UiO : Department of Informatics
University of Oslo

Investigating the performance and applicability of object tracking in a real-time distributed video pipeline

Øystein Skonseng Landsverk

Master's Thesis Autumn 2014

Investigating the performance and applicability of object tracking in a real-time distributed video pipeline

Øystein Skonseng Landsverk

Abstract

Great technological advances in the last decades have enabled the creation of multimedia systems that capture high resolution images at high frame rates. As more and more digital video content is generated, we strive for ways to automatically analyze and understand video through computer vision, for categorization, searchable video and other purposes. Object recognition forms an important part in this evolution, where objects in video are found and identified.

In this thesis, we introduce a real-time distributed video pipeline that produces high resolution panorama video. Created as a diverse tool for capture and analysis of sports footage, the system allows for real-time generation and viewing of a cylindrical panorama covering an entire soccer field using five industrial cameras. In three case studies, we investigate the potential for ball and object tracking within such a panorama, by installing the system in both indoor and outdoor locations using different configurations. By selecting three current state-of-the-art object tracking algorithms and through implementation of three of our own tracker algorithms, we evaluate the performance and feasibility of object tracking in a real-time video pipeline.

We detail and evaluate the performance of these six trackers in all case studies, and learn that there is often a trade-off between tracking robustness and execution speed. Trackers that maintain adaptive appearance models to track unknown objects, struggle with handling the workload of the high resolution video produced by Bagadus. By developing tracking algorithms specific to each sequence, we are able to achieve usable performance by reducing computations, at the sacrifice of portability. We recognize that especially ball tracking in team sports is a very challenging case, and hope to provide knowledge useful for further research.

Contents

List of Figures

List of Tables

Acknowledgements

I wish to thank my supervisors Pål Halvorsen, Håkon Stensland, Vamsidhar Reddy Gaddam and Carsten Griwodz for excellent discussions, feedback and informative meetings during the work on this thesis. I also wish to thank my fellow students Martin Alexander Wilhelmsen, Ragnar Langseth, Asgeir Mortensen and Sigurd Ljødal who also have been working on the Bagadus system, for useful input on my work on object tracking, as well as useless and fun ways to procrastinate.

Last but not least, I wish to thank my beautiful wife, Mariann, for her continued support and cooking of meals during these busy times.

Øystein Skonseng Landsverk

Oslo, August 15, 2014

Chapter 1

Introduction

1.1 Background

Object recognition is central when analyzing the ever-increasing amounts of digital video produced today. Through the use of computer vision algorithms, video streams can be analyzed to help computers understand the video content. Object tracking is categorized as an important part of the computer vision field. At a high level of abstraction, computer vision is simply put the ability of computers to see. It aims to understand what a digital image contains in the same way humans do, through processing and analysis. Great technological advancements during the last decades have made high computing power available at a low price, and thus made it possible to analyze images from high quality cameras in real-time to extract very useful information. Computer vision and object tracking have a wide array of possible applications; machine vision, image classification, traffic analysis, automated medical diagnosis, unmanned aerial vehicles, automated surveillance and many more. In order to be able to create systems that provide services like automated surveillance, we need to be able to extract information from the cameras' images, i.e., what type of objects are present. The system might need to recognize humans and their actions, and perhaps set off an alarm if a person is seen in a restricted area, or doing something suspicious. More recent technical innovations such as Google's self driving cars [24], present new opportunities for safe driving. These make heavy use of computer vision techniques to not only localize itself in the environment, but to track and calculate the trajectories of cars and pedestrians, and respond properly to their movements and actions.

In sports, accurate object tracking can be extremely useful. The tracker can help provide detailed statistics about the movement of both the players and the ball, giving the team's coaches and players a unique opportunity to analyze their play. Statistics such as running patterns, player speeds, distance traveled and ball possession metrics are easily extracted given an accurate tracker. Tracking is useful not only for analyzing a team's performance, but can also aid referees in making well-informed decisions (e.g., goal line and offside), provide interactive video solutions to users and much more. However, ball tracking in team sports has proven to be a highly complex problem. The ball often is partially or fully occluded by the players, and might follow very unpredictable trajectories during tackles and dribbles. Furthermore, the small size and high speeds of the ball require either expensive high-end cameras or introduction of additional computational complexity to get a clear image. Many team sports are also located outdoors, subject to great illumination variations. These are a only a few of the challenges to be overcome for a ball tracker aimed at use in team sports. However, the gain of extracting such information from recorded sports videos is enormous, and has spawned a significant amount of research in the field of object tracking and detection.

The Bagadus system [48, 20, 55] is unique in its approach to integrate a sensor subsystem, an analytics subsystem and a video subsystem, providing a complete prototype for video extraction and statistical data analysis. Bagadus uses a sensor system [58] coupled with belt packs and radio towers for tracking the players, and enables complex queries against the data collected. However, there are no such sensors inside the ball. In this thesis, we investigate and evaluate object tracking approaches to track the ball in the panorama video output from Bagadus. Such accurate object tracking of the ball can further improve the system by providing information on its position either for online purposes during play, such as automatic guidance of a camera, or for post-game analysis (running patterns, ball possession metrics or more).

1.2 Problem Definition / Statement

In this thesis, we investigate the field of object tracking and how it might apply to panorama video generated from a system such as Bagadus. Object tracking, and ball tracking in particular, is a hard problem that we approach step wise through a series of case studies. We explore how tracking algorithms perform within the boundaries of the system, where a multitude of high quality cameras can be combined to provide a 4K resolution panorama. Lots of research has been produced in combining object tracking with broadcast soccer footage [69, 38, 43, 47], but these tracking algorithms do not operate within the context of an video pipeline producing such extreme image resolutions. Also, Bagadus aims to provide real-time services to its end users, and an object tracker would thus need to conform to the strict, and very hard-to-meet, real-time processing requirements.

We aim to provide usable object tracking as an end result, where the system can leverage accurate tracking information of the object in question, to further expand its usage and offer new services that was previously neither present nor possible. We investigate what the current state-of-the-art is able to offer within the field of single object tracking, and the possibilities and potential of applying these algorithms to a real-time high resolution video pipeline. The system is installed at three different locations, producing footage for three different case studies that we investigate. We make use of the knowledge gained on each use case to implement more specific tracker algorithms that are adapted to the conditions in the sequence at hand, and see how well these compare to the state-of-the-art. We learn what is necessary in order to cope with the challenges in the case studies by looking at the success and failure cases of each tracker in turn.

Ultimately, we wish to use the knowledge gained in the case studies to develop a well functioning object tracking algorithm that be implemented as a pipeline module for integration into the Bagadus pipeline prototype. However, we recognize the large amount of research already done in this field, and that a tracking algorithm with the needed overall performance has yet to be presented. Instead, we aim to provide a step on the way by isolating failure cases and analyzing what different approaches to tracking are able to do, and where and why they fail. This knowledge can then hopefully be leveraged to increase tracking performance by for example combining the features of multiple tracking algorithms.

1.3 Limitations

The work does not provide a fully operating module to integrate into the pipeline. Our main focus is on the trackers' ability to accurately locate the object, and we only analyze performance of CPU implementations. However, the tracker algorithms are evaluated with pipeline integration as a final goal, specifically aimed at real-time operation, and we leave optimizations for future work. Also, trackers are normally evaluated on a large amount of sequences (typically at least 10), but due to the limited amount of footage produced by our prototype system, we present only three case studies. These studies are selected to provide a natural progression towards ball tracking in Bagadus, and aim to cover conditions likely to be seen. Also, as we will see, many systems exploit the possibilities given by multiple cameras, specifically to triangulate the object using information from each camera. However, all of our case studies uses a single camera array providing only one view point, thus eliminating the possibility of leveraging information from multiple viewpoints.

Also, we only provide background information on the Bagadus system, and do not dive too deep into implementation specific details of modules generating our input images, as this lies outside the scope of this thesis.

1.4 Research Method

The research method used in this thesis is based on the *design* paradigm described in *Computing As a Discipline* [11] by the ACM Task Force. The two other paradigms, *theory* and *abstraction*, are rooted in mathematics and the scientific method, respectively, and do not easily map to our work. We instead approach our work from an engineer's perspective using the *design* method. Our research is based on extending the older versions of the Bagadus system through redesign. A new panorama pipeline system is designed, implemented and tested. The main prototype is installed at Alfheim Stadium (Tromsø, Norway) for daily use by the soccer team Tromsø IL. Additionally, the prototype system is installed and tested in two more locations to provide further coverage of real-world usage scenarios. As opposed to constructed scenarios, these conditions serve as an excellent basis for testing and evaluating the performance and applicability of object tracking in real-world scenarios.

1.5 Main Contributions

In this thesis, we have investigated various approaches object tracking in a real-time distributed panorama pipeline and determined its feasibility. We have shown how well the current state-of-the-art solutions within single object tracking are able to track our object of interest in three different case studies. We have studied how these compare to our own tracking implementations that exploit knowledge about each specific case study at hand, and given performance tables and plots to represent the results. In our work on this thesis, we have presented 3 of our own tracking implementations, namely the Color Tracker, the Optical Flow Tracker and the Background Subtraction Tracker.

We have shown that object tracking algorithms able to provide usable results in several unrelated scenarios are computationally expensive, and thus more than often exceed the real-time threshold. The tracker algorithms were either far above the real-time threshold with respect to execution time, or did not provide the needed level of accuracy when tracking the object in the video. When using extreme resolutions such as 4K, we clearly saw the trade-off between robustness and execution speed. This effect can be diminished by exploiting information on each use case, where the algorithm can make more assumptions and reduce the number of computations. For example, where applicable, the color tracker often achieved superior results compared to the more complicated adaptive tracking approaches of RTCT, TLD and Struck. The Color Tracker could use knowledge of the object's color, given offline, while the adaptive trackers must provide enough computational complexity to handle more contingencies, such as all possible colors. Similarly, in chapter 5, we knew that our object of interest was the only object moving for the majority of the frames. Therefore, we could utilize background subtraction to locate our object. However, such reasoning and specific implementations limits the portability of the tracking algorithms, as algorithms developed for one use case was more difficult to apply on a use case with other conditions. Our three case studies also provided an increasing level of difficulty for the trackers. We have seen that when tracking objects with few features and little internal color variations, the trackers experienced difficulty establishing a proper appearance model for the object. This led to high inaccuracy and often lost frames. For example, in our first case study, we were able to leverage information about the object to create a fast and accurate tracking using detection by color. However, due to the lack of portability in such an approach, the algorithm was unable to provide satisfactory results in the second case study as we struggled to identify sufficiently unique object colors.

Particularly, we have seen that ball tracking is a highly complex use case where the small size of the ball, large illumination variations, heavy motion blur and occlusions were the most prominent challenges. It is essential to elegantly cope with, or rather overcome, these challenges in order to provide any kind of useful tracking in an analytical system such as Bagadus. We have seen how the current state-of-the-art, along with our own implementations, attempts to handle the challenges and where they failed and succeeded. We recognize the complexity of ball tracking, especially in a high resolution panorama capturing team sports, and provide knowledge that can be useful for further research. However, we have shown that useful performance can still be achieved, by lowering portability. During our work on this thesis, we have submitted a paper where we present an object tracking module prototype integrated into the pipeline, where color was used to track a moving person and guiding a virtual camera in a live setting [19]. Also, we have published a paper during early development of the Bagadus system, where we contributed discussions and development of the web interface where users could browse the recorded events [55].

1.6 Outline

We continue this thesis in Chapter 2 by providing a review of the previous recording pipeline, describing the design and architecture of the system, along with a brief description of each module. We then go on to describe how the limitations of the previous system are addressed to form a more scalable system, and give an overview of the new hardware and software architecture. Each module in the pipeline is then detailed in order of execution, from capturing the frames to writing them back to disk, or presenting real-time video to the system's users. In Chapter 3, we delve into the field of object tracking. Here, we look into central components in tracking algorithms, such as object representation and feature selection. We list our goals and requirements for the tracker, and dive into related work to select three tracker algorithms that represent the current state-of-the-art. In chapter 4, we present our first case study, where 2 cameras are used in an indoor lab setup. Each tracker's performance is evaluated in turn, along with a detailed implementation description of the three self-made tracking algorithms. At the end of the chapter, we give a review of our results along with tables and plots showing statistics about all trackers' performance on the sequence. Chapter 5 is our second case study, captured outdoors using 3 cameras. Similar to Chapter 4, we give performance reviews and evaluations of each tracker. Our final case study is presented in Chapter 6, where we use the setup of the full system prototype, configured with all 5 cameras. Here, too, we evaluate the trackers and show their strengths and weaknesses, along with performance summaries in the form of discussions, tables and plots. In Chapter 7, we summarize our work and list our main contributions, before we conclude the thesis by reviewing possible future work.

Chapter 2

The Recording Pipeline

2.1 First panorama pipeline

The initial prototype of Bagadus was presented by S. Sægrov et al. in [48]. The system integrated a camera array video capture system with the ZXY Sport Tracking System for player statistics (speed, position, heart rate etc.) [58], and a system for human expert annotations. The system made it possible to play back events created from the captured video, with player tagging and tracking, digital zooming and many other features. [20] introduced changes that enabled most of the system to operate in real-time, except for the the non-optimized, sequential stitching operation. A dynamic and real-time stitcher was presented in [57, 54], along with a complete and fully operational system running in real-time. The system was installed and used at Alfheim Stadium in Tromsø, Norway, for the team Tromsø IL which plays in the Norwegian premier league.

In order to fully understand the reasoning behind the choices made in designing the new system, we provide a quick review of the structure of the most recent version of the Bagadus system (see [55]). The system was divided into three subsystems: analytics subsystem [26], sensor subsystem [58], and video subsystem [55]. In this chapter we will focus on the video subsystem, as this is where the object tracker operates, and refer to [55] for more details and information on the other two subsystems.

The recording pipeline part of the video subsystem is built around an image pipeline with a set of modules running in sequence: the output of one module was the input of the next (seen in figure 2.1). will have a look at each of the modules in turn along with a description, and explain how they work together to form a fully functioning recording pipeline operating in real-time.

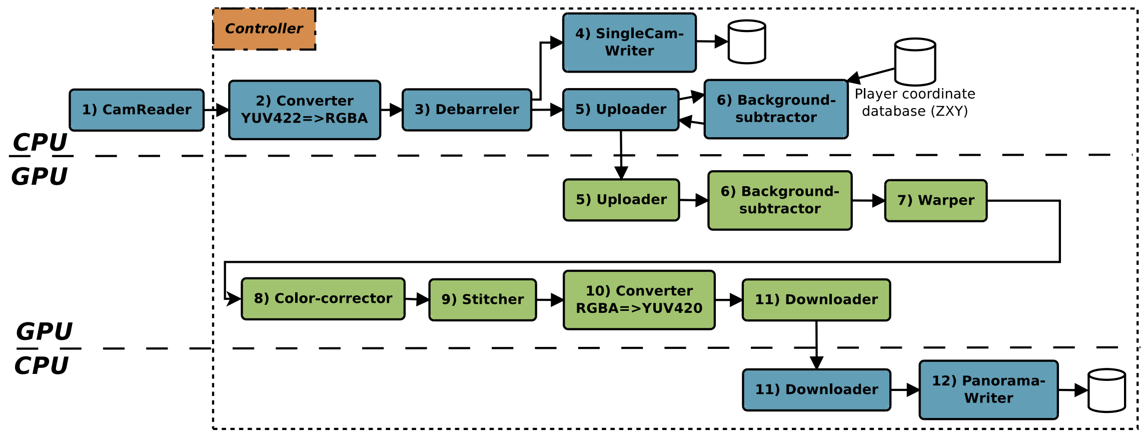


Figure 2.1: Architectural overview of the pipeline

Cameras - Capturing the frames

At the core of the video subsystem lie the cameras. The cameras used in the old system are four Basler acA1300-30gc industrial Ethernet-based cameras [1], with 1/3-inch imaging sensors supporting 30 frames per second (FPS) and a maximum resolution of 1294×964 pixels (this is in practice limited to 1280×960 by the Basler Pylon drivers). They are all mounted with Kowa 3.5mm wide angle lenses, with a field of view of about 68 degrees each. This is enough to cover the entire field with four cameras, with sufficient overlap for camera calibration and to identity common features for the stitching algorithm. All of the four cameras are mounted on one of the long sides of the field. An image of the camera coverage is shown in figure 2.2 (this is configured similarly in the new pipeline discussed in chapter 2.2).

In able to fully synchronize frame capture on the four cameras and to keep the frame rate correct, the cameras are interconnected over Ethernet through a triggerbox created by Simula Research Laboratory. The triggerbox signals the cameras to open their shutters and capture an exposure. The maximum supported frame rate by the cameras and their drivers is 30 FPS, and the triggerbox is thus programmed to operate at a frequency of 30 Hz. To synchronize the frames with the positioning data provided by the ZXY sensor system [58], the capture machines are connected to an NTP server. This synchronization provides millisecond-precise timestamps to each of the files written to disk, making it easy to play back fully synchronized video streams from the different cameras at the same time.

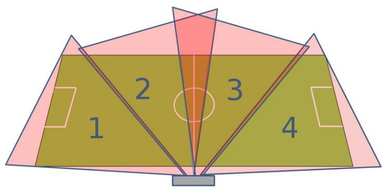


Figure 2.2: Figure showing the view and overlap of the cameras

CamReader - Fetching frames into memory

Since the cameras deliver 30 frames each second, each module needs to be done processing a frame within $\frac{\text{1 second}}{\text{30 FPS}} = 33ms$, in order to adhere to the strict real-time requirement of the system. As long as this requirement is met by all the modules, and excluding the fact that the pipeline initially needs to be "filled", the whole system will operate in real-time.

The CamReader module acts as an interface between the Basler cameras and the image pipeline, and it is the first module to receive the frames. The Northlight library [20] contains a wrapper around the camera drivers, and is used to access the cameras and retrieve the frames. Four threads (one for each camera) are used to process the frames and read them into memory. The cameras output the frames in the YUV color space. Y describes the luma component (image brightness), while U and V are the two chrominance components (determining color). The pixel format used is YUV422. Since humans are more sensitive to changes in brightness than color, U and V can both be subsampled without image degradation noticable to the human eye. More information on YUV and chrominance subsampling can be found at [67].

Converter - Converting from YUV to RGB

The second module in the pipeline is the Converter. The Converter is the simplest in the pipeline and is responsible for converting from the YUV422 pixel format back into RGBA. It runs as a single threaded CPU process and the conversion is done through Northlight, by converting in the following manner: YUV422 -> YUV420 -> RGBA. The conversion is done due to the simplicity of working with the RGBA format, and to allow for coalesced memory accesses and faster execution for later modules which operate on the Graphical Processing Unit (GPU).

Debarreler - Removing barrel distortion

All of the cameras are mounted with wide-angle lenses and these introduce quite a lot of barrel distortion. Barrel distortion causes image magnification to decrease with the distance from the optical axis (image center). This results in poor visual quality, and it is the job of the Debarreler to fix the distortion by straightening the lines in the image. This is necessary in order to align the four images properly when stitching them together into a panorama image. The Debarreler runs fully on the CPU and, like the CamReader, have four threads (one per camera). Each thread uses a set of a barrel distortion coefficients[1] belonging to the camera which delivered the frame, and executes a debarreling algorithm part of Intel's Open Source Computer Vision Library [12], using a nearest neighbor interpolation .

SingleCam Writer

After all the frames have been retrieved by the CamReader, and cleared of barrel distortion by the Debarreler, the frames are passed to the SingleCam Writer module. Since the system aims to deliver individual streams from each of the four cameras, these frames are now almost ready to be written to disk. As in the Debarreler, one thread is assigned to each camera. Each thread first converts the pixel format from RGBA back to YUV420, in order to prepare for video encoding (Northlight needs YUV), and then stores the frame in a buffer. If the buffer has reached 90 frames (3 seconds of video), all frames received so far are written to disk as a H264-encoded video file. The file is given a sequentially numbered name with millisecond precision. If there is less than 90 frames in the buffer after storing the frame, the module waits for the next set of frames to arrive.

Although independent video streams from each of the different cameras are very useful, another goal of the system is delivering a full panorama image as well, by stitching together the four frames into one big image overlooking the whole field. Even though the system allows for switching between the four different streams, a football field is big and a panorama can be very useful, e.g. viewing the movement of defenders during an attack or vice versa. Creating a full panorama image demands further processing and the first step is passing on the four frames to the Uploader.

Uploader - Transferring work to the GPU

Most of the following modules makes use of the high computing power of the GPU, and the main task of the Uploader is to transfer workload from the CPU to the GPU. It asynchronously uploads the four single frames from the computer's local memory (DRAM), to the memory of the GPU. The Uploader also contains a CPU process which calculates some look-up maps (these will be further explained in section2.1) needed by the Background Subtractor (BGS) and uploads these to the GPU as well.

Background Subtractor - Preparing for the stitcher

The role of the BGS is to analyze each image and separate the background (stadium, ground, sky etc.) from the foreground (the players). The purpose of the look-up maps mentioned in section

[1]these are calculated in advance as part of the pipeline configuration

2.1, is to help the subtraction algorithm by providing information on which pixels the players are part of, and which pixels are part of the background. These pixel look-up maps are calculated by using positioning information given by the ZXY sensor system, and speed up the execution of the BGS. If the look-up maps are not provided, the BGS has a fallback mode which processes the whole field, and not just the pixels which contain players. Since the system now knows where in the image all of the players are (either through the lockup maps provided by the Uploader or by fully calculating it), the Stitcher can avoid placing a seam through a player[2].

Warper - Aligning the frames

The Warper module is responsible for doing the final preparation of the images for the Stitcher. The Stitcher assumes that the images provided are a perfect fit, so the images are warped so they all have the same horizontal and vertical position, angle and size. Since the Warper actually changes the images, the foreground masks (player -> pixel look-up maps) given by the BGS will also need to be warped in order to reflect the changes made. This way the Stitcher can still make use of the foreground masks to avoid players during the stitch, despite the warp.

Color Corrector - Correcting color differences between cameras

There is one module left before the frames are ready to be stitched together: the Color Corrector. The four cameras are mounted a few centimeters apart and face different angles toward the field. They individually adapt to the lighting conditions and this inevitably creates slightly differing exposures. An image in one camera may be a bit brighter/darker than the same image in another camera, or one camera may be in the shadow while another is in the sun. These differences will be very clear when they are stitched together to form one image, and it is the job of the color corrector to adjust the colors so they merge seamlessly together to create a color-consistent panorama.

Stitcher - Creating the panorama image

The Stitcher runs fully on the GPU, and is the module responsible for actually merging the four single frames together into a large panorama. This is done by calculating where the seams between the frames should be. In the first draft of the algorithm, a static stitch was used, which cuts straight through the image. This can cut players in two and create ghosting effects, so we need a dynamic stitch that avoids areas corresponding to the position of the players. First step is to create an adjustable rectangle across the static stitch seam area. The seam algorithm is based on Dijkstra's graph algorithm [16], which is a method to find the shortest path to all other nodes from a center node. This is made use of by making the nodes in the graph represent pixels, and the edges contain weights based on the color differences between them. The weight function calculates the absolute color differences between corresponding pixels in the frames we are stitching together. The final score of the weight function is a sum of the absolute color differences between corresponding pixels in each of the frames, where high similarity gives a low score, and the value of the foreground masks for those pixels. If a player is present in one of the pixels, the function returns a high weight in order to avoid the seam going through a player. All together this approach leads the stitcher away from pixels which differ a lot, or contain a player. We want the seam to be invisible, and thus aim towards highly similar areas, like empty spots on the field with consistent colors.

After calculating the weight between all the nodes in the graph, we are left with a Directed Acyclic Graph, pointing upwards from the bottom of the image. Dijkstra's algorithm then is

[2]This causes ghosting issues: the same player is seen in two adjacent images in the panorama.

run to calculate the minimal path cost from the bottom of the seam area to the top. A buffer is allocated to hold the full panorama image, and all the pixels from the warped frames are then copied into the correct position.

Converter (RGB->YUV)

The finished panorama image is now stored in a buffer, and ready to be written to disk. To be able to use the H264 encoding mechanism in Northlight, the panorama's pixel format needs to be converted back into YUV420 (like in section 2.1). This module just exactly the opposite of the Converter module in section 2.1, but runs on the GPU instead. Image processing and converting between pixel formats is very easy to parallellize and thus fits perfectly to execute on a GPU. The GPU executes the following conversion: RGBA -> YUV444 -> YUV420.

Downloader - Transferring the result back to CPU

The downloader module does the opposite of the Uploader module described in section 2.1. The pipeline has now done most of its work and we have a finished panorama ready to be written to disk. The panorama currently exists in the global memory of the GPU, and it needs to be transferred back to CPU memory. The Downloader runs as a single thread on the CPU, and all it does is run a simple synchronous transfer function provided by CUDA, copying the entire panorama buffer back to CPU memory.

PanoramaWriter - Writing the panorama to disk

The pipeline has reached the very last module, and now does exactly the same as the SingleCam Writer module described in section 2.1. The finished YUV420 panorama image is written to disk by a single threaded CPU process, as a 3-second long H264-encoded video file, numbered like the other files and given a millisecond timestamp. All of the four single camera streams along with the panorama are now possible to play back on any device that supports H264 decoding.

The Controller - Handling communication and execution

In the center of this modularized approach is the Controller module. It relays all communication between the modules and the pipeline, and provides a central authority for the system. It starts by initiating the cameras, and feeding the frames into the pipeline. After all modules are finished processing their current frame, the Controller handles all data transfers to initiate the next step of the pipeline. After all data transfers are complete and each module has received the next frame, the Controller signals all modules to process the new frame. Any frame drops during execution are journaled for system profiling and statistical purposes.

