UiO : Department of Informatics
University of Oslo

Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views

Ragnar Langseth

Master's Thesis Spring 2014

Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views

Ragnar Langseth

Abstract

Today, we are continuously looking for more immersive video systems. Such systems, however, require more content, which can be costly to produce. A full panorama, covering regions of interest, can contain all the information required, but can be difficult to view in its entirety.

In this thesis, we discuss a method for creating virtual views from a cylindrical panorama, allowing multiple users to create individual virtual cameras from the same panorama video. We discuss how this method can be used for video delivery, but emphasize on the creation of the initial panorama. The panorama must be created in real-time, and with very high quality. We design and implement a prototype recording pipeline, installed at a soccer stadium, as a part of the Bagadus project. We describe a pipeline capable of producing 4K panorama videos from five HD cameras, in real-time, with possibilities for further upscaling. We explain how the cylindrical panorama can be created, with minimal computational cost and without visible seams. The cameras of our prototype system record video in the incomplete Bayer format, and we also investigate which debayering algorithms are best suited for recording multiple high resolution video streams in real-time.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Bagadus System . . . 2
1.3 Problem Definition . . . 6
1.4 Limitations . . . 7
1.5 Research Method . . . 8
1.6 Main Contributions . . . 8
1.7 Outline . . . 9

2 Recording Pipeline **11**
2.1 Motivation . . . 11
2.2 Related Work . . . 12
2.3 Pipeline Features . . . 12
2.4 System Design . . . 13
2.4.1 Modules . . . 13
2.4.2 Active modules . . . 14
2.4.3 Internal module design . . . 15
2.5 Distributed Recording - Machine Setup . . . 15
2.5.1 Current machine setup . . . 16
2.5.2 Communication . . . 16
2.6 Nvidia CUDA . . . 17
2.6.1 Programming using CUDA . . . 17
2.6.2 Memory types . . . 18
2.6.3 Performance pitfalls . . . 19
2.6.4 Summary . . . 20
2.7 Camera Reader Module . . . 20
2.7.1 Camera Array . . . 20
2.7.2 Exposure Synchronization . . . 21
2.7.3 Capture of alternating exposure . . . 23
2.7.4 Determining the exposure values for HDR . . . 24
2.8 Dolphin Producer/Consumer Modules . . . 26
2.9 Frame Synchronization Module . . . 27
2.10 CUDA Uploader and Downloader Modules . . . 29
2.11 CUDA Bayer Converter Module . . . 30
2.12 CUDA HDR Module . . . 31
2.12.1 HDR algorithms . . . 32

2.12.2 Application for our pipeline 32
2.12.3 Choice of algorithm 32
2.13 CUDA Dynamic Stitcher Module 34
2.14 X264 Encoder Module 35
2.15 Visual Results 37
2.16 Performance 40
2.17 Discussion and Future Work 43
2.17.1 Module placement 44
2.17.2 Module separation 44
2.17.3 HDR exposure times 44
2.17.4 Automatic recording system 45
2.17.5 Further scalability 45
2.17.6 DASH streaming - video transcoding 45
2.18 Summary 46

3 Bayer Demosaicking **49**
3.1 Bayer Filter 49
3.2 Motivation 50
3.3 Interpolation and Demosaicking Artifacts 51
3.4 Non-Adaptive Demosaicking Algorithms 53
3.5 Adaptive Demosaicking Algorithms 56
3.6 Chroma Median Filtering 58
3.7 Related Works 59
3.8 Implementations 59
3.8.1 Optimizations considered 60
3.8.2 Memory accesses 61
3.8.3 Multiple passes 62
3.8.4 Kernel design 63
3.8.5 YUV conversion 64
3.9 Evaluation 64
3.9.1 Accuracy of image reconstruction 64
3.9.2 Quality of edge preservation 67
3.9.3 Visual assessment of color spill 67
3.9.4 Algorithm execution performance 69
3.9.5 Kernel evaluation 71
3.9.6 Pixels per thread 71
3.9.7 Median filter 72
3.9.8 Conclusion 73
3.10 Bagadus Scenario 74
3.11 Summary 76

4 Cylindrical Panorama Generation **79**
4.1 Motivation 80
4.2 Cylindrical Projection Algorithm 81
4.3 Projection Algorithm with Corrections 83
4.4 Offline Lookup Table 84

4.5 CPU Stitching Prototype 85
4.6 GPU Stitching Prototype 86
4.7 Dynamic Seam 87
4.7.1 Graph weight calculation 89
4.7.2 Seam fluctuations 90
4.7.3 Seam blending 91
4.8 Implementation 93
4.8.1 Individual image projection 93
4.8.2 Dynamic seam cost creation 94
4.8.3 Dynamic seam path finding 94
4.8.4 Filling in the panorama 94
4.8.5 Blending the seam 95
4.8.6 Summary 95
4.9 Execution Performance 99
4.10 Summary 102

5 Virtual View Video Presentation **105**
5.1 Virtual View 106
5.2 Client Generated Views 107
5.3 Server Generated Streams 109
5.3.1 Static camera streams 109
5.3.2 Dynamic camera streams 111
5.4 Automatic Camera 112
5.5 Summary 114

6 Conclusion **117**
6.1 Summary 117
6.2 Main Contributions 118
6.3 Future work 119

A Accessing the source code **121**

List of Figures

1.1 Example of zooming into 1K quality video 3
1.2 Example showing the quality degradation of debarreling 3
1.3 Wide rectilinear projection 4
1.4 Example showing zooming in a rectilinear panorama 5
1.5 Example of challenging lighting conditions 6

2.1 Distributed recording pipeline 14
2.2 Showing parallax effect for an overlapping region 21
2.3 Mounted camera array 21
2.4 Panorama with independent auto exposure vs pilot camera approach 22
2.5 Asynchronous exposure synchronization 23
2.6 Low and high exposure input for HDR 25
2.7 Regions of interest used for determining auto exposure for HDR 26
2.8 Plot showing bandwidth achieved over Dolphin link 27
2.9 Example of a bayer image and its reconstruction 30
2.10 Lighting challenges at our case study stadium 31
2.11 Sample output from two implemented HDR algorithms 33
2.12 Input and output images for the stitcher module 34
2.13 Comparison between the old Bagadus pipeline and our new pipeline. 38
2.14 Example of zooming into generated panorama 39
2.15 Example of zooming into raw camera stream 39
2.16 Overall pipeline performance, without HDR 41
2.17 Overall pipeline performance, with HDR 42
2.18 GPU single execution 43

3.1 Example of a Bayer pattern 49
3.2 Conversion matrix from RGB to YUV 50
3.3 YUY2/YUYV byte representation 50
3.4 Loss of detail from Bayer demosaicking 52
3.5 Demosaicked images suffering from zippering artifacts 52
3.6 Demosaicked images suffering from color spill 53
3.7 Bayer Pattern centered on a red pixel 53
3.8 Visualization of 4-pixel debayer kernel 63
3.9 Lighthouse image used for reconstruction comparison 65
3.10 PSNR comparison of multiple demosaicking algorithms 66
3.11 Demosaicking edge test 68
3.12 Comparison of color spill after demosaicking 69

3.13 Performance comparison of multiple demosaicking algorithms 70
3.14 Performance evaluation of two kernel implementations 72
3.15 Comparison of real bayer images 75

4.1 A cylindrical texture re-projected into rectilinear panorama 80
4.2 Example of individual images projected and stitched into a panorama 81
4.3 Creating panorama pixels from a captured image 82
4.4 OpenCV-based program for creating offline panorama 85
4.5 Comparison of interpolation method for stitching 87
4.6 Showing how a dynamic seam avoids slicing through players 88
4.7 Dynamic seam graph 88
4.8 Visualizing the cost components used for dynamic stitching 90
4.9 Showing the effect of seam blending 92
4.10 Ghosting introduced by feathering 93
4.11 Initial individual image projection of Bagadus stitch 95
4.12 Individual projections of Bagadus stitch overlapped 96
4.13 Visualization of the cost of dynamic seam, for the Bagadus stitch 96
4.14 Visualizing the dynamic seam, for the Bagadus stitch 97
4.15 Final stitched panorama, before blending 97
4.16 Final panorama, after blending 98
4.17 Highlight of the result of blending 98
4.18 Performance of each stitching step 99
4.19 Profiling timeline of each stitching step 100
4.20 GPU utilization for stitching kernels 101

5.1 Panorama video with shown ROI and virtual view 105
5.2 The intersection of the ray from the virtual view with the unit cylinder 106
5.3 Core execution times for various resolutions 107
5.4 Virtual view demonstration presented at ACM MMSys 2014 108
5.5 A plot showing the time required to download panorama video segments 109
5.6 HTML5 interface for expert annotated events 110
5.7 Block diagram of OpenGL streaming delivery pipeline 111
5.8 Schmitt and adaptive trigger plot 113
5.9 Preference distribution of tracking models from automatic camera user study . 114
5.10 Preference distribution of man vs machine, from automatic camera user study . 115

List of Tables

2.1 Hardware used for the processing machine in our prototype system 16
2.2 CUDA memory types 18
2.3 Execution performance of the HDR implementations 33
2.4 Summary of drop rates at multiple frame rates 40

3.1 Resource requirements of bayer demosaicking algorithms 71
3.2 Evaluating the effect of the median filter 72
3.3 Summary of demosaicking algorithm's visual performance 73

4.1 Time measurement of trigonometric functions 84

Acknowledgements

I want to thank my supervisors Pål Halvorsen, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland and Carsten Gridwodz who have helped me greatly by providing discussions, feedback, guidance and short-notice paper deadlines to adhere to. I especially want to acknowledge Vamsidhar, for his great work on the virtual viewer, which became the motivation for my work. I also wish to thank Kai-Even Nilsen for helping us with the on-site installation of our prototype.

Additionally, the other students working on the Bagadus project have been great to work with, and I wish to acknowledge their high quality work. Specifically, I wish to thank Sigurd Ljødal for his cooperation in the design and implementation of our prototype system. Asgeir Mortensen, for a continuous stream of random facts and statements, along with the design and 3D printing of our camera mount, which simplified my task. I also wish to acknowledge Lorenz Kellerer for his great work on HDR, allowing us a smooth integration with Bagadus.

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

Oslo, April 30, 2014
Ragnar Langseth

Chapter 1

Introduction

1.1 Background

In recent years, we have seen a major growth in various multimedia solutions. The increased bandwidth capabilities of the Internet today allows for new applications that provide high quality video, streamed directly to the user. Online solutions are becoming preferable to more traditional viewing platforms, e.g., television, because it offers more individualized and interactive solutions. Users can now typically navigate between multiple video feeds, choose content and play on-demand video. However, every choice that the user is given means that more content must be created at some point. This means that there is a great need for scalable solutions for content production and distribution. The new content can be produced by adding more camera feeds, different viewing angles, parallel recordings or manual annotations and commentaries. These are expensive solutions in terms of manpower and hardware requirements, i.e., very difficult goals to achieve for smaller systems.

The overall cost of creating this content is too daunting for many scenarios, which is why more automatic solutions are beginning to appear. By automating multiple steps in the recording and distribution stages, cheaper and more scalable systems can be created. Here, there are many possible approaches that can be explored [1–8]. Robotic camera control, such as [6], can eliminate the manpower requirements for the recording of video, both in concrete camera control and production. Robotic cameras are, however, still limiting the number of concrete camera streams possible and do not provide an individualized viewing experience for each user.

Some systems have begun to find methods of providing interactive solutions that do not require additional recording content. By manipulating a finite set of data, some systems can construct a personalized viewing experience based on user input. Many approaches use the large amount of information contained in a full panorama video, covering all areas of interest. This way, the system can allow users to focus on the region that they deem most interesting, generated from the same recorded video. This is of particular interest when looking at arena sports, as the areas of interest are all confined in a localized physical stadium that can be covered in a single panoramic view. For example, Mavlankar *et al.* [3] describe a pan-tilt-zoom streaming solution for soccer, allowing the user to view cropped and zoomed parts of the panorama. The user can then freely pan, tilt and zoom the camera into the pre-generated panorama video, creating an interactive viewing experience. Additionally, such systems can also remove the manual requirement for camera control during the recording, as the panorama video is generated from statically placed cameras and the user personally controls the virtual camera.

In order to provide a detailed zoomed view to the user, giving a visually pleasing viewing experience, the quality of the initial panorama video must be as high as possible. High quality streaming is becoming more common due to the increased availability of high bandwidth. Video with 1K resolution is becoming the norm for streaming purposes and some services have begun to offer 2K, and even 4K [9], resolution video streams. Panorama video with the same level of quality typically requires significantly higher resolutions of 4K or even 8K video. Many systems choose to scale down the panorama video before transmitting this over the network, but this sacrifices the resolution available for zooming purposes. Another alternative is to deliver smaller, cropped views from the video server, transmitting only the regions that the user is interested in. This solution, however, can be challenging to scale as well due to the required per-client processing on the server.

1.2 Bagadus System

The Bagadus system [10–15] is a system that can generate a real-time high resolution panorama of an entire stadium, focusing on soccer analysis. This systems was created to allow coaches quick access to recorded video events, and scalable video delivery systems are beyond its initial scope. Also, because it has been intended as an analytical tool, its focus has not been creating a visually pleasing video that can be used for entertainment purposes.

Bagadus was presented in [15] as an integrated system for sports analysis and automatic video extraction. Analysis data, in the form of video and player tracking, could be recorded, processed and stored for easy extraction. The primary goal was to create a tool for providing a user with the relevant information, primarily in the form of video, from pre-recorded events. The system focuses on automatizing the data extraction, providing the user with quick access to video, tracking information and various player statistics. The system was designed with any arena sport in mind, using soccer as an example scenario.

The system was further extended in [11] to include a panorama pipeline, stitching together multiple camera feeds to provide a full rectilinear panorama. This was further improved upon, and presented as a real-time system in [12]. We will not include a complete description of the Bagadus system, but we refer to [11, 12, 15] for full details. Now, we will present the primary limitations of this system, as it is presented in [12], from now on referred to as the *old Bagadus system*.

While high quality is not essential for a sports analysis tool, we recognize that many details can be lost due to low quality video. If used for entertainment purposes, quality will obviously have a significant influence on the viewers perception of the system. Due to the static camera setup, some situations, e.g., far side of the field, may require zooming to provide proper coverage.

The visual quality of the single camera streams in the old Bagadus system was primarily limited by the hardware. The cameras used [16] delivered a resolution of 1294×964 pixels, providing a good visual overview of the soccer field. However, when zooming into this video, the low resolution quickly became a visual annoyance, as can be observed in figure 1.1. The raw video provided by these cameras also contain significant barrel distortion, primarily due to the fairly cheap wide angle lenses used. Due to the straight lines of a soccer pitch, the barrelling effect is especially visible in this scenario. It is also not possible to apply regular stitching

(a) Zoom with nearest neighbour interpolation

(b) Zoom with lanczos interpolation

Figure 1.1: Example from the old Bagadus system, where we performed 300% digital zoom into 1K quality video

algorithms on images suffering from barrel distortion, as this effect is expected removed beforehand. In order to utilize the frames in the live recording, they needed to be debarreled. This is a lossy operation that degraded the quality, primarily in each of the four corners. Due to real-time requirements, the old system was unable to perform pixel interpolation, deferring to nearest neighbour, which causes aliasing of edges and jagged objects. In figure 1.2 we show the difference in quality, when the same scene is captured at the corner of the image, as opposed to the center.

The Bagadus system has so far primarily utilized the single camera streams, as they provide better individual coverage of each area than the panorama video. The implementation of the

(a) Scene cropped from the center of the image

(b) Scene cropped from the corner of the image

Figure 1.2: Example of the same scene, captured by two different cameras, after debarreling. Note the increased blur and jagged edges in 1.2b as a result of warping and interpolation

panorama pipeline proves that a high quality panorama can be generated in real-time, but it has yet to be fully utilized in the Bagadus system as an analytical tool. The panorama gives a good visual overview, but is hard to manipulate in post processing. Its incredibly wide resolution of 6700×960 makes it challenging to see any details when viewed in its entirety. It also makes it very challenging to transfer in real-time over networks for streaming purposes. Scaling down the video is a possibility, but this will further reduce the already low vertical resolution. Another possibility is to utilize the pan-tilt-zoom streaming solution presented by Mavlankar *et al.* in [3]. However, there are several issues with this approach when applied to the Bagadus system. We have already established that the original video quality, from each individual camera stream, is of fairly low quality and varies greatly depending on the region of interest.

The homography stitch presented in [11] produced a rectilinear projection, ensuring that all lines remain straight. Such a projection becomes extremely distorted when the field of view is

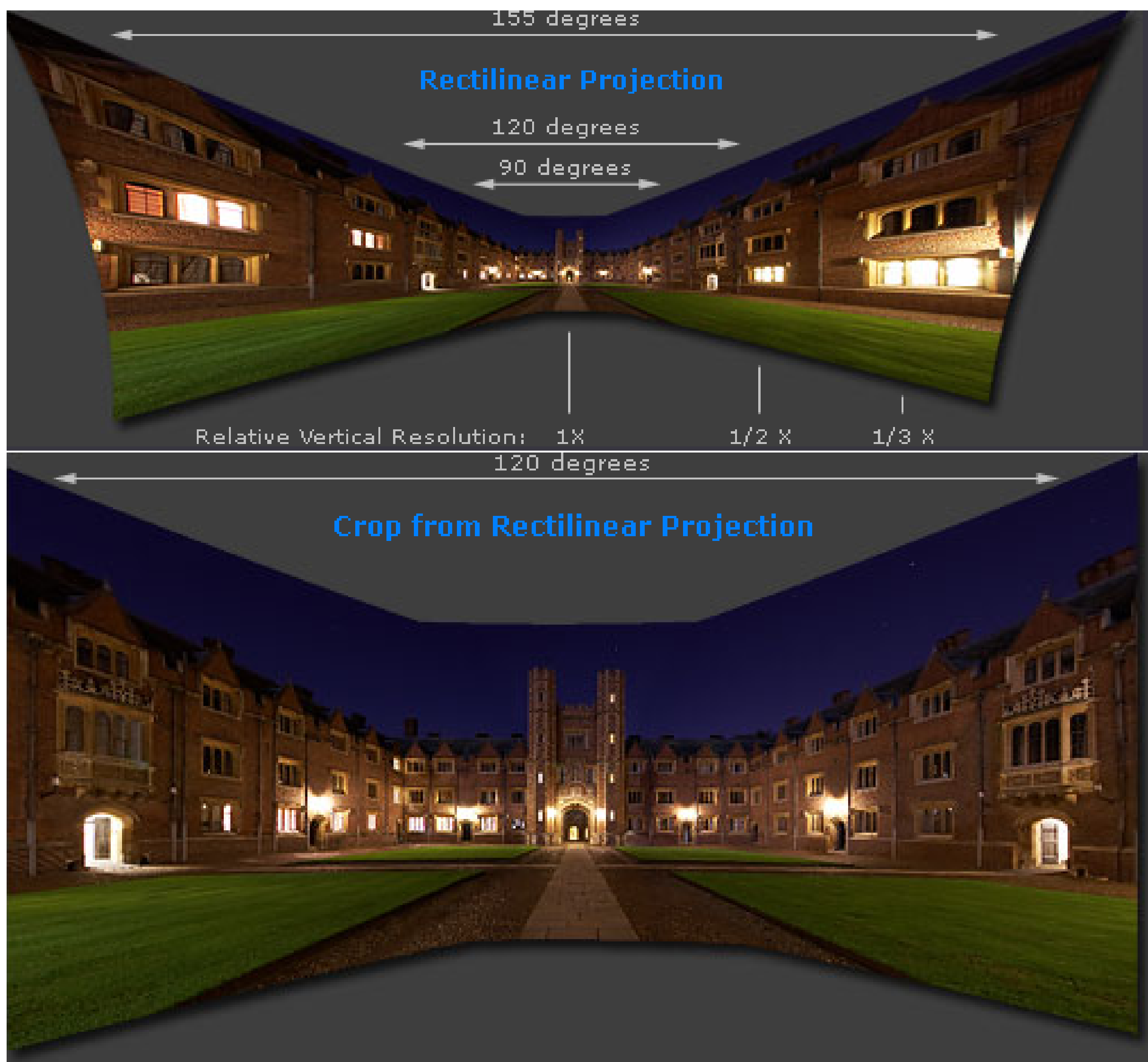


Figure 1.3: Figure showing how a rectilinear panorama expands pixels as the horizontal field of view increases. Note that the Bagadus system requires a field of view of 160-170°, greater than displayed in this figure. Figure from [17].

(a) Zooming into the center of the pitch

(b) Zooming into goal area

Figure 1.4: An example of zooming by 200% in a rectilinear panorama (lanczos interpolation). Note that in the original raw camera streams, these areas were of virtually the same quality, i.e., nearly unaffected by barrel distortion.

greater than approximately 120° [18]. The projection attempts to maintain a perspective view, causing pixels on each side of the panorama to be expanded. This is shown clearly in figure 1.3. We can see that the required horizontal resolution expands significantly as the field of view increases, and pixels on each side are bloated. This takes up needless space and reduces the relative vertical resolution, which in turn makes it difficult to see details in the center of the image.

In the Bagadus system, the low relative vertical resolution meant that it became very challenging to see players moving on the opposite side of the field. The panorama from the old Bagadus system covered nearly 170°, significantly higher than that of figure 1.3. The four images used to create the panorama were 1280 pixels wide, with a combined overlap of approximately 100°, and yet the panorama required a full 6700 pixels. This shows that the pixels were artificially expanded, and that the old Bagadus panorama potentially suffered in quality from using a rectilinear panorama. The loss of quality due to distortion was further amplified by the need for nearest neighbour interpolation in the live recording pipeline in order to stay within real-time requirements. The effect of the reduced image quality, due to warping and nearest neighbour interpolation, can be seen in figure 1.4, where we zoom into different regions on the panorama. We see that the goal area, in the left side of the panorama, appears coarse and jagged, as all edges produce strong aliasing artifacts.

The old Bagadus system also suffered significantly during challenging lighting conditions. As can be seen in figure 1.5, the strong sun covering half the field forced the cameras to either overexpose the sunny regions, or underexpose the shaded regions. The stitched panorama suffered especially from this, as the color correction module was unable to adapt to the significant changes in exposure on each of the camera streams. This can also be observed in figure 1.5, as the two center cameras each selected widely different exposure settings, and the color correction module was unable to adjust this.

One of the goals of the Bagadus system has always been the use of inexpensive hardware.

Figure 1.5: An example of challenging lighting conditions, where a large portion of the field is underexposed

The old Bagadus system was able to record and process the panorama on a single computer, utilizing high-end off-the-shelf hardware, along with all four individual camera streams. This put significant strain on the CPU and the system proved challenging to scale further, e.g., accommodate higher resolution, higher frame rate or more cameras. There are several key optimizations, e.g., moving more modules onto the GPU, that could be performed in order to improve the pipeline performance, but this is still a very limited degree of scalability. In order for the system to scale significantly we would need the ability to distribute the workload to multiple machines.

1.3 Problem Definition

Multiple camera views is vital when providing video from an automatic recording system for arena sports. A manual camera operator can provide a single camera stream, targeted at the most likely region of interest. This requires manpower, and makes it impossible for later viewing of alternative regions of interest. Several recorded camera streams provide a limited number of options, but have additional space and management requirements. Robotic cameras can remove some of the manual requirements, as well as potentially reducing the number of required cameras, but are limited to a single video stream per camera. A panorama video can contain all the required information in a single video stream, providing a full view of the entire arena.

Viewing a single large panorama can be difficult, however, both due to the incredibly large field of view, making details difficult to discern, as well as the large file size. The Bagadus system [10–12] has yet to fully integrate the panoramic view as an analytical or recreational tool.

In this thesis, we will research and develop a new method of zooming and panning into a panorama video, creating an individualized virtual view. We wish to explore how an alternative panoramic view, a cylindrical panorama, can be used as an intermediary representation before video presentation. By projecting the cylindrical texture back onto a planar surface, we can create multiple virtual views from the panorama video, allowing the user to zoom, pan and tilt the virtual camera. A virtual camera can be used in multiple ways, allowing for several video extraction systems to coincide. We must determine how these systems can be designed and implemented, as well as the benefit each system has. Requirements such as scalability, user

interaction and visual quality are key issues that must be addressed, something we will begin to scratch the surface of in this thesis.

The fundamental principle of these virtual view presentation systems is that they are all designed around the same cylindrical panorama, emphasising the need for visual quality in the initial recording. The primary goal of this thesis is therefore to create a single multi-purpose high resolution cylindrical panorama that can be delivered to multiple users through live streaming solutions or post-recording video extraction. The initial recording must be done in real-time, storing an efficiently encoded panorama video that is optimal for video streaming and post-processing. The panoramas visual quality must be high and clear of visual artifacts to ensure user satisfaction and accommodate the zooming associated with the virtual view. We will investigate several key methods to increase the visual quality of the panorama video and evaluate which design and implementation decisions will need to be made in order to preserve as much of the original camera quality, minimizing distortion and visual artifacts. We will use the previous Bagadus panorama pipeline for comparison, considering this a re-design and upgrade of an existing system. This includes the addition of new cameras and hardware, which we must determine how to best utilize, in order to create the best possible quality within the given time constraints. With the addition of new hardware, a new evaluation is required to determine what complex modules from the old Bagadus panorama pipeline can be bypassed in order to create a more lightweight system, while at the same time provide a higher quality final result. This also means increasing the frame rate of the panorama pipeline to allow for smoother playback, which introduces new challenges. However, increasing the frame rate can also open up the possibility of adapting the video stream to difficult lighting conditions, utilizing the concept of high dynamic range video.

The Bagadus system shows that it is possible to create a real-time panorama pipeline on a single computer, but we wish to create a more scalable system. This means both a scalable recording pipeline and investigating the possibilities of scalable video extraction. We will be exploring how the recording pipeline can be extended through the use of distributed processing, with modules executing on multiple machines as a part of the same system. As the system scales, new problems arise that must be addressed. We will need to look into methods of dealing with the increased data rate and design algorithms for a system that can be extended beyond our intended prototype. The high data rate also has a direct effect on how we can read the original camera stream, requiring an algorithm for reconstructing the initial raw video frames into usable full-range color images. Therefore, we investigate how to best implement such an algorithm, given our target system.

Because of the strict real-time requirements of the presented problem, we need to provide an efficient implementation. We have seen how the GPU architecture can provide an incredible performance boost to video processing, and wish to explore this concept further by evaluating and optimizing algorithms for the GPU architecture, using the CUDA framework.

1.4 Limitations

In this thesis, we will not be giving a full background description of the original Bagadus system, as this has been done sufficiently in [11, 12, 15]. We consider this thesis to be a natural continuation on the Bagadus panorama pipeline and as a result we will not provide many comparisons between CPU and GPU implementations, as it has already been proven that a high

resolution panorama pipeline can be generated in real-time. As a result of this, most algorithms have been implemented exclusively on either CPU or GPU, only highlighting the specific platform dependent optimizations utilized where applicable.

Due to the long distance from our installed prototype and expense of creating a local replica, we have been forced to work on limited hardware. Therefore, our implementations, optimizations and performance evaluations will generally be directed at a specific set of hardware components.

1.5 Research Method

In this thesis, we will be following the *design* methodology described by ACM Task Force in *Computing As a Discipline* [19]. This involves the design, implementation and evaluation of a prototype panorama pipeline, as a natural continuation of an existing system, as well as the integration of various post-processing methods. The recording prototype is deployed in a real life scenario, and several video extraction systems have been implemented and tested on real footage. The use of a real prototype allows us to see all the practical problems associated with a full deployment. It allows us to verify, to a greater degree, the effectiveness of our proposed solutions, as well as affirm the realizability of the designed system.

1.6 Main Contributions

In this thesis, we show the versatility of a cylindrical panorama when used as an intermediary representation. We present the ability to provide users with personalized or static virtual views, all created from the same panorama video. These can be delivered to the user through multiple video extraction systems. We present a few such systems in this thesis, and we attempt to show that there are several unexplored areas in this field, opening the doors for further research into virtual view based systems.

One of the main contributions of this thesis is the implementation and installation of a high resolution panorama pipeline installed at Alfheim Stadium in Tromsø. We show how the Bagadus pipeline can be improved by scaling the system to use more cameras, with higher resolution and provide a more visually pleasing panorama output. The resulting pipeline can process a larger panorama at twice the frame rate, with additional quality improvements. For example, we have investigated how high dynamic range images (HDR) can improve the visual quality during difficult lighting conditions, at the cost of reduced frame rate. The system utilizes several key concepts of distributed processing, as we coordinate the recording of multiple camera streams across multiple machines, using low latency communication technologies.

We investigate how the current common Bayer demosaicking algorithms can be implemented on a modern GPU, determining which algorithms are best suited for this architecture given a high resolution real-time video system. Alternative comparisons rarely include the GPU architecture when analyzing the performance or complexity of various algorithms. We provide an evaluation on their visual output, complexity of implementation and computational efficiency on a modern GPU.

During the work on this thesis, we have published or submitted several research papers.

Bagadus: An Integrated Real-Time System for Soccer Analytics [10] Here we implemented an initial video extraction system for the Bagadus system to, providing a simple web interface for event-based video extraction.

Interactive Zoom and Panning from Live Panoramic Video [13] In this paper we present an earlier CPU-based panorama pipeline prototype, along with a video presentation system for client-based live virtual view generation.

Be Your Own Cameraman: Real-Time Support for Zooming and Panning into Stored and Live Panoramic Video [20] For this paper, we provided a demonstration of the above system, allowing participants to actively zoom and pan into our panorama video. The video was either generated through a live on-site feed with the new GPU-based panorama pipeline, showing the portability of our system, or pre-recorded data from our soccer stadium prototype.

Soccer Video and Player Position Dataset [21] Here we present a soccer dataset, based on tracking information and recorded video. The video was recorded with our CPU prototype pipeline, as well as an earlier single camera pipeline.

The Cameraman Operating My Virtual Camera Is Artificial: Can The Machine Be As Good As A Human? [22] In this paper, we present an evaluation of several implemented methods of automatic camera algorithms, for operating the virtual view.

An evaluation of debayering algorithms on GPU for real-time panoramic video recording [23] Here, we show the results from our work on efficient debayering algorithms on GPU, as used by the proposed panorama pipeline.

Real-Time HDR Panorama Video [24] This paper details the implementation of the high dynamic range video, as a part of our panorama pipeline. We show the result of several implemented algorithms, and evalute their visual and computational performance viability in our system.

Using a Commodity Hardware Video Encoder for Interactive Video Streaming [25] This paper details a video presentation system for low-latency server-based live virtual view generation. This focuses on fast re-encoding of the virtual view, before presentation.

1.7 Outline

We begin this thesis by presenting our new real-time recording pipeline in chapter 2, explaining the overarching design of the system and each of its individual modules. This chapter will cover the entire creation process of the panorama, from fetching raw video from the cameras to encoding the finished panorama video. In chapter 3, we will take a look into the Bayer demosaicking process, a step required in our presented recording pipeline. We will present multiple alternative algorithms before describing how they have been implemented. Finally, we evaluate their performance, both visually and through quantifiable measurements, to determine the

best fitting algorithm for our scenario. In chapter 4, we describe in further details the stitching operation, how each image is projected and stitched in an offline capacity to create a lookup table for fast real-time stitching. We will also present the continued work towards improving the stitching module in the old Bagadus system. The new proposed method of creating virtual views, from the same original panorama, is detailed in chapter 5. We will explain how this method can re-project the same panorama input file into an infinite number of virtual views, creating a unique user experience. Here, we also detail multiple ways to utilize the panorama video, allowing for several video extraction system to coincide. These include live streaming, either personalized views or pre-determined static viewpoints, or post-game extraction based on complex SQL-queries. Finally, we will conclude this thesis in chapter 6, where we summarize our findings and discuss future work.

Chapter 2

Recording Pipeline

In this chapter, we will present a new real-time panorama recording pipeline. First, we will repeat and discuss the motivation for re-designing this pipeline. We then go on to explain the design of the system. Next, we present some general information on programming with CUDA, as multiple modules utilize this framework. Then, we go on to present each of the modules active in the pipeline, and their primary function. Next, we present some evaluation on the visual result of the panorama and performance of the system. Finally, we provide some discussion on alternative design choices and future work.

2.1 Motivation

In chapter 1, we discussed a growing new trend of panorama video. More and more researchers are beginning to exploit the large amount of information that can be contained within a single panoramic image. This can make it a great intermediary representation, allowing for region-of-interest based viewing that can be constructed individually for each viewer. We wish to create a high resolution panorama, that facilitates easy extraction of several concurrent virtual camera views. While creating a visually pleasing panoramic view is among our goals, we are primarily interested in creating an ideal intermediate representation. We see the limitations of a panoramic view, that its sheer size limits the users ability to discern details, and wish to focus on achieving a high visual quality upon video presentation of a limited region of interest.

A rectilinear panorama, as presented in the old Bagadus system [12], has several flaws that limits its suitability for virtual view extraction. We saw in chapter 1 that this pipeline produced a panorama with very varying levels of quality. We also saw that target scene, at least based on the current camera placement, ha a field of view far greater than what is commonly recommended. A cylindrical panorama, as described earlier, can cover the same field of view with far less warping of the original pixels. This means that more of the original image quality is preserved. The resulting cylindrical panorama is smaller and easier to transport over network, but still contains all the required visual data.

The old panorama pipeline was quality bound by the original 1K cameras. We wish to scale the system further, by introducing new cameras. This gives us more data to process and new challenges to tackle, but can allow for a better visual result. The old system was not able to easily scale to a higher data rate, as the system was already hard pressed to stay within real-time boundaries. Additionally, the change of hardware and physical configuration changes the system requirements, as new modules are needed while others may become obsolete. Therefore,

we opted for a full system redesign. However, there will still be many elements transferred over, as we can design and implement a new system based on the knowledge gained from the old Bagadus panorama pipeline.

2.2 Related Work

Panorama video has become increasingly popular, as our ability to process and present higher resolution videos have improved. As a result of this, we are starting to see several systems appear that focus on delivering panorama video.

Qamira [26] and K2 Panoramic Video [27] are two commercial systems that focus on real-time panorama video of soccer. K2 Panoramic Video uses a small camera array to record the full panorama, along with manual meta information, which can be used in post-processing for video analysis. Qamira uses an ultra-high definition panoramic camera, comprised of 16 individual cameras, to give a full view of the soccer pitch. This system uses expensive hardware, with dedicated servers for real-time stitching, far beyond the price range of the Bagadus system. Both of these systems also produce a rectilinear panorama, which only allows for crop-based virtual views. Another such high-cost system is described in [28], where a large 60K panorama is recorded and displayed on a wall of tiled screens.

Adam [29] presents a system for stitching a stereo view in real-time, using a GPU-accelerated implementation. It can create the panorama at 37 fps, but is limited to two 1600×1200 pixels input images. This system also focuses heavily on the challenge of objects close to the cameras, and avoiding the associated parallax effect [30]. In the Bagadus scenario the closest objects are several meters away, and we have seen that players appear small enough that parallax errors can be avoided through a dynamic seam [31].

We also see several portable panorama recording system, intended to recreate a feeling of immersion, without a fixed scene. GeoView [32] is an intelligence system that records a 360° spherical panorama at up to 10 frames per second, but this is far below our frame rate and captures an unnecessarily large field of view. This also excludes the use of other omnidirectional cameras, capable of capturing a full 360° view, which are typically also fairly expensive. FlyCam [33] is another mobile system, intended for classroom lectures, video conferencing and other indoor environments. This is a cheap and mobile system, but does not focus on video quality. Additionally, our Bagadus scenario is intended for a stadium, i.e., static, setting.

In summary, the current available panorama video recording systems struggle with producing the level of quality required for virtual cameras, at an acceptable frame rate. Several commercial expensive systems can produce high quality results, but beyond the price range of the Bagadus system. Most of these systems also produce rectilinear panoramas, while we wish to explore the efficiency of a cylindrical panorama for post-processing and virtual view generation.

2.3 Pipeline Features

The implemented prototype pipeline can be configured in multiple ways, but we should provide the basic configuration it is designed for. We create a final panorama of 4096×1680 pixels from 5 statically placed cameras. The output is encoded as short H.264 video segments that can be easily accessed by an HTTP server in order to support live streaming possibilities. The

maximum allowed frame rate, determined by the camera model, is 50 frames per second, which is our intended default rate. This can be scaled down at runtime if desired, for example due to file size restraints. This in turn means that most modules of the pipeline will have a deadline of 20ms per frame.

In order to combat situations with strong sun, creating a strong contrast between the sunny field and shaded areas, the pipeline can produce high dynamic range video by toggling *HDR mode*. This limits the output frame rate to 25 frames per second, but may significantly improve the visual result in some weather conditions. We will discuss this mode later in this chapter, particularly in section 2.12.

2.4 System Design

The entire recording pipeline has been partitioned into smaller modules, all performing a subset of tasks. These are performed once for every video frame, in a pipelined fashion. This means that for every iteration of the pipeline, every module works on a single input frame and produces a single output frame. As long as all parallel steps in the pipeline deliver their output within the real-time deadline, e.g., every 20ms, the overall system remains real-time. In order to create a distributed and scalable system, these modules can execute on separate machines. Our recording pipeline consists of multiple recording machines, three in its current configuration, and a single processing machine to combine all the input streams into a single panorama output.

When a new frame is initially created, a header is included to contain some frame specific meta information, such as recording timestamp and exposure setting. This can be modified or appended to later in the pipeline, as long as we ensure the accuracy of the meta information for the modules that depend on it.

2.4.1 Modules

In the previous recording pipeline [12], we found the modules to be too tightly coupled, making modifications difficult. We therefore tried to create a simple shared interface between each module, ensuring that we could easily combine, remove and replace modules at will. We decided on a simple pull-based approach, where each module will request a new input from its producer-module as frequently as possible. Additionally, the module must also be able to serve the next module, which will request a new input in the same fashion. In this sense, each module, except for the first and last, behaves as both a producer and a consumer.

This interface makes it very easy to combine different modules and simplifies testing significantly, as each module can be tested separately. Having a simple integration process and ability to test is essential when creating a robust system. It also simplifies the configuration part when deploying the system in new environments or skipping some modules for different scenarios. The interface is, however, very abstract and simplistic when in comes to meta information and multiple input sources. It is designed to deliver a single "resource", defined as a continuous block of memory with some meta information attached. Therefore, the primary constraint when combining and exchanging modules is that they agree on the binary representation of the data, i.e. specific pixel format or memory layout. Additional information, such as masks, must be placed within the same memory space. Although this is a constraint, we have found it easy to work around, as nearly all data is statically sized and based on the resolution of the input or the

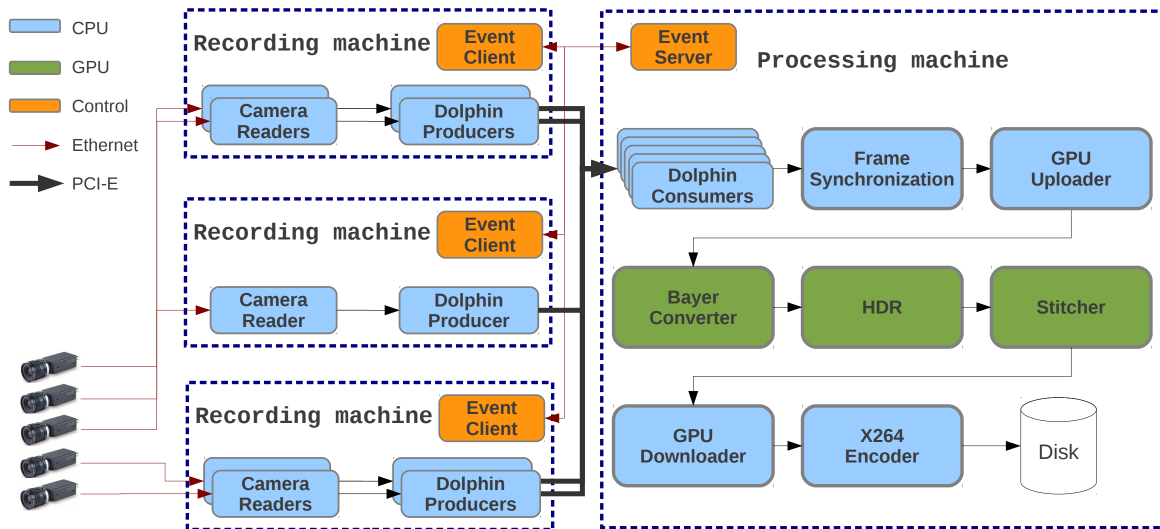


Figure 2.1: Distributed recording pipeline

output data. This can be accessed by static offsets within the buffer, which do not change within the programs execution.

In order to utilize the power of the GPU architecture, several modules deliver and receive data in GPU device memory. For these modules we have a simple wrapper, where the meta information is delivered in CPU memory along with a pointer to GPU device memory. Additionally, some modules expect the data to arrive in a regular one-dimensional array, similar to what we know from CPU-based memory, while others expect a special two-dimensional `cudaArray` (see section 2.6) which is designed for a 2D memory layout. The module interface requires all modules to be able to deliver both options, but in reality only one is used. The format of the output is decided based on the preference of the proceeding module. The alternate format is typically implemented through a memcopy operation, as this is only useful for compatibility and debug testing.

2.4.2 Active modules

The panorama pipeline has ten modules, shown in figure 2.1.

Camera Readers: Reads the raw video frames from its specified camera, and synchronize exposure configuration with the other camera readers.

Dolphin Producers/Consumers: Transfers the raw video frames over PCI Express (PCIe) from one of the recording machines to the processing machine, handling each camera stream separately.

Frame Synchronization: Joins individual, asynchronously captured, video frames from each camera stream into full sets of frames. This module also ensures that the system remains real-time if a camera fails to deliver a frame on time.

GPU Uploader: Uploads the video frames asynchronously onto the GPU, for faster processing.

Bayer Converter: Converts the input frames from their original downsampled pixelformat, the Bayer format captured by the camera, by interpolating the missing color values.

HDR: Combines two frames, captured with different exposure times, into a single high dynamic range image. This is an optional module, intended for sunny days with strong shadows, as it effectively halves the frame rate.

Stitcher: Performs a cylindrical projection of all frames, and stitches these together into a full cylindrical panorama.

GPU Downloader: Downloads the video frames back into CPU memory, in preparation for the last module.

X264 Encoder: Encodes the individual video frames into H.264 playable video and stores these to disk.

2.4.3 Internal module design

Each module is internally responsible for buffer management and thread communication. Output buffers are typically created by the following module, using a normal 2-frame rotating buffer, which is passed along when requesting new input data. Each consumer-module will create its own thread for requesting new input data, continually calling its producer-module immediately until the end of input. The previous module will generally utilize this thread as its primary thread of execution, but may also create additional computation threads. Each module is internally responsible for ensuring that the previous output has been delivered before requesting another input frame, beyond the current processing. Most modules implement this through a barrier, shared between the consumer thread and the producer function call.

