UNIVERSITY OF OSLO
Department of Informatics

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching

Master's thesis

Espen Oldeide Helgedagsrud

Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching

Espen Oldeide Helgedagsrud

Abstract

The Bagadus system has been introduced as an automated tool for soccer analysis, and it is built up by an analysis subsystem, tracking subsystem and video subsystem. Bagadus allows for simplified soccer analysis, with the goal of improving athletes' performance, by automating the integration of these subsystems. The system is currently installed at Alfheim stadium in Tromsø, Norway. An important part of the video subsystem is the creation of panorama videos from four HD cameras. However, the stitching pipeline for panorama video generation in the first version of the system did not manage to do this in real-time.

In this thesis, we present how to build an improved panorama stitcher pipeline that is able to stitch video from four HD cameras into a panorama video in real-time. We will detail the architecture and modules of this pipeline, and analyze the performance. In addition we will focus on the stitching component, and how that can improve the overall visual quality of the output panorama.

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, who have been a great help, providing feedback, discussions, guidance and advice for the development of the Bagadus system and this thesis. In addition, Kai-Even Nilsen for all help with installation and support at Alfheim.

I would also like to thank and acknowledge the work done by Marius Tennøe, Mikkel Næss, Henrik Kjus Alstad and Simen Sægrov, who all helped create and improve the Bagadus system. They have all been invaluable sources for discussions, feedback, advice and help during this project.

Finally, I wish to thank my family and friends for all their support.

Oslo, April 2013
Espen Oldeide Helgedagsrud

Chapter 1

Introduction

1.1 Background

Today, a great number of sports clubs invest time and resources into analyzing their game performance. By allowing trainers and coaches access to vital game information, either manually or through automated systems, the performance of the players and the team can potentially be greatly improved. In soccer, these analysis tools have always played an important role, and examples of pre-existing ones are Interplay [1], Prozone [2], STATS SportVU Tracking Technology [3] and Camargus [4].

These systems all provide a wide range of different game and player data (e.g., player speed, heart rate, fatigue, fitness graphs, etc.), but some also contain video and annotation subsystems. For instance, the system from Interplay supports game annotations through a trained operator, allowing interesting and useful situations in the game to be correctly marked and played back. The SportVU system on the other hand operates directly on video, and allows tracking of players using only camera feeds. While using only video to track players works, it is very demanding and often has less than ideal accuracy. For better tracking of players, a system like ZXY Sports Tracking [5] can be used. This system uses radio transmitters on each player to detect absolute positions on the field, but also additional give additional data such as speed and heart rate.

A common use of these analysis systems is the ability to play back video of important events in the game. These can then be used by the coach to give visual feedback about these situations directly to the team. There exist several systems for such solutions, like having a dedicated camera man to catch these events occurring, which can prove expensive both in equipment cost and man-hours. More common perhaps is the use of several cameras to record the entire field at the same time, thus capturing all important events regardless. A solution like this would also allow for creation of a stitched panorama image of the entire field, which can give a very good overview of everything going on. Camargus is an example of such a system which utilizes 16 cameras to provide a full panorama view of the entire field. Camargus does, however, not currently support any form of annotation system.

All these systems cover different subsystems, so for proper analysis several of these elements should be integrated together, which currently requires a lot of manual steps. To address this, we introduced Bagadus [6]. Bagadus is a fully automated system that

provides a camera array for video capture, a sensor system for player positions and statistics, and support for event annotations. These events can both be generated from an expert operator, or automatically based on data from the sensors, and coupled with the video system it allows for instant playback when needed. The positional data from our sensors also allow us to pinpoint player location both on the field and also in our video streams, thus enabling video tracking of individual (or a group of) players. For playback, Bagadus supports both single camera footage, but also a stitched panorama of the entire camera array. The generation of this panorama was initially designed to be performed in real-time, but the current implementation is far from optimal and performes nothing close to this constraint. The resulting panorama in this implementation is also of rather poor quality, with several introduced visual artifacts.

1.2 Problem Definition

The main goal in this work is to improve the performance and output of the Bagadus panorama stitching pipeline. Prior work on such stitching already exist, as seen in [7–11] and also in Camargus [4]. However all of these have various issues (such as expensive equipment, low visual quality and performance), making them non-ideal for our project, .

In this thesis we will look at how we can improve the old Bagadus panorama stitcher pipeline, both with regards to performance and visual quality. We will look in great detail at all the modules that make up the system, but our emphasis will lie on the stitcher component. The improved pipeline we describe in this thesis is currently installed and in use at Alfheim Stadium, Tromsø, Norway.

To improve the performance, we investigate how the existing system can be split in to several individual modules, and then sequentially assembled into a pipeline. As part of the modules, we will also look into how we can use heterogeneous processing architectures to speed up parallelizable tasks and achieve even better performance. The overall goal is to create a full panorama stitching pipeline that can do on-line processing of four camera streams in real-time. Along the way, we will investigate the architectural changes needed to support this ambition, and also the modules and algorithms required for it to work.

As for improving the visual quality, we will introduce several specific modules with this very goal in mind. Our main emphasis, however, is on the stitcher module, and how that can be improved to offer a much higher quality panorama output for our scenario. We will detail the stitcher from the very first static version and then investigate how it can be gradually improved to a much more powerful dynamic version.

1.3 Limitations

In the first implementation of our stitcher [6], the selection of algorithms for the panorama stitching process is discussed in great detail. We therefore use these and do not spend time going further into how these works as it is beyond the scope of this thesis. We do, however, research how many of them can be improved, both in regards to speed and visual quality.

1.4 Research Method

In this thesis, we evaluate the design and implementation of the improved Bagadus system prototype. The system is currently deployed in a real life scenario at Alfheim stadium in Tromsø. As research method, we use an equal to the *Design* paradigm, as described by the ACM Task Force on the Core of Computer Science [12].

1.5 Main Contributions

The main contribution of this thesis has been the creation and installation of an improved panorama stitcher pipeline as part of the Bagadus system at Alfheim. The new system stores both stitched and non-stitched footage, and performs fast enough to fulfill the real-time requirement imposed on the system. This is all achieved on a single high-end computer with commodity hardware. We have also improved the visual fidelity of the final output panorama, both by adding extra pipeline modules for this purpose and by improving pre-existing ones. Introducing a new dynamic stitcher, we have also improved the quality of the seam used to make the final panorama. By implementing this pipeline, we have shown that it is possible to make a real-time system for generating video panorama using a large amount of data by using external processing units such as GPUs.

We have also been able to submit and publish a poster at the GPU Technology Conference 2013, in which our system was presented and described [13]. In addition we have also submitted a paper to ACM Multimedia 2013 [14] detailing our pipeline.

1.6 Outline

In chapter 2, we start by describing the existing Bagadus system. Since our improvements are all made from this old setup, it is important to have a proper understanding of how it originally was designed and implemented. Following this, we move on to our actual improvements, by detailing our improved pipeline step-by-step in chapter 3. This will go deeper into each module, and detail exactly how they work. In chapter 4, we describe our emphasized module, namely the stitcher. Here, we take a look at its inner workings, and how it evolved from the initial version to what it is today. In the final chapter (5), we summarize our findings, and look at some future work.

Chapter 2

Bagadus

To make stitched panorama video a prototype pipeline was created. This first implementation was named Bagadus and it consists of two programs, the main pipeline part (used for generating our video) and a player (used to play back the recorded video). In this chapter we will investigate the specifics of this prototype, how everything is set up and how the components work together. Finally, we will look at the actual programs themselves, the pipeline and the player.

2.1 The Basic Idea

As discussed in chapter 1 and explained in [6, 15, 16], existing soccer analysis systems, like Camargus and SportVU, contain several subsystems, such as video recording and annotation. A problem with these systems is that they require manual steps to integrate into a larger system. These manual steps can be error prone and can introduce performance overhead, making the system much slower than a fully automated one.

The basic idea of Bagadus is therefore to integrate all components and subsystems needed for a soccer analysis system into a fully automated system. To be able to do so, Bagadus contains three main subsystems: The video subsystem, which records and stores footage of the game, the analytical subsystem, which allows for tagging and storing events, and the tracking subsystem, which tracks players, player data and statistics. A diagram of the general Bagadus architecture can be seen in figure 2.1. Here we see the video subsystem consisting of several cameras covering the whole field, and also the storage pipelines for both panorama and single camera footage. The tracking system is shown as several antennas surrounding the field, which collects player data and position from the sensors the players are wearing. The analytical subsystem is shown as a coach annotating events using a smart phone during a game session.

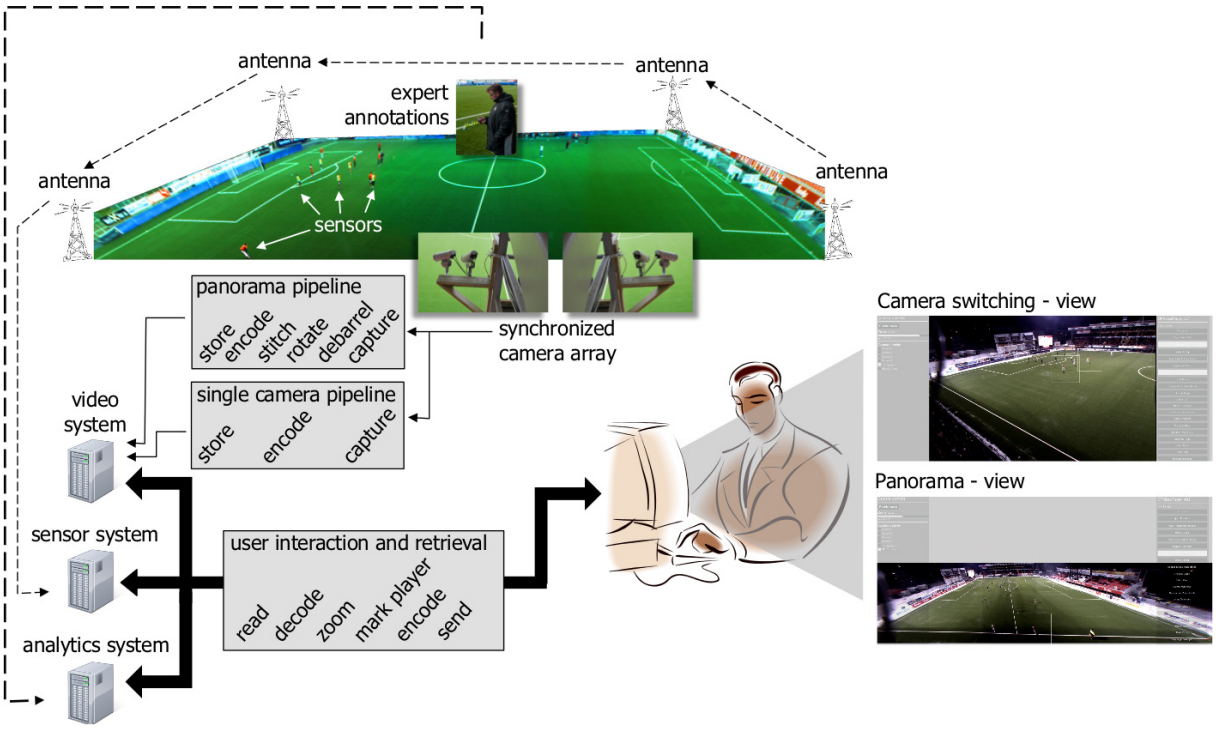


Figure 2.1: Bagadus setup at Alfheim

2.2 Video Subsystem

Video capture is one of the most important parts of Bagadus, as it provides the data we need in all following steps. In this section we will look closer at how our cameras are set up and how the captured frames are synchronized.

2.2.1 Camera Setup

The camera setup at Alfheim stadium consists of four Basler Ace A1300 - 30gc cameras (appendix A.3.1) set up to cover the whole soccer field. The cameras are mounted inline, two on each side of the stadium's center television gantry (used by broadcasters during matches). To make the cameras combined field of view cover the whole field; we use 3.5mm wide angle lenses. This gives us the coverage we need, but introduces an optical phenomenon called barrel distortion that must be corrected for (covered in section 2.4.3).

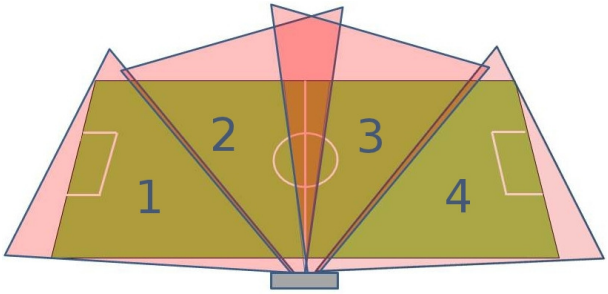


Figure 2.2: Camera setup

2.2.2 Frame Synchronization

Since we are stitching together frames it is essential that all four frames received from the cameras are taken at the exact same time. To synchronize the cameras we use an external device called a triggerbox. The triggerbox is custom built at Simula for this project and contains circuitry to send a shutter pulse to the cameras at a selectable rate. In our case the triggerbox is set for 30 Hz, giving us 30 frames a second from the cameras. Each triggerbox has output ports to support up to 4 cameras, but these boxes can be daisy-chained to allow for even more outputs.

This first prototype pipeline used two dedicated recording machines, both running two cameras, to capture our video. To be able to match up the exact timestamps of the frames between the two machines, a custom made TimeCodeServer was set up to synchronize the internal clocks of both machines.

2.3 Analytics and Tracking Subsystems

To make Bagadus a useful tool for coaches, both an analytics and a tracking subsystem is supported. The analytics is in form of an annotations system support integrated in our Bagadus player, and the tracking is positional data delivered from a radio tracking solution. We will describe these systems, namely Muithu and ZXY, in this section.

2.3.1 Muithu

Muithu [17] is a coach annotation system currently in use at Alfheim stadium. Muithu allows coaches to interactively annotate a game in progress using a smartphone application and automatically synchronize these events with the corresponding video. The current Muithu setup is currently off-line, so annotations and video are captured separately and then recombined during pauses in the game (half-time or end of match).

Since Muithu already is supported and in use on Alfheim, it was always a goal to get support for it in Bagadus. The player for playing back data from the prototype pipeline has some basic support for Muithu events, albeit more as a proof of concept than a real usable implementation.

2.3.2 The ZXY System

To be able to monitor individual players on the field, a tracking solution by ZXY Sport Tracking AS (ZXY) [5] is installed on Alfheim stadium. The solution is based on wireless sensor belts (figure 2.3(a)) that each player wear around his waist and that sends out data on the 2.45GHz band. Around the stadium, there are several big antennas (figure 2.3(b)) used to pick up these signals, and use them to triangulate the exact position of the player. The belts can also transmit other information, like the players step frequency, heart rate and speed.

(a) Sensor belt used in the ZXY-system

(b) One of the ZXY-antennas at Alfheim stadium

Figure 2.3: ZXY equipment

The ZXY Sensor Data

Positional data from the ZXY system at Alfheim gets sampled at a rate of 20Hz and then stored in an onsite SQL-database. The position is based on ZXY's own coordinate system, and is stored as a 2d-point (x,y). According to ZXY [5] the system itself now delivers up to centimeter accuracy, but the older system installed at Alfheim only has an accuracy of ± 1 meter.

Each belt has their own unique id, and this id gets mapped to a player name through the database. There is no automation for this, so this must be done manually before each game in order to connect player names with positions. Events such as the kickoff and half-time are also manually inserted into the data, to enable synchronization between video and the positional data.

Mapping ZXY Sensor Data to Pixels

As mentioned the data received is a 2d-point in a coordinate system. The coordinates are based on real-life field measurements of a horizontal soccer field where *(0,0)* is upper left corner, and bottom right is the size of the field (width, height). Alfheim measures 105×68 meters, so the effective data range is *(0,0)* to *(105,68)*. Since the measurements of a soccer field are standardized and well known (Figure 2.4), we are able to map players to an exact 2d field position using the ZXY data.

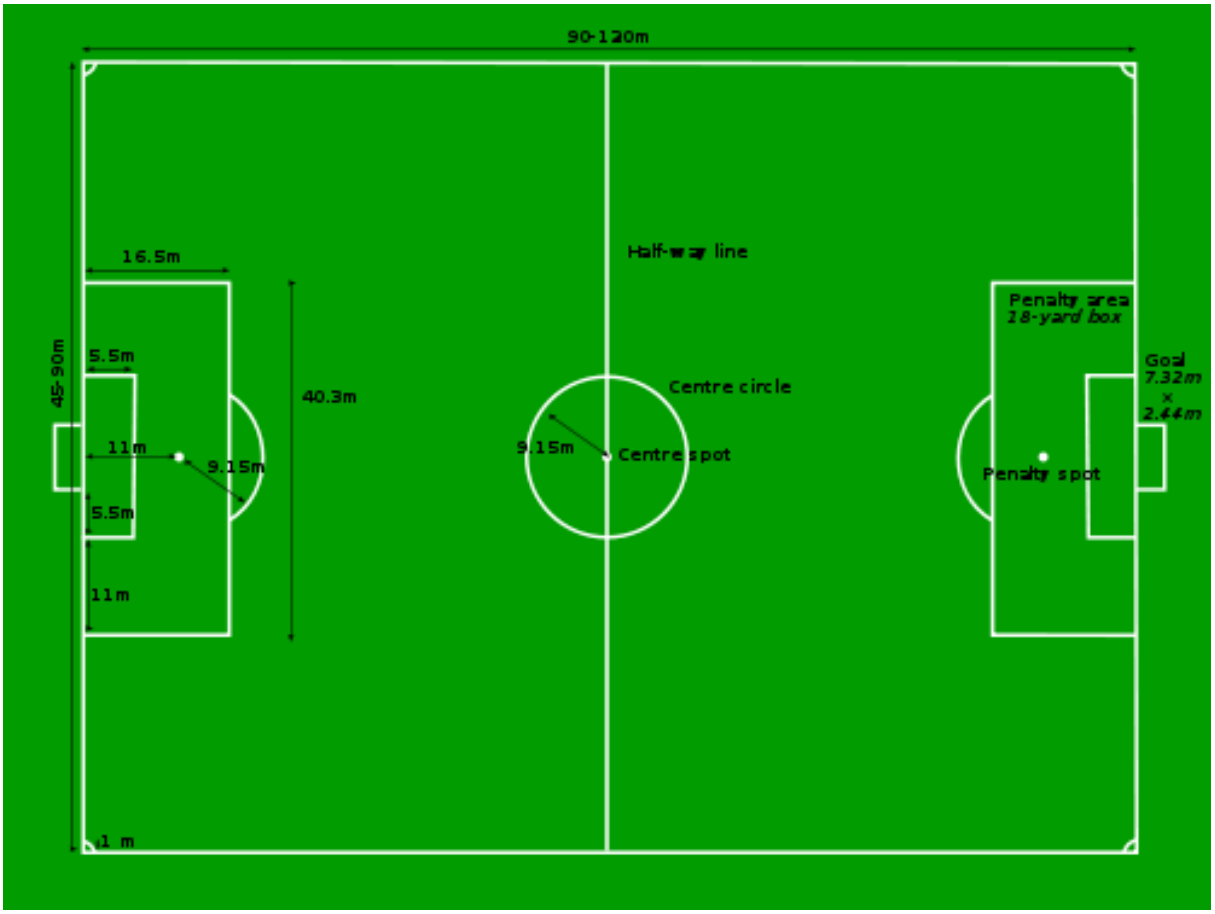


Figure 2.4: Stadard football field measurments

Since our cameras are not placed directly over the field pointing down we can not use these coordinates directly to get player positions in our camera frames. Warping the data in a way to make that kind of mapping possible is, however, doable and we will look at how that is done later.

Possibilities Using ZXY

While the ZXY-system was originally intended as an analytics tool for coaches, there are many other useful applications of such a setup. For instance, by knowing the positions of players on the field and how to translate it to the video stream one can track individual players across several cameras, and also provide area crops or digital zoom on one or more players. In Bagadus ZXY is primarily used in the player (section 2.5), but we will later look at how we integrate it for more uses as part of our improved pipeline setup.

2.4 The Bagadus Prototype Stitching Pipeline

The prototype panorama stitcher was created to generate a full panorama video stream from our four camera setup. It is made up of several modules connected together to form a pipeline, where each module passes its processed frame to the next and so on. Having such a modular setup means that improvements and modules easily can be inserted or swapped out should the need arise. In this section, we will discuss how this pipeline works, from the software it uses to the modules themselves.