Limitations

The system contains a number of weaknesses that we aimed to address when designing a new pipeline. The system was designed to be modular, but the modules were very tightly coupled and did not keep a clear and distinct role, making module replacements difficult. The Controller was in charge of all buffers, data flow and signaling, complicating the process further. More noticeable however, is the visual quality of the output video. This is mainly caused by the hardware used, meaning the fairly low camera resolution of 1294x964 pixels. On the single camera streams overlooking the field they deliver an acceptable quality, but zooming in on areas of the panorama does not retain a consistent high visual quality. Moreover, due the barrel distortion introduced by the lenses (detailed in section 2.1), the image also suffered further degradation through the Debarreling step. The interpolation introduced unwanted effects and the image quality suffers, especially around the edges of the frame. Large changes in illumination and lighting conditions also present a great challenge to the system. The Color Corrector module was introduced to reduce the inter-camera color differences, but the system is unable to provide a proper stitch during challenging conditions. We refer to the master's thesis of Marius Tennøe [56] and [57] for example figures and further details of the pipeline.

The system has been through many iterations of improvements ([48], [20], [55]), by redesigning the system, offloading work to GPUs etc. However, since the system is running on one capture machine and resource utilization is close to maximum already, scaling the system to use more cameras or higher resolution is severely limited. More pipeline modules could be moved to use the GPU, but these kinds of optimizations are not enough to provide the needed scalability. In the next section we will look at how these limitations were addressed, and how the system was changed in order to handle higher resolution cameras and additional functionality.

2.2 Redesigning the Pipeline

We will start by introducing an overview of the new system's architecture and module design. Next, we take a look at how the new high resolution cameras are configured, and also show the new data distribution scheme, enabling much higher scalability. We then go on to provide a detailed description of all modules part of the pipeline, presented in order of execution for easy reference.

2.2.1 Architecture and module design

Like in the older versions of Bagadus, the system is designed as an image pipeline where a set of modules is running in sequence. However, there is no centralized control unit similar to the Controller that relays signals such as when to process the next frame. Instead, in order to keep the modules decoupled and easily replaceable, the new module design is based on using two simple shared interfaces. Their design utilize a pull-based approach, modeled through both Producer and Consumer interfaces which are implemented by all modules. Each module can act as a Consumer, by requesting the next input frame from its Producer module as often as possible. Also, the module will serve as a Producer by providing its output to any relevant Consumer module. The interfaces represent frames and their meta information (e.g. timestamp) as a single continuous block of memory, and the buffer address is passed on to whichever module needs to access the memory location. This modular design makes pipeline reconfiguration very trivial, by enabling us to easily swap, replace or remove modules at will. Furthermore, testing is much easier as we can seamlessly run a single module at a time, or arbitrary combinations of modules. Each module only needs to make sure all methods defined in the interfaces are implemented, and conform to the binary data representation in the shared buffer.

The architecture of the new system is shown in figure 2.3. GPU modules are marked green, while CPU modules are colored blue. Orange modules are part of the event system, and the red and black lines represent an Ethernet or high-speed PCIe connection, respectively. Here, we see that the 5 cameras used are shared between three recording machines, running in a $2 + 1 + 2$ configuration. Each of the recording machines contains a Camreader module responsible for retrieving the raw video frames from the Basler cameras. Also, a Dolphin module is present, responsible for ultra-low latency and high bandwidth transfer of the frames to the processing machine, where they are gathered. The processing machine then feeds the frames through the image pipeline.

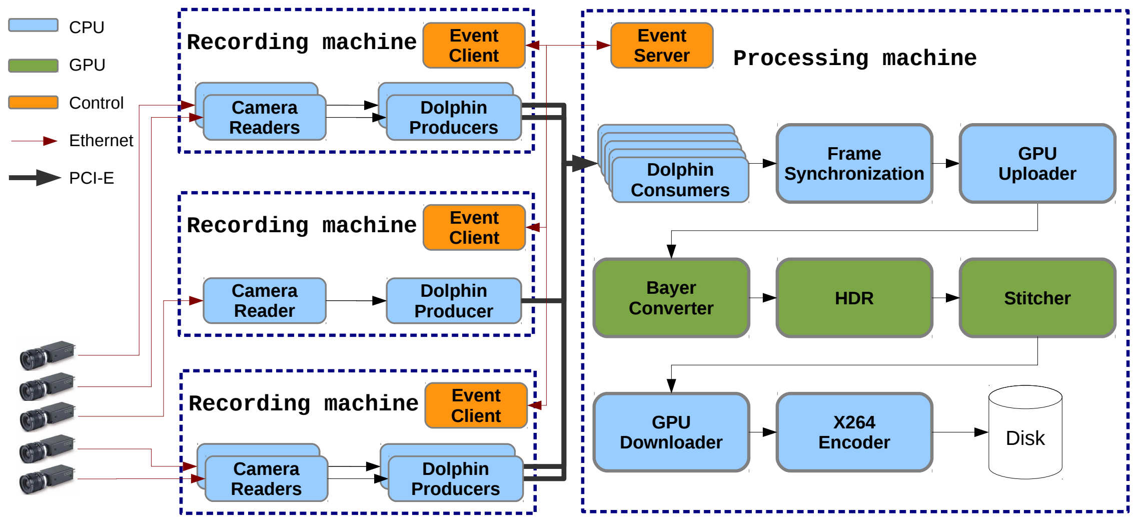


Figure 2.3: Overview of system architecture showing the three recording machines, the processing machine and all modules.

There is also an event system on each machine. The processing machine is the event server and signals all capture machines using Libevent [39] over TCP connections. Possible signals include start/stop recording, or configuration changes. For example, when doing exposure synchronization across cameras, the recording machine with the master camera first generates a synchronization event that is sent to the server. The server then broadcasts this information to the other recording machines running the slave cameras.

2.2.2 Camera configuration

On the new version of the system, the old cameras are replaced by five Basler acA2000-50gc industry vision cameras [2]. The frame rate is almost doubled, from 30 to 50 frames per second (FPS), and the maximum resolution delivered is increased to 2046×1086 (up from 1294×964 pixels). The new 8mm lenses [37] virtually eliminate the barrel distortion introduced by the old lenses, and thus enable us to skip the lossy debarreling module from the old pipeline. The five cameras are rotated 90°, giving a vertical Field-of-View (FOV) of 66°, in order to maximize the resolution of the panorama (the overlap is still enough to cover the entire field, but we achieve higher vertical resolution). Furthermore, the cameras are mounted in a circular pattern, i.e. pitched, yawed and rolled, to look directly through a point 5cm in front of the lenses, in an attempt to reduce the parallax effect (see figure 2.4). These measures ensure only small adjustments are needed in order to create a properly stitched panorama. In order to keep the inter-camera colors consistent even throughout heavy illumination variations, the center camera is configured to run auto-exposure once every 4 seconds, and the others are continually notified of the exposure parameters by the camera reader module (see section 2.3), and adjusted accordingly.

2.2.3 Distribution of workload

In order to cope with the extra workload produced by the new cameras, the new system is distributed across multiple machines, instead of relying on only one capture machine like the previous pipeline. The major increase in bandwidth and resource requirements is too much for a single Gigabit Ethernet network to handle, and to provide the speed and bandwidth necessary, each computer was equipped with a Dolphin Interconnect Solutions PCIe adapter [52]. The

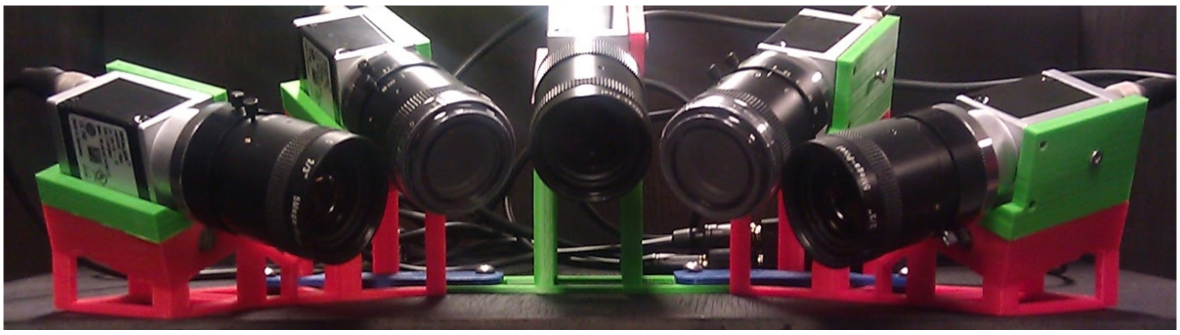


Figure 2.4: Camera setup

cards are interconnected through a PCIe switch [53] to form a high speed network and provide ultra-low latency Remote Direct Memory Access (RDMA). Each frame is transferred with a single DMA transfer at a rate of 19.41 Gbps and a total latency of 1.26 ms per frame (see [17]).

2.3 Modules

Here, we present the modules in the order they are run, and provide details on each module underway.

Camera Reader

As in all the other previous versions of the pipeline, the camera module is the first module to run. It is responsible for retrieving the raw frames directly from the cameras [2], and this is done through the Basler Pylon SDK [50], combined with a wrapper for their API written by Alexander Eichhorn at Simula Research Laboratory (part of their Verdione project). Similar to the old pipeline, a triggerbox [34] is used to signal the cameras when to capture an exposure. As before, the box triggers a signal *frame rate* times per second. The exposure configuration is also synchronized with the other camera readers, to create a consistent exposure across all frames[3]. This ensures that the inter-camera colors are as close to identical as possible, making the seams of the final panorama smooth. After retrieving the frames from the cameras, they are marked with a Unix timestamp. The time synchronization is implemented by connecting all machines to a common NTP server [41], like in the previous pipeline [57]. This timestamp is used by e.g. the Frame Synchronizer module in selecting the correct frames to be part of the current frame set (more on this in the next sections). Accurately timestamped video frames are extremely important for succeeding modules, particularly to avoid feeding the Stitcher module frames captured at different times.

Dolphin Producers/Consumers

After the raw frames are retrieved from the cameras by the Camreader module, The Dolphin modules are responsible for transferring them to the processing machine. The producer module starts by filling a local buffer with a frame. As soon as the consumer module (located at the processing machine) polls the producer (current recording machine) for data, the frame is transferred across the Dolphin PCIe link, through an RDMA request. When the transfer is complete, the consumer module is notified and the frame is passed on to the Frame Synchronizer module (this module will move the data out of the buffer, making it available for reuse).

[3]The center camera (master) sends its configuration to the processing machine, which in turn broadcasts to the other (slave) cameras.

Frame Synchronizer

The process described in the above section will happen for every capture machine in the system. In this way, the Frame Synchronizer receives frames from all (currently 3) machines. The goal of the module is then to synchronize these single frames and gather them into framesets. The synchronization is done by examining the timestamp of each frame, which is defined in the frame header. If the differences between frame timestamps are less than $\frac{1\,\mathrm{s}}{2\text{ x FPS}}$, they are evaluated as part of the same frame set, and are afterwards inserted into a short buffered queue[4]. Whenever the module is polled by a Consumer, either the first set available in the queue will be transferred, or an interrupt timer is set. The module sleeps and waits for either the timer to expire or a new set of frames is inserted to the queue. If a frame set is not complete within the timeframe, it is dropped and the previous full set is duplicated in its place. This way, we enforce the real-time requirement. At 50 FPS, however, frame drops occur on average less than 0.2% (although more often when HDR is enabled), and duplicates proved an adequate solution[5]. At the end, the final framesets are copied to a continuous memory location accessible by the next module, along with a single common frame header.

GPU Uploader

executing on the GPU, the Uploader module is responsible for transferring the frames provided by the Synchronizer, from CPU memory to device memory (onboard memory of the GPU). Using NVIDIA's Compute Unified Architecture [13] (CUDA), the frames are transferred using an asynchronous copy operation. To stop the CPU from swapping out the memory pages containing the frames to disk during the transfer, the CPU memory is pinned (locking the frames to memory).

Bayer Converter

The pixel format of the images captured by the Basler cameras is *Bayer GR-8*. Unlike RGB, a raw image only contains a single color in each pixel (using one byte per pixel). The Bayer Converter module expands this limited color information into three full channels of RGB colors, through a process called *debayering*. Next, this temporary RGB pixel format is converted to *YUV444 interlaced*[6], which is the format now used throughout the rest of the pipeline. YUV444 uses one byte in each of the three channels, but an extra byte is added for optimal data alignment.

Many algorithms have been presented for the debayering step, and we refer to Chapter 3 in the master thesis of Ragnar Langseth [35] for implementations and a survey of current debayering algorithms in use today, as well as runtime analyses of their performance in our pipeline.

HDR - High Dynamic Range

In order to cope with challenging light conditions, one can make use of a concept called *High Dynamic Range Imaging* (HDRI). Cameras and other Low Dynamic Range (LDR) devices such as computer screens, are unable to capture or display the full dynamic range (ratio between dark and bright regions) present in the real world. These devices therefore only consider a subset. By capturing a scene using two different exposure settings, HDRI algorithms can construct a single High Dynamic Range (HDR) image from the two LDR images, and are thus able to represent the full range. These algorithms work by first doing a process called *radiance mapping*, which do

[4]This process is slightly different when HDR is enabled, as the HDR module effectively halves the framerate.

[5]These drops often occur when the cameras change their exposure setting, but is rare enough to be unnoticeable by the user

[6]Interlaced means all three components are stored continuously in memory, ensuring quick and easy access for the Stitcher module

the work of combining several LDR images into one HDR image. Then, a *tone mapper* is used to map, or compress, the range of the HDR image into a range representable by LDR devices, for e.g. displaying images on a computer screen or printing them to paper[7].

Naturally, a high amount of images captured with different exposures is desirable (more samples can possibly result in a wider range), but we can only "afford" two in our pipeline. With a framerate of 50 FPS, we extract our samples by alternating between capturing low and high exposure images. These two samples are then combined to form a single image, effectively reducing the framerate in half. In our pipeline this helps smooth the image contrasts, i.e. reduce the strong shadows occuring during sunny days, or dampen the effect of very bright light. Later, in chapter 3, we will introduce object tracking into our pipeline. The computer vision algorithms used in object trackers heavily utilize colors and contrasts in images, and an HDR module can provide a better basis by reducing unwanted visual artifacts such as reflections on the object's surface, shadows and extreme exposures.

In order to get the necessary amount of samples, the module's input is two full framesets (in YUV444), i.e. 2 x 5 images when using 5 cameras. The HDR algorithm [36] currently running in our pipeline makes use of global image information such as average luminosity, and is therefore run across the whole frameset simultaneously, instead of processing each image separately. This is to ensure consistent colors and exposure, and helps the Stitcher in its work to merge frames into a panorama. The module's final output is a single frameset, in YUV444. An example of the HDR module's effect is shown in figure 2.5. In relation to work done on the HDR module, a paper detailing different approaches to HDR along with performance evaluations was presented in [31]. Also, a survey of real-time HDR algorithms was given in the master's thesis of Lorenz Kellerer-Pirklbauer [32].

Although potentially very useful, the HDR module is not always necessary for serving the users with high quality video, as the lighting conditions might already be good enough (e.g. after sundown). Additionally, due to its inherit framerate reduction, this module is configured to be an optional part of the pipeline. If this module is disabled, the Dynamic Stitcher is run directly after the Frame Synchronizer module.

Dynamic Stitcher

The next module to run is the Dynamic Stitcher, and executes on the GPU. Its main task is to stitch all frames in the frameset into one, large image.

The first step is to project each of the 5 single images separately onto a surface. In the previous pipeline, the images where projected onto a vertical plane, forming what is called a *rectilinear* panorama. There, the straight lines in the source images produce straight lines in the final panorama. However, there are disadvantages to this approach, because the large field of view tends to exaggerate perspective. The effect of stretching the images to fit the flat surface during panorama creation becomes apparent around the edges of the frame, where the pixels, and thus also objects, are distorted. When zooming in on the edges of the frames, this effect is easy to observe. The new camera array provides the same large field of view as the old and is approximately 160°. However, to avoid the negative effects of rectilinear projection, the images are instead projected onto a cylinder (or more precisely, the inside of a cylinder). Cylindrical projections maintain a more accurate perspective and naturally preserve more of the original image quality, but at the cost of rendering straight horizontal lines as curves. This happens as the cylinder is "rolled out" as a flat image during display, and is shown in figure 3.4. The cylindrical projection was, however, mainly chosen to facilitate *virtual viewing*. This allows a user to generate an infinite amount of personalized views from a single panorama, unlike a simple crop, and is further described in [18].

[7]See [46] for more information on HDR.

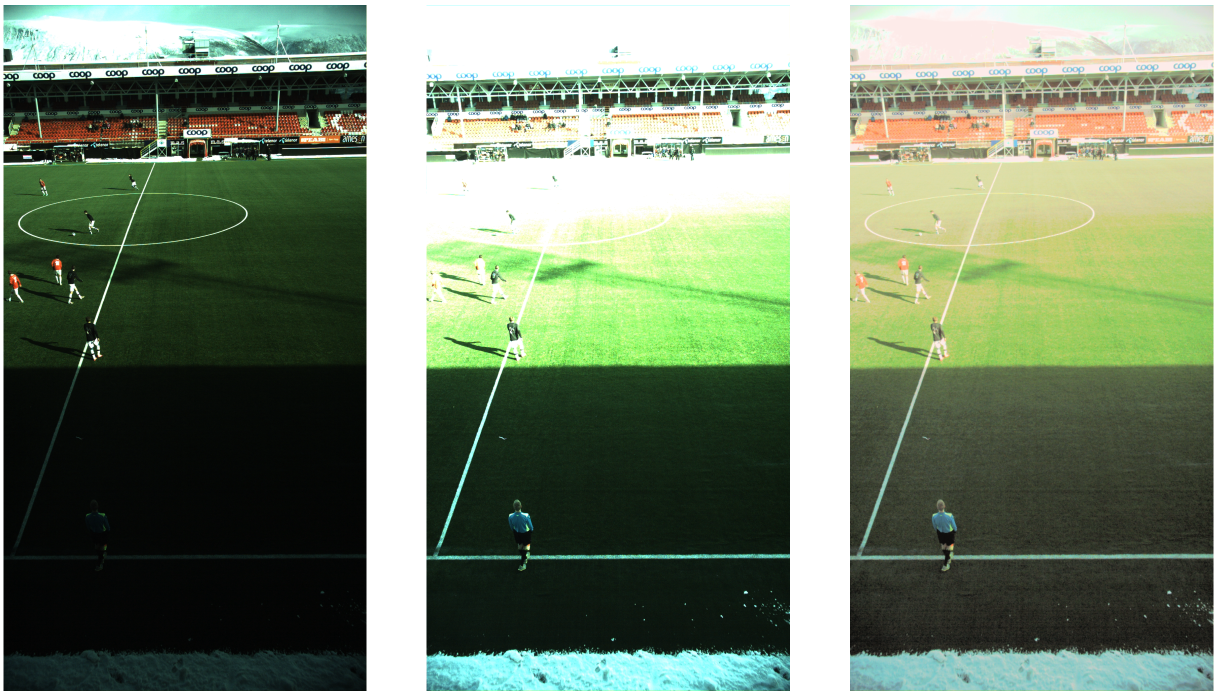


Figure 2.5: Images illustrating the effect of running the pipeline images through the HDR module. The leftmost image (low) is combined with the middle image (high) to produce the rightmost image.

The Stitcher assumes that all images are aligned with pixel-perfect accuracy, but the physical camera rig is calibrated on a best effort and does not provide such a guarantee. The task of correcting calibration errors by aligning and rotating images in this way was previously performed by the Warper module, but is now done directly during the projection phase. Due to the computational complexity of the correction algorithm (execution time of approximately 300 milliseconds), this is done offline and manually, using a separate program. After projecting the frameset onto a common surface and aligning them, they are stitched together. The stitching algorithm now used is an extension of the algorithm presented in the thesis of Espen Helgedagsrud [23], part of the previous pipeline version. In addition to perfect alignment, stitching algorithms assume that colors are perfectly consistent as well. To ensure consistent coloring across frames, the old Bagadus pipeline made use of a Color Corrector module described in section 2.1. This is now replaced by a process called *seam blending*, implemented directly in the Stitcher. More about this process and further details on the implementation of the Dynamic Stitcher can be found in the master's thesis of Ragnar Langseth [35].

The total execution time of the module is 5.99 milliseconds on average per frame, or 166 frames per second, which means it runs well below our real-time threshold. The final image is downsampled from YUV444 to YUV422, and the image resolution is also reduced to 4096 $\times$ 1800 in order to conform to the standard for 4K video, enabling much higher video portability.

GPU Downloader

The next module to run is the GPU Downloader module, running on the GPU. The panorama is now in its final form and is ready to be written to disk. First, however, the panorama must be transferred to the CPU for the video encoder to process it. Like in the GPU Uploader module, this is done by asynchronously copying the panorama from GPU memory to pinned CPU memory using CUDAs memory API functions.

H264 Encoder

Running as the last module in the pipeline, we have the H264 Encoder module. Executing fully on the CPU, its job is to encode the panorama using the H.264 video compression standard [25] into MPEG-4 AVC video segments, and writing the results to disk.

The encoding is done through the widely used *libx264* library [44]. Video encoding is highly CPU intensive and encoding our panorama utilizes almost the entire CPU. The library has a vast amount of configuration parameters and settings, but also provide overall *profiles*. In our pipeline, we configure x264 with the *high* profile setting ensuring high image quality, and the *ultrafast* preset to keep the encoding times real-time (further stressing the CPU). H.264 makes use of the fact that from frame to frame, there is often change only in small areas at a time, while the rest remains static. This is exploited heavily in H.264 and leads to much faster encoding when there is little happening in the frame. Similarly, when there is lots of movement in several areas of the image, e.g. when cameras produce noisy output due to dark lighting conditions or there is heavy rain, the encoding takes much longer. The file sizes also decrease and increase, respectively (anywhere from roughly 3-16 MB for each segment). This ultimately leads to a very unpredictable execution time, and if the encoder fails to meet the real-time requirement, previous frames are duplicated to catch up.

After encoding 3 seconds of video, the filestream is closed and the contents are written to disk, where the filename is a combination of the timestamp of the first frame in the segment, and a sequence number. This allows easy and highly accurate video extraction for later, offline purposes. Also, the file is added to a live manifest which can be easily utilized for live streaming over e.g. the Hypertext Transfer Protocol (HTTP), adding yet another opportunity for video extraction.

2.4 Summary

In this chapter, we have given a presentation of the old version of the Bagadus system. An architectural overview was provided showing each module's place in the pipeline, as well as a detailed description on each module's implementation in turn. We learned that the frames are first captured by the cameras, and after being processed by each module in the pipeline, both single streams from each of the 4 cameras, as well a rectilinear panorama, are written to disk for later extraction. We reviewed the limitations in the initial design, and saw that the tight coupling between modules and the controller made module replacements difficult. Additionally, we also saw a general low visual quality, caused by low resolution cameras and the perspective warps generally present in a rectilinear panorama.

Next, we introduced our redesign of the pipeline, where the system was scaled to handle new and higher resolution cameras for increased visual quality. The cameras were rotated 90°, to enable a higher vertical field of view, and mounted in a circular pattern to reduce the parallax effect. Instead of using one capture machine, the video capture was split into three different machines that handled one camera each. The frames were then transferred over a high-speed PCIe link, providing much higher bandwidth and lower latency compared to the Gigabit Ethernet link used in the previous pipeline. The Frame Synchronizer module was responsible for gathering frames from each capturing machine and ensuring the frames were captured at the same time. The five single frames from each camera were then compiled into a synchronized frameset to prepare for panorama stitching. We also gave a detailed description of the optional HDR module, which combined several exposures to construct a single HDR image that ensured consistent coloring and a compressed color range during extreme lighting conditions. Next, we reviewed the improvements on the stitcher module, which was presented as an extension of the stitcher in the old pipeline. Here, a cylindrical panorama was generated in real-time on a GPU, enabling a more accurate perspective while also facilitating virtual viewing. As a replacement of the Color Corrector module, the new stitcher made use of *seam blending* to ensure smooth transitions between frames in the panorama. Finally, we reviewed the H264 module, responsible for encoding and writing the finished panorama to disk.

We have seen how the redesigned distributed system using higher quality cameras is able to remain real-time, while also offering new services that were previously unavailable, such as virtual viewing. In the next chapter, we will introduce object tracking. Here, we outline important concepts and related work, and investigate the requirements and challenges of a tracker running on the high resolution cylindrical panorama produced by the new recording pipeline.

Chapter 3

Object tracking

In short, object trackers aim to track an object's movement through a series of frames. First, the tracker is initialized with the object's position in the first frame of the video, given by a bounding box that is either drawn directly by the user, or preconfigured. Next, the tracker will attempt to generate the object's trajectory by estimating its position in each of the following frames. Additionally, the tracker might provide more information, such as the object's texture, orientation or shape. Trackers can be very fast, but rarely robust. If the object leaves the frame completely, there is no way for the tracker to recover. This issue is addressed by combining tracking with a separate research field, called object recognition or, more specifically, object detection. Object detection is massive field of research, and aims to solve the problems of recognizing any object present in the scene, and if so, what kind of objects (cars, people, buildings etc.). We will not dive deep into this field, but there has been significant progress in object detection, where much is instantly applicable to tracking. Instead of only initializing the object position and appearance on the first frame, we can utilize object detection to periodically repeat the process during the tracking, increasing the tracker's robustness. Using detection on every frame, we can track the object by simply linking our detections, forming what is called *tracking-by-detection*. This form of tracking has gained a lot of popularity, particularly due to its ability to handle more challenging conditions than what tracking can provide on its own.