2.5 Distributed Recording - Machine Setup

We can see in figure 2.1 that the panorama pipeline is distributed across a total of four machines. We wished to design a system that allows modules to execute on separate machines, indistinguishable to the surrounding modules. We believe this can be possible for our system because of the limited control code required to execute the system. Most control signals propagate through the modules as a part of the header information, meaning that nearly all live communication is performed on the first module of the pipeline. Because of this, the system has very little inter-module communication and system control outside the modules themselves. This makes it easier to distribute the pipeline.

By distributing the workload, or even just facilitate the possibility, the system can also easily scale to a higher number of cameras than our current prototype. We established early that the video camera driver is highly interrupt-driven, and requires quick context switching between threads. When surpassing 3-4 cameras, this significantly affects overall processing performance and we begin to see frame drops as a result. Therefore, in order to provide a truly scalable solution, we distribute the system across multiple machines, raising several important challenges, primarily related to data transfer, communication and synchronization.

CPU	Intel Core i7-3930k @ 4.4 GHz
GPU	Nvidia GeForce GTX 680
RAM	32 GB DDR3 @ 1866 MHz
HDD	Samsung SSD 840 Series, 500GB
IX	Dolphin IXS610 PCIe Host Adapter

Table 2.1: Hardware used for the processing machine in our prototype system

2.5.1 Current machine setup

The system currently uses three recording machines for handling five individual camera streams, and a single processing machine for stitching the five sources into a single panorama, and encoding it. For now, we have only offloaded the initial capture of the data, but in the future this functionality can be extended to the video processing as well, e.g., offload video encoding or even GPU-based modules onto different machines. One challenge is that it simplifies synchronization if we have all input sources present on a single machine at any given time. This means that the input bandwidth to this machine may become the system bottleneck as a result. This setup can be easily modified to include additional machines, or swap any of the existing.

Table 2.1 shows the specifications for the processing machine used in our prototype installation. The recording machines use Intel Core i7-2600 CPUs at 3.40GHz, 8GB DDR3 RAM and no GPU. The high speed interconnect transfer network uses a Dolphin IXS600 8-port PCIe Gen2 switch, with adapters for each machine in the cluster [34]. The cameras used are Basler acA2000-50gc [35], with 8mm AZURE-0814M5M lenses [36].

2.5.2 Communication

Each of the recording machines will connect to the single processing machine, either for a specific recording or as continually running daemon applications. When a new recording begins, some initial configuration variables are transmitted to each of the recording machines. The recording machine must know its own local configuration, e.g., camera addresses, as well as the global per-session state information, e.g., recording frame rate or exposure information. When the system is operating in daemon mode, the modules do not fully terminate. We have little control over the camera driver, whose context we terminate between recordings and re-initialize once we begin streaming.

All communication goes through a single machine, which currently is the processing machine, that broadcasts all information to each recording machine using LibEvent [37] over normal TCP connections. We typically use one-way signal based control, where the processing machine sends specified control packets to signify certain events, such as start/stop recording or change live configuration parameters. A more frequent event is exposure synchronization, which is performed a few times per minute to adapt to the ever-changing lighting conditions of an outdoor setting. An event is then generated on the recording machine that initiates the synchronization, which is transmitted to the processing machine where it is broadcast to all recording machines. More information on this step in section 2.7.2.

Additionally, we need a medium for transferring the actual raw video frames from the recording machines onto the processing machine. With five 2K resolution streams, this has significant bandwidth requirements, as the combined bandwidth of all video streams at 50 frames

per second is $2040 \times 1080 Bytes \times 50 fps \times 5 = 525 MB/s$. This is approximately four times higher than regular gigabit ethernet. We transfer the data using Dolphin interconnect solutions and their SISCI-API [38]. This uses a PCIe based high-speed interconnect switch, forming a cluster of connected machines. Machines of the same cluster can then perform direct memory accesses to read/write data at very low latency. More on this in section 2.8.

2.6 Nvidia CUDA

In the old Bagadus system, we saw that it is possible to achieve better performance by utilizing the GPU architecture [12]. We will therefore primarily be looking at GPU implementations, using the CUDA framework, in this thesis.

The Compute Unified Device Architecture (*CUDA*) [39] is a parallel computing platform developed and created by Nvidia to allow easier programming of the GPU. It is supported by all modern Nvidia GPUs and is designed to work across multiple architectures. Some functionality has been added through the years that isn't supported by all devices, and Nvidia generalizes this into *compute* classifications. In this thesis, we will focus only on devices with compute capabilities 2.1 or higher, and all performance evaluations and optimizations will be targeted at devices with compute 3.0, i.e., the Kepler architecture [40].

The low cost of the modern graphics processors, compared to their computational power, has made the architecture attractive for alternative use, besides graphics rendering. CUDA is heavily focused on *general purpose computing*, allowing the use of common programming languages, e.g., c++, to create GPU accelerated applications. This often makes it very easy to port existing CPU applications to CUDA.

There are of course also many image processing optimizations available though the CUDA framework. Along with the speed, simplicity and low hardware cost, this makes CUDA well suited for our recording pipeline. As we will discuss performance and implementation challenges for several modules in this thesis, we will now give a brief overview of the terms, benefits and challenges associated with CUDA programming. For a more in-depth explanation, see [41].

2.6.1 Programming using CUDA

Functions executing on the device, GPU, using CUDA are called *kernels*. When a kernel is launched, one must specify the number of parallel threads that should be created, allowing the same kernel to be executed by hundreds or thousands of threads in parallel. These threads are considerably more lightweight than CPU threads, designed for extreme parallelism and fast context switching. CUDA controls the internal thread scheduling, attempting to achieve as many concurrent active threads as possible. Threads are combined into small groups called *warps*, e.g., 32 threads for Kepler architecture, that have the same instruction scheduling. In practise, this means that every thread within the warp executes the same instruction, upon different data, step by step. This internal scheduling is hidden to the programmer, but must be considered when writing CUDA programs.

To the programmer, the threads are divided into thread *blocks* which together form a *grid* of thread blocks. Threads within a block can communicate and synchronize internally, but not across the entire grid. The programmer is responsible for declaring the size of each block and the number of blocks to launch, which combined results in the total number of launched threads.

Memory type	Location	Cached	Speed	Access Scope	Data lifetime
Global	Off-chip	No	100x	All threads	Alloc $\Rightarrow$ dealloc
Texture	Off-chip	Yes	1-100x	All threads	Alloc $\Rightarrow$ dealloc
Constant	Off-chip	Yes	1-100x	All threads	Alloc $\Rightarrow$ dealloc
Shared	On-chip	-	1x	Threads within block	Block
Registers	On-chip	-	1x	Single thread	Thread
Local	Off-chip	No	1-100x	Single thread	Thread

Table 2.2: CUDA memory types. Table from [42]

These dimensions are known within the executing kernel, as well as a unique thread and block index combination.

By default, the launch of all kernels and device utility functions will be queued until previous kernels have terminated. Alternatively, we can associate a kernel with a CUDA *stream*. All kernels associated with a given stream are synchronized in relation to each other, but can execute in parallel with all other streams. This also allows CUDA to better utilize the card, as kernels can be scheduled to run consecutively or in true parallel. This way, the pipeline can share resources between modules, instead of executing all kernels sequentially. There is also work that must be done on CPU to launch and synchronize kernels, but by using streams this can often be done automatically by CUDA, while the card is actively processing something else. Many utility functions also support asynchronous operations, e.g., `cudaMemcpyAsynchronous`, when associated with a stream.

2.6.2 Memory types

There are multiple types of memory offered by CUDA. In order to create fast CUDA applications, the programmer must know when, and how, to use the various forms of memory for maximum performance. Host (CPU) memory is inaccessible directly from kernels, and data must first be transferred to the device using utility functions. Table 2.2 shows a summary of the available memory types.

Each thread has a small number of on-chip 32-bit *registers* for local variables and intermediate storage, statically allocated for the lifetime of the thread from a shared register-file. If the thread requires more private memory than available, it will utilize private off-chip *local* memory with significantly higher access-times.

Each block of threads can utilize a limited amount of *shared* memory, located on-chip. This is shared within the scope of a single block and allocated for the lifetime of the block. Shared memory is particularly useful for repeated read and write operations, as well as synchronization within a block. It is only visible within the scope of the block, and results must be written back into global memory before termination.

Global memory is visible to all threads, located off-chip in the GPUs dedicated DRAM which is considerably larger and slower than the on-chip memory. Data allocated in global memory must be actively deallocated, similar to that of regular CPU memory. Newer devices allow for caching of global memory, at the expense of less available shared memory, but this is often not as efficient as CPU caching due to the increased number of active threads. *Constant* memory is a cached version of global memory, designed to allow faster read-only access to a limited number of constants shared by all threads.

Texture memory is also visible to all threads and located off-chip, but support multi dimensional caching. It is optimized for graphics processing, where we often use less predictable access patterns, e.g., pixel operations that depend on neighbouring pixel information. Texture memory is read-only, but newer devices also support writing by using *surface* memory. Both types of memory can be bound to a, potentially the same, *cudaArray* which is optimized for texture accesses and cannot be accessed directly. These arrays must be allocated and deallocated in the same way as global memory.

2.6.3 Performance pitfalls

CUDA is designed to make good performance very easy to achieve, but in order to provide truly great performance there are several pitfalls that should be avoided. Some of these are dependent on the specific device architecture, while others are more general. The Nvidia visual profiler [43] can often give great hints towards improving the performance of a system, or individual kernels, and has been used when optimizing our system.

Occupancy is a term for how well the device is being utilized, measuring how many concurrent threads can be executed compared to the theoretical maximum. This is very device dependent, as different architectures have different capabilities. We mentioned how threads are grouped into warps and blocks. There are limitations on the maximum number of active blocks and warps per block, meaning that the size of a block can have great impact on the overall occupancy. Additionally, shared memory and registers are allocated from the same memory pools, which can reduce the occupancy. In the case of too few registers, we can choose between using slow local storage and reduced occupancy.

We have described the various memory types available, which often have a large impact on performance due to widely different access times between on-chip, off-chip and on-chip caching. While texture memory allows for faster random accesses, global memory allows for higher bandwidth throughput for sequential memory accesses. Memory accesses from within a single warp can be coalesced into a single transfer, given that all accesses are within the same 128 byte cache line. This is called *memory coalescing*. In many programs, this pitfall can be the most challenging to avoid and cause the worst loss of performance. Irregular accesses will result in higher bandwidth requirements, longer access times and worse cache performance.

When using shared memory, *bank conflicts* arise if multiple threads attempt to access the same memory bank, typically a 32- or 64-bit word, during the same instruction cycle. This results in sequential access, unless the value accessed can be broadcasted to every thread in the warp.

We mentioned that there is a single instruction scheduler for every warp. This means that if there are *branching* code blocks within a kernel, e.g., if-statements, that evaluate differently within the execution of a warp, the remaining threads will need to stall while the branching code is being executed. All branches must be executed for the entire warp, given that the branching conditional statement evaluates true for a single thread.

Operations that require significant inter-thread synchronization are rarely fast. It is very often preferred to perform the same calculation in multiple threads, as opposed to heavily synchronized approaches. Any form of explicit synchronization requires threads to stall and significant scheduling, which is typically slow on all architectures.

More information on how to avoid these pitfalls, as well as other unmentioned problems, can be found in Nvidia's best practise guide [44].

2.6.4 Summary

In this section, we presented the primary benefits, limitations and performance pitfalls of the CUDA framework. While this is far from a full description, it should be sufficient for our purposes. For the remainder of this thesis, we will frequently refer to the terms and conclusions discussed in this section, without explaining these further.

2.7 Camera Reader Module

We will now begin discussing each of the individual modules of our panorama pipeline. The first step in the recording pipeline is the camera reader module, responsible for the capture of raw video frames. This is primarily done though the Pylon SDK [45], using an API wrapper created by Alexander Eichhorn at Simula Research Laboratory. However, several modifications had to be done to this wrapper library in order to support all of our functionalities.

Each camera stream is completely independent and executed in different threads, across the three recording machines. They all continuously attempt to fetch a new frame as fast as possible and deliver it to the Dolphin transfer module. Whenever a frame is retrieved, a local unix timestamp is applied to the frame header, and the frame is passed to the next module. This timestamp is important, as it is used for frame synchronization and for later video extraction of the panorama. To ensure that the timestamp is accurate across machines, every recording machine is connected to the same NTP time synchronization server [46], ensuring minimal clock drift.

In order to create a smooth transition between the stitched frames in the panorama, each source frame must be captured at the same time. Even small deviations can be seen when soccer players move through the stitching seam. The cameras are therefore triggered by the same Arduino device [47], which sends a trigger signal every $\frac{1sec}{framerate}hz$. This ensures that the cameras are only capable of capturing exactly $framerate$ times per second. It also means that the frames will typically be either completely synchronized, or one trigger interval off, making it easier to identify when a camera falls behind and loses a frame. When operating at a frame rate as high as 50, this step is not vital to the overall system. However, we want the system to function well across all lower frame rates as well, where these small differences matter.

2.7.1 Camera Array

In our prototype system, installed at Alfheim Stadium in Tromsø, Norway, we have five Basler acA2000-50gc [35] industrial cameras. They are placed approximately 5 meters above the ground, 5 meters behind the sideline and 3 meters right of the center line. They are capable of delivering a maximum resolution of 2046×1086 pixels at 50 frames per second over a gigabit ethernet interface. The output pixel format is a variation on the Bayer pattern [48], a raw format that must be expanded before regular usage, which we will discuss in section 2.11 and chapter 3.

We use 8mm lenses [36] which limit the vertical field of view to 37°. This would be insufficient to cover the entire soccer pitch vertically, based on our aforementioned camera placement. However, by rotating the cameras 90° we can circumvent this and achieve a higher vertical resolution, at the expense of more cameras. We calculated that with five cameras, rotated 90°, we would be able to cover the entire soccer pitch horizontally, ca. 160°, with approximately 4° of

Figure 2.2: Parallax effect, as two images show the same overlapping region with slightly altered background, due to a change in perspective as the camera moved. Images from [30].

overlap between each camera for stitching purposes. As an effect of this, the images need to be rotated later in the pipeline, currently in the stitching module.

Typically, panoramic image mosaic algorithms are based on the concept that each image is taken from exactly the same origin, only rotated on an axis perpendicular to the ground. If the cameras are placed far apart, close object will appear with different background in the two images. This is called parallax effect [30], and an example is shown in figure 2.2.

This presented us with a problem, as we need multiple cameras, with the center camera pointed downwards at approximately 9° for an optimal view. We designed and 3D-printed a small camera mount, where each camera is pitched, yawed and rolled to look through the same point ca 5cm in front of the lenses. This is shown in figure 2.3. Note how these cameras are oriented to reduce parallax effects, physically placed as close together as possible.

2.7.2 Exposure Synchronization

The cameras allow for continuous automatic or manual exposure configuration. In an outdoor setting, lighting conditions can change quite rapidly, and we found that some form of adaptive exposure control was required. The video camera is able to determine the optimal automatic

Figure 2.3: Mounted camera array

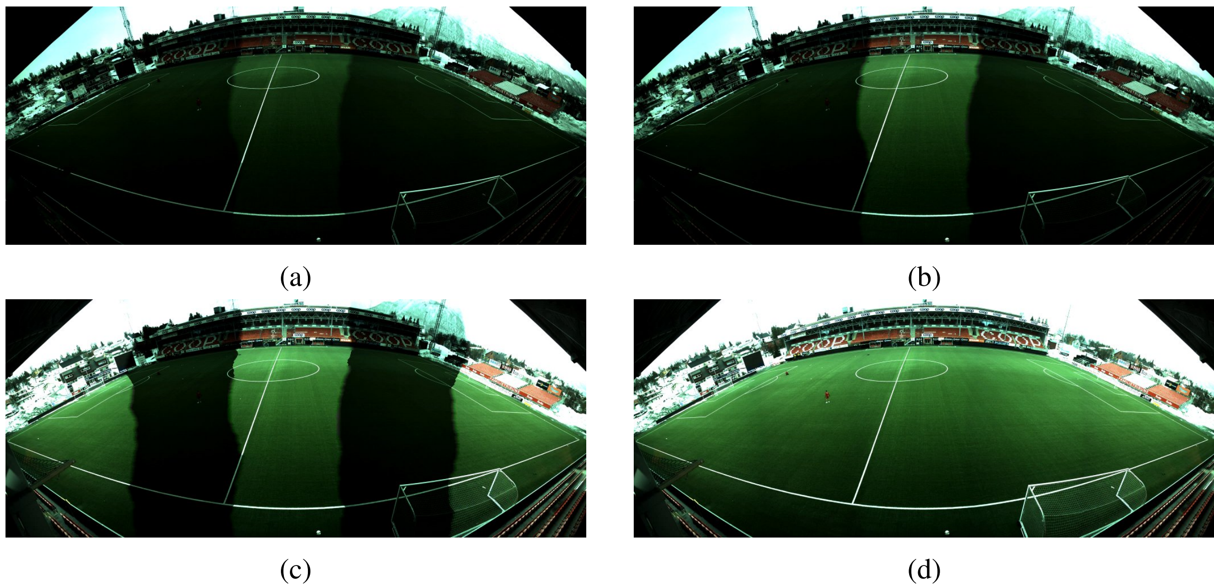
(a) Panorama with independent auto exposure

(b) Panorama with pilot camera exposure synchronization

Figure 2.4: Panorama with independent auto exposure vs pilot camera approach

exposure, however this will be independently set for each camera. This causes potentially large differences, which becomes a visual annoyance when the images are stitched together. This can for example be a major problem if some cameras contain areas that are particularly bright or dark, e.g. strong sun, bright snow or strong shadows. This was explored in [49], where multiple exposure synchronization approaches were tested. One such approach was to use a single pilot camera to control the automatic exposure, and broadcast this configuration to the other cameras. For this to function, we must ensure that all cameras are physically configured identically, in order for the same software configuration to produce equal results. In our recording pipeline, we use the center camera as a pilot camera, controlling the exposure values of the entire camera array.

When doing any form of image mosaicking, color correction generally needs to be applied on all the individual source images. This was done in the previous panorama pipeline [12], but has so far been found unnecessary in our new system. Given our identical cameras, lenses and capture software, it proved sufficient to use an identical exposure setting on every camera. This can be observed in figure 2.4. Here, we show the difference between a panorama where each



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

Figure 2.5: Panorama during the first exposure synchronization, four frames captured 20ms apart. In (a) and (b) the center camera has begun the process of auto exposure. The resulting exposure is read and asynchronously pushed to the other cameras, however, the resulting value does not reach two cameras in time for frame (c).

camera is set to automatic exposure and a panorama with exposure synchronization, based on the pilot camera approach. We see that the individual auto exposure sometimes can produce a good result, e.g., the two rightmost images, but other times may yield very different exposure times. The effects of this is even greater on sunny days, as the differences increase. Even though we believe the pilot camera exposure synchronization approach is sufficient, we can see some vignetting effects from the camera. This is why we perform some local blending of the seam, as should be visible in figure 2.4.

As previously mentioned, a static exposure setting is not acceptable in an outdoor environment. Instead we allow a single camera to set automatic exposure at a given interval, for example every ten seconds, and then broadcast the resulting exposure value to all the other cameras. This is done asynchronously by sending a exposure event message to the system server, i.e. the processing machine, which further broadcasts the message to all connected camera modules over TCP. This is not a time critical operation, and can be done through a best effort approach on each individual camera stream. Figure 2.5 shows an extreme case, where each camera asynchronously receives the first exposure update. Once the recording is up and running, there are typically only minor changes on each update.

2.7.3 Capture of alternating exposure

In this system, we also present a method for creating high dynamic range video, in order to combat the challenging lighting conditions of a large outside environment in the presence of strong sun. We will explain this in further detail in section 2.12.

This mechanism is achieved by capturing the initial video frames with alternating exposure, switching between high (bright) and low (dark) exposure times. This requires a very fine tuned control of the camera device, which is designed for asynchronous live configuration. Addi-

tionally, we will always have one or more frames in transit, as we are at least guaranteed that the next frame has already started its exposure duration by the time we receive a video frame. This is due to the fact that we have a minimum latency for transporting the data across the gigabit ethernet link, known as the transmission time, which in our case can be computed at $\frac{2040 \times 1080 \times 8bit}{10^9 bit/s} = 0.0176s$. At fifty frames per second, this is nearly a full frame in transit at all times. Additionally, we cannot guarantee that we do not occasionally drop frames or that frames arrive at the same constant rate. Therefore, if we merely attempt to swap exposure at a specific time interval we may be unlucky and choose a time that interferes with the capture of the next frame. Setting the exposure configuration requires multiple round trip times to the camera that may take a varying amount of time. It is also challenging to keep track of which frames are actually belonging to which exposure levels in these situations, as this was earlier unknown to the camera wrapper API we are using.

The way we solve this problem is through the Pylon SDK's event system, allowing us to receive a quick notification packet over the ethernet connection when the exposure of one frame is complete. This typically arrives ca 18ms before the resulting full frame is available. We create a new lightweight thread that merely listens for the end-of-exposure events, and records the frame number along with the expected exposure. Then, it immediately reconfigures the camera device before the exposure of the next frame. We require that the exposure time of one frame is short enough to guarantee us approximately 10ms to configure the camera between the end of one exposure and the beginning of the next, which should be sufficient in most scenarios. If we run out of time once or twice, it will mean that a frame is dropped. Besides, we are communicating over a direct ethernet link, not an open network, so there are fairly small variations in round trip times.

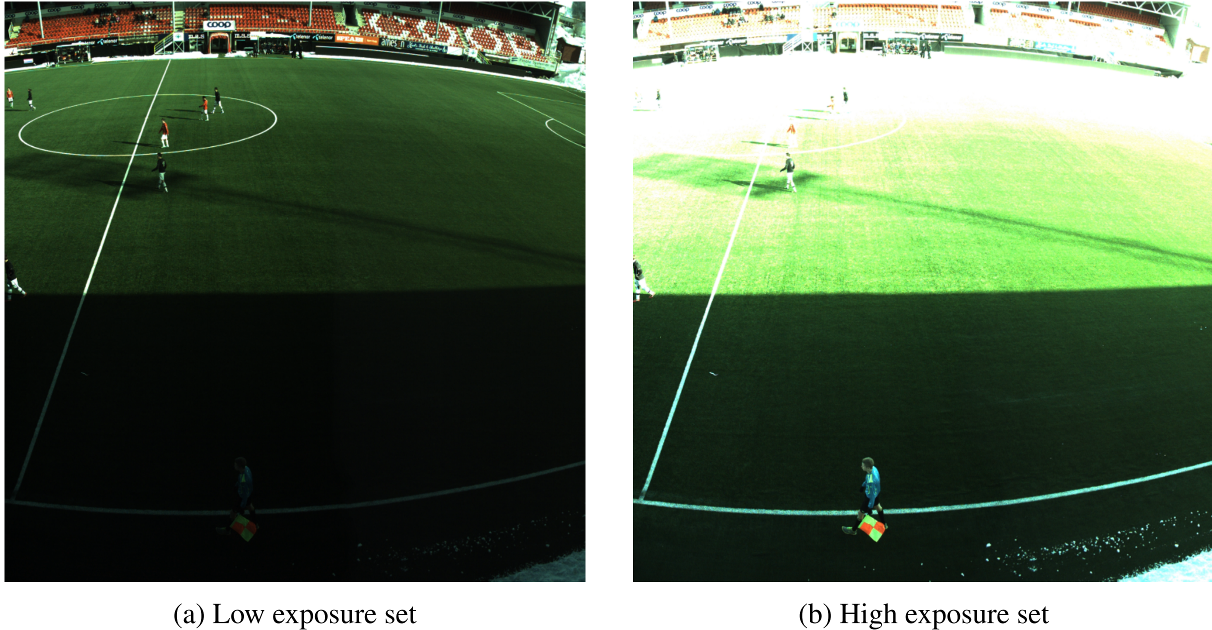
When receiving the full frame data, we now know exactly what the exposure configuration was at the exact recording of that frame, information we can attach to the frame meta data and deliver to the camera reader module. In situations where we are unsure of the exposure, we decide to drop the frame due to its infrequency. The reason for this is that the sudden appearance of a very bright or dark frame is highly noticeable to the viewer. If the frame is dropped, the synchronization module will duplicate the previous frame instead, which we believe is a preferred solution.

The camera reader module will combine two frames of alternating exposures, as soon as possible in the same fashion as normal, before delivering it to the rest of the system. The remaining modules of the system will then be operating at half the frame rate. This method is very simple, as most modules do not need to know whether HDR is active or not, they will merely be initialized with twice the size. We also accept that some frames within the same stitched set will have been captured up to 20ms apart, as the streams are likely to desynchronize slightly over time as random drops occur.

2.7.4 Determining the exposure values for HDR

The two exposure times to use must be selected with care. Figure 2.6 shows an example output, with the low and high exposure images stitched together. Note how the two images compliment each other by showing details in different regions. This is vital to the success of the HDR process. By selecting exposure values that are too close together, we see less gain from the HDR module and we may see a lot of noise in dark areas, as these are boosted.

This is especially challenging as we still wish to utilize dynamic exposures, setting auto-



(a) Low exposure set

(b) High exposure set

Figure 2.6: Sample exposure sets, used by the HDR module, showing the low and high exposure frames. Note that these have been stitched for visual purposes, though in the actual module they are computed on the raw input image set, before stitching.

matic exposure on a single camera and allowing the other streams to duplicate the configuration. Setting automatic exposure, in our system, takes two or three frames when operating at 50 fps before the values are successfully updated. In this period we decide to drop the frames, causing a small hiccup of 80ms. However, we found that the automatic exposure updates can be fairly infrequent, i.e. every ten or twenty seconds is sufficient.

The actual low and high exposure times can be determined in several ways. One approach is to use two regions of interest, one positioned in the sunlight and one positioned in the shadow. This works well, although the resulting exposure values tend to be too far apart for the HDR module to handle it well. However, we saw that this was difficult to do in real-time updates, because changing this region of interest in the camera is extremely slow. Another approach is to perform auto exposure with a single region of interest, preferably solely in the sun as this is more sensitive to changing light conditions, and then set the other exposure time as a static offset or a static percentage increase/decrease. This can work well in many situations, but we do not necessarily know the difference between the two regions, as this scales with brightness of the scene. We could also set a few static values, and use whichever is closest based on the result of the auto exposure.

We chose to use a mixture of these two approaches, by first defining the two regions shown in figure 2.7. Then, we spend a few seconds at the beginning of the recording to estimate the optimal exposure times in each of these two regions. This gives us the relative difference between the two regions. This difference is typically way too large, and we adjust the high exposure time E_{high} by:

$$E_{high} = E_{low} + \frac{E_{high} - E_{low}}{2} \tag{2.1}$$

Then, we use the low exposure region of interest, i.e., sunny region, to determine the automatic exposure of E_{low} throughout the recording, updated as aforementioned. The high expo-

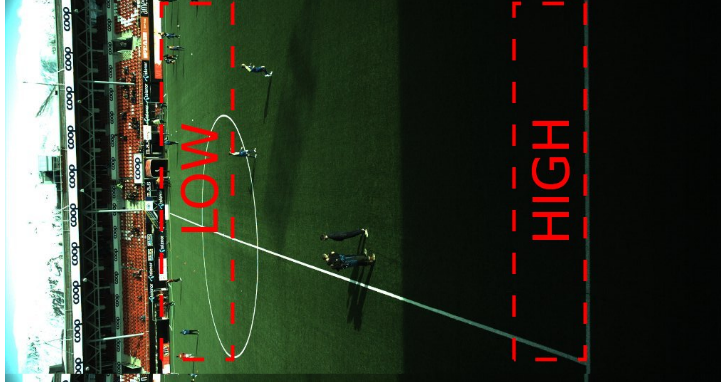


Figure 2.7: Regions of interest used for determining auto exposure for HDR

sure time is determined by:

$$E_{high} = E_{high} + \frac{(E_{updated} - E_{low}) \times E_{low}}{E_{high}} \tag{2.2}$$

where $E_{updated}$ is the new low exposure time. This way, we maintain a correlation between the two exposure times. This assumes that the initial correlation was accurate, however, which may not be the case if the weather conditions move from cloudy to sunny during the recording. Note that this functionality was added quite recently, with very few days of sun to test the different approaches before the delivery of this thesis.

2.8 Dolphin Producer/Consumer Modules

The Dolphin producer and consumer modules are responsible for transporting the raw video frames from each of the recording machines to the processing machine.

Transferring these frames across the network requires very high bandwidth, too high for regular gigabit ethernet. Instead, we use a high speed PCIe based network created by Dolphin Interconnect Solutions [34]. This hardware, along with the SISCI-API [38], allows us to perform direct memory accesses (DMAs) to all machines in the same local cluster. By using a PCIe link, we can achieve significantly higher throughput than ethernet, at a lower latency.

Figure 2.8 shows the achieved bandwidth when copying data between machines, with a varying message size. We transport each frame as a single message which means that we can achieve maximum throughput. This yields an approximate end-to-end transfer time of 820µs, for each frame of 2040×1080 plus the small header of 32 bytes. When HDR mode is active, this number is doubled as two frames are transferred at once. Based on the achieved bandwidth, we also see that the system can be scaled much further. We should be able to transfer $\frac{2574MB/s}{2040 \times 1080} = 1225$ frames per second. With a frame rate of 50 fps, this means that we can theoretically support 24 separate camera streams. This number can also increase when transporting between other machines, as the primary bottleneck of the system is the processing machine. Figure 2.8 also shows that we can transport data directly onto the GPU device, without a performance loss.

The dolphin producer and consumer uses two rotating buffers, on each computer. The producer module fills its local buffer with a frame from the camera reader. When the consumer module sends a signal that a buffer is ready to be written to, the data is transferred across the link, through a DMA request. In practise, this signal will be received before the data is ready

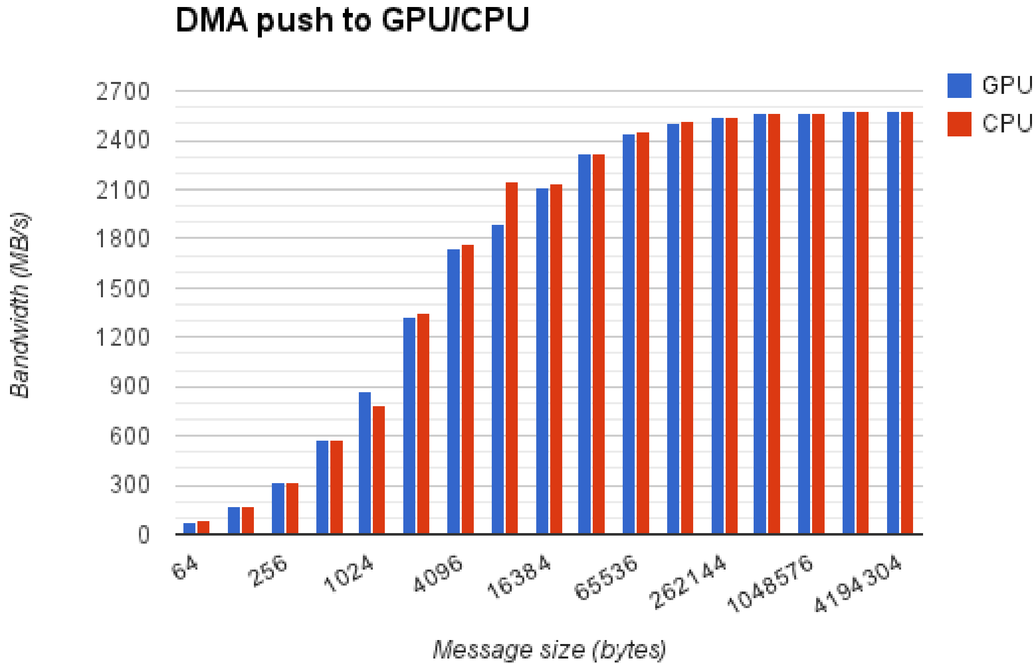


Figure 2.8: Plot showing the achieved bandwidth, by transferring different sized messages across a Dolphin link. This shows the time required to write data into a remote CPU/GPU buffer, originating from CPU memory. Figure from [50].

for transfer. When the DMA is complete, the consumer module is notified and the frame can be delivered to the frame synchronization module. The synchronization module will copy the data out of this buffer, as dynamically mapping up dolphin-accessible memory is slow. This means that when the frame is delivered, the buffer can be re-used.

2.9 Frame Synchronization Module

The frame synchronization module is responsible for combining the individual camera streams into finite sets of frames, and ensuring that the system remains real-time.

Each dolphin consumer will deliver incoming frames to the frame synchronization module immediately, using a push-based interface. When a full set of frames is gathered, they are inserted into a short buffered queue. When a module requests a frame-set, either the first set available in the queue will be delivered or an interrupt timer will be set until the next real-time deadline. The thread sleeps, either until the timer expires or until a new set is inserted. If no set is added, the previous frame is duplicated. This ensures that the pipeline remains real-time, even if the cameras occasionally drop frames.

In order to determine if two, or more, frames from different camera streams belong to the same frame set, we analyze the timestamps defined in the frame header. They are determined to be equivalent if the timestamps differ by less than $\frac{1s}{2 \times fps}$. In section 2.7, we mentioned that the cameras are triggered by the same Arduino device running a custom triggering software [47].

This, along with the use of an NTP server for all recording machines, means that all frames will either be captured less than 2-3 ms, or approximately $x \times \frac{1s}{fps}$, apart.

When a new frame is inserted, from any camera, it is placed into either the *current set*, the *next set* or a new set is created. When a new set is created, meaning that we have received a frame belonging to $current + 2$, we determine that no additional frames will arrive belonging to the current set. If the set is incomplete, the module will classify this as a frame drop. The arrival of a new frame is easiest explained by the pseudo code in algorithm 2.1.

Algorithm 2.1 Arrival of a new frame for synchronization

```
Data: currentSet, nextSet
Require: frame ≠ null ∧ index ∈ [0–4]
  function PUTRESOURCE(frame, index)
      if currentSet = empty ∨ frame.time ≡ currentSet.time then
          currentSet[index] ← frame
      else if nextSet = empty ∨ frame.time ≡ nextSet.time then
          nextSet[index] ← frame
      else
          if currentSet.isComplete then
              ADDTOQUEUE(currentSet)
          end if
          currentSet ← nextSet
          nextSet ← empty
          nextSet[index] ← frame
      end if
  end function
```

When recording with HDR active, the evaluation of which frames belong to the same set may cause problems. In the camera module, we join two sets of frames into a single pair of low and high exposure frames. This means that they capture frames at twice the frame rate of the rest of the system. Therefore, if the frame synchronization module believes the frames should arrive every 40 ms, their timestamps may in fact differ by 20ms as they desynchronize by one trigger interval. This is because one camera may record its high exposure frame, while another camera is recording its low exposure frame. We modify our equation for determining if two frames are of the same set to account for this as such:

$$\frac{3}{4 \times fps} > (Frame.time - Set.time) >= -\frac{1}{4 \times fps} \tag{2.3}$$

This way, all frames must belong to a single set, and frames can be offset by half the frame rate.

In order to maintain consistency, we discard all incomplete sets. Instead, we duplicate the previous set of frames. This decision is based on the infrequency of dropped frames, measured at an average below 0.2% at 50 fps. This number may often lie though, because drops very commonly occur sequentially or in a particular pattern. For example, we frequently see a frame drop during the automatic exposure synchronization process. When recording HDR video, we are in fact guaranteed three dropped frames at every exposure update frequency, as this is how long the camera driver needs to find a new auto exposure. Either way, these numbers are deemed unnoticeable for the user.

The buffer queue in the synchronization module allows the pipeline to have a little flexibility on its real-time constraint, as there may be small variances in execution time. If the buffer reaches a small limit, e.g., 5 frames, it means that processing is going too slow, and frames are discarded. This is something that should never occur, and is not really handled in our system. As of yet, we have only seen this occur when writing files to an external file system or running experimental modules. In the future, this could perhaps be handled by introducing variable frame rate in the video encoder.

The frames are given a single shared header and placed continuously in memory, before being delivered to the next module. In the case of HDR recordings, this module must be aware of this mode. The frames are then separated based on exposure value, creating one full set of dark frames and one full set of bright frames.

2.10 CUDA Uploader and Downloader Modules

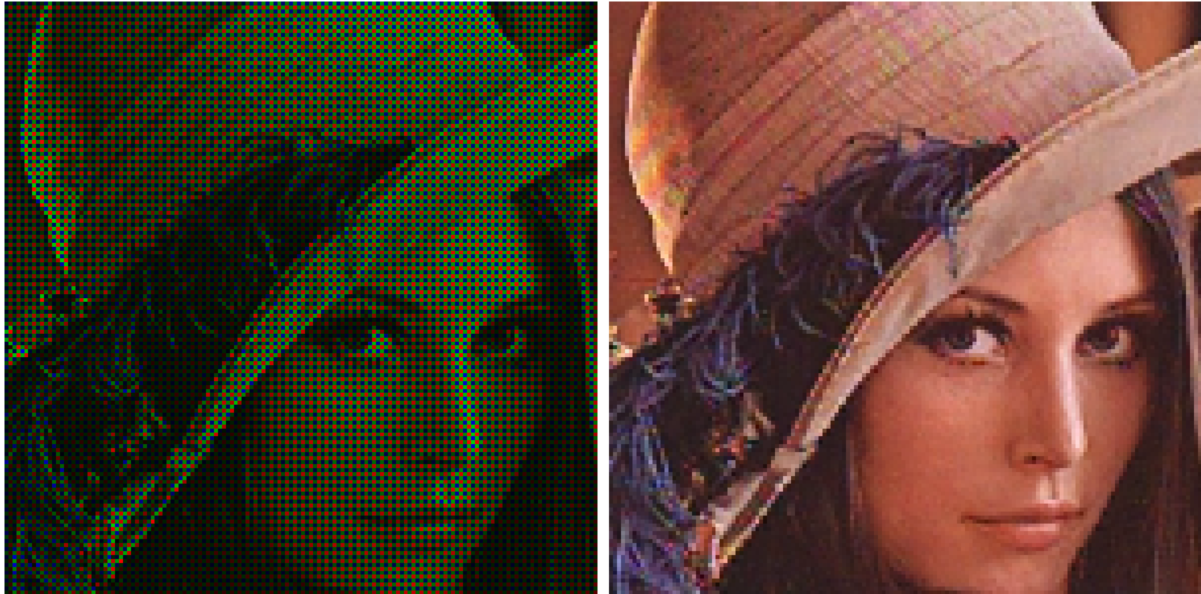
The next modules will be executing on the GPU, which means that we need to transfer each set of frames into GPU memory. The CUDA uploader module is responsible for this. The data, currently five continuous images, is copied asynchronously from a single CPU array to a single GPU array, using either `cudaMemcpyAsynchronous` or the cudaArray variant `cudaMemcpyToArrayAsynchronous`. To allow asynchronous transfer, the CPU memory must be *pinned* to ensure that the CPU pages are not swapped out in the middle of a copy operation. The uploader module will place the data in a two dimensional `cudaArray` by default, specified by the bayer converter module. The frame header is passed along in CPU memory once the transfer is complete.

When the data is finished processing, i.e., stitched into a complete panorama, the CUDA downloader module is used to transfer the panorama back into CPU memory for video encoding. There is little difference between the uploading and downloading process. The data is copied asynchronously into a pinned memory array. This presents us with a problem, given our overall modular system design, as the buffer allocated by the encoder module is not pinned. Currently this is simplified by using a temporary pinned buffer and a second CPU-to-CPU copy operation into the buffer allocated by the encoder. This is an obvious source of optimization, at the expense of tighter module coupling.

In the master thesis of Marius Tennøe [42], the PCIe bandwidth requirements of the old bagadus pipeline is defined at 737MB/s upload and 291MB/s download. Despite having significantly higher resolution images and frame rate compared to the old bagadus pipeline, the total bandwidth requirements of our uploader and downloader remain low due to a more compressed panorama output and the input pixel format. The required bandwidth can be measured by equation 2.4.

$$\begin{aligned} uploadBandwidth &= (2040 \times 1080 \times 5)\,\frac{bytes}{frame} \times 50\frac{frames}{second} &&\approx 525\frac{MB}{s} \\ downloadBandwidth &= (4096 \times 1680 \times 2)\,\frac{bytes}{frame} \times 50\frac{frames}{second} &&\approx 656\frac{MB}{s} \end{aligned} \tag{2.4}$$

As we can see, this is well below the PCIe 3.0 maximum bandwidth of 16GB/s per direction [51].



(a) Bayer patterned image (b) Reconstructed image

Figure 2.9: Example of a bayer image and its reconstruction

2.11 CUDA Bayer Converter Module

The first module to execute on the GPU is the bayer converter module. The pixel format read from the cameras is *Bayer GR-8*, a raw and incomplete format, where each pixel does not carry a full set of color information. The bayer converter module is responsible for converting the input images from the Bayer pixel format, one byte per pixel, into YUV444 interlaced, four bytes per pixel.

YUV is a common pixel format for video, consisting of three color channels. The Y-component represents luminance, the sole component of grayscale images, while two remaining channels, the chrominance, represents the red and blue differences. The chrominance channels can typically be subsampled, as rapid transitions of chrominance is rare and difficult for the eye to detect. More information, along with an overview of possible format representations, is given in [52].

Because the Bayer pixel format is an incomplete format, this is not merely a single step conversion. The bayer pixel format is based on the common RGB pixel format, where a pixel is represented by its red, green and blue color values. In the bayer pattern, however, only a single color is present in each pixel. The primary purpose of the bayer converter module is therefore to expand this limited color information into three full channels of RGB colors, a process referred to as *bayer demosaicking* or *debayering*. This process is visualized in figure 2.9, where the bayer pattern has been imposed, stripping away two color channels in each pixel and then reconstructed by the demosacking algorithm. The secondary job of the module is to convert the final RGB pixels into the YUV colorspace used in the remaining modules of this pipeline, which is done using a quick matrix multiplication.

This module receives all five input images stored consecutively in memory and treats all the incoming data as a single, tall image. This is made possible by the lack of context information required, and the fact that image borders will be hidden by the stitching process. The input will

be bound to a texture, and is therefore received in a single cudaArray from the uploader module.

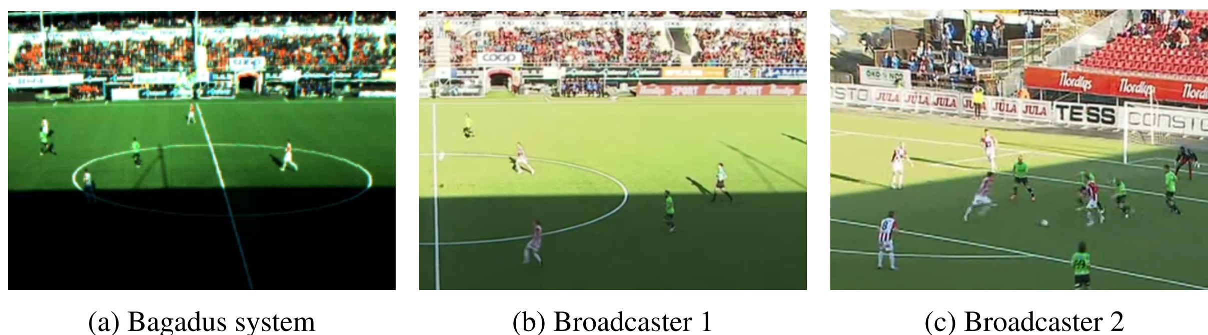
The output of this module is also stored in a cudaArray, written to using surface memory after its conversion to YUV444. The output pixel data uses an interlaced format, i.e. the Y-,U- and V-components are consecutively stored in memory, to provide optimal data locality for the stitching module. Although there are only three channels, Nvidia recommends using four bytes per pixel due to the typical benefits of 32bit memory alignment.

