) \approx \textbf{737 MB/s}$$

$$Downloader:$$

$$(6742(panoramaWidth) \times 960(panoramaHeight)$$

$$\times 1.5(YUV420p) \times 30(fps) \approx \textbf{291 MB/s}$$

As the calculations indicates the bandwidth is now where near its limits. For the Uploader, we have approximately 15.3 GB/s of free bandwidth and for downloading, we have 15.7 GB/s free.

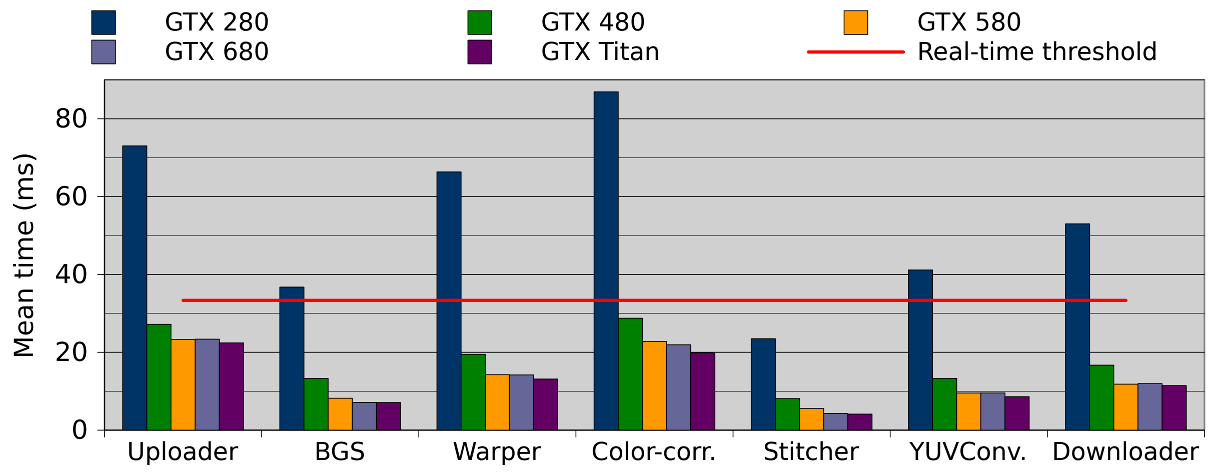


Figure 3.28: GPU architectures, pipeline execution results.

3.8.7 CPU Core Speed

Due to the large amount of threads executing in the pipeline, the program needs to run on a CPU that fulfills its requirements. To find the CPU requirements, we have done testing to see how the pipeline works with different clock rates. A higher clock rate gives better CPU performance, if the program are able to make use of the additional resources, i.e., the higher the clock rate, the more instructions. The tests are performed using TM1, by over clocking the CPU. As seen in the figure 3.29, all the CPU modules presented here are below the threshold for all the different clock rates and can conclude that tuning the clock rate has a small impact.

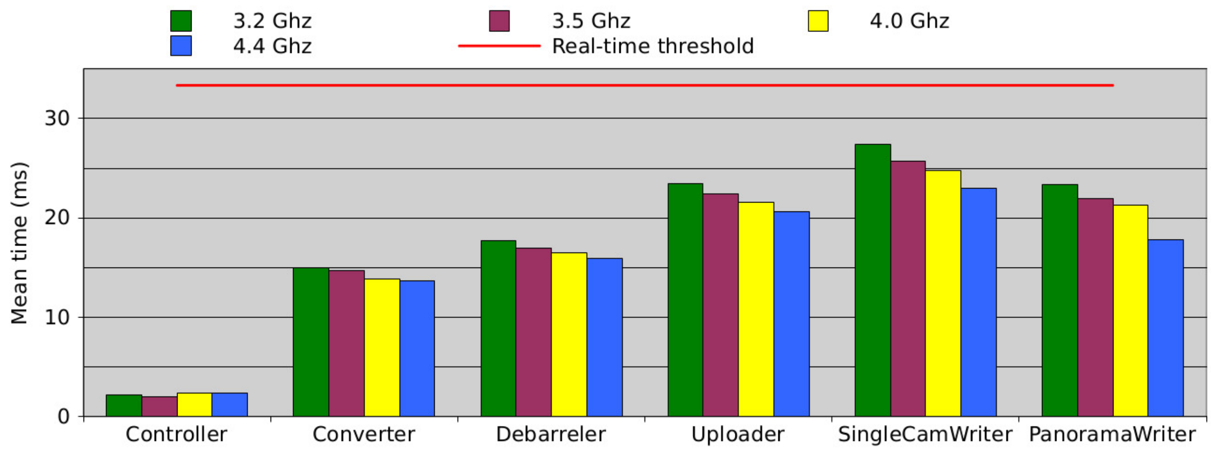


Figure 3.29: CPU Core speed, module times executing on different clock rates.

Write Differences

The clock rate differences, although small, has an impact on the writer differences.We can see that when setting a clock rate below 4.0 Ghz, the timings are slightly above our threshold.

This means that the pipeline will have more frames that uses processing times above the threshold. When the occasional OS spikes and interrupts happen, with a low clock rate, this will take more processing power from the pipeline, i.e., causing more frame drops.

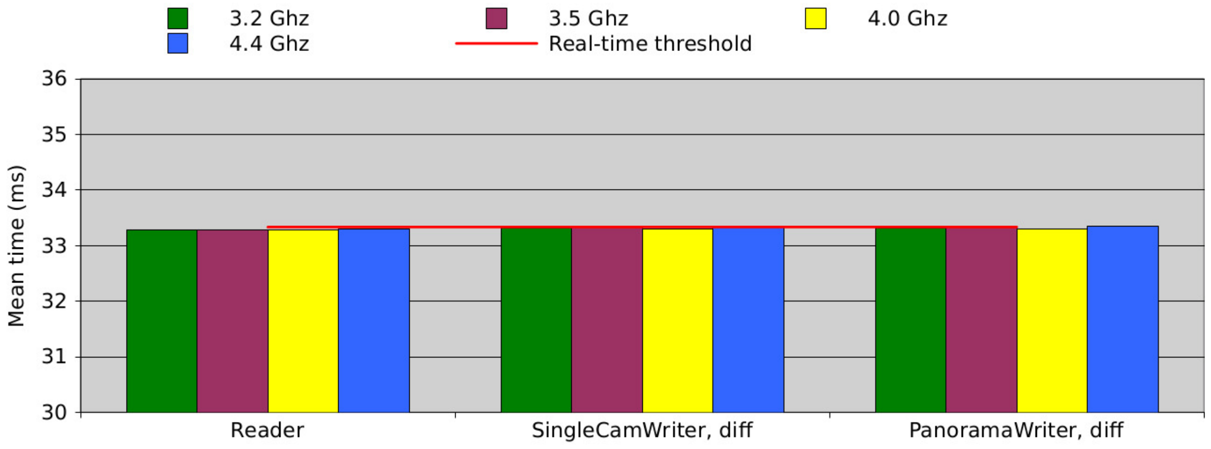


Figure 3.30: CPU core speed, write diffs

3.8.8 Core Count Scalability

To see how the pipeline would perform with different amount of cores, we have done several tests limiting the amount of cores using the *taskset* [56] command. Using TM4, we where able to see how the system would scale from 4 to 16 physical cores. The core count scalability can be seen in figure 3.31. Since the our panorama pipeline is using a high number of threads (currently 86), we should see differences in the timings when scaling down to the number of cores. As the diagram 3.31 shows, the higher the amount of cores, the better the results.

When using only four cores, with the current clock rate of TM4, we see that the Uploader as well as the SingleCamWriter and PanoramaWriter, takes up more time than our threshold. As far as scaling goes, with the TM4 hardware setup, the differences between 12 cores and 16 cores are minimal, meaning that 12 cores should be efficient to achieve maximal pipeline performance.

The modules that takes up the most time when using four cores, is also the modules having the best gains when scaling up the number of cores. As mentioned earlier, the pipeline itself sees a performance gain mainly do to it executing a large number of threads. This is also the reason why the Debarreler, Uploader, SingleCamWriter and PanoramaWriter responds best to the increased amount of cores. All these modules use a larger number of threads then the other modules. The Debarreler makes use of one additional thread per camera, while the writers uses several threads for writing and encoding the file to H.264. The Uploader uses double buffering and asynchronous transfers, making it dependent on hardware threading capabilities. Do to core frequency and the fact that it runs on one thread, the Controller is almost twice as slow on TM4 (2.0 GHz), compared to TM1 (4.4GHz), as it cannot make use of the parallelizing capabilities TM4 gives.

It is also worth looking into the number of frame drops. If modules use more time then the threshold, the will increase the overall performance of the pipeline, thus leading to frame drops. Table 3.5 shows the frame drops when scaling the system. As we can see, when the module times is above our threshold, the number of dropped frames is much higher compared to when the frames gets processed within its time limit. Both the frame drops from the cameras, as well as the pipeline are low and stabilizes as long as the frame processing is running smoothly.

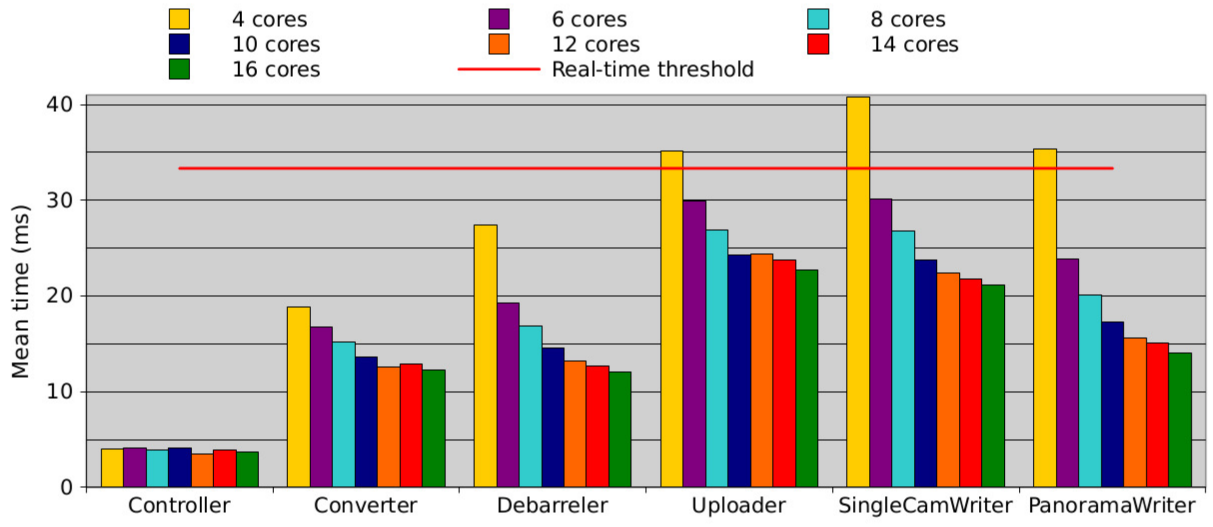


Figure 3.31: Core count scalability, pipeline execution results.

Frame Drop type	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Camera drops	75	26	7	9	6	8	8
Pipeline drops	729	327	67	0	6	3	3

Table 3.5: Core count scalability, dropped frames when running the pipeline (per 1000 frames processed).

Write Differences

Although the diagram in figure 3.31 indicates that using 6 cores is sufficient for real-time execution of the pipeline, figure 3.32 tells us something different. Here we can see that 10 cores are required in order to have a real-time pipeline delivering frames every 33 ms.

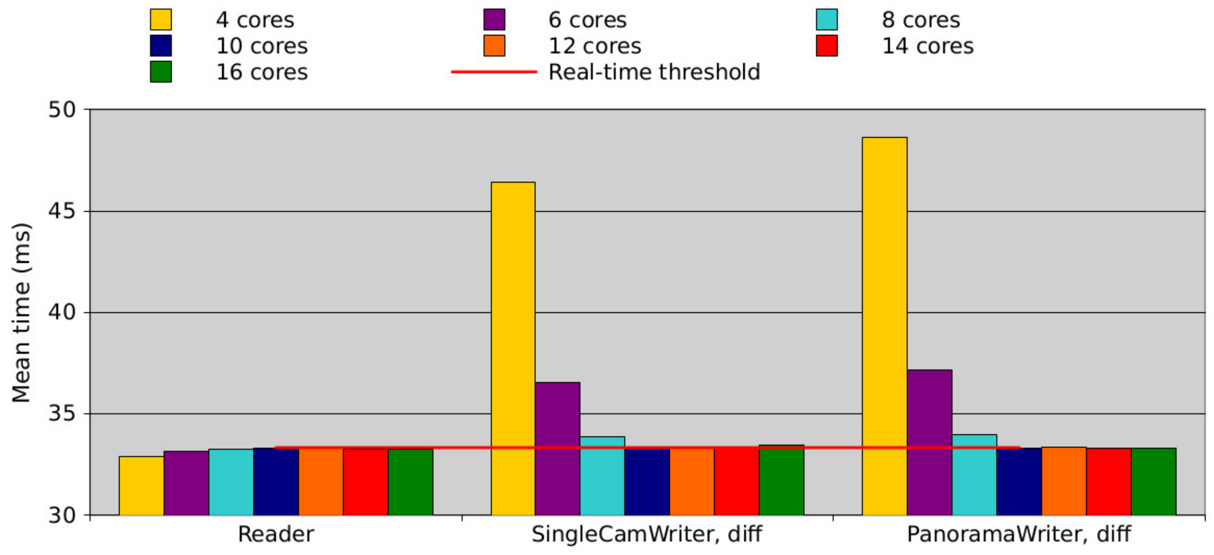


Figure 3.32: Core count scalability, read/write differences.

3.8.9 Hyper-Threading

Hyper-threading (HT) [57] is a Intel implementation used to improve parallelizing abilities in programs using PC microprocessors. A computer has a set of physical processors, and for each one of these, with the help of HT, the OS can address two virtual/logical cores. The goal is to share workload between the two logical cores which in turn will decrease the number of dependent instructions. Since it appears to the OS that it has twice as many cores as there are physical ones, it can schedule twice as many processes. Another feature is that two or more processes can use the same resources. It is known that HT will improve performance within some environments, but may impact performance in a negative manner in others.

To test the effect of hyper-threading on the pipeline, we have executed the program with and without HT on a different amount of cores. With a low amount of cores, using hyper-threading has a positive impact in orders of magnitude.