The purpose and end goal of an object tracker is easy to describe and formulate as a problem, but the task has proven to be very difficult. Ideally, we would want a tracker that can track arbitrary objects in arbitrary situations. We would like to be able to track any object regardless of its appearance or moving speed, while still keeping execution in real-time. However, no such general algorithm has been proposed providing the needed reliability and speed of execution (object tracking is very often aimed at real-time applications). Instead, domain specific trackers are made to address the scenario at hand, by limiting the problem and adapting the tracker to its operating environment. For example, while tracking a geometric shape like a ball, the additional mechanisms for recognizing a human shape are not needed. Conversely, a tracker directed at humans walking through a corridor does not need to handle the high speeds that balls can have in certain sports.

3.1 Background

When designing a full object tracker, there are several key points which greatly affect the foundation and abilities of the tracking algorithm. Reviewing these aspects will help understand the parts that make up a tracker, and how it affects performance. We start by delving into the object representation, which determines how the algorithm represent the object in memory. Next, we have a look at feature selection, where the object's identifying traits are leveraged to separate it from other elements in the scene. Finally, we briefly discuss object detection.

3.1.1 Object representation

Choosing a good representation for our object is crucial, because of its strong coupling with the tracking algorithm. The tracking algorithm makes use of the representation to recognize the object in the scene, as we will see, and help facilitate the tracking. The object must be modeled, and according to A.Yilmaz et al. in [66], an object can be represented by shape and appearance. A number of possible representations are presented, but we will have a look at the ones best suited for our domain. The shape of an object can be represented in a number of ways:

Points: The object is represented by a point, the centroid, or a set of points. In general, the point representation is suitable for tracking objects that occupy small regions in the image.

Primitive geometric shapes: Object shape is represented by a geometric shape, usually a rectangle or an ellipse. These shapes define the boundaries of the object. An ellipse or circle is fairly close to how the ball will look most of the time. The tracker might begin by locating circle or ellipse like objects in each image, generating a set of possible ball candidates.

Object silhouette and contour: The object is defined by its boundaries, but is not restricted to simple geometric shapes. This can be used to track more complicated shapes like humans or animals.

After determining a good shape representation, we must establish a good way of modeling the appearance of our object.

Probability densities of object appearance: These approaches calculate probability densities for the object appearance features and look for peaks in the distribution in order to help locate the object. A popular combination is selecting color as a feature, and using the peaks in the probability densities of histograms to find the regions most similar to the object of interest [10].

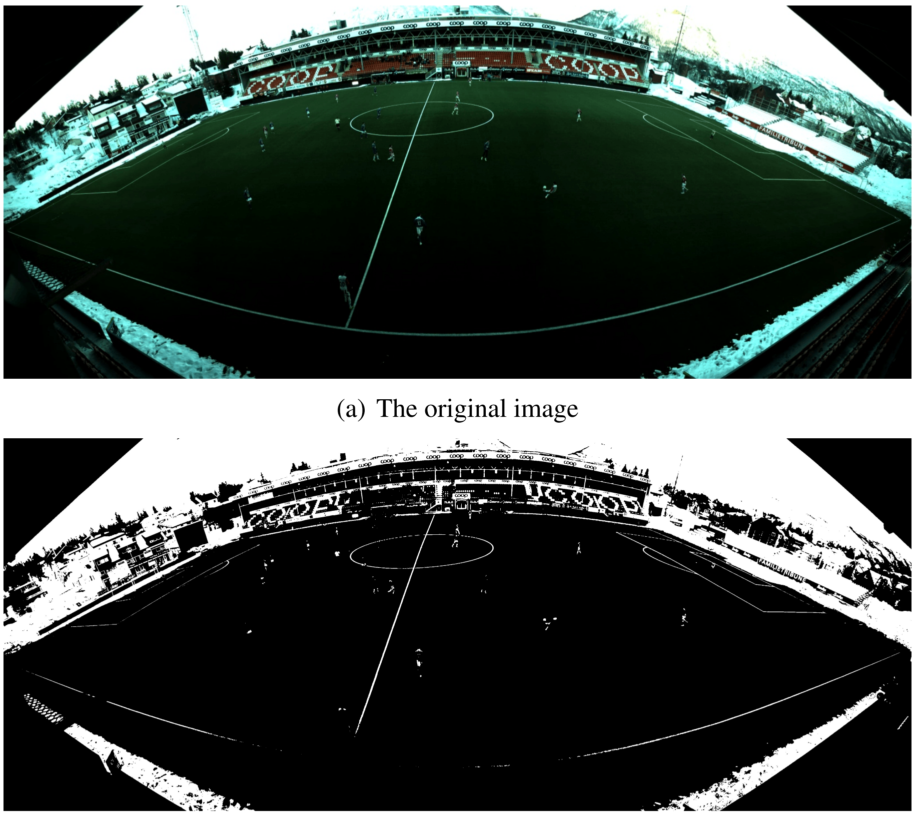
Templates: Example images of the object are used as templates to match over the image. This approach is very susceptible to changes in viewing direction, lighting/color or scale variations, and hence, an extensive set of images is needed. We can never enumerate all possible variations and establishing a robust and useable appearance model in this way is thus very challenging in dynamic environments.

Active appearance models: Landmarks, the set of key features part of the object (such as the outer boundary of a ball), are used to define the object shape. The shape and appearance is modeled simultaneously by creating an appearance vector containing texture, color or gradient magnitude for each landmark. Vectors are then generated for a set of sample images and used as input to a training phase using, e.g., Principal Component Analysis (PCA). PCA is a classic statistical method that reduces (the vectors') dimensionality by locating the most essential attributes and variables, called the Principal Components, while still retaining as much as possible of the variation present in the data. This compacts our object model, and is also efficiently computed[1]. The same approach can also be applied to multiview appearance models, where different views of an object are encoded as well.

[1]For more information on PCA, refer to [27]

3.1.2 Feature selection

Feature selection is the concept of identifying certain traits of the object of interest, like shape, color and surface texture, that ideally are unique in feature space and consistent throughout the tracking. These features are then used to separate, or segment, the object of interest from other parts of the scene. This not only aids the tracking, but also helps to avoid object representation drift. Selecting features is strongly coupled with the object representation. If a geometric shape was used, the edges of the object might be a wise choice. Or similarly, if the representation is based on texture or surface appearance, color could be used as a feature. An important observation, though, is to consider what features are actually present in the background or domain already, and aim to select features that separate the object from the scene. If only color is used to identify for example a ball, shirt colors or background objects similar to it will confuse the tracking algorithm (see figure 3.1). However, color has still proven to be very useful, and performs well in many situations.


(a) The original image

(b) The thresholded image

Figure 3.1: Image run through a white color filter. We can easily see the noise introduced by the white sky and remains of snow on the outside of the field.

Feature selection is an integral part of object tracking, but properly selecting features can be a difficult process. Completely unique features rarely exist and objects might not have any clear features at all. The texture of a ball is unclear and difficult to extract on cameras installed in a configuration overlooking a soccer field, far away from the ball. Most people would also agree that a ball is round, but during motion blur, this is no longer the case, as seen in figure 3.3(c). Moreover, the edges of the ball might also appear broken due to internal color variations (like black spots on an orange ball), as illustrated by figure 3.2.

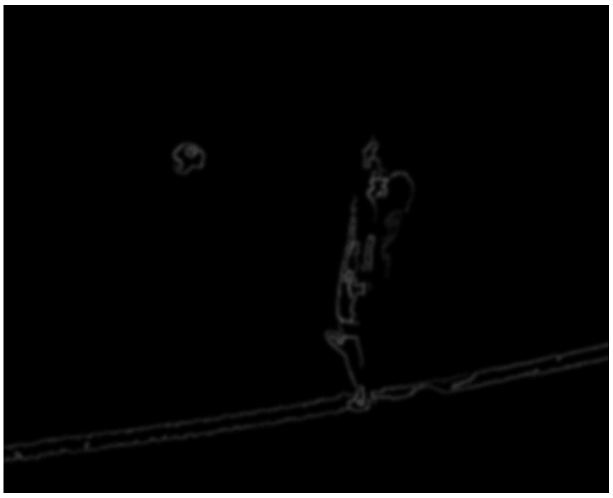


Figure 3.2: Roundness of ball is lost after edge detection

3.1.3 Object detection

After establishing an object representation and what features are to be used, an object detection mechanism is required in order to initialize the tracking algorithm. The detection step can either be performed only in the first frame (the tracker is then responsible for not losing the object until the sequence is finished), or it may be run on every frame, forming what is called tracking-by-detection [3, 9, 29], as mentioned previously. This approach has seen a big rise not only because of the great progress in object detection in recent years, but because of its robustness and ability to handle more challenging cases, such as occlusion and object exit/reentry. Due to the complexity of ball tracking in team sports, we will aim to detect the ball in every frame, and using the detections to facilitate tracking. We will present possible approaches to detection and tracking along with our case studies in chapters 4, 5 and 6.

3.2 Goals and requirements

The main purpose of our research is to introduce an object tracking module into the pipeline. Given a frame, which is already processed by the panorama stitcher and possibly other preceding modules, as input, the tracker module outputs an accurate estimate of the 2D position of the object's center as a pair of (x,y) pixel coordinates. For each frame (or sets of frames), these pairs can either be written to disk for later reference, or be passed on to succeeding pipeline modules directly for online use. The tracking data can be used for a great number of purposes, such as guiding the virtual camera and thus automating an otherwise manual task performed by a cameraman [18]. Also, in soccer, the tracking data can help referees in making difficult decisions about incidents such as goals and offsides. Offline, statistics like ball possession percentage, speed during kicks or passes, and the number of completed passes can also be extracted.

Regardless of the scenario in which the tracker operates, our most important goal for the tracking module is the ratio between tracked frames and the total amount of frames. This metric will be called *retrieval rate*. We will define two submetrics, given in percentage, where the first is the *accurate retrieval rate*, represented by the following equation:

$$\frac{F_a}{F_t} \times 100 \tag{3.1}$$

where F_a represent the number of accurately tracked frames (accuracy will be defined in a moment) and F_t represents the total amount of frames in the sequence. The second metric will be called *inaccurate retrieval rate*, and is shown in the following equation:

$$\frac{F_a + F_i}{F_t} \times 100 \tag{3.2}$$

where F_i is the number of tracked frames that are classified as inaccurate, while F_a and F_t remain the same as above. We define an *accurate* frame to be one where the tracker's estimation of the object's location is within a permitted range of the ground truth (the "true" position of the object). We will define an accurate frame by the following equation:

$$abs(X_e - X_g) < T \ \wedge \ abs(Y_e - Y_g) < T \tag{3.3}$$

where X_e and X_e is the tracker's estimation of the X and Y coordinate of the object's center, respectively. X_g and Y_g is the X and Y coordinate of the object's center, as given by the ground truth. T is our accuracy threshold, or range, which will initially be defined as *25*. This equation must hold for a frame to be classified as *accurate*. For an *inaccurate* frame, this value of T is set to *35*. If the tracker's estimate fails to stay below the threshold for an inaccurate frame, i.e. larger than *35*, the frame is counted as a *lost* frame.

A retrieval rate of 100% means that the tracker was able to output the object's position in every frame it was given, with satisfying accuracy. First and foremost, high retrieval rate is achieved by implementing a robust tracking algorithm that is able to handle the challenges present in the domain, such as occlusions, changing lighting conditions and possibly unpredictable movements. Secondarily, we must be able to redetect the object should the tracking itself fail. For example, losing track of the object can happen during very difficult single frames, or if the object leaves the frame completely. Furthermore, we would like a certain level of accuracy. Accuracy is defined as how close the tracker's estimated position of the object is to the object's "true" position (what is called the *ground truth*). Accuracy is dependent on the resolution of the images being processed, but in general we wish to achieve a high enough accuracy to remove any possible ambiguity, especially in regards to other objects in the scene. Naturally, we thus consider the size of the object we wish to track, relative to other objects. Humans from surveillance footage are often close to the camera(s), and pixel-perfect accuracy is simply not needed. Since the entire body is tracked, a certain level of inaccuracy will still provide satisfactory and usable results. In tracking applications where the object is potentially only a few pixels wide, such as the ball tracking case study seen in chapter 6, a much higher accuracy is required.

It is important to once again note that high retrieval rate is our primary objective. A highly accurate tracker is useless in a real-time scenario if it simply cannot locate the object in most of its frames. Although a secondary objective, high accuracy is extremely valuable and is often necessary to provide any real usable results. The tracker's output could be used to e.g. generate ball possession statistics, but such a metric is useless if not accurate, even if very high retrieval rate is achieved.

Another very important property, is the tracker's ability to process frames quickly. As mentioned briefly already, object tracking is very often used in real-time applications. A tracker with 100% retrieval rate and a pixel-perfect accuracy is useless if it cannot process and deliver the frames when they are needed. What is defined as real-time depends on the resolution of each frame, and the number of frames per second (FPS). The real-time requirement can be represented by the following equation:

$$\frac{\text{1 second}}{\text{FPS}} = 40ms \tag{3.4}$$

where FPS is the framerate of the cameras used, which is *25* in our case. The tracker must be able to process one frame in *40* ms in order to meet the real-time requirement of the system.

3.3 Related work

The possible usages of having a successful tracker are practically endless and thus object tracking has been the subject of a lot of research over the years. As mentioned in the introduction, this has spawned a number of different ways to approach the tracking problem. Some track the object through established points in the object combined with motion correspondence between frames [60], while others do shape matching based on geometrical models [49]. Successful and more recent approaches include [10], which utilize color histograms to represent the target coupled with the popular mean shift procedure for tracking; [3], which classifies objects and online updates of the appearance model; and [29], with their Tracking-Learning-Detection (TLD) framework. Another popular approach is tracking through prediction, using statistical theory like Kalman or particle filters, as seen in [42, 4]. These and similar approaches can be of great aid in scenarios where measurements are noisy or missing, and can help guide the system through uncertainty (e.g., during occlusion).

In the commercial world, there is reliable software available for ball tracking. Hawk Eye [22] provides robust and precise tracking for sports such as tennis and cricket, and is used heavily in the real world (its Goal Line Technology was used in the English Premier League during the 2013-2014 season). When tracking in tennis, Hawk Eye uses a multitude of high speed cameras to capture the scene from multiple views. Using video processing[2], the ball's 2D position is detected in every frame in each camera, while using reference points such as the court lines to compensate for camera movement. Leveraging the 2D position information from each calibrated camera, a 3D position is triangulated. Run on every frame of the video, this procedure generates a set of 3D positions, and finally a ball trajectory, which in turn is used to calculate contact points on the field. This ultimately aids referees determine whether the ball bounced outside or inside the court. Single object tracking in these kind of scenarios is a problem now considered solved, but the system has limitations. A tennis ball is very rarely occluded due to the individuality of the sport, and due to the use of multiple cameras, a good view of the ball can be found at all times. Furthermore, the lighting conditions are fairly stable and the system uses expensive hardware and high framerate cameras to reduce the effect of motion blur caused by the high velocity of the ball. However, no tracking is provided for team sports such as associative football (besides goal line assistance), basketball and ice hockey.

Instead of having to create a generic tracker[3], making good use of contextual information is key to increase the reliability and performance of a tracking algorithm. Examples of usable context information in soccer, is reasoning about the movement of players and their relation to the ball's movement. Some approaches thus aim to jointly track both players and the ball, to e.g. handle situations where the player and ball trajectories overlap in space-time [69, 38, 43]. However, for ball possession, these rely on ad-hoc rules that were manually configured. Exploiting the fact that soccer is a sport based around the ball, [47] made separate physics-based motion models for different phases the ball was in (flying, rolling, in-possession). Similarly, in 2013, Xinchao Wang et al. [61] presented a complete tracking system centered around tracking the players and assigning ball ownership, before utilizing player trajectories to achieve reliable ball tracking (searching around the vicinity of the player in possession of the ball). This is very different from standard approaches where ball ownership is determined only after a confident ball candidate has been generated. Although aimed at arena sports in general, the system was primarily developed for basketball, where substantial amounts of occlusions occur constantly when the players are in possession of the ball. However, as an indoor sport, the lighting conditions

[2]As a commercial system, Hawk Eye's source code is unavailable to the public. Implementation specific details are thus not provided.

[3]Generic in the sense that the tracker would work in any scenario irrespective of conditions and other challenging factors.

are more stable.

While the system presented in [61] shows very promising results, multiple cameras from different views are a necessity to facilitate its triangulation features. The multiple cameras guide the system through occlusions, but our one, although very wide, panorama, does not provide the algorithm with enough viewpoints (cameras are only mounted from one perspective in all case studies). For proper coverage, the Bagadus system could be scaled up to handle multiple cameramounts producing panoramas from several different angles. With proper camera calibration and intercamera mapping, a similar approach could be applied to our case studies. Accurate player tracking, which we use to determine ball ownership, is already supported in Bagadus through the ZXY system. At this time, however, we only have one perspective, and are therefore unable to leverage such functionality. Also, this would demand a quite large amount of cameras to provide the needed coverage for the large area of a soccer field, although only one additional camera array would be of great aid to the tracker.

Although our main objective is to do research towards a tracking module for use in Bagadus, we also investigate the performance and usability of object tracking in other scenarios. Our three case studies together aim to provide an insight into what services object tracking can provide to a system, and how it may be implemented. In the next section, we go on to present common challenges faced in object tracking in general, and discuss how these map to our three case studies.

3.4 Challenges and limitations

There are many challenges to be dealt with in order to provide stable and robust object tracking, especially in a demanding scenario such as Bagadus. A survey was made in 2006 by A. Yilmaz et al. [66], giving a complete categorization, comparison and description of many different trackers. A number of challenges were mentioned, and we present the most relevant here, along with others introduced by our domain.

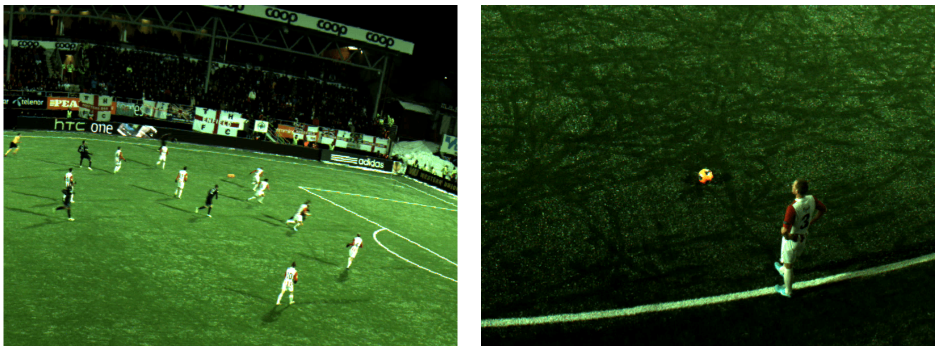
- *Occlusions:* In most scenarios, the object to be tracked is not the only one in the frame, and it may be partially or fully obstructed visually by other objects in the scene. This is called an occlusion. Occlusions present a great challenge to the tracking algorithm not only because a full view of the object is obstructed, but because whatever is obstructing the object may also be very similar in e.g. color, texture, shape or size. Ideally the tracker needs to yield useable results even with very small (or no) visible object parts. Particularly in arena sports like soccer, due to players continually occluding the ball during play. Also, while tracking humans in surveillance footage, similar to the case studies in chapters 5 and 4, the object might be partially covered by other people or structures in the scene. As we have seen, trackers often deal with these issues by making use of several cameras covering multiple views, decreasing the chances of lost tracking and effectively increasing its robustness.

- *Illumination changes:* To successfully track an object among other objects not of interest and avoid mixing with the background, we must establish distinguishable features of that object. Many trackers represent the object by using color histograms. A color histogram is a way to represent an image or an image region by its distribution of colors, where each color is represented as a number of how many pixels have that specific color[4]. By extracting the image region containing an object, we can compute its color histogram and create a model for the object appearance. Most color bands are sensitive to illumination

[4]This only applies to digital images.

variations (changing lighting conditions), and hence presents a challenge to the tracker when the histogram's values will change correspondingly.

- *Object appearance:* The shape and texture of the object can change during tracking. In applications tracking varying shapes like animals or humans in motion, the tracker needs to handle large internal variations (moving arms, legs etc.). This is however not a substantial challenge because the ball is a rigidly shaped object and will have few internal variations. However, the ball will have large scale variations due to cameras only being mounted on one side of the field. Furthermore, high speeds can make the ball stretch out. Figure 3.3 shows how the ball appearance changes as an effect of position 3.3(a) and 3.3(b), and speed 3.3(c).


(a) The ball is at the far side of the field, and appears small (b) The ball is close to the cameras, and appears big

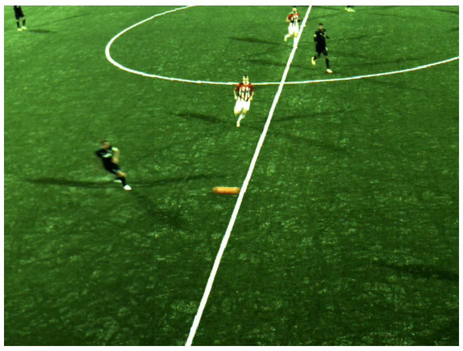
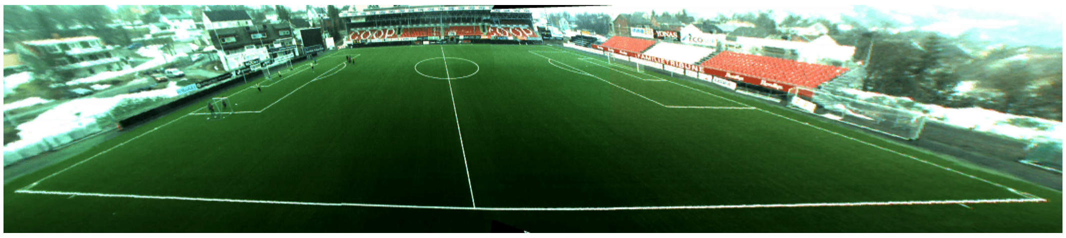

(c) The speed of the kick is too high for the cameras' supported framerate, and results in motion blur

Figure 3.3: Object appearance variations

- *Object motion:* The motion of the object can be hard to estimate due to high speed, resulting in large movements from one frame to the next, and complex patterns. In a soccer match, the ball trajectory is unpredictable caused by sudden changes in velocity and direction, e.g. during tackles and dribbling.

- *Processing requirements:* Object tracking is usually aimed at real-time applications and thus subjected to hard processing requirements. A tracker must be able to keep up with the image input rate in order to track the object through every frame. Even though much research has been done in object tracking, most trackers [4] work with images with much lower resolution than what our cameras produce. The new cameras in combination can provide a panorama image of up to 4450 $\times$ 2000 pixels. The difference between the old pipeline (max resolution of 1280 $\times$ 960 pixels) and the new is shown in figure 3.4. Although this gives an increased degree of texture and detail, it presents a great challenge to process in real time with a framerate of 25 images per second. However, the full frame rate is not always necessary to successfully track an object and the tracker can potentially increase its performance by avoiding processing of each frame, as detailed in [33]. Dropping frames to improve performance can be tempting in a real-time system, but due to the high ball speeds, dropping too many can cause considerable changes between frames and possibly lead to a failed tracking.


(a) Output image from the old panorama pipeline described in section 2.1

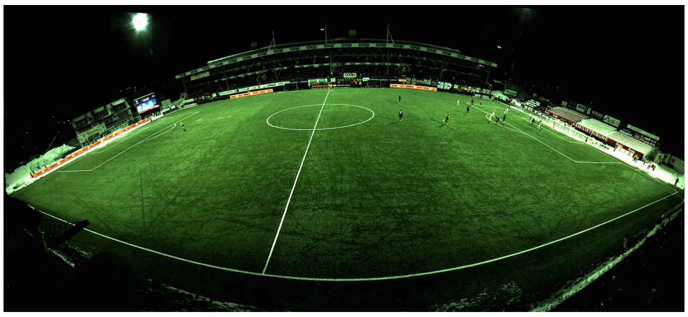

(b) Image from the new and improved pipeline, described in section 2.2

Figure 3.4: The output video of the new pipeline is of much higher quality than the previous version, but the workload of the pipeline modules is increased considerably.

- *Object representation drift:* Some algorithms use color histograms and similar features to measure similarity between detected candidate objects, e.g., the mean shift algorithm found in [10] and derivatives (CAMShift). These types of object representations rarely do well in matters of strong occlusion because the object's color histogram can change drastically and results in drift when the occluding object's colors interfere.

- *Object entry and exit:* During a soccer match, the ball will most likely be kicked completely out of the frame at some point. The tracker thus needs a way of redetecting the ball to initialize the tracking once the ball reenters, even when the position of reentry is different from where it exited the frame. Also, in footage from for example surveillance systems, objects are continually leaving and entering the frame, also requiring automatic redetection.

There are clearly a number of substantial challenges to overcome and cases to cover in order to provide useable object tracking of high-resolution images in real-time applications. Many solutions have been presented, where some of the most promising ones make use of multiple cameras to cover additional viewpoints. Even though our system outputs a panorama with very wide coverage, only one viewpoint is provided. We are thus forced to instead introduce further computational complexity as a way to cope with the lack of information. In the next section, we dive into the current state-of-the-art in single object tracking.