Choosing a bayer demosacking algorithm for this module can greatly affect the quality of the final panorama, and multiple algorithms have therefore been investigated. This is detailed in chapter 3, along with further information on the implementation of this module.

2.12 CUDA HDR Module

In chapter 1, we saw that the old Bagadus system struggled with challenging lighting conditions at the prototype installation site. This can be observed in figure 2.10, where we see that even professional broadcasters, with better equipment and manpower to perform live adjustments, struggle with the strong shadow.

We could likely achieve a better result through a manual exposure configuration, updated live, optimal for the given time of day and lighting conditions. This is done by the broadcasters in figure 2.10, but we need a method to perform this automatically, without human interaction. A solution to this problem is to use multiple low dynamic range (LDR) images, sampled with different exposure levels, to construct a single high dynamic range (HDR) image. For optimal quality HDR images, it is preferred to have as many samples as possible. However, in our approach we cannot afford having more than two different exposure samples. This is because more samples means that the frame rate must be reduced, since our cameras are capped at 50 fps. Additionally, if we use too many frames there will be too much time in between the individual images, which can cause visual artifacts in areas of movement, such as ghosting or blurring effects. Therefore, we aim to record at 50 fps, alternating between a low and high exposure image, and produce a single 25 fps video output. The capturing part of these images was detailed in section 2.7.3, and an example of the resulting images was shown in figure 2.6.



(a) Bagadus system (b) Broadcaster 1 (c) Broadcaster 2

Figure 2.10: Lighting challenges at our case study stadium. Comparison with two professional broadcasters. Note that images are from the same game, but different situations.

2.12.1 HDR algorithms

To convert the two LDR images into a single HDR image, a radiance mapper is used to fuse the images together. This fused image cannot be displayed by regular devices, so this pixel values must be compressed into a displayable range through a tone mapping process. We present an evaluation of multiple combinations of these two steps in [24], directed towards our pipeline. In this thesis, we only present the most promising implementations for our scenario.

One of the most cited approaches to radiance mapping is presented by Debevec *et al.* [53], where the authors try to recover HDR radiance maps from photographs. It performs an estimation of the cameras response function and then a weighted selection process. In this process, the information is extracted from mid-tone regions of different exposures. We made a GPU implementation for the second step alone, since the first step needs to be performed once in the lifetime for a camera. Although other algorithms were evaluated and implemented, we saw best results from this algorithm.

An algorithm for tone mapping was proposed by Larson *et al.* in [54]. This approach performs tonal compression, by creating a look-up table per frame to represent a desired histogram. Unlike a simple histogram equalization, the target histogram is computed taking human contrast sensitivity into account.

Another approach was presented by Reinhard *et al.* in [55]. Here, they try to emulate a technique called "dodging & burning" [56], a well established technique in traditional photography. This approach relies on the information from local neighbourhood for tonal compression. Adaptive Gaussian kernels are employed along different dimensions to average the exposure value, the adaptive nature is from the fact that the size of these kernels depends on the local contrast changes. The algorithm uses multiple gauss kernels to improve the quality of the image, by smoothing out the noisy output of the radiance mapper.

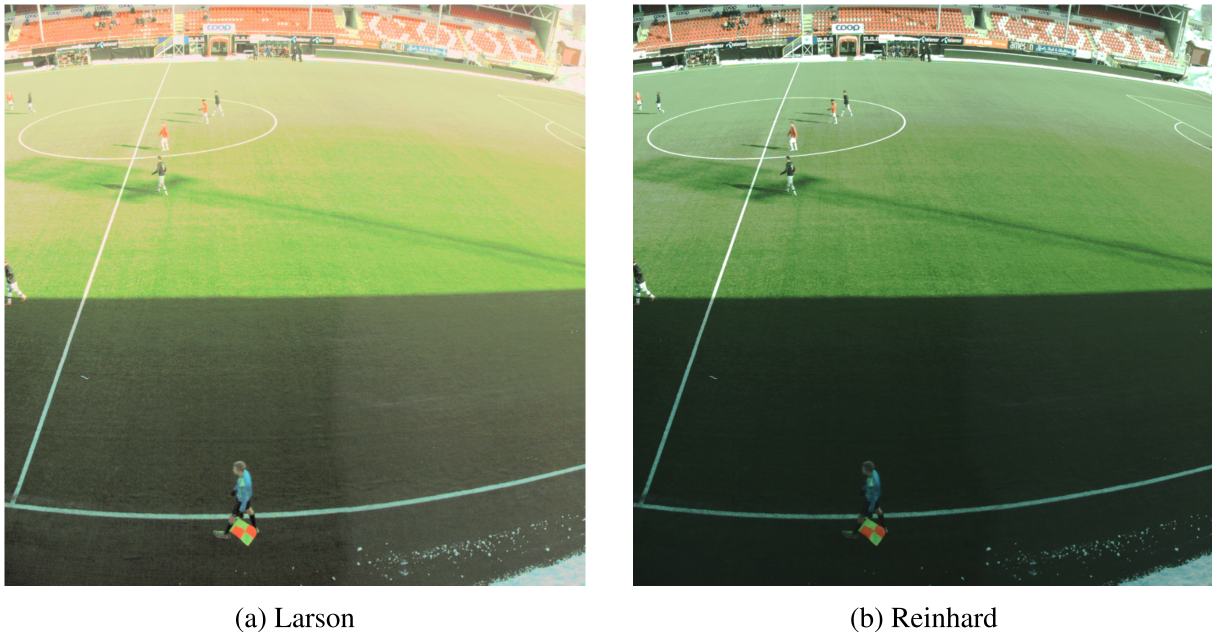
2.12.2 Application for our pipeline

In our pipeline, we execute the HDR module after the debayering step and before the images are stitched. This simplifies the stitching process, but it means that the HDR module must use a full $2 \times (2040 \times 1080 \times 5)$ input. The HDR process is also performed on the entire image, similar to what we described for the debayer module, instead of individually for each image. This, however, is vital for the overall system because the tone mappers will utilize global information, such as the overall image luminosity, to set the tone level. This could result in very different tone levels for neighbouring images and create visible seams once the images are stitched together.

The module executes on the GPU, and receives two sets of five input images, all arranged in a cudaArray as a $2040 \times (10 \times 1080)$ image. The input pixel format is YUV, with 4 bytes per pixel. Its output format is equivalent to its input, only with half the number of pixels.

2.12.3 Choice of algorithm

The visual result of the two tone mappers, using the Debevec radiance mapper, can be seen in figure 2.11. We see that Larson's algorithm boosts the overall brightness of the image, creating an overall too bright result. However, we see that that details in shadowed regions are highly visible. This algorithm boost the signal in the dark areas, which introduces a lot of noise to these regions. We see that, when the signal is highly boosted, there are differences between



(a) Larson (b) Reinhard

Figure 2.11: Here we show the output of the two tone mappers, using the Debevec radiance mapper. These are constructed using the input from figure 2.6.

Larson	Reinhard	
	1 gauss kernel	3 gauss kernels
24.77 ms	39.94 ms	66.78 ms

Table 2.3: Execution performance of the HDR implementations

the two cameras. Therefore, this algorithm can make otherwise invisible stitching seams highly visible. Reinhard's algorithm produces a more naturally colored image, but may appear too dark in areas of shadow in some situations. However, we see that most details in dark areas are still boosted to the point of visibility. We also see some differences between the two images, as a result of the boosting, but this is minor. Based on these results, we prefer the second approach because it produces a more natural image, while still providing a strong enough signal to discern details in dark regions. The lack of noise also improves the visual result, and effectiveness of the stitching module.

The two tone mappers vary significantly in complexity, as the Reinhard algorithm is much slower. This algorithm must also allocate an extra 126MB of memory for every gauss kernel. The original paper recommends using eight gauss kernels, but due to memory restrictions we limited this number to three. Based on visual evaluation, the difference between this number of kernels is minor and we believe that the best approach for our scenario is to use a single gauss kernel. Table 2.3 shows the execution times of the two algorithms when executing alone on the GPU, with two different numbers of gauss kernels. We see that there is a great disparity between the two algorithms. Larson's algorithm is well within real-time boundaries of 40ms, and can function in our pipeline at 25 fps. With one gauss kernel, we see that Reinhard's algorithm is strictly speaking within the real-time limits, but running it along with the other modules of the pipeline adds approximately 10ms to the execution time. Because of this, the pipeline is currently using Larson's algorithm. However, we saw that Reinhard's algorithm performed

visually better. This algorithm can be used if a frame rate of 20 is deemed acceptable, as opposed to 25, and can potentially be optimized to function at 25 fps as well.

2.13 CUDA Dynamic Stitcher Module

The next module, executing on the GPU, is the stitching module. It is responsible for stitching together the five input images, and create a single cylindrical panorama. This is shown in figure 2.12, though in practise each of the five input frames are stored as a single tall image, rotated 90°.

First, each of the input images are separately projected, based on a lookup table generated offline (see chapter 4). This projection differs from the one presented in the old Bagadus system, there performed in a separate warper module, in that it projects onto a cylindrical surface instead of a vertical plane. This causes the vertical lines to bend, as seen in figure 2.12. This type of projection is performed to allow for a virtual view delivery system, as we will detail in chapter 5. The projection step also performs the image rotation, and pixel interpolation.

The rest of the stitching module can be considered a natural extension of the dynamic

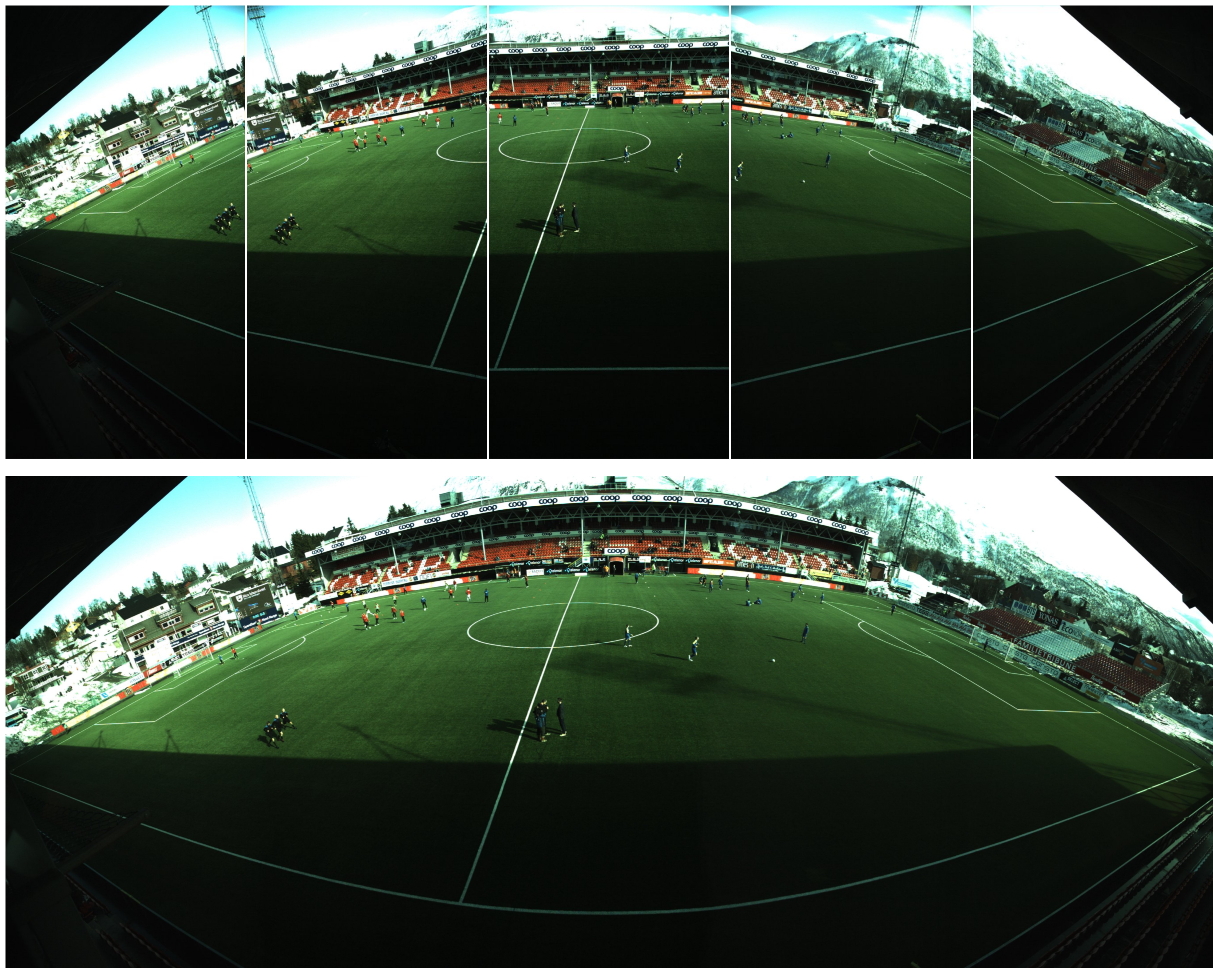


Figure 2.12: Input and output of the stitcher module. Note that the input images have been rotated and aligned horizontally for easier demonstration.

stitcher presented in the old Bagadus system, specifically the thesis of Espen Helgedagsrud [31]. Performing a static seam cut may introduce artifacts, as it is challenging to perfectly project and align the images. We have adapted, and modified, dynamic seam seam algorithm of the old Bagadus system, which was designed as to avoid stitching through players. The dynamic seam is calculated for every frame, but designed to avoid moving unnecessarily. Once the images have been stitched together, we also perform a slight seam blending to reduce potential color differences between overlapping images. In the old Bagadus pipeline, this was handled by performing color correction, but we choose a more localized approach because our differences are much smaller.

The input to this module is five images, stored consecutively in a cudaArray bound to a texture. We can use the height of one image multiplied with the image index to read from the correct image. The data is in YUV444 format, with four bytes per pixel. We downsample this, to YUV422, and write the output in planar format as required by the encoder module.

A more detailed view of the stitching module will be given in chapter 4, where we describe each step of the module, along with design and implementation details.

2.14 X264 Encoder Module

The final module in the pipeline is the X264 encoder module, responsible for encoding raw video frames into the common H.264 video format and writing the video to disk. Encoding the video is essential for long term storage and to transport the data over any network. The module outputs short video segments that are well suited for real-time streaming applications or post-game video extraction. These files are structured using date, timestamp and sequence number as file names. The date and timestamp is retrieved from the frame header, dating back to the creation of the frame on the recording machine, making it possible to store frames with extremely accurate time information when extracting video later.

The encoder module receives raw planar YUV422 from the stitcher module. We found that there is little difference between using YUV422 and the further downsampled YUV420 in terms of performance, but may result in improved visual results. The differences are, however, minor. The X264 encoder module is therefore also capable of encoding YUV420 if that should prove sufficient in the future. The output dimension of the panorama video is determined by the lookup table created for the stitching module. We could have used a larger video resolution, e.g., 4400×1800. However, we decided to perform some downscaling when performing the image projections, in order to stay within the common maximum supported horizontal resolution of 4096 pixels. This is the default for 4K video, which is supported by far more devices than higher resolutions. We may also see better performance, both when encoding and decoding, by using multiples of 8, 16 or 32. By slightly reducing the output image resolution we saw a great reduction in file size and increased encoding speed.

The module is executed entirely on the CPU. Video encoding of a high resolution panorama requires a significant amount of computational resources, utilizing nearly the entire CPU in order to remain real-time. Additionally, we want to keep the output file size low, which stresses the encoder further. The actual video encoding is performed using the *libx264* library [57]. This library allows us to configure the encoding process significantly and optimize for performance, compression rate and visual quality through an intimidating list of parameters [58]. However, it is generally considered best to stick to one of the default configurations, limited to setting an

overall *profile*, *preset* and *tuning*. We use the general *high* profile, which produces a standard high quality video stream that can be decoded by nearly all common video players, ensuring that no potentially unsupported features are used. We use the *ultrafast* preset configuration, which is required to maintain real-time encoding, by significantly reducing the time required to encode each frame. This preset reduces the visual quality of the encoding, but primarily affects the final file size. Setting ultrafast preset will typically mean that the encoder will not spend as much time searching for the best motion vector, instead settling for a worse match. A good motion vector means that fewer bits are required to represent one macro block, typically 8×8 pixels.

When performing live streaming today, we typically divide the video into short segments, varying between one to ten seconds. This introduces a few seconds of delay to the user, but simplifies the online video delivery. Internet is designed for throughput, i.e., fast transfer of large files, and not consistent connections between multiple clients. It is therefore more robust, efficient and scalable to deliver larger segments, compared to delivering each individual frame. More information on this can be found at for instance [59], describing Apples live HTTP streaming approach. In our system, we therefore divide the video into small segments, default at 3 second duration. Upon the creation of each file, the file path is inserted into a system database for later search and extraction. Additionally, we maintain a live manifest of the files belonging to the current recording that can be used for live HTTP streaming. In other words, the encoder module attempts to facilitate several possible methods of live and offline video extraction methods. These will be more looked at in chapter 5.

When encoding H.264 video there are typically three types of video frames utilized. I-frames, or keyframes, contain all the information required to display that frame. P-frames uses the concept of *motion prediction* to re-use information from previous frames, potentially specifying a pixel region based on its transition from the previous frame to the current, instead of its direct image data. Because there is typically little movement in large portions of all video from one frame to the next, this can reduce the output file size significantly. In our soccer stadium prototype, the majority of the video is static background, with minimal areas of movement. B-frames can use information from future frames, as well as previous, but is not used in this pipeline as it requires more time and lookahead, as well as being unsuitable for video streaming. The interval between each keyframe, the *GOP* size, in the stream has a great impact on the output file size. Keyframes contain significantly more information, but can be faster to encode because no motion prediction is required. When decoding the video stream, the decoder must find the first keyframe before it can begin playback. In our pipeline, we want to allow for live streaming and wish to maintain a relatively low file size. Therefore we use a single keyframe at the beginning of each file segment, the minimum required to decode each file individually.

When we have encoded a full video segment, we need to complete that file and create a new segment. Because the encoder uses a slight lookahead buffer, there may be residual data that needs to be finished before we can open a new file. Therefore, the switch to a new file can create some spikes in encoding time that are adapted for by the small buffer in the frame synchronization module. This allows the encoder a couple of frames to catch up to real-time.

This module is different from the other pipeline modules when it comes to performance, as the time to encode each frame is highly dependant on the visual data. Recording a static scenery with minimal movement requires little motion prediction and can be encoded relatively fast, while significant movement, along with for example flickering lights or falling snow (a real

world problem in our Tromsø case study), can cause significant fluctuations in encoding times. When the scenery is particularly dark, the cameras will compensate by boosting the signal and create visual noise. We saw earlier that the HDR module can also produce this effect, by boosting the signal in shaded areas. This can cause the encoder to be significantly slowed, as the noisy areas differ from frame to frame even if the actual scenery remains static. In these situations, the file size can also grow to upwards of ten times the regular size. Very bright scenery can also have a negative effect, as high valued pixels cannot be compressed as well as low valued. This can both increase the file size, and increase the time required to compress the pixel values. In summary, this module has the most unpredictable processing times, i.e. harder to ensure to always stay within real-time constraints. If the module ever falls behind real-time, we are forced to duplicate frames, i.e. encode the same frame twice and rely on the small speedup this gives. While the H.264 standard supports a variable frame rate, this is not well supported by all decoders and can produce problems in our video delivery systems.

The final file size of the encoded video is very important, as it directly affects what video delivery systems are possible. We saw that the frame rate of the video matters very little, since a large part of the file size is used for the single keyframe. Additionally, when operating at 50 fps, as opposed to say 25 fps, the relative motion between two frames is very low. This means that with a higher frame rate, better motion vectors can be found and we need an overall fewer bytes to represent each pixel. We see no more than a 10% increase in file size by doubling the frame rate, from 25 to 50, during regular conditions. This results in, on average, file sizes of approximately 3-6MB which is equivalent to ca. 12Mb/s. At 50 fps, 150 frames per file, the single keyframe uses approximately 20% of the entire file. This shows that P-frames allow for very high compression rates. In different weather conditions, or with high levels of noise introduced, we may see file sizes of up to 16MB per file. Sizes over 8MB, however, are extremely infrequent and introduced by the weather conditions.

Comparatively, the old Bagadus panorama pipeline produced a final file size of approximately 35MB, despite a 10% lower resolution, lower frame rate (30fps) and using the further downsampled YUV420 pixel format. The quality was also lower, as a result of poor configuration and higher CPU load. This shows that our new pipeline produces a more efficiently encoded panorama, better suited for video streaming and future extraction.

2.15 Visual Results

In figure 2.13, we see a comparison between the old Bagadus panorama and our new pipeline. These are captured with approximately the same lighting conditions, but we see from figure 2.13a that the old panorama pipeline struggled with camera exposure, especially in combination with snow. The most striking difference, however, is the relative vertical resolution. The new pipeline produces a panorama with only 10% more pixels, but since we do not excessively warp the image to expand pixels, we produce a much higher resolution overall. This is especially apparent when viewing the opposite end of the field.

We can see that the panorama in 2.13b contains no visual artifacts due to image misalignment. The old Bagadus pipeline had a different camera placement, with a large gap between the two center cameras, which made it difficult to stitch together these two images. This can be seen at the opposing stands in figure 2.13a, where the two images are vertically misaligned. The dynamic seam in our new pipeline, however, is able to make a smooth transition in the

(a) Old pipeline output

(b) New pipeline output

Figure 2.13: Comparison between the old Bagadus pipeline and our new pipeline.

entire image. The white lines of the soccer pitch are perfectly sliced, without mismatch. However, even though we cannot identify the seam itself, we see that it is possible to discern the approximate borders of the five images. This is not always the case, but depends on the lighting conditions. It is also visible in the old Bagadus system, but the color corrector hid this with varying levels of success. In our system, this stems primarily from the cameras vignetting effect, which causes a relative darkening of the image borders. This would likely be less apparent with more horizontal overlap between the images. However, we will see in chapter 4 that this effect is invisible when we zoom into sections of the panorama. Our eyes are able to discern the subtle color difference between the two image regions at a distance, but when viewing only a small section of the panorama, it is far more difficult.

In chapter 1, we expressed problems with the visual quality of the old panorama pipeline. We especially noted the difference in quality depending on the region of interest. When performing a cylindrical projection we see greatest warping around the seams. If the images need zero rotation (rare, but possible) then there should be no warping effects at the center of each image projection, while pixels farther away from the center will be compressed. Additionally, there is little warping at the center height of the image, meaning that the maximum is near each of the four corners. More warping is required when images must be rotated significantly,

(a) Zoom with nearest neighbour interpolation

(b) Zoom with lanczos interpolation

Figure 2.14: Example from our panorama pipeline, where we perform 300% digital zoom. Note that this is a similar region as the one shown in chapter 1, figure 1.1.

(a) Zoom with nearest neighbour interpolation

(b) Zoom with lanczos interpolation

Figure 2.15: Example of zooming into the same region in the original camera stream. Note that this stream has still been debayered.

though with a good physical camera configuration this can be avoided. More information on this process is given in chapter 4.

In figure 2.14, we can see an example of zooming into the very far side of the soccer pitch. There is a seam going directly through the this area and it is nearly the highest vertical area of interest, which means that it is likely the lowest quality. Of course, there are areas that are farther away that will yield lower resolution, but that is a physical limitation of the cameras placement. Figure 2.15 shows the same region on one of the original camera streams, directly after the debayering process. By comparing these two images, we see that the original image quality has been preserved well. The image appears smoother than the original due to the cubic interpolation in the stitching module, but most details can still be discerned.

Figure 2.14 also shows the efficiency of the dynamic seam, exposure synchronization and seam blending. We mentioned that we can sometimes see the color difference between each

	30 fps		40 fps		50 fps	
	mean	max/sec	mean	max/sec	mean	max/sec
Without HDR	0.009%	3.33%	0.009%	2.5%	0.017%	6%
With HDR	2.41%	20%	1.977%	20%	2.028%	24%

Table 2.4: Summary of drop rates at multiple frame rates. This is the camera reader frame rate, the output frame rate will be halved for the HDR mode. Each recording was between 40 and 200 minutes long. Note that auto exposure was performed every 10 seconds which always causes 3 frames to drop in HDR mode, regardless of frame rate.

image, but that this goes away when zooming into the region. We intend for this panorama to be used primarily for virtual viewing, as we will present in chapter 5, where the entire panorama will rarely be displayed. Therefore, we deem it more important that the seam looks natural when zoomed in.

In summary, we see great visual improvement from the old Bagadus panorama. Our generated panorama has a greater vertical resolution, better preservation of image quality and a more compact format as a result of not bloating pixels. We see that the dynamic seam is very difficult to detect, and produces no visible artifacts. We did see some color differences between the images when viewing the full panorama, but this is hidden when only displaying smaller regions.

2.16 Performance

In order for the pipeline to run in real-time, each module must be able to consistently stay below the real-time threshold. In this system, we target two real-time deadlines depending on the recording configuration. The entire system must be able to handle 50 fps, i.e., 20ms real-time threshold, without HDR. When HDR mode is toggled, this reduces the frame rate to 25, and the threshold is increased to 40ms. However, this also increases the data rate for most modules.

It is challenging to properly time the recording stages of the pipeline, as the camera reader module is restricted by the cameras frame rate. We can prove that is real-time from the low rate of dropped frames, which occurs when one of the camera streams is unable to deliver a frame within a real-time deadline in the frame synchronization module. Remember that a frame drop occurs whenever *any* of the streams drop a frame, meaning that the drop rate for each individual camera stream is much lower. A summary of frame drops is given in table 2.4. Here, we show the mean percentage of frames dropped by the frame synchronization module. This drop rate can vary depending on the recording, and the drops are usually associated with auto exposure updates. However, we see drop rates of less than 0.2% consistently through long recordings. However, this is not necessarily an important number if all drops occur over short spans of time, e.g., dropping 20 frames in a row. Therefore, table 2.4 also shows the maximum percentage of frames that were dropped within the span of 1 second. If we look at the HDR recording at 50 fps, meaning that the frame synchronization module was operating at 25 fps, we see that at some point in the recording 24%, or 6 out of 25, frames in one second were duplicates of another frame. This is something an observant user may notice, but as long as it occurs infrequently

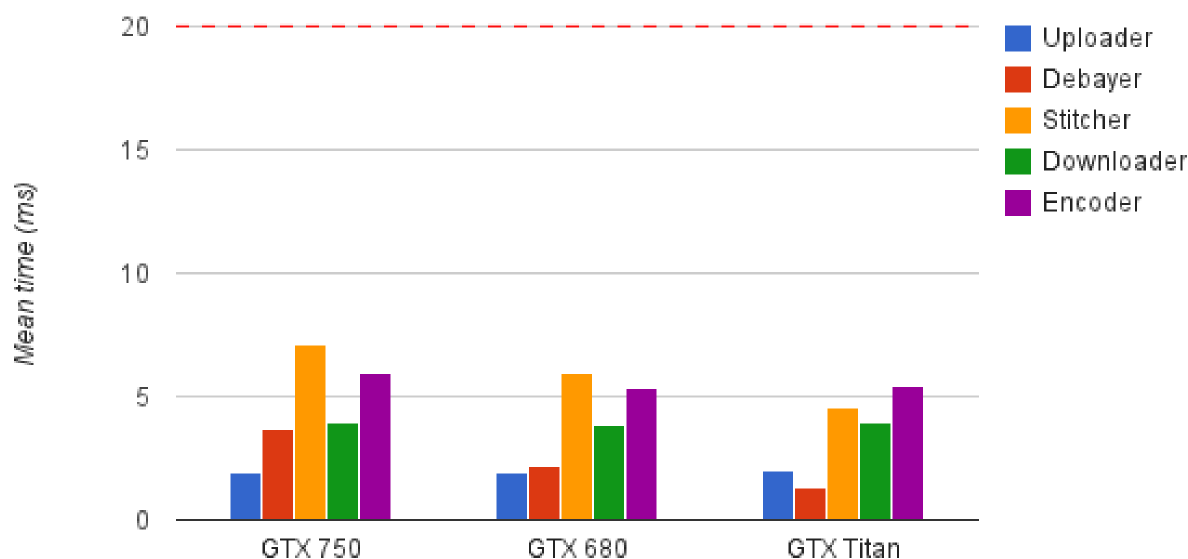


Figure 2.16: Overall pipeline performance, without HDR mode. Note that our prototype installation uses a GTX680 GPU.

it will not diminish the viewing experience. Given the numbers of the non-hdr recordings, the user will likely never notice due to the high frame rate.

We see that the numbers are consistently higher for HDR recordings, due to the intentional dropping of frames during auto exposure. With the presented frame rates, the minimum drop percentages are 2%, 1.5% and 1.2%. With a lower frame rate, we must drop more frames on average to perform exposure updates. We also see that, at maximum fps, there are more additional frame drops and we saw that it very frequently dropped 4 frames instead of 3 during the exposure updating, along with a couple of extra frames. This is likely due to the strict real-time requirements of alternating the exposure, as we more often fail to configure the camera fast enough. However, overall, we believe these drop rates to be acceptable and not visually apparent to the user.

We saw in section 2.8 that it takes less than a millisecond to transfer a frame across the high speed interconnect link, plus some time for signaling. This is doubled with HDR active, but so is the real-time limit. The frame synchronization module is also a very quick step, primarily slowed by having to copy the frame out of the Dolphin modules buffer and waiting for the mutex associated with depositing a frame. However, we saw that these times were overall short, and would have caused a higher rate of dropped frames if operating too slow.

It is perhaps more important that each processing module is below the real-time limit, since a failure to do so will cause frames being skipped. In figure 2.16, we see the mean execution time of each module over 1000 frames. This test was performed on a machine with the same hardware specifications as our prototype system, though with additional GPUs for comparison. We can see that every module is well within the real-time requirements of 20ms, with all evaluated cards. It is worth noting that it is difficult to measure the encoding module accurately because it uses multiple (default 18) threads, and may keep residual data across frames. Even though its execution time here is measured at 5.43ms, the encoder will still perform work outside of this. If we continuously encode as fast as possible we will see encoding times of up to 9ms. Perhaps a better measurement for showing the relative workload of the encoder is the CPU

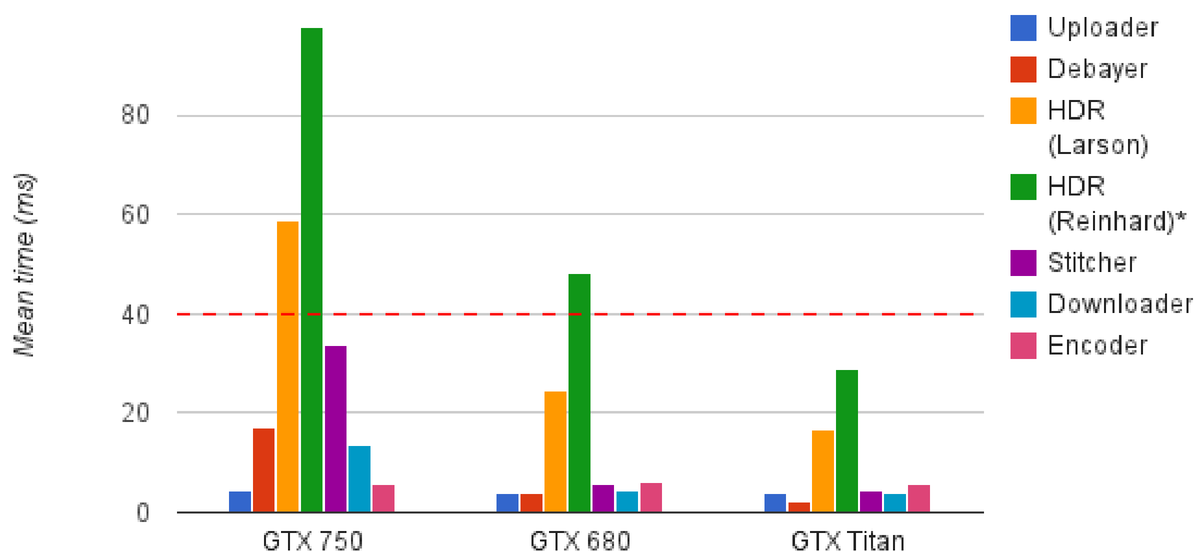


Figure 2.17: Overall pipeline performance, with HDR mode. Note that our prototype installation uses a GTX680 GPU.
* Reinhard's algorithm execution time belongs to a different program execution, remaining times are from the same execution.

utilization percentage, which is normally around 20% of maximum. The encoder module, as mentioned in section 2.14, does not have a constant execution time, and it varies somewhat with the scene. The program executions used for the measurements in this section had a dynamic scene, with a lot of moving players, but little noise and other negative effects.

The overall pipeline performance with HDR active is shown in figure 2.17. Here, we see that only the GTX Titan GPU is able to execute Reinhard's algorithm in real-time. The GTX 680 can execute Reinhard's algorithm at 20 frames per second, as it is barely below 50ms per frame. However, it only has about a millisecond of margin, which is likely a bit too close for stable recordings. The current default HDR algorithm, Larson's, can be included in the pipeline, however, as it is well within real-time requirements with the GTX 680. Neither of the HDR algorithms are real-time with a GTX 750, and we see that this card performs worse kernels scheduling than the other two by slowing the other modules significantly. We saw in figure 2.16 that the stitching module spent 7ms on the GTX 750, which is increased to 33.8ms when adding the HDR module. This has to do with how the modules are timed, as other kernels will be actively occupying the card within the timing window. This essentially means that the timing of the slowest module is especially high, as it will likely include the execution of all other modules. We see that this is also the case for the GTX 680 to a lesser extent, while the Titan card seems to be able to distribute resources more efficiently. This makes sense, as this is primarily a card designed for general purpose GPU programming and not for graphics rendering.

The effect of this is shown in figure 2.18, where we show the execution time of each of the GPU modules when executing alone. This was achieved by delaying all modules until the previous had completed, causing sequential execution. We see that both HDR algorithms are now nearly below the real-time threshold. Again, we see nearly no difference in any of the modules on the Titan card, meaning that the card performs extremely good scheduling. When we execute only a single module, there will typically be gaps between executions for performing

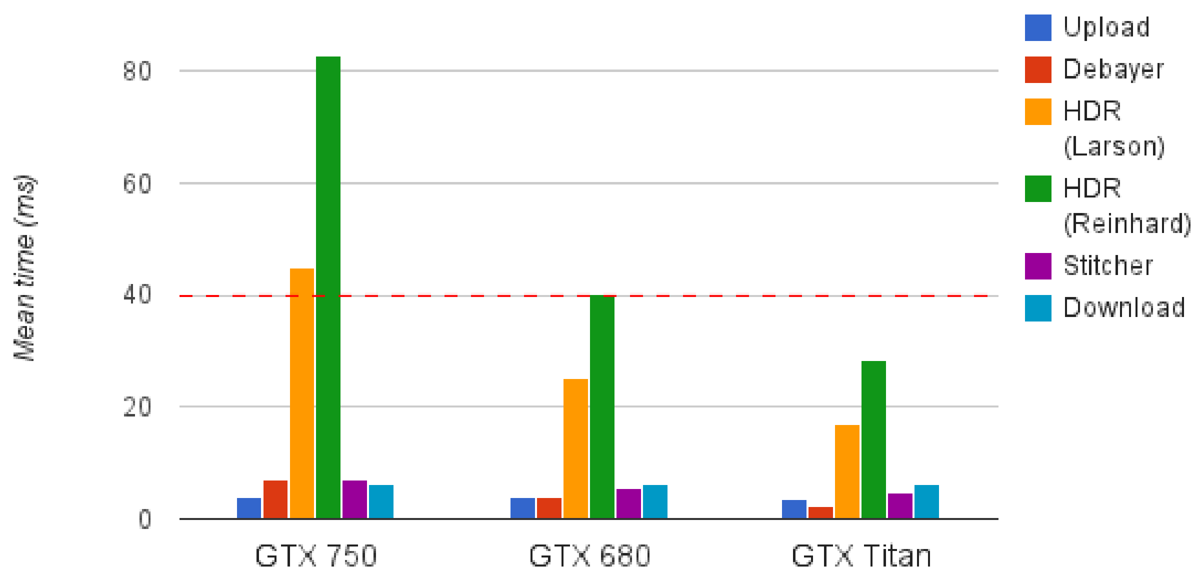


Figure 2.18: GPU single execution, where each module executes alone, without interference from other modules.

some CPU operations, synchronization or launching new kernels. These gaps can be filled by other modules if scheduled efficiently.

The upload and download modules seem fairly unaffected by changing GPU, and we see that these can execute efficiently in parallel with the other modules. We also see that the downloader is consistently slower than the uploader, even though the bandwidth requirements are only slightly higher (stated in section 2.10).

In summary, we see that the panorama pipeline can operate in real-time at 25 or 50 fps depending on whether HDR is active. The frame synchronization module will occasionally duplicate frames, primarily due to exposure synchronization, but these drop rates are well below visible limits. The processing modules of the pipeline executes below real-time requirements for 25 fps when using Larson's HDR algorithm, and with better hardware it should be possible with Reinhard's algorithm. When executing the pipeline without HDR, we see that we are well below the real-time requirements for 50 fps. This means that we can potentially scale the system further, by adding additional modules or additional cameras. However, we see that in order to scale the system with HDR active we would need to perform further distribution or optimizations.

2.17 Discussion and Future Work

The pipeline, as it is presented, has been created from a series of design decisions. There are several other choices that could have been made, or extensions to the system that can be added in the future.

2.17.1 Module placement

The order and placement of the modules may impact the system as whole. We elected to perform all processing on a single machine, but we could also have performed some steps on each recording machine for a more distributed system. The debayering step would be the most logical to perform on the recording machine, but this would also significantly increases the required bandwidth for transferring the raw images, as they would then contain all color channels. The CPUs of the recording machines are not particularly stressed from the video recording, but they require fast context switches. We saw that when these machines were stressing the CPU, we would begin to lose frames. This was especially the case with HDR, as the thread responsible for re-setting the exposure time, between each frame, has very short real-time requirements. Additionally, these machines have very limited GPUs in our prototype installation.

We saw that the HDR module still struggles to remain real-time with the best performing algorithm. This could be performed on the recording machines as well, if we were to install better GPUs in our prototype. Another possibility is to perform the HDR step after the panorama has been stitched, as the stitching module reduces the image size to $\frac{4096\times1680\times100}{2040\times1080\times5} = 62.5\%$ of the original size. However, this would mean performing the stitching module at twice the frame rate. More problematic is the fact that the stitching module would create very strange artifacts, as the two overlapping images could result in widely different seams. The stitching module would struggle in very overexposed or underexposed areas. Alternatively, we could create a more hybrid variant of the stitching module that handles two frames at the same time. Using both frames as an input, it could create its pathfinding cost based on both sets of images and thereby create the same seam in both the low and the high exposure panorama. This could be worth investigating in the future.

2.17.2 Module separation

The modules themselves could also be split into other parallel steps. The stitching module contains several sequential steps that could be pipelined, as detailed in chapter 4. We believe that only the initial image projections would likely benefit from being detached. This is due to the synchronization required between projecting each image and calculate the resulting seams based on the projected images. Both the bayer converter and the HDR module contains multiple passes, that could be pipelined.

Whether it is a benefit to split a module varies a bit depending on the system, however, as we saw that the kernels are typically scheduled quite sequentially anyways. Most commonly, the benefit occurs when a single slow step that does not fully occupy the card depends on the information from a previous step, such as the dynamic seam creation. This step will be slow regardless, and it would be preferable if it was scheduled as soon as possible since other kernels could possibly run concurrently.

2.17.3 HDR exposure times

HDR mode was added fairly recently, and we have not yet been able to determine the optimal way to set the low and high exposure values. Our current approach is highly dependent on the initial calibration, i.e., that the lighting conditions at the very beginning of the recording are indicative of the remaining recording, and we are not sure about the optimal ratio between the two values.

We mentioned that it takes a very long time for the pilot camera to change its region of interest, making it impossible to use two different ROIs for auto exposure without incredible frame drops. Another approach that could be worth looking into is to use two pilot cameras for this, where one camera controls the low exposure and another camera controls the high exposure. This way, we would avoid the issue of swapping between two regions, and we could provide better live auto exposure configurations.

2.17.4 Automatic recording system

The system, in its current state, is operated manually for each recording. However, we are also in the process of implementing a daemon functionality where the pipeline is always running, ready to record based on external signaling. In the future, this can be fully implemented by using a web crawler to determine when there will be activity on the soccer pitch, either matches or practise sessions, and automatically begin recording.

2.17.5 Further scalability

In the presented pipeline, we have only distributed the recording step of the pipeline. This could be improved in the future, if new modules need to be added or when adding more cameras. We also saw that the HDR module was unable to execute all implemented algorithms in real-time, which could be fixed by distributing the workload to other machines. The high speed Dolphin cluster, used for transferring the raw frames between machines, still has a lot of theoretical bandwidth left. This means that we could spread the processing onto multiple processing machines.

The Dolphin transfer modules currently transfer frames into CPU memory, but we saw in section 2.8 that it is also possible to transfer directly into GPU memory. We see little or no overhead when transferring frames into GPU memory, as could be seen in figure 2.8. This way, the frames do not need to be uploaded afterwards, and we can save some memcpy operations. This means that we must also perform frame synchronization as a part of the Dolphin module, by polling the state of multiple buffers across the cluster. The latency of short messages is so low that this is highly feasible. Once the frames are synchronized and joined into sets, without requiring them to be located on the same machine, we can more easily distribute the workload. For example, we could join one set of frames, yet send the individual frames to different machines for processing. In this way, we can avoid wasting bandwidth for transferring frames back and forth for distribution.

When the system executes without HDR, we saw that there is still a lot of remaining time on the processing machine. This means that we could likely add multiple cameras to extend the system, improving the resolution of the generated panorama. In section 2.8 we also saw that the high speed network theoretically supports 24 concurrent cameras at 50 fps.

2.17.6 DASH streaming - video transcoding

Another important scalability step is the delivery of the video. We have shown that our system can produce high quality video, segmented for easy live streaming. However, in order to truly support effective live streaming, we need to be able to provide a varying bit rate depending on the users network connection. We will discuss this in more detail in chapter 5, but it is

important to note that the panorama is so large that it may be challenging to transport it over many network connections. By adding a transcoding step to our pipeline, encoding the video into multiple bit rates and resolutions, we can extend the delivery systems with full MPEG-DASH [60] functionality. This is also something that can be easily distributed, by using another processing machine for transcoding the video.