As the histogram in figure 3.33 shows, the modules that are most effected when limiting the amount of cores is the writer modules. When using only 4 cores, we can see that HT makes a big difference, but it is still not enough to keep the system within its real-time requirement. The differences also visible when using 8 cores, but with the use of HT, we can keep the pipeline just within the threshold. Note that on the GPU modules, the HT actually had a negative effect on the performance, but since modules already are fast it does not effect the overall pipeline performance.Being that the GPU modules executes most or all of its code on the GPU, it cannot make use of the HT capabilities the CPU can offer.

When we use 16 cores the HT has no positive effect on the pipeline. The performance is just below the performance results when running without HT. The reason for this is that the when using 16 cores the effects of HT is not needed for the program, since the number of cores already covers the pipeline need for CPU parallelization. The extra time it takes for working on 32 visual cores makes the timings slightly higher than when having HT disabled.

To see how the HT affects the visual result of the pipeline, we have added a table that shows the frame drops when the pipeline have processed 1000 frames. As seen in table 3.6 and in 3.5, when the modules do not manage to finish its workload within the pipeline, the result is a large amount of frame dropped effecting the visual output of the system. As explained in section 3.8.2, the frame drops processed is how many frames that got written to file. The total number of frames that would have been processed if there where no frame drops would have been the 1203 dropped frames in addition to the 1000 processed frames written to file, i.e., $1203 + 1000 = 2203$ frames.

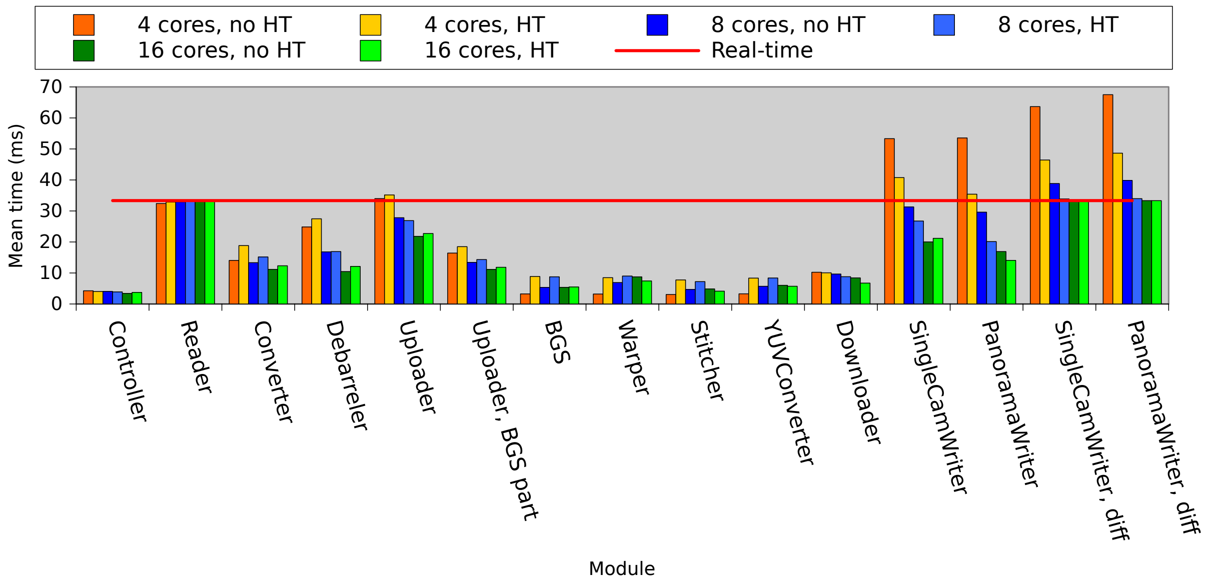


Figure 3.33: Hyper threading, pipeline execution results.

Frame Drop type	4 cores	4 cores, HT	8 cores	8 cores, HT	16 cores	16 cores, HT
Camera drops	223	75	54	7	5	8
Pipeline drops	1203	729	477	67	3	3

Table 3.6: Hyper Threading, dropped frames when running the pipeline (per 1000 frames processed).

3.8.10 Frame Drop Handling

The other test results described in this section does not involve frame drop handling as described in section 3.5.2. By enabling the frame drop handling in the pipeline, the results will not show reasonable numbers with respect to performance timings. This is do to the fact at if a module uses more time than our threshold, the processed frame will be dropped. The result is that almost none of the frames will be processed, giving the impression that a result that would be bad, now looks good. Pipeline results with frame drop handling is shown in figure 3.34.

When the frame drop handling is enabled, to see how this effects the end result refer to table 3.7. Here, we can see that the amount of dropped frames using several of the GPU's is far to high for our program.

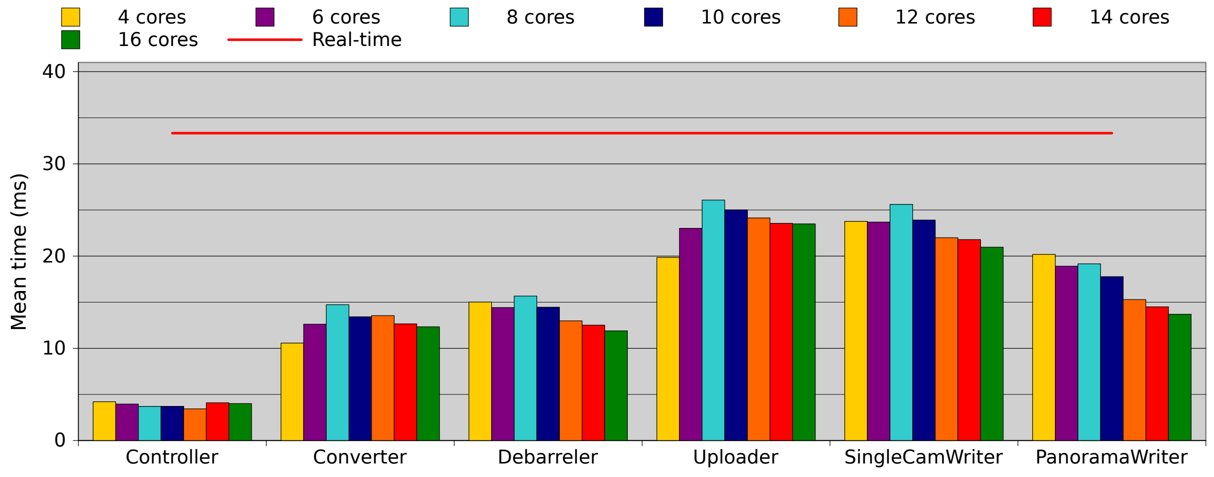


Figure 3.34: Timing results with frame drop handling

Frame Drop type	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Camera drops	41	33	7	4	3	4	4
Pipeline drops	343	177	37	6	2	7	3

Table 3.7: Core count scalability with frame drop handling enabled, dropped frames when running the pipeline (per 1000 frames processed).

Write Differences

Figure 3.35 showd the time differences when reading from the cameras and writing to disk. When running the pipeline with 8 cores or less, the results are similar to the ones presented when we tested the pipeline without frame drop handling in section 3.8.8.

Unlike the modules, the difference times is not positively effected by a low overall mean time in the pipeline. This is do to the design of the pipeline. The writers will never have a mean below our real-time threshold, but slow iterations and poor hardware will cause the mean value to increase. Comparing the times in figure 3.35 with figure 3.32 shows that the timings when enabling frame drops are better. This shows that the frame drops minimizes the overall workload, but the writers are still to slow do to pipeline hardware demands.

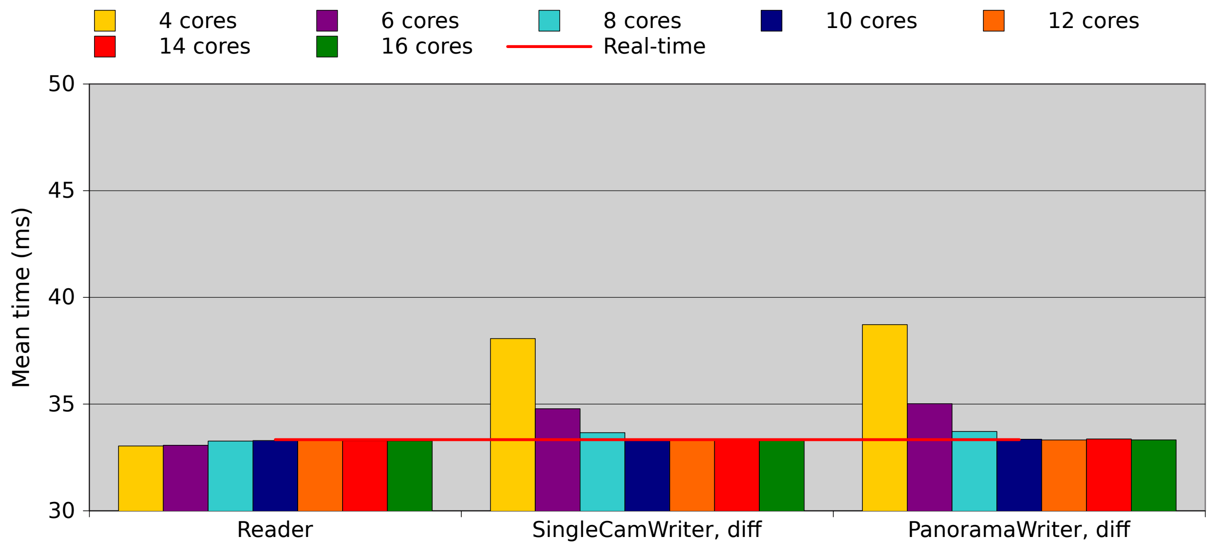


Figure 3.35: Core count scalability with frame drop handling enabled, writer and reader differences (per 1000 frames processed).

3.9 Future Improvements

We have in the previous sections described the pipeline as a whole and shown that it can create a panorama video in real-time. Although the pipeline fulfills its real-time goals, there is still room for improvement with regard to visual output and performance optimizations. Currently, all the modules is running within our real-time requirement, but they can and should be further optimized as a scaling of the system will give the modules more data to handle. Adding additional modules to pipeline will also increase the overall workload on the GPU and CPU, thus also effect the timings of the modules already implemented.

As stated in the previous sections, some of the modules can be further divided into smaller program parts to improve the overall pipeline design and divide the workload over a bigger number of subcomponents. Encoding as it is implemented in the newest version of the pipeline are located in the writer modules, SingleCamWriter and PanoramaWriter. For further system scaling, the encoding will need to be an independent module as it takes up most of the resources in the module they are currently located in. The creation of metadata for the BGS should also be moved from the Uploader and into its own module. Other modules implemented on the CPU could be moved to the GPU to lighten the host load and free more resources for addition modules or other program tasks.

Removing dependencies on open source library is also something that can be considered to improve the overall program performance. By implementing a component for resolving barrel distortion, we can remove most of the OpenCV dependencies and execute more code on the device. As the debarreling module is one of the most time consuming processes in the pipeline, this should be high on the list for future improvements.

Just as the prototype, some of the values used for camera calibration and warping is hard coded and therefore limits the scalability of the system. To solve this, a program should be made to calculate these values and then write them into a XML file that can be parsed at panorama pipeline startup.

It should be considered to scale up the system by using better cameras than can give a more visually pleasing result. If this where to be done, one should also discuss and evaluate the possibilities of dividing the pipeline workload on two or more GPU's. By cutting the frames horizontally, one should be able to let each part execute on a separate GPU, thus be able to process the data in real-time. The only modules that cannot be directly split between the GPUs would be the PanoramaStitcher and the ColorCorrection.

Another interesting change to the pipeline would be to use RGB instead of RGBA. As explained earlier in the thesis, we use RGBA to improve the memory transfer operations between the CUDA device and the CPU. Using RGBA however has an additional cost in memory as each pixel now take up four bytes instead of three. If we can cut the load by of data transferred, we could potentially see a good performance gain in the overall pipeline execution. If we where to look at minimizing the data load so the pipeline can handle higher resolution images it would also be interesting to see if we can use YUV420p throughout the pipeline.

Final improvements to the pipeline involves the camera setup as it is not optimal and currently effects the visual result regarding both the colors and the overall image quality.

3.10 Summary

We have in this chapter described our implementation of a real-time panorama video creation. By dividing the program workload into subcomponents, referred to as modules, we have been able to parallelize the overall workload of the pipeline. Each of these modules is responsible for its own set of image operations and to make sure that they finish within the Bagadus real-time requirement. We have discussed the overall design of the pipeline as well as gone into detail when describing the modules and how they work. We have also looked into the hardware setup and described solutions regarding encoding and storing.

By executing the pipeline with different hardware specifications, we have tested the different aspects of the pipeline and how they work under various conditions. The tests gives indications to where the systems bottlenecks lies and what needs to be done for further scaling and visual improvement. Lastly, we have shortly described shortcomings of the system with regard to camera setup and settings, as well as suggested possible solutions to current pipeline problems.

In the next chapter we will look further into the aspect of improving the visual quality of the pipeline output. By investigating methods for color correction in the program, we will present the algorithm implemented and how it performs in different test environments.

Chapter 4

Color Correction

4.1 Motivation

As we have seen in chapter 2, there is obvious light and color mismatching between the images in the final panorama. To make these differences as small as possible, we have implemented an algorithm for color correction. The method works in linear RGB color space to find the differences between adjacent images in the sequence and uses this information to correct the final panorama.

In this chapter, we present the color correction method applied to the panorama pipeline. We describe the algorithm in its original form and present the modifications we have done to make it applicable to our scenario. By giving visual and computational results of the algorithm, we explain the trade offs we have made to implement a simple color correction method that runs in real-time.

4.2 Related work

Color correcting a panorama is a method often used to resolve or minimize color and luminance differences between the frames that makes up the final stitch output. [22].

Today, there exists several systems and algorithms that offer different solutions to the problem (e.g., [19, 20]). One of these methods was developed by Au and Liang in 2012 [21]. Their algorithm is able to improve the visual result of a panorama by applying color correction. Using mobile phone hardware, they can correct for color differences in a six-image sequence in 5.5 seconds. Although these results are from a cell phone and it should be possible to improve the performance of the algorithm, the algorithm repeats several of its steps, making it not well suited for a device implementation.

Another interesting color correction scheme is presented in [58]. They propose a method for automatically selecting the reference image, thus making the whole color correction algorithm automatic. The method is also computationally fast, being able to do most of the color correction within 35.5 ms. The paper in [59] also describes an algorithm for efficient correcting of image brightness and saturation. With the help of blending algorithms in removes the color differences in the adjacent images.

A color histogram based color correction method is proposed in [60]. Although the method gives good results, the paper does not describe the computational performance of the algorithm and therefore, we do not look further into their work.