3.5 Establishing test trackers

As implementing a full tracker is a challenging task, we wanted to explore how already established trackers would perform on video from Bagadus, as a supplement to our own implementations. A fair comparison and evaluation of all the object trackers being continually produced by research is difficult, as each tracker's data annotation format is different, as well as how and on which sequences the experiments are run. A proposed solution to these issues of uniting trackers and to fairly compare the results, was presented by Y. Wu et al. [65], through a common framework. The authors provide a code library and framework including a large suite of trackers, a dataset containing over 50 fully annotated sequences, and finally a robustness evaluation. Three measures make up the robustness evaluation; one-pass evaluation (OPE), where the initialization of the bounding box is set to the ground truth on the first frame[5]; temporal robustness evaluation (TRE), where the tracker starts at different frames; and finally, spatial robustness evaluation (SRE), where the tracker starts using different bounding box sizes and positions during initialization (e.g. including less or more of the background, or only parts of the object). These metrics are meant to model the inaccuracies of an object detector, and provide a broader test of the tracker's capabilities than what OPE would provide alone. Also, each sequence is tagged with an attribute, denoting what situations (or challenges) it presents:

Illumination Variation (IV)
the illumination around the object is significantly changed.

Out-of-Plane Rotation (OPR)
the target rotates out of the image plane.

Scale Variation (SV)
the size of the object changes.

Occlusion (OCC)
the target is partially or fully occluded.

Deformation (DEF)
the object deforms and loses its original shape.

Motion Blur (MB)
the object appearance is changed due to high speed.

Fast Motion (FM)
the motion of the ground truth is larger than 20 pixels per frame.

In-Plane Rotation (IPR)
the target rotates in the image plane.

Out-of-View (OV)
some portion of the target leaves the view.

Background Clutters (BC)
the background is similar in color or texture to the object

Low Resolution (LR)
number of pixels inside the ground-truth bounding box is less than 400.

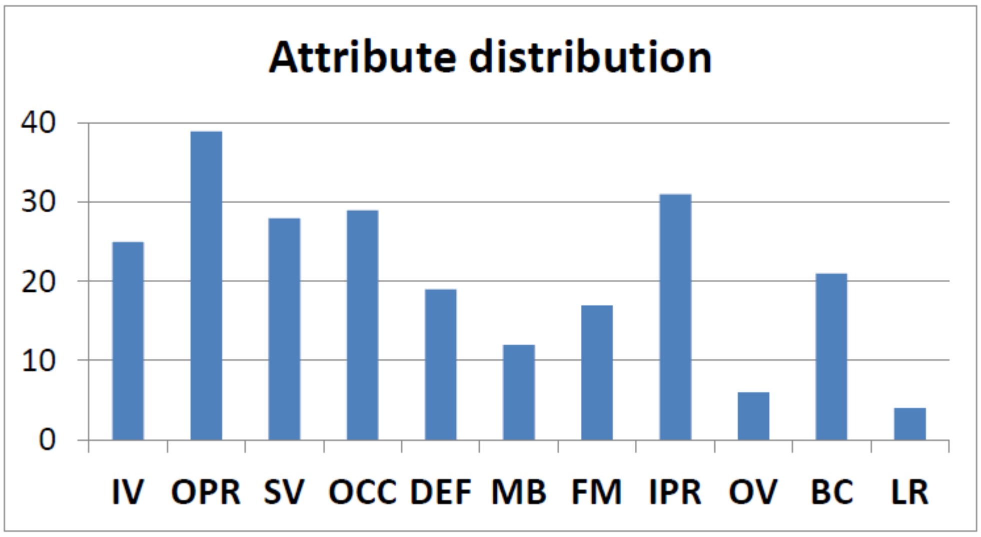


Figure 3.5: Distribution of sequence attributes across the dataset. Attributes are listed along the x-axis, while the number of sequences with each attribute listed on the y-axis. Image from [5].

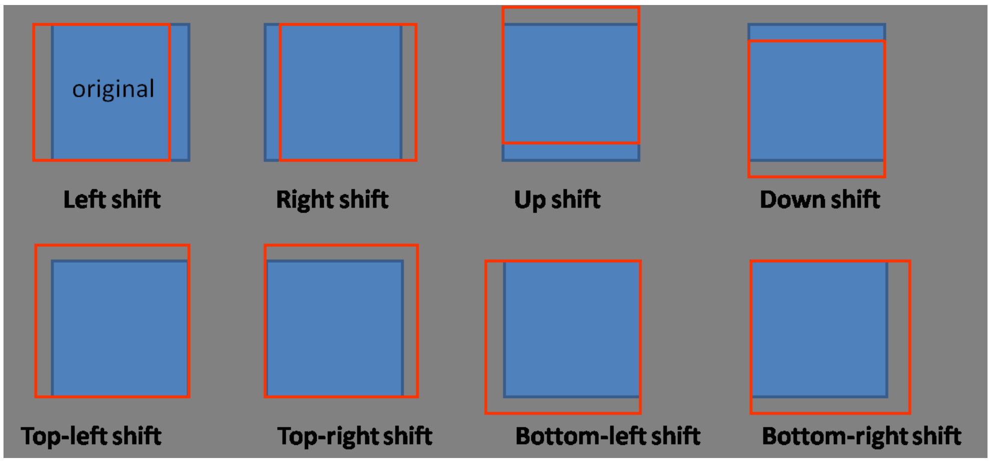


Figure 3.6: Image showing the bounding box shifting operations done during SRE. The shifts are 10% of width or height of the ground truth bounding box. Image from [5].

Figure 3.5 shows the distribution of attributes across the entire testset. The full dataset of sequences on which all trackers were tested, can be found at [5].

A total of 29 trackers were tested and benchmarked on all 50 sequences in the dataset, with all three robustness evaluation configurations enabled. “For OPE, each tracker is tested on more than 29,000 frames. For SRE, each tracker is evaluated 12 times on each sequence, where more than 350,000 bounding box results are generated. For TRE, each sequence is partitioned into 20 segments and thus each tracker is performed on around 310,000 frames.” [65]. Combined, all these sequences and measures provide a substantial amount of information about the trackers’ performance. Figure 3.7 shows a summary of their results, where the tracker names are listed on the X-axis, while the Y-axis represents the success ratio. The success ratio is defined as the ratio of the frames whose tracked box has more overlap with the ground truth than a set threshold. The tests are run using threshold values between 0 to 1, and the area under curve (AUC) is then used to rank the tracking algorithms. The numbers in the top center above the bars represent the initial bounding box size. “X 1.0” is the original bounding box size, while “X 0.8” and “X 1.2” represent a 20% reduction and increase, respectively.

[5]This is considered the norm and how trackers are normally initialized.

We used the results from this benchmark as a basis for selecting which trackers to best test and represent the capabilities of the current state-of-the-art within single object tracking. As will be discussed further in the case studies, we will be focusing our work on tracking-by-detection, which provides not only automatic initialization, but also helps restart the tracking if it fails. These detections are often inaccurate, and thus knowing the trackers we choose have performed well during SRE (spatial displacement of bounding box) and TRE (tracking starting from different frames) is an important observation. Furthermore, as seen in the attribute distribution of the sequences, many of the tested sequences include Occlusion (OCC), Illumination Variation (IV), Scale Variation (SV), Background Clutters (BC) and, although to a lesser degree, Motion Blur (MB) - all of which are highly applicable to the panorama videos presented in our case studies.

As shown in figure 3.7, SCM [70], Struck [21] and TLD [29] got the three highest summary scores. The authors of SCM, however, do not provide publicly available source code usable in our project. Struck and TLD are therefore the only two trackers from the benchmark used in our testing, along with another state-of-the-art tracker named Real Time Compressive Tracker that we use as a replacement for SCM, which will be discussed later in this section. We now go on to detail how each of these three trackers work, along with additional reasons to why they were chosen.

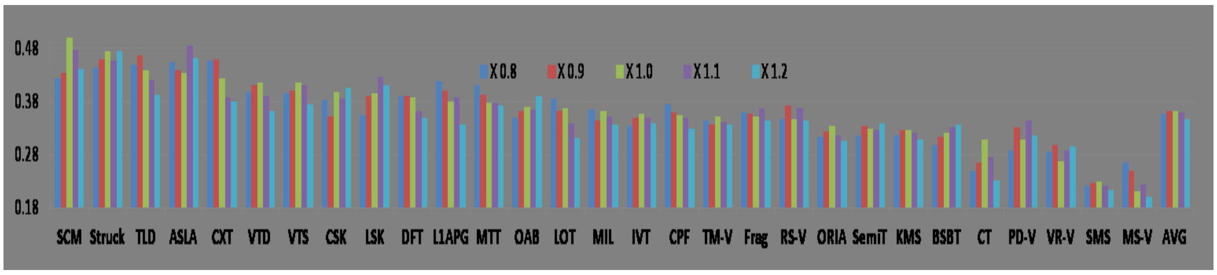


Figure 3.7: Performance summary for the 29 trackers initialized with varying bounding box size, seen in the top center above the bars. AVG (the rightmost one) illustrates the average performance over all trackers for each scale. Image from [5].

3.5.1 Tracking-Learning-Detection

The Tracking-Learning-Detection tracker [29] (TLD) has received much praise in the academic world for its ability to provide robust tracking of arbitrary objects in real-time applications, even with heavy camera motion. Its work is based around the fact that neither tracking nor detection can solve the object tracking problem independently. They must work together, and can mutually benefit each other. This is done by letting the tracker follow the object on every frame, while the detector accumulates the object's appearance, possibly correcting the tracker online[6]. Additionally, the object detector itself is trained, by estimating its errors. The detector is then updated by the output of two independent learning experts, reflecting the appearance changes in the object. Also, one of its most important goals, is the ability to work in real-time applications.

TLD is presented as a framework, consisting of three essential components: *tracking*, *learning* and *detection*. An overview of the framework is shown in figure 3.8.

- *Tracker component:* The tracking component of TLD works under the assumption that the object motion between frames is limited, and that the object is fully visible. It is based

[6]Here, online refers to work done runtime, i.e. when the algorithm is running. Offline work usually entails training the tracker by processing training examples which prepare the tracker for runtime.

on a Median-Flow tracker [30], extended with failure detection. Basically, the object is represented by a set of points within the object's bounding box. On each frame, the displacement of these points are estimated, along with their reliability. The most reliable displacements are then used to establish a model for the motion. We refer to the citation for further details on the tracking algorithm itself, as we instead wish to focus on its place in the framework.

- *Detector Component:* The detector maintains a model of the object. On each frame, the detector does a full scan of the image, looking for hits matching this object model. This matching procedure is done by use of a *binary classifier*. In tracking, such a classifier works by processing an image region as input, then generating either a positive or negative sample, based on its similarity with the current object model. The parity of these samples represents whether this region is accepted as the object or not. The output of the object detector is the positive samples from the classifier. The key aspect, however, is that instead of matching against a static model of the object, the learning component updates the detector's model. This enables the detector to adapt to changes appearing in the object throughout the video sequence, hence increasing its robustness. Such updates of the object model during runtime is known as *adaptive tracking*.

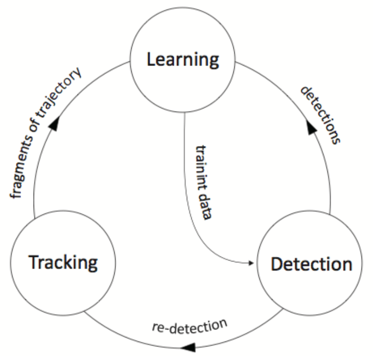
- *Learning Component:* The learning component is the key component in the framework. Its first task is to initialize the detector. This is done by using the information in the bounding box set on the first frame. A scanning grid is used to select the 10 closest cells to the initial bounding box. For each cell, or box, 20 geometric transformations (shifts, scale change and in-plane rotation) are added, with Gaussian noise on pixels. These 200 image patches are labeled as positive samples of the object, and are used to update the detector's classifier and also create the detector's initial object model. Also, negative samples are collected from the surroundings of the initial bounding box, in order to better discriminate the object from the background[7].

The detector has now been initialized with a model that can be used to detect the object in subsequent frames. In order to be able to follow any changes in object appearance throughout the tracking, this model is updated during runtime. The crucial component in this framework is how the detector is trained. The authors name this new learning paradigm *P-N learning*. For each frame, the output of the detector is analyzed by two "experts": *P-expert*, which recognizes missed detections, and; *N-expert*, which recognizes false positives. In short, the *P-expert* checks all negative samples to see if any of them should have been labeled as positive. The *N-expert* does the opposite, and thus checks whether any positive samples should have been negative. The learning framework is shown in figure 3.8(b).

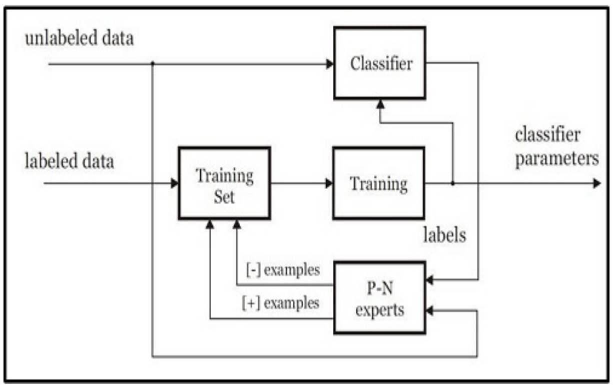
The learning component oversees the performance of both tracker and detector, and generates new training examples to correct any errors it estimates from the detector's output. The bounding boxes of both the tracker and the detector is combined into a single bounding box, which is the final output of the framework. Further details on TLD can be found online [28] or in the research paper [29].

TLD has shown to perform well in long sequences, which is paramount in a tracking application aimed at 45-minute halves like in soccer, and live streams in general. The benchmark also points out that approaches such as TLD is attractive when tracking fast moving objects. However, the performance of TLD decrease with the increase of the bounding box size, and thus, it might be less resistant to background clutter than Struck.

[7]This approach is called a discriminative tracker, where the goal is to establish a decision boundary for a binary classifier which discriminates the object from the background. Generative trackers, on the other hand, search the image for regions most closely matching the object model, disregarding the background information.



(a) The block diagram of the TLD framework



(b) Overview of the P-N learning components

Figure 3.8: An overview of the framework is seen on the left, while the learning component is shown on the right. Images from [29].

3.5.2 Structured Output Tracking with Kernels

Structured Output Tracking with Kernels (Struck) [21] is a framework for adaptive visual object tracking based on structured output prediction (SOP). Unlike classification, which outputs either a positive or a negative result based on the input, SOP aims to learn a function $f\colon \mathcal{X} \to \mathcal{Y}$ mapping inputs $x \in \mathcal{X}$ to complex and structured outputs $y \in \mathcal{Y}$. For example, in Natural Language Processing, SOP can be used to output predictions about the structure of a sentence, and label the words (which ones are the nouns, verbs etc.). Or, in object tracking, predict what the object is likely to look like in the next frame. In practice, this can be viewed as a learning task with multiple classes (many possible structures) instead of binary classification. Struck is presented as a flexible framework, where the kernel (used for evaluating patch similarity) can be easily modified to use suitable features (also, features can be combined through averaging several kernels).

Most adaptive tracking-by-detection methods approach the tracking problem by online training of a binary classifier which is given input in the form of labeled samples. First, the object location is estimated, and samples around this location are extracted. The classifier then attempts to maximize a similarity function between the sample and the current object model. Next, these samples are labeled according to a learning scheme so as to train the classifier. However, the classifier knows nothing about how these samples were extracted, and the classifier is then, in a way, unaware of the tracker's end goal. The authors of Struck discuss this decoupling. While the classifier's goal is to maximize its similarity function, the tracker's goal is to estimate object location correctly. Consequently, it is argued that the sample(s) that generates the maximum classification score, might not necessarily be the best estimate of the object's location. Most state-of-the-art tracking-by-detection methods try to address this issue by increasing the classifier's robustness against poorly labeled samples (label noise), for example by using different ways of learning. This often includes e.g semi-supervised learning or multiple-instance learning [3].

Struck's solution to the aforementioned problems, is to avoid the intermediate step of sampling and labeling. Instead, the problem is reformulated as the task of directly *predicting* the change in object location across frames, and thus merging the learning and tracking state. The prediction is done by a function, $f\colon \mathcal{X} \to \mathcal{Y}$, where $\mathcal{X}$ is an image patch and $\mathcal{Y}$ is the space of all translations/transformations (rather than binary labels). This function is learned in the Structured Output SVM framework presented in [59], where the learning method of [7, 8] is extended to tracking. A Support Vector Machine (SVM) is a general-purpose classification scheme that finds a separating hyperplane that maximizes the margin between two classes (in our case, positive or negative samples). The margin is defined as the distance from the closest point, in each class, to the hyperplane (which separates the two least similar points, or samples). In effect, the SVM will split all object samples into two classes, where the hyperplane is located in-between the least similar samples, with maximized distance, representing its corresponding class. These distinctive samples are called support vectors. We note that the number of support vectors grow linearly with the amount of training data. To avoid such unbounded growth, which naturally occurs in real-time applications such as tracking, Struck introduces a budgeting mechanism that limits the number of allowed support vectors (it is shown that the overall performance is still not degraded noticeably).

In essence, Struck is an online version of Structured Output SVM. This results in an adaptive tracking framework, where the accumulated positive support vectors at the last frame describe the history of the object's appearance changes through the sequence.

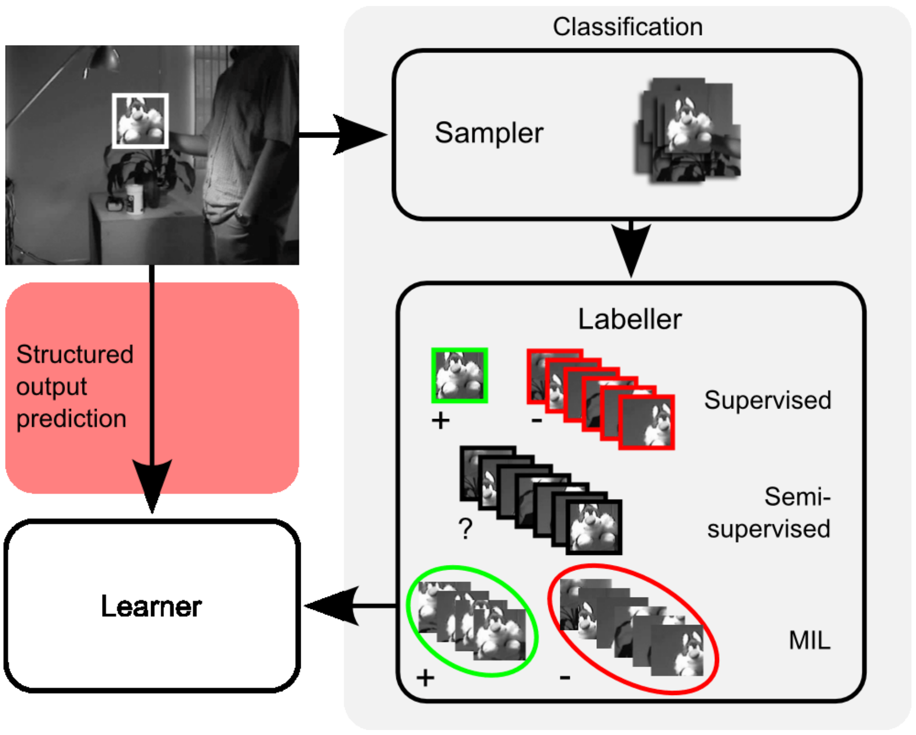


Figure 3.9: An overview of how Struck works compared to traditional learning approaches, in which a sampler and labeller selects samples as input to the learner. Struck, on the other hand, works directly on the tracker's output. Image from [21].

Results from [65] show that Struck is less sensitive to scale variations than other well-performing methods. Dense sampling based trackers like Struck and TLD, have also shown to perform well when tracking fast moving objects. One reason is that the search ranges are large, and the large and the discriminative models are able to separate the targets from the background clutters. The results also suggest that Struck's approach to structured learning is effective in dealing with occlusions compared to other trackers in the benchmark, which is a big advantage in applying a tracker to team sports (where occlusions are extremely common). It is also helpful that performance is shown to be as good, or even better, when the bounding box size is increased (background is often included in the bounding box).

3.5.3 Real Time Compressive Tracker

The Real Time Compressive Tracker (RTCT) [68] is a more recent algorithm which performs favorably against other current state-of-the-art trackers. The authors of RTCT have set up their own experiments, comparing its performance with 7 other trackers on 20 challenging sequences.

Their results indicate that the algorithm performs well compared to the other trackers (including Struck and TLD), making it a valid replacement for SCM and another good representative for what single object trackers are able to do today.

In general, trackers can be categorized as either *generative* or *discriminative*, where their names are given from their appearance models. The generative trackers learn a model to represent the object, and then search the image for regions with the highest similarity, disregarding any information about the background. Discriminative trackers, on the other hand, work by explicitly discriminating the object from the background (models both), often by establishing the decision boundary for a binary classifier. RTCT attempts to combine the merits of generative models with discriminative models, by representing the object well through features while at the same time using these features to separate the object from the background. It provides classic adaptive tracking-by-detection, by initializing the algorithm with a bounding box and learning a classifier through training sample extraction. In the following frames, the detector samples patches and feed them into the classifier, which finally selects the sample that maximizes its similarity measures. The measure is based on features for each image patch, which are normally extracted and represented in the tracker as a vector with dimension equal to the number of features found. The novelty of RTCT, however, is that its feature extraction is based on *compressive sensing* (CS).

The feature extraction is in essence a dimensionality reduction problem which can be solved in a number of ways, and is often done by using PCA as detailed in section 3.1.1. RTCT instead uses *random projection*, which has been shown to be much faster and equally as good [6]. The projection is done using a very sparse random measurement matrix[8] which is calculated offline. This process allows the classifier to compare features in a much lower dimensional space, enabling high efficiency and a small memory footprint. In summary, the tracker operates as follows:

1. The initial bounding box is given by the user on the first frame, and the classifier is initialized.
2. On each frame, a set of image patches are sampled around the previous object location. Features are then extracted through random projection.
3. The classifier is used to find the sample with the maximum similarity to the object model.
4. Positive patches (close to the current location) and negative examples (patches further away) update and learn the classifier

Further details on the tracker can be found in the research paper [68].

3.5.4 Tracker configuration and output

Each of the 3 trackers comes with a number of different configuration parameters, where most are specific to each tracker due to their inner workings. There are an enormous number of possible configurations to run and test, and we will thus not address all these enumerations. For fair comparison, we have empirically configured each tracker individually for best possible performance on each case study. Also, all trackers were given the same initial bounding box on the first frame of each sequence to be on equal ground. All trackers were configured to output their results to a file, where each frame is given a line in the format of “framenumber x y”.

[8]This matrix satisfies what is called the Restricted Isometry Property. Basically, this means that when using the matrix to project data onto a lower dimension, the distance between data points in the higher dimension is retained, i.e., we preserve the original information. The data, however, must be compressive, such as image or audio.

Average time per frame and the bounding box size is given as well. Then, for each tracker, a separate resultparser application reads in the data, looping the output results from all frames. For each frame, the ground truth is also extracted, and the tracker accuracy is measured against the two accuracy constraints described in equation 3.3.

3.6 Summary

In this chapter, we have presented an introduction to the field of object tracking. We gave a description of the most important components necessary to form a tracking algorithm, such as object representation, feature selection and object detection. We saw that we could represent our object by modeling shape using e.g. points, geometric shapes or contours, and appearance through probability densities, image templates or active appearance models. We reviewed the advantages and disadvantages of possible approaches to feature selection, and saw that we could use for example color, edges and texture.

We gave a detailed descriptions of our goals and requirements for a tracker in Bagadus, and established that we primarily need high retrieval rate, but also high accuracy if the tracking results are to be used by other services. In addition, we saw that with a framerate of 25 FPS, we must be able to process each frame in *40* ms to stay within the system's real-time requirement. Next, we reviewed related work in object tracking, and learned that popular and widely-used systems are available in the industry [22], but with limited functionality. We also saw that many try to tackle the challenge of team sports by tracking both players and the ball, and reasoning about the gameplay in different ways [38, 43, 47, 61, 69]. Especially [61] presented an exciting solution, but required cameras covering multiple views. We continued by listing a number of challenges that are necessary to overcome in order to provide stable and robust object tracking, where occlusions, illumination changes, object motion and processing requirements were extra important. Next, we presented a benchmark [65] that we used as a basis for selecting three trackers to represent the current state-of-the-art within single object tracking. We learned that Tracking-Learning-Detection [29], Struck [21] and Real Time Compressive Tracker [68] were good candidates, and gave a detailed description of each tracker and its inner workings. To close out the chapter, we described the output format and configuration of the trackers, and introduced two accuracy constrains that we use in the case studies when measuring the trackers' accuracy.

In the next chapter, we start our work on the first case study. Running a test setup in a lab at the Simula Research Laboratory, we stripped down the original Bagadus system to run using 2 out of 5 cameras. This enabled us to do early development and testing of ideas using video that is easier for a tracker to handle than the challenging footage produced by the full Bagadus system.

Chapter 4

Case study 1 - Pink Panther

4.1 Introduction

The footage from the Alfheim Stadium in Tromsø¸ is very challenging for a tracker, and in order to be able to test ideas and concepts early in the development process (like integrating ball tracking with the virtual viewer), a simpler scenario was needed. To achieve this, a full single machine test pipeline was installed in an office at Simula Research Laboratory. Instead of running the original full configuration setup with all 5 cameras, only 2 cameras were used. The *200* output images of this lab setup provide us with very useful test material for researching the potential of a tracker in a fully operational system running at real-time speeds.