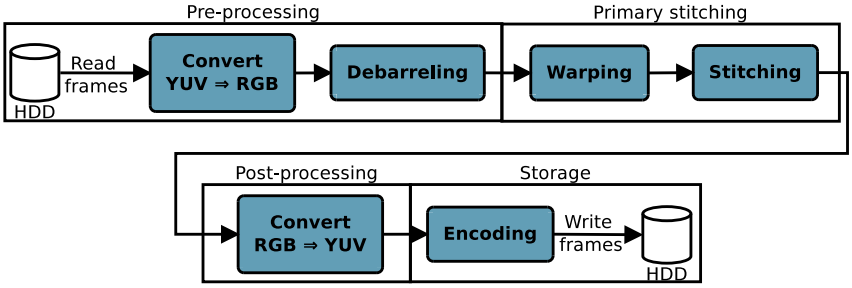


Figure 2.5: The Bagadus prototype pipeline architecture

2.4.1 Software Frameworks

To be able to work on the captured camera data, we first need to interface the cameras, get the data and then do some pre-processing for the image to look right. To do this, we use two different software libraries suited for the job.

Northlight

Northlight (libNorthlight) is a framework for capturing and streaming video developed internally here at Simula. Northlight's most important job in our pipeline is to interface the Basler capture API, and provide the frame data we need directly from the cameras. Northlight also handles conversion between some frame formats (like YUV and RGB), and encoding our frame data to video. It should be noted that Northlight itself does not handle all these operations itself, but instead wraps common tools and libraries (like *swscale* [18] for conversion and *x264* [19] for encoding) that does the

actual work. In such a way, Northlight can be seen more of as a high level helper framework than a common C++ library.

Northlight contains special data types which we use throughout the whole bagadus pipeline for working on single video frames. Northlight also provides built in functions for converting between these structures and the more general *cv::mat* format used by OpenCV.

OpenCV

OpenCV (Open Source Computer Vision Library) [20] is an open source general computer vision library for C++. OpenCV contains a lot of general purpose functions and algorithms useful for image manipulation and computation. In the Bagadus pipeline OpenCV is used to remove the effects of barrel distortion from our input frames (a process called debarreling), which occurs due to our cameras wide angle lenses (more detailed at 2.4.3). OpenCV is also used to calculate and warp the frames correctly for our output panorama (see 2.4.3).

OpenCV has a primitive image data type called *cv::mat* that we use for all OpenCV operations. This format also allow us to dump frames directly to disk as jpeg, which helps immensely for debugging and visual inspection along the pipeline. Since libNorthlight can convert its internal data types directly to *cv::mat* and we basically use either of these formats along the entire Bagadus pipeline, we are able to save frames from any part of it to disk as needed.

2.4.2 Color Formats

Two different color formats are in use in all our systems, and should therefore be explained properly. The cameras themselves can deliver a multitude of different formats through their API, but in our case YUV is used. Internally however most of our functionality is made to work on the RGB color format, so conversion is required (detailed in section 3.7.2). We will only touch briefly upon the two formats here, for a much more in-depth look at them see [6].

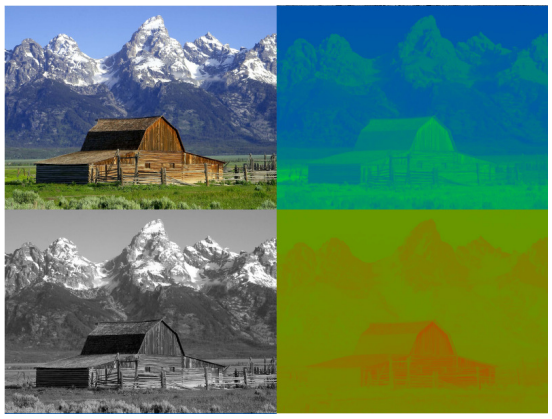
RGB and RGBA

RGB is a commonly used color format and represents a color as an additive blend of the three different primary colors (Red, Green and Blue, hence the name). RGBA is RGB with an additional alpha channel for setting pixel translucency. RGB supports several color depths, but the most common variant (called "True Color") is using 8 bits per channel resulting a total of 24 bits per pixel for RGB and 32 for RGBA. Since we use 32-bit "True Color" RGBA for all RGB operations in our pipeline any future reference to "RGBA" will now mean this format.

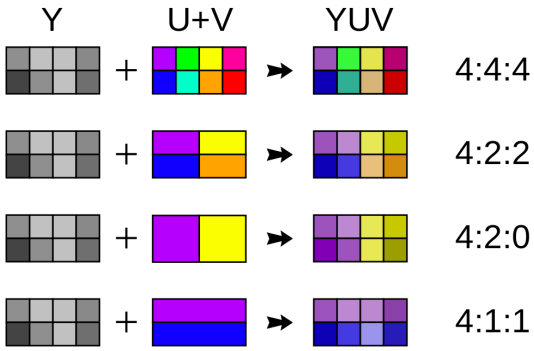
YUV

YUV [21] represents the color as three components, luma (brightness, known as Y') and chroma (color, U and V). This stems from the days of analogue TV signal transmission, where engineers needed a way to add color to an already existing black and white

signal. By adding the two chroma channels to the existing b/w signal you get a full color signal. An example of the three different channels and how they look combined is shown in figure 2.6(a).



(a) Left side: Original image (top), Luma (Y′) component (bottom), Right: U (top) and V (bottom) chrominance components



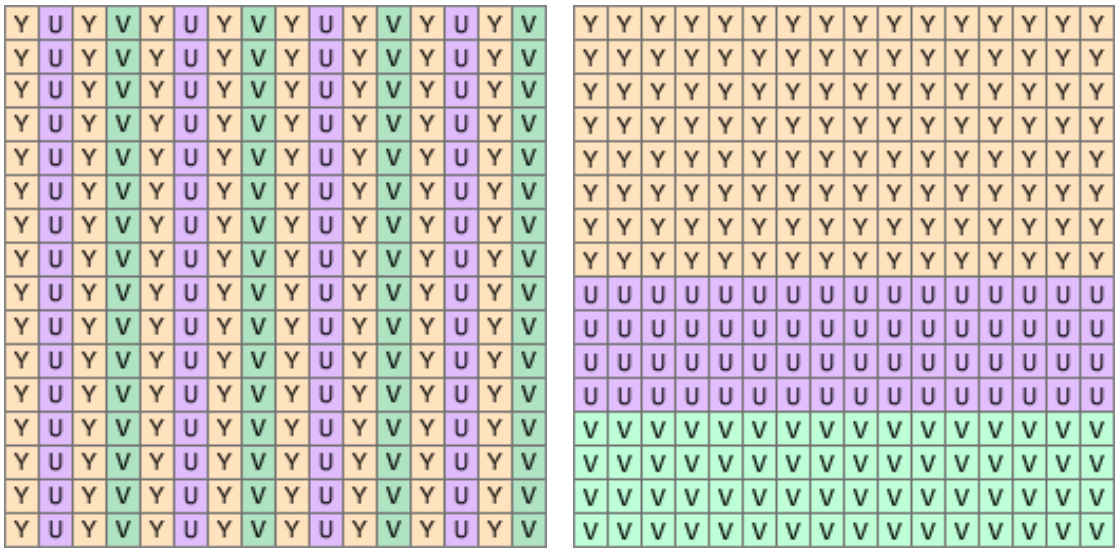
(b) Different subsampling in YUV

Figure 2.6: YUV color model examples [6].

Since humans are more sensitive to black and white information contra color [22], YUV allows for something called subsampling reducing the size needed for a full image. Using subsampling the color channels are only sampled in a given ratio to the luma channel (see figure 2.6(b)), thus reducing the color accuracy of the output, but also lowering the size needed.

Packed and Planar

Both RGB and YUV have two different ways to store individual pixel data, namely packed and planar. Packed saves all components of the given pixel sequentially at the pixel's offset in memory. Planar first saves the entire first component for all pixels, then the entire second component for all pixels, etc. Figure 2.7 has examples of how the pixel data in a YUV4:2:0-image is stored in both packed and planar formats. Here, we can see that in cases where the channels are not of equal size (like with the chroma subsampling of YUV4:2:0) the planar format requires less space, as the non-sampled values are not stored. In the packed format all the values; will always be stored to according to format, and non-sampled values will simply be set to 0.



(a) Packed pixel data (b) Planar pixel data

Figure 2.7: Packed and planar example showing storage of a YUV file with 4:2:0 subsampling

2.4.3 Pipeline Architechture

The pipeline is made up of several sequential modules each one performing a different task. By keeping each step as modular as possible, adding or removing additional steps of functionality becomes very straightforward and simple to implement. Figure 2.5 shows the full prototype pipeline with all its modular steps. We now investigate each of the steps a bit closer, before we dive into the specific implementation details in chapter 3.

Reader and Converter Step

The first step of the pipeline is to get the frame data. In this first prototype, we used pre-recorded footage in the YUV-format dumped directly to hard disk. The reader's job is therefore simply to read these files and pass along the frame data to the next step in the pipeline. After the frame is read into memory, we do a conversion step to convert the YUV data to the RGB color space we use for the rest of the pipeline.

Debarrel Step

As mentioned earlier, we use 3.5mm wide angle lenses on our cameras, which introduce an optical effect called barrel distortion (figure 2.8). Barrel distortion bends straight lines radially outwards from the center in a shape resembling a barrel, thus the name. To be able to do further work on the frame data, this unwanted effect needs to be corrected for, and that is what the debarrel step does. Debarreling counters the barrel distortion by trying to rearrange the pixels to be as rectilinear as possible. For this to work the debarreler must be configured correctly for the lens of the camera in use, and we investigate this closer in section 3.7.3.

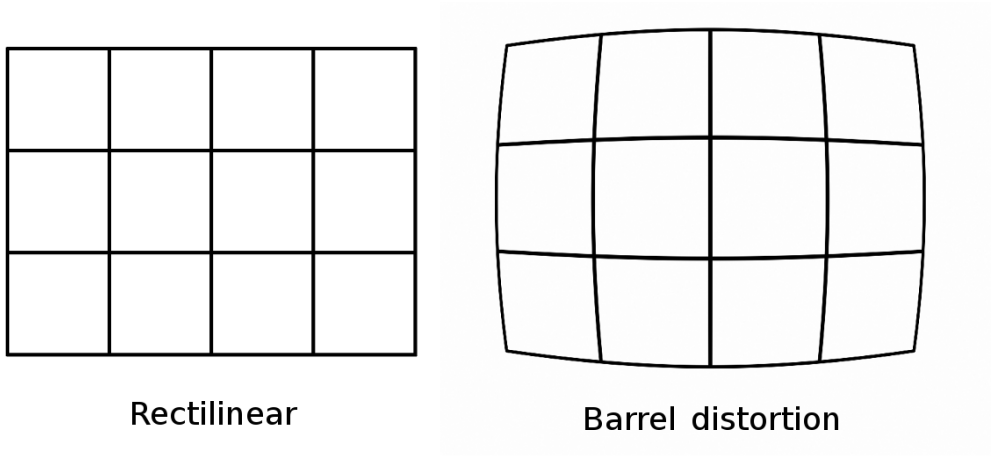


Figure 2.8: Difference between an image that is rectilinear and one that has barrel distortion

Warping Step

Since all our cameras are positioned at different points, each facing the field at a separate angle, we need to do another processing step before we can stitch the frame data together. The individual frames must be manipulated in such a way that they all share the same perspective as one of the other cameras (called head). This operation is called image warping, and is done by moving each pixel in the frame using a pre calculated transformation matrix (detailed in section 3.7.7).

Stitching Step

After the warping step is complete, we can create the full panorama output image by copying the four camera frames into a single output frame. This step is called stitching and is done by first finding the overlapping sections between the camera frames (figure 2.9), then choosing the cut offsets within each of these regions. In this prototype pipeline the cuts are strictly vertical and static, and never changes throughout the execution. When the offsets are found we simply block copy the visible parts of the respective frames into the new panorama frame. Figure 2.10 shows how the final output looks. Since the emphasis of this thesis is the dynamic stitching, we will look a lot closer at the specifics of the stitching in both section 3.7.9 and the whole of chapter 4.

Second Conversion Step

Our data up till now have been in RGB, but the encoder in the following step needs data in YUV4:2:0 to work correctly. This step is therefore just a conversion of the whole RGB panorama frame to a YUV version.

Storage Step

The final step in the pipeline is storing the stitched panorama frames to disk. The choice of what format we use for the storage is an important one for many factors

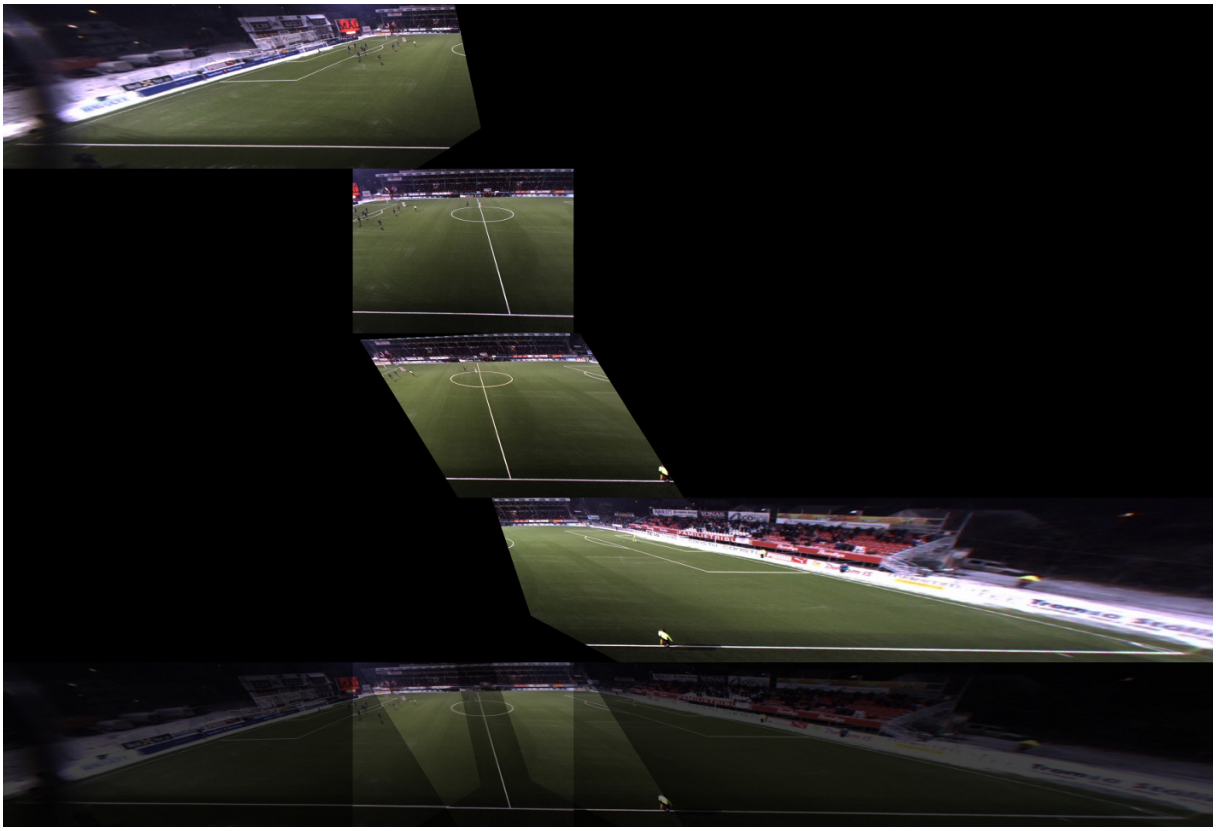


Figure 2.9: Our four warped images at the top, shown overlayed at the bottom.

Figure 2.10: Stitched panorama output.

(including quality, performance and size) and is discussed in [6], we also look at some alternatives in section 3.8. In this prototype pipeline, we encode the frames as lossless H.264 in 90-frame chunks (3 seconds) and write them directly to disk.

Since H.264 has no support for metadata, the initial recording timestamp is appended to filename of the first file written. By knowing both the initial time and frame number we can calculate the exact timestamp of each frame easily. This is needed for synchronization with time based systems (such as ZXY), but is also useful for finding the seek offsets of exact times in the video.

2.4.4 Performance

The performance of this first prototype pipeline stitcher can be found in table 2.1. From reading the numbers, it is immediately clear that this is not running real-time and that the performance in general is pretty slow. Especially the stitching has bad per-

formance, using as much as 88% of the total time. The reason for this is simply that this first prototype version was made without performance in mind, so no sort of optimization was attempted. In the later improved version of the pipeline (chapter 3), we introduce real-time as a fixed constraint, and achieve very different performance numbers.

Step	Mean time (ms)
YUV ⇒ RGB	4.9
Debarreling	17.1
Primary stitching	974.4
RGB ⇒ YUV	28.0
Storage	84.3
Total	1109.0

Table 2.1: Old pipeline performance

2.5 The Bagadus Player

The Bagadus player is the playback part of the Bagadus pipeline. It is built ground up just for our needs, and is written in *C++* using a combination of OpenFrameworks [23], OpenCV and libNorthlight. The main purpose of the Bagadus player to be a nice user interface for displaying the data generated in our pipeline. A video demonstration of the player can be seen in [24]. We will walk through some of the most important features of the program:

2.5.1 GUI

The Bagadus player features a user friendly and easy to use interface. It is built using OpenFrameworks' *ofx_Gui* [25] add-on library, which comes with high level functionality for creating and managing user interfaces quick and easily.

In the center of player application is the main camera view. This part of the interface is used to display the currently selected camera or the whole panorama frame if stitched video mode is selected.

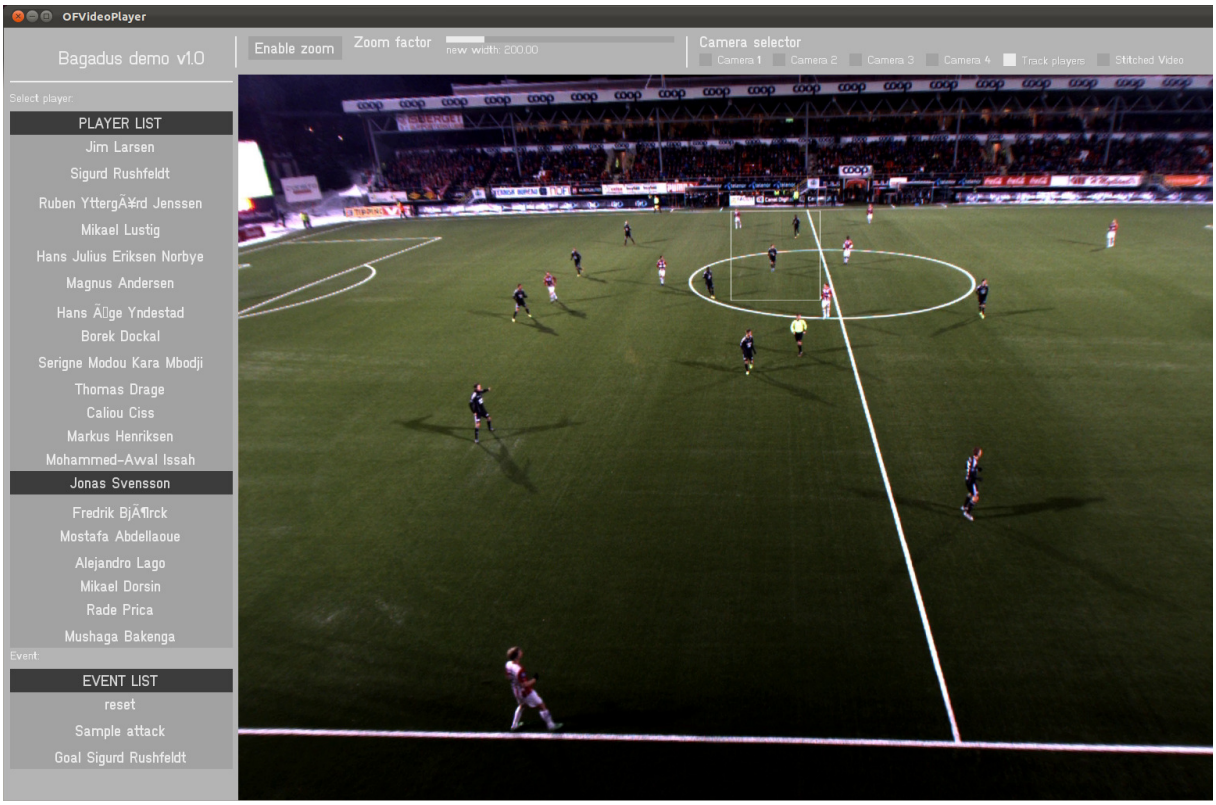
On the left side of the Bagadus player there is a list of all players and a smaller event list directly below it. The player list gets populated by names from the ZXY-database, which are manually entered before each match. Clicking on one or more of these entries turns on tracking for all the selected players. All tracked players get a rectangle overlay on the video showing their exact position in the frame (shown in figure 2.11(a)). The rectangle will only be drawn for players who are present in the main camera view, however selecting the panorama mode will show the entire field, and thus all selected players will always be drawn.