We could also utilize the NVENC hardware encoder, which would reduce a lot of overhead from our pipeline. This hardware encoder is located on the GPU, close to the data outputted by the stitching module. This means that we would not need to download the stitched panorama into CPU memory for encoding, instead only downloading the much smaller encoded video in order to write it to disk. This means significantly less data to transfer across the PCIe link. Most importantly, however, is that it does not take away resources from the existing application, as the hardware encoding is performed in a dedicated chip, with minimal overhead.

2.18 Summary

In this chapter, we have detailed the panorama recording pipeline. We began the chapter by discussing the motivation behind re-designing the system, pointing to the inadequate solutions currently available and the low visual quality of the old Bagadus system. Then, we gave an overview of the new pipeline features and the overall system design. We explained that the pipeline is composed of several modules that operate on a single frame, which must all be real-time. We presented the pipeline as a distributed system, with modules related to the camera recording executing on different machines. The system uses very simple asynchronous signals to communicate, but can be controlled from a single machine. Next, we discussed the CUDA framework, as this is used by several modules. We explained how programming with on the GPU differs from programming x86 architectures, and detailed several performance pitfalls that should be avoided.

Then, we began discussing the individual modules of the pipeline. We presented a new camera configuration, designed to avoid a lot of the challenges in the old Bagadus system. However, by adding HDR to the pipeline we also introduced new challenges related to the capture of video frames. We saw that we are able to transfer the raw frames across a high speed interconnect network, with a lot of potential for further upscaling. There is still a lot of remaining bandwidth on this network, and the possibility of transferring the data directly to GPU. We did not go into great detail on the debayering and stitching modules, as these will be covered in chapter 3 and 4. However, we saw that the choice of HDR algorithm was very important for the performance of the entire pipeline. This module was able to show details in the entire panorama, despite strong shadows present in some regions. However, we also saw that this was computationally expensive and with varying visual quality. Then, we discussed the challenges of encoding the video into playable video segments, and saw that we could produce a very high quality stream at a much lower file size than the old Bagadus system.

Next, we evaluated the visual performance of the panorama pipeline. We saw major improvements from the old Bagadus system, and we saw very high preservation of the original image quality. The performance of the pipeline was far below real-time limits, when executing without HDR, with nearly no frames dropped. We saw that the visually best HDR algorithm was not real-time, but that the second best alternative was well below real-time requirements. Next, we discussed several alternative design choices that could have been made, along with

suggestions for future work.

Overall, we saw that the recording pipeline can generate a 4096×1080 panorama in real-time, with several different lighting conditions. This generated panorama has been found to be well suited for virtual viewing solutions, given its relative high quality, low file size and lack of negative visual artifacts.

In the next chapter, we will see how best to implement the debayering module. We will see that a lot of different algorithms can be used for this, which impacts the overall performance of the recording pipeline we have presented here.

Chapter 3

Bayer Demosaicking

In this chapter, we will go into more detail regarding the research put into the implementation of the Bayer converter module in the recording pipeline described in chapter 2. We will first explain the use of a *Bayer filter* and explain the process of *Bayer demosaicking*. Then, we will go on to explain some of the existing algorithms and show how a selection of implemented algorithms fared with regard to quality and performance on a GPU. For this, we will use the *Kodak Lossless True Color Image Suite* [61], an image dataset used in most Bayer demosaicking papers, by first artificially enforcing the Bayer pattern. Finally, we will evaluate the implementations on real images from our Bagadus scenario, investigating which implementation is best suited for our purposes.

3.1 Bayer Filter

Digital cameras today will generally utilize a single CCD or CMOS image sensor per pixel, each responsible for determining the amount of light (number of photons) registered during the exposure of the image. In order to find accurate color information, the light first goes through a wavelength filter. This filter will remove all light outside a specific range, e.g., 635–$700nm$ for the color red. In order to capture a full RGB image, one must capture three separate images with three different filters; red, green and blue. However, most cameras perform a single pass [62], recording only one color per pixel, filtered through a *Color Filter Array* (CFA). The Bayer filter is perhaps the most common filter used today. An example of this CFA is shown in figure 3.1.

In the Bayer pattern, each pixel can only directly determine either red, green or blue. The remaining color information must be extracted from the surrounding pixels. This process is

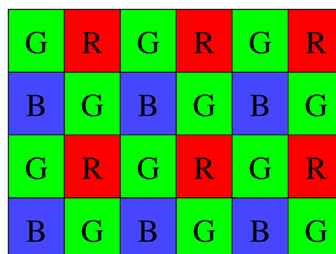


Figure 3.1: Example of a Bayer pattern

$$\begin{pmatrix} Y \\ U-128 \\ V-128 \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.331 & 0.500 \\ 0.500 & -0.419 & -0.081 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

Figure 3.2: Conversion matrix used from RGB to YUV [63]

what is referred to as *Bayer demosaicking*, or *debayering*, accomplished through the use of various interpolation techniques.

Green is sampled twice as often as red/blue due to the fact that our eyes are more sensitive to green. More specifically, our eyes are most sensitive to rapid changes in luminosity, i.e., the intensity of the pixel. Green is the primary component in representing luminosity in multiple color spaces, including YUV which is used in our video pipeline. In the YUV color space, green is significant in all three components as can be seen in the conversion matrix in figure 3.2, where the absolute value of the center column represents the influence of the green channel. Red and blue make up the primary components of the chromatic channels, each representing the red and blue difference compared to the luminosity.

3.2 Motivation

The cameras used in the Bagadus pipeline can perform hardware conversion from raw Bayer into the YUY2 pixel format, a variation on YUV422 with two bytes per pixel, see figure 3.3.

This hardware conversion produces a decent visual result, but is impossible to do at high frame rates. The limiting factor here is the gigabit ethernet link between the camera and the recording machine, which cannot exceed 1Gb/s. The maximum frame rate can be determined by equation 3.1.

$$fps = \frac{1Gb/s}{framesize} = \frac{1 * 1024^3 b}{1s * 2040 * 1080 * 16b} = 30.46 \tag{3.1}$$

The camera driver isn't able to fully optimize the ethernet link and will deliver only the raw Bayer pattern at frame rates above 25. We believe this frame rate to be slightly too low for soccer, but we have earlier discussed the use of high dynamic range video in the recording pipeline. In order to utilize HDR video we must read twice the frame rate from the camera as the delivered result. This would yield a frame rate of 12.5, certainly well below acceptable standards. In order to read the maximum frame rate of 50 from the cameras, the debayer module was added to the pipeline.

Multiple methods for Bayer demosaicking exist today, providing varying level of quality. Common to all available algorithms is some loss of detail, as well as visual artifacts as a result of incorrect interpolation. These are detailed in section 3.3. Most real-time video systems

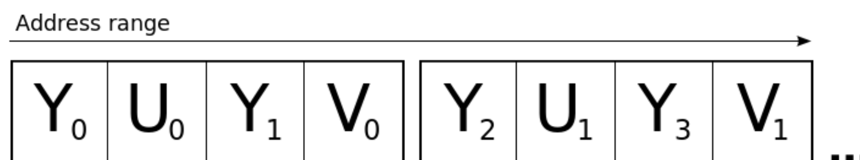


Figure 3.3: YUY2/YUYV byte representation

utilize fairly simplistic algorithms. However, in chapter 5, we detail a system for zooming in the generated panoramic video. This means that any visual artifacts created from Bayer demosaicking will become more obvious. Given that this is a real-time system we cannot necessarily rely on the most computationally heavy algorithms, especially as this is merely one module in a larger pipeline with shared resources. Therefore, we evaluate the trade-off between quality and execution time.

Most Bayer demosaicking algorithms are solely directed toward single images, without the speed requirement of real-time video. Most video systems use fairly basic algorithms and we wish to explore how more complex algorithms fare when implemented on a high end GPU. The GPU often has very different criteria for how well two algorithms perform compared to each other, an area that has not been well explored. Therefore, this chapter goes into greater detail than perhaps necessary for the rest of the thesis.

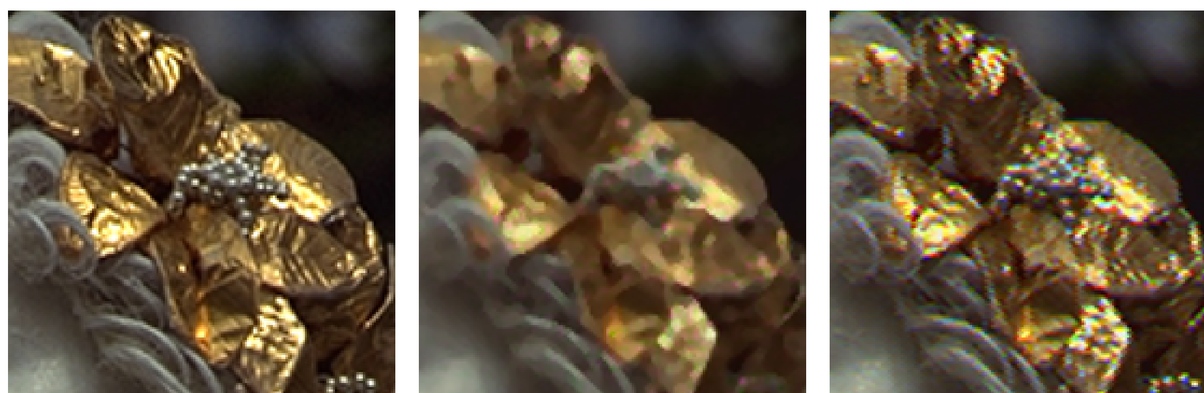
The debayering process could be performed multiple places in our recording pipeline. We chose to place the module on the GPU of the processing machine for several reasons. The nature of debayering is highly parallelizable, well suited for the GPU architecture. It also requires memory access patterns, i.e., 2D localized accesses, that are far more inefficient on a CPU. The CPU of the processing machine is also highly stressed from performing video encoding, while the recording machines have very high real-time thread scheduling requirement when reading from several cameras. An important reason to perform the operation on the processing machine is that transferring raw video frames between machines requires a very high bandwidth. The Bayer pattern only requires one byte per pixel, while the resulting YUV is saved with four bytes per pixel. This would therefore quadruple the required bandwidth for transferring the raw frames. Limiting the output to three bytes per pixel would reduce the efficiency of both this module and the following module, as 32 bits per pixel is far more efficient to work with on CUDA. We could also perform the downsampling of the chroma channels in this module, delivering YUV422 instead of YUV444. This would only double the required bandwidth, but may cause complications in later modules. The debayering process should not be a very computationally complex operation, and should only be a minor part of the overall pipeline. If the system is scaled further, e.g., by adding more cameras or add new modules, we may need to perform a new evaluation on the optimal placement of this module.

3.3 Interpolation and Demosaicking Artifacts

Interpolation is the process of using existing samples of known values to create an estimation at a position more accurate than the original sample rate. In the Bayer pattern, the red and blue color channels are sampled once per four pixels, and the remaining three pixels must use an estimation based on surrounding color samples. Similarly, the green channel is sampled only at every second pixel.

Demosaicking artifacts are the result of incorrect interpolation or limitations of the demosaicking process. Many regions are particularly difficult to reconstruct, due to a high level details or rapid changes in luminosity within the region. Most algorithms will either cause smoothing/blurring of these regions or create rapid unnatural changes in color, as shown in figure 3.4.

Interpolating along edges can cause *zippering artifacts* by incorrectly interpolating pixels across an edge, seen in figure 3.5. In these cases, it is generally favorable to ignore some



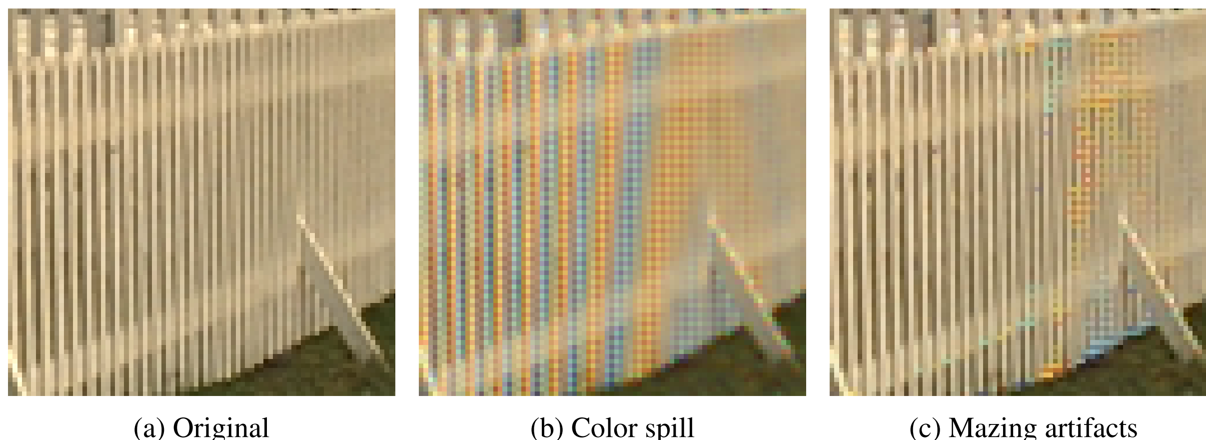
(a) Original (b) Loss of detail from blurring (c) Rapid changes in color

Figure 3.4: Loss of detail from Bayer demosaicking

Figure 3.5: Images showing zippering artifacts along horizontal, vertical and diagonal edges

samples and interpolate along the edge instead. In this chapter, we may refer to edges as rapid changes in luminosity, as not all rapid transitions within a single RGB color channel constitute the presence of an edge.

There are several different forms of *false color artifacts* that frequently occur when demosaicking Bayer images. Single pixels with significantly different hue, i.e., the relation between two color channels within a pixel, compared to its neighbouring region can occur in areas of great detail when all channels are interpolated independently. When this occurs consistently due to a localized pattern, we may refer to it as *color spill*. This is especially visible in white surfaces and edges because the inter-channel dependency in the RGB colorspace increases along with the intensity. This can be particularly difficult to remove and is often easy to spot without the need to zoom. *Mazing artifacts* can occur when algorithms fail at selecting the optimal direction in direction-oriented algorithms. Mazing artifacts are very distinct and can often look more unnatural than regular color spill, despite having fewer “incorrectly” interpolated pixels. Our eyes are good at detecting straight lines, and sudden breaks in alignments is very distinctive. These effects are shown in figure 3.6.



(a) Original (b) Color spill (c) Mazing artifacts

Figure 3.6: Images showing how a consistent pattern can create multiple false color artifacts

In the next two sections, we will go through some of the most common Bayer demosaicking algorithms available and mention how they handle or avoid these artifacts. There are many small variations on these algorithms and they all borrow heavily from each other, but we attempt to show the various techniques that can be utilized.

3.4 Non-Adaptive Demosaicking Algorithms

Image demosaicking algorithms are often separated into two major groups, adaptive and non-adaptive. Non-adaptive algorithms will treat all pixels equally, while adaptive algorithms can treat certain pixels differently based on the surrounding area. Adaptive algorithms generally use thresholding values to determine whether to avoid some pixels, while non-adaptive may apply weights, i.e. value individual pixels more/less, before averaging. Due to pixel weighting, this distinction can be slightly fluid, but we have elected to categorize all algorithms that never discard individual samples as non-adaptive.

Simple non-adaptive algorithms suffer significantly from zippering artifacts, as pixels are interpolated across edges instead of along. This generally also introduces more false colors artifacts. Similar to most non-adaptive algorithms, however, is that they are quite fast to com-

R1	G2	R3	G4	R5
G6	B7	G8	B9	G10
R11	G12	R13	G14	R15
G16	B17	G18	B19	G20
R21	G22	R23	G24	R25

Figure 3.7: Bayer Pattern centered on a red pixel

pute and are very easy to parallelize. The fact that every pixel is treated identical is particularly relevant on a GPU architecture, as it allows for zero branching and consistent memory access patterns.

The most simplistic demosaicking algorithm is *Nearest neighbour*, using no pixel averaging. As there is never one single closest color, unless we are working at sub-pixel accuracy which is beyond the scope of this thesis, it will normally select a consistent direction to use. For example, in our implementation, we choose to always select the pixel directly left, up or upper left diagonal, in this preferred order. The result has extreme zippering artifacts and false colors, as well as loss of detail. However, it is the fastest algorithm.

Bilinear interpolation uses the average value of the two or four nearest neighbour pixels of the specific color. For example, looking at figure 3.7, we can determine some missing values by:

$$G13 = \frac{G8 + G12 + G14 + G18}{4} \qquad B13 = \frac{B7 + B9 + B17 + B19}{4}$$
$$B8 = \frac{B7 + B9}{2} \qquad R8 = \frac{R3 + R13}{2}$$

Bilinear interpolation produces a quite smooth and good result in homogeneous areas, i.e., small color differences, but often fails in areas of great detail. It is generally considered the cheapest of the acceptable algorithms, often used in real-time video systems due to its low complexity. It produces significant zippering and false colors.

Bicubic interpolation uses the weighted average value of the 16 nearest neighbouring samples. The samples are weighted based on their spatial distance to the interpolated pixel. This is generally known to produce smoother images than bilinear, but is not as commonly used for Bayer demosaicking, as it generally produces the same artifacts associated with bilinear interpolation, but at a greater computational cost.

Many of the false color artifacts produced by the above mentioned algorithms are the result of too abrupt changes in hue between pixels. David R. Cok explains in [64] that there is a constant inter-channel relation between the color differences in an image. This means that the difference between two color channels per pixel changes gradually, not suddenly. *Smooth hue transition* uses the green channel as a baseline for interpolating the red and blue channel. Because green is sampled twice as frequent, it is statistically the most accurately interpolated channel. The green channel can be interpolated using different algorithms, but generally bilinear or cubic is used. Once green has been interpolated for all pixels, red and blue channels are interpolated as such (again, see figure 3.7):

$$B13 = \frac{G13}{4}\left(\frac{B7}{G7} + \frac{B9}{G9} + \frac{B17}{G17} + \frac{B19}{G19}\right) \tag{3.2}$$

$$B8 = \frac{G8}{2}\left(\frac{B7}{G7} + \frac{B9}{G9}\right) \qquad R8 = \frac{G8}{2}\left(\frac{R3}{G3} + \frac{R13}{G13}\right) \tag{3.3}$$

Smooth hue transition eliminates many small false color artifacts compared to bilinear and cubic interpolation, but still struggles with edges and color spill. The algorithm is slowed by the extensive use of division, and may also introduce problems with division by zero.

Another approach is to use the chrominance components to provide a better estimate for the luminescence of a pixel. Malvar *et al.* [65] assume that an edge has a stronger luminescence

component than chrominance component. Their proposed *gradient corrected bilinear interpolation* performs regular bilinear interpolation to interpolate missing channels, but uses the color information already present in the pixel to correct the result. For example, given a red pixel, it performs bilinear interpolation on the other nearest four red samples. If this value is significantly different from the original red sample, there is likely a large change in luminosity present in this pixel. Referring again to figure 3.7 we can calculate the green component in pixel 13 as such:

$$\Delta r = R13 - \frac{R3 + R11 + R15 + R23}{4} \tag{3.4}$$

$$G13 = \frac{G8 + G12 + G14 + G18}{4} + \alpha \Delta r \tag{3.5}$$

The gain parameter α determines the intensity of the correction, found to be best at $\alpha = \frac{1}{2}$ in [65]. The other channels are interpolated in a similar fashion, but we refer to [65] for a more detailed explanation of these. This algorithm generally preserves significantly more details than the previous non-adaptive algorithms, due to the increased preservation of luminescence. It still suffers, to a less extent, from zippering artifacts and produces the same amount of color spill. Despite requiring more computations and pixel lookups than smooth hue transition, it may still prove move efficient as it does not need an initial pass to interpolate the green channel. This algorithm is one of the most used today.

Weighted directional gradients uses the same gradient correction principle, but instead of performing a single bilinear interpolation, it decomposes the interpolation into finite directions. This algorithm is presented fully in [66]. A contributing value is calculated for all four directions, or eight by including diagonal directions, which when combined and averaged would produce the same interpolation as the previous presented algorithm [65]. Each direction is additionally weighted based on the spatial correlation between pixels in the given direction, its gradient. The gradient of a direction can be defined as the absolute difference of the original pixel compared to a pixel of equal color in the given direction. The inverse gradient $\frac{1}{gradient}$ gives us the correct weight. Again looking at figure 3.7, we find the contribution of the right direction for interpolating green at pixel 7 as such:

$$G_{right} = G8 + \frac{B7 - B9}{2} \tag{3.6}$$

The inverse gradient α is found as such (note that 1 is added to the denominator to avoid division by zero in homogeneous areas, it has no impact because it is done equally to all weights [67]):

$$\alpha_{right} = \frac{1}{1 + |G6 - G8| + |G8 - G10| + |B7 - B9| + \frac{|G2 - G4| + |G12 - G14|}{2}} \tag{3.7}$$

The contribution and weight of the other directions are found in the same fashion. The final green value can be computed by:

$$G7 = \frac{\alpha_{left} G_{left} + \alpha_{right} G_{right} + \alpha_{up} G_{up} + \alpha_{down} G_{down}}{\alpha_{left} + \alpha_{right} + \alpha_{up} + \alpha_{down}} \tag{3.8}$$

We will not go into further details on how the red and blue channel are interpolated, as it uses the same base principles, but refer to [66]. This algorithm generally produces few artifacts, while also doing a good job of preserving the luminosity in detailed areas. It also avoids zippering artifacts very well. Color spill can still occur in very difficult areas, though significantly less than the aforementioned algorithms. It has fairly high complexity, especially in the number of pixel lookups required, and requires three passes (though it can be simplified to use two).

3.5 Adaptive Demosaicking Algorithms

The constant hue relation from smooth hue transition established previously, as well as its use of green as a baseline, is a concept used by many adaptive algorithms. Because of this, many algorithms are composed of the same two passes, the green channel and the red/blue channels. In [68] it is assumed that the difference between the red/green and blue/green channel varies at a rate lower than the sampling frequency. Therefore, as long as the green channel is well interpolated, we should be able to reconstruct the red and blue channel based on their correlation to the green channel. For brevity, we will not go into great detail regarding the blue and red channels in the remainder of this section. When mentioning a constant hue approach we are referring to this method, showing interpolating of blue at pixel 13 in figure 3.7 with all green values already interpolated:

$$B13 = G13 + \frac{B7 + B9 + B17 + B19 - G7 - G9 - G17 - G19}{4} \tag{3.9}$$

Edge sensing interpolation is a simple extension of bilinear interpolation. It uses threshold values to determine if a given pixel is located on a vertical or horizontal edge. The assumption is that if there is a significant change between two of the nearest neighbour pixels, then there is likely an edge on/between them. In this case, the algorithm opts not to use these pixels for interpolation. More specifically, the edge sensing algorithm determines the color change in the horizontal and vertical directions. If one of these is greater than a threshold, the direction with the smallest change is chosen for interpolation. This is shown in algorithm 3.1.

Algorithm 3.1 Edge sensing interpolation of green channel, based on figure 3.7

$\Delta H \leftarrow |G12 - G14|$
$\Delta V \leftarrow |G8 - G18|$
if $\Delta H > threshold \vee \Delta V > threshold$ **then**
  if $\Delta H < \Delta V$ **then**
    $G13 \leftarrow \frac{G12+G14}{2}$
  else
    $G13 \leftarrow \frac{G8+G18}{2}$
  end if
else
  $G13 \leftarrow \frac{G8+G12+G14+G18}{4}$
end if

In our implementation, it will only choose directions for the green channel, which proves sufficient for removing vertical and horizontal zippering artifacts, but instead often creates consistent lines along edges as the red and blue channel is erroneously interpolated, and no form of inter-channel correlation is utilized. Because this algorithm only takes two directions into account, it also often fails along diagonal lines and may create mazing artifacts to a small extent in situations where the incorrect direction is selected.

Hamilton and Adams improve upon the edge sensing method in [69] by using a second order gradient to determine the presence of an edge. *Edge directed interpolation* determines the presence of an edge by the laplacian, i.e., the divergence of the gradient between enclosing pixels, of the green channel and the gradient of the red or blue channel. The horizontal gradient

value of pixel 13 in figure 3.7 is found by:

$$Grad13_H = |G12 - G14| + |2R13 - R11 - R15| \tag{3.10}$$

The vertical gradient is found through the same method, and the smallest of the two gradients determines the direction of interpolation. The gradient correction component from [65] is applied to sharpen the image and reduce interpolation artifacts. The blue and red channels can be interpolated in a similar fashion, using diagonal gradients, but in this thesis we found an overall better, and faster, result by using the simpler constant hue approach as done in [68].

Variable number of gradients, as it is presented in [70], is very similar to the aforementioned weighted directional gradients. Gradient-based weights are generated in a similar way, but it also uses a dynamic gradient threshold to filter out the worst directions altogether. It has quite high complexity, as it uses all eight directions (N, NE, E, SE, S, SW, W, NW), but is generally implemented in a single pass. Variable number of gradients typically produces good results, especially at predicting edges, but performs overall very similar to the non-adaptive variant.

The *adaptive homogeneity-directed demosaicking* algorithm, presented by Hirakawa in [68], is one of the most effective demosaicking algorithms used today. The edge-directed approach presented by Hamilton and Adams is improved upon by using the homogeneity within a localized area to determine the best approach. The mazing artifacts associated with edge directed algorithms occur when straight lines are suddenly broken, as a few pixels are interpolated along a different direction than the surrounding pixels. By imposing a local homogeneity, these situations can be significantly limited. The homogeneity of a pixel is presented by Hirakawa as the number of neighbouring pixels with similar color. The algorithm will first interpolate the green channel twice, once using only the vertical interpolation method used in the aforementioned edge-directed algorithm and once using the horizontal method. For each pixel, in both images, the local homogeneity is calculated in the CIELAB color space [71]. The final green value of a pixel is selected from one of the images based on whichever image has the highest homogeneity for that pixel. This method produces very good results and minimal artifacts. It especially excels at removing color spill. This comes at the cost of very high complexity. The algorithm also uses significant post-processing artifact removal, see section 3.6.

The homogeneity based approach presented by Hirakawa is not well suited for real-time video and our system. A simplification to this approach is presented in [72], where the decision of what direction to select is based solely on the two green interpolations. The optimal direction is based on the smallest gradient change in the difference between the green and the red/blue channels within a localized area, e.g., a 5×5 pixel grid. This method skips several costly steps of interpolating the red/blue channel multiple times, but is more likely to cause visual artifacts as a result of imposing the homogeneity. Another variation is to look at the homogeneity of a point as the number of neighbouring points that share the same preferred directional interpolation, as well as the intensity of that preference. We propose a new algorithm, *homogeneous edge-directed interpolation*, that is very similar to that of [72] and based on this principle. This is a three pass algorithm, which first calculates the horizontal and vertical gradients and saves the preference towards one direction in a homogeneity map. The second pass will interpolate the green channel by selecting either horizontal or vertical direction, based on the local homogeneity, or both directions if there is little difference. The pixels used to determine the homogeneity are the nine spatially closest red/blue pixels, forming a diamond pattern 4 pixels wide and tall. The third pass interpolates the red and blue channel in a constant hue approach. Pseudo code for this method is given in algorithm 3.2.

Algorithm 3.2 Homogeneous edge-directed interpolation

for all red and blue pixels (x,y) **do**
  Calculate $gradient_{Hor}$ and $gradient_{Ver}$ based on [69]
  $homogeneity[x][y] \leftarrow gradient_{Hor} - gradient_{Ver}$
end for
for all red and blue pixels (x,y) **do**
  $direction \leftarrow \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+1} homogeneity[i][j]$
  if $|direction| < threshold$ **then**
    Interpolate green with gradient corrected bilinear interpolation
  else if $direction > 0$ **then**
    Interpolate green with the vertical method
  else
    Interpolate green with the horizontal method
  end if
end for
for all pixels **do**
  Interpolate missing red and blue values with constant hue approach
end for

This algorithm is fairly successful at removing the mazing artifact, but is not able to remove all the color spill as successfully as Hirakawas original homogeneity-based approach. The complexity of the algorithm is, however, significantly reduced. This algorithm has two green half-passes, i.e., only red/blue pixels are touched, and one full pass, opposed to the original with two green half-passes and five full passes (this may vary depending on the implementation).

We know that applications implemented in CUDA struggle when requiring multiple synchronization steps and iterative algorithms. Therefore, some algorithms, such as [73], that perform multiple passes with incremental improvement have been omitted in this thesis. We will instead include a common optional post-processing method in the next section that can be applied after any of the presented algorithms. Algorithms where it is believed impossible to compute all pixels in parallel have also been omitted.

3.6 Chroma Median Filtering

Many algorithms utilize an optional post-processing step for false color suppression. We have earlier discussed how color hue transitions slowly within localized areas of natural images. Applying a median filter on the difference between the green channel and the red/blue channel has been proven to remove many false color artifacts created by the demosaicking process [74].

In our system, we will eventually convert the images into the YUV colorspace. We explained earlier how the chromatic channels (U and V) represent the color difference of blue and red respectively, compared to the luminosity channel. This is exploited in [75], where they propose applying a median filter directly on the chromatic channels in a 3×3 grid. This means that for each pixel, we find the median value of the nine nearest pixels separately for both chromatic channels. This is a potentially iterative process that can be repeated multiple times with diminishing returns. Hirakawa and Parks propose three iterations of median filtering in [68]

based on empirical testing. The filter performs some smoothing, which can reduce the overall image quality, and may even occasionally introduce color artifacts with repeated iterations. Using larger grids than 3×3 is also possible, but exponentially increases the complexity of the computation and increases the chance of introducing new color artifacts.

Median filtering is a computationally complex operation, as it requires partial sorting of nine elements, assuming the typical 3×3 grid, for every pixel in the image. It is therefore rarely used for real-time image applications, despite being common in many demosaicking algorithms. The approach found fastest for this, both on CPU (assuming -O3 compiler optimization) and GPU, uses a series of C swap-macros, swapping two elements if the first value is greater than the second. In order to find the median value of nine elements we must perform a total of 30 swap operations. With our five high resolution images of 2040×1080 pixels, this means a total of $5 \times 2040 \times 1080 \times 2channels \times 30 \approx 6.6 \times 10^8$ swap operations for each iteration of median filtering.

3.7 Related Works

The Bayer filter is widely deployed, which means that there are some existing implementations that could be used. For example, the OpenCV [76] image library has both CPU and GPU implementations available. The GPU implementations include regular bilinear interpolation and the gradient corrected algorithm [65]. In this thesis, we wanted a broader evaluation of multiple algorithms, particularly some more complex. Another GPU implementation of [65], along with several optimizations, is detailed in [77]. Here, the authors show how the algorithm can be implemented in OpenGL, with minimal branching and use of vectorization. However, they also state that a general purpose GPU programming language, e.g., CUDA, should be more efficient if available.

Fastvideo [78] has released debayer software for Windows using CUDA, showing very impressive performance results. This features the gradient corrected algorithm and an unmentioned algorithm, again limiting the number of options. Unfortunately, it is also restricted to the Windows platform.

An implementation and optimization of the adaptive homogeneity-directed demosaicking algorithm is detailed in [79]. This has been implemented with CUDA, and provides a good visual result. However, from the description and performance measurements we saw that this algorithm would not be well suited for our real-time pipeline.

3.8 Implementations

The algorithms implemented are a subset of those presented in the previous sections: *nearest neighbour*, *bilinear*, *smooth hue transition* (bilinear interpolation of the green channel), *gradient corrected*, *weighted directional gradient* (original version and constant hue version, with four or eight directions), *edge sensing bilinear*, *edge directed* and the proposed new *homogeneous edge directed* algorithm. These have been selected to provide a varying degree of complexity and expected visual quality. They have been implemented on a GPU, using CUDA. A few of these were initially implemented on CPU, but were ported to GPU.

3.8.1 Optimizations considered

During the implementation, we have performed several key optimizations for the GPU. Specifically, we have optimized all algorithms for the Kepler architecture [40], as our recording pipeline uses a Nvidia GTX 680 GPU. Performing a full optimization for every algorithm is beyond the scope of this thesis, but we have performed several generic optimizations and ensured that all algorithms have been implemented efficiently enough to provide a fair performance evaluation. The most important of these optimizations are listed below. More details surrounding them, and CUDA specific terms, can be found in section 2.6.

Occupancy: None of the implemented algorithms are computationally very complex. This means that we believe all CUDA kernels should manage to achieve full theoretical occupancy, i.e., launch the maximum number of threads (2048 on Kepler). Our workload is more than large enough, and highly parallelizable, to limit this problem primarily to keeping the number of required registers, i.e., local per-thread variables, low. It may also vary how many pixels each thread should compute in its lifespan, as opposed to launching new threads. Even though the maximum number of threads that can execute in true parallel is 2048, greater performance will usually be achieved by spawning more. However, there is also some small (very small compared to CPU threads) overhead for creating threads and so we must here find a balance.

Reduce pixel lookups: For each pixel, all algorithms must access a number of surrounding pixels to interpolate the two missing channels. For some algorithms, e.g., the weighted directions, this number is quite extensive. Many values are used multiple times, which means that we may wish to load the pixel value only once. However, this increases the required number of registers, which in turn may reduce the kernels occupancy. Additionally, spatially close pixels will need to load a large amount of overlapping pixels. This means that it may be more efficient to calculate multiple pixels in one context, while the pixels are already loaded. Here, we must again be careful not to affect occupancy.

Fast pixel lookups: Even if we can reduce the number of pixel lookups, which we cannot always do, we must also ensure that each pixel lookup is as fast as possible. This is perhaps the most important optimization we can perform. Accessing regular global memory is slow if not coalesced, which is difficult to do for many of the algorithms. For each pixel, the pixels of interest are spatially close in two dimensions. Using texture memory, cached and optimized for two-dimensional lookups, can yield a significantly better result.

Data locality: In many cases it may prove efficient to keep separate data, that will be used together, sequential in memory. In our case, this involves keeping all information about a single pixel sequentially stored in the same array, instead of spread over multiple arrays or different sections of an array. It is generally far quicker to only read from a single array, than to spread the data around. This is due to cache efficiency, i.e., there is no need to cache from multiple areas in memory, and that memory accesses are always 32bit on Kepler. This means that if we wish to access a single byte, we will actually perform a 4-byte read operation. Therefore, it is quicker to perform a single 4-byte lookup, than four 1-byte lookups. Using larger, multi-byte, datatypes is therefore preferable whenever possible, as the only negative side effect is a potential small reduction in cache efficiency. The reduced cache efficiency can occur because the cache is of limited size, and can therefore hold fewer pixels at any given time.

Avoid branching: When implementing these algorithms, it is essential to avoid branching code. Given the Bayer pattern, we must perform different operations for the different kinds of pixels. If we were to run all the kernels on a single pixel, performing one of four branches, each thread would be forced to idle potentially 50% of the time. This is because each branching operation within a single thread warp must be executed by all threads in the warp, causing each branch to be processed sequentially. The adaptive algorithms also require branching, as they perform an evaluation based on their local values. These can usually be reduced to simple statements, and we can limit the operations that must be performed within each branch. When implementing any program, a common source of branching is exception handling. Typically, the majority of the execution is spent in a few, short code segments, while the majority of the actual code written by the programmer deals with exception cases. For us, this is primarily limited to image boundaries. However, CUDA textures allow us to perform out-of-bound texture lookups, e.g., by clamping the values to within the legal range. Since we stitch the debayered images into a panorama in our pipeline, we do need to use the 2-3 pixels near each border. This means that we can ignore all out-of-bound handling, and ignore image borders.

Arithmetic operations For all architectures, some operations are slower than others. Unlike a CPU, GPUs have especially fast floating point arithmetic operations, often faster than integer operations. Floats of 32 bits are faster to work with, but take up more space than individual bytes. Division is an iterative process, that requires several cycles to perform. Performing a floating point division takes nearly ten times as long as a multiplication, which means that whenever possible we use the multiplication equivalent, e.g., $\frac{x}{4} = 0.25 \times x$. Where this is impossible, e.g., smooth hue transition algorithm, we use approximation intrinsics instead. We saw that the approximation was accurate enough, and could decrease runtime of one kernel (smooth hue transition) by 26%. We also used other intrinsics, where possible, with significantly less performance gains. Other arithmetic operations are overall very cheap on the GPU, and they can often be performed while memory read operations are being handled. As a result of this, many kernels are memory bound because they do not perform enough computations in comparison to memory operations. We saw, through profiling, that we were able to avoid this with most algorithms.

3.8.2 Memory accesses

The first choice, when implementing on CUDA, is selecting a memory layout, and how to access the required data. The primary performance limitation of an algorithm on CUDA is often its memory accesses, and how well these can be optimized. Arithmetic operations are usually only a few processing cycles, while reading from global memory may require hundreds of cycles if the accesses are not coalesced.

We have implemented all algorithms using the same base principles, as they are primarily differentiated by the number of required passes and the number of pixel lookups per pass. The initial Bayer image is bound to a two dimensional texture, giving us the benefit of two dimensional caching when performing multiple texture lookups per pixel. The use of textures is essential, as most algorithms would be difficult to implement with good memory coalescing using global memory. For example, the weighted directional gradient with eight directions requires a total of 99 pixel lookups (not unique, but in total), per non-green pixel, when performing its green interpolation pass. In most kernels, we tried to perform as few texture lookups

as possible and rely on temporary storage when using the same pixel multiple times. However, with the weighted directions algorithm this increased the local register requirements for each thread, reducing the number of concurrent threads that could execute. Instead, we observed better results when performing duplicate texture lookups.

This could also have been accomplished using shared memory, but would be harder to coordinate and more device specific. In order to accommodate the ideal 128 threads per block for maximum occupancy of the Kepler architecture, using a 5×5 pixel lookup grid, we would need to load a total of $5 \times (128 \times 2 + 4) = 1284$ bytes per block. This becomes problematic when crossing image row boundaries, and may prove difficult to optimize for most horizontal resolutions. We believe that the quality of caching from using a single texture is more beneficial, and produces a better result than shared memory. By opting to not use shared memory, we can also utilize a larger general cache, as this typically uses the same memory pool.

The algorithms also differ in the memory requirements, primarily as each additional pass uses an additional temporary buffer. In this thesis, we have not optimized for low memory consumption, but for overall execution time. This means that we often duplicate data instead of reading from multiple arrays within the same kernel. This primarily applies if we need multiple texture lookups from both arrays.

3.8.3 Multiple passes

Most of the algorithms utilize multiple passes, usually one initial green pass followed by a red and blue pass. These are implemented in essentially the same way, using a temporary texture with two bytes per pixel, for saving the green value and either a red, blue or empty value. Using a single texture for this provides much better data locality and cache efficiency, increasing performance significantly over using two separate textures. In order to write the temporary values, we utilize surface memory.

The homogeneous edge directed algorithm uses two passes to interpolate the green channel. In the first pass, the green value is computed both based on the horizontal and the vertical interpolation method. Additionally, we calculate the directional preference. These values, along with the original red/blue value, are written to surface memory with 4 bytes per pixel. It proved significantly faster to keep this data localized in one array, despite having to perform nine texture lookups when we determine the localized directional homogeneity.

The original weighted directional gradients uses two passes to interpolate the red and blue channels. The second pass fully interpolates the red and blue pixels, leaving the green pixels untouched. This data is then used in the third pass to complete the remaining red and blue values. This implementation uses a full four bytes per pixel to ensure data locality for the final pass, but this may not be ideal. It is generally considered more efficient to use four bytes per pixel instead of three, due to memory alignment, but in our case, we have only half the pixels carrying three values and the other half (green pixels) carrying a single value. We saw early on that the red and blue interpolation performed in the original algorithm would be quite costly. Therefore, we opted to implement two variations of this algorithm, adding an alternative version that borrows the constant hue correction-based approach from the edge directed algorithms.

All of the initial green passes need little to no branching, as the non-green pixels can be treated identically. None of these algorithms need to know whether the current pixel is red or blue, which also simplifies implementation. The other passes, however, must take pixel color

into consideration. This means that these kernels have a greater number of possible implementation options.

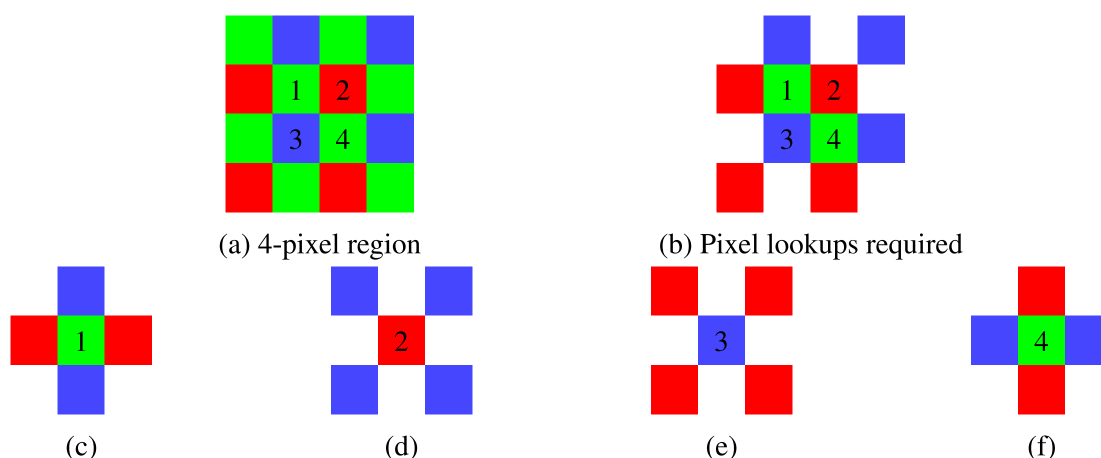
3.8.4 Kernel design

When implementing the kernels that needed significantly different behavior depending on the pixel color, we wanted to avoid branching code. We also, initially, did a fairly generic design for ease of implementation.

2-pixel kernel The first implementation interpolated each pixel separately, performing all pixel lookups required. However, we ensured that we always processed two pixels consecutively. The kernel would first determine if the two pixels are of a green/red row, or a blue/green row. This evaluation would always yield the same branch, within a warp, except for warps that crossed row boundaries. With our target horizontal resolution of 2040, this meant that only one warp out of $\frac{2040pixels}{32 \times 2pixels} = 31.86$ would encounter branching. Then, it would interpolate the two pixels separately, before performing the YUV conversion. Every algorithm had its final pass implemented in this way.

4-pixel kernel We saw that the previous kernel usually covered a lot of overlapping pixels, and still required some branching. This can be seen in figure 3.8, which highlights a four-pixel region. In the edge directed final pass, each pixel must perform five lookups. However, we can see that if we interpolate these four pixels together, we only need a total of ten lookups. For other algorithms, that required more pixel lookups, this overlap was even greater. Therefore, we tried to interpolate four pixels at the same time, covering the four possible branches.