We investigate the performance of a multitude of tracking approaches where the task is to track a pink beanie in an indoor office environment. One of our first intuitions when looking at tracking the ball in the Bagadus system, was using a color filter (in early testing footage the ball used was colored orange). Furthermore, the color was unique in the scene and presented a nice opportunity to use tracking-by-detection. However, we needed a simpler scenario, and constructed a lab setup where we used a beanie, colored with a strong pink. This allowed us to avoid some of the more difficult challenges (e.g. occlusions and fast motion), and provided us with an easy way to investigate how color tracking performs and compares to other tracking mechanisms.

We will start by presenting a number of simplifications we can make in this case study. Next, we go on to investigate the 3 trackers described in 3.5, along with 3 of our own. These are then detailed and compared, and a performance summary is given at the end of the chapter [1]. During work on this case study, we have published a paper where the approach of the Color Tracker was used to enable automatic guidance of a virtual view camera, following a person wearing the pink beanie [19].

4.2 Simplifications

Compared to the full pipeline described in section 2.2, there are a number of simplifying assumptions that can safely be made. Many of the limitations stated in section 3.4 do not apply for the footage provided by the new setup, and a few of the challenges are now much easier to handle. As we will see, these changes will allow us to greatly increase the reliability and performance of our tracker.

- *Lower resolution:* Since only 2 out of 5 cameras are being used, and because of camera overlap, the resolution is reduced from 4450×2000 to 2936×986 pixels (about $1/3$

[1]Throughout the case studies, we will freely reference numbers from the performance summary found at the end of each chapter

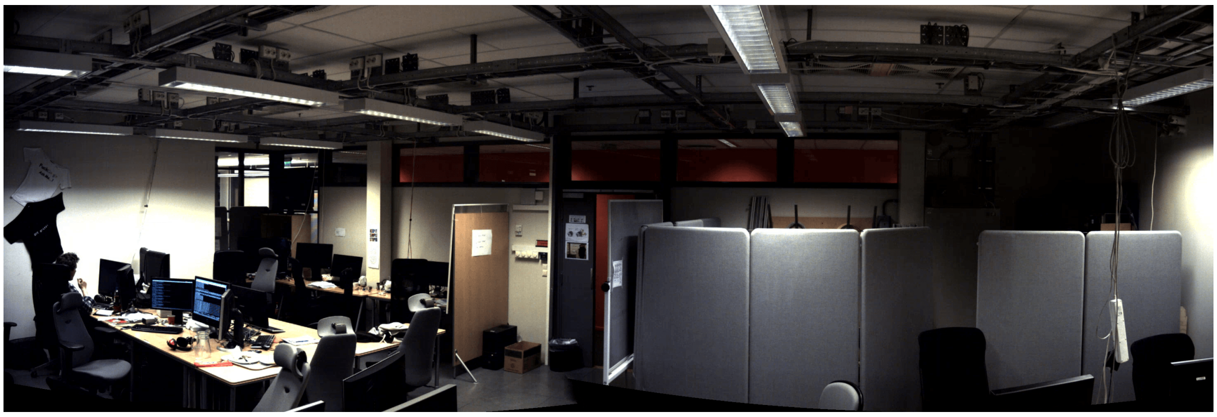


Figure 4.1: An image from the test setup (panorama from two cameras)

reduction). This gives the tracker less pixels to process per frame and thus a significantly reduced workload, increasing its performance.

- *No players:* Except from only a few frames in the beginning of the test sequence, there is only one person in the footage throughout the tracking. Occlusions become almost a non-factor. The few frames where the tracking is lost due to occlusion, we can simply interpolate as soon as the object has been reacquired by the tracker. This assumes that the tracker is given a headstart of a few frames compared to what frames the system actually outputs to the user.

- *Strong colors:* The beanie used in the footage was chosen specifically because of its strong colors. Furthermore, because the colors of the beanie are barely present elsewhere in the scene, we have a unique opportunity to confidently use color as an identifying feature.

- *Large object:* Compared to the ball in the original pipeline, the beanie itself takes up a much higher percentage of the screen's pixels. A number of false positives can arise when color is being used as a feature, but many of these can be eliminated through a priori information such as knowing that the object of interest is above a certain size throughout the tracking.

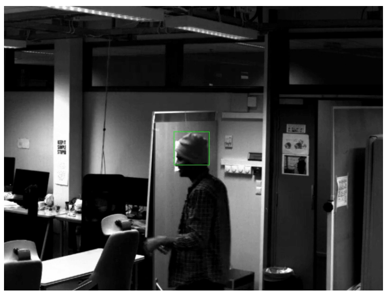
- *Motion:* The motion of the beanie during the tracking is much simpler than that of a ball. Firstly, the beanie does not nearly move at the same speed, meaning small changes from frame to frame. Second, there are no abrupt changes in direction like during tackles and dribbling, as mentioned in section 3.4.

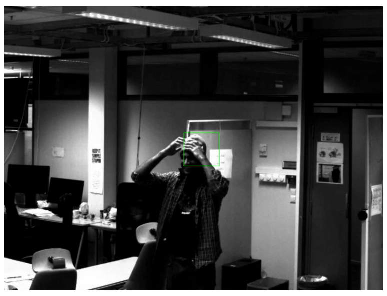
4.3 Evaluating and comparing tracking algorithms

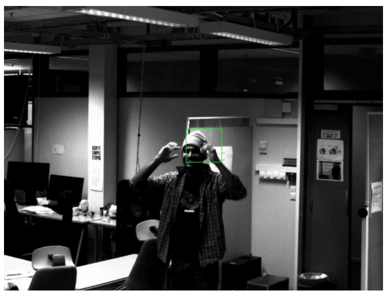
4.3.1 Structured Output Prediction using Kernels

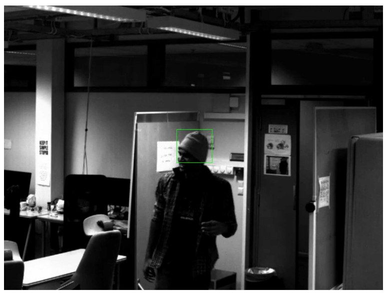
We start by evaluting the performance of the Struck tracker, which was detailed in section 3.5.2. The images in figure 4.2 show a timeline for how the tracking progresses throughout the video. Struck performs well in the beginning of the sequence, and is able to sustain through the occlusions occurring around frames 25-50. However, it starts drifting in frame #87, where the support vectors are polluted by background clutter. The lack of lighting in the image causes the few features in the beanie to disappear. The strong edges in the background are mistaken for the beanie features, and the tracker incrementally adds background segments into the positive support vector samples. This effected is clear in figure 4.3, where we see several positive samples (green bordered image patches) that contain the background clutter. This causes irreparable drift in the object representation and the SVM now wrongly classifies these clutters as part of the object, and tracking fails. The previous (correct) positive samples are still present, but the distinctive appearance of the edge in background is enough to mislead the classifier.

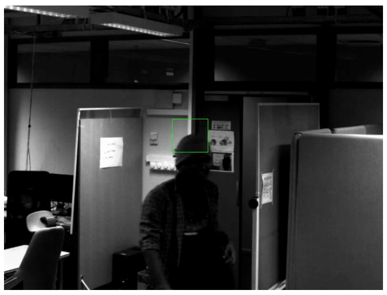
The sequence consists of *200* frames, of which *89* is tracked in total. These frames are tracked continuously (from frame 1 to 89), and the final inaccurate retrieval rate stops at *44.5%*. Struck spent *318* ms on average processing each frame, which is far from meeting our real-time requirement.


(a) #6


(b) #28


(c) #48


(d) #85


(e) #87

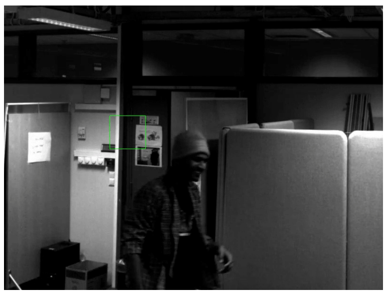
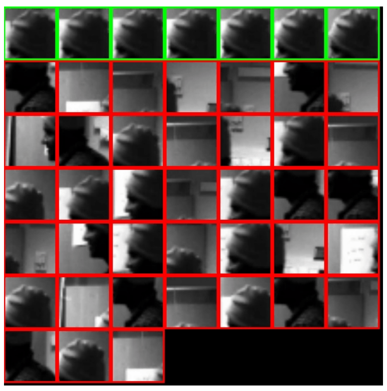
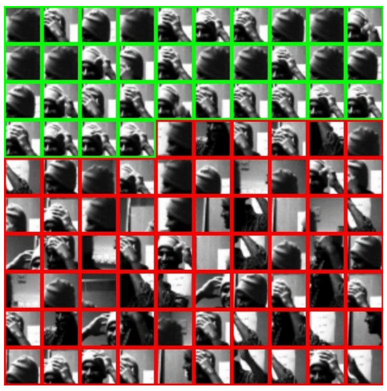

(f) #97

Figure 4.2: The green rectangle marks the Struck tracker output


(a) Support vectors at frame #6


(b) Support vectors at frame #48

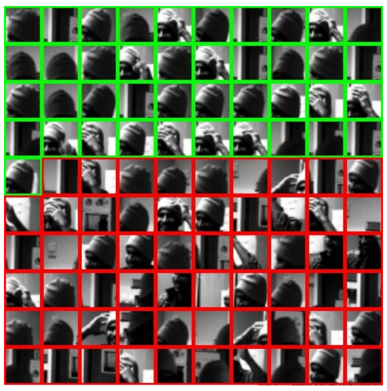
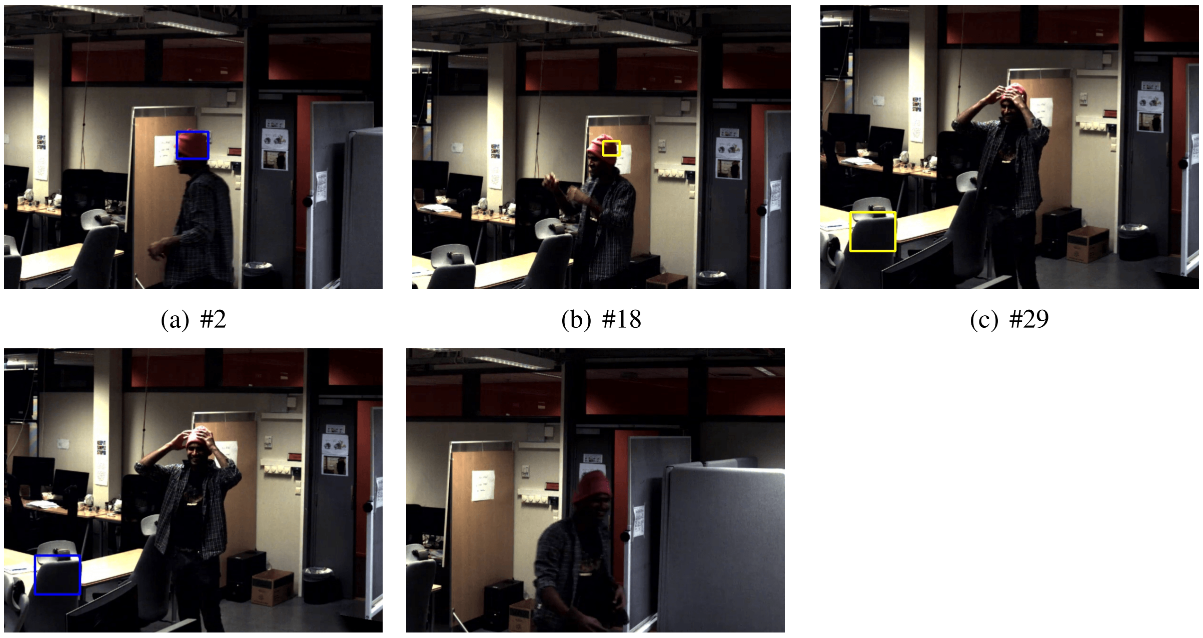

(c) Support vectors at frame #97

Figure 4.3: Images show how the support vectors are polluted by the background. Positive samples are marked with green borders, while negative samples are marked red.

4.3.2 Tracking-Learning-Detection

We again provide a timeline of the tracking, now shown in figure 4.4. As we can see, TLD is able to track the object for roughly 18 frames before shifting focus completely in frame #29. The tracker is not sure of its accuracy (depicted by a yellow rectangle), but establishes confidence in frame #35. The TLD tracker is using a specialized version of tracking-by-detection, which we described in detail in section 3.5.1. When the Median-Flow tracker fails and loses the object, the detector kicks in to correct the tracker. The detector and classifier are initialized with labeled samples from the first frame, and redetection in later frames is thus heavily dependent on the appearance model that was generated from these samples. As we can see, TLD computes the region marked in frame #35 as sufficiently similar to the region it was initialized with in frame #2 (frame #1 is not shown, but there differences are minimal). Furthermore, the redetection functionality in TLD is there to redetect objects that disappear from the video completely, and this is done without regards to proximity, i.e. new object candidates are freely selected from all pixels in the image. Consequently, in our case, it leads to a misdetection of a region several hundreds of pixels away from the previous position of the object. When reaching frame #92, the tracking is already completely lost.

This results in a total of *184* lost frames out of the total sequence length of *200*, and an inaccurate retrieval rate of only *8%*. It is also by far the slowest of the trackers, with *376* ms on average per frame. This is due to its constant computationally heavy attempts of redetecting the object for the remainder of the sequence.


(a) #2 (b) #18 (c) #29

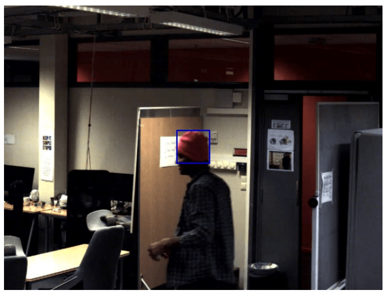
(d) #35 (e) #92

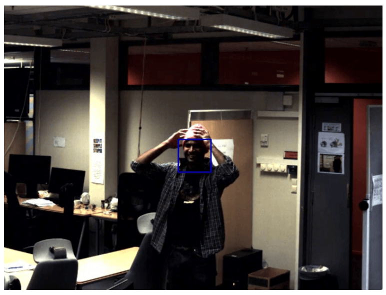
Figure 4.4: The rectangle marks the TLD tracking. Yellow and blue rectangle represent low and high confidence, respectively.

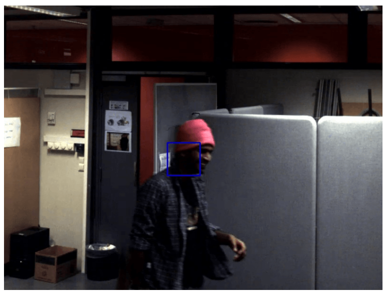
4.3.3 Real Time Compressive Tracker

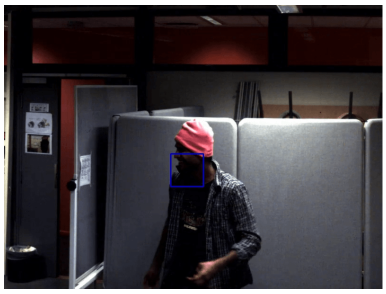
As we can see from figure 4.5, RTCT is able to keep its focus on the target for quite a few frames, but at the cost of precision. In frame #35, the occluding hands have made tracker's focus shift to the face. This distance to the ground truth is outside our permitted range for accuracy, resulting in a lost frame, and the tracker is not able to recover from this error in the following frames. Looking at the performance summary in table 4.1, only the first *4* frames are within our permitted range of drift from the ground truth, and the longest continuous amount of frames tracked is *8*. Just like the previous two trackers, it is unable to redetect the object properly when tracking is lost. Or more specifically, in Struck, the object representation drifts, while TLD simply loses the object. RTCT has a different problem, in that although its feature extraction is very fast, it is highly inaccurate in this case. This stems from the lack of features present in the beanie, which really has only two distinctive features 1) the strong color, and 2) the shape. The beanie's surface area is smooth and plain, in contrast to, say, human faces which naturally have lots of appearance variations such as darker regions around the eyes, cheek lines, mouth, nose and facial hair. The lack of features leads the tracker away from the beanie and towards the edges of the bounding box, where there are stronger features. The effect of this is, as mentioned, seen in frame #35. The feature extraction misleads the classifier and it starts tracking of the wrong region. This process repeats throughout the sequence, causing the tracking to jump around in the image, as seen in the remaining frame samples #102, #127, #144 and #185.

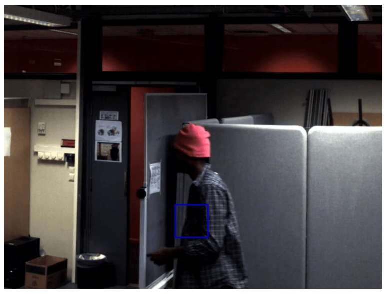
Although the tracker does keep focus on other irrelevant pixels part of the object, its high inaccuracy leads to lost frames and a very low retrieval rate (1/3 of Struck, albeit twice as high as TLD). In section 3.5.3, we detailed the inner workings of RTCT. The advantage of compressive sensing and the sparsity of the random measurement matrix becomes clear when we look at the execution time. Each frame is processed rougly *300* ms faster on average than Struck and TLD, which is a substantial speed difference. An average processing time per frame of only *47* ms puts it very close to our real-time requirement, without a single code optimization.


(a) #1


(b) #35


(c) #102


(d) #127


(e) #144

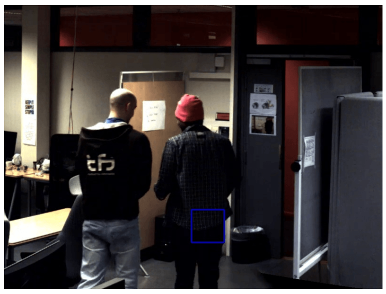
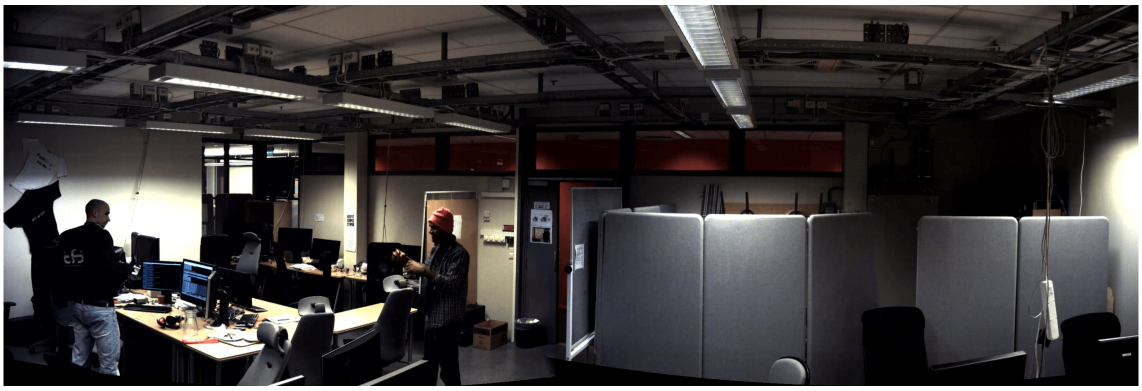

(f) #185

Figure 4.5: The blue rectangle marks the RTCT tracking

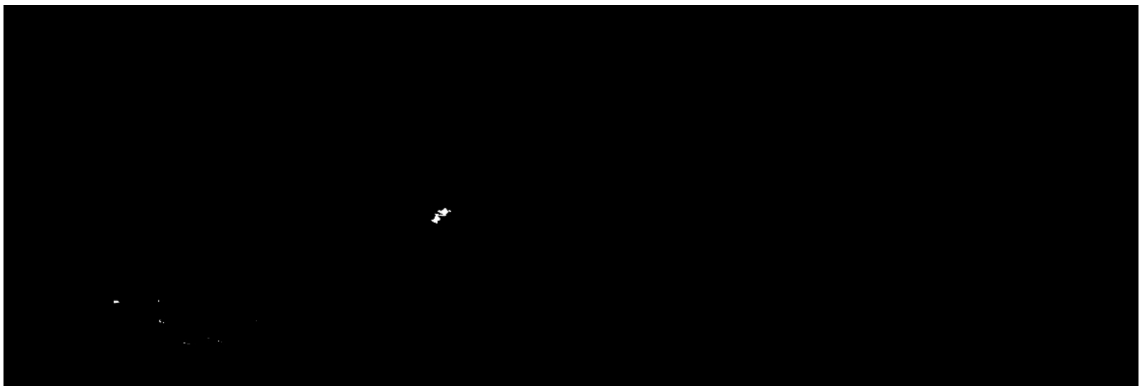
4.3.4 Color Tracker

After testing Struck, TLD and RTCT, we wanted to explore how well we could match these state-of-the-art trackers using our own implementations. Adaptive trackers are meant to work on arbitrary objects, i.e. they have to take into account the lack of a priori information about both the object and its environment. However, exploiting such knowledge can help a tracker tremendously, as fewer assumptions have to be made. In this case study, we specifically chose a strongly colored beanie when we set up our experiments, for this exact reason.

As mentioned in section 4.2, this sequence is an excellent example of where using color as a feature can work well. There are barely any other areas in the image but the beanie which have the same color, and this is the key assumption we will base our tracking on. We will start by first providing a full presentation of our color tracking algorithm, and then go on to present the performance results in section 4.3.4, as has been done with the Struck, TLD and RTCT.



(a) Original image



(b) Thresholded image

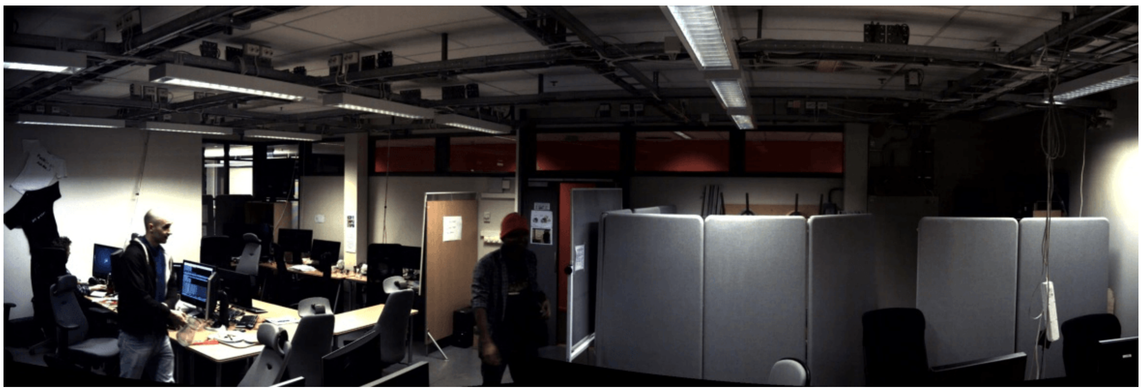
Figure 4.6: A naive color thresholding operation on pink. White pixels in 4.6(b) means the original are within the defined range of pink

Figure 4.6 shows the initial naive attempt of running a simple color thresholding filter on pink, i.e. removing any pixels that are not within a range of pink. The largest continuous white area in the thresholded image is clearly the beanie, but there is still noise in the image (seen on the lower left side of the image). This means a few other small areas are also within our range of pink. Despite the noise, this still provides us with a very usable result and serves as an excellent basis for isolating the beanie and tracking it. Although the frame used in figure 4.6(a) shows promising results, most color bands are very sensitive to illumination changes. Therefore, in video sequences where the lighting conditions change drastically, relying solely on this approach is not a good idea. Figure 4.7 demonstrates the issue with two examples frames that have no pixels within the threshold, and we are left with a black binary output image in both cases (not shown). In the next section, we look at some possible solutions to the problem, without moving away from using color filtering as detection.

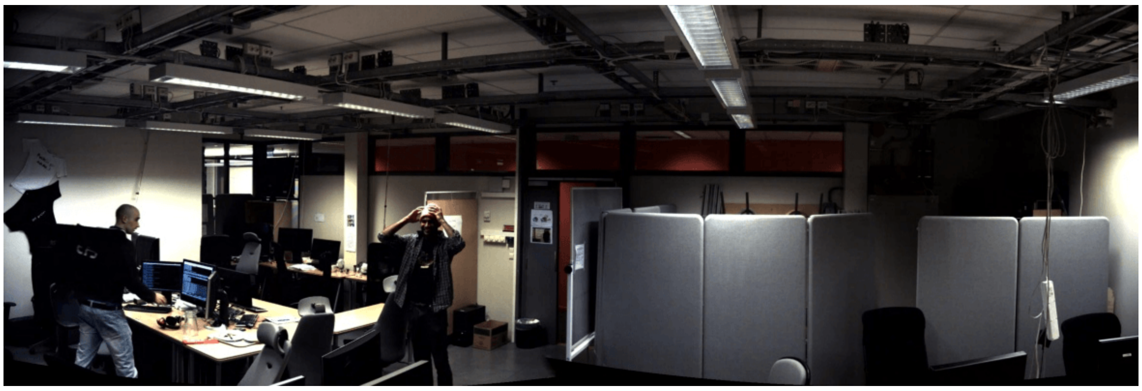
Handling failing frames

Since the normal naive thresholding operation fails during difficult lighting conditions (either very bright or very dark), an additional mechanism is needed to handle those cases. We start by addressing the case where there is insufficient lighting.