The event list is a proof-of-concept demo using mock Muithu-event-data. Clicking an entry in this list plays back the current event by forwarding the video to the time of the occurrence and enabling tracking on all players associated with said event.

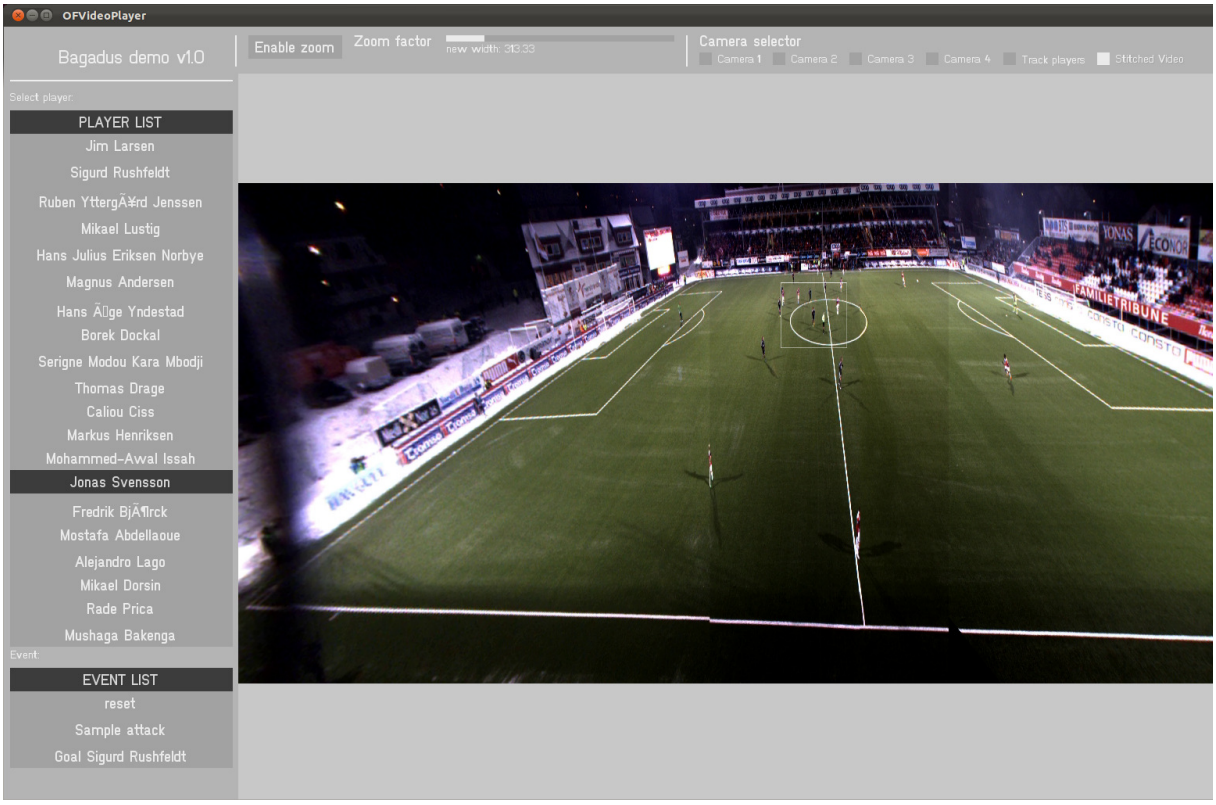
Directly over the main camera view there are several controls, including zoom, camera, tracking and stitched video toggle. Starting with zoom this enables digital

zoom on one or more tracked players by simply enlarging the ZXY-rectangle area of the player or players. The quality on this zoom is somewhat lacking due to the low resolution of our cameras, but in the future this can be greatly improved by using higher resolution camera equipment.

To the right of the zoom is the camera control. This allows us to manually select which one of the four cameras we want to view in the main camera view. If you select "Track players" instead of a camera the player will continuously change the active view to whichever camera has most of the players you are currently tracking. The final mode ("Stitched Video") switches the mode to the full panorama video stitched together from all the cameras (figure 2.11(b)).



(a) Tracking a single player



(b) Same player outlined in the stitched panorama view

Figure 2.11: Bagadus player application screenshots

2.5.2 ZXY Integration

The player has full ZXY integration in order to track player positions on the video streams it displays. It manages this by fetching and parsing the ZXY data directly from our tracking database using SQL. Since this first prototype uses fixed cut stitching, we know how much each of the cameras cover, and by checking which part of the seam a player is on we can calculate the exact camera he appears on. Then, by running the 2d ZXY-coordinates through a ZXY warp lookup function for the camera in question; we get the exact pixel position of this player.

Using this approach, we can trace a player across the whole field by swapping the active camera for the main view whenever the player crosses the seam between two cameras. This functionality can be enabled in the bagadus player by checking the "track players" checkbox at the top of the user interface (see figure 2.11). If more than one player is selected, it will automatically select the camera with most of the selected players present.

2.6 Summary

In this chapter, we have described our first prototype implementation of our panorama video pipeline, nicknamed "Bagadus". We started by looking at the hardware setup needed for such a system, then at some of the pre-existing analytical systems already in use at our location at Alfheim stadium. Then, we walked step-by-step through the modules of our first pipeline prototype, before we finally took a look at the player application built to facilitate the system.

As the benchmarks in section 2.4.4 stated, this version of the prototype is severely lacking in performance, and has much room for improvement. One of our goals is to be able to deliver processed video to the coaches during half-time, so this prototype pipeline is not fast enough. The output panorama video from this pipeline also contains visual artifacts, such as color differences between the cameras and players getting distorted in the static cut of our stitcher. To improve these shortcomings, an improved version of the pipeline was created, and we will continue in chapter 3 detailing this.

Chapter 3

The Improved Pipeline

3.1 Motivation

The first prototype pipeline showed that it is indeed possible to create a panorama stitching pipeline for a scenario such as the one we have at Alfheim. One of the goals of this master project, however, is to do so in real-time, which our first prototype did not. Our work thus started on making a better and improved pipeline with such a goal in mind, and this is what we'll look closer at in this chapter.

3.2 Related Work

Real-time panorama image stitching is becoming more popular, and there exist many proposed systems for it (e.g., [7–11]). We are also seeing support for this feature in modern smart phone operating systems like Apple iOS and Google Android, which supports stitching of panorama images in real-time. The definition of real-time, however, can differ greatly based on context, and these devices usually would classify 1-2 seconds as within their acceptable real-time bounds. In video, real-time has another meaning, as the panorama must be generated no slower than the video frame rate, e.g., every 33 ms for 30 frames-per-second (fps) video in our scenario.

Camargus [4] is one of these existing systems, and the developers claim it can do panorama video from a 16 camera array setup in real-time. Unfortunately Camargus is a commercial system, so we have no further insights to the details. Another example is the Immersive Cockpit [26] system, which generate stitched video from captures with a large field-of-view, but does not focus on visual quality. They are able to generate 25 fps video using 4 cameras, but there are visual limitations to the system making it unsuited for our scenario.

A setup resembling ours is presented in [27], which does stitching on GPU. The performance good enough for real-time, but it is limited to two cameras and only produces low resolution images. On the other side of the spectrum we have Haynes [28] and the Fascinate [29] project, which both produce high resolution videos, but require expensive and specialized hardware.

In summary many related panorama systems exist, but none are able to meet our demand of being a low-cost implementation able to create full panorama video using

four cameras at 30 fps. Thus, we have implemented our own panorama stitching video pipeline which utilizes both CPU and GPU to satisfy our demands. The system is presented and detailed in the following sections.

3.3 Architecture

The architecture of the new pipeline shares the same modular design as the old one, but with several new additions. Most important is that we now do much of our calculations on GPU using nVidias CUDA parallel programming framework. The pipeline is therefore split into a CPU and a GPU part, as shown in the full layout chart (figure 3.1).

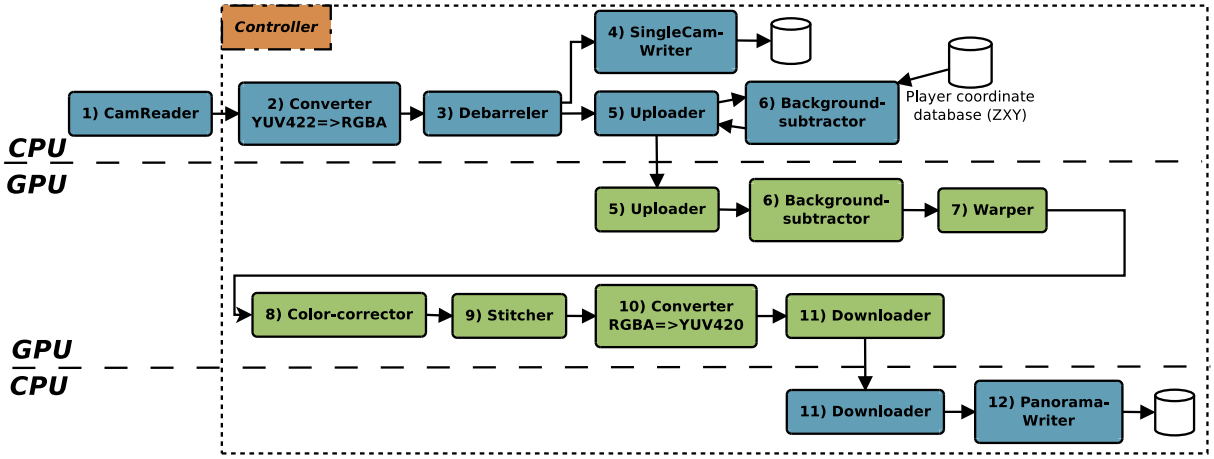


Figure 3.1: Pipeline architechture

Furthermore, several new modules were added and also and a new controller module to synchronize all our steps. Since we have so many modules now, the controller is essential for keeping the flow, and moving data between our modules. An example of how the frames advance through our pipeline is shown in figure 3.2. As we see the frames goes from module to module, and all the modules are executed in parallel every 33 ms. It is the controller's job to move the data between the modules, and it does this either by coping the data between the module buffers, or change the write pointers used in the modules to point to another buffer. In this new pipeline, we also use the ZXY positional data directly in some of the computations, so a step for fetching those is also present.

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 3.2: Illustration of how frames flow through our pipeline.

As we did in chapter 2, we will now step through all the components and modules of this new pipeline, and look at how they work together. In this section, we will also look a lot more detailed at the implementation details regarding each component.

3.3.1 Nvidia CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform created by Nvidia. CUDA allows you to write code that gets executed on the massively parallel graphical processing units (GPU), allowing for extreme performance boost for suitable applications. This generally means tasks that can be heavy parallelized, as they gain most from this kind of architecture. Unfortunately CUDA implementations are not trivial to do, and certain problems are a lot better suited to be solved on regular processor than on a GPU.

In our case, CUDA is used for a whole section of the pipeline (as shown in section 3.3). As moving data to and from the GPU is a big performance hit, the pipeline is designed so that once the data is uploaded on our card it stays there as long as possible. This also means that we might use a GPU-solution on some cases where a CPU version could be more efficient, as moving the data to the processor and back would introduce too much overhead. By keeping everything on the GPU for as long as possible we are able to tap the great potential of CUDA while keeping the overhead of data transfer to a minimum. CUDA is well documented [30–32], so no further technical coverage about it will be provided here.

3.3.2 Improved Setup

As mentioned in section 2.2.2, the prototype pipeline was set up with two machines capturing two cameras each. This lead to some problems with synchronization between the machines, as the clocks on the two machines would drift slightly. After looking a bit at the performance number it became clear that it was possible to run all four cameras from the same machine, thus avoiding the problem with synchronization all together. This is now implemented in the new CamReader module (section 3.7.1).

In this new setup we have also made the ZXY database server synchronize its time with the same NTP time server as we use on the pipeline machine. This solves an

earlier problem where the clocks on the individual machines would drift slightly over time, messing up the ZXY-data synchronization with our frames.

3.3.3 Initialization

When the pipeline starts it first parse the input parameters. The current supported arguments are a recording start timestamp and length of the recording. If the set time has not occurred yet the process will wait until it occurs before it starts up the pipeline. The reason for this waiting is to be able to schedule multiple pipeline runs without them interfering with each other. Since we only can have one pipeline process initialized at the time, we have scheduled processes wait before any initialization. When the pipeline is ready it will initialize CUDA and set the active CUDA device, which will be the GPU with the highest number of CUDA-cores. It then creates a new PanoramaPipeline object, and launches it.

Since there is a variable amount of delay from the start of the pipeline to the recording begins, we actually begin initialization 30 seconds before the specified record start timestamp. 30 seconds are more than enough, but it's better to get a few frames extra, than to miss the kickoff.

3.3.4 Controller

The pipeline now has a central controller to coordinate and synchronize all our modules. The modules themselves do not communicate between themselves at any point, so all interaction between them is done through the controller. The controller does this using signaling, mutexes, barrers, flags, etc.
The pseudocode of the controller looks like this:

1. Create and initialize all modules. The initalization in each module's constructor will be run.
2. While the pipeline is running:
   (a) Wait until the CamReader has new frames.
   (b) Get the frames.
   (c) For all modules, move the data from the output buffer of the module into the input buffer of the next. This is done either by swapping pointers or using memcpy. Pointer swapping only occurs in steps where the input and output buffers are of the same type and size.
   (d) Check for (and handle) any framedrops.
   (e) Broadcast signal to all modules (except the reader) telling them to process the new data in their input buffers.
   (f) Wait for all modules to finish using a barrier.
3. Cleanup and terminate.

It should be noted that all this logic leads to a certain amount of time overhead in the pipeline from the controller. Thus when looking at the module timings the controller overhead must also be added to the total time. The CamReader is the only module where this can be ignored, as it works largely independent of the controller.

3.4 Module Design

Our modules all follow the same general design. Each module has a module controller which is responsible for all communication with the pipeline controller thread. The pseudocode for the module controller looks like this:

1. While the pipeline runs:
   (a) Wait for the signal from the main controller.
   (b) Increase internal frame counter.
   (c) Execute module's main processing function.
   (d) Wait for all module threads to finish (using a barrier).

The execution in step 1c differs a bit based on the type of module. On single threaded CPU-modules the execution step actually runs the processing code itself. On multithreaded CPU-modules, however, it will signal its subthreads to do the actual processing. Lastly for GPU-modules the controller just launches the module's CUDA kernels, which does the processing directly on the GPU.

3.4.1 Buffers

All our modules in general have two set of buffers, input and output. Exceptions are of course the start and ends of the pipeline, as the reader gets data directly from the cameras and the writers output to disk. A full list over our buffers is found in table 3.1.

The CPU models have their buffers located in ordinary system memory (RAM) while the GPU modules have them in shared memory on the card itself. The Uploader and Downloader modules moves data between CPU and GPU and must therefore must have buffers on both sides. The Uploader has an extra set of GPU buffers as it uses double buffering when transferring data.

This input/output buffer model was designed to make modification and addition of new modules as easy as possible. As long as the module reads the format of the previous' output buffer, and itself outputs in the format of the next module's input buffer, adding it to the pipeline is trivial. It is also very straight forward to re-route the data between different modules, or skip some of them entirely, with just a few code changes in the pipeline controller.

It should be noted that in our implementation all data movement between modules that use buffers of the same type and size are done using pointers instead of actually copying the data. This is done by setting the input buffer pointer of the next module to the allocated output buffer of the previous one, and then moving the output pointer

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 3.1: Pipeline module buffers

of the previous module to the allocated input buffer of the next module. These pointers will be swapped every pipeline cycle so that the reading and the writing always happen on different buffers.

3.5 Frame Delay Buffer

Since there is a small delay from the capture to the ZXY positional data is available from the database, we need a way to stall the processing of frames until this data is ready. This delay is approximately 3 seconds, and there is also the query execution time (around 600-700 ms) that must be factored in. Since the first module that needs ZXY data (Background Subtraction) much faster than this delay we use a delay buffer to hold the frames until we can process them. The buffer is located between the Debarrel and the Uploader module as we wanted it as close to the first module that needs ZXY while still being on the CPU for easier implementation. The size of the buffer is 150 frames for each camera, 600 frames total. At 30 frames per second, this is a delay of 5 seconds. The size can be manually configured if a longer or shorter delay interval should be needed.

3.6 Frame Drops

Our improved pipeline also introduces handling of frame drops, both from cameras themselves but also internally in the pipeline itself. A certain, albeit low, amount of frame drops will always occur while running the system, but it is essential to keep it as low as possible so it does not affect our output.

3.6.1 Camera Frame Drops

Camera drops happens when the Basler API code in our CamReader fails to return a frame. This can happen from time to time due to several factors like unexpected errors with the cameras, timing errors in the triggerbox or high CPU load. We handle missing frames from the camera by simply re-using the previous read frame. This is generally not noticeable as long as the occurrence of camera drops are very rare, which they usually are under normal runs of the pipeline.

3.6.2 Pipeline Frame Drops

Since our pipeline is on an extremely strict real-time constraint, we must handle frame runs going over this threshold by dropping the frame in question. Modules taking too long can happen for a lot of reasons, but most common are overloaded CPU, OS interrupts or file access and IO taking too long. The CamReader module reads in new frames by overwriting the old buffers, so if a run takes too long we risk that the frames from the camera gets overwritten by the next set before they get processed.

We solve this by having a frame counter for each camera in the CamReader, which gets incremented whenever it reads new frames. The pipeline Controller then checks

this counter every time it moves the frame data from the reader. If it's the expected next in sequence everything is processed normally. But if there is a gap, we know we have missed one or more frames and these will be flagged as dropped. The controller does this by pushing the frame numbers of the missing frames onto the drop counter list in all modules of the pipeline. On each iteration in the pipeline all the modules checks whether the received frame is in this list, and if so they do *not* process it. The writer modules are a bit different and handle these dropped frames by writing the previous written frame again. This keeps the frame count consistent and avoids skipping in the output videos, but can be very noticeable if heavy frame loss should occur.

3.7 Pipeline Modules

We now look at all the modules of the pipeline in detail. We walk through them following the general layout used in our pipeline chart (figure 3.1).

Figure 3.3: The CamReader module

3.7.1 CamReader

The reader is the first module of the pipeline, and it reads frames into the system. The current pipeline reader reads data directly from our Basler cameras, which are set up to deliver YUV4:2:2 frames, and sends these along the pipeline. This first reader-step of our system is designed to be very modular, and in our prototype pipeline and earlier tests a file reader was used to read pre-stored frames from disk instead of streaming directly from camera. Other cameras, or video systems, can be easily supported in the future by simply writing a custom reader for it.

It should be noted that the frame source of the reader (in this case the cameras) is what dictates the real-time threshold of our entire pipeline. Our cameras delivers frames at a steady 30 fps (frames per second), which means that each module must be under 1/30th second (33 ms) to keep everything real-time.

Implementation

The CamReader is a pure CPU-module and runs using 1 thread per camera (for a total of 4 in our case). Each thread interfaces its associated camera using the Basler API wrappers in libNorthlight and return frames in the YUV4:2:2 format. The cameras support a theoretical maximum resolution of 1294 x 964, as seen in A.3.1. But the driver actually limits this to 1280 x 960 in our scenario, so that is the resolution we are using.

The pseudocode for the CamReader looks like this:

1. While the CamReader threads are receiving frames and the pipeline runs:
   (a) Try to get next frame using a timeout 34 ms.
   (b) On timeout: Clone previous frame, but use current timestamp.
   (c) On success: Save the frame with the current timestamp in the module output buffer.
   (d) Wait for the other reader threads to finish before continuing.

Since the camera shutters are synchronized directly by the external trigger box mentioned earlier, we are sure that we get four frames taken simultaneously at an exact 30 Hz interval.

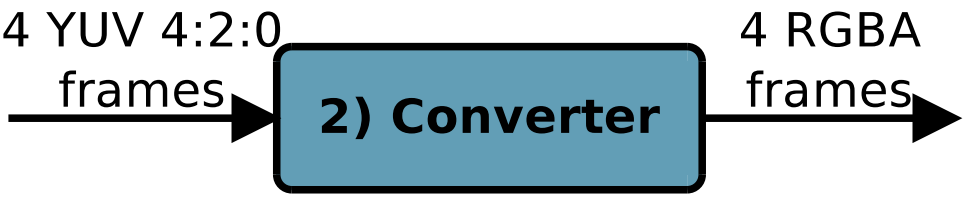


Figure 3.4: The Converter module

3.7.2 Converter

Since our pipeline use the RGBA color space for most of its internal operations we must convert our incoming YUV4:2:2 frames before they can be processed further. While the pipeline in theory could be YUV all the way through, we went with RGBA internally due to it being simpler to understand and work with. RGBA was chosen over regular RGB as some of the modules, especially the background subtractor, works more efficient using that format (detailed in [33]).