We saw that the original weighted directions algorithm could not be implemented efficiently in this way, as it required too much intermediate storage. It uses a much larger region than the



(a) 4-pixel region

(b) Pixel lookups required

(c)

(d)

(e)

(f)

Figure 3.8: Visualization of 4-pixel kernel implementation, using edge directed as an example. The four pixels to calculate are highlighted in (a). In (b) we show which of the surrounding pixels must be read as well. In (c,d,e,f) we show the lookups required for each of the four pixels. Also note that all red/blue pixels also contain a green value, previously calculated.

other algorithms, and each lookup is of four bytes each. Since we, at this point, realized the inefficiency of this algorithm we did not implement this method for this particular algorithm.

3.8.5 YUV conversion

The final step for all algorithms is the conversion to YUV colorspace. This is performed for each pixel once all RGB channels are known, i.e., part of the final pass of each algorithm. The conversion is performed based on the matrix in figure 3.2, and written to surface memory, interlaced, with four bytes per pixel.

We also proposed using median filter on the two chroma channels, in post processing. The proposed median filter step can be performed multiple times, by alternating between two buffers, each bound to a CUDA surface object. This was implemented by first reading in the 9 pixels to apply the filter on, separating the two channels into two arrays. Then we performed a highly optimized median filter on each of these arrays.

3.9 Evaluation

We have presented several implemented algorithms, and in this section we will evaluate these algorithms and narrow down our possible options. In this section, we will be using the aforementioned Kodak image suite [61]. The Bayer pattern is initially artificially imposed by subsampling the RGB channels and then reconstructed with the implementations described in the previous section. We will then evaluate each algorithm with these metrics:

Accuracy of reconstruction for each of the color channels, based on the mean squared difference from the original image.

Edge preservation to see how each algorithm handles strong straight and diagonal edges, based the level of produced zippering artifacts from aliasing.

Color spill to visually compare each algorithms ability to handle patterns prone to produce color spill.

Execution time to determine which algorithms are real-time suitable.

3.9.1 Accuracy of image reconstruction

The first criteria we will evaluate is the visual quality of the images. The perceived quality of an image can be a highly subjective and not necessarily directly tied to how accurately the image is reconstructed. The best performing algorithms focus on creating a natural looking image, void of obvious visual artifacts, instead of a perfect reconstruction. However, the accuracy of the reconstruction is perhaps the easiest to evaluate, and gives us a strong indication of how the algorithms fare in relation to each other.

Peak signal-to-noise ratio (PSNR) is a common measure for the quality of image reconstruction, expressed in decibels. PSNR is based on the mean squared difference between a

reconstructed image and its original, and can be defined by equation 3.11 for an image with N pixel values.

$$PSNR = 10 \times log_{10} \left(\frac{255^2}{\frac{1}{N} \times \sum_{i=0}^{n-1} (Recons[i] - Orig[i])^2} \right) \tag{3.11}$$

Equation 3.11 treats each color channel equally, but we have established earlier that the RGB color channels are not visually equivalent, i.e., there are typically different artifacts associated with a high mean square error in the green channel compared to the red and blue channel. Therefore we compute PSNR separately for each channel. The use of PSNR as a metric can be disputed, as it may often produce inaccurate results for some image transformation when used as a metric for visual quality. However, for Bayer demosaicking, we are primarily looking for a direct reconstruction. One effect that we should be aware of, however, is that algorithms that perform edge detection may sometimes displace an edge by one pixel, instead of causing zippering artifacts. This will typically be preferable, and visually invisible, but may negatively impact the resulting PSNR.

In our recording pipeline, we do not need to care about image borders/edges, as they are cropped away in the final stitched image. Because of this, there is no edge handling in the implementations which lead to many odd artifacts occurring. Therefore we filter out a few border pixels when measuring PSNR.

Most of the algorithms perform well in homogeneous areas, but small differences can accumulate to falsely represent the quality of the reconstruction. More accurately, we are not particularly interested in the homogeneous regions, as any deviations from the original are likely to be minimal and unnoticeable. In [66] it is proposed to calculate an edge mask from the original image, and filter out all the pixels with a low gradient value. The Sobel operator is a first-order differential edge detection operator, computing a weighted average of the horizontal and ver-

(a) Original lighthouse (b) Sobel edge detection (c) Edge-filtered mask

Figure 3.9: Lighthouse image used for reconstruction comparison

tical gradients of a pixel. This is applied to a grayscale version of the original image, and is implemented as specified in [80], but using the slightly different 3×3 Scharr operator as shown in [81]. After computing the edge magnitude, we create a mask by filtering out all pixels below a threshold, found empirically.

We selected the lighthouse image seen in figure 3.9, as it includes several challenging regions for reconstruction. Other images were tested, yielding similar results, but were omitted for brevity. Figure 3.9 also shows the edge magnitude given by the edge detection and the resulting filtered areas.

The resulting PSNR values can be seen in figure 3.10. Good reconstruction of the green channel typically means good preservation of edges and sharpness, while the red and blue channels strongly indicates the algorithms ability to avoid false color artifacts.

The algorithms that interpolate each channel separately, i.e., not accounting for inter-channel relations, perform poorly on the sparsely sampled red and blue channels. This can be seen especially in the bilinear and edge sensing bilinear algorithms. On the other hand, all the algorithms with a green PSNR above 34 utilize the same gradient correction principle when reconstructing the green channel. This helps prove the effectiveness of this method.

We can also see that the best performing images all use a constant hue approach, solidifying our earlier claim that the interpolation of the red and blue channel is primarily limited by the accuracy of the green interpolation. Smooth hue transition also performs very well, but is limited by the initial bilinear interpolation of the green channel, and would likely perform much better with a different algorithm for constructing the green channel. The weighted directional gradients algorithm proves significantly worse when performing directional weighting on the red and blue channel, and is improved by enforcing the constant hue. Based on these results

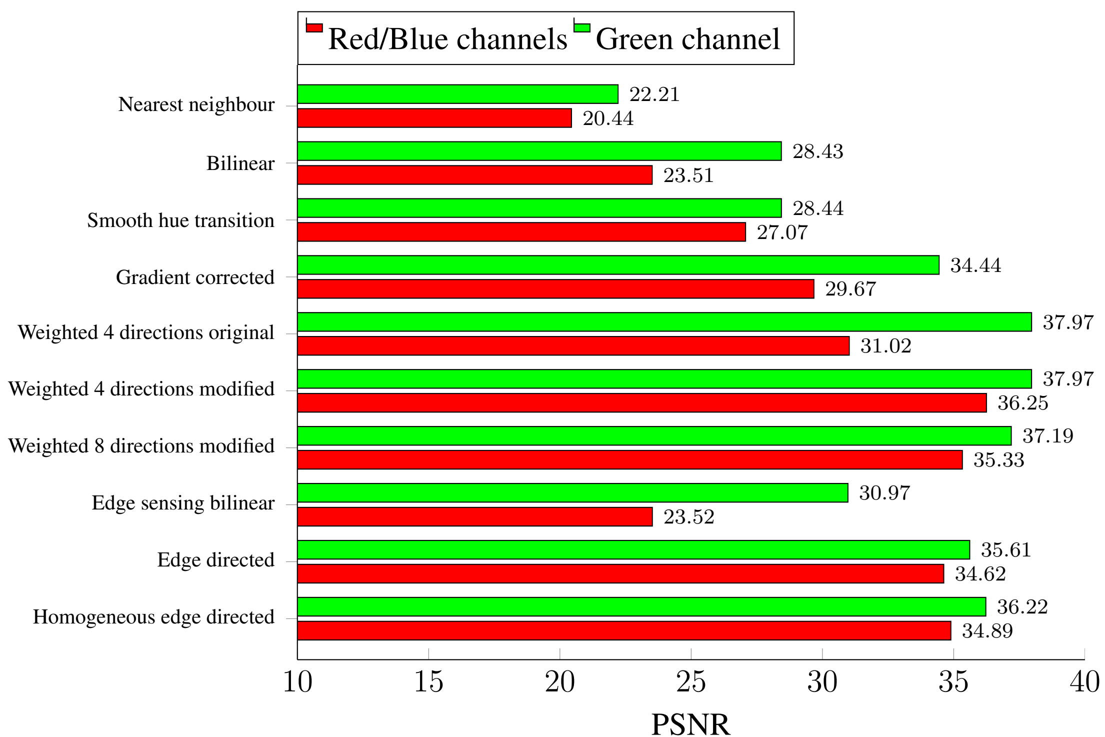


Figure 3.10: PSNR comparison of our selected implementations in edge regions on lighthouse image, higher values are better

we opt to use only four directions to the weighted directional gradients algorithm, and use the constant hue implementation instead of the original weighted approach to the red and blue channel, as the added complexity fails to produce better results.

3.9.2 Quality of edge preservation

We will now take a closer look at each algorithms ability to preserve straight and diagonal edges. Figure 3.11 shows how the various algorithms adapt to particularly difficult edges, exhibiting varying degrees of visual artifacts. While such clear edges and rapid transitions are far from anything we would see in a natural image, it produces an exaggerated effect to visualize the limitations of each algorithms edge handling.

Looking at figure 3.11 we see very clearly the zippering artifact in the bilinear, smooth hue transition and gradient corrected algorithms, as a result of not performing any weighting or edge sensing. All the algorithms that utilize gradient correction have approximately 50% smoother color artifacts, attributed to the gradient correction coefficient defined at $\frac{1}{2}$, which reduces the visual impact of the artifacts in more realistic images.

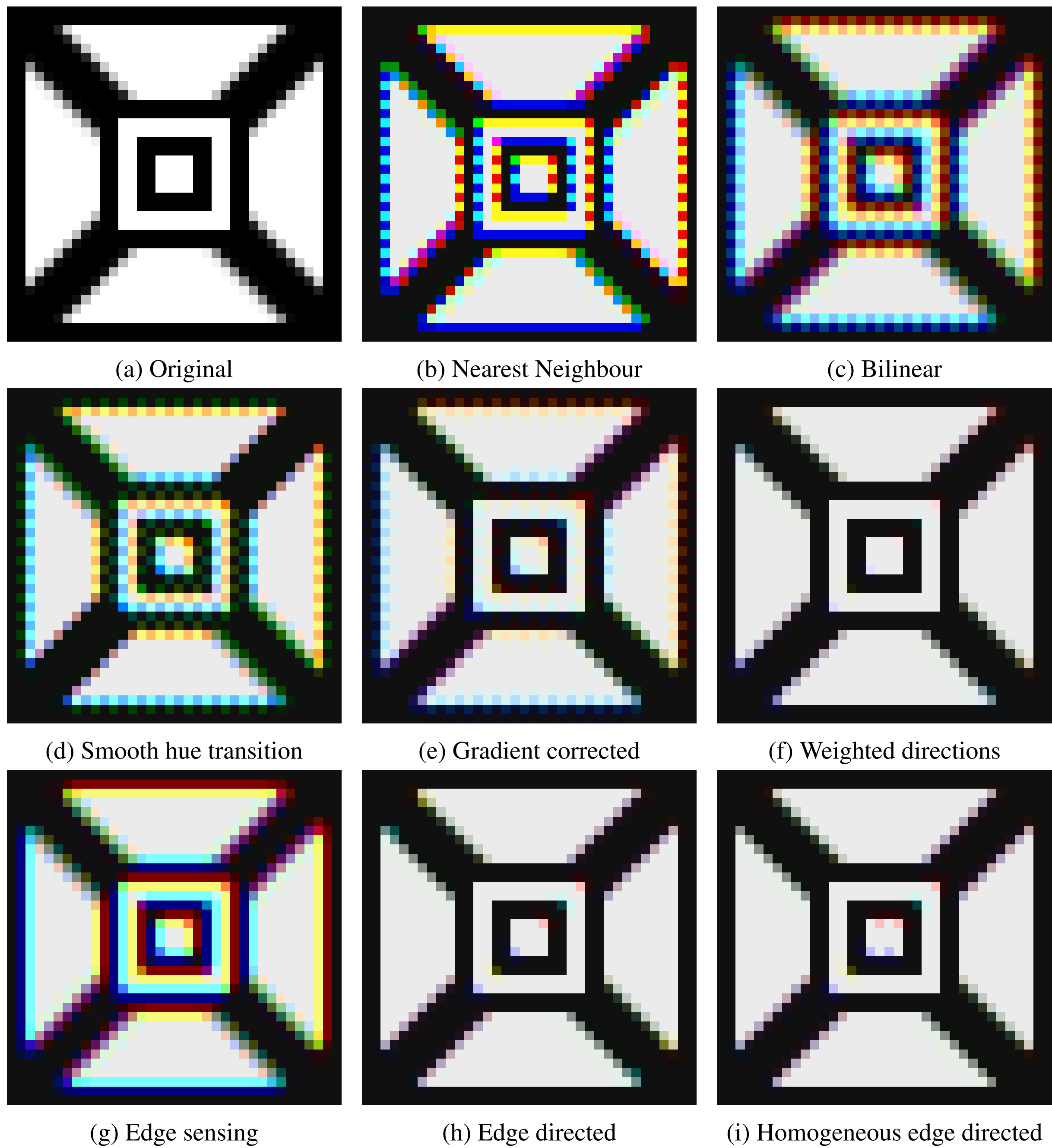
The edge sensing algorithm only adapts for edges in the green channel, which essentially removes the zippering artifact. The color spill is instead consistent and fairly limited to the red and blue channel, conforming the to PSNR measurements of figure 3.10. The edge directed algorithm performs very well along horizontal and vertical lines, but produces a few errors in corners and along the diagonal lines. This is to be expected, as it only accounts for two directions. The homogeneous approach introduces a few more artifacts as a result of enforcing homogeneity, but one could argue that it produces better results along diagonal edges.

The weighted directional gradient performs visually best in this test, as it produces only minor color artifacts in corners and along diagonal lines. While it is difficult to see in the figure, it produces all the same artifacts in corners as the edge directed algorithm. However, they are virtually indistinguishable from the white background due to the low weighting.

3.9.3 Visual assessment of color spill

The final quality evaluation we will make is how well the top algorithms manage to avoid or minimize color spill. This is typically the most challenging visual artifact, as well as the most prominent in some images. The lighthouse image is again selected, as every algorithm struggles with color spill in this image, to varying extent. Figure 3.12 shows how the, so far, highest rated of our implemented algorithms handles the worst region. We have omitted several algorithms here, due to their bad results in the edge test. These algorithms all produce significantly worse results than shown in figure 3.12.

We see in figure 3.12 that the gradient correction algorithm is unable to remove the color spill, producing a very visible and consistent pattern. The weighted directional gradients produces the same effect, but significantly less prominent. The color spill is also more regionally contained, i.e., not visible except for the absolute worst area. This consistent pattern is also overall less visible than the mazing artifact in the edge directed algorithms. Whenever the edge directed algorithm chooses the wrong direction, the result is typically significantly more visible. The edge-directed algorithms perform fewer mistakes than the weighted directional gradients, but at a much higher cost. In images with minimal color spill prominent areas, we predict the weighted directional gradients to outperform the edge directed algorithms as a result.

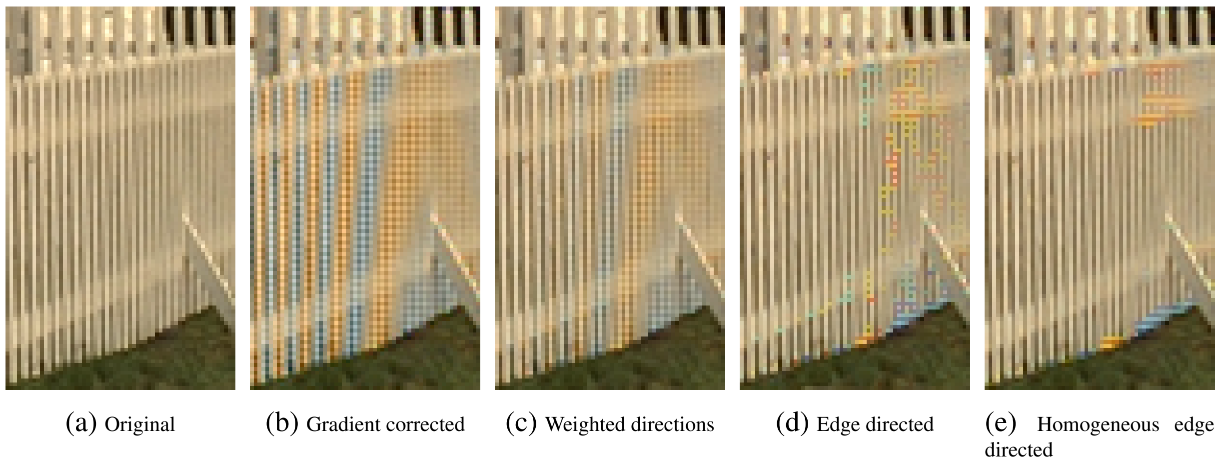


(a) Original (b) Nearest Neighbour (c) Bilinear

(d) Smooth hue transition (e) Gradient corrected (f) Weighted directions

(g) Edge sensing (h) Edge directed (i) Homogeneous edge directed

Figure 3.11: Black and white edge test, 36×36 pixels



(a) Original (b) Gradient corrected (c) Weighted directions (d) Edge directed (e) Homogeneous edge directed

Figure 3.12: Color spill comparison using the fence in the lighthouse image

The homogeneous edge directed algorithm is fairly effective at removing the mazing artifact. There is still quite a bit of residual color spill, but it is not as visually obvious. The algorithm reduces the number of times the edge prediction fails, yielding much fewer breaks in the straight vertical lines. It is worth noting that we also implemented a variation of the gradient selecting homogeneity approach of [72] that resulted in better removal of color spill. However, this algorithm caused several other artifacts in other areas, providing an overall worse visual quality. We believe that our homogeneity based algorithm performs best in this test, but that the weighted directional gradients may produce a smoother pattern in areas without, or to a smaller extent, risk of color spill that may be preferred.

3.9.4 Algorithm execution performance

The primary requirement of our Bayer demosaicking algorithm is performance, i.e., real-time execution time. The pipeline described in chapter 2 requires the module to process five 2K resolution images at fifty frames per second, i.e., $5 \times 2040 \times 1080$ pixels in less than 20 milliseconds. We must also consider that there will be other modules executing in parallel, perhaps with higher priority, using the same shared resources.

Performing a full optimization for every implemented algorithm is beyond the scope of this thesis, and there are likely some small performance gains that could be achieved in each CUDA kernel. We have, however, performed individual profiling of every kernel, ensuring that they all use the optimal thread, block and grid configuration (see section 2.6). Each algorithm is able to achieve full theoretical occupancy, and over 90% in practice, without having to resort to slow local storage. However, the algorithms vary to some extent in efficiency of texture lookups.

Each implemented algorithm has been benchmarked on an Nvidia GTX680 graphics card and an Nvidia Quadro K2000, using the mean conversion time of 2000 frames. The frames used are of the same size as the pipeline, to provide easier comparison. Also note that the GPU used in our current prototype configuration is a GTX680. The full results can be seen in table 3.1, and a plotted summary in figure 3.13.

As we can see from figure 3.13, all of the evaluated algorithms are well within our 20ms real-time constraint. The primary difference between algorithms is the number of required passes and texture lookups, shown table 3.1, not necessarily their computational complexity.

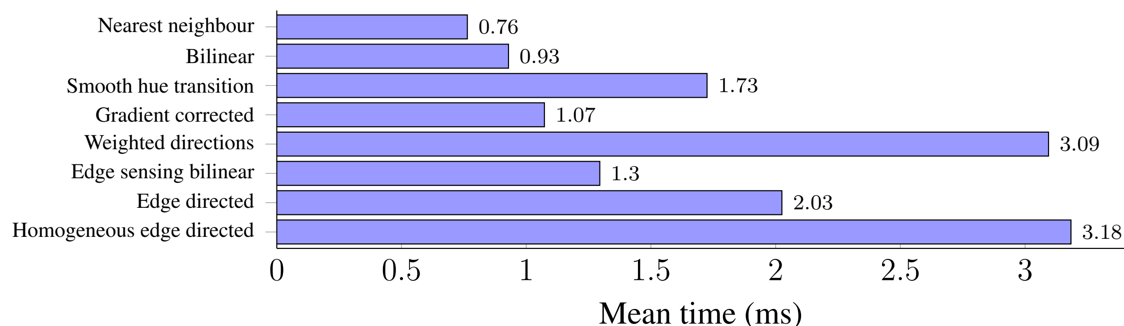


Figure 3.13: Performance comparison of multiple demosaicking algorithms on $5 \times 2040 \times 1080$ images, performed on a GTX680 GPU

An exception to this is the edge sensing bilinear algorithm, which is significantly slower than all other single pass implementations. This algorithm, and its implementation, differs very little from regular bilinear, but is slowed down by branching code. In number of computations and texture lookups, the gradient corrected should be among the slower algorithms. However, we see the benefit of only performing a single pass. The modified weighted directions algorithm, with only two passes, is nearly as slow as the three pass homogeneous edge directed algorithm. This is due to its complex first pass, as can be seen in table 3.1. Overall, we see that the algorithms perform quite similarly. The quickest algorithms, nearest neighbour, is only slightly faster than the bilinear. This, we attribute to having too few arithmetic operations. Because it does no averaging, it consists of only texture lookups and the YUV conversion. This can show a pretty decent baseline for how long it takes to loop through the entire image and convert the output pixels to YUV.

The disparity between single pass algorithms and those requiring multiple passes may increase when incorporated into the panorama pipeline, as other kernels may be scheduled in between each pass. However, this can in fact increase the efficiency of the card utilization, as there is no time spent in between passes. This is minimal however, as CUDA schedules kernels very efficiently when operating on the same stream, without explicit synchronization. Every kernel fully utilizes the card when they run, meaning that no other kernels will likely be scheduled simultaneously. If they were, it would benefit those algorithm with fewer texture lookups, emphasizing also the byte size of each texture lookup, as the efficiency of the caching may be reduced.

We saw that the byte size of each texture lookup mattered very little. The algorithms that use temporary storage, for saving results of a previous pass, use larger datatypes. This affects the number of bytes read from each texture lookup, but we have earlier mentioned that CUDA performs 32-bit addressing for all read operations. From table 3.1, the penalty for reading larger datatypes, up to 32bit, seems very low. We can see this by comparing for example the final pass of the original weighted directions algorithm, having 124 32-bit lookups, and the first pass of the version with 8 direction. Though it is not apparent from this table, these two kernels are computationally quite similar. Although the latter requires slightly fewer arithmetic operations, it needs to read 2.5 times as many bytes. The fact that it is still significantly faster shows the efficiency of 32-bit read operations.

Algorithm	Execution time (µs)		µs / pass (GTX680)			Lookups / 2×2			Temporary storage
	Quadro K2000	GTX680	$1st$	$2nd$	$3rd$	$1st$	$2nd$	$3rd$	
Nearest neighbour	4489	764	681			7×1			
Bilinear	5516	929	852			14×1			
Smooth hue transition	9383	1725	748	870		12×1	14×2		$2 \times x \times y$
Gradient corrected	6370	1073	993			24×1			
Weighted 4 directions original	49212	9061	2112	2087	4696	78×1	92×2	124×4	$6 \times x \times y$
Weighted 4 directions modified	19052	3094	2050	894		78×1	10×2		$2 \times x \times y$
Weighted 8 directions modified	31594	7994	6793	895		198×1	10×2		$2 \times x \times y$
Edge sensing bilinear	6470	1295	1219			14×1			
Edge directed	10941	2025	987	881		20×1	10×2		$2 \times x \times y$
Homogeneous edge directed	20029	3184	1106	1032	905	18×1	18×4	10×2	$4 \times x \times y$

Table 3.1: Summary of all algorithms resource requirements. Execution was measured with a 2040×5400 resolution image. Note that, in addition to each pass, some CPU overhead is required for preparing buffers and launching kernels. For each pass, we show the number of texture lookups per 2×2 pixels, i.e., 2 green, 1 blue & 1 red, of either 1, 2 or 4 bytes each.

3.9.5 Kernel evaluation

In section 3.8.4, we presented two different kernel implementations. The initial, simple approach, computed two pixels at once. The second approach computed four pixels at once, and performed only the minimum required texture lookups. This is quite apparent from table 3.1, as the original weighted directional algorithm has not been optimized using this second approach and therefore have a disproportionately large number of texture references. We also have not optimized the initial green pass in the same fashion, as there is significantly less overlap in pixels read.

Both implementations of the final pass of each algorithm are shown in figure 3.14. Here we see that the 4-pixel kernel is superior for all implementations. We see immediately that the smooth hue transition benefited greatly from this change, though we acknowledge that there was likely something odd in the 2-pixel implementation that caused it to be so slow. This implementation also unable to achieve full theoretical occupancy, limited at 67% due to high register use. This indicates that this initial implementation could likely have been optimized further. The 4-pixel kernel also utilizes the native division approximation function, which computes the approximate division at half the number of cycles as a full division. This approximation, however, yielded identical PSNR and visual result. Because this algorithm relies heavily on division, this modification was able to speed up this kernel by nearly 20% (208µs).

Based on figure 3.14, it is otherwise difficult to see a strong pattern of why the benefit was greater for some implementations. We can, however, deduce that the gradient corrected implementation likely has a larger gain, due to having the highest number of pixel lookups.

3.9.6 Pixels per thread

We tried several different variations for the number of pixels to compute for each thread. We have mentioned that kernels compute 2 or 4 pixels at once, but each thread can iterate through multiple pixel-blocks in its lifetime. One can either launch very many threads, with short lifetimes, or few threads with longer lifetimes. Note that “few” threads in this context is still several thousand.

For the 2-pixel kernels, we consistently saw the best results when each thread computed 32 pixels, i.e., iterate 16 times. The only exception to this was the nearest neighbour algorithm,

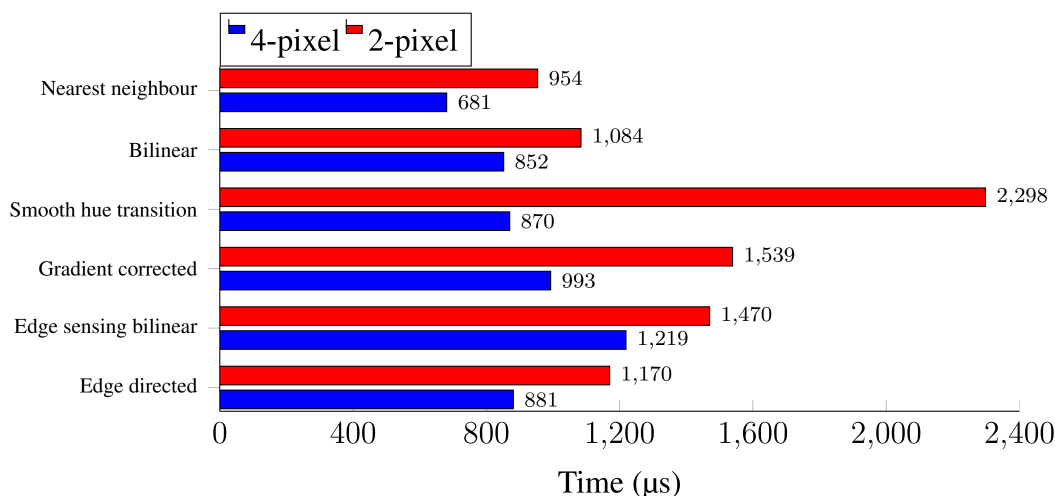


Figure 3.14: Performance evaluation of the final pass kernels, using the 4-pixel variant and the 2-pixel variant. Note that the modified weighted directional gradients and homogeneous edge directed algorithm also use the edge directed kernel for their final pass.

which performed significantly better when computing only 16 pixels. The 4-pixel kernels performed best when computing only 8 pixels each, iterating twice.

3.9.7 Median filter

When introducing the median filter for post-processing, we expressed concerns in regard to computational complexity. Therefore, we evaluate whether this is feasible in real-time, and if the quality improvement is worth it. Table 3.2 shows the effectiveness of the median filter on the PSNR of the red and blue channel, the green is virtually unaffected. The filter removes several of the worst artifacts, and creates an overall smoother image with fewer abrupt color changes. However, on images that have few artifacts to begin with it has little effect, as can be seen by the minor differences in PSNR for the weighted directional gradients algorithm. Each iteration of the median filter is measured at approximately $5.1ms$ given the regular 2K resolution image set, which is slower than any of the standalone algorithms by far. We believe that this is too long to warrant even a single iteration, as we are likely better off by then using a better initial algorithm.

#filter iterations	Bilinear	Weighted directions
0	23.507	36.068
1	26.349	36.247
2	27.167	36.358
3	27.316	36.397

Table 3.2: Evaluating the effect of the median filter on red/blue PSNR

3.9.8 Conclusion

In this section we have evaluated the implemented algorithms. We have given a quantifiable measurement for the accuracy of the image reconstruction, PSNR, which led us to eliminate the two most complex algorithms. Using eight directions for calculating the weighted directional average proved to degrade the quality of the prediction, limiting us to four directions. We also saw that relying on the correct interpolation of the green channel and maintaining a constant hue relation between the channels proved more effective than performing directional weighting when interpolating the red and blue channel.

The visual black and white edge test showed the ability of each algorithm to adapt to sharp edges. We saw how the simple algorithms struggled along any edge, most providing severe zippering artifacts, while the edge directed algorithms struggled slightly with diagonal lines. The weighted directional gradients algorithm was able to adapt nearly perfectly to all edges, and minimize the visual impact of each artifact.

Based on the performance evaluation of each algorithm, we determined that all of the remaining algorithms are acceptable choices for real-time video. The smooth hue transition algorithm, after having fared quite poorly in the visual tests, was excluded because of the efficiency of algorithms with better visual results. We also saw that the difference in performance between the medium scoring algorithms, e.g., the gradient corrected algorithm, and the fastest nearest neighbour and bilinear algorithms were less than expected. We believe that the difference in quality far outweighs the increased runtime, and we decide to eliminate the most simplistic algorithms. A summary of the performance evaluation of each algorithm was presented in table 3.1. Based on the fairly significant computations required to perform the median filtering, we also believe that the minor increase in quality is not worth the required resources for a real-time video system.

We could also see that our implementation of the gradient corrected algorithm was within 90% of the number provided in [78], which achieved 21 Mpix/s on a GTX Titan GPU. Our implementation measured close to 20 Mpix/s, which shows that our algorithms are likely efficiently implemented, given that we also perform the additional YUV conversion step. Without this step, we could perhaps achieve better results than provided in [78].

The remaining algorithms were all compared based on their ability to suppress color spill,

Algorithm	PSNR		Zippering	False colors	
	Green	Red/blue		Frequency	Intensity
Nearest neighbour	22.21	20.44	Very strong	Very high	Very strong
Bilinear	28.43	23.51	Very strong	Very high	Very strong
Smooth hue transition	28.43	27.07	Very strong	High	Strong
Gradient corrected	34.44	29.67	Strong	High	Strong
Weighted directions original	37.97	31.02	Very weak	Medium	Medium
Weighted directions modified	37.97	36.25	Very weak	Low	Weak
Edge sensing bilinear	30.97	23.52	Weak	High	Very strong
Edge directed	35.61	34.62	None	Very low	Strong
Homogeneous edge directed	36.22	34.89	None	Very low	Strong

Table 3.3: Summary of each algorithm's visual performance. We rank the algorithms based on the best and worst within each category, not amongst all alternative algorithms excluded from this thesis.

which occurs frequently in certain consistent patterns. The homogeneous edge directed algorithm proved best at removing the color spill altogether, while the weighted directional gradients was able to minimize its intensity significantly, which could result in a more aesthetically pleasing result in many situations. We elect to keep all of these algorithms for consideration, as they range quite well in both visual and computational performance, as we yet do not know the required performance and quality when operating in the real panorama pipeline. A full comparison, on each visual metric, of the implemented algorithms can be seen in table 3.3.

In the next section, we will look specifically at the Bagadus scenario, and determine which algorithm fits best in our final pipeline.

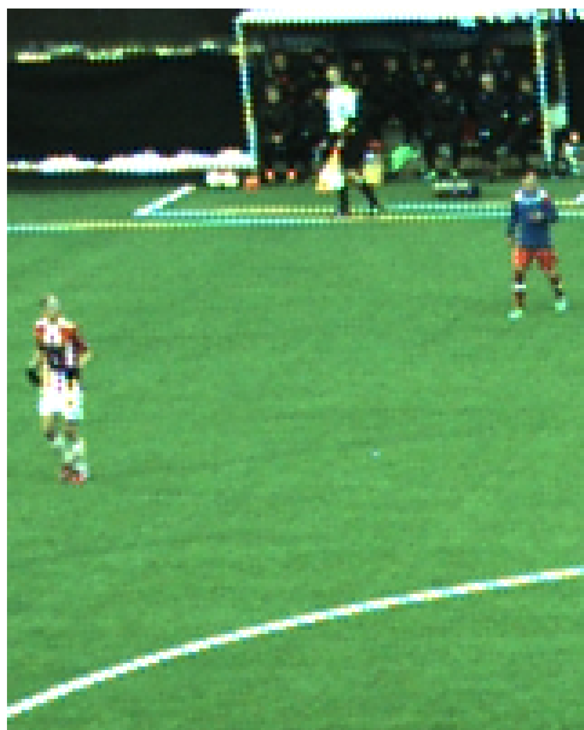
3.10 Bagadus Scenario

Until now, we have only evaluated the reconstruction of existing images that have been converted into Bayer format. In this section, we will be using footage directly from the camera, which often reduces the number of visual artifacts. The Kodak image set we have used so far has been selected for its challenging regions, meaning that the previous evaluations can be seen as worst case scenarios.

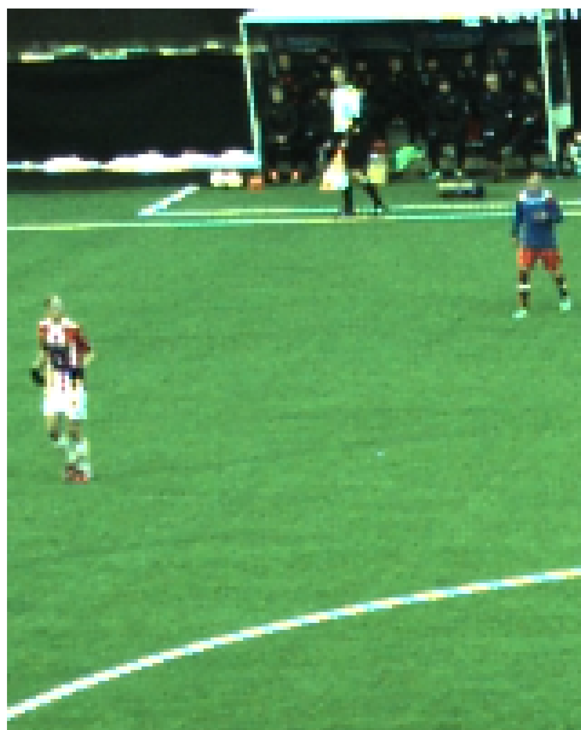
In the Bagadus scenario we are primarily interested in avoiding visual artifacts from static background, i.e., the soccer pitch, stadium etc. The high frame rate means that artifacts in areas of significant movement can rarely be observed by the viewer, assuming it is only present within a single frame. What we want to eliminate is the color spill that typically forms on the white lines of the soccer pitch or in some regions of the stands, as well as other effects that may distract the viewer. Yellow and blue color spill is common around these white lines, often merely as a result of not adapting to edges. When combined with the other modules of our pipeline, primarily the video encoding step and the chroma subsampling, the zippering artifact often becomes indistinguishable.

Certain situations with little light may produce images with quite a significant amount of noise, as the cameras attempt to boost the signal through gain, causing noticeable differences in the normally homogeneous grass pitch. In these situations, we can see mazing artifacts in the grass with the edge directed and homogeneous edge directed algorithms. This is only visible on still images, not the final video, but can be observed as an overall increase in noise. However, these situations are not our prime concern, as the soccer pitch is typically well lit. It is also worth noting that the overall color spill changes depending on the lighting conditions, as the algorithms typically perform slightly worse on bright images.

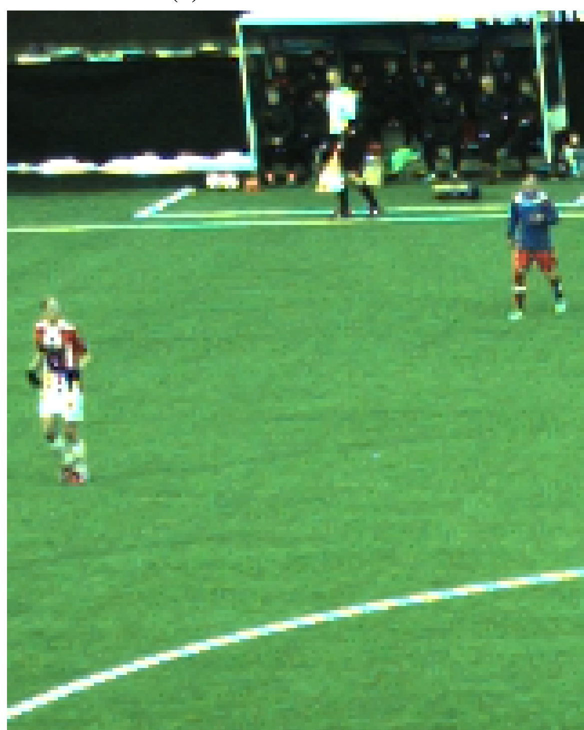
In figure 3.15, we see the four remaining algorithms applied to the same raw image, captured by our prototype system, before being warped or encoded. The most distinctly different algorithm is the gradient corrected, which produces a fairly noticeable zippering artifact along both curved and straight lines. As aforementioned, the zippering artifact is mostly removed in the final video, but still leads to a more intense color spill. Weighted directions algorithm produces the least intense color spill, though the difference is very small and can be difficult to discern from these images.



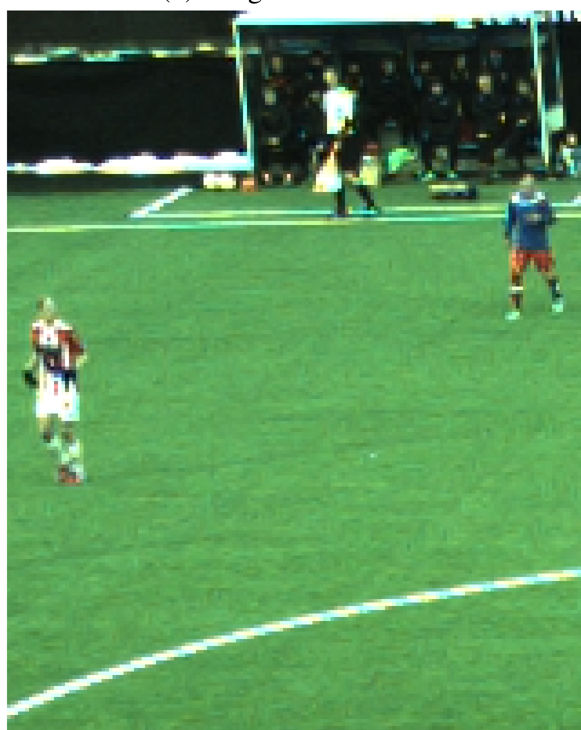
(a) Gradient corrected



(b) Weighted directions



(c) Edge directed



(d) Homogeneous edge directed

Figure 3.15: Comparison between algorithms on real images from our prototype system. Here we highlight a cropped section of 160×200 pixels where artifacts were most apparent, from the same raw image. Note the consistent spill of yellow and blue along the white lines, as well as the zippering artifacts on the gradient corrected image.

There are very few natural sharp edges that strongly favor the two edge directed algorithms. Imposing a local homogeneity in the choice of edge direction has little impact, as this is primarily targeted at consistent straight horizontal or vertical lines. This also means that we do not see any mazing artifact from the edge directed algorithm, which we noted to be its primary drawback. Most artifacts occur in the curving lines, where the edge directed algorithm typically struggles, which is why we believe the intensity of the color spill is slightly higher for the edge directed algorithms, as opposed to the weighted directions. Another small downside to these two algorithms is the slightly reduced quality in homogeneous areas, such as the grass, as half the interpolation samples are typically discarded when interpolating the green channel. In practise, we are unable to notice this and it may be worth noting that cubic interpolation is performed by the stitching module, which helps smooth out these homogeneous areas in the final panorama.

From a visual assessment and performance evaluation, we believe that the best performing algorithm is the edge directed algorithm. It produces nearly as good visual results as the best algorithms, at a lower computational cost. In our real images we see no mazing artifacts and the algorithm performs better than expected on diagonal lines. The weighted directions algorithms produces the best visual result, with minimal color spill and strong preservation of details. However, we do not believe this difference to be significant enough to warrant the increased runtime, even though both algorithms are relatively quick. If slightly better detail is desired, we believe the weighted directions is the best second option, as we judge both algorithms to be more than capable for our Bagadus scenario.

3.11 Summary

In this chapter, we have looked into multiple algorithms for reconstructing the full RGB color channels from the raw Bayer patterned images. We have explained the process of Bayer demosaicking, and shown that many algorithms share the same concepts, such as maintaining a constant hue. We also detailed the most common visual artifacts that can occur due to algorithm limitations, as well as showing how some concepts target individual artifacts.

We presented multiple algorithms in this chapter, detailing their approach and complexity. These were divided into adaptive, i.e., treating individual pixels differently and excluding some interpolation samples to improve the result, and non-adaptive algorithms, i.e., performing the same calculations in all pixels. The adaptive quality was typically reserved for edge handling, as algorithms attempt to interpolate along edges instead of across. We also presented an optional post-processing step, chroma median filtering, that attempts to remove visual artifacts present in the chromatic color channels, though this was found too costly for our system.

Then, we showed how a large subset of the presented algorithms were implemented on the GPU architecture, highlighting the CUDA optimizations that were required for high performance. We saw that most of the algorithms could be implemented in much the same way, primarily differentiated by the number of passes. The key optimization we utilized was the use of texture memory and kernel design, emphasising the efficiency of the two dimensional texture caching provided by CUDA. Some of the algorithms also had additional passes that required some unique design. We also evaluated multiple kernels designs, which showed the efficiency of data locality and reduced texture lookups.

We then evaluated all of the implemented algorithms, using a set of images from the Ko-

dak Image Suite [61]. We evaluated the algorithms on their ability to accurately reconstruct an image, based on the *peak signal-to-noise ratio* in artifact-prone regions. Among other things, this test showed the importance of utilizing the inter-channel relation between the green channel and the red/blue channel. We then performed a visual edge-test, which showed each algorithms ability to adapt to straight and diagonal lines. Here we saw that the most simplistic algorithms produced significant false colors around edges, while the more complex algorithms were able to adapt efficiently. Again we saw that, despite producing artifacts, maintaining a constant hue reduced the visibility of the false colors. Next, we performed a visual assessment of the top scoring algorithms ability to avoid color spill in a particularly difficult image region. We saw that none of the algorithms were fully able to remove the color spill, but the resulting artifacts varied in intensity and size of affected region.The final evaluation was a performance comparison between the algorithms. We saw that they were primarily differentiated by the number of required passes, followed by the number of texture references required. Each implemented algorithm performed well below our real-time requirement.