To summarize, the color correction methods described in the above studies [21,58–60] have interesting features, but most of the algorithms do not fulfill our demands with regards to computational speed and ability to run on a GPU. Instead, the algorithm by Xiong and Pulli was implemented [22], as it is fast and fits well onto a GPU architecture.

4.3 Gamma Correction

A color space is said to be linear if doubling a value in the space gives a color that is twice as bright [61]. Almost all devices, including CRT displays, have a signal-to-light intensity or intensity-to-signal characteristic that is not linear [62]. To display colors in a linear space, one needs to modify the incoming image or signal in a way that removes the non-linear property from the display device. This modification of the incoming image is known as gamma correction and is done in the hardware that capture or creates the image. In our case the cameras does the gamma correction.

All CRT and other image devices have a transfer function that is approximated fairly well by a power function. This power function makes use of gamma as an exponent, and is as follows:

$$LinearRGB = Voltage^{\gamma}$$

This function, called a Display gamma function is used to map linear RGB to a non-linear RGB space, used by CRT and other media devices like LCD monitors etc. to display the image. The most common used gamma value for such devices are 2.2 and is used throughout this thesis. To remove the changes the display function would due to the image values recorded by our cameras, the cameras need to apply gamma correction. This correction is done with the following function:

$$GammaRGB = LinearRGB^{\frac{1}{\gamma}}$$

By applying this function to the data in the device, the resulting output from the screen will result in our original linear values read by the camera.

The gamma function graph 4.1 shows what would happen to the values by not performing gamma correction. The blue line indicating the linear RGB intensity would result to the bottom curve if the cameras or do not apply the gamma correction to the linear RGB colors. The linear RBG intensity (ranging from 0-1) would at the most decrease with over half the value resulting in a image that is darker than it should, having only half the brightness.

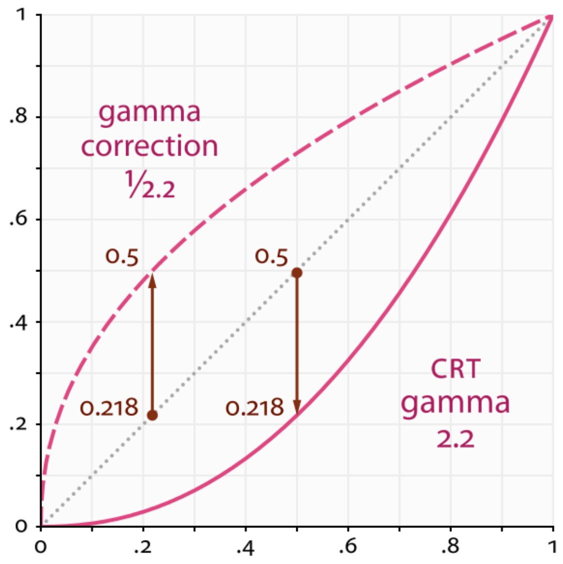


Figure 4.1: Gamma function graph.

4.4 Real-time Color correction

For efficient color correction in our panorama pipeline, we have implemented the algorithm by Xiong and Pulli [22]. When presented in the paper, the algorithm describes a scenario using one camera and rotating it to create a sequence of images. By comparing the overlapping regions of the images, they are able to correct for color differences in a panorama. The color correction method starts by finding a set of local color coefficients that corresponds to the individual changes for each image. This is done for each image channel, thus getting optimal results. Then in combination with global coefficients, they apply the correction to each image.

Unlike most other color correction methods, the algorithm works in linearized RGB by removing gamma correction from the pixel values before finding the correct coefficients. To further smoothen the colors in the output, they apply global coefficients for each channel to reduce the individual changes in each image and reduces the possibility of saturating the panorama. The steps mentioned above are done before the algorithm is applied to a panorama stitching program that further optimizes the result with the help of blending.

The scenario presented in the paper by Xiong et al. is different then our current setup, but should be highly applicable to our camera setup as well. As we do not want blending in our pipeline duo to ghosting artifacts created by moving objects, this part of the algorithm was not explored or researched in this thesis, and therefore the test results will still to some extend show visible image seams.

4.4.1 Color Correction algorithm

Here, follows a more detailed explanation of the algorithm steps performed:

1. The first step of the algorithm is to locate overlapping regions between images. These is currently done manually at the setup phase of the program.

2. Each image has a set of color correction coefficients. The set has one coefficient per channel, (either r,g,b = 3, or r,g,b,a = 4). These are found by first selecting a head/base image. In our implementation we use the leftmost image as head, and set its color coefficients to 1. Xiong et al. [22] select the head image by finding the image in the sequence with the best color difference. Due to the limited amount of images in our setup, we do not apply this to our color correction module. Next, we calculate coefficients for the remaining image sequence by doing the following:

   (a) compare the overlapping regions for image[i] and image[i-1].

   (b) calculate the difference for each channel c, to find the individual color correction coefficient for image[i].

   The formula for the calculation is shown below

$$\alpha_{c,i} = \frac{\sum_p (C_{c,i-1}(p))^\gamma}{\sum_p (C_{c,i}(p))^\gamma} \qquad c \in R, G, B \qquad (i = 2, 3, .., n)$$

   where p represents the pixel, while i is the i-th image in the sequence. $C_{c,i}$ is the color value at pixel p.

3. To smoothen the overall color of the panorama and minimize the chance of image saturation caused by the individual color coefficients, we find global coefficients for each color channel. These changes are later applied to each image.

$$g_c = \frac{\sum_{i=1}^{n} a_{c,i}}{\sum_{k=1}^{n} a_{c,i}^2} \qquad c \in R, G, B \qquad (i = 2, 3, .., n)$$

   Here, g_c is the global adjustment coefficient for channel c. The formula is found by solving an objective function, i.e a function that finds an optimal factor that for the color correction aims to minimize the changes in each image.

4. As a final step before doing the actual color correction, the algorithm finds the final correction coefficients for each image by multiplying the individual correction coefficients with the global coefficients for each channel.

$$C_{c,i}(p) \leftarrow (g_c \alpha_{c,i})^{\frac{1}{\gamma}} C_{c,i}(p) \qquad c \in R, G, B \qquad (i = 1, 2, .., n)$$

   Here $C_{c,i}(p)$ represents the final correction coefficient for channel c in image[i].

5. After the algorithm have computed its final coefficients, the images can be corrected.

Figure 4.2 shows the work flow of the algorithm.

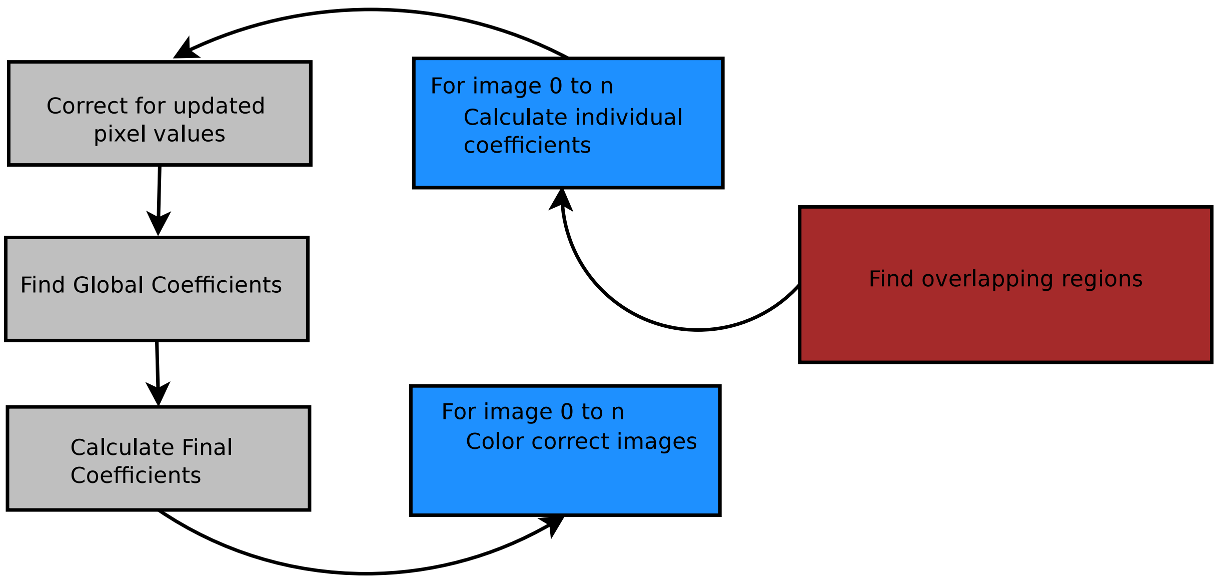

Correct for updated pixel values
For image 0 to n Calculate individual coefficients
Find overlapping regions
Find Global Coefficients
Calculate Final Coefficients
For image 0 to n Color correct images

Figure 4.2: Color Correction work flow.

4.4.2 Algorithm limitations

As described in the original work [22], there are limitations to the algorithm. The color correction is based on the assumption that the panorama is created by rotating a single camera while capturing images from different angles to get a large field of view. As mentioned before, our scenario is somewhat different.

- By limiting the camera position, you also minimize the chance of having problems caused by lighting conditions and different shading for the individual images. In our current setup, as explained in chapter 2, we have two and two cameras placed close together, but the middle ones are separated by approximately 5 meters. As we will see later in this chapter, this can have an impact on the performance of the algorithm.

- The described method is also sensitive to outliers in the overlapping regions. Although the algorithm itself does not depend on exact pixel to pixel mapping, with a limited amount of overlap, pixel outliers may have an impact on the end result.

- Light conditions may also disrupt the algorithm. Given bright light, often combined with snow or other reflecting surfaces in large regions of the frame, this will negatively effect the end result. Given bright colors close to the rgb maximal range of 255 in several images and color coefficients that indicate that one of the frames should be darker, the algorithm will darken areas in the panorama that should be kept light.

- As with other panorama color correction algorithms, to remove the seams in the final image, it also dependents on good panorama stitching and image alignment that does not create visible artifacts in the panorama.

- Given bad input images caused by different camera exposure time etc., the differences between the images will disrupt the colors in the final panorama. With overall too bright colors from the source images, the image will be over-exposed and in some cases saturated in such a degree that it cannot be solved by the algorithms global coefficients.

4.5 Color Correction - Vanilla version

The first version of the color correction involves no modifications to the original algorithm [22]. The implementation is done completely on the CPU. To test the algorithm and how well is suits our scenario, we provided a high precision implementation entitled the Vanilla version. The specific version was built according to the steps described in section 4.4.1. As performance speed was not an issue in the first implementation, there are no improvement steps for faster execution in the Vanilla version of the color correcter. To get better overlapping regions, and make it easier to find them, the color correction is applied after the images are given the same angle of view (warped).

Figure 4.3 shows four images taken from our test case in Tromsø. As can be seen by the image, a combination of moderate hardware, poor camera settings and special light conditions makes the camera output hard to work with and not visually pleasing.

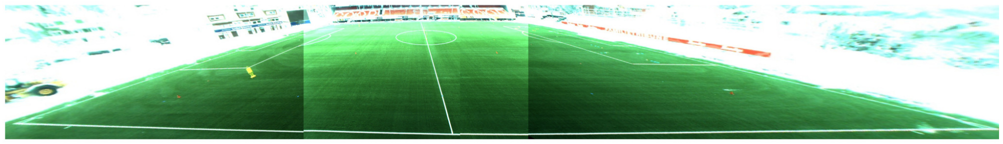


Figure 4.3: Panorama without color correction and dynamic stitching.s

4.5.1 Implementation details

Here we will give a detailed description of the implementation. In the following explanation, we assume that the overlapping regions have been found and that the images are in a sequence where the first image is the leftmost and the last image are from the rightmost camera.

Below follows a list of important points of information regarding the algorithm:

- As in the original definition of the algorithm, we use the first image in the sequence as head. The heads correction coefficients is set as 1, meaning that the other images in the sequence will be corrected according to the illumination levels and color values in this image.
- The overlapping regions is one single region of the same size in each frame.
- Gamma coefficient is set at 2.2.
- Since we are working on RGB(RGBA, but A is empty) values in the pipeline, the image is of size height$\times$width$\times$(number of color channels), meaning that when accessing memory space image[i][pixel], pixel = pixel$\times$channels in the code. This means that pixel is the position of R, pixel+1 is G and pixel+2 is B.

Finding individual color coefficients

When we have initialized and allocated the necessary memory, we start by finding the color coefficients for each image. First we set the first image (head) color coefficients to 1, then we start by finding the color coefficients of the remaining images with the help of the formula in step 2, section 4.4.1.

By removing the gamma correction from the image data, we are able to work in linear RBG space. Since the cameras perform gamma correction by using a transfer function with $\frac{1}{2.2}$ as a gamma coefficient, we remove the gamma correction by raising the pixel value in $\gamma$to work on a image space that have a linear correspondence to actual color values [22].

Algorithm 3 shows the psuedo code used for finding the individual image color coefficients, the first step in the color correction algorithm.

Algorithm 3 Pseudo code for calculating channel coefficients, a triplet for each image

```
double r1, g1, b1;
double r2, g2, g3;
for All images i in the sequence do
  if image[i] == head then
    Set all channel coefficients in the image to 1, i.e. $I_c = 1$;
    $ColorCoefficients[i][c] = 1$;
  else
    Find overlap pixel (A pixel has one byte per color channel)
    for Overlapping region in image[i-1] do
      $r1+ = image[i-1][pixel]^{2.2}$;
      $g1+ = image[i-1][pixel+1]^{2.2}$;
      $b1+ = image[i-1][pixel+2]^{2.2}$;
    end for
    Find overlap pixel (A pixel has one byte per color channel)
    for Overlapping region in image[i] do
      $r2+ = image[i][pixel]^{2.2}$;
      $g2+ = image[i][pixel+1]^{2.2}$;
      $b2+ = image[i][pixel+2]^{2.2}$;
    end for
    $ColorCoefficients[i][r] = r1/r2$;
    $ColorCoefficients[i][g] = g1/g2$;
    $ColorCoefficients[i][b] = b1/b2$;
  end if
end for
```

Since we find all the color coefficients for each image before we correct the differences in the frames, there will be inconsistencies in the end panorama. This is due to the fact that we find the correction coefficients based on old pixel values. This means that each frame to the right gets corrected based on the old values from the frame to the left. To solve this problem, we update the values by multiplying them with the coefficients from the image to its left as seen in algorithm 4

Algorithm 4 Pseudo code for updating the individual coefficients

```
for each image in the sequence, except head do
  $ColorCoefficients[i][r] = ColorCoefficients[i][r] \times ColorCoefficicents[i-1][r]$
  $ColorCoefficients[i][g] = ColorCoefficients[i][g] \times ColorCoefficicents[i-1][g]$
  $ColorCoefficients[i][b] = ColorCoefficients[i][b] \times ColorCoefficicents[i-1][b]$
end for
```

Finding the global color coefficients

To smoothen out the color differences in the final panorama and make the local changes as small as possible, we need to multiply a global coefficient with the individual color coefficients. To find these global coefficients, we apply algorithm 5. As described in step 3 from section 4.4.1, we find a factor that for the color correction minimizes the changes in each image from the sequence.