Implementation

The Converter is a single-threaded CPU module that takes four YUV4:2:2 frames from the previous reader step and converts them into four RGBA frames. The conversion itself is done using libNorthlight's *VideoConverter* class, which again is based on *swscale* and has conversion support for several formats including YUV4:2:2 and RGBA. Unfortunately there is no direct converter from 4:2:2 to RGBA, so in order to get the desired result we have to go from 4:2:2 to 4:2:0 to RGBA. A more direct conversion would probably be faster, but since this module already has real-time performance, no extra time was spent on trying to speed it up. This module also runs single threaded for the very same reason, it is fast enough and adding more complexity would be a

waste of time unless absolutely needed.

The pseudocode of the Converter module looks like this:

1. For all cameras:
   (a) Convert input frame from YUV4:2:2 to YUV4:2:0.
   (b) Convert this YUV4:2:0 frame to RGBA.

Figure 3.5: The Debarrel module

3.7.3 Debarreler

Since our cameras utilize 3.5mm wide-angle lenses to capture video, the captured frames will always show some sign of barrel-distortion. The Debarreler's job is to rectify this as much as possible and make the lines in our frames as straight as possible. For the Debarreler to work correctly it must first be calibrated properly, so we discuss that step before we look at the implementation.

Calibration

Before we can run the debarreling step on any frame data, we need to find the debarrel coefficients needed to normalize the output as much as possible. The amount of barrel distortion differs in all lenses, so in order to get the best frame data possible all cameras needs to be calibrated separately. Calibration is done by taking a series of pictures of an easy to detect chessboard pattern in different positions and angles, and then running these through OpenCV's autocalibrator. Given that the chessboard is clearly visible in all shots, and that the different angles and positions together cover most of the lens' field of view, very good results can be achieved. As mentioned earlier, the calibrated debarrel coefficients are unique to optics in the lens used in the calibration process, so if the lens is changed at any point the whole process needs to be done again. Figure 3.6 shows an example of the calibration. The calibration step is also discussed with greater detail in [6].

Implementation

The Debarreler is a multithreaded CPU module that takes four RGBA frames in and returns four debarreled RGBA frames as output. The debarreling itself is done using

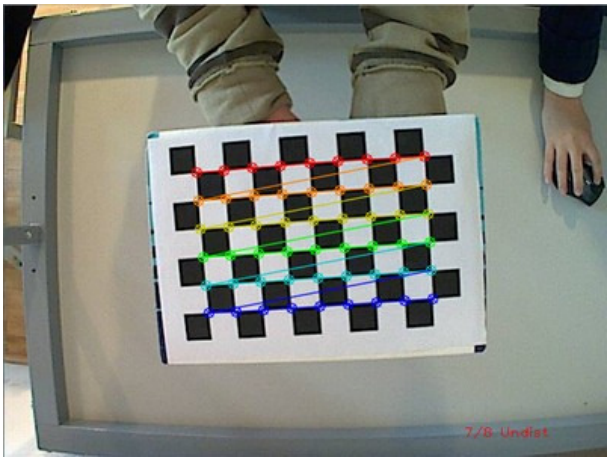


Figure 3.6: Debarrel calibration in OpenCV. The dots and lines are placed by OpenCV to show that it recognize the chessboard.

OpenCV, which remaps the pixels of the frame using the debarrel coefficients we found in the calibration step. Since running four of these operations sequentially used longer time than our real-time threshold allowed for, it was split up in four separate threads running in parallel.

The pseudocode for the debarreler:

1. For each debarreler thread (one for each camera):
   (a) Run OpenCV's debarrel-function using the correct debarrel coefficients on the assigned frame.

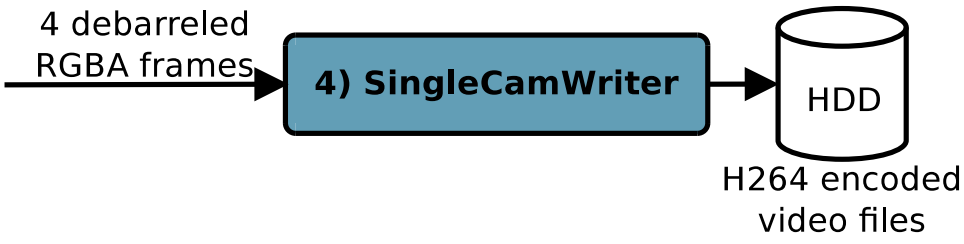


Figure 3.7: The SingleCamWriter module

3.7.4 SingleCamWriter

Our pipeline does not only provide a single panorama, but also the four individual camera streams unstitched. The singlecamwriter is in charge of dumping these individual camera streams to disk. This step is done after the debarreler as the files are more useful without the distortion.

Implementation

SingleCamWriter is a multithreaded CPU module that takes in four debarreled RGBA frames and writes them to disk. For performance we run each camera writer in its own thread, with all four running in parallel. The frames are encoded as 3 second long H.264 segments and then written to disk. The H.264 encoding is a bigger topic in itself, and we will look a bit closer at it in section 3.8.3. The files are stored in a folder structure where each camera has its own folder, and the file names consist of the file number and a timestamp.

Pseudocode for the module looks like this:

1. For each SingleCamWriter thread (one for each camera):
   (a) Convert input frame from RGBA to YUV4:2:0.
   (b) If current open file has 3 seconds of data: Close it and open a new one with updated timestamp and file number.
   (c) Encode the converted input frame to H.264.
   (d) Write the H.264 frame to our current open file.

The conversion is done using libNorthlight (as detailed in section 3.7.2). We do conversion, encoding and writing to disk in this single module simply because works under the real-time constraint. The encoder is the most demanding operation in the module, so if given a bigger load to encode (i.e., frames of bigger dimensions), splitting all the operations to separate modules would most likely be required.

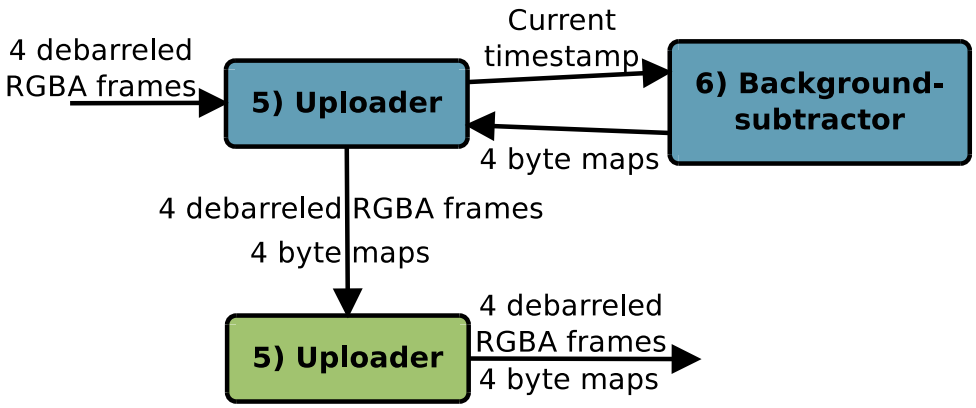


Figure 3.8: The Uploader module

3.7.5 Uploader

We now move most of our processing over to the GPU, and the Uploader is the module in charge of moving all relevant data over to our graphics card. The Uploader also does

some CPU-processing for the BackgroundSubtractor GPU module that gets uploaded to the GPU among the other data.

Implementation

The Uploader is a single threaded module that runs on both CPU and GPU. It takes four RGBA frames as input and returns 4 RGBA and 4 byte maps on the GPU (described in section 3.7.6) as output. The data is uploaded to pinned GPU-memory using the asynchronous *cudaMemcpyAsync()* using double buffering (further detailed in [33]).

The Uploader also does some CPU-work for the later BackgroundSubtractor module (we look closer at what in the related module section). The byte maps from this work are uploaded exactly the same way as the RGBA frames.

The pseudocode for the module looks like this:

1. If the BackgroundSubtractor module exists: calculate player pixel byte maps
2. For each camera:
   (a) Asynchronous transfer the input framedata from CPU to GPU-memory
   (b) If the BackgroundSubtraction module maps were calculated: Transfer them asynchronously to GPU-memory.

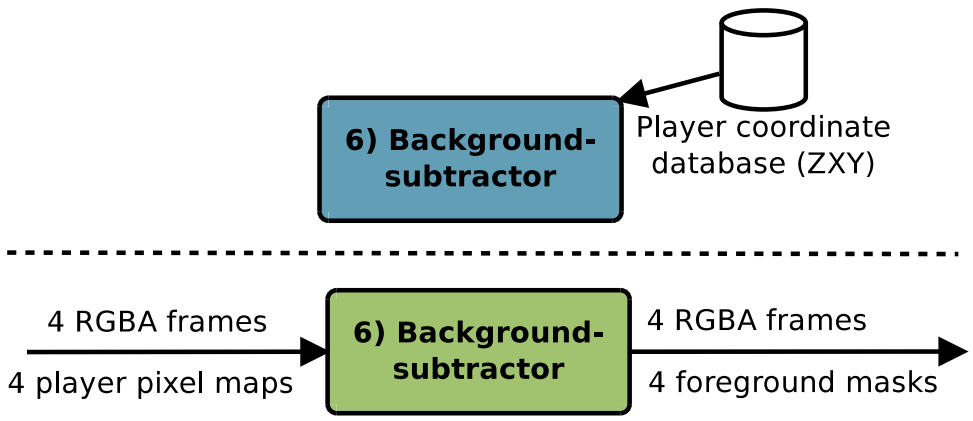


Figure 3.9: The Background Subtraction module

3.7.6 BackgroundSubtractor

The BackgroundSubtractor (BGS) is a brand new module in this new pipeline. It is used to analyze a video stream and finding out which pixels are foreground and which are background. It does this based on special algorithms developed for this purpose and we can use the result to improve later modules in the pipeline (e.g. the accuracy of the stitcher module). The BackgroundSubtractor uses ZXY data to improve its accuracy,

which explains why it needs both a CPU and GPU part. Background subtraction is a large topic, so for further details about how it is used in our project please consult [33].

Implementation

There are two parts of the BackgroundSubtractor module, one on CPU and one on GPU. The CPU part is responsible for calculating the player pixel lookup maps based on ZXY data. These maps are simple bitmaps (of the same size as our frames) specifying whether or not players are present on each pixel. For getting the ZXY data we have a dedicated thread that checks the database for ZXY samples when needed. Note that the CPU part of the BackgroundSubtractor is executed through the Uploader module for convenience; it could very well be split into its own custom module if needed.

The GPU part of the BGS runs the actual background subtraction on the frames. It takes four RGBA frames and corresponding player pixel lookup maps and returns the four unmodified RGBA frames and corresponding foreground masks as output. The foreground masks are pixel maps with separate values for foreground, foreground shadows and background, and this is what is used in later modules. Note that the ZXY-based pixel lookup maps provided from the CPU module only is a kind of optimization of the actual subtraction allowing it to only run on areas of the frame where there are players present. It is not necessary for operation, so we have a fallback mode in the pipeline for instances where ZXY is not available.

The pseudocode for the BGS ZXY retrieval thread (*CPU*-side) looks like this:

1. While the pipeline is running:
    (a) If the cached ZXY data table in memory is below a certain threshold, get more and update the table.

The creation of the pixel lookup map (*CPU*-side) looks like this:

1. Create new bytemap with the dimensions of our frames (1280 x 960).
2. Get the cached ZXY-data for the sample that matches the current frames' timestamp.
3. For all players:
    (a) Translate ZXY coordinate into pixel data.
    (b) Set the translated pixel positions in the pixel lookup map to 1.
4. Return the byte map.

The execution of the BGS on the *GPU*-side looks like this:

1. For all cameras:
    (a) For every pixel: calculate the pixel status to either background, foreground or shadow.

Result

The result of the BackgroundSubtractor module is a foreground mask that can be seen in figure 3.10. We use these masks later in our pipeline to determine which pixels in our frames contain players and which do not.

(a) BGS input frame (b) Resulting foreground mask

Figure 3.10: Result of background subtraction

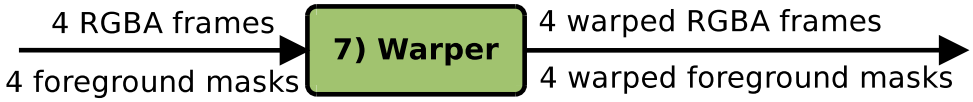


Figure 3.11: The Warper module

3.7.7 Warper

The warper module is used to prepare the debarreled frames for the next step in the pipeline, the stitcher. Since our cameras are covering different areas of the field with varying angles, getting our source frames as aligned as possible is essential for the stitch to be good. The warper works by using one of our four frames as a base reference and then "warping" all the others to fit the original perspective as much as possible. Ideally, if you were to overlay the warped images over the base it should look like one continuous image. The warper works by simply applying a pre-calculated transformation matrix to each pixel in the frame.

Since we have corresponding background masks from the *BGS* for each frame we also warp these to match our warped frames. This is required since we want to be able to use the background data later in our pipeline, and the foreground masks should always match the frame data.

Implementation

The warper is pure GPU module. It takes 4 RGBA frames and 4 foreground masks as input, and returns 4 warped RGBA frames and 4 warped foreground masks as output. Our implementation of the warper is based on the call *nppiWarpPerspective_8u_C4R* from NVIDIA's NPP library [34]. This is the GPU equivalent to the OpenCV call *cvWarpPerspective* that we used on our first CPU prototype. The warp call needs a warp matrix for each of the cameras to be able to warp the perspective correctly and these are found using several OpenCV techniques (detailed in [6]). We must also set the interpolation method used in the warping, and for performance reasons we are using nearest neighbor, which runs very fast and looks good enough for our use.

The pseudocode of the Warper looks like this:

1. For all cameras:
   (a) Warp the input frame using *nppiWarpPerspective_8u_C4R()*.
   (b) Warp the foreground mask using the same NPP-call.

Result

The result of the Warper can be seen in figure 3.12. This shows a real frame of our pipeline before and after the warper has processed it.

(a) Image *BEFORE* warping

(b) Image *AFTER* warping

Figure 3.12: Input and output of warp step

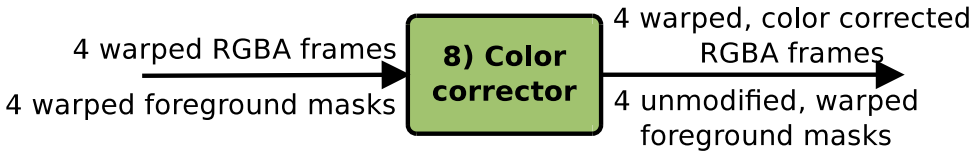


Figure 3.13: The Color Correction module

3.7.8 ColorCorrector

Since we have four different cameras, all with different perspectives and thus slightly different lightning conditions, we need a module to even out the differences as much as possible before stitching. The big color differences between the stitched frames was one the reasons why our fixed stitch in the prototype pipeline was so visible, so normalizing the differences in the stitch area is an important step for a good seam. Color correction is a large topic, so for further details about how it works and is used in our project please consult [35].

Implementation

The ColorCorrector (CC) is a GPU-module that takes 4 warped RGBA frames and 4 warped foreground masks as input, and returns 4 color corrected RGBA frames and the unmodified foreground masks. The foreground masks are simply passed through, as the CC has no use for them. The color correction itself works by first finding the overlapping areas of the frames and then using the colors in those overlapping regions to generate correction cooeffecients that are then applied to all pixels of the frame.

The pseudocode of the ColorCorrector looks like this:

1. One camera is used as base and therefore has its correction coefficients set to 1.
2. For the remaining cameras:
   (a) Calculate the current camera's correction coefficients based on the color differences in the overlapping area between the base camera and the current camera.
3. Calculate a global correction coefficent based on all the cameras to normalize the output.
4. Calculate the final coefficients by multiplying the individual coefficients with the global one.
5. Run the actual color correction on all frames, using the final coefficents.

Results

Figure 3.14 shows an example of the ColorCorrector module. Here we can observer that the image without color correction has much more visible seams than the corrected one.

(a) Stiched image *without* color correction

(b) Stiched image *with* color correction

Figure 3.14: Result of color correction. Image is cropped for illustration purposes, and use the original fixed cut stitcher.

Figure 3.15: The Stitcher module

3.7.9 Stitcher

The stitcher module takes the processed frames from the warper and merges them together to the final panorama image. The stitcher assumes that all received input frames are properly aligned, and so its only job is to select which pixels from which frame goes into the panorama output. In this section we look at the original fixed cut pipeline stitcher, although a much better dynamic stitcher is already implemented. This new stitcher and how it works will be covered much more in-depth in chapter 4.

Implementation

The prototype pipeline version of the stitcher is a very simple and straight forward GPU module. It takes 4 RGBA frames and 4 warped foreground mask and return 1 stitched RGBA panorama frame. The foreground masks are not used in this version of the stitcher. The seams of this stitcher are based on fixed cut offsets, which were found manually by visual inspection. By overlaying the frames and finding the overlapping sections, a straight vertical line was chosen as the cut line. All pixels on the left side are taken from the left picture, all pixels on the right from the right frame. The actual copying of the pixels is done using *cudaMemcpy2D* between the input and the output buffers. Since the cuts are fixed and rectangular, we can simply copy the whole chunk of data from each frame using only a single memcopy operation.

The pseudocode of the Stitcher looks like this:

1. For each camera:

    (a) Copy the chunk of data between the previous cut offset and the next cut offset from the current frame to the output panorama frame. The leftmost camera use 0 as the previous cut offset, and the rightmost use the width of the panorama as next cut offset.

Result

Figure 3.16 shows an example of the Stitcher module. Here we can observe our four input frames stitched together to our final panorama output.

Figure 3.16: Result of stitcher, here shown slightly trimmed and without color correction to outline the seams.

Figure 3.17: The YUV Converter module

3.7.10 YUVConverter

Before the panorama frame can be encoded and written to disk it needs to be in the correct format. In the previous SingleCamWriter this was done directly in the module as the performance allowed for it. For the panorama however the conversion process is slower, and thus must be singled out in its own module so that its writer does not cross the real time threshold. The CPU version of this module always turned out too slow, so we moved it over to GPU to get the performance needed.

Implementation

The YUVConverter is a GPU module taking a single RGBA panorama frame as input and returning a YUV4:2:0 panorama frame. The input is first converted from RGBA to YUV4:4:4 using the NPP library. This is done since NPP contains no functions to convert directly to YUV4:2:0, so we first convert to YUV4:4:4 and then do a manual conversion to YUV4:2:0 with our own code. Note that this could be done directly using one step using custom code, but since the current solution works and has good enough performance we did not spend more time on improving it.

The pseudocode of the YUVConverter looks like this:

1. Convert the input frame from RGBA to YUV4:4:4 using *nppiRGBToYCbCr_8u_AC4P3R()*.
2. Copy the returned Y' channel into the correct position of the output buffer.
3. For all returned U and V channel samples:

(a) If to be sampled into the output YUV4:2:0 frame: Copy to correct position in output buffer.

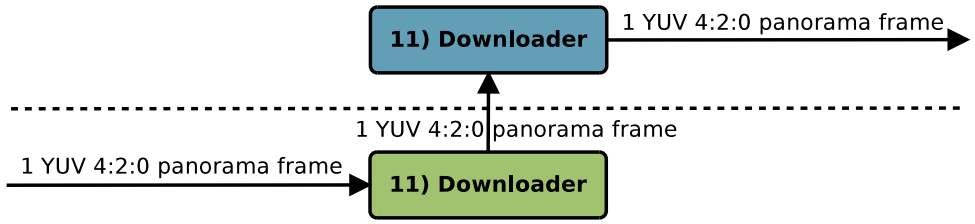


Figure 3.18: The Downloader module

3.7.11 Downloader

Since we are finished processing the panorama frame on the GPU side we use the Downloader to bring the data back to the CPU side. The Downloader is very similar to the Uploader we looked at earlier, but has no other tasks except for the actual downloading part.

Implementation

The downloader runs as a single threaded CPU module. It takes 1 YUV4:2:0 panorama frame as input on the GPU side and returns 1 YUV4:2:0 panorama frame on the CPU side. As mentioned this is a lot simpler than our previous uploader as it has only 1 frame to download, and doesn't do any extra processing tasks. Unlike the uploader the actual transfer is done synchronously with no double buffering as the performance is good enough without.