There are several possible solutions, and one could be to expand the range of the color filter to include the "darkened" version of the object color. However, this leads to a very high amount of false positives due to the darkened pink being too close to other regions in the image. These



(a) The beanie is too dark for the thresholding operation (below threshold)



(b) The beanie is too bright for the thresholding operation (above threshold)

Figure 4.7: Beanie color is above/below the threshold in the color filter. Running these through a pink filter would leave a black image in both cases.

false positives merge into the true positives and the object is no longer isolated from the rest of the pixels, making the tracking impossible. Another solution is to boost the brightness of the image when the tracking is lost, in an attempt to reacquire the object. This will push the object's colors back up into the defined range, and tracking resumes. So as to reduce workload and prune the search space (image pixel area), we only perform the boost on a region of the image. This region of interest (ROI) is defined as a small area of *200x350* pixels surrounding the latest known position of the object. There is more horizontal than vertical movement in the video, and the region is thus made wider than it is high. Whenever the tracking is lost, we can boost the ROI, and try reacquiring the object. If this fails, the area is either too bright to begin with (meaning the object's pixels are above the defined range), it has fallen outside the ROI or the object is fully occluded.

Handling overexposed regions is much harder because the original pixel color cannot be inferred and retrieved from the object's, now white, pixels. Reducing the brightness would only change the colors of the normally or underexposed regions, which the object is not a part of. As object detection in overexposed frames is impossible using this simple brightness alteration, introducing more complexity is unavoidable if one were to cover these cases and maintain tracking.

One way to handle this is by utilizing the natural frame delay in the pipeline, which is caused during startup when the pipeline must be "filled" with images before it can operate in real time. Another way is to simply delay it further manually, providing real-time tracking, but a few frames behind the live video. Either way this gives the tracker a head start. When an object disappears, and then reappears, the tracker can redetect the object and interpolate the missing points in between the two known positions. Another, viable, alternative, is using the highly popular Kalman filter [62][2]. The Kalman filter has a wide array of possible uses, but can in object tracking for example be used to guide the system through occlusions. The filter can make predictions on the future locations of the object by analyzing its previous states, such as position, velocity and direction, and estimate the next state (new object location).

Color Tracker algorithm

We present here a complete algorithm for the tracking of the beanie, using color as a feature including the additions described in section 4.3.4. We will go through each part of the algorithm, and delve into the details of the implementation to see how these work together to form a successfull tracking.

Converting from YUV to RGB

In order to make OpenCV able to manipulate the pixels, a conversion from YUV422 to RGB is needed. This is done through a very simple matrix operation and although it takes roughly *300ms* executing on the CPU, this operation is the perfect task for a GPU and will easily be within our real-time constraint if ported to a GPU. The conversion is described by the following equation:

$$\begin{aligned}R &= 1.164(Y-16) + 1.596(V-128)\\ G &= 1.164(Y-16) - 0.813(V-128) - 0.391(U-128)\\ B &= 1.164(Y-16) + 2.018(U-128)\end{aligned} \tag{4.1}$$

where Y is the luminance component, and U and V are the two chroma components.

Thresholding and detection

We now have the correct pixel format, and the next step is the thresholding operation that will help us detect the object. Our first and most important task is defining a proper color range for pink, but another problem arises when the lighting conditions change. RGB was designed for computers, not to model human perception, and thus inherits some unwanted properties. In the RGB color space, changes in lighting will affect more than one component, making it difficult to create a good range. In YUV, our original format, this is possible, but OpenCV is unable to work with this colorspace. Instead we convert our colorspace into HSV [63]. HSV share an important property with YUV, where the pixel intensity (Y in YUV, and H in HSV) is separated from the pixel's color information. This gives us the opportunity to define the color range with Hue (and partly Saturation), while the Value independently can have a much larger range, covering cases with both under- and overexposed regions.

The color filter serves as our detector, and is the most important component in the tracking. The beanie is colored pink, and we define the range as a set of two HSV color vectors [H,S,V], where min is [170,100,100] and max is [190,255,255]. The color thresholding operation is trivial, and is represented by the following equation:

$$\begin{aligned}dst(i) = H_l <&= H_s <= H_u \wedge\\ S_l <&= S_s <= S_u \wedge\\ V_l <&= V_s <= V_u\end{aligned} \tag{4.2}$$

[2]Citation is an introduction to the Kalman filter, as we were unable to locate the original paper from 1960.

where *dst* is the destination image, and *i* the pixel index. H_l, S_l and V_l represent the lower bound for the pixel's color value, while H_u, S_u and V_u represent the upper bound. H_s, S_s and V_s are Hue, Saturation and Value for the source image pixel.

This is run on every pixel in the source frame, and if the pixels' values fall in between the lower and upper values for each channel, the corresponding pixel in *dst* is set to 255 (i.e., white), and 0 (i.e., black) if it is not. This leaves a binary image where all pixels with colors within the permitted color range are white, and all other pixels are black, ideally resulting in an image containing a beanie-shaped white region on a black background. However, due to the simplicity of the filter, and because there is a high probability that some pixels not part of the beanie will fall within the permitted range, the image will contain noise (see figure 4.6(b)), and the beanie itself will also be segmented due to internal color variations (for example, some areas might be covered by shadows while others are not).

Processing the binary image

To remove the noise introduced by the thresholding operation and to extract the best candidate for the beanie, an optimized version of an algorithm called *Connected Components* [64] (CC) is run. The algorithm scans through the binary image and assigns a group to each pixel, called a label, where each label represents a continuous region of equal values. This "connects" all the continuous regions of white pixels in our binary image into blobs. We loop through all label groups to locate the one with the highest number of members, meaning the largest blob in the image. As long as this blob's pixel area is bigger than our set minimum of *100* pixels, the beanie is considered found (we assume that no other regions of the image with the same colors are equal to or larger in size than the beanie). For each pixel in the blob we store a coordinate pair (X,Y). To get an approximation for the blob's center, we compute their average, and this is the final output of the tracker.

The CC algorithm is quite costly and to achieve the fastest possible execution time, we reduce the workload of the costly CC algorithm by only labeling the ROI around the object's latest known position, as we did with the brightness increase. The full image is processed only on the very first frame, to detect the initial location of the object and create a ROI. Compared to processing the full image, the workload using the ROI is equal to only $\frac{200\times350}{2936\times986\times100} \times 100 = 2.42\%$

A pseudocode of the full Color Tracker algorithm can be found in the appendix, section A.1.

Performance

We have now detailed how the Color Tracker works, and we move on to analyze its performance. Once again, we provide a timeline for the tracking, shown in figures 4.8, 4.9 and 4.10. The green rectangle illustrates the search area for the blob-detection heuristic, which is detailed in the *getBlobs* function, part of the pseudocode. The region used for the brightness boost is not shown in the images. The red circle is the output of the tracker, i.e. where the algorithm assumes the object to be in the given frames. Images labeled "<frame number> - original" show the original image, with overlays of the ROI and the tracker output. Images labeled "<frame number> - threshold" show the output of the filtering operation for that given frame. Figure 4.9 illustrates the case where the beanie is too dark to be caught by the filtering, resulting in a black image. The brightness of the ROI is then boosted, and the tracking resumes.

As we can see, the tracker is able to sustain through the occlusions at the start of the sequence (frame #25), providing *68* frames of continuous tracking. In figure 4.9, frame #88 demonstrates the case where the target is underexposed, falling outside the range of our color filter and resulting in a black binary image. To counteract the illumination change, the algorithm boosts brightness of the ROI around the latest known position, and successfully reacquires the beanie. With an average speed of *77ms* per frame this approach is able to accurately track *171* of the 200 frames.

Despite *21* inaccurate frames, the tracker is able to stay on target and produce a retrieval rate of *96%*. This demonstrates several of the advantages of color tracking. Firstly, detection by color filtering is highly insensitive to occlusions, since the colors of the object is still present even when large parts of the object are not. Also, it works great as a detector as it matters little if the object is lost for a few frames. It can simply be redetected the instant the object is back in the correct color range. This is in great contrast to Struck, TLD and RTCT, which all are unable to reacquire the object if it is ever lost[3]. Consequently, the Color Tracker would likely be able to sustain tracking far beyond the 200 frames in this sequence, unlike the others, given that the colors of the beanie stay within the range present in the first 200 frames. Despite these advantages, tracking by color also has several limitations. First of all, we need to tune our color filter offline to match the object's colors on each sequence. Second, the object's colors might not be unique to the scene, as we will see in chapter 5. These limitations narrows the applicability of pure color tracking, but it is often combined with other features, to ensure higher robustness and cover more use cases.

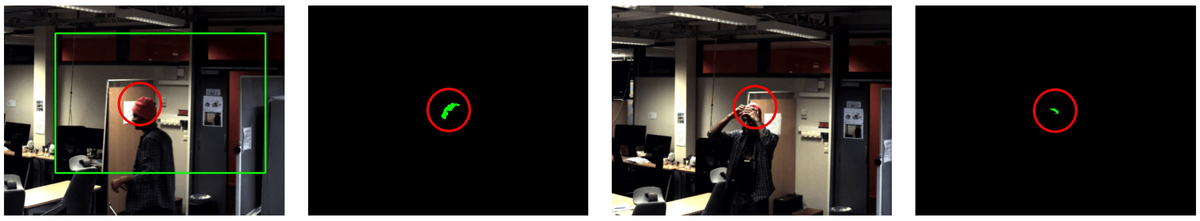

(a) #2 - original (b) #2 - threshold (c) #25 - original (d) #25 - threshold

Figure 4.8: Images show tracker output and binary result from the color mask

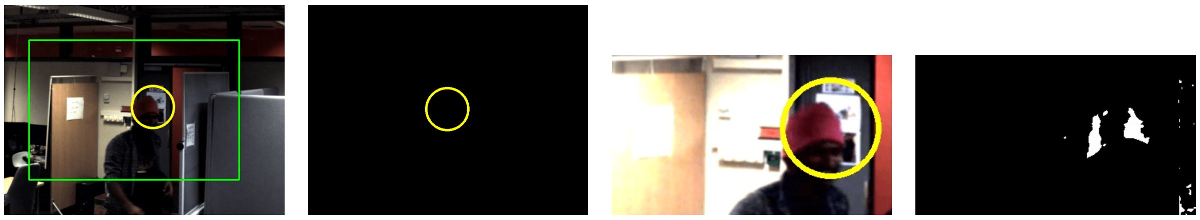

(a) #88 - Original (b) #88 - Binary (c) #88 - ROI with brightness boost (d) #88 - ROI binary

Figure 4.9: Example of when the image gets too dark. The brightness in the ROI is boosted and the object is reacquired

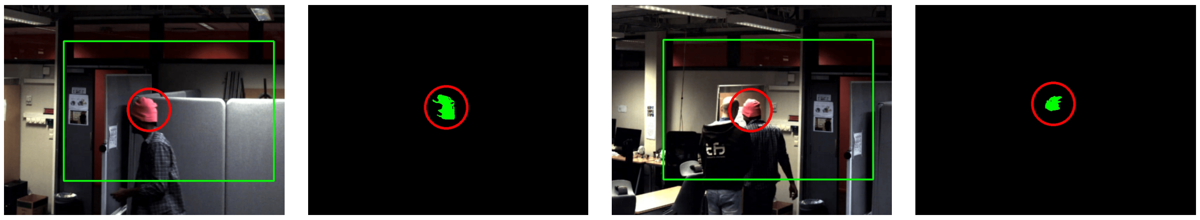

(a) #143 - Original (b) #143 - Binary (c) #198 - Original (d) #198 - Binary

Figure 4.10: An example where strong color results in a much larger binary blob

[3]This is not true for all sequences, but it is true for the sequence at hand.

4.3.5 Optical Flow Tracker

Another popular approach in tracking is exploiting the fact that an object is moving relative to the scene or the observer, through an approach called optical flow. It is an algorithm which simply put computes two-dimensional displacement vectors containing direction and length, modeling the motion from one image to the next (demonstrated in figure 4.11). Considering the object of interest is one of two objects moving in the image, we can make use of optical flow to facilitate our tracking. A common approach is first to locate features in the region occupied by the object, and then determine where those features, or points, will be in the next image by using optical flow. We wish to investigate how such an approach can work in the enviroment in our sequence. Note that this is not tracking-by-detection, as in the previous approaches. Struck, TLD and RTCT search each frame of the sequence for the appearance model that they generated on the first frame (and updated in subsequent frames). Our Color Tracker simply detects the object by its colors, but does so on each frame as well, therefore also providing tracking-by-detection. In the Optical Flow tracker, however, the features are established in the very first frame, and then *only* tracked for the remainder of the sequence[4]. There is no redetection of these features/points, but instead their displacements across frames facilitate tracking.

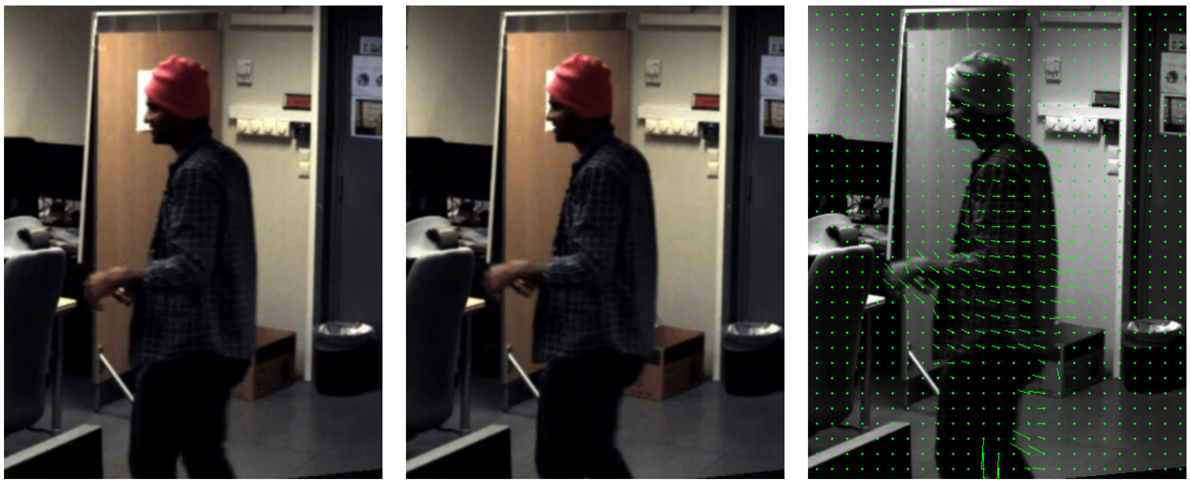

(a) Frame #4 (b) Frame #5 (c) Vector field

Figure 4.11: Last image shows the vector fields computed from the two previous images

Optical Flow Tracker algorithm

In order to locate the features of the beanie, we run an OpenCV algorithm based on [51] (GF), marking the most prominent corners in the region of the image that our object occupies. This can be seen in figure 4.12, where the yellow rectange marks the image region that is searched by the GF algorithm. Our object representation is now point-based (shown in blue in the figure), as described in section 3.1.1, this *goodFeaturesToTrack* initializes our point-based tracking algorithm. After points have been established for the first frame, we use OpenCV's version of the highly acclaimed optical flow algorithm by Lucas-Kanade [40] to track these points in the subsequent frames. As input, the algorithm is given the previous frame (currently our first frame) and its points, along with the next frame. The algorithm then computes the displacements of the points and thus their location in the next frame. A pseudocode for our Optical Flow Tracker algorithm is shown in figure 1.

[4]We could "refresh" the features by detecting new ones after a given amount of frames, but later in the tracking we can not confidently say that the object is the sole occupant of the new bounding box, as we could in the first frame. There is little value in detecting features in the background.

Algorithm 1 Optical Flow Tracker

```
1:  while there are frames available do
2:      if first frame then
3:          goodFeaturesToTrack()
4:      else if features found in frame n-1 then
5:          opticalFlow(previous frame, previous points, current frame)
6:      end if
7:
8:      if points were found in current frame then
9:          object_pos ← compute average of points
10:         object_pos ← remove drift points(object_pos)
11:     end if
12: end while
```

Performance

As seen in the timeline in figure 4.12, features are established in frame #2. The yellow rectangle denotes the ROI we have created, in order to avoid detecting features in other parts of the image. This is very important due to the huge number of edges and gradients created by all the straight lines present in the image. After the feature detection, we compute the centroid of all the points by averaging out their coordinates. Any point more than 35 pixels away from the centroid is removed, to filter out false positives and avoid drift issues.

It is easy to see from the images which edges and high contrast areas are used for tracking, such as in frame #57 and #75, and when the hands obstruct the beanie in frame #36, the weakness of this approach becomes apparent. Since these features are not refreshed during the course of the tracking and only computed in the initial frame, the tracking immediately shifts to the occluding object (new edges are introduced and mistaken for the original features). A similar mistake happens in the frames after #75, where the person is walking past a strong edge in the background. The extremely high-contrast area (white next to black) confuses the algorithm, and we see the result in frame #138. Altough extremely fast with an average execution time of only *19* milliseconds (40% of the second fastest tracker), this leads to a fragile and non-robust tracking, particularly vulnerable to occlusions. Suprisingly, it is still able to track longer than both TLD and RTCT. It is able to continuously track the first *50* frames, and *57* of the 200 frames in total, resulting in an accurate retrieval rate of *28.5%*.

4.3.6 Background Subtraction Tracker

Further exploring the value of locating the movement in an image, we investigate a simple tracking algorithm using background subtraction. This can help greatly in eliminating uninteresting regions in the image, and ideally leave only the object of interest. Confidently extracting features then becomes not only much easier due to fewer elements, but also faster as much fewer pixels need to be processed. The algorithm starts by running an OpenCV implementation of the adaptive background substraction algorithm (BGS) described in [71]. This algorithm assumes that each pixel's value can be modeled using a Gaussian mixture model. First, a model of the background is initialized, and then updated to adapt to any changes in the next frames.

This removes the uninteresting noise in the background, as shown in figure 4.13, and produces a binary image of the detected foreground. After obtaining the binary image, the BGS tracking algorithm processes the image in the same way as the Color Tracker 4.3.4. However, the Color Tracker assumes that the biggest blob in the binary image is the beanie. This is a fair and useful

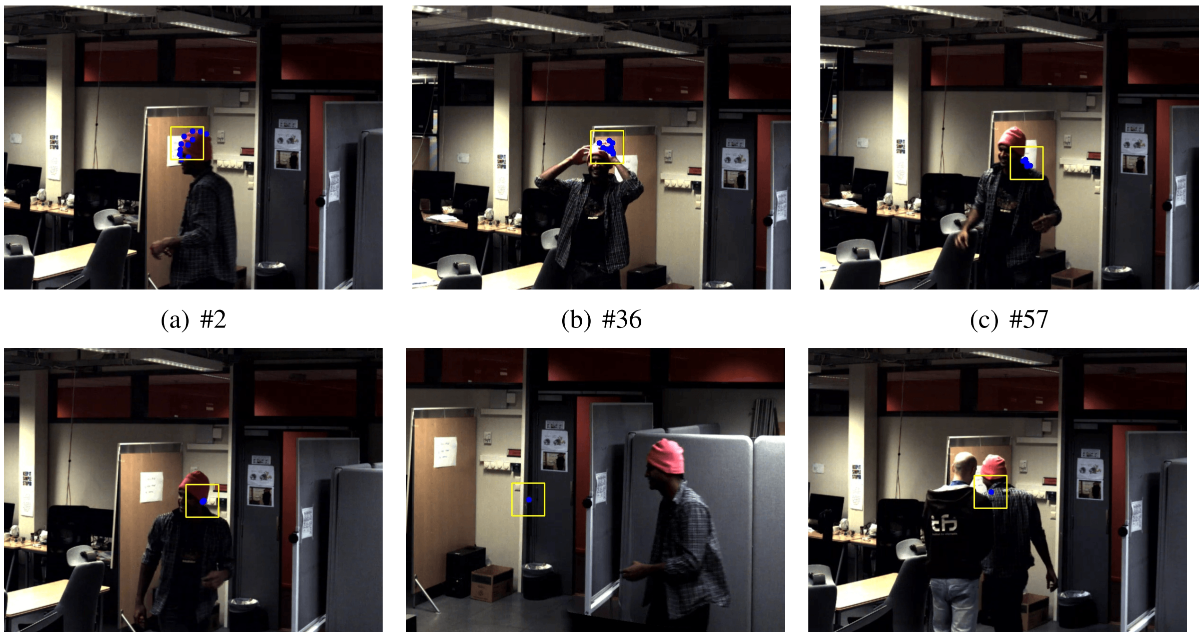

(a) #2 (b) #36 (c) #57
(d) #75 (e) #138 (f) #197

Figure 4.12: The yellow rectangle marks the search area, and the blue dots represent the features found in the area.

assumption in this case study. The base assumption of the Background Subtraction Tracker (BGST) algorithm is now that the detected foreground is what we want to track, and also, that the biggest blob in the foreground is in fact the object of interest. The latter does not hold for this sequence, as there are two people moving largely in the same area, producing two large blobs in binary image from the background subtraction output. Furthermore, if the correct person was indeed tracked, distinguishing the body from the beanie is non-trivial, giving us the ability to track the individual, but with high inaccuracy with regards to the beanie. This algorithm was originally made for the case study in chapter 5, but is included here for completeness. See the appendix A.2 for a psuedocode.

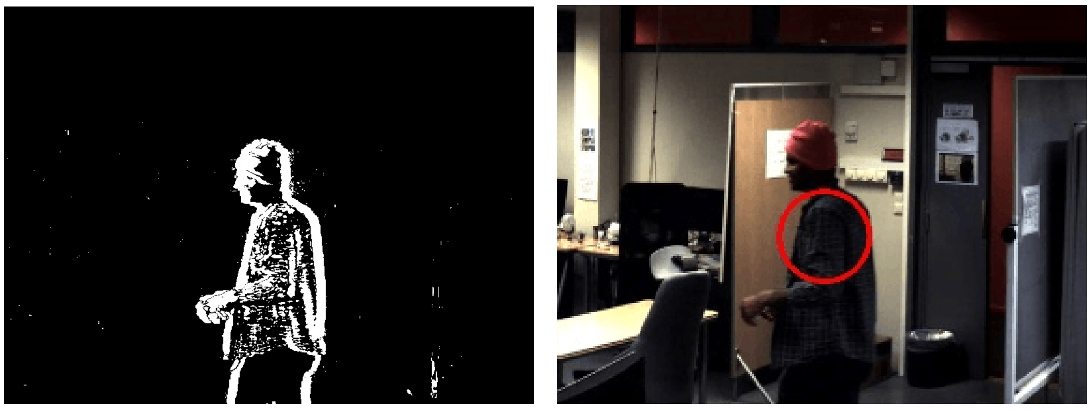

(a) #2 - BGS output (b) #2 - Tracker output

Figure 4.13: Example of background subtraction and the corresponding tracker output. We can see the bad effect of the blob coordinate averaging, as the center of the blob is not the beanie.

4.4 Performance

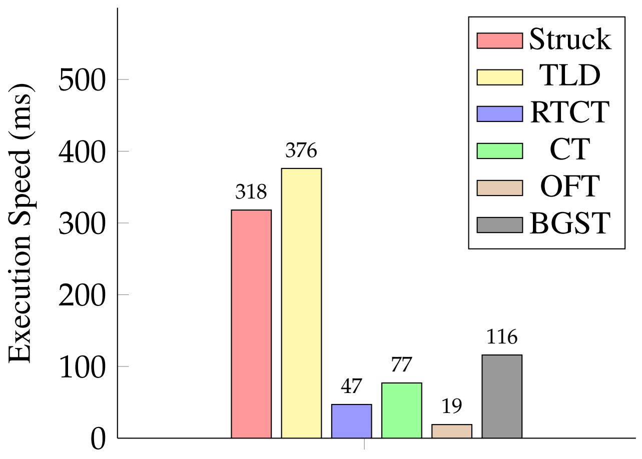
Table 4.1 shows a performance summary of running all trackers through the sequence. *Average speed* is measured in milliseconds and depicts the average speed in milliseconds a tracker used per frame. For defintions of *accurate*, *inaccurate* and *lost* frames, we refer to equation 3.3. The *accurate* and *inaccurate* retrieval rate is defined in equations 3.1 and 3.2, respectively. By using these metrics, we can not only measure how many frames were tracked in total, but also extract information about how accurately the object was tracked during those frames. Also, we introduce abbreviations for our own tracker implementations that will be used throughout the rest of the thesis, namely 1) *CT* = Color Tracker, 2) *OFT* = Optical Flow Tracker and, 3) *BGST* = Background Subtraction Tracker.

	Struck	TLD	RTCT	CT	OFT	BGST
Average speed (ms)	318	376	47	77	19	116
Accurate frames	85	8	4	171	49	3
Inaccurate frames	4	8	25	21	8	8
Continuous frames	89	16	8	68	50	5
Lost frames	111	184	171	8	143	189
Accurate retrieval rate	42.5%	4%	2%	85.5%	24.5%	1.5%
Inaccurate retrieval rate	44.5%	8%	14.5%	96%	28.5%	5.5%

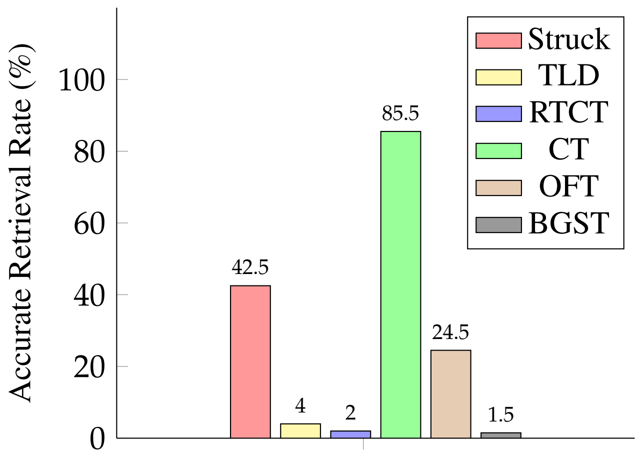
Table 4.1: Green color indicate the best performance while blue color indicate the second best.