Algorithm 5 Pseudo code for finding the global coefficients

```
double divident;
double divisor;
for all channels j (r,g,b) do
    $dividend = 0.0, divisor = 0.0$
    for all pictures i in the sequence do
        $dividend+ = ColorCoefficients[i][j];$
        $divisor+ = ColorCoefficient^2;$
    end for
    $globalAdjustementCoefficients[j] = \frac{divident}{divisor};$
end for
```

Calculating the final color coefficients

To find the final coefficients used for the color correction, we multiply the global coefficients with the local coefficients. This divides the differences among all the images in the sequence and minimizes the individual changes. Since we removed the γ coefficient when calculating the individual and global coefficients, it is important that it is applied to the image again. By applying a gamma of $\frac{1}{2.2}$ to the correction values, we add the gamma correction we previously removed when finding the individual color coefficients. Being that the devices displaying the image has a gamma coefficient of 2.2, it is important that we reapply the gamma correction to compensate for the non-linear property of the displays. The algorithm in 6 shows what needs to be done to find the final coefficients.

Algorithm 6 Pseudo code for finding the final coefficients used when correcting the image

```
for each images i in the sequence do
    for each channels j in the color space do
        $finalColorCoefficients[i][j] =$
        $(globalAdjustmentCoefficient[j] \times ColorCoefficients[i][j])^{\frac{1.0}{2.2}};$
    end for
end for
```

Color correcting the image sequence

When doing the actual color correction, the algorithm presented in algorithm 7 is used for the Vanilla implementation. The algorithm is executed once for each image in the sequence. As we can see, the code is simple and easy to parallelize making it well suited for the GPU and our panorama pipeline design. As mentioned earlier, RGB ranges from 0-255 in value, where high values give brighter colors and values close to 0 gives dark colors. To handle the cases where there are bright colors and we correct the image with a value higher than 1.0, an example being that the entire sky is white, we make sure that we do not go above the value 255. Since we use unsigned char, a bigger value will wrap around, causing the light color to become dark. Therefore, we have the extra test in the code that keeps the color values within the RGB range.

The pseudo code in 7 finalizes the color correction and given images without clear saturation, should remove color differences in the images and create a visually pleasing panorama.

Algorithm 7 The pseudo code shows the actual image color correction (i.e. changing the picture values)

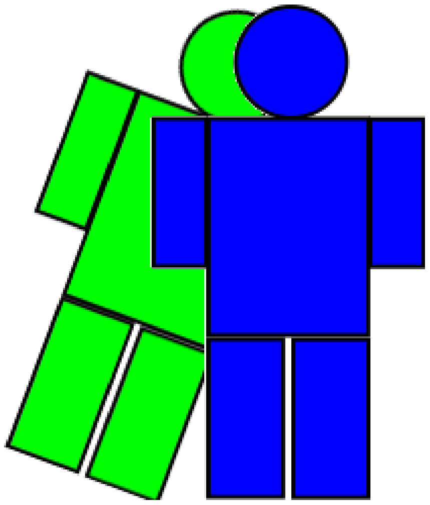
```
for all pixels i in image do
    if image[i × channels] × finalCoefficients[0] > 255 then
        $image[i \times channels] = 255$
    else
        $image[i \times channels] = image[i \times channels] \times finalCoefficients[0]$
    end if
    if image[i × channels+1] × finalCoefficients[1] > 255 then
        $image[i \times channels + 1] = 255$
    else
        $image[i \times channels + 1] = image[i \times channels] \times finalCoefficients[1]$
    end if
    if image[i × channels+2] × finalCoefficients[2] > 255 then
        $image[i \times channels + 2] = 255$
    else
        $image[i \times channels + 2] = image[i \times channels] \times finalCoefficients[2]$
    end if
end for
```

4.6 Color Correction - Minimizing the overlapping areas

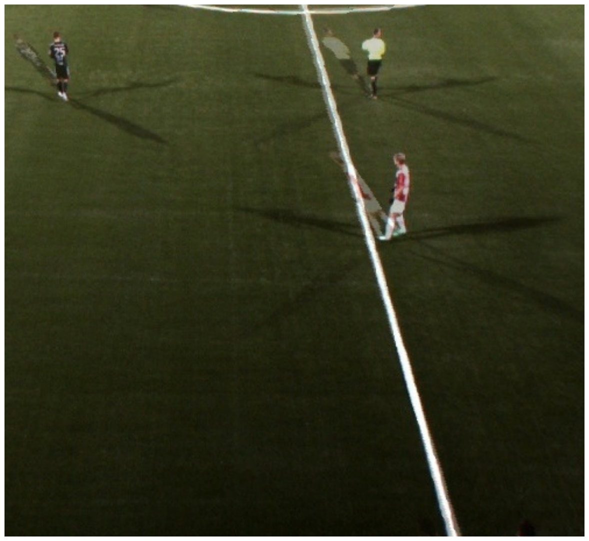
The algorithm bases its coefficients on the overlapping areas of adjacent images and does not acquire accurate pixel to pixel mapping. To further minimize the computation time, we have limited the number of pixels being used for calculation, thus minimizing the algorithm overall time. As long as there are no outliers in the overlapping regions, using smaller overlapping regions should not affect the end result as the overall color and luminance differences in the frames should be the mostly the same throughout the image.

Minimizing the overlapping regions however, requires that the image processing steps performed before the color correction, manages to create a good overlapping region where the pixels are well aligned. The overlapping regions should also be as close to the stitch as possible, since finding the color coefficients near the seam makes it more likely that we can still remove the seam even if the image does not have the same overall color and luminance differences.

There is also issues duo to the fact that we use warping when stitching, this results in object misalignment as seen in figure 4.4 and increases the possibility for outliers in the overlapping regions. This will effect the correction algorithm independently of the size of the region, but the bigger the overlapping area, the smaller the impact of the outliers.



(a) Player misalignment.



(b) Ghosting effect example.

Figure 4.4: Player misalignment and ghosting caused by warping the frames.

4.6.1 Performance

Figure 4.5 shows the panorama using different sized overlapping regions. The method has been applied to a test panorama using the old setup due to the good visual output compared to footage taken with the new setup. Image in figure 4.5(a) shows the result when using a 10×10 overlapping area. The only seam still clearly visible is the one between camera two and three and this is most likely duo to the distance between the cameras as they are placed with several meters apart. Comparing the image using the 10×10 overlapping area with the other ones,figure 4.5(b) and figure 4.5(c) shows that there is little difference between using a small overlapping area, half of the overlapping region and using the entire region. In all images the only clearly visible seam is the one between frame two and three. This shows that if we can process more frames and data in a short amount of time, using a small overlapping region might be worth the computational gain.

Due to the operations done to calculate the coefficients there is also a limited amount of ways we can optimize the code. Table 4.1 shows the computational time required when using different sized overlapping regions. Note that the code calculating the color coefficients is not further optimized than in the Vanilla version. Due to the visual results from the these test, we see that the visual gain is minimal and therefore we did not spend any more time optimizing a code that runs in real-time and gives similar visual results regardless of the overlapping region being 10×10 or the entire region.

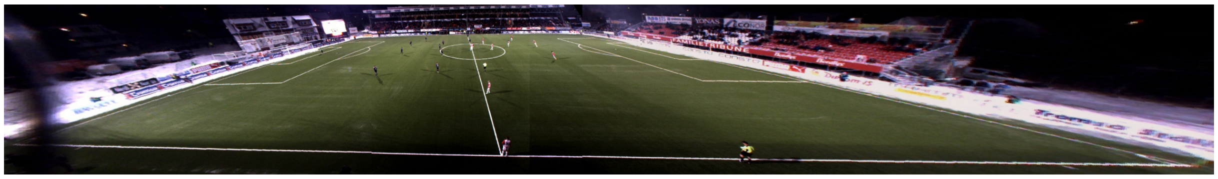
When calculating the color coefficients in a 10×10 region, we use 100 pixels per image. Half of Max is in this case a 182×345 overlapping region and max size gives us a 364×690 sized region. As we can see from the numbers in table 4.1, the increases in time is quite large and the only one that uses a amount of time that leaves a reasonable number of recourses left for the other steps in the algorithm is the 10×10 region. We have also included a 10×100 and a 100×100 sized region to show that the increase in time is pretty equal to the increase in pixels. Using a region ten times as big equals a time consumption of almost ten times. From these tests we can conclude that with the current implementation the optimal overlapping regions will be somewhere between 10×100 and 100×100 pixels.

Given that this is only a small part of the algorithm, we can only use a limited amount of the 33 ms threshold when finding the individual color coefficients. This means that calculating the color coefficients needs to be a trade of between visual result, coefficient computation time, and the computation time for the rest of the algorithm. Using a limit amount of resources also frees GPU capabilites for the rest of the pipeline modules.

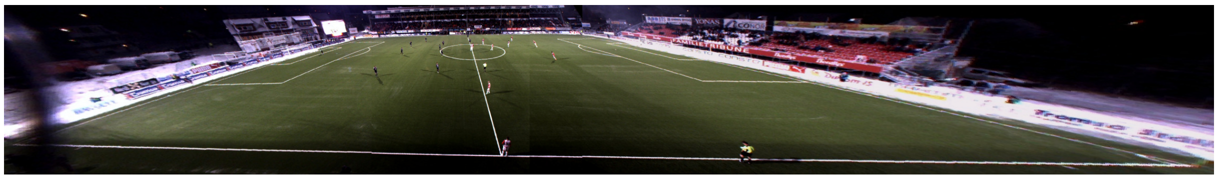
Overlap size	10 × 10	10 × 100	100 × 100	Half of Max	Max size
Mean (ms)	0.142	1.081	10.543	110.668	441.8

Table 4.1: Overlapping regions calculation (in ms), without gamma on TM3.

(a) Color correction using only 100 pixel overlapping region.



(b) Color correction using half the overlapping region.



(c) Color correction using the entire overlapping region.

Figure 4.5: Color correction applied to panorama with different sized overlapping regions.

4.7 Color Correction - Pipeline Version

As the color correction will be used as an important module in the Bagadus panorama pipeline, we evaluated the effect of removing the global adjustment and the gamma step for the correction. Since we have a limited number of cameras (four), we need an algorithm that is computationally efficient while still being able to providing pleasing visual results.

In a long sequence of images, minimizing the individual adjustment and smoothening the color and luminance difference over the entire sequence will give a overall better end result where you minimize the possibility where the images in the sequence will look bright on one side and dark on the other. Using only four cameras removes this issue and we can therefore remove this part of the algorithm as it will not make much effect at the end result.

Removing the global adjustment also removes a remarkable processing part of the algorithm, since the first image in the sequence can be skipped when doing the color correction.

Due to camera calibration steps, the warping being affected by our setup and hard coded variables that is not optimal, we have to limit the overlapping region to squares on the pitch close to the seam where the images have the best match. As a result, outliers as explained in section 4.6 might make small changes to the color coefficients. The overlapping region between camera two and three at our testbed can be seen in figure 4.6.

In the current pipeline version we have also removed the gamma step of the algorithm duo to the problems with the current camera output not creating visually pleasing results, so not working in linear RGB does not make a noteworthy difference.

The pipeline implementation does its current color correction in a sequence:

1. For the entire image sequence, except the first in the sequence.
    (a) First Find the color coefficients for image frame f_i.
    (b) Perform color correction for the image.

The overall algorithm, as some of the other pipeline modules, can still be further optimized, but since it does its processing within the program threshold, we have not done more to optimize its speed performance. When running the implementation outside the pipeline, the module works slightly faster as the overall workload on the GPU is much smaller then when all pipeline modules are executed. The table 4.2 shows the timings of the different algorithm steps used in the pipeline version of the color correction. Some additional operations for moving data between host and device is also in the implementation, but the operations are fast and therefore not represented in the table. When we performed testing on the pipeline, the color correction implementation was sequential in the way that as soon as we found the color coefficients for one image, we corrected that one and then repeated this with the next image in the sequence and so on. The newest implementation ready for use in the pipeline first finds the individual color coefficients, then updates them, then color corrects them. It also does the individual color coefficients calculation slightly more efficiently. Other the that it is similar to the original algorithm except that as we mentioned, the pipeline module work in gamma corrected space and do not make use of global coefficients.

	Finding color coefficients	Updating color coefficients	Color correct image
Mean time(ms)	0.14	0.001	18.312

Table 4.2: Color correction timings (in ms), time used for different algorithm steps on TM3.

To perform the actual color correction, we use the CUDA parallelizing capabilities. By using the device properties, we find the optimal number of pixels to calculate per kernel, as well as the number of blocks and number of threads per block we should use. For speed performance results of the algorithm implemented in the pipeline, please refer to section 3.8.

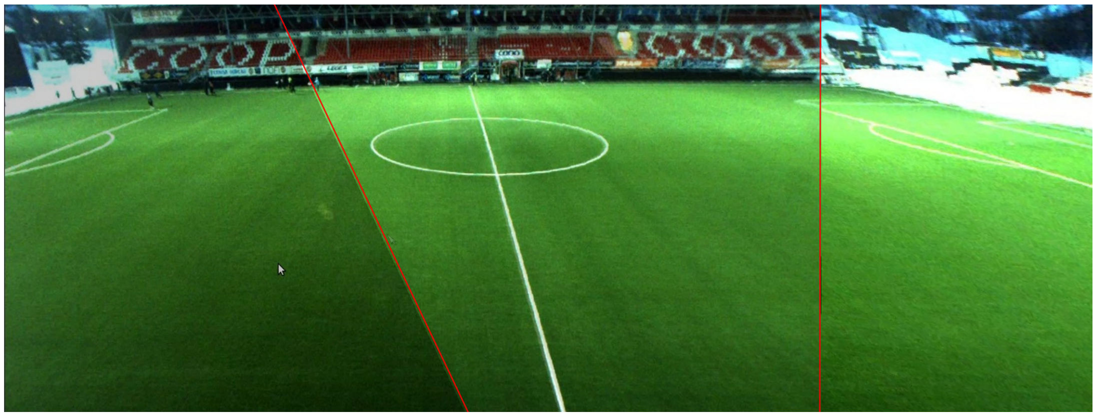


Figure 4.6: Overlapping regions between warped images from camera two and three in the pipeline.