The pseudocode of the Downloader looks like this:

1. Copy the panorama frame from GPU to host (CPU) using *cudaMemcpy()*.

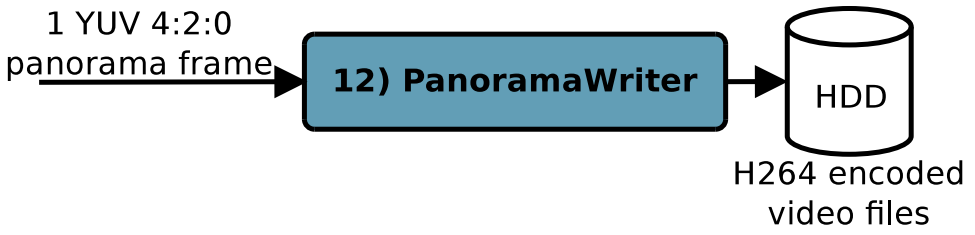


Figure 3.19: The Panorama Writer module

3.7.12 PanoramaWriter

Once the panorama frame is downloaded back to CPU we need to write it back to disk, and this is the job of the PanoramaWriter. The PanoramaWriter shares much of its functionality with the SingleCamWriter (section 3.7.4) but works with a single panorama image instead of the four separate camera streams. There is also no extra conversion done in this module, as the received frame already is in the correct format.

Implementation

PanoramaWriter is a single threaded CPU module, and takes in one YUV4:2:0 panorama frame and writes it to disk. As with the SingleCamWriter it encodes and writes the panoramas in 3 second H.264 files, where there filename consist of a file number and a time stamp. As for file structure the final encoded frames are written to a different folder than the single camera videos created by the SingleCamWriter.

The pseudocode for the module goes as follows:

1. If current open file has 3 seconds of data: Close it and open a new one with updated timestamp and file number.
2. Encode the input panorama frame to H.264.
3. Write the H.264 frame to our current open file.

3.8 Storage

Storage is a very important part of a content pipeline such as this. Choice of output format can have drastic impact on important factors such as size and processing time. Several different formats were considered for our pipeline, and we will discuss some of them briefly here.

3.8.1 Raw YUV

Raw YUV is the simplest way of storing the video data possible. It works by appending several individual YUV-frames into a big one and then write that to disk. While this approach is fast and easy it does not come without drawbacks. Firstly the format has no header or metadata, so to read the files you must know the exact size and format of the single frames in advance. Also while there are some basic YUV4:2:0 support in players such as VLC [36] the overall support for the format is lacking, which means that end users probably need a special built client to view the video.

Size wise raw YUV is also pretty lacking. While YUV in general works by using lossy subsampling of the chroma channels, the compression ratio of this technique is pretty low, leaving a pretty large final size. Since there are no complex compression algorithms in use actually reading and writing these files are very fast, with the only CPU hit being IO-overhead.

It should be noted that the original Bagadus prototype pipeline used raw YUV as storage. Since it used its own player for playback the issue of format support was

irrelevant, and since it was done off-line using large disks the size or performance did not matter much either.

3.8.2 XNJ

XNJ is a custom file format we developed internally to store frame data. As we needed to store frames lossless with a custom metadata it was designed from scratch with such goals in mind. XNJ is a container format which supports storage of any kind of payload data, to which it will apply optional lossless compression (LZW). The metadata part of the file consists of a variable length XML-document that can contain any amount of user set metadata. Both these parts gets combined then prefixed with a special XNJ-header containing useful parsing information like the sizes and bit-offsets for each part.

A full working implementation of a XNJ writer and reader were made, and several tests using payload frame data in YUV4:2:0 were tried. We found that real-time read-/write performance using a moderate level of lossless compression is possible. This includes parsing of the metadata from XML to usable variables in our code. In the end however, we did not end up using this solution, as we found other ways to store the needed metadata, and the overhead of XNJ therefore made it unsuited for our use.

3.8.3 H.264

H.264, also known as MPEG-4 part 10 or AVC, is an advanced motion-compensation-based codec standard for video compression. It offers a very good size to visual quality ratio, and is commonly used in many popular applications (e.g. Blu-ray films, video steaming services like YouTube and Vimeo). Since we wanted the output from the improved pipeline to be readable from a wide range of devices H.264 quickly became a viable alternative, especially since there exists hardware acceleration support for the format in many modern smart phones and tablets. After some testing we found that it worked very well for our project, and implemented it as the standard storage option for the improved pipeline.

Encoder

Encoding in done as part of the two writer modules in our pipeline. The encoder's job is to take the stitched frame data downloaded from the GPU and encode it into H264-frames that then can be written to disk. Our encoder is based around the H264 wrapper code in libNorthlight, which again uses the popular library x264 [19] for the actual encoding. The reason for using H.264 is mainly the good compression rate possible, making the output much smaller than an equivalent encoded in raw YUV4:2:0.

Realtime

In our initial setup of the pipeline the encoder was one of the parts that did not achieve real-time performance during runtime. Some effort went into trying to speed it up by changing the H.264 profiles used in the encoder, but our preset values were already

	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: H.264 performance. Encoding 1000 frames on an Intel Core i7 (table A.2) using x264, includes writing to disk. AUTO is based on the amount of cores, in our case it resulted in 6 threads.

optimized for speed, and there were no real gain. Looking into other ways to speed it up we discovered that the northlight code interfacing x264 forced it to run single threaded, which is hardly ideal for performance.

x264 has two threading models that can be used, *slice* or *frame based*. *Frame based* is, as the name implies, based on frames, and here each thread will handle their own frame. This is good for scenarios where you have access to multiple frames at once, but hardly ideal for our sequential pipeline where each frame is processed from start to finish before a new one is fetched. Some very mild speed improvements were seen using this model though, but we suspect that is simply because each frame getting their own thread simply speed up the loading and disposal/cleanup of said frame, not anything related to the actual encoding.

The other model is *slice based* threading, and based around "slicing" frames up into several chunks and let each thread work in parallel on their own slice. This is much more suited to our pipeline, as we only work on a single frame, and parallelizing the work should ideally provide quite a bit of speedup. While there are some penalties associated with using slices for encoding [37], none of them had much effect in our situation, and we got a nice speedup from it (see table 3.2).

3.9 Pipeline Performance

As we saw in section 2.4.4 the old pipeline was nowhere near real-time, as it was not one of the goals of that implementation. However with this new pipeline we introduce real-time as a new constraint and performance is therefore an essential attribute of the whole system. The timings of all our modules running on an Intel Core i7 (table A.3 in appendix A) can be seen in figure 3.20 and in table B.1. Here we can see that all modules individually perform well under the real-time threshold, even including the slight controller overhead. As we saw in figure 3.2, the modules are all running parallel, so as long as all the modules are under the threshold the whole pipeline will also be. To show that the complete pipeline is running real-time we use another metric.

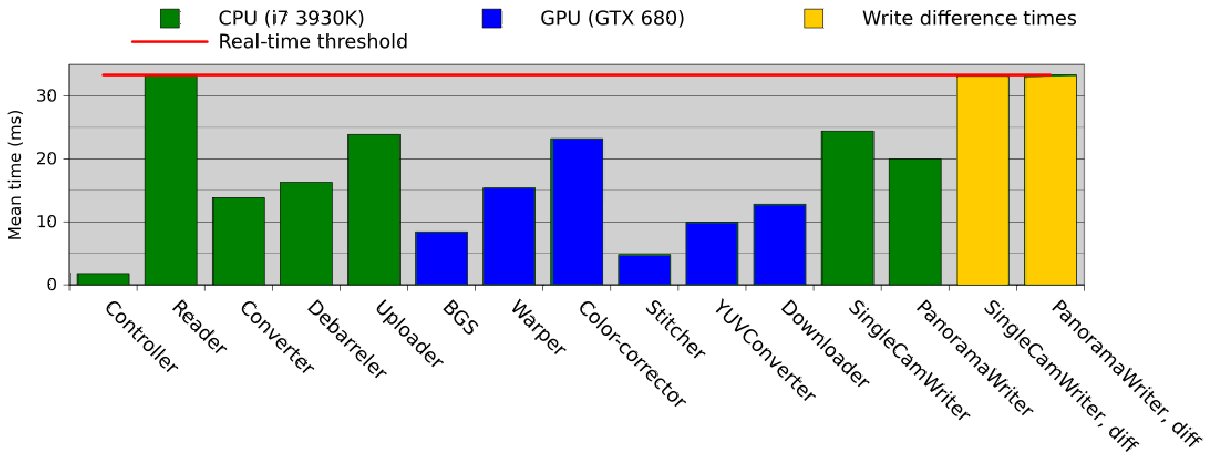


Figure 3.20: Pipeline performance on an Intel Core i7 (table A.3).

3.9.1 Write Difference Times

To show that the pipeline indeed run in real-time we use the write difference times to show that our output frames gets written to disk within the allocated time. The write difference is the time from a previous frame write finished to the next. For our pipeline to be proper real-time this delay can not be longer than our time constraint (33 ms). Should modules start to take too long time the pipeline would be delayed and the write difference time would quickly go above our threshold. The write difference times are therefore a good way to see the general performance of our complete pipeline, end to end.

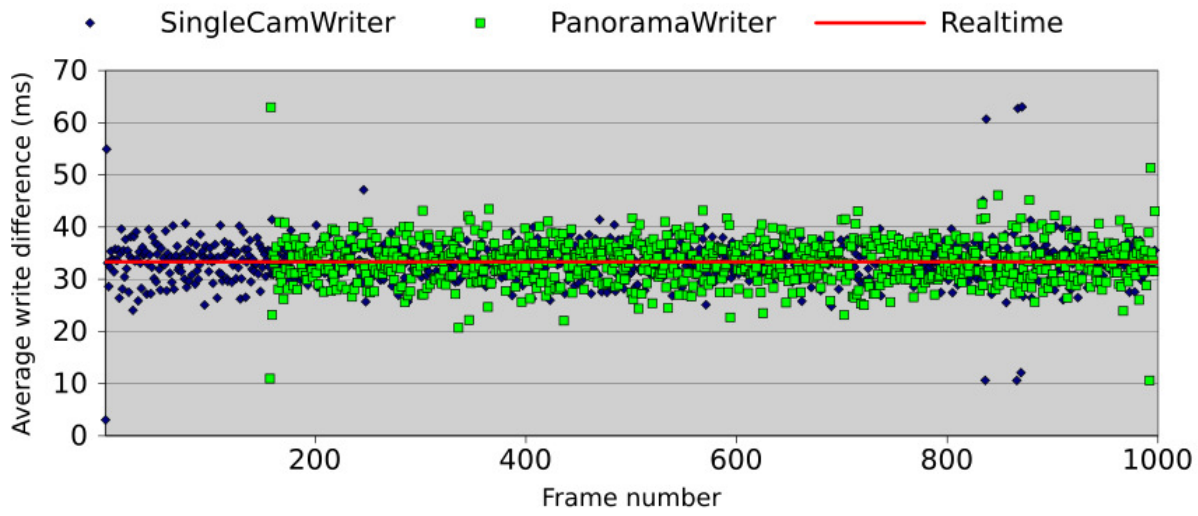


Figure 3.21: Pipelien write difference plot

The scatter plot in figure 3.21 shows the write difference of a 1000 frame run on an Intel Core i7 (table A.3). While there is a spread variation throughout, the average is still at 33 ms which means that the performance is good. The delay between Single-

CamWriter and PanoramaWriter is due to the frame delay buffer (section 3.5) which makes the later part of the pipeline wait for the ZXY data to sync.

3.9.2 Comparison with Old Pipeline

Only a few of the modules in the old prototype pipeline are directly comparable to those in our new pipeline. A graph of the relevant ones can be seen in figure 3.22 and table B.2. The warper, stitcher and RGBA to YUV converter gain an impressive speedup of 8.8×, 106× and 2.7× respectively. The reason why we get so good numbers is likely because the original pipeline was built without much optimization in mind, and the comparable modules all gain much speed from the parallelization introduced with CUDA.

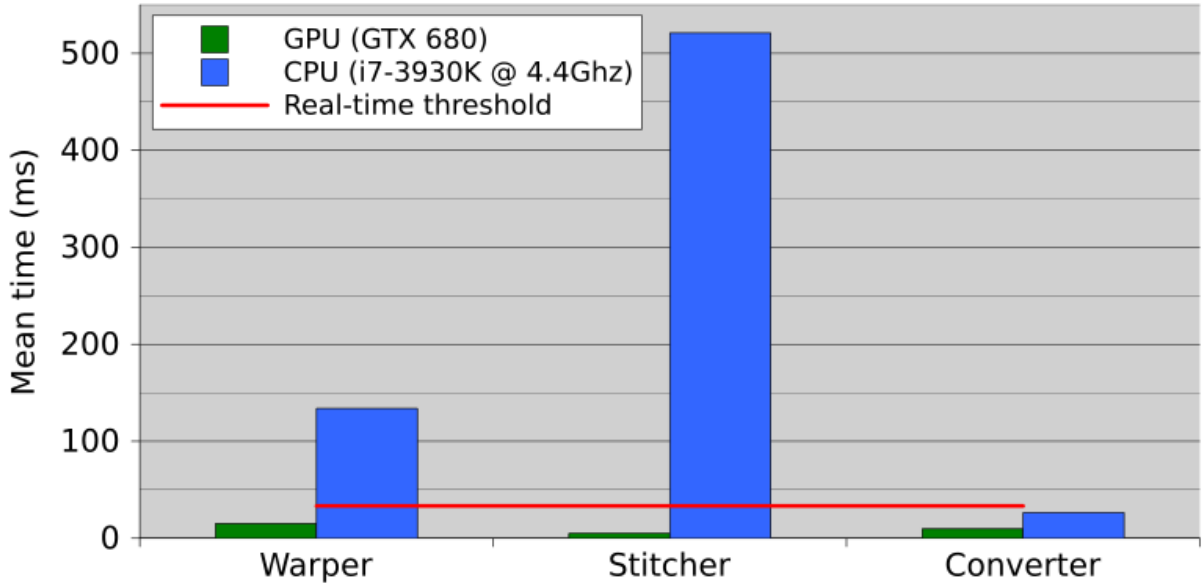


Figure 3.22: Old (CPU) vs. new (GPU) pipeline.

3.9.3 End-to-end Delay

When we say that our pipeline is running real-time, we mean that each step of the pipeline performs equal to or below our real-time threshold (i.e., we can output frames in the same rate as they come in). Naturally the entire pipeline will not be under this limit, but as long as each frame traverses each module at this speed we get frames out of the end of the pipeline with the same speed as we read them from the cameras. Each module will however add delay, and the total time it takes from the start of the pipeline to a final panorama is written to disk looks like this:

$$(10 + 150) \times 0.033 seconds = 5.33 seconds$$

We have 10 modules, each processing the frame at 33 ms along the way. There is also our 150 frame delay buffer (section 3.5) which must be waited out before we see our final frame on the other side. Most of this delay is created by the use of the delay for

the ZXY data, but 5 seconds should still be within an acceptable delay period for most real-world scenarios.

3.9.4 GPU Comparison

The pipeline was tested with several different GPUs to benchmark the performance, and the results can be seen in figure 3.23 and table B.3. All the runs were executed on an Intel Core i7 (table A.3) with the GPU being the only variable. A detailed list of the different cards used and their specifications can be found in table A.5 and A.6 in appendix A. Basically higher number indicates the newer card, with Titan being the newest one.

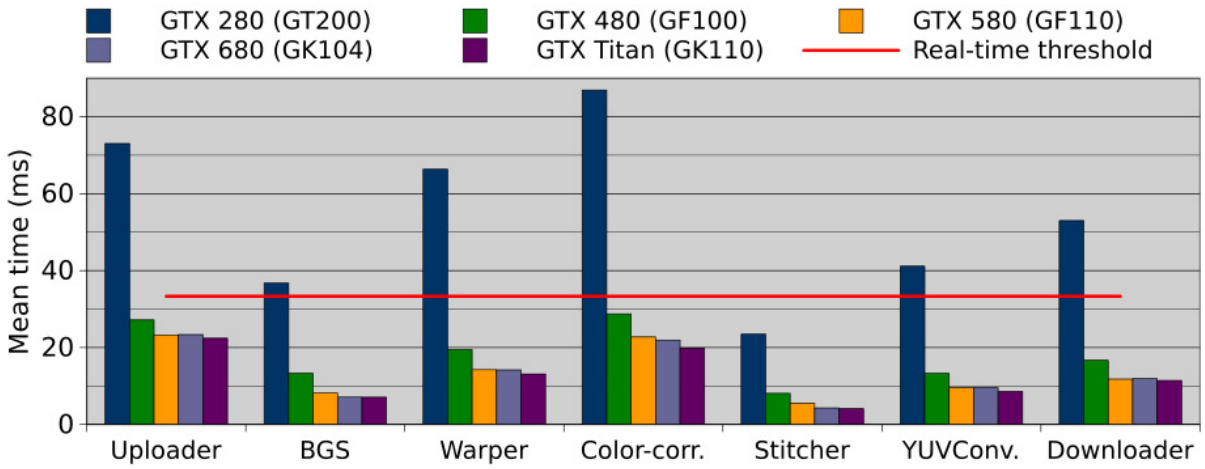


Figure 3.23: Pipeline GPU comparison

We see that there is a clear pattern in the graph, the more powerful the GPU the better the performance. All the cards from the GTX 480 and newer run the pipeline under the real-time constraint with no problem. The older GTX 280, however, can not keep up. We attribute this to the fact that GTX 480 and newer supports compute version 2.0 which allows for concurrent kernel execution. Without this support, all the CUDA code gets run serially, thus severely impacting performance. This would be the case with GTX 280 card which only has support for compute version 1.3.

3.9.5 CPU Core Count Scalability

It is also interesting to see how performance scales along with CPU cores. During development we got access to an Intel Xeon machine (table 3.3), a high end computer with 16 physical cores. This proved very useful for CPU-testing, as we could enable and disable cores as we wanted, and thus could get a good amount of different data sets. Figure 3.24 shows a graph of our findings, corresponding numbers can be found in table B.4.

Computer name	Lincoln
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 3.3: Lincoln specifications.

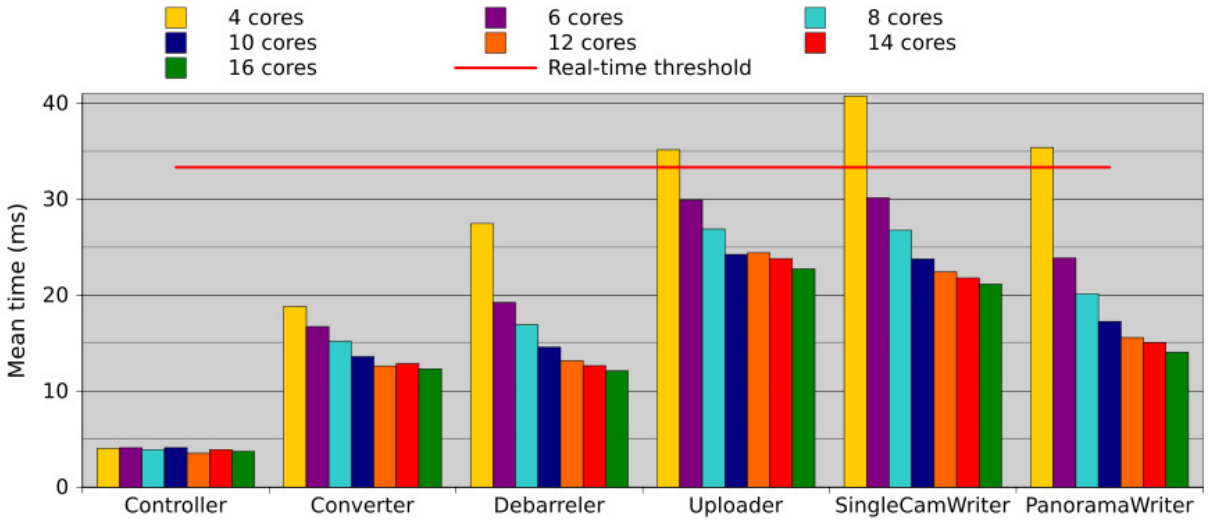


Figure 3.24: CPU core count scalability

We see in the graph that the pipeline scales nice with the core count. The gains seem to flatten out a bit however, and from 12 cores and up the increase in performance between steps are minimal. We also see that the modules using threading (Debarreler, Uploader and the writers) are most positively affected by the core increase. This is expected, as the OS can spread out the different threads to more cores.