Finally, we evaluated the best performing algorithms with real images from our Bagadus prototype system, establishing their suitability in our scenario. We explained how the effect of each visual artifact and the difference between the algorithms is nearly hidden by the later modules. What is typically visible is the blue and yellow color spill along the white lines of the soccer pitch, which we saw present in all implementations. We saw that the weighted directions algorithm again produced the smoothest result, but also that the edge directed algorithm produced a nearly equivalent result. Therefore, we concluded that these two algorithms are the best suited for our system, depending on the performance requirements, using the edge directed algorithm as a default.

In the next chapter, we will look into the stitching module of the recording pipeline presented in chapter 2. We will see that the choice of stitching algorithm can greatly affect the final panorama, and that several steps can be taken to reduce visual artifacts.

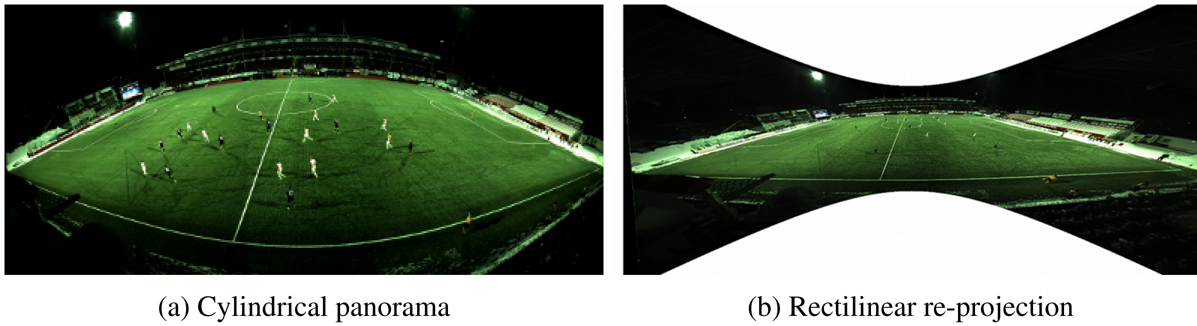
Chapter 4

Cylindrical Panorama Generation

In this chapter, we will explain the panorama generation part of the pipeline. We will be basing this on the panoramic stitcher module presented in the old Bagadus system [10], and consider the chapter a natural continuation of this work. However, in chapter 1 we discussed problems with the homography stitch of the Bagadus system. We will present an alternate method of projecting the images before stitching, creating a *cylindrical panorama*. Cylindrical panoramas have existed for many years and this type of projection is likely the most common when creating panoramas with a horizontal field of view greater than 120°. The panorama in our system has a field of view of ca. 160°.

Naturally, a lot of existing software has been developed for the creation of panoramas, especially in the recent years of digitized photography. Liu *et al.* details a proposed algorithm for cylindrical image projection on GPU in [82]. Here, a broad algorithm for cylindrical stitching is given, which has greatly influenced our work, but does not provide an implementation or algorithm for each required step. Many applications, such as Panorama Tools [83], allow users to manually create panoramic views from digital images. Fully automatic solutions, such as AutoStitch [84], can use feature matching to correctly warp and align the images into a full panorama without user interaction. This provides a fast way to create panoramas that are taken using commodity cameras, e.g., modern mobile devices. These methods are, however, typically designed for single images and require significant processing for creating each individual image. When working on single images, it is acceptable to spend a couple of seconds to create a single panorama, which is not the case when recording real-time video. Most applications are also difficult to configure automatically, designed for a particular workflow that does not fit our recording pipeline. Libraries such as OpenCV [76] provide some support for creating cylindrical projection mappings, as well as automatic stitching functionality, but this can be challenging to configure and extend for video. The automatic stitching in OpenCV is far from real-time, as was explored in [11]. Therefore, we believed we would achieve a better final output by implementing this creation ourself.

In this chapter, we will first present the motivation for transitioning from the rectilinear homography stitch of the old Bagadus system to a cylindrical panorama. We will then present the algorithm for creating the cylindrical panorama. We will show how this process can be performed offline, allowing the use of a lookup table for real-time stitching both on CPU and GPU. We will then present a continuation of the dynamic stitching process presented in the old Bagadus system, as well as detailing the implementation used in our recording pipeline. Finally, we will evaluate the presented work and conclude the chapter.



(a) Cylindrical panorama (b) Rectilinear re-projection

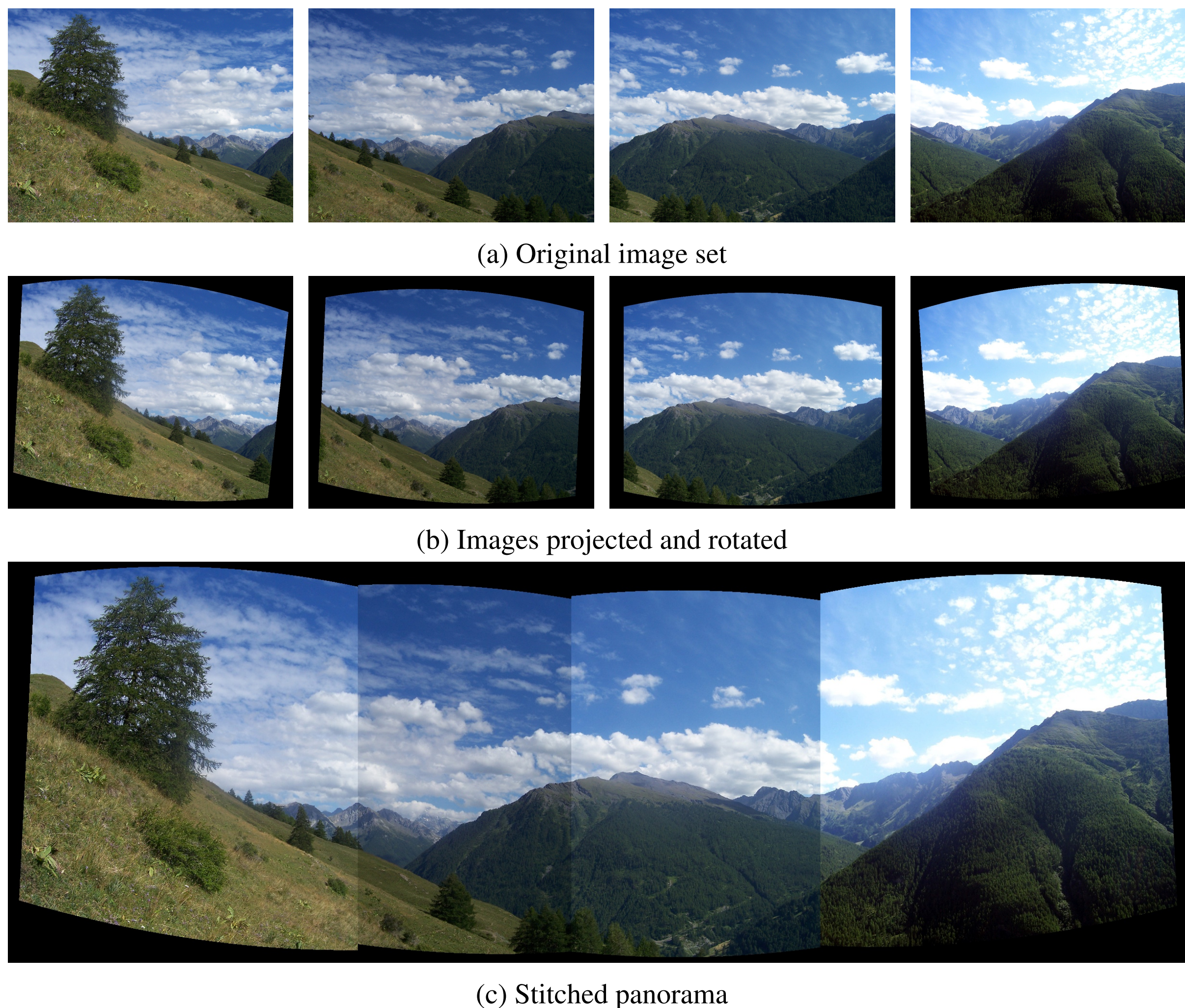
Figure 4.1: Example showing how a cylindrical texture can be re-projected into a rectilinear panorama. Note how the horizontal lines bend after the cylindrical projection in 4.1a, but appear straight in 4.1b.

4.1 Motivation

In chapter 1 we described several limitations of the homography stitched panorama generated by the old Bagadus pipeline. This pipeline produced a panorama that had an incredibly high horizontal resolution, as the edge pixels were stretched out due to the large field of view. Such a planar projection is unable to create a 360° panorama, in fact anything above 180° is impossible. According to H. Woeste in [18], the limit to a typical rectilinear panorama is believed to be around 120°, horizontally and vertically. Attempting to expand the field of view further in either direction will result in extreme warping effects, as the edges of the panorama are stretched into the correct perspective. The panorama video will then be significantly wider than required, which makes it difficult to discern objects far away when viewed in its entirety. Given that our prototype system requires a field of view of approximately 160° to cover the entire soccer pitch, we believe that this type of panorama is not well suited. A cylindrical panorama allows for a full 360° horizontal field of view, and introduces less warping of the original input images at more than 120°. This means that more of the original raw image quality is preserved, and the pixels are not artificially expanded through warping. The resulting cylindrical panorama contains significantly less needless data towards the edges of the panorama.

The distortion is based on the field of view of the source image. If a single image covers a large field of view, it will cover a greater portion of the cylindrical surface, and will therefore be distorted further. In our scenario, we have a relatively small horizontal field of view per camera, approximately 37°, due to the rotated positioning. Images will also not appear natural if the vertical field of view is large, i.e., above 120°, and in these situations it is common to use a spherical projection instead of a cylindrical panorama. Despite the rotational positioning, our cameras only have a 67° vertical field of view.

Cylindrical panoramas are created by stitching together multiple images that have been projected onto the inside of a cylindrical surface, as seen in figure 4.2. Imagine standing at the center of a large cylinder, and having each of the images projected onto the surrounding wall, creating an effect similar to that shown in figure 4.1b. Each image is projected separately, but when stitched together form a large panorama. When the cylindrical surface is unrolled and viewed as a simple planar surface, the image will appear distorted. Horizontal lines become curved, while vertical lines remain straight, as can be seen in figure 4.1a. The negative effect of bending horizontal lines becomes very apparent given the straight lines of a soccer pitch. However, in this thesis we will be focusing on using the panoramic video as an intermediary



(a) Original image set

(b) Images projected and rotated

(c) Stitched panorama

Figure 4.2: Example of individual images projected and stitched into a panorama. Source images from [85]

representation. The primary motivation for transitioning to a cylindrical panorama is the development of the re-projection method, allowing for personalized virtual viewing, that we will describe in chapter 5. This opens up multiple new possibilities of post-processing, as we will also discuss. Figure 4.1b shows how the cylindrical panorama can be re-projected onto a planar surface, creating a rectilinear panorama in post-processing. The originally bent horizontal lines are reconstructed and appear straight, yielding an output quite similar to that presented in the old Bagadus system. However, this method of re-projection can also create an infinite number of possible views, constructed from the same original cylindrical panorama.

4.2 Cylindrical Projection Algorithm

In order to create the final cylindrical panorama, each input image is projected separately onto a cylindrical surface. This projection involves virtually placing each input image outside the cylinder, its plane tangential to the cylindrical surface, seen in figure 4.3. We define the camera location at the center of the virtual cylinder. Every pixel on the cylindrical surface, i.e., the final

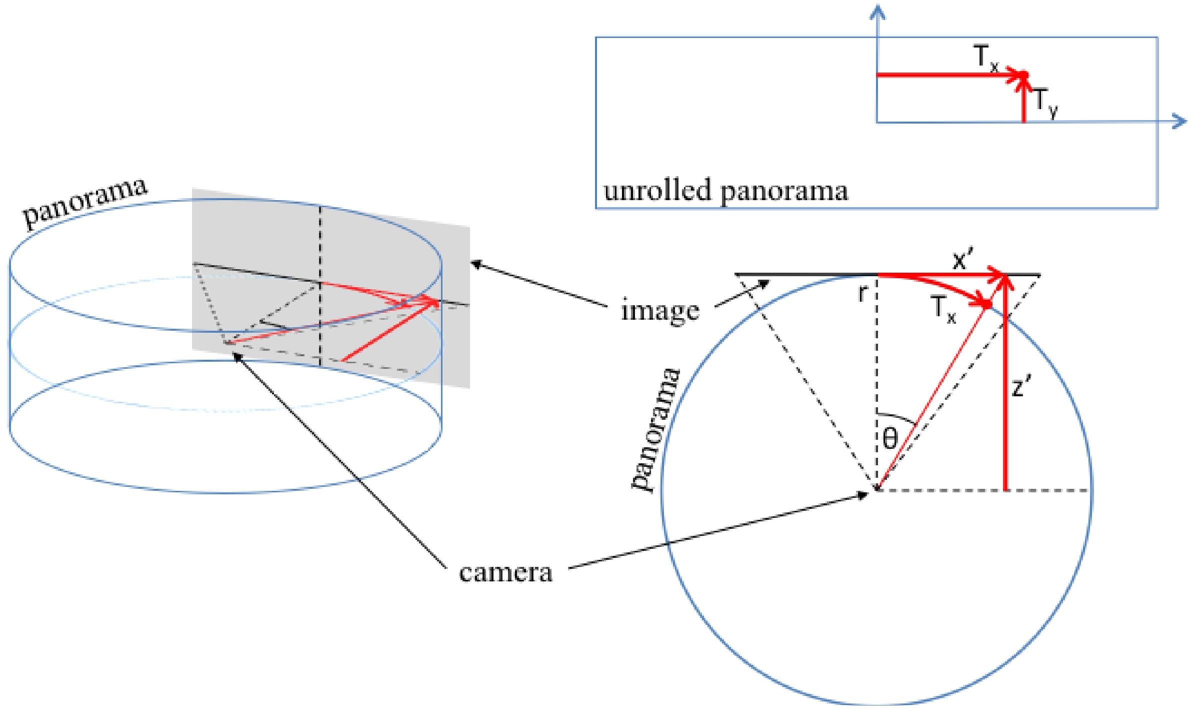


Figure 4.3: Creating panorama pixels from a captured image

panorama image, can be determined by tracing a ray from the camera center point, through the point on the cylindrical surface, and find its intersection with the input image plane.

The cylindrical surface is internally represented as a regular unrolled image, which can be stitched together based on the original physical orientation of the cameras during capture. To project each input image, the algorithm must know its pixel dimensions and its horizontal field of view. In our target system, all input images are of the same specifications, which simplifies this process. The radius (r) of the cylindrical texture we want to project onto is defined by the relation between the pixel width (W_{input}) and the field of view (fov), shown in equation 4.1.

$$r = \frac{W_{input}}{2 * tan(\frac{fov}{2})} \tag{4.1}$$

The field of view is determined by the video camera, which in our pipeline is 37°. The width is determined by the vertical resolution, 1080 pixels, as our cameras are rotated 90°.

To stitch the images together we must know the horizontal angle of each cameras viewing axis, in relation to each other. We assign each input image a horizontal angle (α), representing the angle between a center point (center of the output panorama) on the cylindrical texture and a ray from the camera center through the center of the input image. In our system, the center camera is assigned $\alpha = 0°$.

In order to avoid missing pixels in the output panorama, we perform a backwards projection. This means that every pixel coordinate (T_x, T_y) on the unrolled cylinder determines the corresponding horizontal (θ) and vertical (ϕ) angles of a ray from the camera center through this coordinate, shown in equation 4.2.

$$\theta = \frac{T_x}{r} \quad \text{and} \quad \phi = arctan\left(\frac{T_y}{r}\right) \tag{4.2}$$

Based on these angles, we can determine where this ray intersects with one, or more, of the input image planes. When creating a simple static seam, we select the source image with the closest α value to θ for each pixel. We then subtract the α value from θ, essentially centering our coordinate system on this image, instead of the original center. The point (x', y', z') in 3D space where the ray intersects the image plane can then be determined by equation 4.3.

$$\begin{aligned} z' &= r \\ x' &= tan(\theta) * z' \\ y' &= tan(\phi) * \sqrt{z'^2 + x'^2} \end{aligned} \tag{4.3}$$

The (x', y') coordinate gives us the specific pixel in the original image, corresponding to the pixel coordinate (T_x, T_y) on the unrolled cylinder.

4.3 Projection Algorithm with Corrections

This algorithm requires each image to be perfectly aligned and rotated. The camera mount presented in chapter 2 provides a good alignment, but there are small misalignments that must be corrected. It is impossible to align our images perfectly to create a smooth panorama without these corrections. The most common approach to this is to project each image separately, and then warp the result to fit together before stitching [82]. This may require a second interpolation step, and we desire a direct mapping from our algorithm. Therefore, we add some parameters that can align the images during the projection phase.

A small, per-camera, vertical correction is easily applied with no additional complexity. We can also very easily adjust the focal length, i.e., the radius of the cylinder, slightly for each camera to provide slight scaling adjustments. Most importantly, however, is the rotation of each individual image. When the image is placed tangential to the cylindric surface in 3D space, we also perform a small rotation of the cylindrical surface, independently for each input image.

A standard rotational matrix [86] is used to compensate for rotation around the cameras x, y and z axis, created by matrix multiplication of the matrices defined in equation 4.4.

$$\begin{aligned} Rotation_x\,(\alpha) &= \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\ \alpha & sin\ \alpha \\ 0 & -sin\ \alpha & cos\ \alpha \end{bmatrix} \\ Rotation_y\,(\beta) &= \begin{bmatrix} cos\ \beta & 0 & -sin\ \beta \\ 0 & 1 & 0 \\ sin\ \beta & 0 & cos\ \beta \end{bmatrix} \\ Rotation_y\,(\gamma) &= \begin{bmatrix} cos\ \gamma & sin\ \gamma & 0 \\ -sin\ \gamma & cos\ \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{4.4}$$

The Cartesian coordinates of each pixel on the cylindric surface must now be multiplied with this rotational matrix, i.e., virtually rotating the cylinder before we project onto it, creating the coordinate (x, y, z). We must then find θ and ϕ based on these coordinates, as defined in

equation 4.5.

$$\theta = arctan\left(\frac{x}{z}\right) \qquad (4.5)$$
$$\phi = arctan\left(\frac{y * sin(\theta)}{x}\right)$$

Note how, unlike equation 4.2, we can no longer rely on the radius as the focal length, increasing the complexity of this calculation. However, once we have found θ and ϕ we can determine the intersection point (x', y', z') in the input image by equation 4.3.

4.4 Offline Lookup Table

The algorithm described above has several key complexity issues that makes it challenging for real-time application. Each pixel in the output panorama must be uniquely projected, i.e., we cannot use a simple linear matrix transformation. Trigonometric functions, such as sine, cosine and tangent operations, are computationally expensive, and multiple such operations have to be calculated per pixel. Given a high resolution panorama, e.g., 4096×1680 pixels, this is a challenging algorithm to perform in a real-time system. In table 4.1, we see the computational cost associated with these operations on a modern CPU.

We believe that this algorithm would be very difficult to perform in real-time on CPU, but could likely be possible on most modern GPUs. However, we have not taken significant steps towards optimizing this, either on CPU or GPU, as we determined quite early that it proved more efficient to utilize a static pixel lookup table, generated once in an offline capacity. This is made possible by our static camera configuration. By creating the lookup table once, we avoid spending valuable resources on computing the same static projections repeatedly. Therefore, we perform the above algorithm to project and align the images into a correctly stitched panorama offline. The resulting mapping between the input images and the output panorama can be written to file and used during real-time recordings.

We use a custom created program to configure and align the panorama, modifying the parameters mentioned earlier. The user is free to modify these values through a simple GUI program, giving an immediate visual representation of how the panorama will look. This can be seen in figure 4.4. It is based on still images, utilizing the OpenCV image processing library [76] for temporary image management and user interaction. This program is created to simplify the process of creating a new setup, which can be done in a few minutes. Once the user is satisfied that the generated panorama is created correctly, the configuration and lookup table

$x \times x$	$sin\ x$	$cos\ x$	$tan\ x$	$arctan\ x$	$\sqrt{x^2 + x^2}$
59.23	1251.36	1241.67	3026.89	971.12	511.12

Table 4.1: Mean time (ms) measurement of trigonometric functions performed on 25000 random numbers on CPU, with regular multiplication provided for reference. Note that simply iterating through all the values, a constant operation for all measurements, takes approximately 18ms.

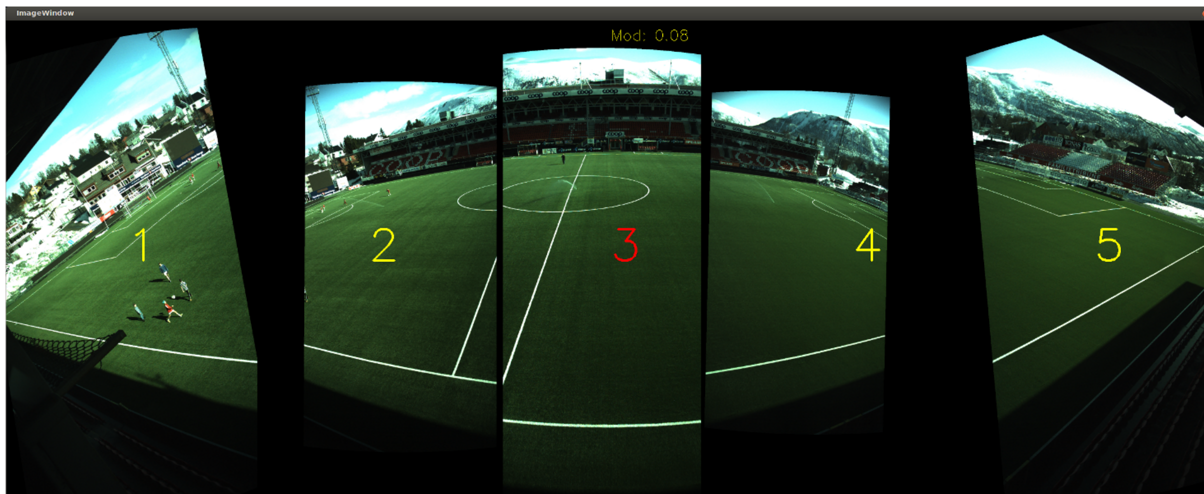


Figure 4.4: Screenshot from the program created to generate the lookup table and manually configure the panorama. The images can be moved, rotated and scaled to create the best possible mapping

is saved. Because the program is executing only on CPU, it typically takes approximately 300 milliseconds to generate the panorama based on user input. As this is too slow to be considered a truly interactive program, it allows the user to work on a downscaled, or upscaled, version if so desired. This may also be an important feature for those users who do not have a 4096×1680 resolution monitor.

4.5 CPU Stitching Prototype

Our initial prototype executed on the CPU, using a very simple lookup table. The use of a lookup table allowed us to stitch the full panorama in real-time, at 25 frames per second, without requiring a GPU. However, due to very tight performance constraints it lacked several intended features, performing nearly no computations during the live recording.

This implementation featured a static vertical seam between each of the input images, and each pixel coordinate was specified with integer accuracy, i.e., nearest neighbour interpolation. For every pixel in the output panorama, we performed a table lookup to determine which source image and pixel should be copied. While this seems fairly trivial, and should be extremely fast, we need to evaluate the data access pattern for this operation. We explained in chapter 2 that our cameras are rotated 90° during the recording, which requires that we rotate the source images before stitching. In practise, we simply modified our pixel indices when accessing the source images, as this is typically considered faster than performing a separate image rotation. However, this method still provides terrible CPU caching. When performing multiple read operations in a large array, good caching is vital to provide good performance. Based on testing, we have seen that simply reversing the indices when iterating through a two dimensional array, e.g., an image, can cause a 1000% increase in execution time, despite performing the exact same calculations and end result. In our scenario, we can fill the pixels vertically instead of horizontally, i.e., allowing us to read the pixels sequentially from the source images. However, each input image is warped by the cylindrical projection and usually at least slightly rotated.

This causes the benefit to be negligible. The data access pattern created by the lookup table simply is not well suited for modern CPU caching, regardless of implementation.

When using one thread per input image (five), this prototype was able to process approximately 35 frames per second. The resources required significantly affected the video encoder, however, which in this system could only handle approximately 33 frames per second as a result. This implementation could likely have been further optimized, but we elected to port the operation onto the GPU in order to extend its functionality and free up valuable resources.

This implementation was used for the creation of the dataset presented in [21], and has been used for the development of several of the systems we will present in chapter 5.

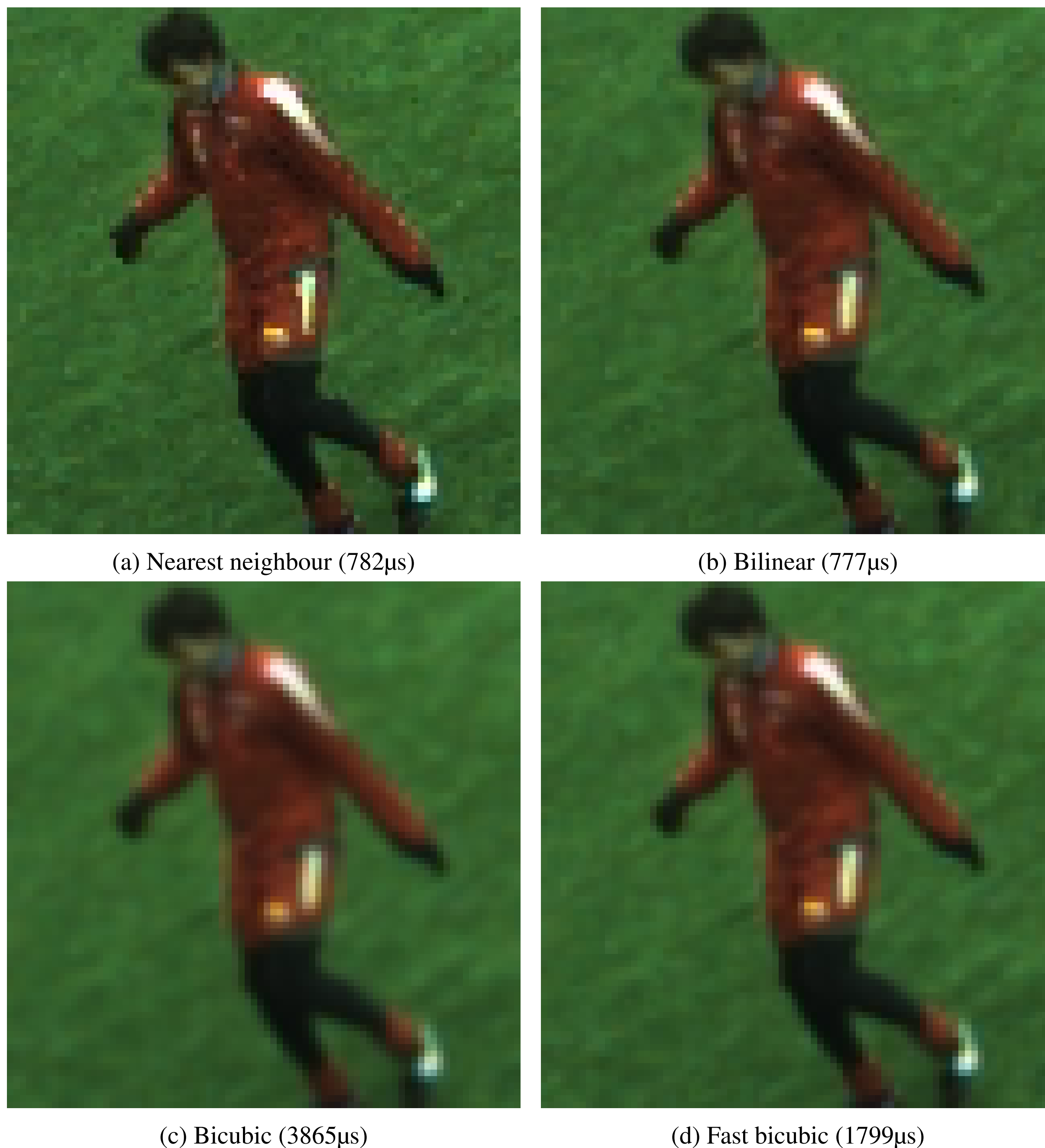
4.6 GPU Stitching Prototype

Despite a working real-time CPU stitching prototype, we wanted to port this over to the GPU, as in the old Bagadus system. This allows us to free up CPU resources and provide a more scalable system that can handle higher frame rates, more cameras and higher resolution. Additionally, there are several functionalities that were impossible in the CPU implementation that may now be possible. On the GPU, we actually have the resources to calculate the image projections in real-time, as this is a highly parallel algorithm. We believe this to be unnecessary, however, and decided to stick with a lookup table, as it still requires significantly less resources.

We mentioned in the previous section that modern CPU caching techniques provide very poor cache utilization with the memory access pattern required by our lookup table. On a GPU, we have the option of utilizing texture memory to address this issue. We explained in section 2.6 that textures allow for two-dimensional caching, which can increase performance of algorithms with unusual (from a CPU standpoint, but not from an image processing standpoint) memory access patterns. Therefore, we use textures for both the lookup tables and the source images.

The short access times when using textures also allow us to perform proper interpolation, instead of being limited to nearest neighbour as we were on CPU. In fact, CUDA textures support linear, i.e., bilinear for 2D textures, interpolation in hardware. This means that we now use a floating point lookup table, as opposed to rounding each pixel coordinate to the nearest integer. Due to the speed of the bilinear hardware interpolation and efficient caching, there are only minor performance differences compared to using a single non-interpolated lookup, essentially yielding an almost free quality improvement. In fact, we see from figure 4.5 that it takes longer to round the coordinates into integers, than to perform bilinear interpolation. This figure shows the visual result of four different interpolation methods. It may be difficult to discern from these images, but the nearest neighbour interpolation results in high levels of noise, e.g., in the grassy areas. With only a little zoom, the image will appear grainy and strong aliasing is present around edges. Bilinear interpolation creates a smoother image, with less noise, and reduces interpolation artifacts such as aliasing. Bicubic interpolation creates an even smoother result, typically also with greater apparent sharpness.

The CUDA Samples [87] features several examples of optimized bicubic interpolation methods, provided by Nvidia. We use a fast bicubic interpolation function, as shown in figure 4.5d, based on four bilinear texture lookups. This produces nearly the same result as basic bicubic interpolation, but at significant performance gain. Using bilinear interpolation also creates a good visual results, but we believe that the gain from the fast bicubic interpolation is worth the extra millisecond of runtime per frame.



(a) Nearest neighbour (782µs)

(b) Bilinear (777µs)

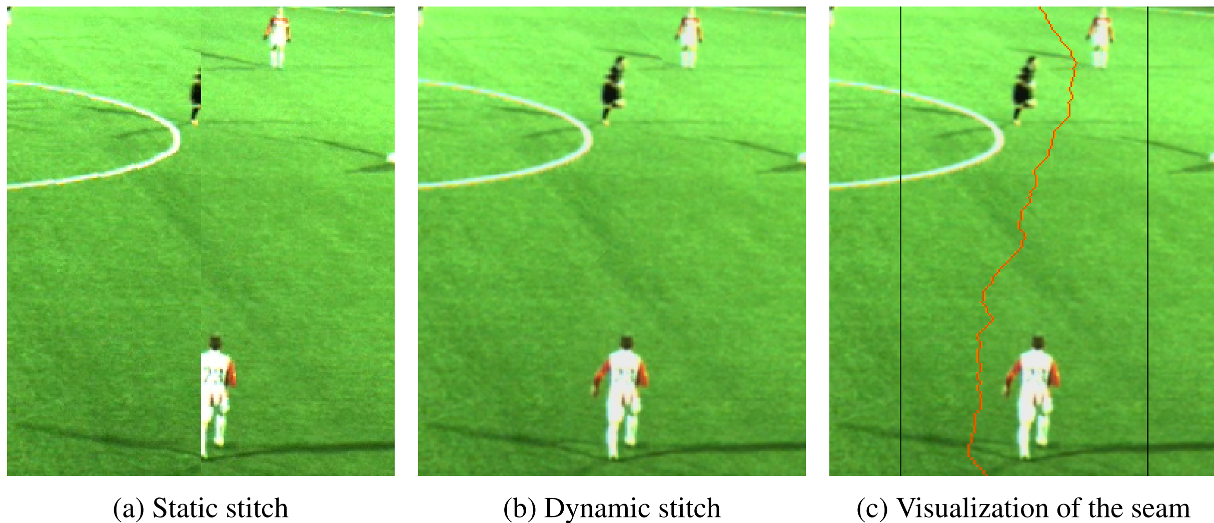
(c) Bicubic (3865µs)

(d) Fast bicubic (1799µs)

Figure 4.5: Comparison of quality as a result of interpolation in the GPU stitching module, zoomed by 300%. We also show the average time required to project the original images, based on the GPU lookup tables and interpolation.

4.7 Dynamic Seam

The old Bagadus system presented the idea of a stitcher module with a dynamic seam in [12], in order to provide a better visual result when players moved near the seam, as can be seen in figure 4.6. Between each of the input images, an overlapping seam area is analyzed to create an optimal slicing between the two images, without creating visual artifacts as a result. These visual artifacts can be highly noticeable when zooming into the panorama, or when players



(a) Static stitch (b) Dynamic stitch (c) Visualization of the seam

Figure 4.6: Example of two players being cut in half when moving through a static stitch, and how a dynamic stitch adapts to avoid this. The red line indicates the created seam, and the black lines indicate the seam area.

move slowly through the seam.

The stitching algorithm in the old Bagadus system treated each pixel within each seam area as nodes in a directed graph, with each node pointing to the three nodes in the next row, i.e., straight up and diagonally up to both sides as shown in figure 4.7. This ensured that when traversing the graph, you would always be moving one step vertically. Dijkstra's shortest path algorithm [88] was used from the bottom of the seam to the top, creating a minimal cost route for the seam, based on a custom weight function. The custom weights were calculated based on the absolute difference between the color values in the two images, as well as the foreground mask created by the background subtraction module in the same system.

In this thesis, we have continued this idea, re-implementing it to better fit our system and improved upon the original algorithm. In order to allow for dynamic stitching, we must modify our lookup table to include a seam area. Essentially, we create five different lookup tables, one for each input image, and define which pixels of the output image they represent. These will have a variable sized overlap, depending on configuration, that can be used for dynamic stitching. Our camera setup features significantly smaller overlap between each image, which

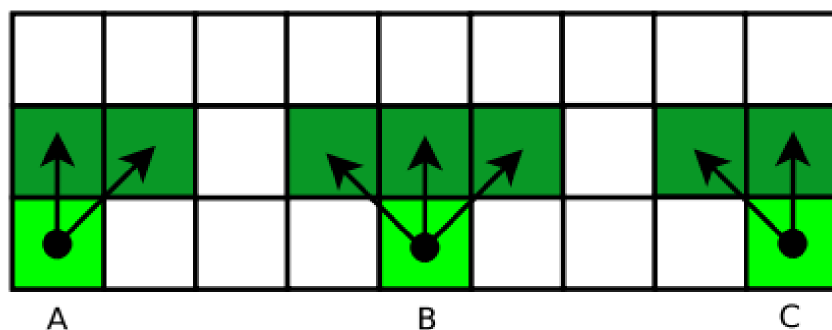


Figure 4.7: How nodes in the graph are connected. B shows a normal pixel with three outbound edges, A and C shows the edge cases only connecting the two adjacent nodes. Figure from [31]

restricts our seam areas to about 170 pixels wide. While this is more than was used in the old Bagadus system, we do not have a freedom to extend the area dynamically as was proposed in the future works section of Espen Helgedalsrud's thesis [31].

It is also worth noting that our current system and method of stitching allows for significantly better static stitches. While there are still some artifacts when using a static stitch, they are not nearly as prominent. As a result, most images in this chapter that visualize the dynamic stitching process have been intensionally misaligned.

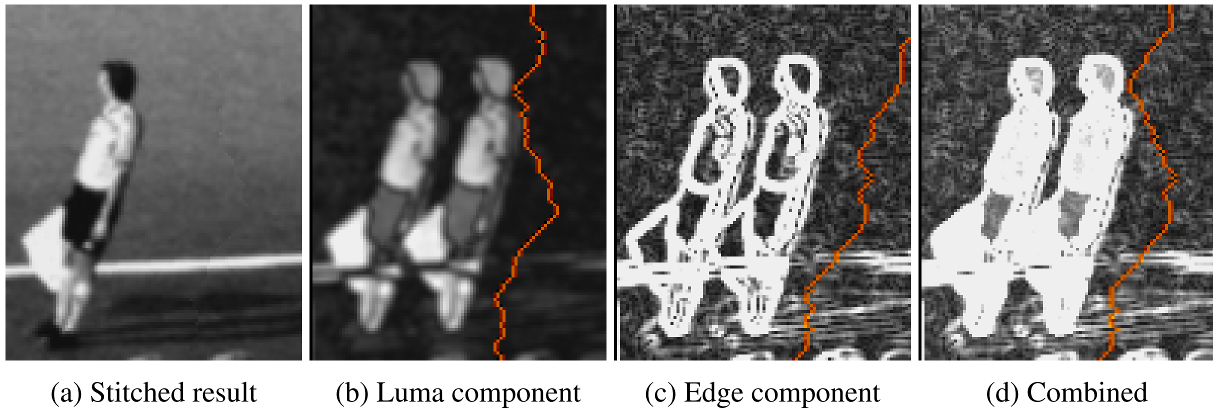
4.7.1 Graph weight calculation

The stitching algorithm from the old Bagadus pipeline relied on the foreground mask created by the background subtraction module, as well as the absolute difference between the color values of two overlapping pixels. We believe that the inclusion of a background subtraction module for this sole purpose is unnecessary in our new pipeline. Looking at only the absolute difference of pixel values can, however, cause problems in situations where the two input images are not equally exposed, i.e., great color differences between the images, or when players are fairly close to the cameras with a homogeneously colored jersey. We believe this to be a minor problem, but we would like to see if perhaps alternative weighting functions can result in better stitches in certain situations.

In our pipeline, we use the YUV colorspace for input data, which has the benefit that the Y-component (luminosity) carries significantly more visual information about a pixel. We believe that the luminosity value of each pixel is sufficient information to provide good weighting. There may be situations where a single sample is inconclusive. We therefore extend our pixel difference function to use a 3×3 pixel grid, also noting the absolute difference between the eight surrounding pixels in each image. This increases the probability of correctly determining if the overlapping pixel areas are in fact homogeneous.

The original algorithm treated all of the RGB color channels equally when determining absolute color difference, which does not accurately represent the visual spectrum. A downside to using luminosity, however, is the fact that intense green and white have similar levels of luminosity. Black, on the other hand, is significantly different from green. In some overexposed situation, i.e., sunlight directly on the green soccer pitch, the algorithm would prioritize avoiding shadows instead of players with white jerseys. This was certainly a problem with the old node weighting in RGB colorspace as well. When there are only small changes, however, the luminosity component does create the smoothest seams possible, in the absence of players.

With the absence of a foreground mask to identify players, we needed a more effective method for avoiding players than just luminosity. Specifically, we want to determine if an object in one image is in fact present in the other at the same pixel coordinate. A simple approach to this is edge detection, i.e., determining the presence and intensity of an edge in each pixel. We therefore add an edge detection component to the cost calculation as well, using the absolute difference between the edge-intensity of the two pixels in question. That means that if a strong edge is present in one image, but not in the other, we do not wish to create a seam in this pixel. Typically, this will produce a strong edge contour around each object which will need to be crossed at high cost in order to stitch through this object. This edge component is significantly better at creating high costs for misaligned players, lines and other objects. However, we still use the luminosity information because it produces a much smoother transition in homogeneous areas.



(a) Stitched result (b) Luma component (c) Edge component (d) Combined

Figure 4.8: Visualizing the cost components used for dynamic stitching. White areas indicate a high cost of stitching through this pixel

Edge detection is performed with Sobel's method [80], using the Scharr operator, as described in section 3.9.1. The horizontal and vertical edge gradients are calculated from the luminosity component in a 3×3 grid in both images, and the absolute difference between the gradient values in the two images are used to determine pixel cost.

The components used for calculating the node weights can be seen in figure 4.8, where the cost of each pixel is represented with white intensity. We clearly see the outline of the person in both images, as both the luminosity component and the edge component increases the cost in these regions. We also see that the edge component is weighted higher than the luminosity component, as it is typically more reliable for object detection. This is deemed more important than locating smooth homogeneous areas of overlap, which is where the luminosity component produces the best result.

4.7.2 Seam fluctuations

An important problem with the old dynamic stitch algorithm is the visual fluctuations created by calculating the seams independently for each frame, i.e., ignoring any frame-to-frame relation. While the seam created may in fact be the best possible option for each still image, this may not be the case when viewing the video output. If the seam changes, especially drastic changes, several times per second it is almost guaranteed to be a visual annoyance to the viewer. Situations when there are two paths very far apart, often surrounding some object, that are nearly equal in cost are quite common. This can cause the seam to jump back and forth, from one side of the seam area to the other, at a high rate. Any misalignment or small color difference between the two images will amplify the visual impact of this effect.

While it is essential that the stitch remain dynamic, we want to reduce the probability of small fluctuations causing widely different seams from frame to frame. A small cost is therefore applied to all nodes in the graph that were not a part of the previous seam. The added cost increases linearly from zero, based on the absolute distance to the previous seam, capped at 15 pixels.

The extra cost must be extremely small, negligible when compared to stitching through a player, in order to not reduce the efficiency of the dynamic stitcher in each frame. However, we found that the additional cost required to remove the seam fluctuations is extremely low, and

does not reduce the algorithms ability to create a smooth seam. Earlier, we mentioned that the primary cost is composed of a luminosity component and an edge gradient component, with the luminosity component having the smallest impact. The luminosity component is calculated based on the absolute difference of the two images, computed from a 9 pixel grid. Comparatively, the added cost of moving the seam more than 15 pixels to the left or right is equivalent to these two luminosity grids having a combined 8 values difference. Given that this is spread over at least a few pixels, this is not visible to the human eye. This shows that the added cost is extremely small, negligible when compared to any form of movement.

The effectiveness of this method may deteriorate in some situations. If the soccer pitch is not well lit, the cameras use a higher gain to digitally boost the signal. This introduces noise to the image, especially visible in normally homogeneous regions. In these cases we still see some seam fluctuations, as small differences become amplified into visible noise. This can be improved by balancing the cost components if we were to target recordings with these light levels. High dynamic range video, as presented in chapter 2, may also opt to boost the original values and introduce noise that can increase seam fluctuations. However, in either of these cases, the fluctuations are still rarely visible.

Seam fluctuations are most visible outside the soccer pitch, near the top and bottom edges of the panorama and the opposite stands. This is because there is typically a lot of small movement in the crowd, an overall lack of homogeneous areas and that we prioritize the soccer pitch when aligning the images. Therefore, we also added the possibility to not perform a full search when calculating the seam. Instead, we opt to ignore a configurable number of pixels at the top and bottom of each stitching area. This range can be set independently for each seam. This means that the seam will be statically set to the previous computed values, for the top and bottom regions. The dynamic seam will then be forced to continue from where the top seam ended, and forced to meet where the bottom seam begins.