4.8 Color Correction Performance

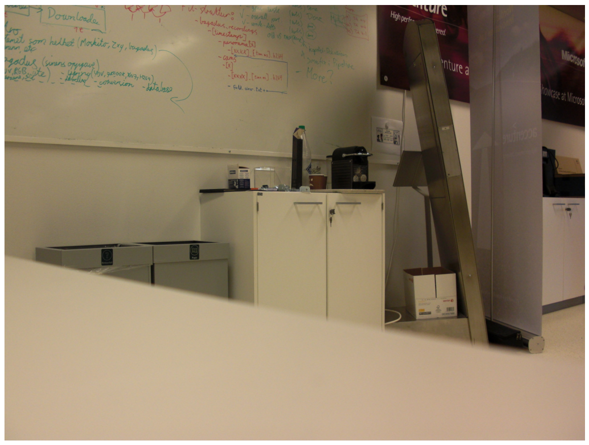
The section presents the visual results achieved by the algorithm in different light conditions.

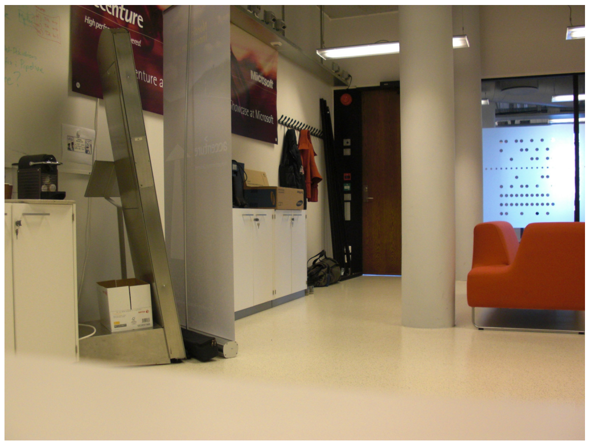
Indoor Scene

To see the general performance of the algorithm and how it handles the indoor scene, we have tested the original algorithm on two different test sets taken indoors. Using a single camera, we have taken four images from different angles indoors. The scenario are in these sets very similar to the one described in the article and it is therefore interesting to see the performance of the method without applying blending. The image sequence is stitched together to create a panorama, but no additional step have been done. This results in misalignment between some of the images and therefore makes the seam visible regardless of color correction method performed.

In the first set, seen in figure 4.7, we have taken a sequence of images from different angles. The set consists of 4 frames and is taken while the camera is standing on a reflecting surface. This causes the clearly visible seam in the bottom of the image and as explained above, the indoor sets are not completely aligned. Figure 4.7 shows the individual frames and the result of the color correction from the original paper applied to the set. As we can see, when comparing the panorama in figure 4.7(f) with the image in figure 4.7(e) the method is able to minimize the color and luminance differences between the individual frames. To completely remove the seams, one would require a better panorama stitch and the use of blending as explained in [22].

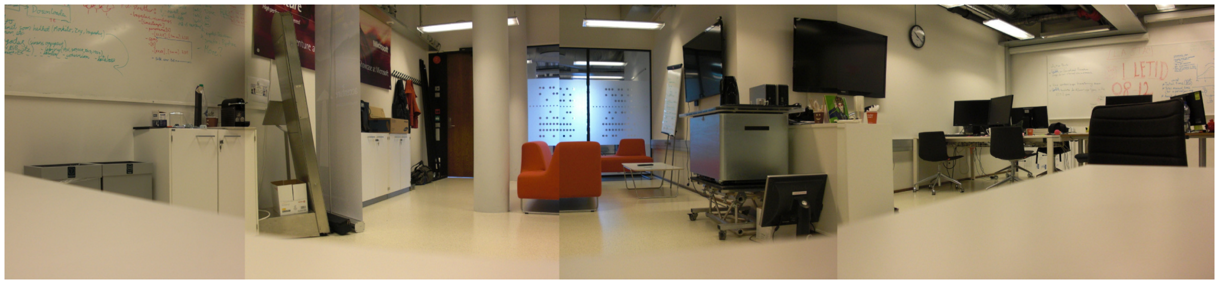
In the second set we tried to minimize the possibility of image misalignment and therefore used a camera to take four individual frames of the same scene where the camera is moved further and further to the left. This means that all the frames are taken from the same angle but approximately 40 cm apart. One result of this is that we have large overlapping regions that can be used to find the color coefficients. Due to the camera being placed on a reflecting white surface, the light is somewhat different on this region of the frames compared to the rest. This results into a stitched image where the seam is visible at these areas in the image. The stitched output can be seen in figure 4.12(e), and the same image without color correction can be seen in 4.11(b).


(a) Leftmost image, indoor scene.


(b) Middle left image, indoor scene.

(c) Middle right image, indoor scene.

(d) Rightmost image, indoor scene.


(e) Stitched panorama of indoor scene, without color correction.

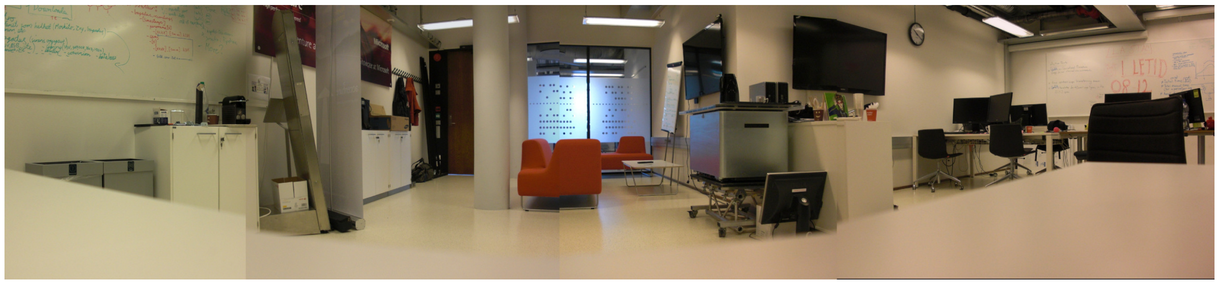
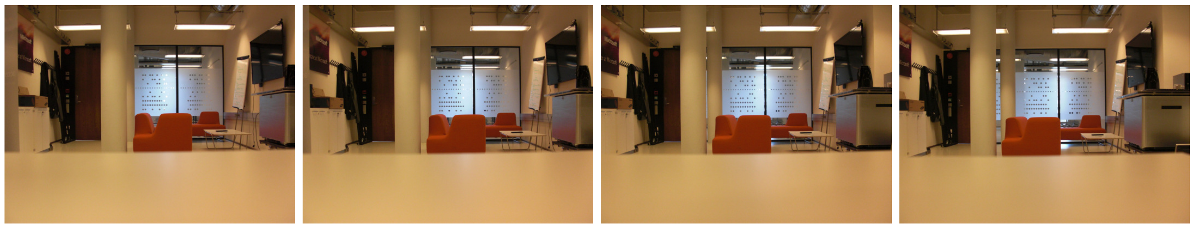
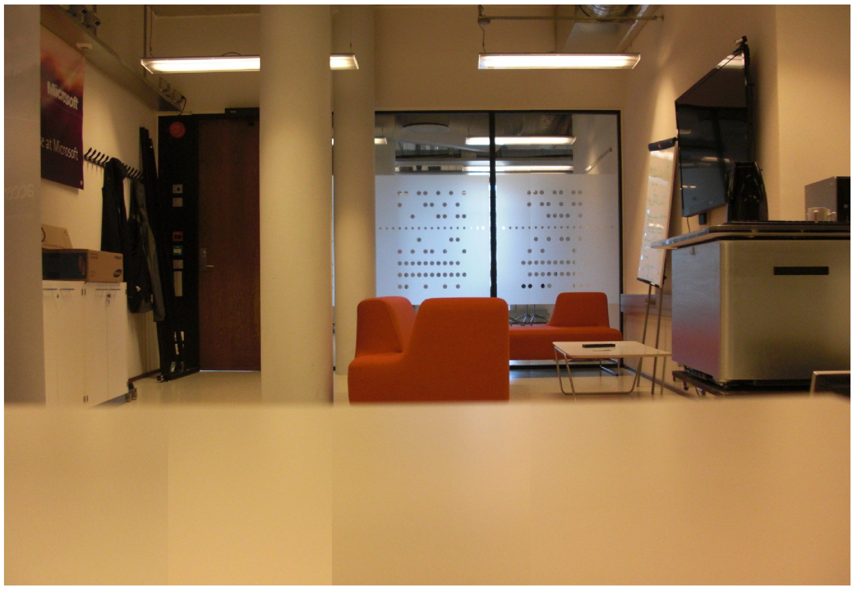

(f) Stitched panorama of indoor scene, with color correction.

Figure 4.7: Color correction, indoor scene, test set one.


(a) Color correction input.


(b) Stitched panorama of indoor scene, without color correction.

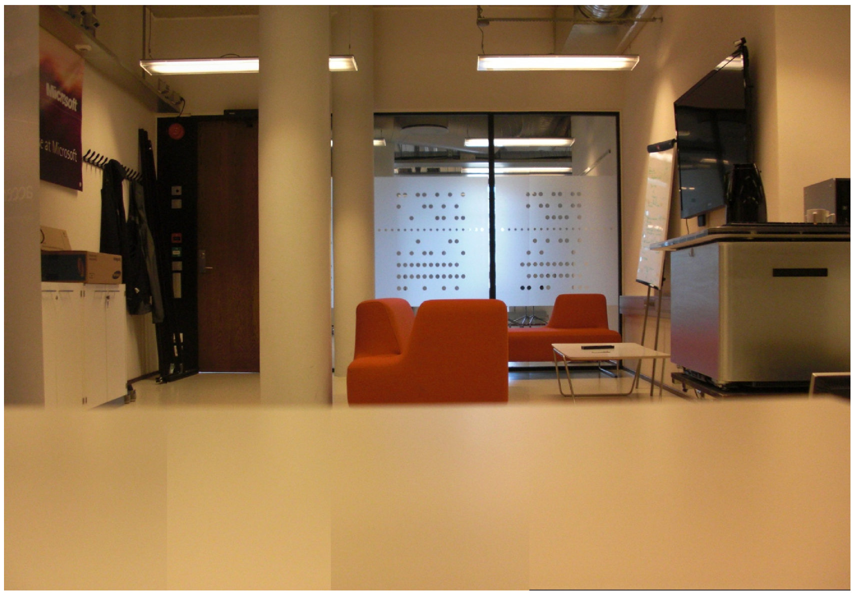

(c) Stitched panorama of indoor scene, with color correction.

Figure 4.8: Color correction, indoor scene, test set two.

4.8.1 Outdoor Scene

To see how the algorithm works outdoor and at our testbed, we have tested the algorithm with and without modifications to see how the different part of the algorithm affects the end result. For testing the different light condition we present four different test sets. In all of them we show the panorama without color correction, applying the original algorithm, finding the coefficients in gamma corrected space, and with the method implemented in the pipeline.

Test Set One

The first set can be seen in figure 4.9. The images are taken late evening April 4th at Alfheim stadium. The top row of the figure shows the frames delivered by the cameras before being warped and color corrected. The colors are very different in the images and except from the rightmost image, the frames are very bright. Due to these obvious color difference the algorithm is not able to correct for the color and luminance differences in the final panorama. Image 4.9(b) shows the resulting panorama without color correction. As can be seen, the individual frames have clear differences. The third image from the bottom 4.9(c), shows the original algorithm applied to the panorama. Here we can see that the differences is minimized but still visible in this test set. Being the case that there are problems with white balancing in the images, the output is not very satisfying and gives the panorama a overall green/yellow look.

The bottom panorama seen in 4.9(e) uses the same algorithm modification as the one implemented in the pipeline. The image is similar to the result using the original implementation and makes it harder to spot the seams. The last panorama presented in the figure shows the resulting image when working in gamma corrected color space. We can see that the differences is much more eminent here than in the other panoramas. Although the seams are more visible in this image, the individual frames display a better range in color and luminance.

Test Set Two

Figure 4.10 shows the frames and the algorithms applied to a frame sequence taken earlier in the evening April 4th at Alfheim. The overall sequence gives brighter images, but the problems between the different frames still exists. When looking at the results from the different color correction algorithms it is obvious that the panoramas are more alike than the ones presented in test set one. Here we can also see that the original algorithm and the one used in the pipeline does the best job when it comes to removing the differences between the individual frames. However, like with test set one, when correcting in gamma corrected space the output have a wider range of color values.

Test Set Three

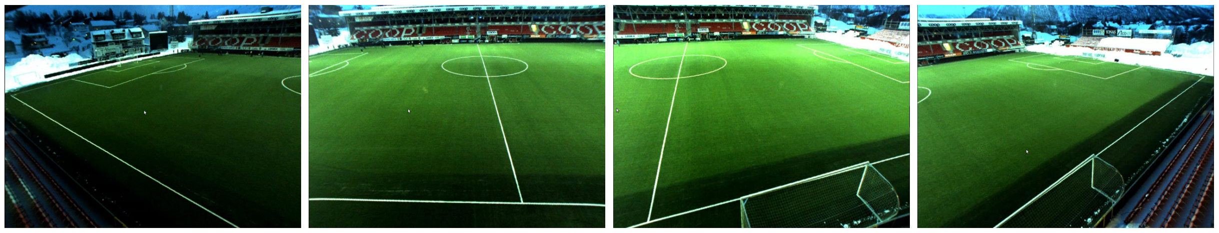
In test set three, the images are taken at daytime April 4th. The resulting panoramas and the individual camera frames can be seen in figure 4.11. The image sequence gives clear indications of the current problems with camera settings and auto exposure. Due to the snow in the leftmost and rightmost frames the images, white balancing causes problems that is currently not handled in the camera setup. The output from the different algorithms presented in figure 4.11 shows that under these light conditions all the algorithms struggle to remove the differences in the image sequence and the result are very much alike. The seam is almost non existent in the bottom of the panoramas, but they are clearly visible at the top of the images indicating that the color and luminance difference in the image is not consistent throughout the frame. This is also a problem with test set one and two.