The GPU modules are naturally not included in the graph as their performance is mostly based on the GPU performance (section 3.9.4). It should also be noted that the performance of the controller is lower on the Xeon machine than on our Core i7 (roughly 4 vs. 2 ms with all cores active). This is because the Xeon runs at the lower core frequency of 2 GHz, while the Core i7 has 4.4 GHz. Single threaded modules like the Controller does not scale well with the amount of cores, but does instead gain a lot from the increased execution speed.

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

Table 3.4: CPU core count scalability for 1000 processed frames (no frame drop handling)

Frame Drops

In table 3.4 we see the number of frames dropped with the different core configurations. Here we see that lowest configurations (4 and 6 cores) do not manage to keep up with the pipeline and therefore dropping a lot of frames. Only at 10 cores and higher are the drops within acceptable ranges. We can assume the tiny drop variations between 10 cores and up are due to other circumstances like OS or IO-interrupts. These drops are also reflected in figure 3.25 where the diffs of the writers does not go under the real time threshold before 10 cores or more.

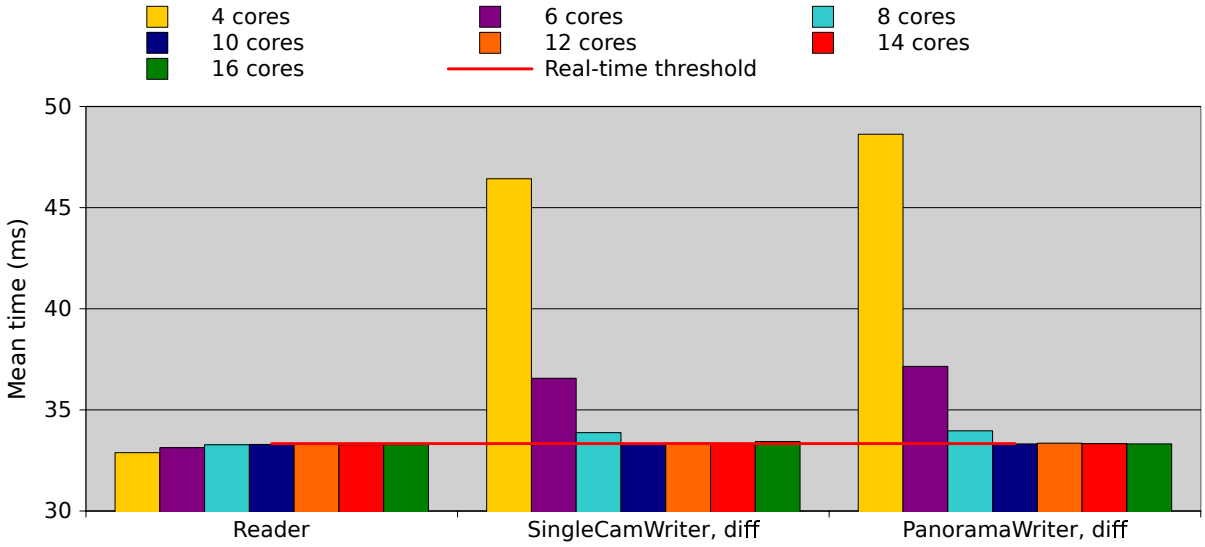


Figure 3.25: CPU core count scalability for reader and writers

Effect of HyperThreading

The previous numbers were generated with HyperThreading (HT) enabled on all cores, so naturally we made some tests with this technology disabled too for comparison. The results can be seen in figure 3.26, table B.5 and 3.27, and the corresponding table B.5.

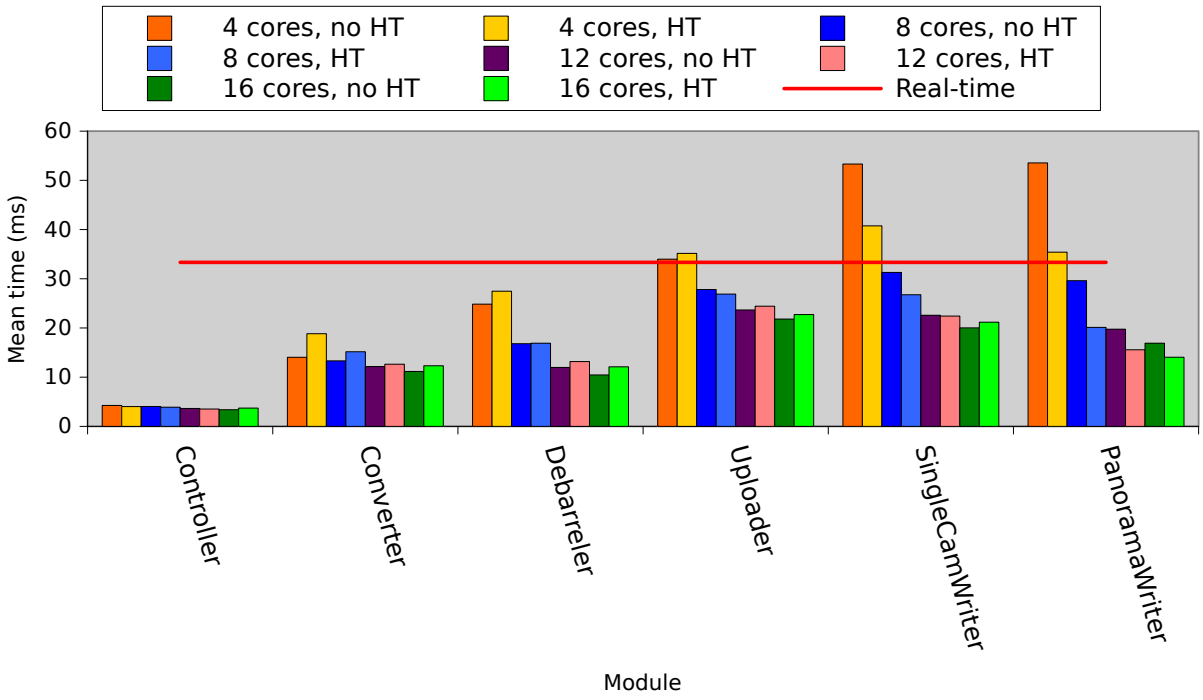


Figure 3.26: HyperThreading scalability

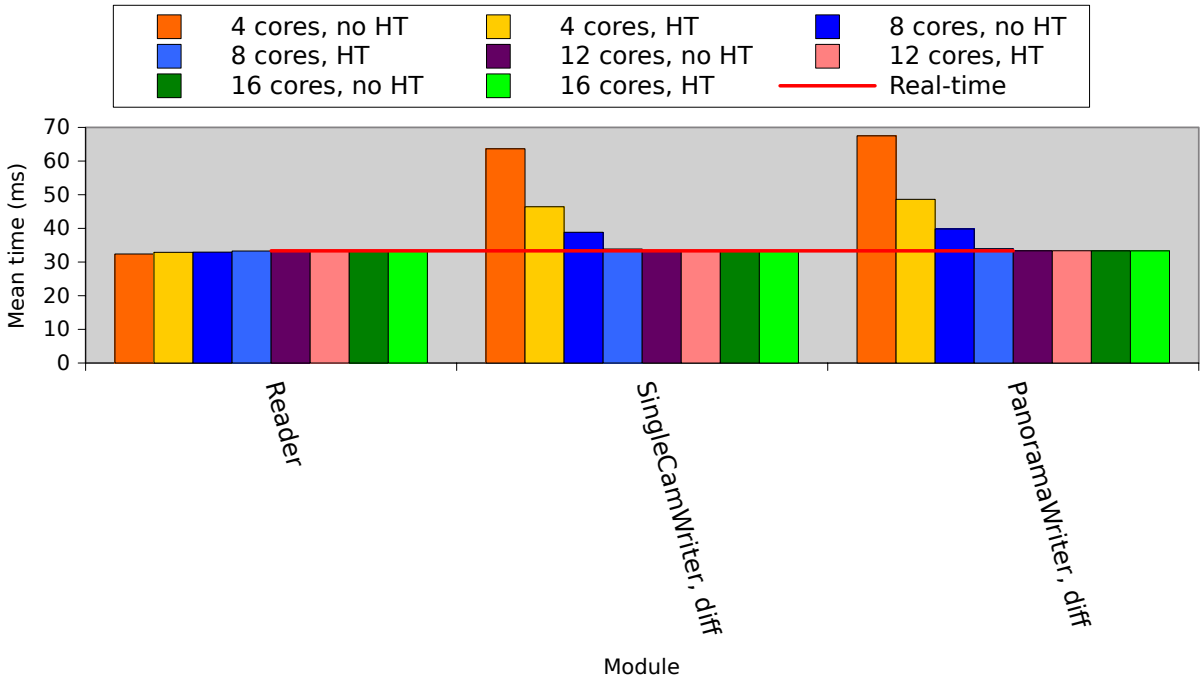


Figure 3.27: CPU core count scalability for reader and writers with HyperThreading

We see that with a lower amount of cores we get a massive increase in performance in threaded modules (e.g. the writers) by using HyperThreading. The other modules generally perform equal or worse. If we look at the drops (table 3.5), we see that HT is

favorable up to 16 cores where the effect drops off.

	4 cores, no HT	4 cores, HT	8 cores, no HT	8 cores, HT	16 cores, no HT	16 cores, HT
Camera frame drops	223	75	54	7	5	8
Pipeline frame drops	1203	729	477	67	3	3

Table 3.5: HyperThreading scalability for 1000 processed frames

Following these results it is very clear that running the pipeline on anything under 8 cores (with or without HT) will not work sufficiently. By using 10 or more cores we get the performance we need, but at this core count disabling HT would prove a bit more efficient than leaving it on.

3.9.6 Frame Drop Handling Performance

Thus far we have shown benchmarks *not* using the pipeline drop handling described in section 3.6.2. This is because this feature can have a big impact on performance if the CPU gets overloaded. To get better benchmark numbers of the modules without any external interference we have therefore left it disabled while testing performance. When running in the real world however this feature will be enabled, so it is also interesting to see how the numbers look when it is actually running. We can see the graph of such a run in figure 3.28 (also in table B.6).

We see that the runs using less than 8 cores seem to get a somewhat unexpected performance boost. The reason for this is simply that using so few cores makes the pipeline drop a lot of frames, which in turn makes the pipeline faster (as no processing is done on dropped frames). The output of such a run will however look less than ideal, as each dropped frame will be replace with the previously processed ones making the video very jumpy. In table 3.6 we see the actual drop rates, and as previously it is only at 10 cores it stabilizes at an acceptable level.

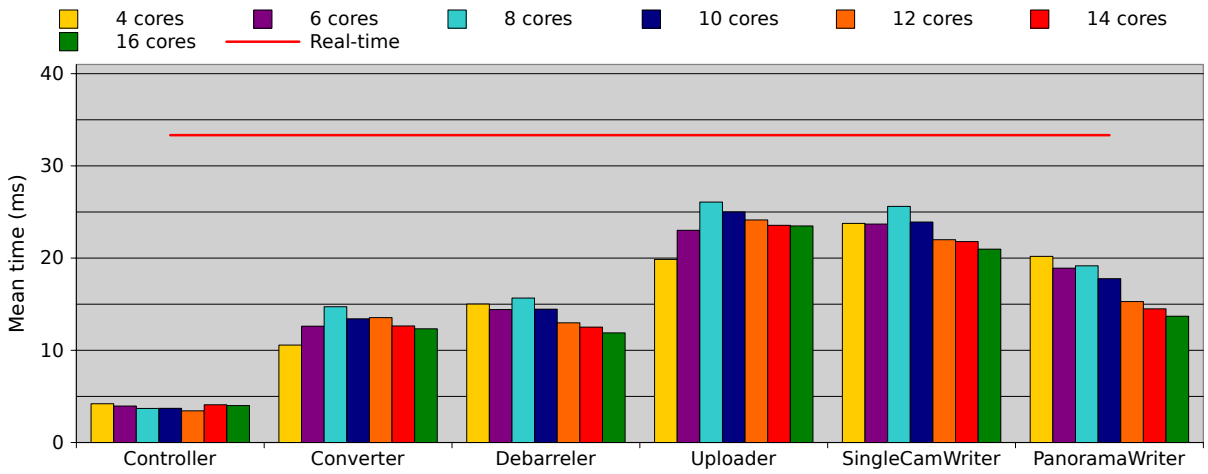


Figure 3.28: Frame drop handling performance

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

Table 3.6: CPU core count scalability, with frame drop handling, frame drops per 1000 frames processed

Write Difference Times

Figure 3.29 shows the new write difference times with frame drop handling enabled. As we see the runs dropping frames (4, 6 and 8) all perform over the real-time threshold. This is because the initial frame will we go over the threshold, dragging the performance down, while the following frames will be skipped to try to catch up. With drop handling on however even the dropped frames will never be processed faster than the real-time threshold, thus never making the average time go down after the initial bump.

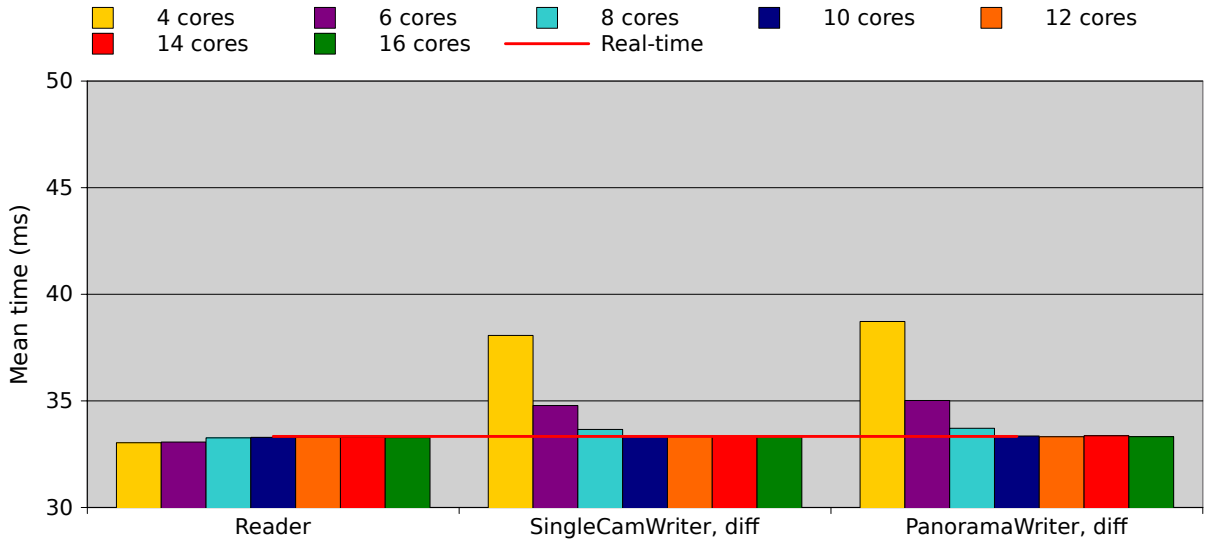


Figure 3.29: Frame drop handling, write difference times

3.9.7 CPU Core Speed Comparison

We also did some additional testing of the pipeline using different CPU core frequencies. Our main testbox (table A.3 in appendix A), which mirrors the machine currently at Alfheim, has an i7-3930K CPU running at 3.2 GHz. Since the machine has adequate cooling and the CPU has an unlocked multiplier, the machine can easily be overclocked to boost performance. We therefore tried our pipeline at several overclocked frequencies to see how it fared. The results can be seen in figure 3.30. The results are as expected a pretty linear drop in processing times when the frequency rises. We also see from the write differences in figure 3.31 that the pipeline still keeps the under the real-time threshold with these new frequencies. So since the overclock to 4.4 GHz proved both stable and added a boost to pipeline performance we decided to keep it.

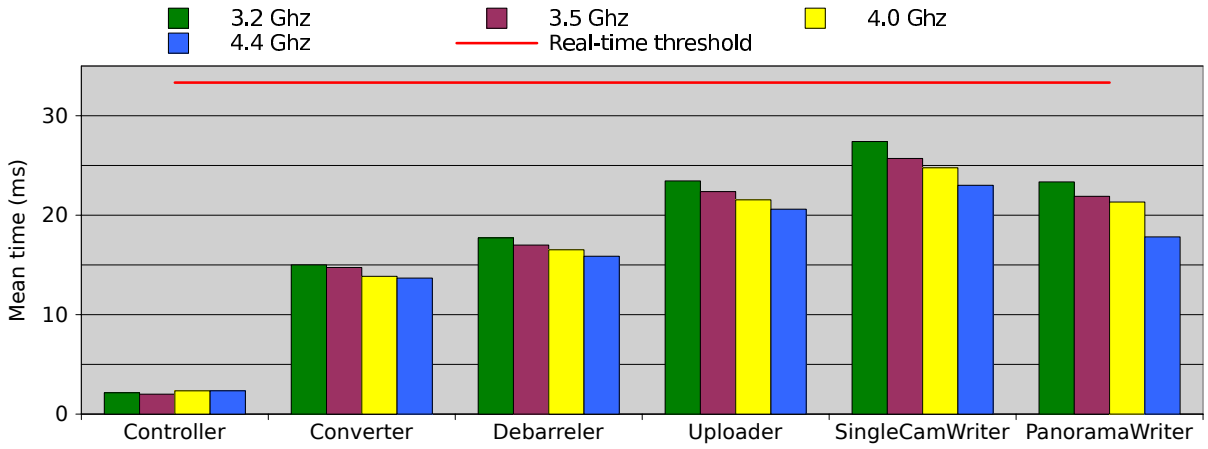


Figure 3.30: CPU frequency comparison

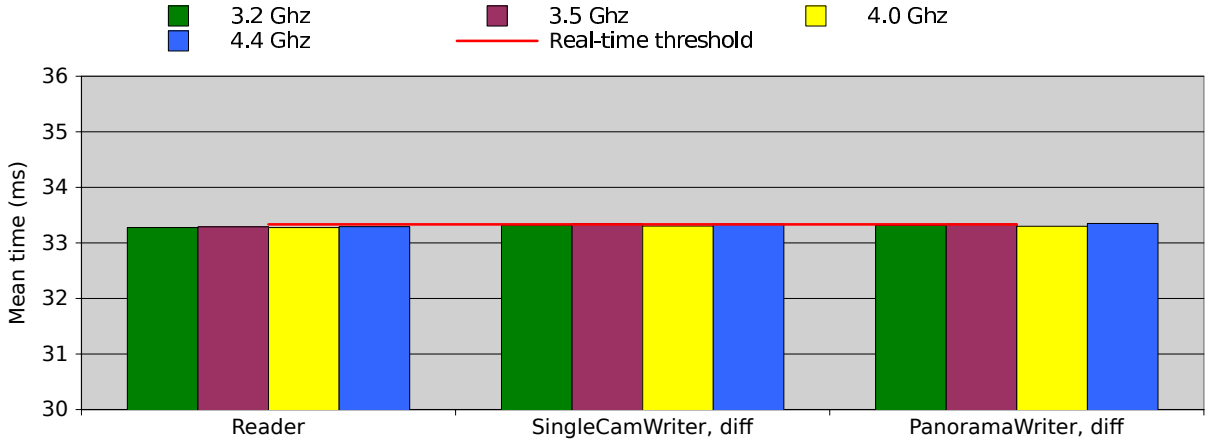


Figure 3.31: CPU frequency comparison, write difference times

3.10 Web Interface

To enable non-technical users to interact with our pipeline a simple web-interface was constructed (figure 3.32). The interface can be opened on any web browser, and gives full access to all functionality from the command line program directly in the browser. The standard workflow of scheduling a record session is as simple as just entering the dates and times into the form and pressing the *schedule* button. A list of currently scheduled tasks are always available at the bottom of the page, and cancelling a job is done by pressing the *stop* button next to it.