We still wish to perform full range path finding at a regular interval, e.g., every five seconds. This is because some seams may not be optimal for long term use, for example because camera exposure configuration takes several seconds to stabilize. Another benefit to using a limited search range is that we reduce the time required to compute the dynamic seam.

4.7.3 Seam blending

The stitching algorithm relies upon the images being perfectly color corrected. The old Bagadus pipeline had a color correction module to ensure this [89]. In our pipeline, we chose to remove this module due to the efficiency of the exposure synchronization of the cameras, as described in chapter 2. This does mean that small variations in overlapping areas may occur. One example of this is vignetting, i.e., a reduced brightness near the edges of the image compared to the center, introduced by the cameras. This is something we see from our prototype system, primarily in bright weather. Therefore, we also perform seam blending to reduce this effect. Great differences are, however, not common in our prototype system, and all figures presented in this section have been modified to increase the lighting difference.

Seam blending is a very common step when performing image mosaicking, stitching multiple images together, even after color correction. Some stitching algorithms will simply perform slight blurring along the seam after the stitching, e.g., through Gaussian filtering [90, p. 265]. This can even out the color differences, but may look unnatural when focusing on objects within the blur due to the loss of detail. A more complex method is pyramid blending [90, p. 463],

(a) Stitch without blending

(b) Stitch with blending

Figure 4.9: Example of two images with large color differences, showing the effect of performing a post-stitch blending. Note that the algorithm is not intended for as large differences as shown in this image, yet still provides a smooth transition in the lower areas where the color differences are smaller

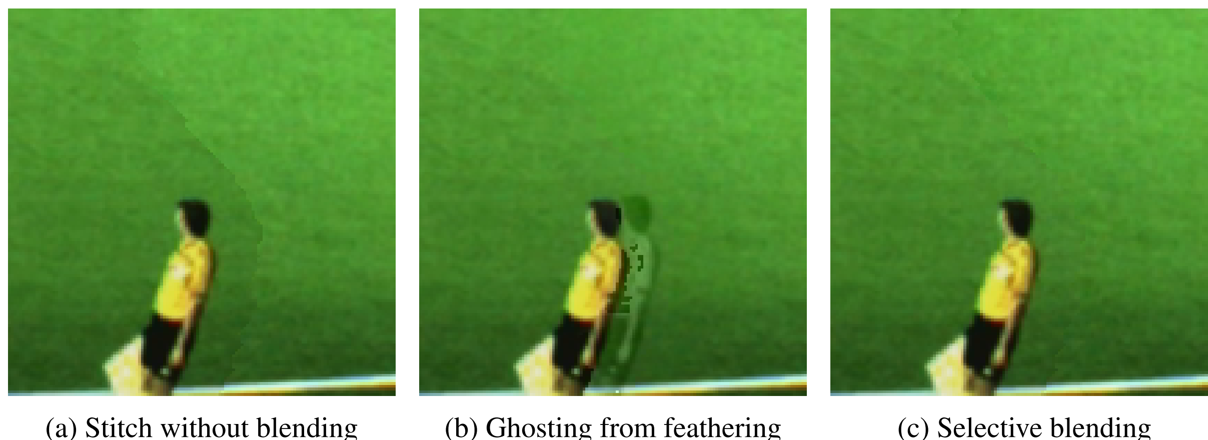
which involves downscaling the seam area into several lower resolutions using a Gaussian filter, before applying a Laplacian transformation on the lower level of the resulting image pyramid. This creates a smoother and more natural looking blend, especially for images with widely different color values, but is computationally expensive. As we expect the images to have very slight color differences, we deem this algorithm too expensive for our purposes, as well as introducing too much loss of detail.

Feathering is a very simple approach that performs a weighted average between the two overlapping images, as shown in equation 4.6. Here i_{max} represents the size of the blending region, and i the index within the region.

$$Output(x,y) = \frac{i}{i_{max}} \times Img_a(x,y) + \left(1 - \frac{i}{i_{max}}\right) \times Img_b(x,y) \tag{4.6}$$

We may not have sufficient overlap if the dynamic seam runs right along the edge of the overlap zone, in which case we shift the blending region to the left or right, producing the same effect. This is guaranteed to create a smooth transition, but may introduce ghosting effects if the images are not perfectly aligned. This can be seen in figure 4.10, where the feathering effect tries to average two widely different areas which results in ghosting.

We propose to perform a selective blend, as seen in figure 4.10, where we ignore the pixels that are widely different. Instead of only looking at the two overlapping pixels when averaging, we determine a single average light difference within a localized blending area. Additionally, we filter out the pixels that have a high edge cost component in the path finding for creating the dynamic stitch, as well as any pixel with too different luminosity. For calculating the average difference within the blending area, we also extend the region vertically, creating a small rectangle around the original seam coordinate. This ensures that we have several samples to judge the overall luminosity change in situations where there are a large amount of filtered pixels. When performing the feathering algorithm, we then do not use the difference at each specific



(a) Stitch without blending (b) Ghosting from feathering (c) Selective blending

Figure 4.10: Example of ghosting artifacts introduced due to feathering

pixel, but rather the total average of the entire localized region. In perfectly aligned images, this will likely produce a worse result than regular feathering, but we remove the potential for ghosting effects. In a worst case scenario, where not enough samples are found, we get the original non-blended seam.

We found it sufficient to perform the blending only in the luminosity component, as we primarily want to focus on removing very slight differences in exposure in the two images. There are also details in implementation that allow this algorithm to remain quite efficient, which would be slowed down significantly by introducing blending of the chromatic channels as well.

4.8 Implementation

The real-time GPU implementation of the panorama stitcher requires one lookup table per camera, with sufficient overlap to perform dynamic stitching. It receives five input images in interlaced YUV format with four bytes per pixel and its output is the downsampled and planar YUV422p.

4.8.1 Individual image projection

The first step is to project each of the input images onto its own cut-out of the panoramic image, performing a table lookup for each pixel in this image. By modifying the indices returned from the table lookup, we can perform an inexplicit rotation of the source images. These indices have sub-pixel accuracy, allowing for bicubic interpolation from the original input texture.

This step also performs the downsampling of the chromatic channels, averaging the chromatic values of two horizontally adjacent pixels. Therefore, each thread retrieves two pixel values in one kernel iteration, and the resulting two YUV pixels are written in planar format. We use regular global memory arrays for this intermediate image projection instead of surface memory, as we are able to coalesce our memory accesses well in future kernels.

This projection can be performed in parallel for all five input images, executing one kernel per image in a separate CUDA stream.

4.8.2 Dynamic seam cost creation

The first step to Dijkstra's shortest path algorithm is determining the weights of each node in the graph. We store this in an array of custom node structures in global memory, as we are able to coalesce the accesses of this graph perfectly whenever handling the graph. Each pixel coordinate within the overlapping region between two images is represented by a node, and each overlapping region can be computed in parallel. We create one thread per node to compute the cost.

The cost of a node is determined by an edge component and a luminosity component. For each node, we first gather the luminosity values, Y-component in YUV, for each pixel in a surrounding 3×3 grid from one image into a local array. Then, we calculate the horizontal and vertical edge gradients for this pixel. We re-use the local array to gather the equivalent grid from the second image, continuously increasing the cost based on the absolute difference between the two corresponding pixel values in each image. The edge gradients for the second image is then determined and compared to those of the first image, increasing the cost based on the absolute difference. Finally, a small cost is added if the node was not a part of the previous seam, before the node is written back into global memory.

4.8.3 Dynamic seam path finding

Once the cost has been computed for all nodes, we can apply Dijkstra's shortest path algorithm. This is difficult to perform efficiently on a GPU, due to the data dependency between a node and all of its predecessors. We create one thread per horizontal node, i.e., the width of each stitching area, which iterates through the graph from the top of the image to the bottom. Specifically, for each iteration we determine the cost of the current node, based on the nodes own cost and the lowest cost of the node's three predecessors. The result is written temporarily into shared memory for the three successive nodes in the next iteration, while the optimal child node is saved back in the original graph. In case of graph boundaries, an infinite cost is instead set to ensure that this path is never selected. We use global memory to store the graph, as we can perform completely coalesced reads and writes, which consists of the local pixel cost and which predecessor node is cheapest. Additionally, we use shared memory for temporarily storing the three predecessor costs for all nodes in the current and next row. This limits the number of required global memory transactions to two 128 byte reads and two 128 byte writes per warp for each iteration.

Once all threads have reached the end of the image, a single thread determines the best final node and iterates back through the graph to locate the seam. Although this is not parallelized, we saw that it had an overall very small impact on the runtime of the kernel. The seam is written as a single array, saving the x-coordinate for each y-coordinate in the graph.

4.8.4 Filling in the panorama

While the dynamic seam is being calculated, we perform a two dimensional memcpy operation in parallel, copying each of the projected views into the final panorama asynchronously. This is done through the utility function *cudaMemcpy2DAsync*. Because we are copying internally on the device, this is an extremely quick operation (<100 µs). In the overlapping areas, only one image is copied. Once the dynamic stitch has been computed, we iterate through all the pixels in the seam area again, launching one thread per pixel. If the seam is on the wrong side of this

pixel, i.e., the already copied data originates from the wrong image, we overwrite the pixel with the one from the correct image. The reason we do this in two operations in this way is due to the speed of the 2D memcpy operation, which may be overlapped by executing kernels.

4.8.5 Blending the seam

The final blending step is optional, as the panorama is already complete at this point. Here, we launch one thread per seam-pixel, performing a modified feathering operation on the horizontally adjacent pixels. First, we determine the luminosity difference between all pixels within a small rectangular blending region, e.g., 40×20 pixels, possibly shifted if the seam is far to the left or right within the overlapping area. This is a very expensive operation, though part of the computations have already been computed during the cost creation step. When we determined the cost of each node, we also wrote the luminosity difference into a different surface array. This also allowed us to easily filter out all nodes whose edge-component was too expensive.

We then determine the average of all the values within the blending region, filtering out those pixels whose difference is above a given threshold. Finally, the modified feathering algorithm is applied, adding the average value multiplied with the feathering weighting factor to each pixel in the center row of the blending region.

4.8.6 Summary

In the last two sections, we have detailed the steps performed for each frame in the panorama pipeline. This has been implemented on GPU, using CUDA.

Individual image projection. The first step, we explained, was the projection of each of the input images. We needed to modify the existing static lookup table, to allow for overlap when stitching the images together. We use a quick bicubic interpolation method, provided by Nvidia, to achieve a better visual result when fetching the original pixels. Here, we perform a downsampling of the chroma channels, and convert the image to planar format. The result of this step can be seen in figure 4.11. The second part of this step is to copy all images into the same array, without stitching the overlapping areas, as seen in figure 4.12

Figure 4.11: Initial individual image projections.

Figure 4.12: After copying all projections into a single image, without stitching

Dynamic seam cost creation. In the second step, we calculate the cost used for the seam pathfinding in all the overlapping regions. We explained that we only look at the luminosity values of each pixel, and calculate the cost based on the local mismatch between the overlapping pixels. The mismatch is calculated from the absolute difference between the 9 spatially closest pixels, and from the difference in edge gradients calculated from the same 9 pixels. In addition, a cost is added for those pixels far away from the previous seam, in order to reduce seam fluctuations. The resulting cost can be seen in figure 4.13.

Figure 4.13: Visualization of the cost in each overlapping stitch region. White areas indicate a high cost. Note the thick transparent black "seam" in each area, the location of the previous seam that has a lower cost for nearby pixels.

Dynamic seam path finding. The third step is to perform Dijkstra's shortest path on the graph filled in by the previous step. Here, we explained that one thread is responsible for calculating the cost at each horizontal location, for each of the seam areas. These threads must synchronize between each vertical step, as they move from the top of the image to the bottom. The line of

the graph that is currently being worked on is read into shared memory, with double buffering, to limit the required global memory accesses. The thread, at any vertical pixel, determines the optimal predecessor node, based on the result of the previous iteration, and writes the result into shared memory to be used in the next iteration. Finally, one thread backtracks the graph and fills in the optimal path. The resulting path can be seen in figure 4.14.

Figure 4.14: Visualization of the resulting seam, based on the cost in figure 4.13. This has been cropped to easier visualize the seam.

Copy stitched. The fourth step consists of evaluating each pixel in all the seam areas. We have already copied the data from one image into this area, so we merely copy those pixels that landed on the wrong side of the seam. Specifically, if we look at one of the seams from figure 4.14, the pixels to the left of the seam originate from the original memcopy operation (figure 4.12), while the pixels on the right side were copied over in this step. The resulting panorama can be seen in figure 4.15.

Figure 4.15: Final panorama, after applying the dynamic seam, before blending is performed.

Blending the seam. The final step is to blend the images together, to smoothen out any minor differences in luminosity between the two images. We explained that a modification of basic feathering is used, avoiding pixels that may cause artifacts if directly averaged. This is done by finding the average difference in luminosity between all the homogeneous pixels within a blending region of 40×20 pixels, around the original seam. This average value is then used to perform a gradual feathering operation to blend the images. The final output panorama, as it is delivered from the stitching module, can be seen in figure 4.16. Figure 4.17 also shows a cropped section where the seam was originally visible, but nearly undetectable after the blending operation.

Figure 4.16: Final panorama, after performing seam blending. Note that we can still see some differences between the images, primarily between the third and fourth image. Seam blending will hide this when zooming in, as can be seen in 4.17, but the color difference may be visible when viewing the full panorama.

Figure 4.17: Highlight of the area where the dynamic seam was most visible, before and after the blending.

4.9 Execution Performance

In this section we do a performance evaluation of the stitching module executing exclusively on the GPU device. The full module uses an average of 5.99 milliseconds per frame, or 166 frames per second, taken from a sample run of 4000 frames. This produced a 4096×1680 pixel panorama from five input streams of 2040×1080, with 168 pixels of overlap, i.e., stitching area, between each image. This shows that the stitching module is running well below the 20ms deadline for fifty frames per second.

The dynamic stitcher module needs to perform several sequential steps to produce the final output panorama, though some of the steps can be performed in parallel. The single slowest CUDA kernel is the creation of the dynamic seam, performing Dijkstra's shortest path on all the seams. The seams can be computed in parallel, but the algorithm is still slow because it requires a synchronization step for every row in the graph. Additionally, we can only launch one thread per column. This algorithms requires very few resources, but is consistently slow. The initial image creation, projecting each of the input images, takes approximately the same amount of time because of the large amount of data that must be handled, as well as the bicubic interpolation of the original pixels. Performing the blending of the seams is also quite slow, depending on the size of the blending region. With a blending region of 20 pixels on each side of the seam, we will need to iterate over 400 luminosity values, for each of the vertical pixels along all seams (1680×4), to determine the average used for blending. This is a quite costly operation.

In figure 4.18, we see an overview of the mean time required to complete each step of the stitching module, with error bars showing minimum and maximum offsets. We can see that when each step is combined, we nearly reach the actual runtime of the module, aforementioned at 5.99 milliseconds. There is some required overhead between each step for synchronization

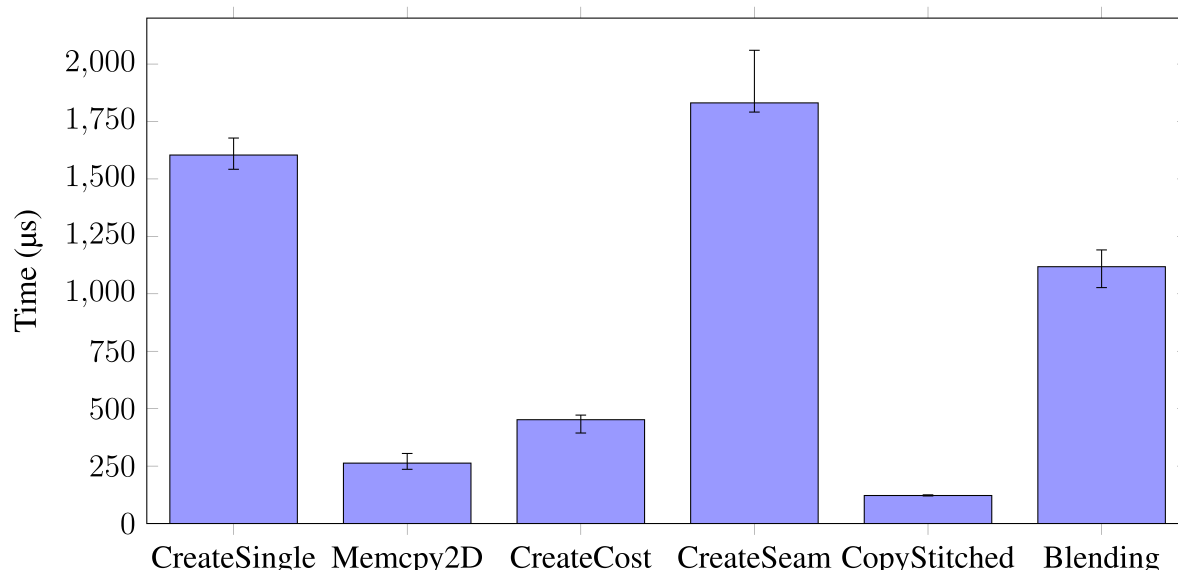


Figure 4.18: Performance plot of each of the steps for stitching the final panorama, performed on a GeForce GTX 680 GPU. Here, we show the average execution time over 4000 frames, with error bars showing min/max, for each sequential step. The panorama of 4096×1680 is created from five cameras, i.e., four seams, and the horizontal size of the seam area is 168 pixels.

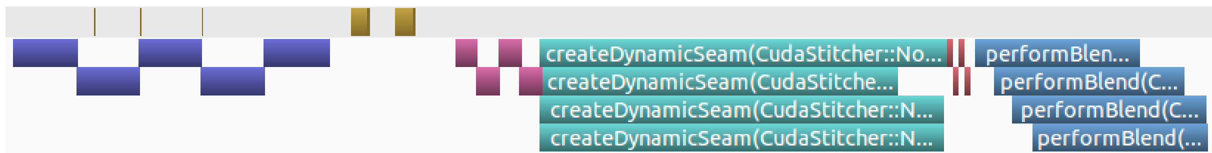


Figure 4.19: Profiling timeline of each stitching step. The top row (orange blocks) shows the 2D memcpy operations, while the second row shows each kernel. The kernels, from left to right, are: CreateSingle (blue), CreateCost (purple), CreateSeam (teal), CopyStitched (red) and Blending (blue).

and small CPU operations, as we need to ensure that the previous step is complete before beginning the next. Kernel block executions can be scheduled very efficiently, but there is a small CPU overhead to launching a kernel of approximately 40-160 microseconds.

It is important to note that all of these kernels are computed per seam, or per image, concurrently. In reality, we could see from profiling results that CUDA schedules the kernels to run nearly sequentially, as the device can only execute a limited number of threads at any given time. Figure 4.18 is based on the time of each step, from the launch of the first kernel to the termination of the last. Some of the kernels, however, overlap due to using very few threads, specifically the kernels responsible for creating the dynamic seam and performing blending. Figure 4.19 shows the timeline, extracted from Nvidias Visual Profiler [43], of the execution of all kernels. Here, we see that the kernels are scheduled quite efficiently, with some time in between individual kernels for synchronization. We see that some kernel executions overlap, while others fully occupy the GPU.

Creating the dynamic seam uses only 168 threads, one per horizontal pixel within the stitching area, meaning that all four seams can be calculated completely in parallel. The blending operation also uses fairly few threads, allowing four kernels to overlap. Looking at figure 4.19, we see less overlap, however, as the time between the start of the first kernel and the end of the final kernel is about 50% longer than the kernel running time. We are unsure exactly why this consistently occurred, but the previous kernel, CopyStitched, has a very short execution time which can introduce some overhead from CPU scheduling which can propagate to the launch of the blending kernels.

The two dimensional memcpy step is a little challenging to present accurately. It is perfectly consistent, in that each operation takes either less than 1 µs or approximately 80 µs, with some synchronization time in between. Each memcpy can also, theoretically, be performed while other kernels run, but we consistently saw that CUDA refuses to schedule more than a couple of these operations while kernels are executing. This can be seen in figure 4.19, as the first three short memcpy operations overlap with launched kernels. However, we see that the card is idle for a long period between the two first sets of kernels. We could therefore say that the memcpy step takes approximately 700 µs, since that is on average how long the card is inactive or copying data. There might also be something specific to our configuration, e.g., badly aligned addresses, when performing this test, since the same two memcpy operations would always be 100 times slower than the remaining eight. We are unsure why this occurs, but we saw that it was consistent across multiple program executions.

In figure 4.20, we show the actual utilization of several resources available on the GPU, for each kernel. Note again that this is per kernel, and that the dynamic seam and blending kernels are able to execute four kernels concurrently. The percentage utilization when four

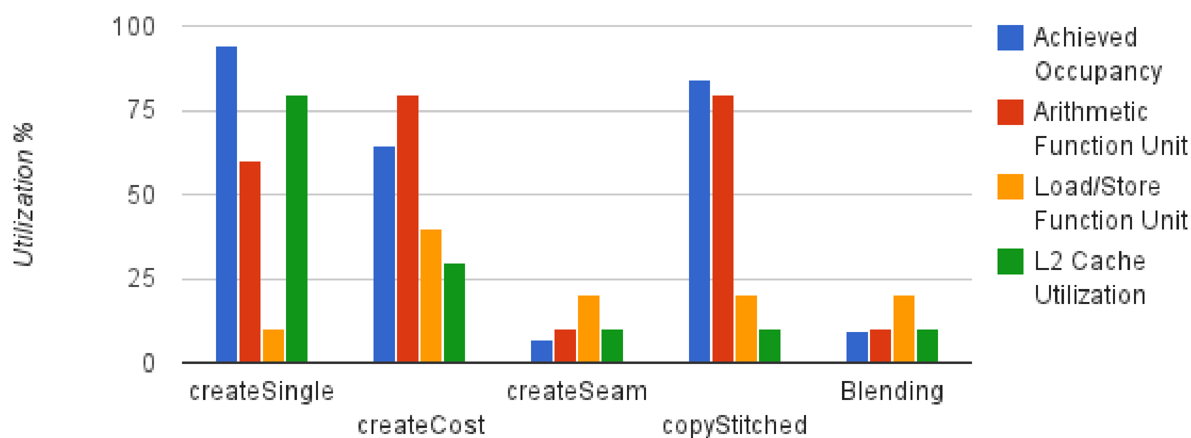


Figure 4.20: GPU utilization of kernel used for stitching. Occupancy shows how many threads, compared to device maximum, execute in true parallel (per kernel, multiple kernels with low occupancy may execute concurrently). Utilization of the function units shows how many, compared to device maximum, are utilized on average during the execution of the kernel. Cache utilization is determined by the profiler tool, as a combination of cache hit percentage and cache size.

kernels execute simultaneously should be quadrupled as a result. We see that the cost creation kernel is unable to achieve a very high occupancy level, even though its theoretical occupancy is 100%. This means that multiple threads are forced to stall, unable to execute. This can be, for example, because of synchronization between threads or branching code. In this case, we saw that at least some of this was due to required branching. Having a high percentage in each of these metrics is typically a good thing, but it depends highly on the workload. We see that the blending kernel performs a very high number of load and store operations, attributed to having to fetch a very large region, without performing a lot of computations for each value read. This is perhaps a source for future optimization. The initial image projections perform very few lookup operations, but work with large amounts of data and a consistent memory access pattern. This results in very high cache utilization. When looking at the cache utilization, we see that this is overall fairly low across the other kernels. This is because we primarily use global memory, which is not cached as efficiently as texture memory. However, having a good cache utilization is not vital if the memory accesses are well coalesced within each thread warp.

In section 4.7.2, we mentioned the ability to reduce the path finding search, re-using the same seam several frames in a row in the top and bottom of the panorama. While this is primarily aimed at reducing video flickering, it also has the potential to reduce the time required to compute the dynamic seam. In this section, we have only evaluated full range searches because we argue that the maximum runtime is more important than the average in a real-time system. The benefit in performance is also highly dependant on the number of pixels to ignore, which may vary across configurations. In our default configuration, we ignore a combined 540 pixels for each of the two outermost seams and 350 pixels for each of the innermost seams, computing the full seam once every 200 frames. This reduces the average runtime of each individual *CreateSeam* kernel call by 25.5%, which leads to an average 7% reduced runtime for the full module. In our example, the seam searches are not equally long, which results in different runtime for the different seams. However, as soon as any kernel is complete we can immediately

begin with the next kernel, i.e., run *CopyStitched*. This is because the path finding uses so few threads that other kernels could execute in parallel, but must wait for the resulting seam before they can be launched.

4.10 Summary

In this chapter, we have looked at the creation of the cylindrical panorama and the panorama stitching module of our new Bagadus recording pipeline. We have discussed the primary benefits and drawback of a cylindrical panorama, as opposed to a rectilinear panorama. We have seen that a cylindrical panorama will preserve more of the original image quality, causing overall less image distortion, as the field of view becomes higher than approximately 120°. We explained that a cylindrical panorama is created by first projecting each image separately onto the inside of a virtual cylindrical surface before stitching these projections together into a full panoramic view.

We then detailed the algorithm for projecting and aligning the images correctly, highlighting its computational complexity. We explained that this is difficult to perform in real-time, but can be easily achieved with a static lookup table generated offline. We then explained how an early CPU prototype was implemented, and the associated challenges of achieving real-time stitching. We saw that the memory access pattern created by the lookup table caused the stitching module to run slowly, despite performing few computations online. When porting the stitching module to GPU, implemented using the CUDA framework, we saw a significant increase in speed. We explained the importance of the two dimensional texture caching and available hardware interpolation.

We then looked at the concept of creating a dynamic seam instead of a regular static seam, as was presented in the old Bagadus pipeline. By assigning a high cost for areas where two overlapping images mismatched, we saw how we could apply Dijkstra's shortest path, or path of least resistance, to produce a smooth seam to avoid these areas, despite flaws in the original static lookup table. The cost for placing the seam at a particular pixel was determined by the difference in local luminosity in both images, as well as the difference of edge gradients. We saw that the edge component proved effective at avoiding players, while the luminosity component selected the optimal path in homogeneous areas. Additionally, we saw that we could reduce seam fluctuations by increasing the cost of moving the seam from one frame to the next. A selective blending approach was able to create a smoother transition across the seam in situations with slight color differences in the two images.

Next, we detailed the steps required to implemented the dynamic stitching on GPU. We highlighted a lot of challenges and benefits associated with the CUDA architecture, showing how the presented algorithms can best be implemented. We showed the visual result of each step, from the initial image projection to the final panorama after blending.

Then, we evaluated the stitching modules performance, highlighting the time spent on each of the sequential steps required to construct the final panorama. We saw that the module performed well below real-time requirements, at an average of 5.99 milliseconds. We also saw that this can be reduced further to 5.57 milliseconds by only performing a full seam search every 200 frames, keeping the seam mostly static in the areas outside the soccer pitch. We explained that this produced less flicking video as well as improving performance.

In summary, our new panorama pipeline is able to stitch the 5 individual camera streams

in real-time, and generate a 4096×1680 panorama video live. However, as we discussed in chapter 1, the panorama itself is not necessarily the best way to present the information to the user, and we therefore present several video delivery systems in the next chapter. Common to these delivery systems is that they can extract virtual views (similar to a TV broadcast) from our generated panorama.

Chapter 5

Virtual View Video Presentation

In chapter 2, we described a system for creating a real-time cylindrical panorama, saved as encoded H.264 video segments, with further details provided in the previous chapter. In this chapter, we will explore how this panorama video can be used to create an immersive viewing experience by re-projecting the panorama into smaller dynamic virtual views. Such a view can be seen in figure 5.1.

We will also explore several possible systems that can be designed around the virtual viewing method. Because they are all based on the same recorded data, multiple systems can exist simultaneously and provide users with alternative solutions. Here, we will explore systems for entertainment purposes, as well as sports analysis.

The virtual views allow us to bypass regular single camera streams. This is why we elected to remove the single camera streams from the real-time recording pipeline. Instead, we can adaptively mimic individual streams by creating dynamic or static views that cover any area on the soccer pitch. Unlike the single camera streams of most systems, these will not be limited by the physical camera orientation or the number of physical cameras. Creating a few pre-configured static views allows for a scalable system, similar to many other systems where one can select one of a finite number of streams. However, more interesting scenarios allow for dynamic virtual views, potentially independent for each viewer.

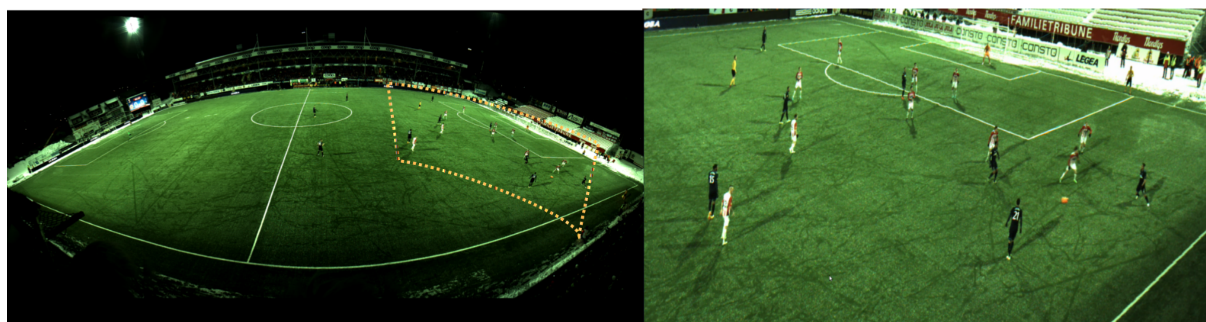


Figure 5.1: Panorama video with labelled ROI (left) and the virtual camera generated (right). It can be observed that it is not a simple crop from the larger video.

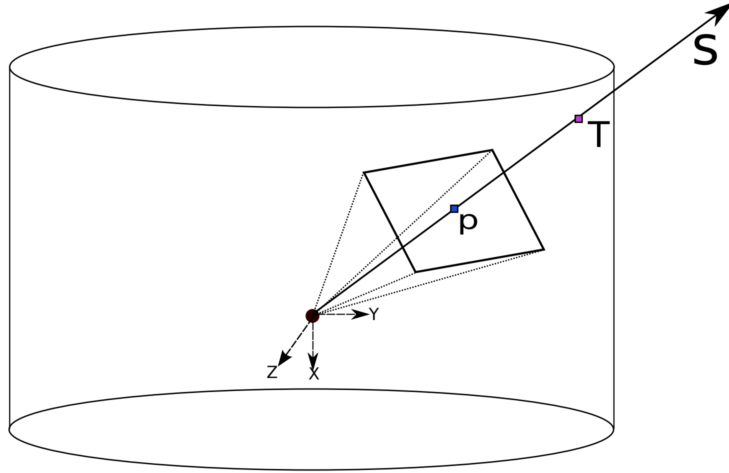


Figure 5.2: The intersection of the ray from the virtual view with the unit cylinder

5.1 Virtual View

The virtual view, as presented in [13], is created by performing a perspective projection of the cylindrical texture onto a planar surface. This process restores the perspective nature of the image, i.e., lines curved during the cylindrical projection step are corrected to again appear straight. The planar surface, i.e., output virtual view, is placed in 3D space, where a ray from each pixel is traced to the center of the virtual cylinder, as seen in figure 5.2.

A pin-hole camera for a point projection, from a 3D point P to an image point q, can be written in the following manner:

$$\lambda q = [K|0_3] \begin{bmatrix} R & 0 \\ 0_3 & 1 \end{bmatrix} \begin{bmatrix} 0_3^T & -C \\ 0 & 1 \end{bmatrix} P \tag{5.1}$$

where R is the general (3×3) 3D rotation matrix as a function of θ_x, θ_y and θ_z, the rotation angles around the x, y and z axes respectively, and K is the camera intrinsic matrix built with focal length f.

Let p be the current pixel in the virtual view. We need to find the ray that passes from the cylinders center, i.e., the perspective origin, to the pixel p. This ray can be represented by:

$$s = \lambda R^{-1} K^{-1} p \tag{5.2}$$

Then, the intersection of this ray with the unit cylinder gives us the exact position on the cylindrical texture. The intersection point (T_x, T_y), seen in figure 5.2, can be found as follows:

$$T_x = \left(\frac{W_p}{FOV} \right) \left\{ arctan \left(\frac{-s(1)}{s(3)} \right) \right\} + \frac{W_p}{2} \tag{5.3}$$

$$T_y = \left(\frac{1}{2} - \frac{s(2)}{\sqrt{s(1)^2 + s(3)^2}} \right) H_p \tag{5.4}$$

where W_p, H_p and FOV are the width, height and the field of view of the panoramic texture respectively. (T_x, T_y) are the coordinates on the unrolled cylindrical texture, i.e., panorama input image. The resulting output pixel value is the interpolated source pixel at (T_x, T_y).

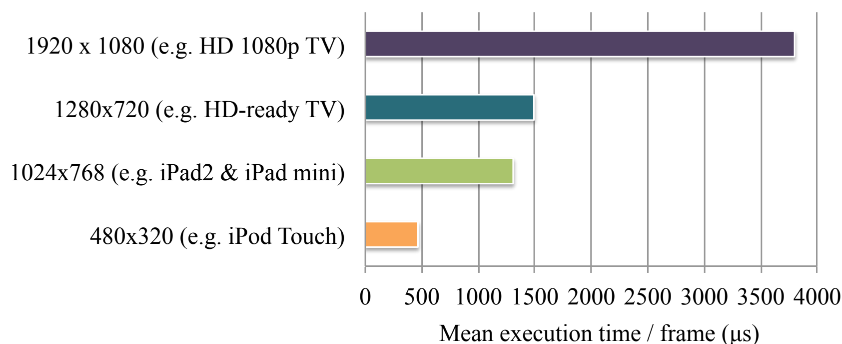


Figure 5.3: Core execution times for various resolutions

The virtual camera can move across the cylindrical texture by moving the output image plane, similar to a regular camera. Specifically, we orient the camera by modifying θ_x and θ_y, which represent the horizontal and vertical angle of the ray through the center of the virtual view. The center ray will always be a normal vector to the center of the cylinder, which means that the plane is also rotated automatically depending on the region of interest. The effect of panning the camera to the left or right is achieved by adjusting θ_x, while adjusting θ_y will tilt the camera up or down. The focal length (f), i.e., the distance from the camera and the plane, determines the level of zoom. This means that the program essentially only needs to modify these three values to provide users with zoom, pan and tilt possibilities for controlling the camera. This makes it fairly easy to provide input to the program, either from the user or an external system.

Generating a virtual view is also a fairly quick operation on a GPU, though there may be some static overhead of transferring the panorama onto the GPU device. The interpolation and ray intersection primarily scales with the size of the virtual view. In figure 5.3 we see the time required to generate the virtual view for several output resolutions. This shows that the system scales with the output resolution, potentially reducing the required computations by allowing a lower resolution.

For the remainder of this chapter, we will look at video extraction systems created around this method, which only modify the input values to the original application to generate the virtual camera.

5.2 Client Generated Views

The primary concern when creating a video extraction solution is its scalability. In the approach presented in [13], we allow each client to generate a local virtual view, built on HTTP streaming. Because each clients only fetches the panorama video, no major computations must be done on the server per client. This lets us utilize well explored HTTP streaming optimizations, such as data replication and caching, allowing potentially thousands of concurrent clients.

Each client will fetch the panorama video, which has been split into short segments for faster downloading, decode each video frame and transfer it to the GPU. Then, the virtual view is generated based on parameters interactively set by the user. The generated view can be directly displayed on screen from the GPU through OpenGL, without transferring it back into CPU memory. All of these steps can be pipelined for maximum performance. The user is free to pan, tilt and zoom the virtual camera, using multiple methods of input. For example, in the demonstration presented in [20] we allowed the use of a joystick/gaming controller to control the cameras movement, providing a smooth and intuitive method of control. In this

Figure 5.4: Virtual view demonstration presented at ACM MMSys 2014. Here, we allowed people to control the virtual view, created live from the two cameras at the top of the image. The panorama was created by one machine, while two clients downloaded it over a wireless network.

demonstration, we used two cameras, mounted at the conference venue, to create a panorama video live. Participants were then allowed to zoom, pan and tilt a virtual camera with less than 3 seconds delay. This can be seen in figure 5.4.

When each client generates the view locally, they are required to download the panorama video continuously from a server, in order to watch a live or previous recording. The scalability of the system is dependent primarily on the individual bandwidth capabilities of each client. The generated panorama video is significantly larger than most streaming solutions offer. Depending on the video frame rate and the encoders ability to compress the video (affected by light conditions, movement, precipitation and more) the generated files will typically be 2-8MB per 3 second segment. This yields a lower bandwidth requirement of 5.6Mbit/s and upper of 22.4Mbit/s, which can be deemed too high for users over a wide-area network. This can be seen in figure 5.5 where we show how this system performs using a loopback interface, i.e., the client and server is running on the same machine, a local-area network and a wide-area network. We see that wide-area network yields unstable results and is not guaranteed to stay within real-time requirements. However, this can be improved by utilizing adaptive HTTP streaming solutions, i.e., scale the video resolution based on the individual client's bandwidth limitations. Also note that, at the time of this experiment, the file sizes of the generated panorama were higher than those presented in chapter 2.

In order to allow for adaptive HTTP streaming, the video must be encoded multiple times, at different resolutions and possibly different levels of quality. However, this puts extra strain on the pipeline presented in chapter 2. We are therefore looking into alternative encoding methods to extend the pipeline, such as a GPU-based H.264 hardware encoder. This field shows great promise and hardware encoding has been shown to significantly outperform software CPU encoding [91], at essentially no computational cost since it is performed on a dedicated chip.

Another future work to explore, in order to reduce the data sent to the user, is tiled video,

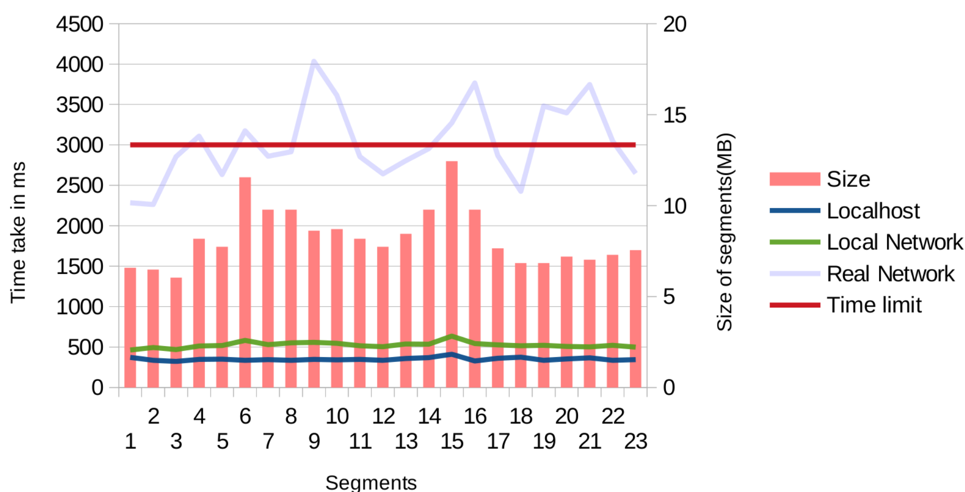


Figure 5.5: A plot showing the time required to download panorama video segments

such as presented in [3, 8]. This means that the panorama can be divided into smaller tiles of different bitrate. The client application could then request low resolution tiles for areas far outside the current virtual camera, maintaining maximum quality of the region of interest, and a lower quality temporarily if the user decides to rapidly pan the camera.

5.3 Server Generated Streams

The generation of an HD virtual view can be performed in real-time with any high-end or medium-end GPU, but this can be difficult to port to other platforms. Additionally, we mentioned the extremely high bandwidth requirements that can be associated with transporting the entire panorama video, a video that the user will likely not view in its entirety. By generating the virtual view on a remote server, we can significantly reduce the required bandwidth by only transmitting the smaller view, as it is presented to the user. This allows the video to be played back on most modern networks.

Server generated streams allow for more lightweight client applications. The client becomes a simple streaming application, with another layer of control for handling user input and transmitting that back to the server. Many devices with low computing capabilities, such as mobile phones, are still able to decode and display video, which can also be scaled down by reducing quality and resolution on the server-generated video. This video stream is then also much easier to display using current standard streaming technologies, i.e., played in a regular browser.

5.3.1 Static camera streams

The Bagadus system is primarily based on easy extraction of video, providing fast and easy access to pre-recorded data, originally intended for coaches to review situations for recent or ongoing games. The first online video streaming interface was presented in [10] as a system for extracting video from static camera streams based on a time-specific event. When an event was generated, either through an online browser-based interface or the Muithu system [92], the server would receive a POST-request over HTTP, specifying event information. A server

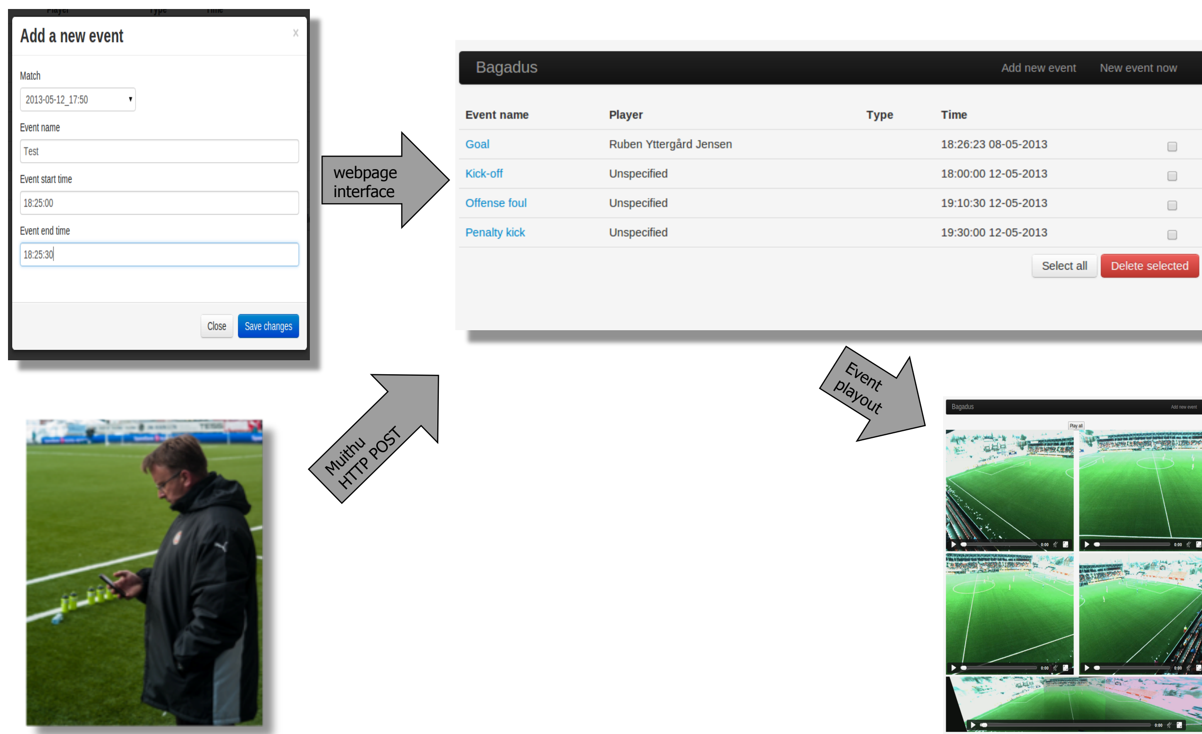


Figure 5.6: The online HTML5 interface used for expert annotated events. Here the events are sorted by player and then time.

back-end would locate the correct video segment, either during or post-recording, and create a short video clip for all cameras. Within approximately 10 seconds, four camera streams and the panorama from the 20 second event would appear online, ready for viewing. This is shown in figure 5.6, and through this system coaches or players could review video from a soccer game during half-time or right after the game.