Test Set Four

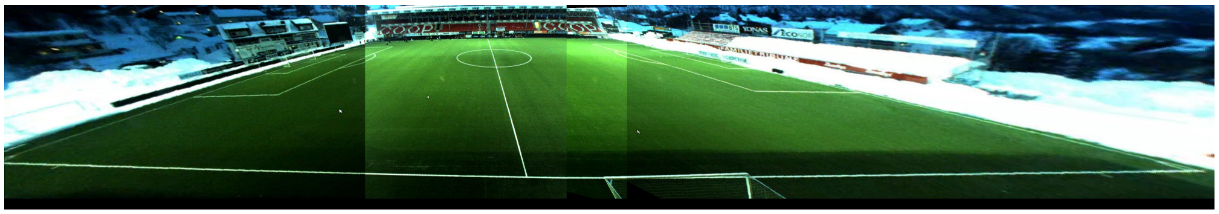
Here the frames are taken where the cameras exposure setting are specifically set for the particular lighting conditions. As with the other sets, the third image from the left in figure 4.12(a) have different luminance levels then the other frames. Overall the frames have better color and luminance values than in the other test sets. The panorama without color correction 4.12(b) shows that the differences between the frames is not as visible as with the light conditions presented earlier in this section. The looking at the final output of the algorithms, we can see that the only visible seam is the one between camera two and three. Since the cameras are placed far apart, the light conditions is different for the cameras causing differences between the frames that the color correction method is not able to account for and remove.

As a general observation, we can see that the color corrector struggles with the current auto exposure settings used in the program. The three first test sets shows that the current camera exposure settings does not manage to capture frames with good color and luminance values. Since we want to have a fully automatic system, we also want to automate the task of configuring the cameras exposure settings. However, this is a work in progress and therefore the results from the new pipeline is not visually pleasing.

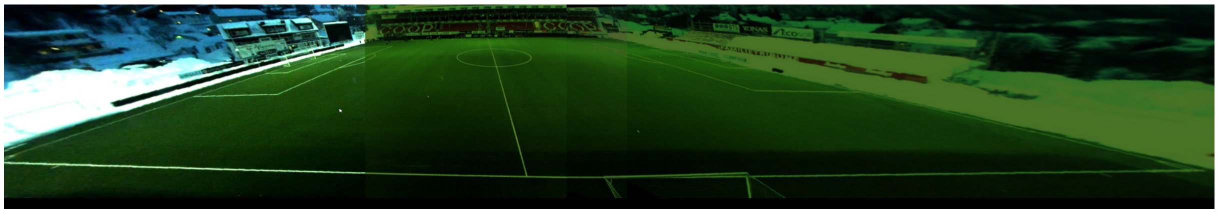
The current camera setup, similar to the one used in the prototype, also causes problems for the algorithm. The algorithm will also in many cases require some blending along the seam as previously mentioned in section 4.4.2, to completely remove it from the pipeline output.



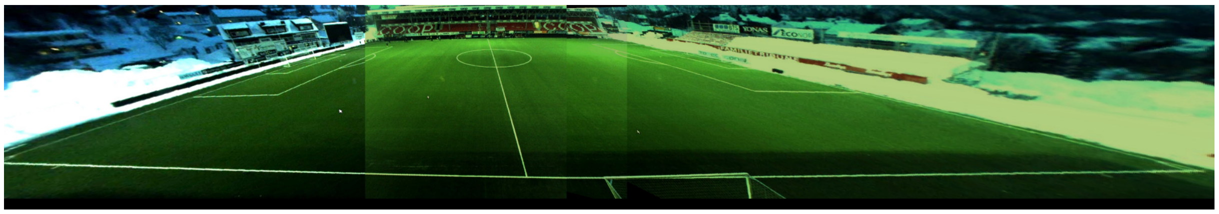
(a) The four frames delivered by the cameras.



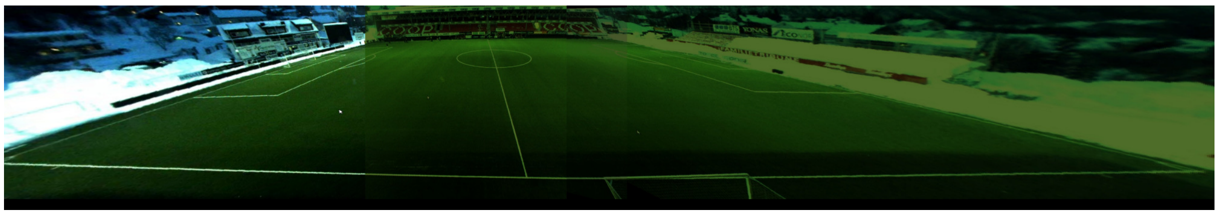
(b) Panorama without color correction.



(c) Color corrected panorama, original algorithm.

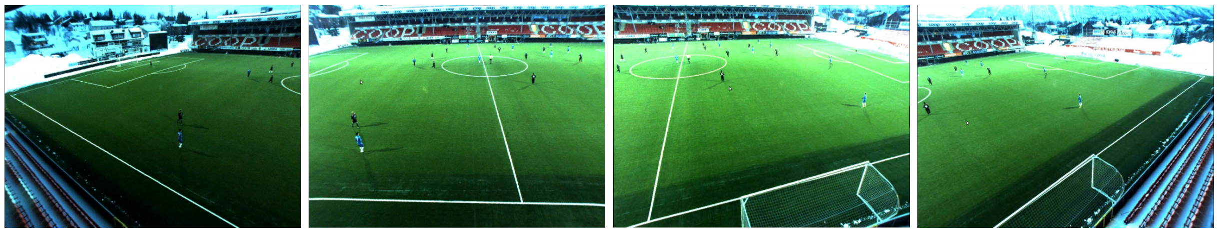


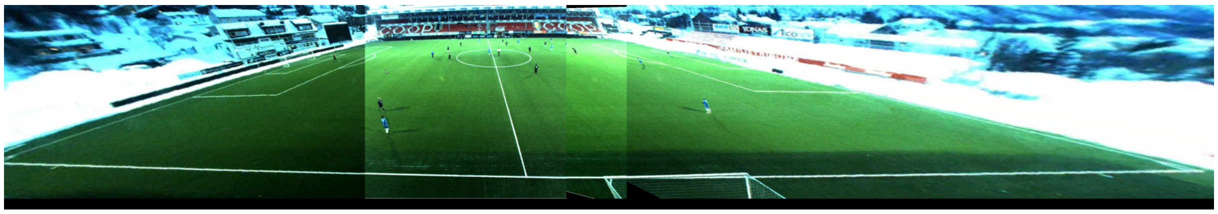
(d) Color corrected panorama, working in gamma corrected space.

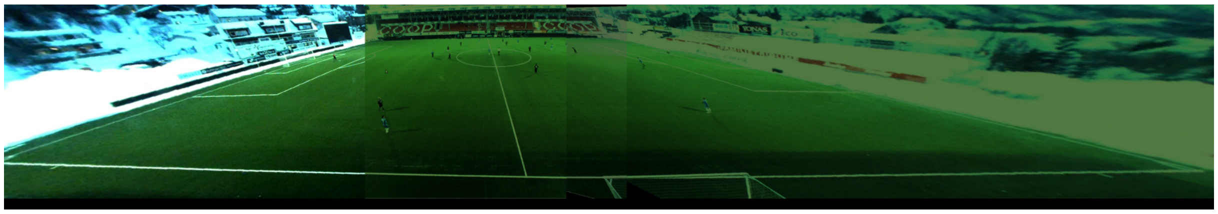


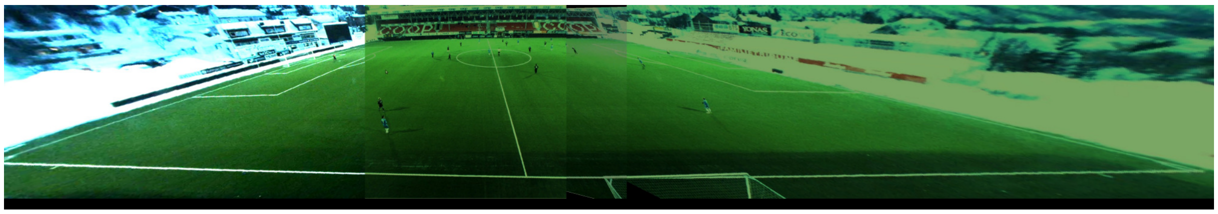
(e) Color corrected panorama, using pipeline version.

Figure 4.9: Color correction, test set one, outdoor scene.


(a) The four frames delivered by the cameras.


(b) Panorama without color correction.


(c) Color corrected panorama, original algorithm.


(d) Color corrected panorama, working in gamma corrected space.

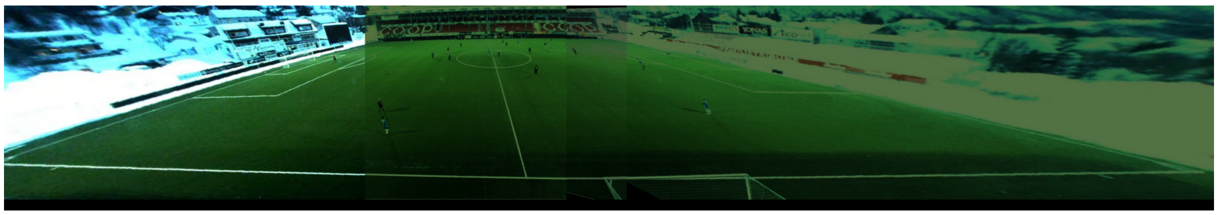
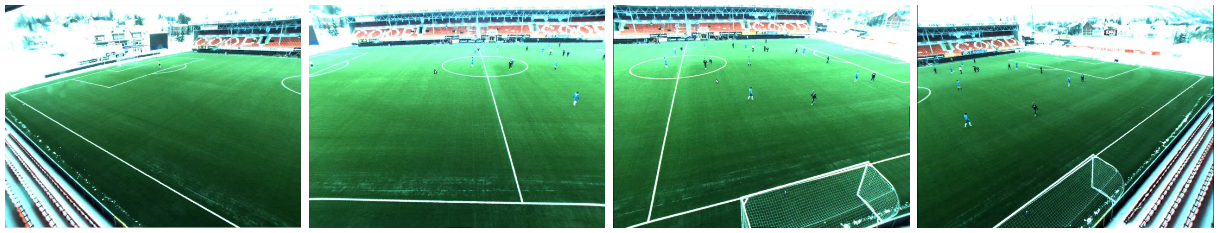
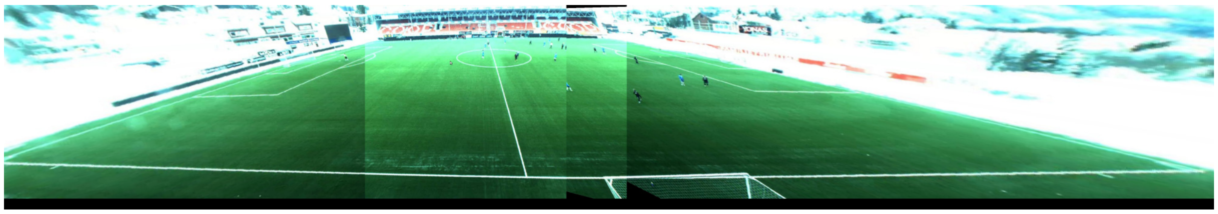
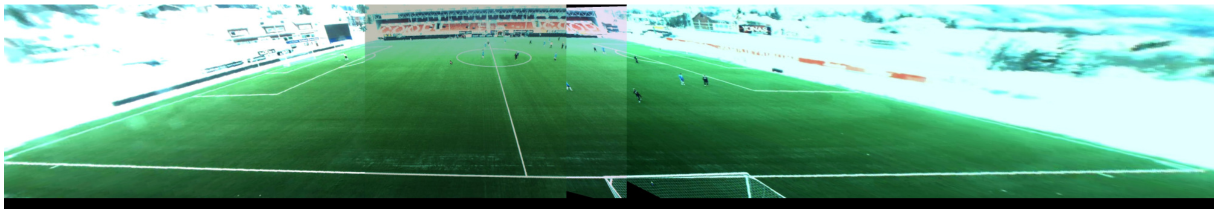
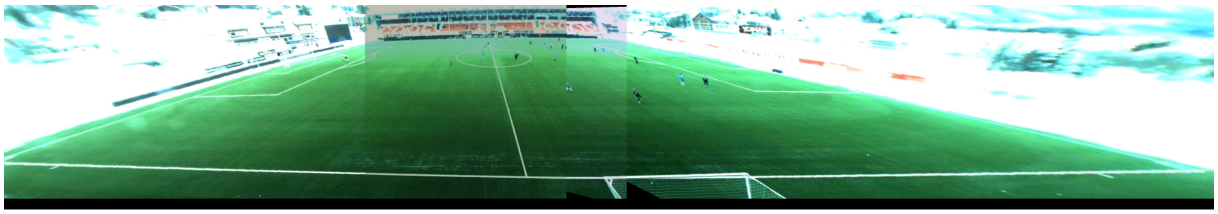

(e) Color corrected panorama, using pipeline version.

Figure 4.10: Color correction, test set two, outdoor scene.


(a) The four frames delivered by the cameras.


(b) Panorama without color correction.


(c) Color corrected panorama, original algorithm.


(d) Color corrected panorama, working in gamma corrected space.