The strength of the off-the-shelf trackers (Struck, TLD and RTCT) is that they do not need to know anything about the object a priori, as they were designed to. On the other hand, this is also their weakness, as they cannot leverage the uniqueness of the specific use case at hand and must adapt on the fly using online learning. Although Struck is a robust tracker with its Structured Output Prediction (more on this in later case studies), it is still only able to provide a *44.5%* inaccurate retrieval rate, executing at *318* ms per frame. Performance plots is given in figure 4.14, and we can see that Struck is the most accurate of all trackers except the Color Tracker, but also the second slowest. TLD did not do well overall in this sequence, and coming in as the slowest of all trackers with an average execution speed per frame of *376* ms. It also is in the bottom three of all performance measures shown in the plot. RTCT is slightly more accurate than TLD in both accuracy measures, but is way more efficient and runs at only *47* ms per frame on average. However, it is still highly inaccurate, and able to track only *3* more continuous frames than the BGST, which was run merely for demonstrating its failure. The redetection in TLD and RTCT makes no use of the prevously tracked position, leading to more severe errors where tracking is resumed after a redetection, but far away from the original object. This effect is not as dramatic in RTCT as in TLD, but still enough to disrupt the tracking and produces bad results.

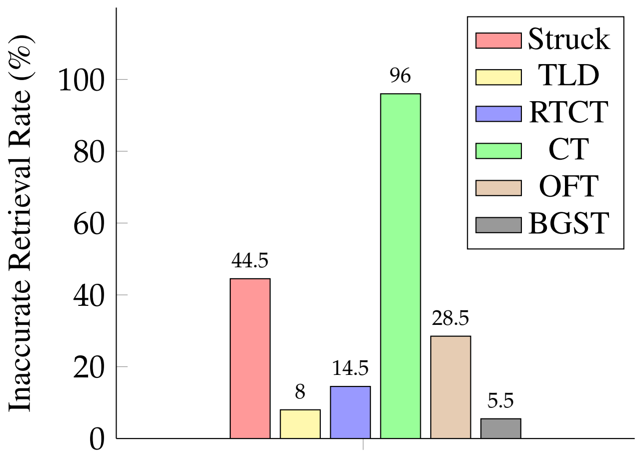
The low computational complexity of optical flow allowed the Optical Flow Tracker to perform far faster that any of the other algorithms (comfortably operating at real-time), but was not able to handle the background clutter properly. The OFT is simplistic and fast, and even though it only provides pure tracking, not by detection, its retrieval rate of *28.5%* is well over both TLD's *8%* and RTCT's *14.5%*. However, due to its lack of robustness, it is surpassed by the Color Tracker in acccuracy, which is also only slightly slower. Tracking by color is, as mentioned, not only robust to occlusions (as discussed in section 4.3.4), but the very low complexity of the thresholding operation allows a fast execution time. The Color Tracker has the best overall performance in this case study and provides a very high retrieval rate of *96%*, along with being the second fastest of the 6 trackers, clocking in at *77* ms on average per frame.



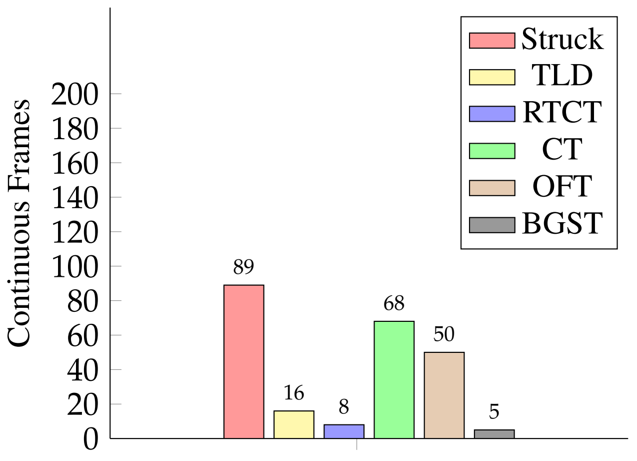
(a) Execution speed of trackers



(b) Accurate retrieval rate



(c) Inaccurate retrieval rate



(d) Number of Continuous Frames

Figure 4.14: Performance plots for Execution Speed, Accurate Retrieval Rate, Inaccurate Retrieval Rate and Continuous Frames for each tracker.

4.5 Summary

In this first case study, we have taken our first step towards object tracking in Bagadus. By setting up a simpler scenario where we could limit some of the hurdles and challenges, we have a proper test setup to explore ideas and the current state-of-the-art in tracking, while still operating within the boundaries of such a system. We have tested and evaluated the performance of 3 current state-of-the-art trackers and compared them with the performance of 3 of our own tracker implementations. These were outlined and detailed in the chapter with complete tracking algorithms, and included by tracking by color, tracking using background information and an optical flow tracker. We learned that the lack of constrasts and features in the beanie was difficult to handle for the adaptive state-of-the-art trackers, as they could not leverage the a priori information we were able to exploit in our own implementations. A simple color tracker that used the strong color of the beanie, coupled with a brightness boost alteration, outperformed the other algorithms by far and was able to stay on target for *192* of the 200 frames in this sequence. We see that exploiting information about the scene is crucial and can greatly increase the retrieval rate of a tracker, despite its simplicity. However, we also discussed that such specific implementations are limited and must be tuned on each sequence to work properly, unlike the adaptive trackers (TLD, Struck and RTCT).

In the next chapter, we test the trackers in a different environment using more cameras and an outdoor setting. The trackers are presented with a greatly increased workload, different lighting conditions and a great deal of occlusions that together will push the trackers' robustness and runtime efficiency even further.

Chapter 5

Case study 2 - Pretty Woman

5.1 Introduction

Moving further towards our full setup in Bagadus, we explore another sequence. The pipeline was installed in an outdoor environment at the University of Oslo, Norway. The number of cameras used was increased from 2 to 3, producing images with a resolution of 2736×1740 pixels. Up from 2936×986 pixels in the previous case study's sequence, this results in a workload increase for the tracker by roughly 60%. The framerate remains at *25* FPS, but the length of the sequence is increased to 315 frames. In contrast to the indoor setup in the previous case study, this sequence was recorded outside at midday during early summer. Figure 5.1 shows the full image overlooking the scene, where our goal is to track the woman (center left) walking up the stairs, and past the blue tent (lower right) to the right edge of the screen. We start by highlighting some differences compared to the previous case study, and how those might affect the basis for the tracking algorithms. We then go on to evaluate the algorithms in the same manner as chapter 4. Finally, their performance is reviewed and we summarize the findings.

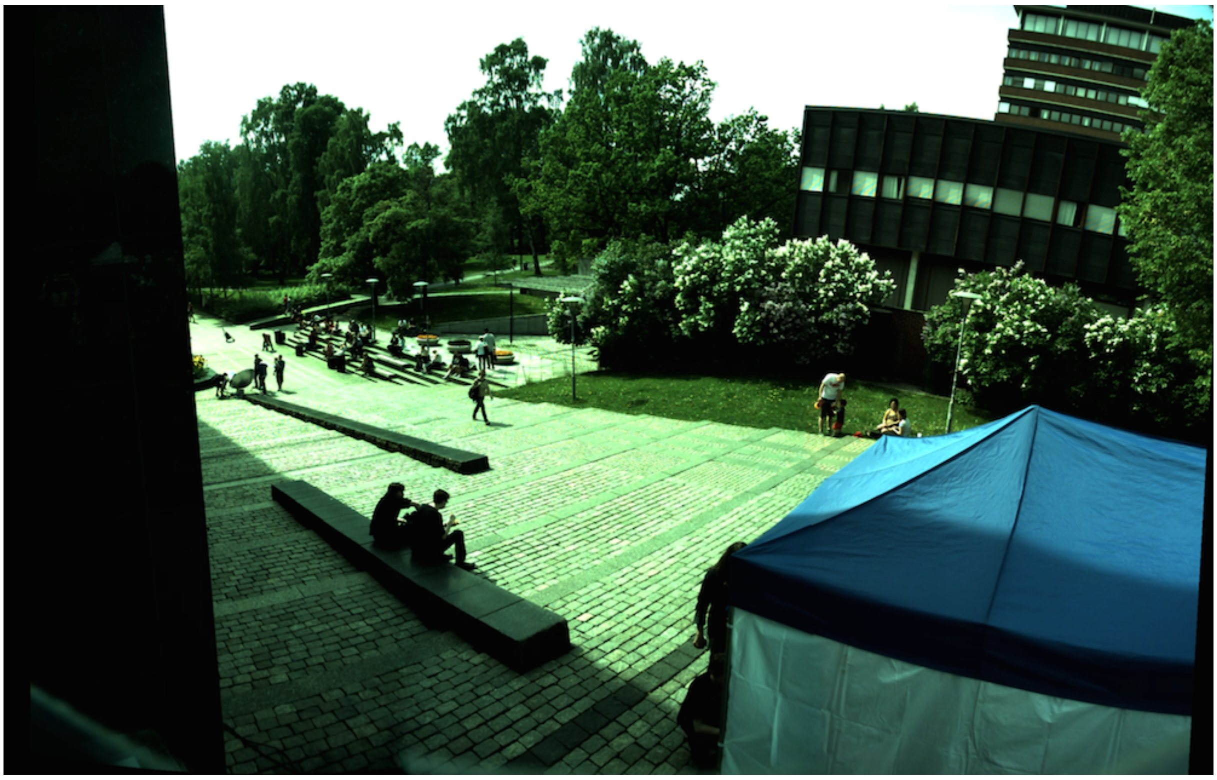


Figure 5.1: Full view of the scene (panorama of 3 cameras)

5.2 Simplifications

- *Lighting conditions:* The lighting conditions are much more stable, and we avoid the large over- and underexposures present in the previous sequence. Even though the object does not move out of the sun and the stable lighting, we can see potentially difficult conditions arising when recording video outdoors, such as strong shadows.

- *Movement:* There is more overall more movement in the image, but the object is, however, fairly isolated. It moves freely with no occlusions for the first *270* of the in total *315* frames, very similar to how a ball would move during passes or shots. Towards the end of the sequence, there are very heavy occlusions as the object mixes with other people in the scene, while also gradually moving behind the tent.

- *Shape:* Although the object in both case studies are of the same size (roughly 5000 pixels), the beanie in the previous case study had a fairly static appearance, apart from what was induced by lighting changes. Now, both the object's appearance and shape continually change throughout the sequence.

- *Features:* There are no longer any strong colors to aid the tracking, but there are high internal contrasts and other features in the object. These features makes the object much more suitable for trackers with adaptive appearance models, such as the 3 off-the-shelf test trackers.

- *Accuracy:* For this sequence, we also wanted to further restrain the allowed drift from the ground truth, and take a deeper look into the accuracy of the trackers. To prepare for the high accuracy needed in ball tracking, we also included measurements where we lowered the threshold for an accurate frame from *25* to *15* pixels, and inaccurate frames from *35* to *25* pixels. The ground truth is not pixel perfect either, so we still allow some leniency. The old accuracy will be referred to as "*medium accuracy*", and the new increased accuracy is from now on called "*high accuracy*".

5.3 Evaluating and comparing tracking algorithms

5.3.1 Structured Output Prediction using Kernels

We now take a look a Struck's performance in this new sequence. Figure 5.2 illustrates the progression of the tracking throughout the sequence.

The feature rich object enables Struck to perform very well in this sequence. It is able to confidently and accurately track the object when it is moving unoccluded, and we can see the SVM's placement, or classification, of support vectors is highly accurate (figure 5.3). More surprisingly, is Struck's outstanding ability to handle the gradually increasing occlusions occurringin frame #293 and onward. When examining the support vectors, we see that the tent is gradually added to the object model with the object in the center, while many of the negative samples are small displacements from the object location.

Using the medium accuracy, Struck proves to be the most accurate tracker in this sequence, but an average speed of *466* ms per frame makes it the slowest of all 6 trackers. However, with *308* tracked frames, its accurate retrieval rate of *98.1%* outperforms the others comfortably in terms of accuracy. This is largely due its ability to handle the significant occlusions happening toward the end of the sequence (seen in frame #293 and #305). The use of structured output prediction makes Struck robust to occlusions, and frame #305 shown in figure 5.3 demonstrates how the positive and negative samples are updated, while still retaining samples of the original, earlier appearance.

Looking at the higher accuracy data, we see that *70* more frames are now classified as inaccurate, leading to a drop in inaccurate retrieval rate by *23.2%*, down to *74.9%*. However, it is still able to track the first *236* frames accurately, meaning the inaccuracy occurs during the background clutter and occlusions toward the end of the sequence. This is not unexpected, since the object's center is completely occluded in those cases. Only *1* more frame is considered lost with the new constraint. In total, Struck was able to track *310* frames using medium accuracy, and *308* with high accuracy, out of a total of *315* frames.

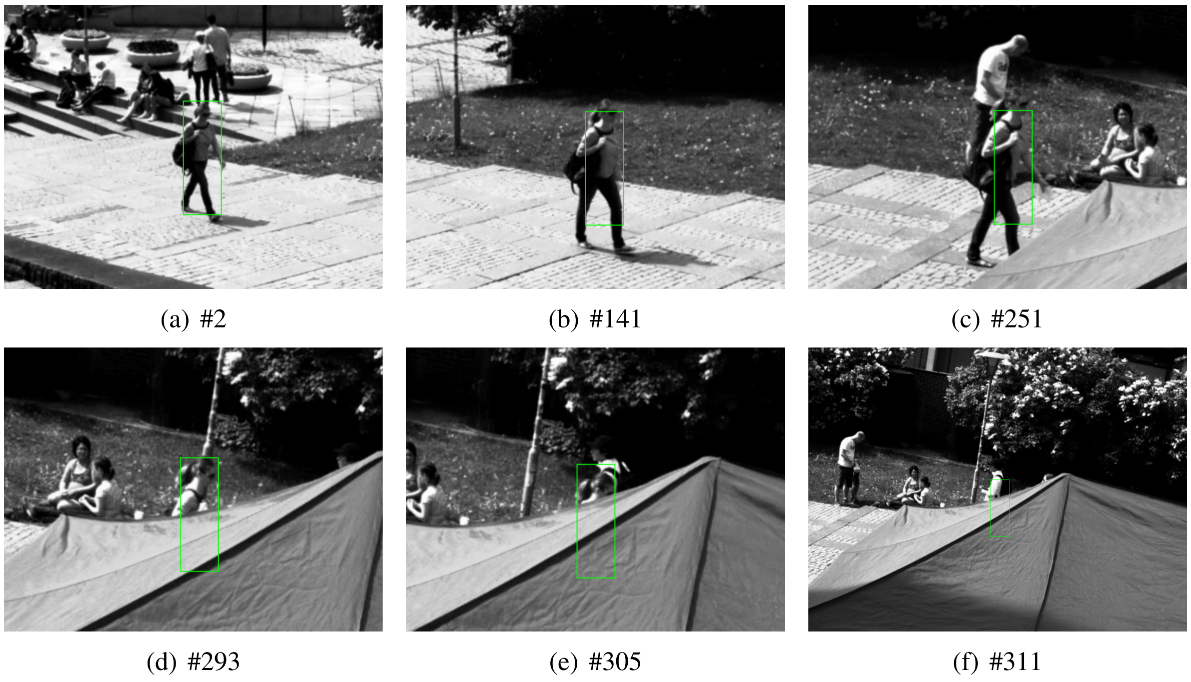

(a) #2 (b) #141 (c) #251
(d) #293 (e) #305 (f) #311

Figure 5.2: The green rectangle marks the Struck tracker output

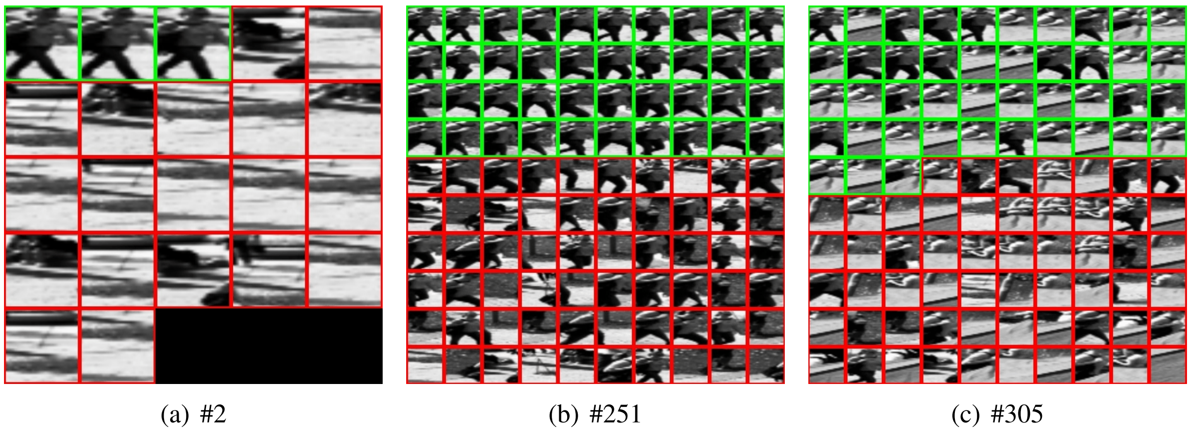

(a) #2 (b) #251 (c) #305

Figure 5.3: Support vectors showing the online appearance model update

5.3.2 Tracking-Learning-Detection

The progression of the tracking is shown in figure 5.4. We can see that TLD is able to keep the same high accuracy as Struck for the first *249* frames in the sequence, while using medium accuracy. Although with lower accuracy, tracking continues until frame #265 before it loses the object. Frame #267 shows the effect of TLD's high sensitivity to background clutter. The classic issue of drift in the appearance model occurs as the background is mistakenly classified as foreground, leading to failed tracking in the subsequent frames. These sensitivities also make TLD unable to handle the increasing occlusions towards the end of the sequence, and thus is only able to track frames in areas where the object moves freely, unoccluded and with minimal changes in the background. The result is an inaccurate retrieval rate of *84.1%* - lower than the accurate retrieval rate of Struck, RTCT and OFT. With an average computation time of *423* ms per frame, it is slightly faster than Struck and the second slowest of the 6 trackers.

Using the high accuracy constraint, we naturally see the same consequences as with Struck. TLD's accurate retrieval rate rate drops from *79%* to *63.3%*, and the new high accuracy inaccurate retrieval rate is now equal to the medium accuracy accurate retrieval rate.

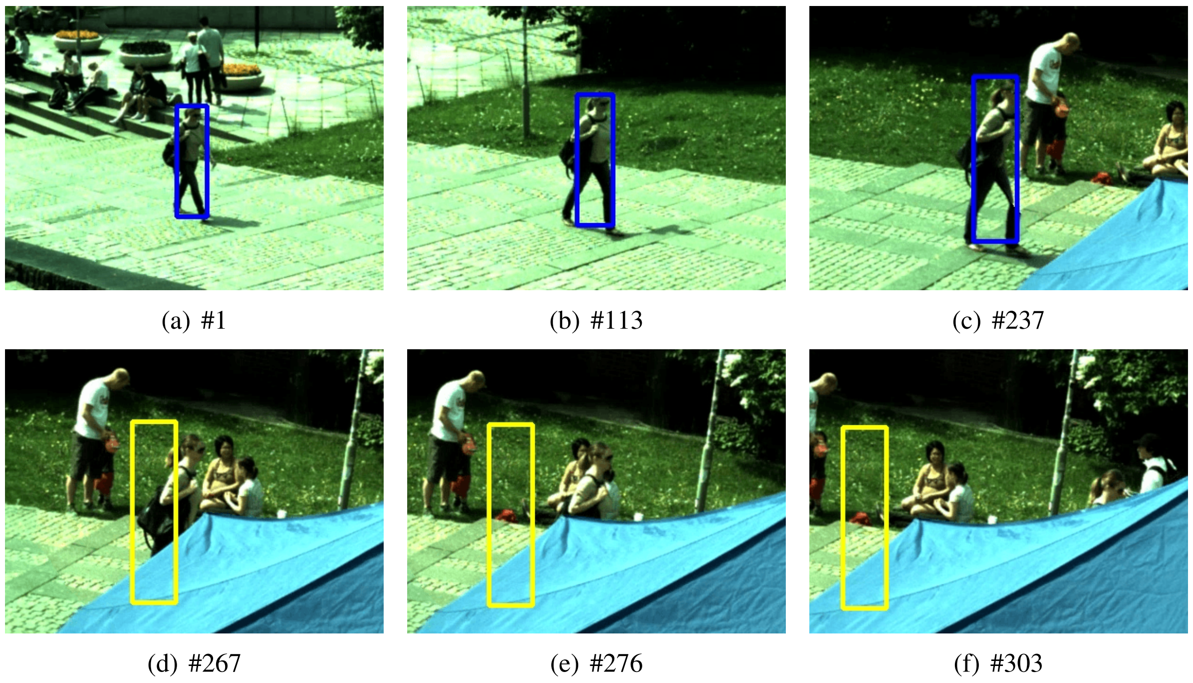
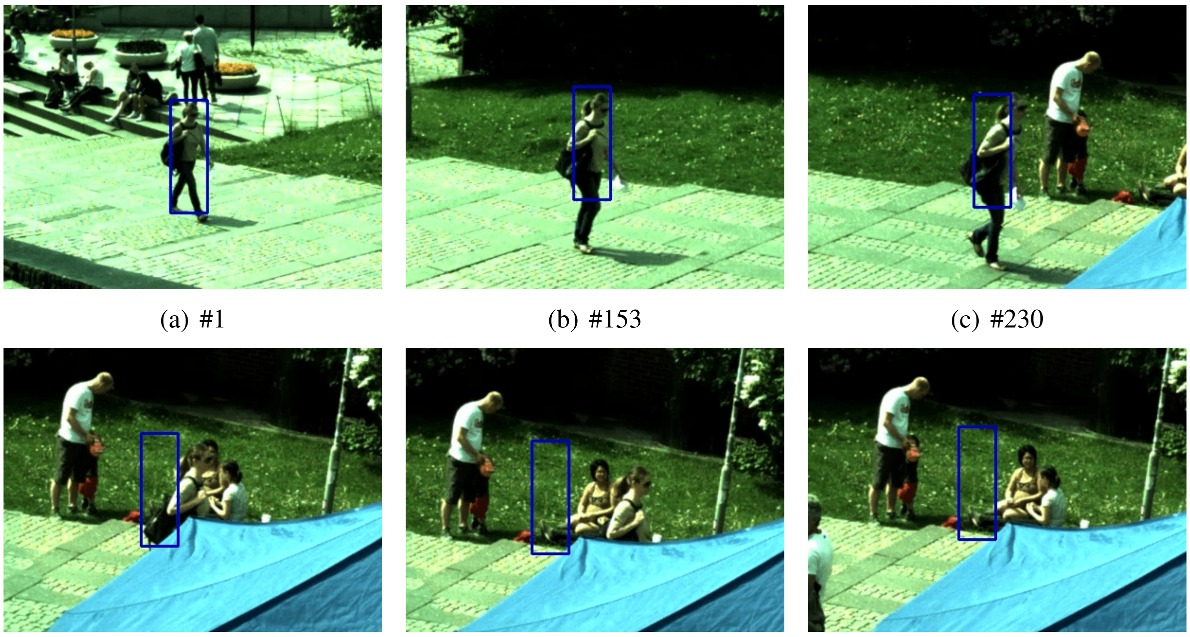

(a) #1 (b) #113 (c) #237
(d) #267 (e) #276 (f) #303

Figure 5.4: The rectangle marks the output of the tracking algorithm. Yellow and blue colors represent low and high confidence, respectively. As seen in the images, TLD adapts its bounding box online while its learning.

5.3.3 Real Time Compressive Tracker

With medium accuracy RTCT performs very similarly to TLD in terms of accuracy, but is able to sustain through the background clutter that confused TLD for a few more frames. A timeline of the tracking is shown in figure 5.5. It is able to track the first *271* frames before it finds interesting features in the background and loses the object. RTCT's slightly higher accuracy results in an inaccurate retrieval rate of *86%*. Out of the *271* continuous frames, only *3* frames were classified as inaccurate, producing an inaccurate retrieval rate only *0.9%* lower than its accurate retrieval rate. With an average per frame of *52*, RTCT is once again able to process the frames faster than most of the other trackers due to its efficiency during feature extraction and classification. It is only surpassed by the Color Tracker, which in comparison does not provide any kind of adaptive tracking.

Reviewing the high accuracy data we see the same effect as with TLD, and the accurate retrieval rate drops significantly (from *85.1%* to *64.1%*). Even though the amount of inaccurate frames is increased substantially, it is able to stay on target and still produce roughly the same inaccurate retrieval rate.