This initial prototype has later been extended in several ways, primarily by allowing for immediate playback, real-time switching of cameras and tagging of players. This system uses the individual camera streams primarily, providing some (four) static views. The system is able to swap between camera streams in less than 100ms, but this speed requires all camera streams to be continuously decoded in parallel. This is not extremely scalable, as H.264 decoding of multiple video streams is resource demanding for the server.

With the camera configuration detailed in chapter 2, each individual camera stream is sub-optimal to view directly, because they have been vertically rotated and aligned for the panoramic stitching. Therefore, this system only produces a single panorama output. However, we can use static virtual views to integrate this system. This also means that we only need to decode the single panorama file, allowing an infinite number of virtual cameras to be added. This system can therefore be integrated with the virtual viewing method by defining a few views that cover various parts of the soccer pitch. Creating each virtual view on the GPU is relatively cheap once the input frame is already in device memory, which only needs to be transferred once for all output angles. This is also very easy to distribute, if we require more static camera streams than can be generated on a single machine, since we only need to duplicate the encoded panorama video through for example a shared file system (over a local network).

This system is focused on providing analytical data to the user, supporting video extract-

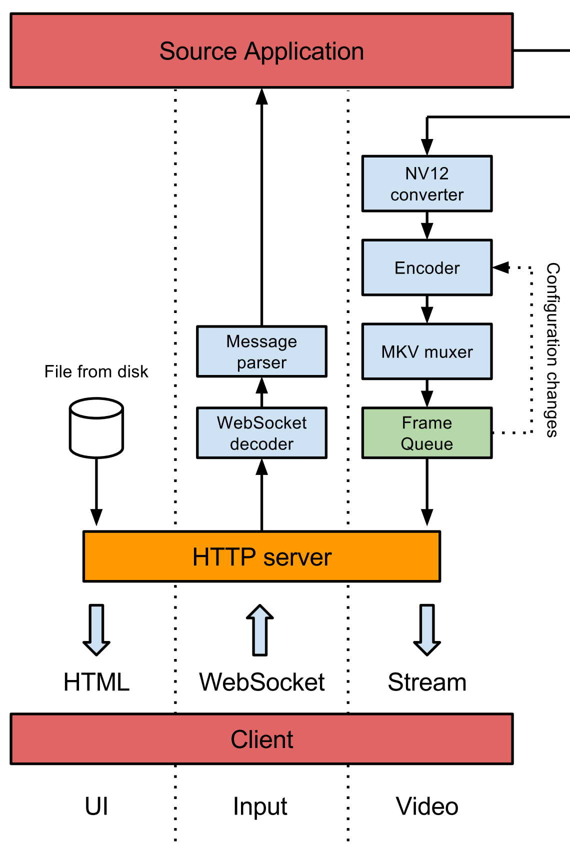


Figure 5.7: Block diagram of OpenGL streaming delivery pipeline

ing through complex SQL queries, based primarily on tracking information, as presented in the Alfheim dataset [21]. The web interface for handling these queries is currently under development and may integrate dynamic camera streams in the near future.

5.3.2 Dynamic camera streams

In order to create a truly immersive viewing experience, the user must be allowed to interact with the application, controlling the virtual camera personally. This is difficult to achieve on lightweight devices, with limited computing power. By generating the virtual camera on the server, the viewing application of the client merely has to download and present the live stream while handling user input. However, this is still a large challenge, as the stream must adapt immediately upon user input to remain interactive.

Such an application has been developed by Martin Alexander Wilhelmsen, detailed in [25, 91]. This system is designed for encoding and streaming any input OpenGL application, while at the same time receiving user input that is forwarded to the source application with minimum latency. This way, the same application for presenting the virtual camera can be utilized by the streaming server, providing user input over a network connection and delivering back the resulting OpenGL output in an H.264 encoded video stream.

An overview of this application can be seen in figure 5.7. User input, in the form of mouse movements and some keyboard commands, are sent directly to the server over an active WebSocket connection, where they are parsed and delivered to the virtual viewing application. The

resulting OpenGL output must then be converted into YUV and encoded using Nvidia's H.264 hardware-accelerated GPU video encoder, NVENC [93]. The encoded video is muxed into playable MKV [94] multimedia format and streamed back to the user application over HTTP. In fact, multiple users can connect to the same video stream and watch the same virtual camera, making this system also scale to a lesser extent. Multiple users is, however, experimental and becomes confusing when two people attempt to control the same camera movement.

This system is highly dependent on latency in order to provide the user with a good interactive feeling. Typically, real-time streaming applications will perform some buffering, but this becomes impossible when the users input is directing the video stream. The user will begin to notice the stream lagging behind if the result of his/her input is not apparent within half a second or less. The presented system is able to display the video with a latency of less than 100 ms + network latency. Network latency is hard to predict, and can vary greatly. This can be challenging in several scenarios, especially as this lightweight streaming approach is partly targeted at mobile devices operating on wi-fi networks. Several concepts are therefore applied to allow for slight variations in delivered frame rate, as frames arrive irregularly. The server must control the users render application directly, ensuring that each frame is delivered and presented on time. If a client begins to lag behind due to network issues, the server will adapt the stream to reduce the bit rate, or insert a keyframe if a frame is lost.

The streaming solution was tested with both libx264 and NVENC in [25], to compare the efficiency of these encoders. We saw that the system, using x264 as an encoder, was able to deliver two 1080p virtual camera streams, from different times in the source video, in parallel at 30 fps. When using NVENC, the system was able to deliver four parallel streams. This could likely be increased if the sessions were more tightly coupled, allowing for more sharing of resources, but it shows that it is possible to generate multiple HD streams on a server in real-time.

5.4 Automatic Camera

Our case study is soccer, which is primarily a passive (for the viewer) form of entertainment. While it is important that the virtual camera remains dynamic, having the user maintain constant control is not always desired. Therefore, we have also looked into automatically controlling the virtual camera to see if this can possibly create a pleasant viewing experience, on par with using a professional camera man. We present this fully in [22].

It is difficult to determine exactly what is the ideal area of interest in all situations, e.g., if a soccer player performs a foul far away from the ball. However, machines are typically quite adept at directly following specific targets, such as the ball or an individual player. However, providing accurate tracking information in real-time is still a difficult challenge. In the Bagadus system, we have tracking information from the soccer players available, and we are working towards tracking the soccer ball. Therefore, gathering the tracking data is beyond the scope of this application, and it assumes that this is provided accurately. The primary challenge when following a specific target is to create natural motion, avoiding small adjustments or too rapid acceleration. Continuously providing an adequate level of zoom is also difficult, as the machine struggles to determine whether the user would prefer an overview or a detailed view. The zoom level in this application was solely determined by the target location on the cylindric surface, zooming based on the relative distance to the original camera placement. However, both *smooth*

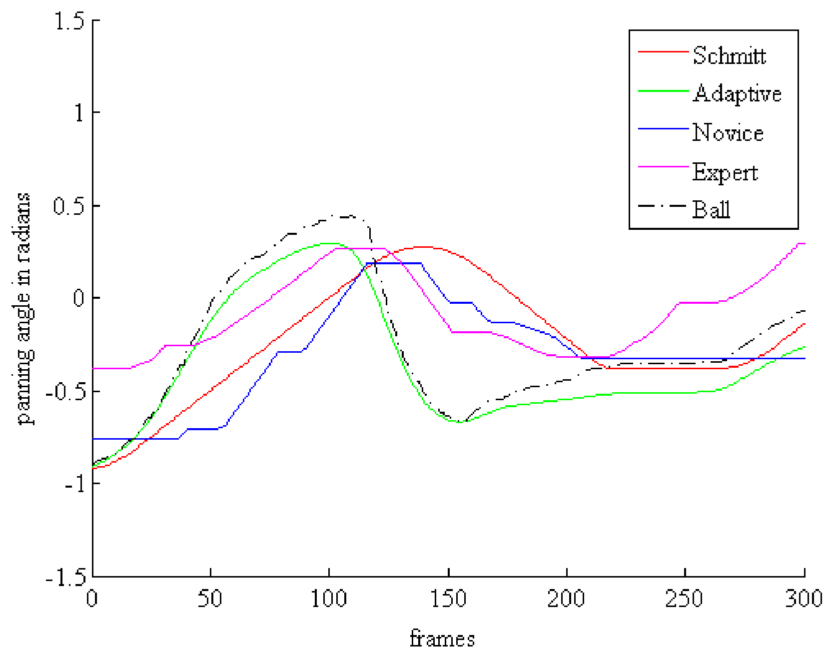


Figure 5.8: Schmitt trigger and adaptive trigger plots for 300 frame segment along with the plots from human operated camera.

zoom and *toggle zoom* were explored, as different methods of transitioning between zoom levels.

To control the panning and tilting motion of the virtual camera, two smoothing models were utilized. A *Schmitt trigger* uses a small virtual bounding box to determine the camera acceleration, accelerating the camera (to an upper limit) whenever the object is outside the box. When the tracking object again enters the bounding box, the camera motion decelerates. This was extended by using an *adaptive trigger*, to better estimate the required velocity of the camera. In this model, the video stream is delayed by one second, which allows the system to utilize future information to first smoothen the initial tracking data. This means that when the target moves rapidly, e.g., when tracking the soccer ball during a long pass, the camera can begin moving before the ball and spread the rapid movement over more frames.

The difference between these models can be seen in figure 5.8. Here, we show how the resulting panning angle as the models adapt to the tracking target, i.e., the ball. We see that the Schmitt trigger reacts much slower to changes, as it waits for the target to leave the bounding box. We also see the maximum acceleration is reached for multiple seconds as it attempts to catch up to the ball. We see that the adaptive trigger, however, begins moving before the tracking object, resulting in a smoother camera movement. These plots, however, cannot fully depict which algorithm performs visually best.

A user study was conducted to evaluate the quality of these models, as quality of experience is a highly subjective notion. We performed a visual stimuli test, presenting users with a series of short sequences. Each user was presented with the same combination of zoom and panning method four times in total. The results in figure 5.9 show the user preference towards each combination. We can clearly see that the majority of users preferred the smooth zooming and adaptive trigger panning model.

Additionally, we explored whether the automatically generated camera could outperform an expert cameraman controlling the system. This can be seen in figure 5.10. We see that the

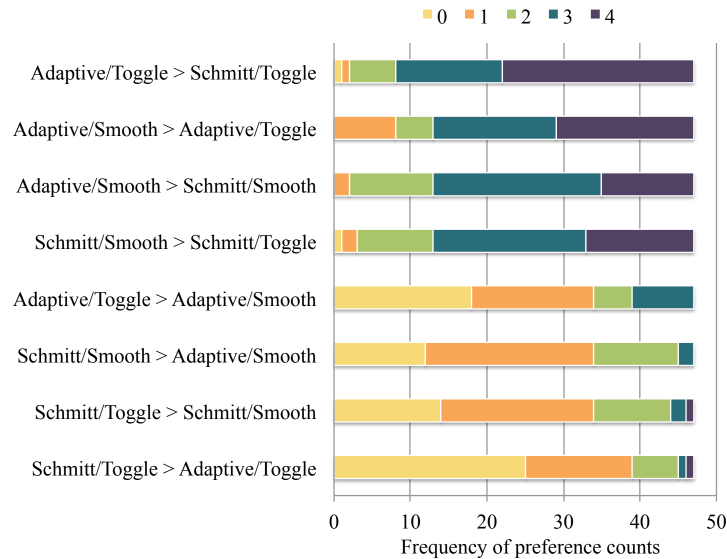


Figure 5.9: Frequency distribution portraying the number of times one stimulus was preferred over its contrast, accumulated across users. For example, in the first line, we see that 25 persons have preferred the Adaptive/Toggle over Schmitt/Toggle in all four repetitions. The maximum count of 4 corresponds to the number of repetitions for each pair of videos. Stimulus contrasts are sorted according to Friedman rank scores and plotted symmetrically.

majority of users preferred the automatic camera over both the novice and the expert camera man, showing that there is strong potential for automatic virtual cameras. We can picture a scenario where users are allowed to either control the video stream directly or toggle automatic following of the ball, or an individual player.

5.5 Summary

Earlier in thesis, we have explained that viewing a panoramic video in its entirety can be difficult. Even though the full panorama contains all regions of interest, displaying it directly may not be the optimal way to present the video to the user. Therefore, in this chapter, we have presented a virtual viewing approach that allows the user to pan, tilt and zoom a virtual camera. We discussed how this approach can be used in several systems for video extraction and presentation, along with their primary benefits, downsides and challenges. We showed how the original panorama video can be re-projected into a normal perspective view based on very simple input control. We emphasized that the virtual view can be generated from the same original panorama video, allowing multiple concurrent users to create individualized views without affecting the original recorded video stream. We explained how this can create many scalable solutions for video extraction and presentation, either streamed live or post-recording.

The first video delivery system we looked at would perform the view generation on each separate client machine, all connected to the same live streaming server. We discussed how a virtual view generated on the client-side can be incredibly scalable because of cheap one-to-many communication due to caching and simple service requests for the server. However, we also highlighted how this method struggles with the video downloading step, as the panorama video has incredibly high resolution and bit rate. We propose transitioning into adaptive HTTP streaming solutions, focusing on attempting to reduce the amount of data that must be sent to

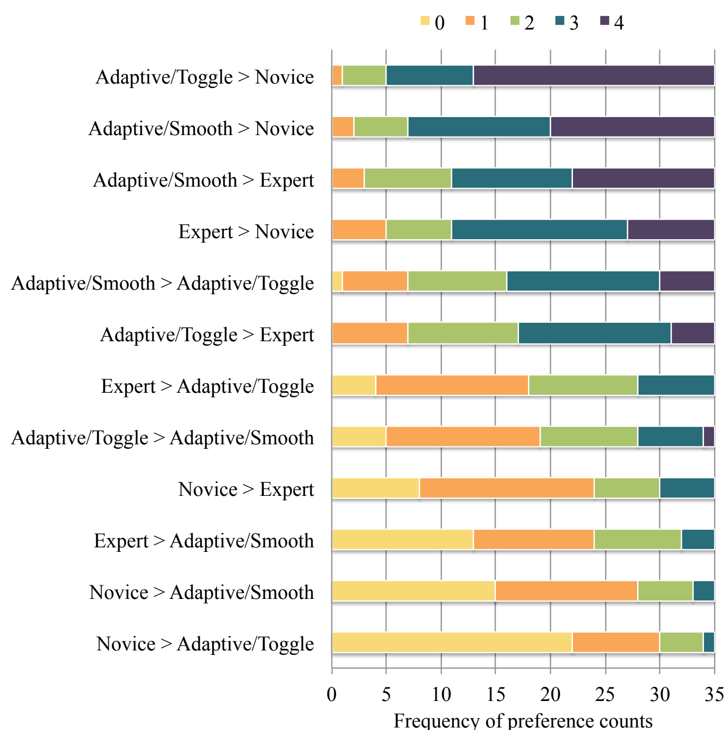


Figure 5.10: Frequency distribution portraying the number of times one stimulus was preferred over its contrast, accumulated across users. Here, we show the best camera tracking model against a novice and a professional camera man. The maximum count of 4 corresponds to the number of repetitions. Stimulus contrasts are sorted according to Friedman rank scores and plotted symmetrically.

the user, either through reduced bit rate or video tiling.

Next, we presented a more lightweight client solution, where the virtual view was generated by a dedicated server, removing the need for complex computations and large data transfer. In this scenario, the output video presented to the user is significantly smaller and can be streamed and viewed as any regular video. We show how an already existing system, designed for static cameras, can utilize the virtual cameras and benefit from this method. The initial online Bagadus video extraction system, intended for soccer analysis, can switch fluidly between the cameras and present the user with data from pre-recorded events with incredibly low latency. We also detailed a system for creating dynamic camera streams, allowing the user to control the virtual camera without performing the heavy computations locally. We showed how this system could immediately adapt the stream based on user input, in order to keep the experience interactive. We explained why low latency is key to such a system, and the challenges associated with controlling the virtual camera from a server. We also highlighted its limited scalability, but also that it allows more lightweight devices to run the application through a regular browser.

Finally, we discussed how we could produce an automatic virtual camera, based on object tracking information. We explained that it is fairly easy to automatically follow an object, once tracking information is pre-computed, but that can often be difficult to make the camera motion appear natural. We explained the models used to provide smoother transitions, in order to ultimately create a more pleasant viewing experience. The efficiency and promise of this system was confirmed in a user study, where most users preferred the automatic camera over a human operator.

Chapter 6

Conclusion

In this chapter, we summarize the work presented in this thesis. We then list our main contributions and present ideas for future work.

6.1 Summary

In this thesis, we have presented a distributed system for recording a high quality panorama in real-time. This system can construct a panorama that is well suited for post-processing, allowing for region-of-interest based virtual views to be presented with high quality. The motivation for re-designing the old Bagadus panorama pipeline was explained in chapter 1, where we emphasized the problems with a rectilinear panorama. Instead, we produced a cylindrical panorama with better preserved quality, while at the same time removing several visual flaws of the old Bagadus system.

The recording pipeline was detailed in chapter 2. Here, we explained the steps that were required to produce the panorama, and how these differed from the old Bagadus system. We wanted a scalable system and showed that we could offload the recording aspect of the pipeline to different machines. We detailed very simple synchronization steps that were required to facilitate the coordinated recording of multiple cameras, spread over different machines, and the challenges with handling such a high data rate. We also discussed how to best use the GPU architecture to speed up processing, giving some insight into the CUDA programming model. Then, we detailed the tasks of each module in the pipeline. We saw that several modules from the old Bagadus system were no longer required, and other modules were added to address shortcoming with the old pipeline. We presented a solution for transferring raw video frames between two machines, at low latency and high bandwidth. The challenging light conditions of the old Bagadus system were addressed by including an HDR-mode, to combine two frames captured at different exposures into a single high dynamic range frame. We saw that this required several changes to the overall pipeline, especially when controlling the video cameras, but that it allowed us to see areas that would otherwise be too bright or dark.

In chapter 3, we discussed the work put into the Bayer converter module. We explained that the pixels read from the camera only contain one color channel each, arrayed in a consistent pattern of red, green and blue. Several algorithms for interpolating the remaining color channels were then presented, and the majority of these were implemented on the GPU. The implemented algorithms were evaluated on their visual quality, artifact suppression and execution performance. We saw that all of these algorithms produced some visual artifacts, but that

most could still produce an acceptable result in our scenario. In terms of performance, we saw that the cost of using a more complex algorithm was quite small. We discussed several GPU specific optimizations, such as optimal memory accesses and kernel design.

The stitching module was presented in chapter 4. We saw that a cylindrical panorama can preserve more of the original image quality. We detailed an algorithm used to project and align each image, and how this could be computed once offline to construct a lookup table. Then, this lookup table was extended allow for stitching on the GPU, with a dynamic seam in overlapping areas. We showed how a dynamic seam can avoid creating visual artifacts, finding the path least visible. We discussed several methods to ensure this produced the best visual result. The implementation of the stitching module was detailed, highlighting performance optimizations and a visual result of each individual step. We saw that the stitching process consisted of several steps of varying execution time, but that the whole module was far below real-time requirements.

Finally, in chapter 5, we discussed several systems for presenting the panorama video, tied together by a virtual viewing approach. We saw how this virtual viewing approach can create multiple virtual cameras, individual to different users, created from the same panorama video. We discussed systems targeted at delivering the virtual view efficiently to the user, focusing on a combination of scalability, low latency and reducing computations. We explained that we can generate the virtual view either on the client side, or the server side. Each approach has several problems to tackle, such as high bandwidth requirements or limited concurrent users, but can be well suited for different situations.

6.2 Main Contributions

In this thesis, we have shown how a panorama with high resolution, high frame rate and high visual quality can be efficiently recorded and processed in real-time. The resulting prototype system has been installed at Alfheim stadium in Norway, and has already been used to record games by the local professional soccer team, as per request of the club. We have implemented a distributed recording system, spanning multiple machines, which is capable of even further upscaling if so desired. This required solving several challenges related to the capturing of the video, such as providing video frames with alternating exposure times at a very high frame rate. The effectiveness of exposure synchronization has also been shown, as we were able to create a visually pleasing panorama without applying color correction. We also provided an effective method for creating a cylindrical panorama, both on CPU and GPU. This method allowed for very easy configuration, making it simple to move the prototype system and still create a seamlessly stitched panorama. We have provided a system that is able to create high dynamic range video in real-time, with significantly higher resolution than existing work. We show that this is a difficult task to perform, and introduces several challenges apart from the algorithms themselves, but can greatly improve the visual quality during challenging lighting conditions.

We have provided detailed information on how to effectively program on the GPU architecture, and shown the immense speedup that can be achieved. This involves taking a detailed look at workloads that are both well suited for the architecture, such as bayer demosaicking, as well as algorithms that are challenging to parallelize, such as Dijkstra's shortest path algorithm. We saw that, through good use of GPU specific primitives and optimization techniques, we could

speed up the GPU implementations significantly.

We have given a detailed evaluation on multiple implemented Bayer demosaicking algorithms. Alternative evaluations are usually restricted to visual results, along with CPU implementation, but we have shown how many common algorithms can be efficiently implemented on a GPU architecture. We saw that this is a highly parallelizable workload, well suited for this architecture. Our results show that it is definitely possible to perform more complex demosaicking algorithms than the current norm, which is bilinear interpolation, for high resolution real-time video systems.

The panorama generated by our system is very well suited for a virtual view approach, which we have also presented. In this, we mean that this panorama has very high quality, compared to its relative file size. By allowing the virtual view generating to occur post-recording, i.e., generated from the same original video, we provide a very scalable solution. Many of these delivery systems have been realized, and full implementations have been able to provide multiple concurrent users with personalized virtual views. This proves the effectiveness of using our cylindrical panorama as an intermediary representation for virtual view generation.

Furthermore, we have published or submitted several research papers during the work on this thesis [10, 13, 20–25], as presented in section 1.6.

6.3 Future work

In chapter 2, we mentioned that further research could be performed to determine the optimal module placement in the pipeline. We saw that, for example, the HDR module can potentially benefit from being performed on the final panorama. This, however, means that several modifications would need to be made to the stitching module. We mentioned that several modules can be split into smaller modules. The stitching module is an example of this, as the initial projection step can easily be pipelined away from the rest of the module. This can potentially reduce some synchronization and idle time, and allow the slow dynamic seam creation to be launched faster.

The recording pipeline still has several areas of scalability that can be explored. It would be interesting to see if we could add more cameras to the system, while remaining real-time. The cameras are currently arranged sideways, with five cameras horizontally. Alternative camera arrays could be explored, for example by using cameras stacked vertically as well as horizontally, with a narrower field of view. Of course, simply using higher resolution cameras, such as 4K cameras, is also a major challenge.

We saw that the HDR module still has some problems with real-time requirements, in that the best performing algorithm is too slow. This means that there is still some interest in optimizing this step, or possibly distribute the workload. It could, for example, be interesting to see if we could perform the slow tone mapping operation on a separate machine.

The stitching module still performs a manual camera configuration. Although this takes little time and only needs to be performed once, due to the static configuration, it means that the system is less mobile. We could extend the offline stitching process to automatically generate the lookup table, based on SIFT [95] feature points extraction. A better solution would also be to create the lookup table at the beginning of a recording, automatically based on slow feature points matching. This would make the system truly portable, able to adapt to the cameras moving between recordings.

The stitching module also had some slow steps, that could likely be optimized further. We saw that the blending step was particularly slow, without a strong need to be. This could likely be further optimized, by using more threads and shared memory. Currently, each thread that is launched handles one pixel on the seam, but must read and average 400 luminosity values. These could be stored in shared memory while multiple threads shared the workload of one of the current threads.

In chapter 5, we presented several video delivery systems. We have, however, only begun to scratch the surface of use-cases for the virtual viewing approach. There are multiple other scenarios that could be interesting to look at, such as video conferencing or live concerts.

We mentioned that client generated virtual views can be challenging with regard to bandwidth limitations, due to having to transport the full 4K panorama. It can be worth looking into improving the streaming system for this approach, to reduce the size of the transmitted video. One way to do this is by transcoding the panorama with multiple bit rates, providing functionality of for instance MPEG-DASH [60]. This also means that the virtual view is more viable on devices with smaller screens, as transmitting a full 4K video is not optimal if the user is operating a, for instance, mobile phone. This can be taken further by performing video tiling, such as evaluated in [96]. By tiling the video, encoding parts of the image separately at different bit rates, the client application can opt to download lower quality tiles for the regions the user is not currently viewing. If the view is far zoomed out, covering a lot of tiles, the user is likely not going to notice a reduced quality. This approach could significantly limit the required bandwidth.

We can also see promise in a system for quick extraction of analytical data and video events, with possibilities of zooming and panning. Currently, in the Bagadus system, this program uses back-end processing to extract, encode and stream the video events. It could be interesting to try to reduce the server-side operations, making it more scalable, by performing virtual viewing on the client side. Another challenge is to run the virtual viewing on client-side in a modern browser, for example by porting the algorithm to WebGL [97].

The automatic camera also has many challenges to address. The Bagadus system currently does not have accurate tracking information for all players, and not for the ball. Ball tracking can be used as input to the automatic camera, and it would be interesting to integrate real-time ball tracking into the panorama pipeline.

Appendix A

Accessing the source code

The source code for the Bagadus project, the described pipeline and its modules can be found at `https://bitbucket.org/mpg_code/bagadussii`.
Access to this repository can be given upon request.

Bibliography

[1] Marcus Wieland, Ralf Steinmetz, and Peter Sander. Remote camera control in a distributed multimedia system. In Bernd Wolfinger, editor, *Innovationen bei Rechen- und Kommunikationssystemen*, Informatik aktuell, pages 174–181. Springer Berlin Heidelberg, 1994.

[2] Peter Carr and Richard Hartley. Portable multi-megapixel camera with real-time recording and playback. In *Digital Image Computing: Techniques and Applications, 2009. DICTA '09.*, pages 74–80, 2009.

[3] Aditya Mavlankar and Bernd Girod. Video streaming with interactive pan/tilt/zoom. In *High-Quality Visual Experience*, Signals and Communication Technology, pages 431–455. Springer Berlin Heidelberg, 2010.

[4] Jinjun Wang, Changsheng Xu, Engsiong Chng, Kongwah Wah, and Qi Tian. Automatic replay generation for soccer video broadcasting. In *Proceedings of the 12th Annual ACM International Conference on Multimedia*, MULTIMEDIA '04, pages 32–39, New York, NY, USA, 2004. ACM.

[5] Xinding Sun, J. Foote, D. Kimber, and B.S. Manjunath. Region of interest extraction and virtual camera control based on panoramic video capturing. *Multimedia, IEEE Transactions on*, 7(5):981–990, 2005.

[6] Peter Carr, Michael Mistry, and Iain Matthews. Hybrid robotic/virtual pan-tilt-zom cameras for autonomous event recording. In *Proc. of ACM MM*, pages 193–202, 2013.

[7] Y. Ariki, S. Kubota, and M. Kumano. Automatic production system of soccer sports video by digital camera work based on situation recognition. In *ISM'06. Eighth IEEE International Symposium on Multimedia*, pages 851–860, 2006.

[8] Masayuki Inoue, Hideaki Kimata, Katsuhiko Fukazawa, and Norihiko Matsuura. Interactive panoramic video streaming system over restricted bandwidth network. In *Proceedings of the International Conference on Multimedia*, MM '10, pages 1191–1194, New York, NY, USA, 2010. ACM.

[9] CNET. Netflix begins 4k streams. `http://www.cnet.com/news/netflix-begins-4k-streams/`. Accessed: 2014-4-23.

[10] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, Øystein Landsverk, Carsten Griwodz, Pål Halvorsen, Magnus Stenhaug, and Dag Johansen. Bagadus: An integrated real-time system for soccer analytics. *ACM Transactions*

on Multimedia Computing, Communications and Applications (TOMCCAP), 10(1s):14:1–14:21, January 2014.

[11] 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 *Proc. of ACM MMSys*, pages 48–59, March 2013.

[12] Marius Tennøe, Espen Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Dag Johansen, Carsten Griwodz, and Pål Halvorsen. Efficient implementation and processing of a real-time panorama video pipeline. In *Proc. of IEEE ISM*, December 2013.

[13] Vamsidhar Reddy Gaddam, Ragnar Langseth, Sigurd Ljœdal, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, and Pål Halvorsen. Interactive zoom and panning from live panoramic video. In *Proceedings of the International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*. ACM, March 2014.

[14] Asgeir Mortensen, Vamsidhar Reddy Gaddam, Håkon Kvale Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Automatic event extraction and video summaries from soccer games. In *Proceedings of the 5th ACM Multimedia Systems Conference*, MMSys '14, pages 176–179, New York, NY, USA, 2014. ACM.

[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 *Proc. of ICDSC*, October 2012.

[16] Basler. acA1300-30gc. `http://www.baslerweb.com/products/ace.html?model=167`. Accessed: 2014-4-28.

[17] Cambridge in Colour. Image projection examples. http://www.cambridgeincolour.com/tutorials/image-projections.html. Accessed: 2014-4-27.

[18] Harald Woeste. *Mastering Digital Panoramic Photography*. Rocky Nook, 1st edition, 2009.

[19] 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.

[20] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Kvale Stensland, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Be your own cameraman: Real-time support for zooming and panning into stored and live panoramic video. In *Proceedings of the 5th annual ACM conference on Multimedia Systems (MMSYS)*. ACM, March 2014.

[21] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vamshidar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, and Pål Halvorsen. Soccer Video and Player Position Dataset. `http://home.ifi.uio.no/paalh/publications/files/mmsys2014-dataset.pdf`, November 2013.

[22] Vamsidhar Reddy Gaddam, Ragnhild Eg, Ragnar Langseth, Pål Halvorsen, and Carsten Griwodz. The cameraman operating my virtual camera is artificial: Can the machine be as good as a human? In *Submitted to Proceedings of the 22nd ACM International Conference on Multimedia*, 2014.

[23] Ragnar Langseth, Vamsidhar Reddy Gaddam, Pål Halvorsen, Håkon Kvale Stensland, and Carsten Griwodz. An evaluation of debayering algorithms on gpu for real-time panoramic video recording. In *Submitted to Proceedings of the 22nd ACM International Conference on Multimedia*, 2014.

[24] Lorenz Kellerer, Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Stensland, Carsten Griwodz, Dag Johansen, and Pål Halvorsen. Real-time hdr panorama video. In *Submitted to Proceedings of the 22nd ACM International Conference on Multimedia*, 2014.

[25] Martin Alexander Wilhelmsen, Håkon Stensland, Vamsidhar Reddy Gaddam, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Pål Halvorsen, and Dag Johansen. Using a commodity hardware video encoder for interactive video streaming. In *Submitted to Proceedings of the 22nd ACM International Conference on Multimedia*, 2014.

[26] Qamira. Qamira: Realtime Virtual Camera Technology. `http://www.midworldpro.com/#!qamira-bvba/c6jy`. Accessed: 2014-4-14.

[27] Match Analysis. K2 Panoramic Video. `http://matchanalysis.com/k2.htm`. Accessed: 2014-4-14.

[28] Content Interface. Panoramic video walls. `http://www.sixteen-nine.net/2012/10/22/software-stitches-5k-videos-huge-panoramic-video-walls-real-time/`, 2012. Accessed: 2014-4-22.

[29] Michael Adam, Christoph Jung, Stefan Roth, and Guido Brunnett. Real-time stereo-image stitching using gpu-based belief propagation. In *VMV*, pages 215–224, 2009.

[30] Panoguide. What is parallax. `http://www.panoguide.com/howto/panoramas/parallax.jsp`. Accessed: 2014-4-28.

[31] Espen Helgedalsrud. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching. `http://home.ifi.uio.no/paalh/students/EspenOldeideHelgedagsrud.pdf`, May 2013. University of Oslo.

[32] iMove. Geoview Immersive Video Systems. `http://www.imoveinc.com/brochures/GeoView_Capture_Systems.pdf`. Accessed: 2014-4-15.

[33] J. Foote and D. Kimber. Flycam: practical panoramic video and automatic camera control. In *ICME 2000. 2000 IEEE International Conference on Multimedia and Expo*, volume 3, pages 1419–1422 vol.3, 2000.

[34] Dolphin. Dolphin high performance products. `http://www.dolphinics.com/products/index.html`. Accessed: 2014-4-23.

[35] Basler. acA2000-50gc. http://www.baslerweb.com/products/ace.html?model=173. Accessed: 2014-4-28.

[36] Azure. Azure-0814m5m lens. http://www.azurephotonicsus.com/products/azure-0814M5M.html. Accessed: 2014-4-28.

[37] libEvent. libevent api. http://libevent.org/. Accessed: 2014-23-4.

[38] Dolphin Interconnect Solutions. Sisci-Api spesification. http://ww.dolphinics.no/download/SISCI_DOC/index.html. Accessed: 2014-4-17.

[39] Nvidia. Cuda parallel computing platform. http://www.nvidia.com/object/cuda_home_new.html. Accessed: 2014-4-28.

[40] Nvidia. Kepler tuning guide. http://docs.nvidia.com/cuda/kepler-tuning-guide/, 2013. Accessed: 2014-4-28.

[41] Nvidia. Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2013. Accessed: 2014-4-28.

[42] Marius Tennøe. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background substraction. http://home.ifi.uio.no/paalh/students/MariusTennoe.pdf, May 2013. University of Oslo.

[43] Nvidia. Nvidia visual profiler. https://developer.nvidia.com/nvidia-visual-profiler. Accessed: 2014-4-28.

[44] Nvidia. Cuda c best practise guide. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2013. Accessed: 2014-2-3.

[45] Basler. Basler pylon sdk. http://www.baslerweb.com/Software_pylon-15606.html. Accessed: 2014-4-28.

[46] Mills et al. . *NTPv4 Specification*. Internet Engineering Task Force (IETF), 2010. RFC 5905.

[47] Martin Stensgård (Simula). Micro trigger box. https://bitbucket.org/mpg_code/micro-trigger-box.

[48] B.E. Bayer. Color imaging array, July 1976. US Patent 3,971,065.

[49] Vamsidhar Reddy Gaddam, Carsten Griwodz, and Pål Halvorsen. Automatic exposure for panoramic systems in uncontrolled lighting conditions: a football stadium case study. In *IS&T/SPIE Electronic Imaging*, pages 90120C–90120C. International Society for Optics and Photonics, 2014.

[50] Sigurd Ljødal. Ongoing thesis: Efficient distribution of resources using dolphin, August 2014. University of Oslo.

[51] PCI-SIG. Pci express faq. http://www.pcisig.com/news_room/faqs/pcie3.0_faq/. Accessed: 2014-4-28.

[52] Videolan. YUV. https://wiki.videolan.org/YUV/. Accessed: 2014-4-26.

[53] Paul E Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *ACM SIGGRAPH 2008 classes*, page 31. ACM, 2008.

[54] Gregory Ward Larson, Holly Rushmeier, and Christine Piatko. A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Trans. on Visualization and Computer Graphics*, 3(4):291–306, 1997.

[55] Erik Reinhard, Michael Stark, Peter Shirley, and James Ferwerda. Photographic tone reproduction for digital images. *ACM Trans. on Graphics*, 21(3):267–276, 2002.

[56] Ansel Adams and Robert Baker. *The print*. Little, Brown, 1983.

[57] Videolan. X264 video encoder. http://www.videolan.org/developers/x264.html. Accessed: 2014-4-27.

[58] Unknown. X264 settings wiki. http://mewiki.project357.com/wiki/X264_Settings. Accessed: 2014-4-27.

[59] Apple. Http streaming architecture. https://developer.apple.com/library/ios/documentation/networkinginternet/conceptual/streamingmediaguide/HTTPStreamingArchitecture/HTTPStreamingArchitecture.html. Accessed: 2014-4-28.

[60] I. Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *MultiMedia, IEEE*, 18(4):62–67, April 2011.

[61] Kodak. Kodak lossless true color suite. http://r0k.us/graphics/kodak/. Accessed: 2014-4-27.

[62] Craig Stark. Debayering demystified. *AstroPhoto Insight*, 2(3):5–9, September 2006.

[63] Fourcc. Rgb to yuv constants. http://www.fourcc.org/fccyvrgb.php. Accessed: 2014-4-28.

[64] D.R. Cok. Signal processing method and apparatus for producing interpolated chrominance values in a sampled color image signal, February 1987. US Patent 4,642,678.

[65] Henrique S. Malvar, Li wei He, and Ross Cutler. High-quality linear interpolation for demosaicing of bayer-patterned color images. In *Proceedings of the IEEE International Conference on Speech, Acoustics, and Signal Processing*, 2004.

[66] Wenmiao Lu and Yap-Peng Tan. Color filter array demosaicking: new method and performance measures. *IEEE Transactions on Image Processing,*, 12(10):1194–1210, Oct 2003.

[67] Flore Faille. Comparison of demosaicking methods for color information extraction. 2005.

[68] Keigo Hirakawa, Student Member, and Thomas W. Parks. Adaptive homogeneity-directed demosaicing algorithm. *IEEE Trans. Image Processing*, 14:360–369, 2005.

[69] J.E. Adams. Design of practical color filter array interpolation algorithms for digital cameras. In *Proceedings of 1998 International Conference on Image Processing*, volume 1, pages 488–492 vol.1, Oct 1998.

[70] Edward Chang, Shiufun Cheung, and Davis Y. Pan. Color filter array recovery using a threshold-based variable number of gradients. volume 3650, pages 36–43, 1999.

[71] Adobe. Cielab color model. `http://dba.med.sc.edu/price/irf/Adobe_tg/models/cielab.html`. Accessed: 2014-4-26.

[72] Daniele Menon, Stefano Andriani, and Giancarlo Calvagno. Demosaicing with directional filtering and a posteriori decision. *IEEE Transactions on Image Processing*, 16(1):132–141, 2007.

[73] D. Zhang and Xiaolin Wu. Color demosaicking via directional linear minimum mean square-error estimation. *IEEE Transactions on Image Processing*, 14(12):2167–2178, Dec 2005.

[74] Ron Kimmel. Demosaicing: image reconstruction from color ccd samples. *IEEE Transactions on Image Processing*, 1999.

[75] Jayanta Mukhopadhyay and Manfred K. Lang. Color demosaicing in yuv color space. `http://www.facweb.iitkgp.ernet.in/~jay/abstracts/conf/confpaper/cint_sbdct_iasted.pdf`. Accessed: 2014-4-28.

[76] OpenCv. Open source computer vision. `http://opencv.org/`. Accessed: 2014-2-12.

[77] Morgan McGuire. Efficient, high-quality bayer demosaic filtering on gpus. *Journal of Graphics, GPU, and Game Tools*, 13(4):1–16, 2008.

[78] Fastvideo. Gpu debayer. `http://www.fastcompression.com/products/debayer/debayer.htm`. Accessed: 2014-4-28.

[79] Sarah J. Fortune. Gpu-based raw digital photo manipulation. *Darwin College Research Reports*, 2010. DCRR-011.

[80] Xuefeng Hou, Yuanyuan Shang, Hui Liu, and Qian Song. Research on the real-time image edge detection algorithm based on fpga. In *Advanced Research on Computer Science and Information Engineering*, volume 153 of *Communications in Computer and Information Science*, pages 200–206. Springer Berlin Heidelberg, 2011.

[81] Guennadi Levkine. Prewitt, sobel and scharr gradient 5x5 convolution matrices. `http://www.hlevkin.com/articles/SobelScharrGradients5x5.pdf`, June 2012. Accessed: 2014-4-24.

[82] Xiaoni Liu, Yinan Lu, Ying Ding, and Jingtao Fan. The research on cylindrical panoramic projection and real-time rotation using gpu. In Zhihong Qian, Lei Cao, Weilian Su, Tingkai Wang, and Huamin Yang, editors, *Recent Advances in Computer Science and Information Engineering*, volume 126 of *Lecture Notes in Electrical Engineering*, pages 105–111. Springer Berlin Heidelberg, 2012.

[83] Helmut Dersch. Panorama tools. http://panotools.sourceforge.net/. Accessed: 2014-4-28.

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

[85] Andrew Bowden. Panoramic source images. http://www.flickr.com/photos/bods/sets/72157623479667942. Accessed: 2014-4-28.

[86] Herbert Goldstein. *Classical Mechanics*. Addison-Wesley Publishing Company, 2nd edition, 1980.

[87] Nvidia. Cuda efficient bicubic interpolation. http://docs.nvidia.com/cuda/cuda-samples/#bicubic-b-spline-interoplation, 2013. Accessed: 2014-4-27.

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

[89] Mikkel Næss. Efficient implementation and processing of a real-time panorama video pipeline with emphasis on color correction. http://home.ifi.uio.no/paalh/students/MikkelNaess.pdf, May 2013. University of Oslo.

[90] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[91] Martin Alexander Wilhelmsen, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Pål Halvorsen, and Carsten Griwodz. Performance and Application of the NVIDIA NVENC H.264 Encoder. http://on-demand.gputechconf.com/gtc/2014/poster/pdf/P4188_real-time_panorama_video_NVENC.pdf. Accessed: 2014-4-27.

[92] D. Johansen, M. Stenhaug, R.B.A. Hansen, A. Christensen, and P.-M. Hogmo. Muithu: Smaller footprint, potentially larger imprint. In *Seventh International Conference on Digital Information Management (ICDIM)*, pages 205–214, Aug 2012.

[93] Nvidia. Nvidia video codec sdk. https://developer.nvidia.com/nvidia-video-codec-sdk. Accessed: 2014-4-28.

[94] Matroska. Matroska Multimedia Container. http://matroska.org/technical/whatis/index.html. Accessed: 2014-4-28.

[95] D.G. Lowe. Object recognition from local scale-invariant features. In *The Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999.

[96] P. Rondao Alface, J.-F. Macq, and N. Verzijp. Evaluation of bandwidth performance for interactive spherical video. In *IEEE International Conference on Multimedia and Expo (ICME),*, pages 1–6, July 2011.

[97] Khronos. OpenGL ES 2.0 for the Web. https://www.khronos.org/webgl/. Accessed: 2014-4-22.