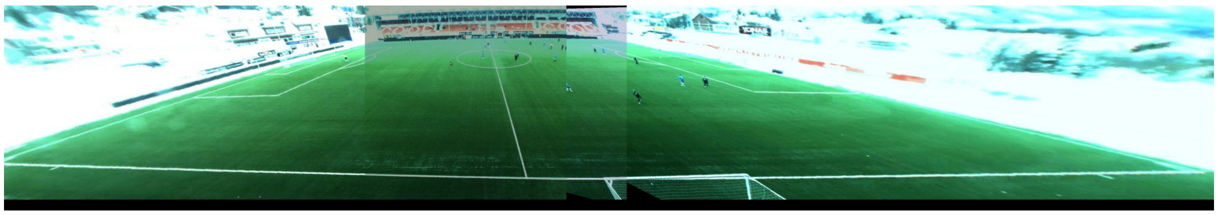
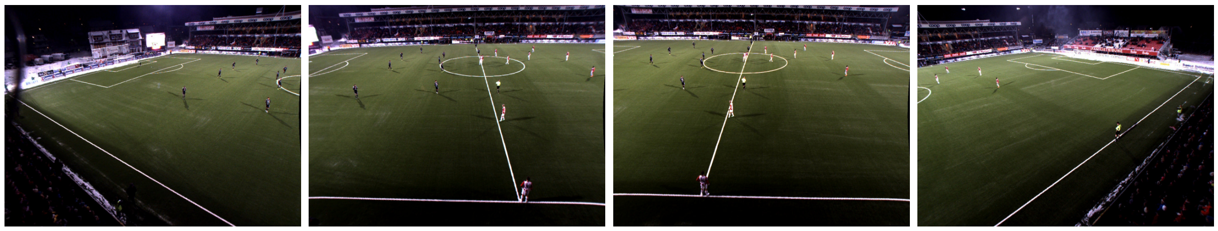
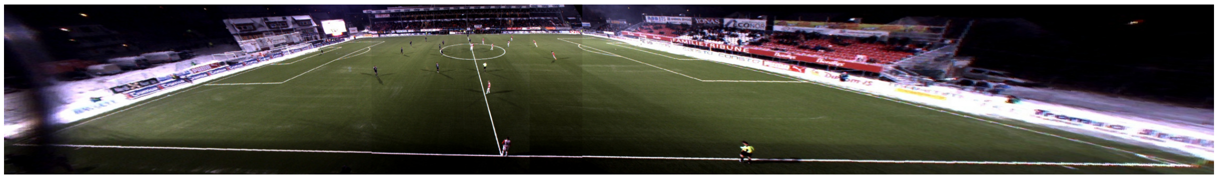
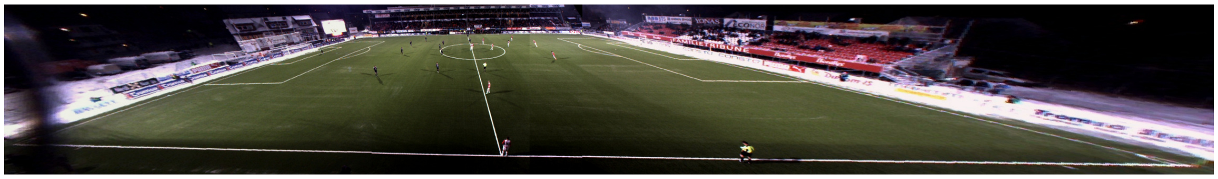
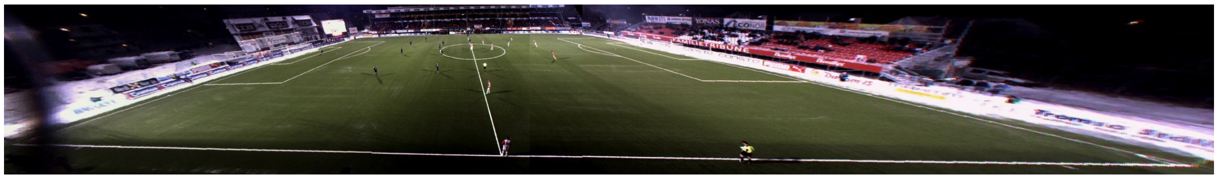

(e) Color corrected panorama, using pipeline version.

Figure 4.11: Color correction, test set three, outdoor scene.


(a) The four frames delivered by the cameras.


(b) Panorama without color correction.


(c) Color corrected panorama, original algorithm.


(d) Color corrected panorama, working in gamma corrected space.

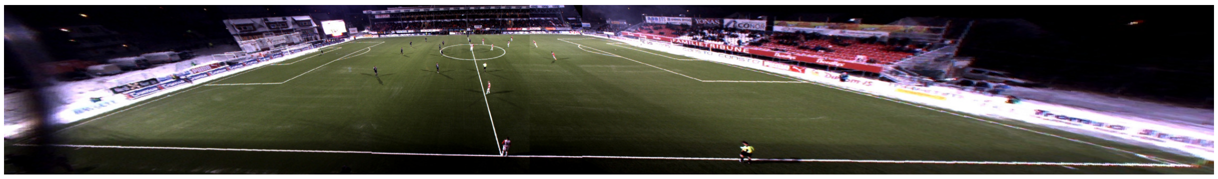

(e) Color corrected panorama, using pipeline version.

Figure 4.12: Color correction, test set four, outdoor scene, using manually set exposure settings.

4.9 Future Improvements

To further improve the color correction in Bagadus there are several things that can and should be done. Changing the camera setup should help minimize the differences in color and luminance between the camera frames. Future work on the camera settings should also be taken into consideration.

One could also look into a solution where you only color correcting the soccer field and avoid the sky and stands. Since the area of importance is the pitch, removing color correcting from the rest of the image can free resources that can be of better use in other parts of the pipeline. In situations as presented with the outdoor test sets, this can give a better output since the sky, already being white, will not be darkened when it should stay light.

Using blending in the pipeline should also be considered. This however, needs to be further researched since we use warping that results in ghosting effects when blending. A solution can be to use the background subtracter (described in section 3.6.7) to see when players are in the blending region, and when they are, avoid them.

4.10 Summary

We have presented a method for minimizing the color and luminance differences between the frames in the panorama. The algorithm implemented uses the overlapping region between the image to achieve its end result. By testing how the algorithm works on different sized of overlapping regions, we have been able to minimize the workload of the algorithm in the pipeline. After presenting and describing the current limitations of the algorithm, we have executed the method on several test sets, both indoor and outdoor to see the overall performance of the algorithm. Lastly, we discuss possible future improvements of the module to improve both computational speed, minimize resources and give a better visual output.

Chapter 5

Conclusion

Here we shortly describe the thesis, our contributions and future work that should be done to improve the system.

5.1 Summary

In this thesis, we have described how we have improved the prototype for Bagadus, a fully automatic sport analysis system by improving the subcomponent responsible for video capture. We start the thesis with a short chapter introducing Bagadus, its different system components and the first prototype. By combining the capabilities of a system for player tracking, an event annotation system and a video capture component, Bagadus is able to deliver all the information needed for team and player development. The systems opportunities is presented by discussing the first Bagadus demo while we also talk about the tools required by the system. The prototypes performance and implementation details is presented at the end of the chapter.

In chapter 3, we move on to the main scope of this thesis, creating a real-time panorama video. By taking advantage of both the CPU and the GPU, we are able to divide the program workload into different modules each responsible for doing its own set of image operations. Letting the modules run in parallel makes it possible to deliver panorama frames at the same rate as they are read from the cameras. The chapter also includes test results that show system limitations and bottlenecks, and gives an indication of the hardware requirements needed for the program to execute normally. At the end of the chapter, we present possible improvements and changes to the pipeline.

To improve the visual output of the pipeline, the possibilities for color correcting is presented and discussed in chapter 4. By comparing the difference in the images, we try to correct them and smoothen the overall color and luminance range over the entire panorama. We also present how the algorithm works under different lighting conditions and discuss the issues regarding current camera settings and setup, and how it effects the output of the pipeline. Lastly, we have discuss possible improvements for the color correction algorithm to make it better suited for our scenario.

5.2 Main Contributions

We have in this thesis presented a system for creating panorama video in real-time, a goal described in section 1.2. By using just four cameras, we have been able to capture and record video covering the whole field. This, in combination with other components for analytical purposes can give game deciding advantages in team sports like soccer.

By dividing the panorama creation workload between the CPU and GPU and using a design that divided the program into subcomponents, we where able to get the panorama creation pipeline up and running at our testbed in Tromsø. The real-life installation at our testbed shows that it is possible to create a real-time panorama video using limited hardware resources. As improving the visual output was one of the goals in section 1.2, we have shown that by implementing and using color correction and dynamic stitching the panorama becomes more visually pleasing.

In the prototype and the system version presented in this thesis we have shown that by combining an event system, a tracking system and a video capture component, we are able to provide analytical data in a short amount of time.

When working on the Bagadus system and creating a real-time panorama pipeline, we have been able to submit and publish a poster for the NVIDIA GPU Technology Conferences 2013 [28] as well as contributed to a paper delivered for evaluation to ACM Multimedia Conference. Furthermore, the system has has attracted a lot of attention and several other soccer clubs has shown interest in Bagadus.

5.3 Future work

Although the current pipeline can be considered real-time, there is still work that can and should be done. With regard to Bagadus as a whole, all of the components are still not working together in a way that fulfills the system goal and when it comes to the delivering part of the system, we still lack a proper and fully functional tool that can help the user take advantage of the system capabilities in real-time. Regarding the pipeline, possible future work is also discussed in section 3.9. Here, we present the possibilities for scaling the system and adding additional modules to further improve the performance, the quality of the outputted panorama and scalability of the system. One should also consider to split the program into even smaller subcomponents to prepare the program for future scaling. Another possible solution for improving the overall pipeline performance can be to go from using RGBA to RGB, giving us less data to process.

Lastly, the color correction is currently sensitive to large differences in color and luminance between the frames and struggles to remove the visible seams do to the lack of blending in the panorama creation. Using blending and other possible solutions to the issues of the color correction module is also discussed in section 4.9.

For the panorama, both the stitch and the color correction would benefit from an improved camera setup. Currently the cameras in combination with the lenses also delivers a less than impressive visual result, so a future task should be to improve the general output from the cameras.

Appendix A

Hardware Specifications

In the following pages we will present the different hardware specifications used when developing and testing Bagadus.

A.1 Test machines

Name	**TM1 (Test Machine 1)**
CPU	Intel Core i7-3930K @ 4.4 GHz
GPU	Nvidia Geforce GTX 680
Memory	32 GB DDR3 @ 1887 MHz
Pipeline output storage	Samsung SSD 840 Series, 500 GB

Table A.1: Test machine 1(TM1), specifications

Name	**TM2 (Test Machine 2)**
CPU	Intel Core i7-2600 @ 3.4 GHz
GPU	Nvidia Geforce GTX 460
Memory	8 GB DDR3 @ 1600 MHz
Pipeline output storage	Local NAS

Table A.2: Test machine 2(TM2), specifications

Name	**TM3 (Test Machine 3)**
CPU	Intel Core i7-960 @ 3.20GHz
GPU	Nvidia Geforce GTX 280
Memory	6 GB DDR3 @ 1066 MHz
Pipeline output storage	Local NAS

Table A.3: Test machine 3(TM3), specifications

Name	**TM4 (Test Machine 4)**
CPU	2x Intel Xeon E5-2650 @ 2.0 GHz
GPU	Nvidia Geforce GTX 580
Memory	64 GB DDR3 @ 1600 MHz
Pipeline output storage	Samsung SSD 840 Series, 500 GB

Table A.4: Test machine 4(TM4), specifications

A.2 GPU specifications

GPU	**Quadro NVS295**	**Geforce GTX 280**	**Geforce GTX 480**
CUDA cores	8	240	480
Graphics clock	540 MHz	602 MHz	700 MHz
Compute capability	1.1	1.3	2.0
Total memory size	256 MB GDDR3	1024 MB GDDR3	1536 MB GDDR5
Memory clock	695 MHz	1107 MHz	1848 MHz
Memory interface	64-bit	512-bit	384-bit
Memory bandwidth	11.2 GB/s	141.7 GB/s	177.4 GB/s
Compute	N/A	Tesla	Fermi

Table A.5: Graphic processing Unit specifications, 1

GPU	**Geforce GTX 580**	**Geforce GTX 680**	**Geforce GTX Titan**
CUDA cores	512	1536	2688
Graphics clock	722 MHz	1006 MHz	837 MHz
Compute capability	2.0	3.0	3.5
Total memory size	1536 MB GDDR5	2048 MB GDDR5	6144 MB GDDR5
Memory clock	4008 MHz	6000 MHz	6008 MHz
Memory interface	384-bit	256-bit	384-bit
Memory bandwidth	192.4 GB/s	192.2 GB/s	288.4 GB/s
Compute	Fermi	Fermi	Fermi

Table A.6: Graphic processing Unit specifications, 2

Appendix B

Detailed Performance Results

In this appendix, we present the performance result tables for more detail.

Module	Mean time (ms)
Controller	1.791
Reader	33.285
Converter	13.855
Debarreler	16.302
Uploader	23.892
Uploader, BGS part*	13.202
Background Subtracter	8.423
Warper	15.391
Color Correction	23.220
Stitcher	4.817
YUVConverter	9.938
Downloader	12.814
Single Camera Writer	24.424
Panorama Writer	19.998
Single Camera Writer, diff	33.339
Panorama Writer, diff	33.346
BGS, ZXY Query	657.597
Camera Frame Drops 1000	4
Pipeline Frame Drops 1000	0

Table B.1: Pipeline performance, running on TM1

Module	GTX280	GTX480	GTX580	GTX680	GTX TITAN
Uploader	73.036	27.188	23.269	23.302	22.426
Background Subtracter	36.761	13.284	8.193	8.240	7.096
Warper	66.356	19.487	14.251	15.141	13.139
Color Correction	86.924	28.753	22.761	22.800	19.860
Stitcher	23.493	8.107	5.552	4.912	4.126
YUVConverter	41.158	13.299	9.544	9.676	8.603
Downloader	53.007	16.698	11.813	12.221	11.452

Table B.2: GPU performance, running on TM1

Module	3.2GHz	3.5GHz	4.0GHz	4.4GHz
Controller	2.147	2.0061	2.342	2.344
Converter	15.003	14.741481	13.855	13.678
Debarreler	17.723	16.996955	16.523	15.874
Uploader	23.447	22.37323	21.545	20.605
SingleCamWriter	27.408	25.704795	24.767	23.008
PanoramaWriter	23.348	21.898721	21.325	17.819
Reader	33.277	33.289341	33.276	33.293
SingleCamWriter, diff	33.321	33.344086	33.303	33.340
PanoramaWriter, diff	33.314	33.339484	33.300	33.350

Table B.3: GPU core speed, running on TM1

Module	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Controller	4.023	4.103	3.898	4.107	3.526	3.906	3.717
Converter	18.832	16.725	15.170	13.601	12.635	12.874	12.319
Debarreler	27.469	19.226	16.903	14.573	13.171	12.659	12.106
Uploader	35.157	29.914	26.883	24.253	24.422	23.814	22.725
SingleCamWriter	40.752	30.160	26.754	23.776	22.416	21.800	21.173
PanoramaWriter	35.405	23.865	20.119	17.272	15.567	15.084	14.050
Reader	32.885	33.132	33.275	33.292	33.287	33.280	33.281
SingleCamWriter, diff	46.427	36.563	33.875	33.317	33.355	33.331	33.438
PanoramaWriter, diff	48.629	37.152	33.965	33.320	33.354	33.330	33.320