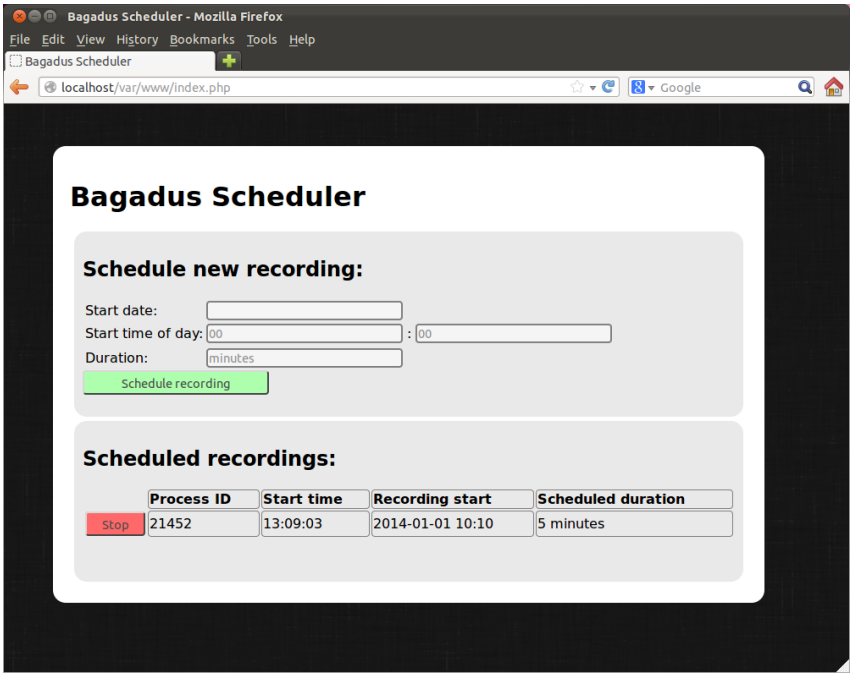


Figure 3.32: The new pipeline's web interface

The web interface is running from an Apache2 installation on the pipeline machine, and the page itself is written in the server side scripting language PHP. Getting the list of scheduled tasks is a simple grep search after pipeline processes on the active process list (from the command line program *ps*), and then parsing the returned matches using the string functions in PHP. Killing entries is done by launching *kill* with the process id found in the previous search. And the actual launching of a new scheduled pipeline run is done by forking a new shell and launching the pipeline process (with the user supplied times and dates) directly from that.

Since the default apache2 user (www-data) lacks privilege to do most of these things some extra configuration steps were needed to get everything working. First of a new user with proper permission to read and write all the pipeline's files must be created and set as Apache2's *RUN_USER*. Secondly apache chroots its running user to the designated web folder (/var/www-data), meaning that you can never go longer up the file tree than that folder, and effectively denying us access to all files we need. We found no simple way to disable chroot in apache, so to solve the problem we simply sat the web folder to the root of the file system effectively making the chroot useless.

3.11 Future Work

There are several issues and potential improvements with this new pipeline, and we will mention some of them here.

A lot of the system per now consists of configurations that are manually found and then hardcoded into the system. This solution requires a complete rebuild if things are changed and is not very elegant for deployment. During our work on the system we

often discussed the need for a separate XML-based program to handle all the configuration data, but unfortunately time never allowed for it to be made.

With regards to color space it could be interesting to see if we could keep the whole pipeline in the same one all the way through. We spend some time converting between RGB and YUV along the way, and it would be interesting to see if it would be possible to avoid this all together. The reason for RGB, as mentioned earlier, is that it is much easier to work with when testing and implementing components. Since we now have a fully working pipeline implementation trying to retrofit it to work directly on YUV all they way through could be an interesting challenge.

Most of the modules are only optimized until they are "good enough", i.e., under the real-time threshold for our current scenario. For future scalability this might no longer be good enough, so more effort should be devoted to improving general performance. Especially modules like the Debarreler, which currently use stock OpenCV code, has great potential for improvement either through smarter or a new from scratch implementation.

In the future we would like to see support for better cameras. This would allow for much better quality pictures and crops, but would drastically heighten the performance requirements. As we have seen we are already pushing the existing hardware pretty hard, so by introducing the increased pixel count from future 2K and 4K cameras would probably lead to many scalability issues. We do however have access to equipment that can help mitigate the load, such as using multiple GPUs in SLI-configuration or Dolphin expansion cards [38] that allows us to distribute processes and memory among several physical machines.

3.12 Summary

In this chapter we have looked at how we created an improved video stitching panorama pipeline based on the previous implementation in our Bagadus system. We first looked at how the setup differs from the old system, before we looked deeper at the implementation of the components. All modules in the system were then throughout explained, and we look closely at how they are put together and how they work. Following this we dedicate a large section to performance numbers of our pipeline, before a brief section about the web interface used to control it. Lastly we have a part about some current issues and future improvements we would like to see in our system.

In the next chapter we will investigate the stitcher module in detail, and how we have evolved it from the simple version in our first prototype, to the much more powerful version currently in use.

Chapter 4

Stitcher

We have described the stitcher earlier, first briefly in chapter 2 and then a bit more in-depth in chapter 3. However, since the main focus of this thesis is the stitching part of our pipeline, we will spend this entire chapter looking at the stitching module in fine detail. As we saw in the earlier chapters, our static seam stitcher produced working panorama images, but with possible visual artifacts. The performance of the first stitcher was also very poor. In this chapter we will first look at how we improved the existing stitcher, before we continue to the new dynamic stitcher that was later implemented.

4.1 Improving the Initial Stitching Code

The initial stitching code (in our prototype pipeline (section 2.4.3)) had pretty bad performance, resulting in very slow execution speed. Since the overall goal of our stitching work is to try to get the code running as close to real-time as possible, some work went into speeding up the existing stitching implementation.

4.1.1 Vanilla Implementation

In the non-modified prototype implementation (section 2.4.3), our stitch was a product of several steps. The following is the pseudocode for the operations used, start to finish:

1. Find shared pixels between the 4 cameras, generate warp transforms.
2. Calculate the size of complete stiched image (full panorama size).
3. Warp pixels.
4. Pad the warped camera images to fit the resolution of the output image. The padding will be applied in such a way that the image is moved to same position it would have in the output.
5. Loop through all pixels of the output image. Choose from which camera to take the value based on static cuts. Since the images have been padded to be directly

overlayed, it's a simple *out(x,y) = in(x,y);* operation, where *in* is set to the correct camera based on the cuts.

6. Crop the final image based on fixed crop values.

Note that the stitcher itself is only step 5, the rest are done in other parts of the system. The approach outlined here is not very smart, and we will now look closer at how it can be improved.

4.1.2 Optimizing Border and Cropping

As previously mentioned, the vanilla approach is slow and not very efficient. The first obvious fix to try to boost the performance is to look at the padding part. Since each of the 4 frames gets padded to the whole size of the calculated output image, we're throwing a lot of memory at a problem that can be solved by a smarter approach. By not padding the pictures and instead using an offset when plotting the pixels in the output image, we can get exactly the same result with a lot less memory.

In the final step, the output image is cropped with some fixed values to shave off some of the rougher parts of the image. Basically, this means that anything outside the cropping zone will not carry over to the final image, and therefore can be discarded. By making the loop setting the pixels in the output to skip these pixels we do get a slight speed improvement over the vanilla version. Using the default sizes and crop offsets, we reduce the amount of horizontal pixels from 10 233 to 5 414 ($\sim$ 53%). An illustration of the cropping can be seen in figure 4.1.

Figure 4.1: Optimized crop

4.1.3 Optimizing Matrix Operations

The next optimization is to look at how the values are set in the output image. In the vanilla implementation, the loop setting the pixels runs through each of the three color channels (RGB) for each pixels and manually setting them based on the cut offsets. This is a *very* slow operation, and by far the biggest bottleneck in the whole program after the earlier cropping optimizations had been implemented. Since the cuts are static we know exactly how many pixels from each camera we want, and implementing a better way of setting the data is trivial.

Our approach is to directly copy the needed data from the selected camera to the output picture using the block memory copy call *memcpy*. Both the camera data and the output image is in the native OpenCV matrix format *CV_8UC3*, where each color channel is stored in an unsigned byte and lying sequentially in memory (i.e., R_1 G_1 B_1 R_2 G_2 B_2 ...). So to copy a line of pixels, we just memcopy a range starting from the offset of the first pixels with the length of the amount of pixels needed

times 3 (since we need all channels). Since our cuts have a fixed size, we know exactly the width we need from each camera, and we can therefore reduce looping through the 5 414 horizontal pixels manually to four *memcpy* operations (one from each camera) for each vertical line. Figure 4.2 illustrates how this copy operation looks.

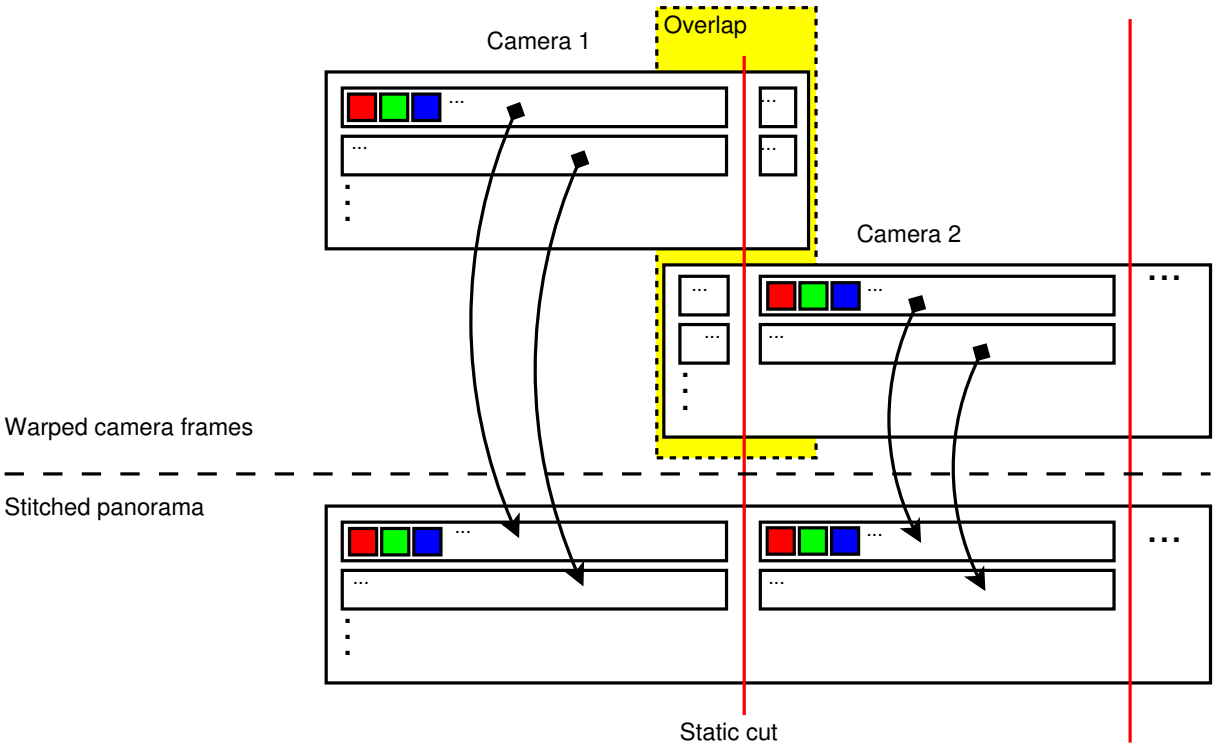


Figure 4.2: Data copy operation in the improved fixed cut stitcher. Each arrow is an individual memory copy operation, and the panorama gets filled in row by row, left to right.

Table 4.1 shows the performance running on an Intel Core i7 machine (See Fillmore, section A.1.1). We see that our implemented optimizations have a fairly good effect on the performance, with the matrix version giving an 82% decrease in run-time from the initial implementation. Unfortunately, the times are still much too high for real-time use, so even further optimization must be investigated.

	Vanilla	Border and crop	Matrix
Time (ms)	998.7	629.6	178.8

Table 4.1: Stiching data for 4 frames

4.1.4 GPU Hybrid Using OpenCV

While the CPU version got much improved using the previous optimizations, the stitching times were still far from ideal, and a totally unacceptable if we wanted it to run real-time. We therefore quickly realized that we needed a better solution to speed up the performance. Since work at that time had started to move the whole panorama

pipeline over to GPU, we also investigated how we could get the stitcher to run on it too.

Since the initial pipeline was built using mostly OpenCV operations and all our frame data was stored in the *cv::mat* format, we first tried using the built in GPU functions. The OpenCV_GPU [39] module can be enabled by recompiling the library with certain flags set, and gives access to some GPU-accelerated functions that can severely boost performance.

A test version that did both the warping and the stitching step of the pipeline on GPU using the GPU-accelerated OpenCV functions was conducted. We quickly discovered that the overhead and execution time using this approach was much worse than the previous CPU-version, and not at all useable in our improved pipeline. Further experimentation using OpenCV was therefore scrapped in favor of building a pure implementation from scratch.

4.1.5 GPU Version Using NPP/CUDA

As the OpenCV_GPU experimentation proved fruitless, we started looking into alternatives to get the stitcher to run directly on GPU. We quickly discovered Nvidia Performance Primitives (NPP) [34] which is a CUDA-wrapping utility library for image manipulation and other common GPU-tasks. NPP contained all the functionality we needed to run both the warper and the stitcher step directly on GPU while keeping the frame data in the GPU-memory between the steps. The actual stitching is done the same way as described in section 4.1.3, but now using NPP specific GPU-functions.

Table 4.2 shows the performance of our NPP-based warper+stitcher running on an Intel Core i7 machine (See Fillmore, section A.1.1). As we see in the table this approach proved very fast, and shows that real-time performance of both the warper and stitcher definitively is possible on GPU. The total run-time is high as it includes the slow NPP GPU-initialization step, but this is only required once at the very start of a run.

	NPP implementation
Warping step	67 µs
Stitching step	38 µs
Total run-time	540 ms

Table 4.2: Stiching and warping times for 4 frames using NPP

With the stitcher now running at microsecond times, further improvement of the fixed cut stitcher is strictly not needed anymore. Using the NPP implementation we can achieve real-time performance, producing the exact same panorama as the vanilla implementation, using only a fraction of the time. With the performance requirement now being covered, we started looking into how to further improve the stitcher using other means, and this is covered in the next section.

4.2 Dynamic Stitcher

As we observed above (section 4.1.5), we could run the old stitcher in real-time on GPU. Since we then needed to move it into its own separate module as part of the new improved pipeline (section 3.7.9), we looked into how we could try to actually improve it at the same time. As the performance requirement was now regarded, the focus would be on the visual quality and how that can be enriched. The panoramas made using the fixed cut stitcher were serviceable, but allowed for visual errors due to its static nature. Hoping to better this, we quickly decided that we wanted to try to make a more dynamic version of the stitcher, which will be detailed in this section.

4.2.1 Motivation

While a static cut stitcher runs very fast and often can give good enough results, there are several scenarios where it will not give optimal output with regards to the visual result. Especially motion or object movement around the seam area can result in visual artifacts and distortion (see figure 4.3). By having a more dynamic seam that can smartly route around important parts of the image, one can achieve a much better looking stitch.

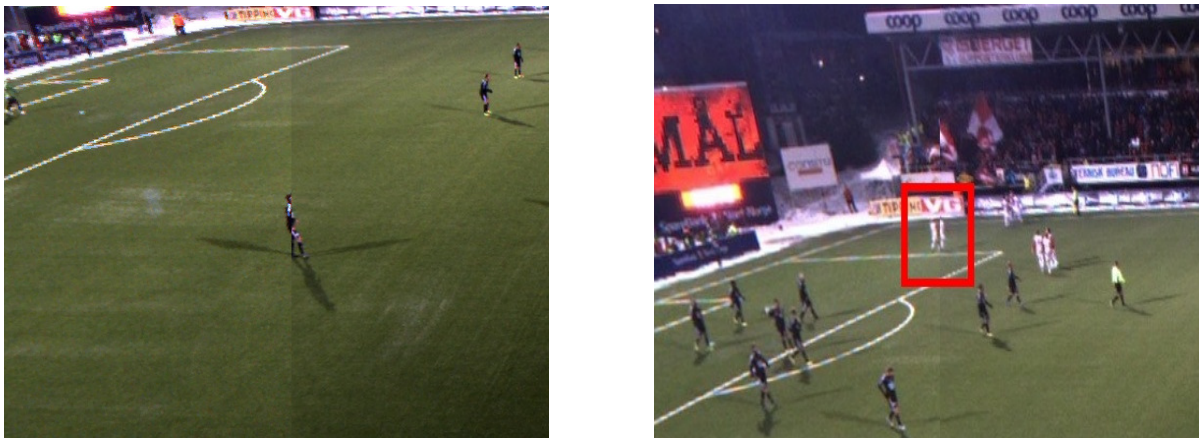


Figure 4.3: Examples of players getting distorted in static seam.

Another big advantage of a dynamic stitcher is that it can take image color variations into account when creating the seam. The idea behind this being that it can route the seam through areas in the overlapping frames where the colors match, making the cut more seamless and harder to spot. This would of course require the stitched images to already be color corrected, so that the frames have a common color balance (which the pipeline does as described in section 3.7.8).

4.2.2 Related work

There are many relevant papers on dynamic stitching, but the approach closest to our pipeline is Mills [10]. While Mills only use color and pixel differences as a base for

seam detection, we also have our ZXY-data to be able to detect potential player pixels, which again mean we can get away with a simpler color difference checker. Mills also uses blending for creating the final panorama, which we do not as it can introduce an unwanted visual effect called ghosting.

We also both use Dijkstra's algorithm [40] (Dijkstra) for finding the actual seam, but Mills does unfortunately not document anything about his actual implementation at all. It is only mentioned that it is done in Matlab, and that it uses 56 seconds on two 300 x 400 pixel images, which is relatively slow.

4.2.3 Implementation

In this section, we will take a deeper look at the implementation details of our dynamic stitcher by walking through each of the steps.

Preparation

For each seam the implementation has several requirements that must be met before it can start working.

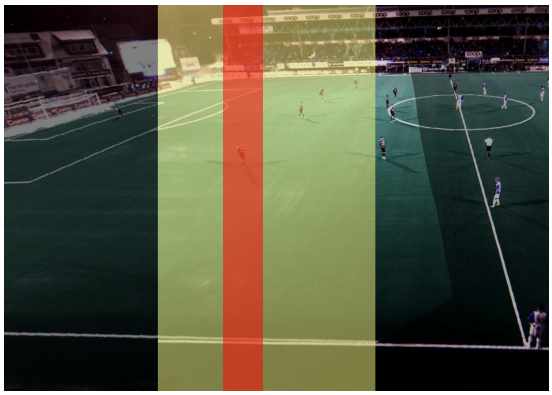
- First is the two images to stitch, which must be properly debarreled and warped to the correct perspective. In our pipeline this is done in previous steps and the frames delivered to the stitcher module are all ready for stitching.
- Second is the offset position for the stitch, for each seam we need the position in the output (panorama) frame where we want the stitch to be. In our implementation these offsets are found manually by overlapping the warped images and finding a suitable position within the overlap (more detailed in the next section). Ideally some sort of automated way of finding the offsets would be preferred, but there were no time to develop such a system.

Once we have these prerequisites, we can move on to the next step.

Find Search Area

After the overlapping parts of the frames have been found and matched, a search region for the seam can be established using a position within the overlap as the main offset. As mentioned earlier the offset value is manually found and hardcoded into our config for each seam. Since our camera setup is static, and the cameras will never move during execution, this is only done once. When we have this value we can create the search area itself by making a rectangle centered at the offset, with an adjustable width, over the pixels in said offset. The width needs to be flexible as the actual overlap can differ between frames, and having a too big area might introduce bad seams. A smaller width also makes the search area smaller, resulting in a small performance increase in favor of a potentially less dynamic seam. To ensure a good seam the search area width should therefore always be bigger than a single player's largest width in the frames in question (to be able to route around him), and never bigger than the overlap of the frames. Figure 4.4(b) shows an example of a typical search area.

(a) Overlapping section between two warped frames.



(b) Yellow shows suitable area for search. Red shows typical seam search area 100px wide.

Figure 4.4: Finding the overlap and seam search area.

Creating a Graph

Our seam detection is done using a simplified version of Dijkstra's algorithm (later detailed in section 4.2.3), so for that to work we first need to create a graph to work on. When the search area discussed above (section 4.2.3) has been found we treat the pixels contained within as a graph in order to make a seam. Each pixel represents a node, but edges between them needs to be established to enable traversing of the graph. How the outbound edges of each node is set up can greatly affect the final seam will look, but in general we want each pixel to relate only to adjacent pixels. If we allow for diagonals, an ordinary pixel in a grid will have 8 adjacent pixels. We will traverse the graph from bottom to top and we ideally want as straight line as possible with no backtracking. We will therefore not add edges going downward from a node, thus reducing the edge count to 6. Horizontal lines in the seam also has a tendency to be very noticeable regardless of the stitch quality, so in order to avoid these we also do not add edges to nodes directly left or right of the current node. And so each node will only have 3 edges, the nodes above, above-left and above-right of the current position. There are of course edge cases at each end of the search width, where the left and rightmost pixel only will have two upwards adjacent pixels. Figure 4.5 shows all three unique pixel positions encountered and how their edges will look. Note that the topmost row of nodes will not have any outbound edges, as we don't want to move out of bounds while searching.