Table B.4: Core count scalability, running on TM4

Module	4 cores	4 cores, HT	8 cores	8 cores, HT	16 cores	16 cores, HT
Controller	4.257	4.023	4.044	3.898	3.391	3.717
Reader	32.392	32.885	32.947	33.275	33.278	33.281
Converter	14.041	18.832	13.319	15.170	11.164	12.319
Debarreler	24.840	27.469	16.808	16.903	10.453	12.106
Uploader	33.980	35.157	27.818	26.883	21.809	22.725
Uploader, BGS part	16.417	18.482	13.405	14.325	11.143	11.834
BGS	3.249	8.862	5.335	8.763	5.361	5.487
Warper	3.212	8.482	6.910	9.009	8.748	7.423
Stitcher	3.059	7.754	4.680	7.209	4.844	4.135
YUVConverter	3.277	8.344	5.705	8.384	6.016	5.716
Downloader	10.233	10.051	9.660	8.796	8.419	6.734
Single Camera Writer	53.313	40.752	31.290	26.754	20.023	21.173
Panorama Writer	53.544	35.405	29.613	20.119	16.903	14.050
SingleCamWriter, diff	63.642	46.427	38.845	33.875	33.323	33.438
PanoramaWriter, diff	67.494	48.629	39.831	33.965	33.319	33.320

Table B.5: Hyper Threading Performance, running on TM4

Module	4 cores	6 cores	8 cores	10 cores	12 cores	14 cores	16 cores
Controller	4.204	3.955	3.694	3.706	3.436	4.094	4.006
Converter	10.566	12.614	14.726	13.419	13.544	12.640	12.335
Debarreler	15.015	14.421	15.666	14.458	12.981	12.514	11.891
Uploader	19.857	23.015	26.076	25.008	24.137	23.554	23.487
SingleCamWriter	23.763	23.689	25.607	23.910	21.995	21.792	20.969
PanoramaWriter	20.187	18.908	19.163	17.771	15.286	14.497	13.695
Reader	33.037	33.070	33.266	33.290	33.301	33.286	33.277
SingleCamWriter, diff	38.070	34.782	33.661	33.352	33.327	33.358	33.324
PanoramaWriter, diff	38.724	35.019	33.715	33.353	33.319	33.366	33.323

Table B.6: Frame Drop Performance, running on TM4

Module	No optimizations	O2	O3
Controller	4.006	4.045	3.821
Reader	33.277	33.308	33.302
Converter	12.335	12.162	12.576
Debarreler	11.891	12.162	12.100
Uploader	23.487	17.336	17.377
Uploader, BGS part	11.644	5.644	5.399
SingleCamWriter	20.969	21.659	21.555
PanoramaWriter	13.695	14.695	14.797
SingleCamWriter, diff	33.324	33.327	33.321
PanoramaWriter, diff	33.323	33.323	33.317

Table B.7: Compiler Optimizations, running on TM4

Module	GPU (GTX 680)	CPU (i7-3930K @ 4.4Ghz)
Warper	15.141	133.882
Stitcher	4.912	521.042
Converter	9.676	26.52

Table B.8: Performance comparison between old and new Pipeline, running on TM1

Module	GPU (GTX 680)	CPU (i7-3930K @ 4.4Ghz)
BGS	9.95108	23.332083
Warper	14.111216	151.33
Stitcher	8.144514	592.236
Converter	13.386499	31.1994

Table B.9: Performance comparison between CPU and GPU implementations, running on TM1

Appendix C

Compiler Optimizations

For further improvements on the pipeline, we wanted to see the optimizations made possible with different GCC compiler options. The diagram in figure C.1 shows the different optimizations flags we tested for the pipeline, O2 and O3. Note that the GPU modules is not shown in the table since GCC, being a CPU compiler, cannot optimize device code. Using external libraries also minimizes the total improvements of the pipeline, as many of the CPU modules make use of them for encoding, conversion and more. The only module that does not make use of external libraries, and sees a increase in speed performance is the Uploader module. The module is about 25% faster with GCC optimizations, regardless of optimization flag (O2 or O3). Note that using O3 makes no difference in the pipeline as appose to applying O2 to optimize.

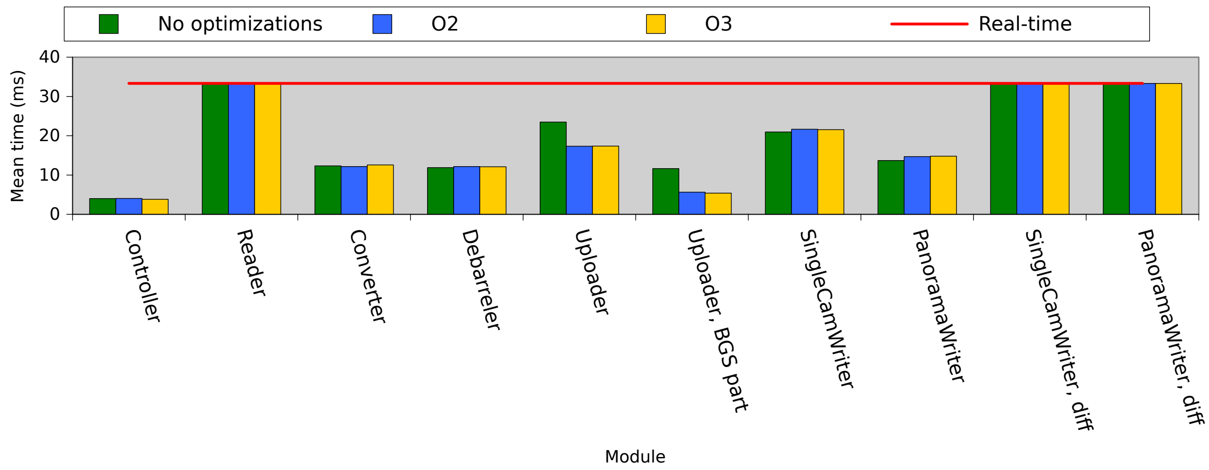


Figure C.1: Pipeline performance with different compiler optimizations; O2 and O3.

Appendix D

Access the source code

To access the Bagadus source code, the panorama pipeline and other projects related to Bagadus go to `https://bitbucket.org/mpg_code/bagadus`. To retrieve the code,
run *git clone git@bitbucket.org:mpg_code/bagadus.git*.

Bibliography

[1] 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.

[2] Yuv. `http://en.wikipedia.org/wiki/YUV`. Accessed April 24, 2013.

[3] Cuda toolkit documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[4] Interplay sports. http://www.interplay-sports.com/.

[5] Prozone. http://www.prozonesports.com/.

[6] Stats Technology. http://www.sportvu.com/football.asp.

[7] ZXY Sport Tracking. http://www.zxy.no/.

[8] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[9] Anat Levin, Assaf Zomet, Shmuel Peleg, and Yair Weiss. Seamless image stitching in the gradient domain. *Computer Vision-ECCV 2004*, pages 377–389, 2004.

[10] Jiaya Jia and Chi-Keung Tang. Image stitching using structure deformation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(4):617–631, 2008.

[11] Alec Mills and Gregory Dudek. Image stitching with dynamic elements. *Image and Vision Computing*, 27(10):1593–1602, 2009.

[12] Yao Li and Lizhuang Ma. A fast and robust image stitching algorithm. In *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, volume 2, pages 9604–9608. IEEE, 2006.

[13] Camargus - Premium Stadium Video Technology Inrastructure. http://www.camargus.com/.

[14] Software stitches 5k videos into huge panoramic video walls, in real time. `http://www.sixteen-nine.net/2012/10/22/software-stitches-5k-videos-huge-panoramic-video-walls-real-time/`, 2012. [Online; accessed 05-march-2012].

[15] Live ultra-high resolution panoramic video. `http://www.fascinate-project.eu/index.php/tech-section/hi-res-video/`. [Online; accessed 04-march-2012].

[16] O. Schreer, I. Feldmann, C. Weissig, P. Kauff, and R. Schafer. Ultrahigh-resolution panoramic imaging for format-agnostic video production. *Proceedings of the IEEE*, 101(1):99–114, Jan.

[17] Wai-Kwan Tang, Tien-Tsin Wong, and P-A Heng. A system for real-time panorama generation and display in tele-immersive applications. *Multimedia, IEEE Transactions on*, 7(2):280–292, 2005.

[18] Michael Adam, Christoph Jung, Stefan Roth, and Guido Brunnett. Real-time stereo-image stitching using gpu-based belief propagation. 2009.

[19] Wei Xu and J. Mulligan. Performance evaluation of color correction approaches for automatic multi-view image and video stitching. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 263–270, 2010.

[20] David Hasler and Sabine SÃ¼sstrunk. Mapping colour in image stitching applications. *Journal of Visual Communication and Image Representation*, 15(1):65 – 90, 2004.

[21] A. Au and Jie Liang. Ztitch: A mobile phone application for immersive panorama creation, navigation, and social sharing. In *Multimedia Signal Processing (MMSP), 2012 IEEE 14th International Workshop on*, pages 13–18, 2012.

[22] Yingen Xiong and Kari Pulli. Color correction for mobile panorama imaging. In *Proceedings of the First International Conference on Internet Multimedia Computing and Service*, ICIMCS '09, pages 219–226, New York, NY, USA, 2009. ACM.

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

[24] 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.

[25] 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.

[26] Bagadus - an integrated system for soccer analysis. `http://www.youtube.com/watch?v=1zsgvjQkL1E`.

[27] Peter Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen B. Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline: preliminary report of the acm task force on the core of computer science. In *Proceedings of the nineteenth SIGCSE technical symposium on Computer science education*, SIGCSE '88, pages 41–41, New York, NY, USA, 1988. ACM.

[28] Marius Tennøe, Espen O Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Pål Halvorsen, and Carsten Griwodz. Realtime panorama video processing using nvidia gpus. GPU Technology Conference, March 2013.

[29] Alper Yilmaz, Omar Javed, and Mubarak Shah. Object tracking: A survey. *ACM Comput. Surv.*, 38(4), December 2006.

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

[31] Basler cameras. http://www.baslerweb.com/.

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

[33] 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/.

[34] Videolan - x264, the best h.264/avc encoder. http://www.videolan.org/developers/x264.html.

[35] Ffmpeg. http://ffmpeg.org/.

[36] 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.

[37] Itseez. Opencv | opencv. http://opencv.org/, 2013.

[38] C. A. Glasbey and K. V. Mardia. A review of image-warping methods. *Journal of Applied Statistics*, 25(2):155–171, 1998.

[39] Patrick Baudisch, Desney Tan, Drew Steedly, Eric Rudolph, Matt Uyttendaele, Chris Pal, and Richard Szeliski. An exploration of user interface designs for real-time panoramic photography. *Australasian Journal of Information Systems*, 13(2):151, 2006.

[40] Ding-Yun Chen, MurphyChien-Chang Ho, and Ming Ouhyoung. Videovr: A real-time system for automatically constructing panoramic images from video clips. In Nadia Magnenat-Thalmann and Daniel Thalmann, editors, *Modelling and Motion Capture Techniques for Virtual Environments*, volume 1537 of *Lecture Notes in Computer Science*, pages 140–143. Springer Berlin Heidelberg, 1998.

[41] Kevin Huguenin, Anne-Marie Kermarrec, Konstantinos Kloudas, and Francois Taiani. Content and geographical locality in user-generated content sharing systems. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 2012.

[42] Omar A. Niamut, Rene Kaiser, Gert Kienast, Axel Kochale, Jens Spille, Oliver Schreer, Javier Ruiz Hidalgo, Jean-Francois Macq, and Ben Shirley. Towards a format-agnostic approach for production, delivery and rendering of immersive media. In *Proceedings of Multimedia Systems (MMSys)*, pages 249–260, February 2013.

[43] Christian Weissig, Oliver Schreer, Peter Eisert, and Peter Kauff. The ultimate immersive experience: Panoramic 3d video acquisition. In Klaus Schoeffmann, Bernard Merialdo, AlexanderG. Hauptmann, Chong-Wah Ngo, Yiannis Andreopoulos, and Christian Breiteneder, editors, *Advances in Multimedia Modeling*, volume 7131 of *Lecture Notes in Computer Science*, pages 671–681. Springer Berlin Heidelberg, 2012.

[44] Nvidia. *NVIDIA Performance Primitives*, september 2012.

[45] Jason Garrett-Glaser. git.videolan.org git - x264.git blob - doc threads.txt. git://git.videolan.org/x264.git, 2010.

[46] Z. Zivkovic. Improved adaptive gaussian mixture model for background subtraction. In *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on*, volume 2, pages 28–31 Vol.2, 2004.

[47] 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.

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

[49] Mark Allen Weiss. *Data Structures and Problem Solving Using Java (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[50] 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, 2013.

[51] Grep command. http://linux.about.com/od/commands/l/blcmdl1_grep.htm.

[52] PS command. http://linux.about.com/od/commands/l/blcmdl1_ps.htm.

[53] Linux / Unix Command: kill. http://linux.about.com/od/commands/l/blcmdl1_kill.htm.

[54] nVIDIA. Nvidia's next generation cuda compute architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`, 2010. [Online; accessed 17-march-2013].

[55] nVIDIA. Nvidia's next generation cuda compute architecture: Kepler gk110. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. [Online; accessed 04-april-2013].

[56] Taskset command. http://linuxcommand.org/man_pages/taskset1.html.

[57] Intel Hyper-Threading Technology. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[58] MuhammadTwaha Ibrahim, Rehan Hafiz, MuhammadMurtaza Khan, Yongju Cho, and Jihun Cha. Automatic reference selection for parametric color correction schemes for panoramic video stitching. In George Bebis, Richard Boyle, Bahram Parvin, Darko Koracin, Charless Fowlkes, Sen Wang, Min-Hyung Choi, Stephan Mantler, JÃ¼rgen Schulze, Daniel Acevedo, Klaus Mueller, and Michael Papka, editors, *Advances in Visual Computing*, volume 7431 of *Lecture Notes in Computer Science*, pages 492–501. Springer Berlin Heidelberg, 2012.

[59] C. Doutre and P. Nasiopoulos. Fast vignetting correction and color matching for panoramic image stitching. In *Image Processing (ICIP), 2009 16th IEEE International Conference on*, pages 709–712, 2009.

[60] Gui Yun Tian, D. Gledhill, D. Taylor, and D. Clarke. Colour correction for panoramic imaging. In *Information Visualisation, 2002. Proceedings. Sixth International Conference on*, pages 483–488, 2002.

[61] Linearity and Gamma. `http://www.arcsynthesis.org/gltut/Illumination/Tut12\%20Monitors\%20and\%20Gamma.html`.

[62] Libpng Appendix: Gamma Tutorial. http://www.libpng.org/pub/png/spec/1.2/PNG-GammaAppendix.html.