Assigning Weights

Each of these edges also needs an associated weight for the seam detection to work in a similar, but simplified version of the Dijkstra algorithm. The weight is based on the colors of the two images we are stitching, and is found using the current function:

$$|Image1(x,y) - Image2(x,y)| + ZXYweight$$

Image1 and *Image2* are the left and right images of the current stitch. The returned value is the pixel color (in RGB). *ZXY weight* is a predefined, and very high, value added to

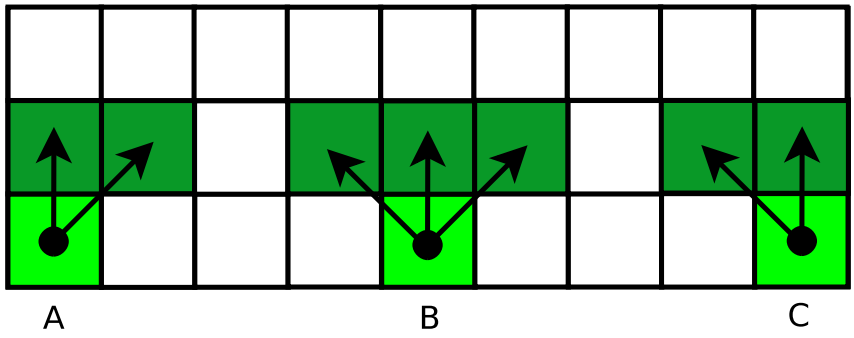


Figure 4.5: How nodes in the graph are connected. *B* shows a normal pixel with three outbound edges, *A* and *C* shows the edge cases only connecting to two adjacent nodes.

the weight if there is a player visible on the pixel in any of the frames (figure 4.6). We use the look up map from the background subtractor to check this. The end result of these edge weights is that paths to pixels where the differences between the stitched images are smaller cost less. And if a player is present in the pixels, the cost skyrocket, making it an unattractive option. It should also be noted that using color corrected images will yield much better seams than non-corrected ones; this is one of the reasons for why we have a color corrector module earlier in the pipeline.

(a) Original frame.

(b) Same frame with ZXY weighted pixels overlayed in white.

Figure 4.6: Example of the ZXY weighted pixels in a frame. White pixels will get an almost infinite edge weight when constructing our seam graph, to avoid cutting through players.

Once we have established both a search area and constructed both the directed graph and set all the edge weights, finding the actual seam is trivial. By applying a shortest path algorithm to our graph, we will get the path through it with the least cost. As mentioned earlier we use a simplified version of the Dijkstra algorithm. Since the cost is modeled directly after the colors in the pixels, the lowest cost path is usually the one with the least variation and thus as seamless as possible.

Dijkstra Algorithm

For solving the graph we use the common Dijkstra shortest path algorithm. Several pre-existing graph libraries and Dijkstra implementations for C++, like Boost Graph Library (BGL) [41], Lemon [42] and LP [43] were tested.

	LP	Lemon	Own
Create graph	248.19	16.24	10.97
Run Dijkstra	829.64	25.92	15.57
Total	1077.83	42.17	26.54

Table 4.3: Time (ms) for the different Dijkstra implementations.

These CPU-libraries would work fine but usually have very tricky initialization steps for mapping the pixels to a graph and also somewhat lacking overall performance (see section 4.3). None of them had code that was easy to port over to GPU either. In the end we decided to go for our own implementation of the algorithm. By making everything ourselves we would not only get fine grained control over how the graph is made, but also allow us to tweak certain properties in the interest of speeding things up.

The Pseudocode for our Dijkstra solver looks like this:

1. Make two empty arrays PrevNode[] and MinDistance[], with the dimensions of the seam search area.
2. Set the bottom start position (lowest cut offset point) of MinDistance to 1.
3. Start from the bottom of the graph, for each row until we reach the top:
   (a) For each pixel P in current row:
      i. If MinDistance[P] is not 0: For all adjacent pixels P2: Value = MinDistance[P] + P2 edge weight. If Value < MinDistance[P2]: Set MinDistance[P2] = Value and PrevNode[adjacent pixel] = pixel.
4. By iterating through PrevNode[] starting from PrevNode[end position] we get the path through the graph.

This is fairly close to the standard algorithm, but since our graph is directed upwards we are never going back to earlier rows. In standard implementations backtracking are solved using slow list operations, but since we are avoiding that we are getting a decent speed boost. This version of Dijkstra is also easy to port over to CUDA, which was an important factor for our pipeline.

CUDA Implementation

Our stitcher implementation is written directly for CUDA as .cu file and compiled together with the rest of the pipeline. It is initialized by the pipeline controller at startup, and is then executed by the Stitcher module as we saw in chapter 3.

We are utilizing the parallelism in CUDA by running parts of our stitching at the same time. The complete stitch is done in two steps. First the seams between our four frames are found using the graph technique detailed above. These three operations, i.e., finding the 3 seams between the four cameras, are all running in parallel. Then when we have all the offsets from the seams, we copy the data from the individual frames to the final output panorama using 32-bit memory transfers as this improves speed. We do this by splitting the whole image we want to transfer into 32-bit chunks, and then by running *all* of the copy operations simultaneously. This is done using CUDAs support for parallel kernel execution. We make one CUDA thread with a simple copy operation for each 32-bit block to copy, and then all the threads are launched at the same time.

4.2.4 Performance

Table 4.4 shows the timings for our dynamic stitcher, both the CPU and the GPU version. Both are well within our real-time threshold (33 ms). The "low"-value on GPU is zero due to our frame drop handling causing the module to not run on dropped frames, while still being timed.

Also note that the CPU-version has a lower mean than our GPU-version. This is most likely due to the GPU-version not being optimized well enough, and it should potentially be possible to even them out with further work on the GPU-code. The reason we are not using the CPU-version is that the rest of our pipeline is on GPU, and movement of data between GPU and CPU incurs a big performance penalty. We therefore keep all data on the GPU for as long as possible to avoid this, even if it means using some GPU-modules that is slower than their CPU counterparts.

	Min	Max	Mean
CPU (Intel Core i7-2600)	3.5	4.2	3.8
GPU (Nvidia Geforce GTX 680)	0.0	23.9	4.8

Table 4.4: Dynamic stitching (ms).

4.2.5 Results

A typical output frame from the dynamic stitcher can be seen in figure 4.7. Note that the cuts are no longer straight, but the seams are still visible due to the color correction being slightly off. In figure 4.8 we see a comparison of the the different stitchers using ideal and properly corrected frame data.

Figure 4.7: Dynamic stitcher output, here shown in full using an earlier color corrector version and experimental white balance correction in our camera driver.

(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.

(d) Dynamic stitch **with** color correction.

Figure 4.8: Stitcher comparison - improving the visual quality with dynamic seams and color correction.

4.2.6 Future Work

The dynamic stitcher was thought out and implemented in a relatively short amount of time. While already working great for our purpose there are some shortcomings and improvements that we would like to see in the future, and we will discuss them here.

Seam Jitter and Noise

The current dynamic stitcher has no kind of condition check, so it will run on every single frame of the video, thus potentially changing the seam every frame too. In an ideal scenario the frames delivered to the stitcher will be perfectly warped with no picture noise and excellent color correction, and the seam will not be visible at any point even if it changes a lot each frame. In real life however, the warps are far from perfect and the color correction might be off at times, making the seams very visible at certain points. Our cameras also introduces a lot of sensor noise (especially in bright conditions), which makes the seam change a lot as it tries to route around it. Since our warps are mostly calibrated around the field itself that part of the image will mostly look good regardless, but the surrounding stadium background (like the bleachers etc.) usually will not. This noticeable seam change in visible areas (however small) is easily seen in the video, and can be perceived as an annoyance for some.

There are several ways to solve this problem; the best is probably better calibration of the whole system, and better cameras or camera configuration to counter the image sensor noise. Recalculating the warp matrixes using calibration points not only on the field, but also on the surrounding geometry should make the whole stitch more seamless, thus making the seam jitter harder to spot. With lower noise the seam will simply change less. Another way to tackle this would be to introduce limiting to the stitcher, either by only running it every x number of frames, or at some kind of event. Since we only really need to recalculate the seam when a player is walking through it, scanning the ZXY-coordinates for such an occurrence every frame and only then run the stitcher could work. Note that there are some gains from running the stitcher every frame that would be lost if this doesn't happen. For instance lighting or other background changes can make the current seam more visible, and since it could potentially take a long time for it to update it would also be noticeable on the video.

Neither of these fixes was tried as there simply was not enough time to test them properly in this thesis.

Dynamic Search Area

The search areas we described in section 4.2.3 are statically set for the reasons discussed there. There can however occur situations where enough players cross the search area simultaneously that the stitcher will have to route through some of them. It would therefore be useful to detect such situations, and change (or move) the search area boundaries accordingly. Note that making this area too large can have a very negative impact on performance as the generated graph size will grow very fast. In a dynamic system great care must therefore be taken not to allow them to grow too much, as this quickly could make the module cross our real-time threshold.

4.3 Summary

In this chapter we have covered the stitcher component in our pipeline. First we looked at how we improved parts of it from the relatively slow original implementation in

Bagadus, and slowly moved it over to GPU. Then we saw how it changed from a simple static cut stitcher to a full blown dynamic one, using ZXY and color data to avoid players and make seamless cuts. We also covered why such a stitcher is useful, and how it is constructed and implemented. We then looked at the performance figures, the output images it produces and finally some future work that remains on the stitcher to further improve the performance.

Chapter 5

Conclusion

In this chapter we present a summarization of our work in this thesis. We then look at our contributions, and finally a brief look at future work.

5.1 Summary

In this thesis we have improved upon the old Bagadus system which we described in chapter 2. We went through the setup of the system, and how the main parts and components worked together to provide a simple panorama stitching solution. We then looked at the performance numbers, and then a bit at the stand alone player made for playing back the generated video.

Following the old setup, we looked at the improved panorama pipeline in chapter 3. Here we described how we restructured the whole original Bagadus setup and moved parts of it from a CPU over to a GPU. We also introduced some new modules and a real-time constraint, and discussed how we are able to get the whole system running on-line by using a modular pipeline approach. We also took a brief look at the custom web interface developed to control the system for people without technical expertise.

Chapter 4 was devoted entirely to the stitcher module. We first looked at how it was improved gradually from the very first version into what it is today. Then we looked at how to improve the stitching itself, and how our static seam stitcher changed into a full dynamic one. We go into details about how the seam creation and actual stitching is performed before we look at how the performance holds up. In the end we looked at some improvements that could be implemented to get it even better.

5.2 Main Contributions

We have shown in this thesis that a real-time panorama video stitching pipeline using four HD-cameras can be made. The pipeline was based on the previous Bagadus prototype, and is currently successfully installed and running at Alfheim stadium in Tromsø. A planned goal of the project is for it to be used for matches in the upcoming soccer season.

We have shown that by utilizing GPUs we can achieve excellent performance and improved visual quality while still keeping the system running in real-time. The whole system is run on a single inexpensive commodity computer using standard off the shelf hardware. The total end-to-end delay of just 5.3 seconds also means that processed video can be delivered to receivers in reasonable time, which for instance means that a coach can view video from the first period during the half-time break.

Lastly we have looked at the stitcher and how we improved it to produce much better seams. By routing around players instead of through them we can produce much cleaner and better looking panorama frames in our system. Combined with the color corrector we are also getting much smoother seam transitions, which again leads to much increased visual quality.

5.3 Future Work

We discussed some of the future work of the pipeline in section 3.11. For instance we would like to see better configuration tools for setting up and deploying the whole system. We also would like better scalability for future improvements, including better cameras. Future work for the stitcher module is mentioned in section 4.2.6. Here, ways to combat seam jitter due to noise were discussed, and also how to a more dynamic search area could be useful.

As for the whole project itself, the overall goal is to be able to deliver a full analysis system, ready for deployment and use at any stadium. There is still work left before the project reaches that point, but this improved panorama stitching pipeline represents a major milestone in the back-end of such a system. For the system to be complete it would also require a powerful front-end to present all the generated data in a useful way for coaches and other users. Work has already started on such a front-end delivery system, and once this is properly established the whole system will be a whole lot closer to its intended goal.

Appendix A

Hardware

A.1 Computer specifications

A.1.1 Fillmore

Fillmore is my primary development workstation at the lab. While it has a fairly good CPU it lacks the GPU-power needed for the more intensive pipeline tasks, which means that these are run on other machines.

Fillmore		
CPU	Type	Intel Core i7-2600
	Frequency	3.40Ghz
	Cores (Real/Virtual)	4/4
GPU	Type	nVidia Quadro NVS 295
	Compute capability	1.1

Table A.1: Fillmore specifications

A.1.2 Devboxes

These computers are various machines set up explicitly for testing our pipeline.

Computer name	DevBox 1
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: DevBox 1 specifications

Computer name	DevBox 2
CPU	Intel Core i7-3930K @ 4.4 GHz
GPU	Nvidia Geforce GTX 680
Memory	32 GB DDR3 @ 1866 MHz
Pipeline output storage	Samsung SSD 840 Series, 500 GB

Table A.3: DevBox 2 specifications

Computer name	DevBox 3
CPU	Intel Core i7-960 @ 3.20GHz
GPU	Nvidia Geforce GTX 480
Memory	6 GB DDR3 @ 1066 MHz
Pipeline output storage	N/A

Table A.4: DevBox 3 specifications

A.2 GPU Specifications

These are the specifications of the various cards we looked at in our GPU comparisons (section 3.9.4).

GPU	G98	GT200	GF100
Name	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

Table A.5: GPU specifications, part 1

GPU	GF110	GK104	GK110
Name	Geforce GTX 580	Geforce GTX 680	Geforce GTX Titan
CUDA cores	512	1536	2688
Graphics clock	772 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

Table A.6: GPU specifications, part 2

A.3 Cameras

A.3.1 Basler Ace

Here follows the specifications of the Basler Ace cameras used in our setup.

Basler Ace A1300 - 30gc

This is the standard camera we use in our panorama stitcher.

Resolution horizontal/vertical	1294 x 964 pixels
Max frame rate	30 fps
Sensor type	CCD
Sensor size	1/3"
Supported Output Formats	Mono 8, Bayer BG 8, Bayer BG 12, Bayer BG 12 Packed, YUV 4:2:2 Packed, YUV 4:2:2 (YUYV) Packed

Table A.7: Specifications for Basler Ace A1300 - 30gc

Basler Ace A2000 - 50gc

The 50gc is better version of the 30gc with a bigger image sensor and support for higher resolutions. We only have one of these, and use it strictly for testing purposes.

Resolution horizontal/vertical	2046 x 1086 pixels
Max frame rate	50 fps
Sensor type	CMOS
Sensor size	2/3"
Supported Output Formats	Mono 8, Bayer BG 8, Bayer BG 12, Bayer BG 12 Packed, YUV 4:2:2 Packed, YUV 4:2:2 (YUYV) Packed

Table A.8: Specifications for Basler Ace A2000 - 50gc

Appendix B

Extra Tables

Overall Pipeline Performance

Referenced in section 3.9.

Computer	DevBox 2
Controller	1.791
Reader	33.285
Converter	13.855
Debarreler	16.302
Uploader	23.892
Uploader, BGS part*	13.202
BGS	8.423
Warper	15.391
Color-corrector	23.220
Stitcher	4.817
YUVConverter	9.938
Downloader	12.814
SingleCamWriter	24.424
PanoramaWriter	19.998
SingleCamWriter, diff	33.339
PanoramaWriter, diff	33.346
BGS, ZXY query†	657.597
Camera frame drops/1000	4
Pipeline frame drops/1000	0

Table B.1: Overall pipeline performance
Mean times (ms)
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Old versus New Pipeline

Referenced in section 3.9.2.

Pipeline Version	New (GPU)	Old (CPU)
Warper	15.141	133.882
Stitcher	4.912	521.042
Converter	9.676	26.520

Table B.2: Old vs new pipeline.
Mean times (ms). DevBox 2.

GPU Comparison

Referenced in section 3.9.4.

GPU	GTX 280	GTX 480	GTX 580	GTX 680	GTX Titan
Uploader	73.036	27.188	23.269	23.375	22.426
BGS	36.761	13.284	8.193	7.123	7.096
Warper	66.356	19.487	14.251	14.191	13.139
ColorCorrector	86.924	28.753	22.761	21.941	19.860
Stitcher	23.493	8.107	5.552	4.307	4.126
YUVConverter	41.158	13.299	9.544	9.566	8.603
Downloader	53.007	16.698	11.813	11.958	11.452

Table B.3: GPU comparison, mean processing times (ms)

CPU Core Count Scalability

Referenced in section 3.9.5.

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
Reader	32.885	33.132	33.275	33.292	33.287	33.280	33.281
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
Uploader, BGS part*	18.482	15.865	14.325	13.171	12.474	12.505	11.834
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
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
BGS, ZXY query†	685.404	671.347	660.456	675.240	692.639	639.769	688.503
Camera frame drops/1000	75	26	7	9	6	8	8
Pipeline frame drops/1000	729	327	67	0	6	3	3

Table B.4: CPU core count scalability, without frame drop handling, mean times (ms).
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

HyperThreading Scalability

Referenced in section 3.9.5

	4 cores, no HT	4 cores, HT	8 cores, no HT	8 cores, HT	16 cores, no HT	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
SingleCamWriter	53.313	40.752	31.290	26.754	20.023	21.173
PanoramaWriter	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
BGS, ZXY query†	680.114	685.404	708.971	660.456	643.523	688.503
Camera frame drops/1000	223	75	54	7	5	8
Pipeline frame drops/1000	1203	729	477	67	3	3

Table B.5: HyperThreading scalability, without drop handling, mean times (ms).
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

CPU Core Count Scalability with Drop Handling

Referenced in section 3.9.6

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
Reader	33.037	33.070	33.266	33.290	33.301	33.286	33.277
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
Uploader, BGS part*	14.859	14.447	14.314	12.913	12.614	12.411	11.644
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
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
BGS, ZXY query†	656.593	679.531	669.598	699.519	641.223	636.265	668.108
Camera frame drops/1000	41	33	7	4	3	4	4
Pipeline frame drops/1000	343	177	37	6	2	7	3

Table B.6: CPU core count scalability, with frame drop handling, mean times (ms).
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Compiler Optimization Comparison

Included for completeness, shows timings using different optimization levels in the GCC compiler.

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
BGS, ZXY query*	668.108	694.356	632.797
Camera frame drops/1000	4	2	3
Pipeline frame drops/1000	3	0	0

Table B.7: Compiler optimization comparison, mean times (ms).
** Not a separate module, but is a part of the total Uploader time usage*
† Not a module affecting the real-time constraint of the pipeline. Is executing separately

Appendix C

Accessing the Source Code

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

Bibliography

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

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

[3] Stats technology. http://www.sportvu.com/football.asp.

[4] Camargus - premium stadium video technology infrastructure. http://www.camargus.com/.

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

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

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

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

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

[10] Alec Mills and Gregory Dudek. Image stitching with dynamic elements. *Image and Vision Computing*, 27(10):1593 – 1602, 2009. Special Section: Computer Vision Methods for Ambient Intelligence.

[11] 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.

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

[13] Marius Tennøe, Espen 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.

[14] Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. Submitted for publication, ACM Multimedia, 2013.

[15] 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.

[16] 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 ACM Multimedia Systems conference (MMSys)*, February 2013.

[17] 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.

[18] swscale homepage. `http://ffmpeg.org/libswscale.html`.

[19] x264 homepage. `http://www.videolan.org/developers/x264.html`.

[20] Opencv homepage. `http://www.opencv.org/`.

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

[22] C.J. van den Branden Lambrecht. *Vision Models and Applications to Image and Video Processing*. Springer, 2001.

[23] Openframeworks homepage. `http://www.openframeworks.cc/`.

[24] Bagadus video demonstration. `http://www.youtube.com/watch?v=1zsgvjQkL1E`.

[25] ofxgui homepage. `http://ofxaddons.com/repos/42`.

[26] 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.

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

[28] 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].

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

[30] Nvidia. *CUDA C Best Practices Guide*, oct 2012.

[31] Nvidia. *CUDA C Programming Guide*, oct 2012.

[32] Nvidia. Cuda programming model overview. 2008.

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

[34] Nvidia performance primitives homepage. https://developer.nvidia.com/npp.

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

[36] Videolan project homepage. http://www.videolan.org.

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

[38] Dolphin interconnect solutions - pci express interconnect. http://www.dolphinics.com/.

[39] Opencv gpu module. http://opencv.willowgarage.com/wiki/OpenCV_GPU.

[40] E.W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[41] Boost graph library homepage. http://www.boost.org/doc/libs/1_53_0/libs/graph/doc/index.html.

[42] Lemon homepage. http://lemon.cs.elte.hu/trac/lemon.

[43] Literateprograms dijkstra implementation. http://en.literateprograms.org/Dijkstra's_algorithm_(C_Plus_Plus).