(a) #1 (b) #153 (c) #230

(d) #269 (e) #278 (f) #302

Figure 5.5: The blue rectangle marks the RTCT output

5.3.4 Optical Flow Tracker

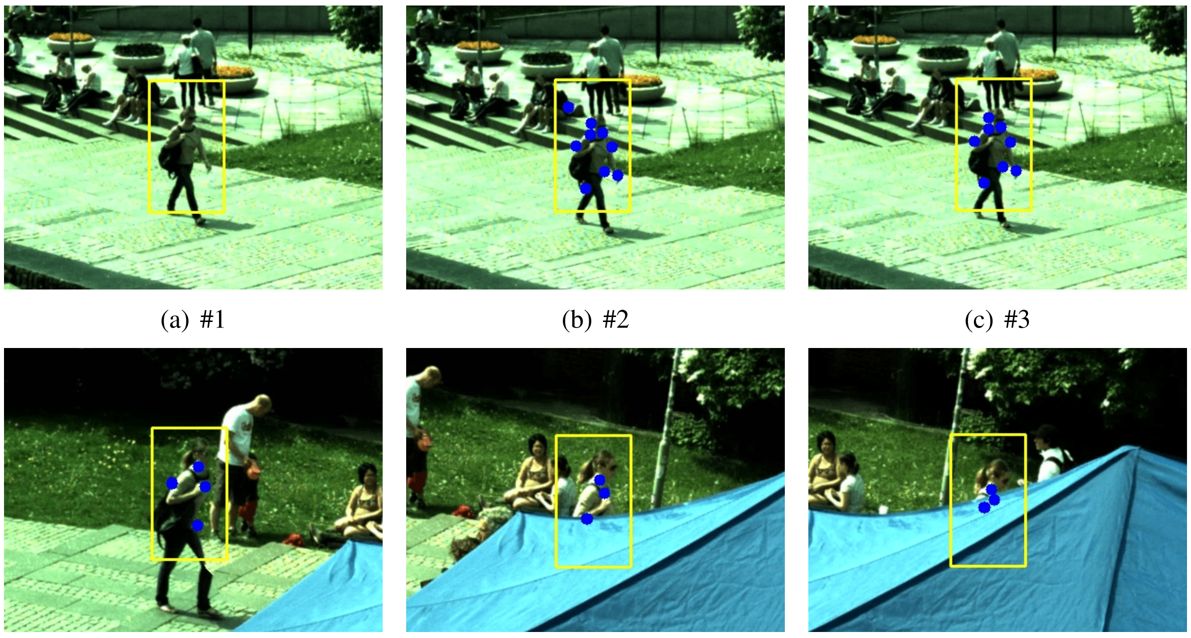
Raising accuracy further, the Optical Flow Tracker is able to track *305* consecutive frames, of which *296* are accurate and only *9* inaccurate (with medium accuracy). Although the tracker does lose some of the features while its tracking, it is able to keep enough of them to stay on target even during the huge occlusions toward the end (seen in figure 5.6, frame #300). With a 94% accurate retrieval rate, it is only *1.2%* less than the highly accurate Struck. Aimed at a real time application, Optical Flow is the clear choice of the two in this case, with an average processing time per frame at only *31* ms (faster than Struck by a factor of *15*), and second best in almost all other performance measures (see table in section 5.4).

Unlike Struck, however, the Optical Flow Tracker is unable to uphold is high accuracy when enforcing the high accuracy constraint. To a great extent this is a consequence of the way it outputs its tracking results, which is calculated as the centroid of all the points/features being tracked. The centroid position fluctuates as the number and position of these features change during the course of the tracking. Unlike TLD and RTCT, where frames are lost only because the background is tracked, it loses much fewer frames, and its high count of inaccurate frames is because the tracked features are far from the object center (where the ground truth is).

Although it does not provide a pixel-perfect tracking, the metric calculating the final position can easily be changed to a more accurate estimate than a pure average of the object features' coordinates, possibly raising its theoretical accuracy. One possibility could be a popular approach used in object tracking, where the amount of overlap between the tracker bounding box and the ground truth bounding box is used.

5.3.5 Background Subtraction Tracker

Due to the nature of the video, we wanted to explore how a background subtraction algorithm could aid in object tracking. In this sequence, the object is fairly isolated and often the only



(a) #1 (b) #2 (c) #3

(d) #235 (e) #283 (f) #300

Figure 5.6: The yellow rectangle marks the region of interest searched for features. The solid blue circles represents all features found.

moving entity in its vicinity, creating an ideal scenario for such an experiment. A pseudocode for the following algorithm can be found in the appendix, in section A.2.

As seen in the first image in figure 5.7, the background subtraction algorithm gives us a binary image of the foreground, and our object is successfully detected in the second frame of the sequence (the first is used to establish a background model). In frame #3, we see the tracker algorithm's effort of limiting the search space for the CC algorithm[1], and this region is marked as a green rectangle.

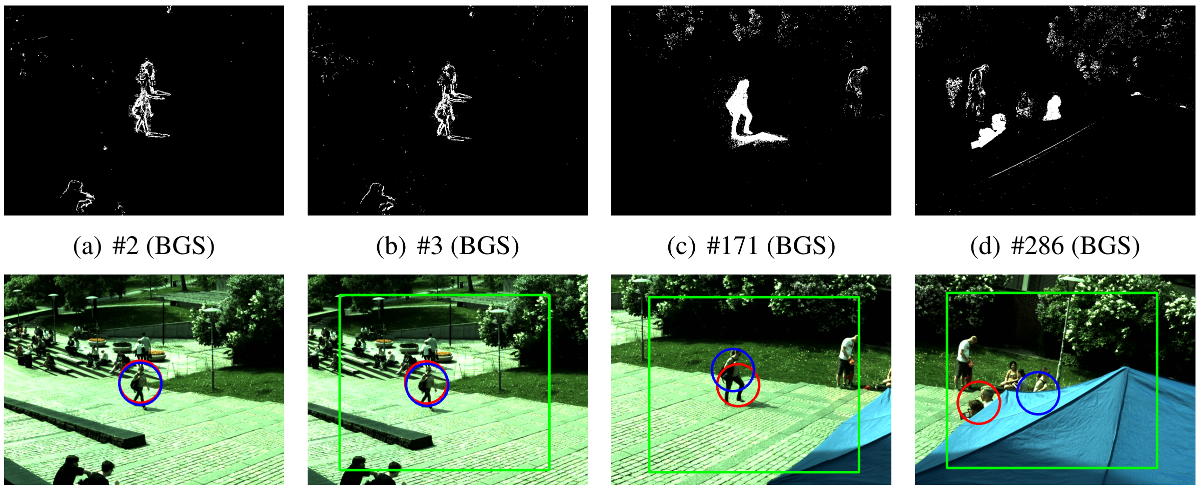
The tracking algorithm is able to track *40* frames continuously, and a total of *66* frames were accurate. Due to the shadow being detected by the BGS algorithm, seen in frame #171, the average of the blob coordinates is too far down in the Y-direction, and the result is a number of inaccurate frames where many are also considered lost. The algorithm is robust against partial occlusions (as long as there is movement), but the algorithm suffers from the same issue as the Optical Flow Tracker. Frames are lost because the tracking position is too far off the object center, but still on target. Frame #286 demonstrates another issue, where other moving objects are crossing into the ROI, and confuses our tracking. The largest blob within the ROI is always selected regardless of how many or how large they are, and as a consequence, the wrong blob is tracked in this case. We could remedy this effect by altering how the blob is selected, but due to the small amount of information available in a binary image, distinguishing blobs is very difficult.

Although not too valuable on its own, such an algorithm can provide very useful information which could be leveraged in combination with another approach, such as optical flow or color tracking.

5.3.6 Color Tracker

The previous case study presented us with a great opportunity to track and detect using the color of the object. The beanie color was very clear, but also close to unique to the scene (few other

[1]Compared to the original image in figure 5.1, this ROI is roughly an 82% reduction in pixel area.


(a) #2 (BGS) (b) #3 (BGS) (c) #171 (BGS) (d) #286 (BGS)

(e) #2 (original) (f) #3 (original) (g) #171 (original) (h) #286 (original)

Figure 5.7: Top row of images show the binary blobs obtained after background subtraction. Bottom row illustrates the output of the tracker. Blue circle indicates the ground truth, red circle is the tracker output and the green rectangle is the search area for the blob-detection heuristic.

image regions passed through the filter). This case presents the issues of tracking using color as a feature, when the object's colors are far from unique. One of the two major colors of the object is black, and due to the large amounts of black color in other uninteresting parts of the image, this cannot be used to uniquely identify our object. In figure 5.8, we see the results of applying a color range aimed at capturing the object's other colors, besides black. These colors are also non-unique and, as they can be found in several other image regions, produces a noisy binary image where distinguishing our blob from the noise is very difficult.

As seen in figure 5.8(a), the biggest blob in the image is located in the top right. This highlights an important limitation when using tracking by colors, as we are unable to provide a usable tracking when the object's colors are non-unique. Also, since there are no other features being used in the tracking, there is no fallback mechanism either.

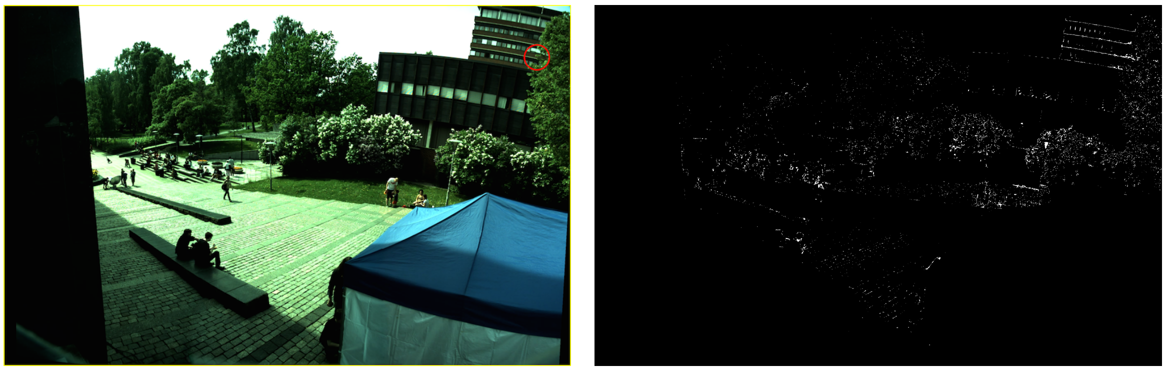

(a) #1 - Original image (b) #1 - Color filtered image

Figure 5.8: The color range of the object is too wide and results in noisy output (rightmost image). Red circle shows the tracker output (leftmost image). The yellow frame border around the original image depicts the search region for the detection (initially the full image)

5.4 Performance

	Struck	TLD	RTCT	OFT	BGST	CT
Average speed (ms)	466	423	52	31	116	248
Accurate frames	308	249	268	296	66	0
Inaccurate frames	1	16	3	9	42	0
Continuous frames	309	265	271	305	40	0
Lost frames	6	50	44	10	207	315
Accurate retrieval rate	97.8%	79%	85.1%	94%	21%	0%
Inaccurate retrieval rate	98.1%	84.1%	86%	96.9%	34.3%	0%

Table 5.1: Medium accuracy. Green color indicate the best performance while blue color indicate the second best.

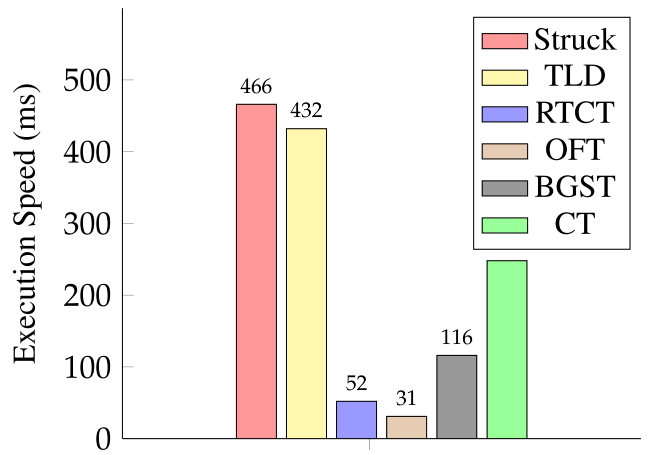
	Struck	TLD	RTCT	OFT	BGST	CT
Average speed (ms)	466	423	52	31	166	248
Accurate frames	236	201	202	176	35	0
Inaccurate frames	72	48	66	120	31	0
Continuous frames	308	243	268	272	18	0
Lost frames	7	66	47	19	249	315
Accurate retrieval rate	74.9%	63.8%	64.1%	55.9%	11.1%	0%
Inaccurate retrieval rate	97.8%	79%	85.1%	94%	21%	0%

Table 5.2: High accuracy. Green color indicate the best performance while blue color indicate the second best.

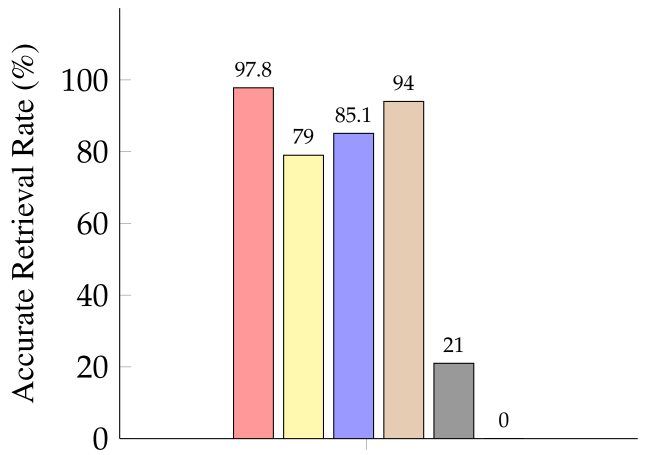
Once again we provide a summary of the all trackers' performance, seen in tables 5.4 and 5.4. Plots are also given for the most important metrics, showing the medium accuracy in figure 5.9, and the high accuracy in figure 5.10. In this case study, we have seen that when an object contains more features, it is naturally easier to track. More internal contrasts, shape changes and appearance variations within the object help the adaptive trackers to establish an appearance model. The object can now more easily be discriminated from the background, thus creating a harder decision boundary for the classifiers and a stronger tracking. This enabled TLD, RTCT and Struck to do much better than on the previous sequence, where there was a lack of features, and track the object in a larger percentage of the frames. However, the extra camera that was added to the system produced a larger workload for the trackers, and this is also reflected in their execution speed. Struck and TLD's execution times were increased by *148* and *56* ms, respectively, while RTCT's execution time only increased by *5* ms. OFT's execution time was nearly doubled, from *19* to *31*, and now only *9* ms below our real-time requirement. We see that the added number of object features also clearly aided OFT too, as it was able to track *96.9%* of the frames in the sequence using medium accuracy. Struck was able to track the highest number of frames, *309*, and was also the most accurate tracker. Struck, in particular, handled the heavy occlusions well and was able to still stay accurately on target. The side effect was an execution speed of nearly $\frac{1}{2}$ second on each frame.

Although CT performed very well in the previous sequence, we saw in this case study that color tracking is not always suitable. The filter must be pretuned to the video in an offline manner, and must be specifically adapted to each use case at hand. We were unable to find a suitable HSV vector to match with the object's colors, without including too much noise. Since the color tracking contained no other fallback mechanism, it was unable to provide any tracking services at all. Despite the lack of proper tracking provided by BGST, we saw that background subtraction can be a very useful tool as moving objects are part of the foreground and can easily be detected.

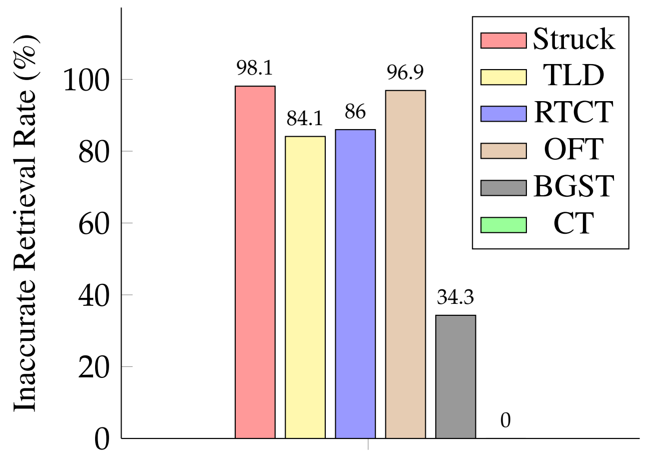
To properly leverage such information and for example be able to distinguish between multiple foreground objects, BGST could be combined with e.g. color tracking. Consequently, both tracking approaches can mutually benefit each other to cover more use cases, and increase the quality of the both approaches.



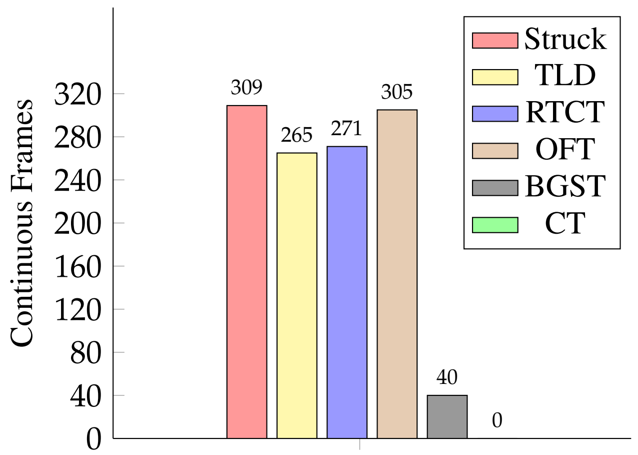
(a) Execution speed - *Medium accuracy*



(b) Accurate retrieval rate - *Medium accuracy*



(c) Inaccurate retrieval rate - *Medium accuracy*



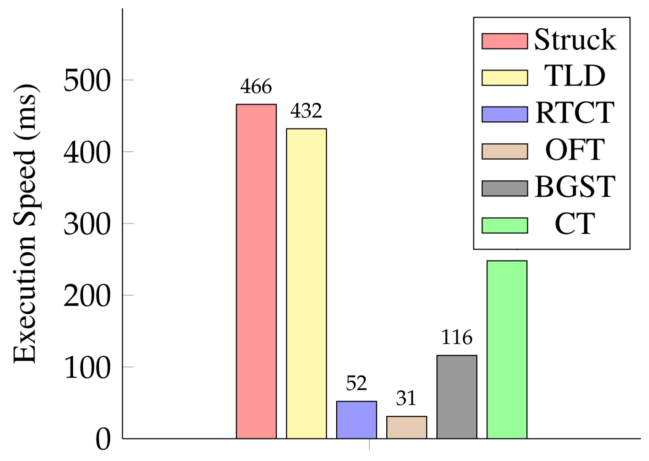
(d) Number of Continuous Frames - *Medium accuracy*

Figure 5.9: Performance plots for Execution Speed, Accurate Retrieval Rate, Inaccurate Retrieval Rate and Continuous Frames for each tracker. Medium accuracy.

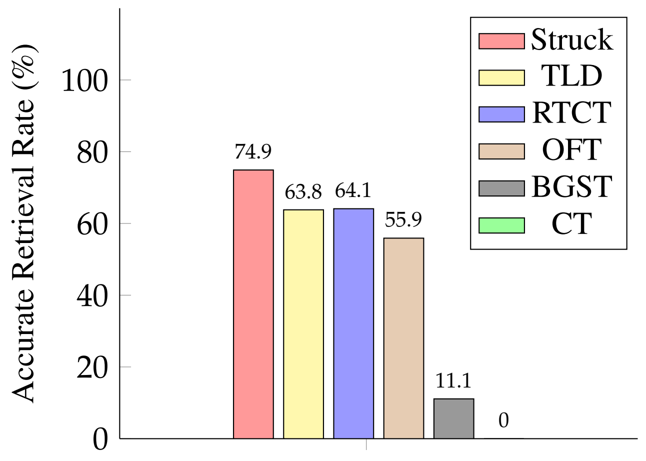
5.5 Summary

In this case study we have reviewed another use case, where the system was installed in an outdoor setting with an additional camera added to the setup. We saw the effect of increased workload as the OFT was now the only tracker able to stay within our real-time requirement. Color tracking, which previously provided usable tracking and fast execution, was not applicable in this case study. Also, we saw that the trackers in varying degree was able to handle the strong occlusions. Particularly, Struck demonstrated robustness, but at a high cost and far from real-time operation.

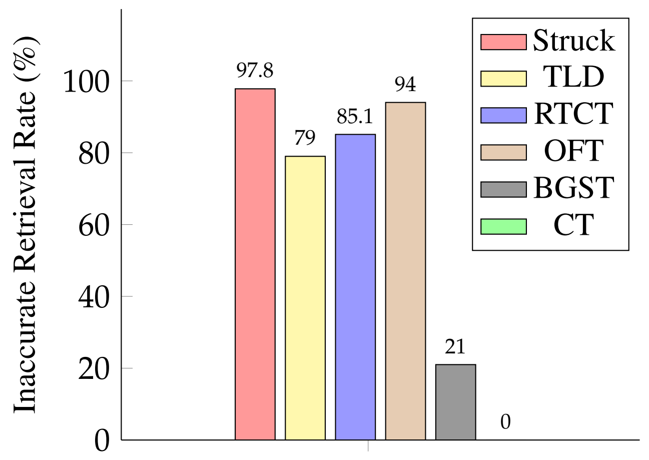
In the next chapter, we move on to evaluate the trackers in the last and most challenging of the case studies. A complete prototype of the system is run, installed at Alfheim Stadium, using all 5 cameras for full coverage of the whole soccer field. Furthermore, the object size is now



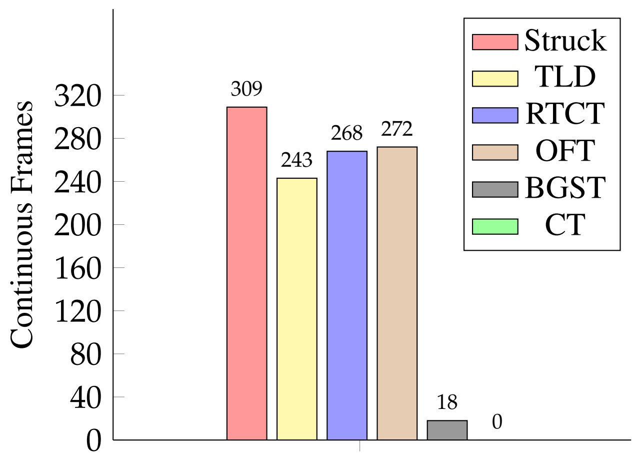
(a) Execution speed - *High accuracy*



(b) Accurate retrieval rate - *High accuracy*



(c) Inaccurate retrieval rate - *High accuracy*



(d) Number of Continuous Frames - *High accuracy*

Figure 5.10: Performance plots for Execution Speed, Accurate Retrieval Rate, Inaccurate Retrieval Rate and Continuous Frames for each tracker. High accuracy.

reduced by roughly *90%* compared to the other case studies. Also, the object is moving very quickly across frames, producing challenging cases of motion blur.

Chapter 6

Case study 3 - Ball tracking

6.1 Introduction

Our final case study entails the most complicated of the three, where we aim to track the ball in video from the Bagadus system. As we have seen, there is a large amount of challenges to be overcome, but we will use the experience from our previous case studies to determine the best possible tracking. As before, the full pipeline is installed, but now using all 5 cameras. The panorama stream now outputs images in a resolution of *4450 x 2000* pixels, increasing the workload further by roughly 50% compared to the case study in chapter 5. The cameras are still configured to run at 25 FPS, and in order to conform to the real-time requirement, each tracker now has 40 ms to process an image of *8 900 000* pixels.

The large variations in lighting conditions in the stadium can create very difficult conditions, especially during daytime with strong sun, as seen in figure 6.1. Due to the enormous challenges already present, we avoid initial testing on these harsh conditions by using footage from a match recorded in the evening instead, offering more stable lighting conditions. The sequence used in the case study is a 3-second segment (75 frames) from a recorded match between Tromsø IL and Tottenham Hotspur F.C, played at 7PM November 28, 2013 (a sample image is seen in figure 6.2). The sequence is taken from the dataset in [45]. Like in the case study in chapter 5, we include performance summaries for both levels of accuracy. The ball occupies a much smaller part of the image compared to the other images, and thus calls for higher accuracy to give meaningful results.

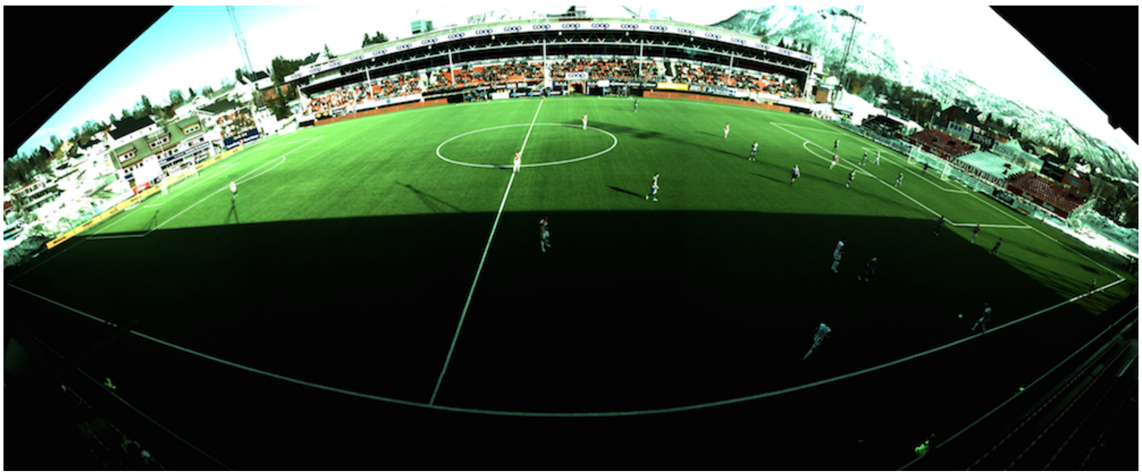


Figure 6.1: A match from June 2014 demonstrates harsh the lighting conditions that might arise

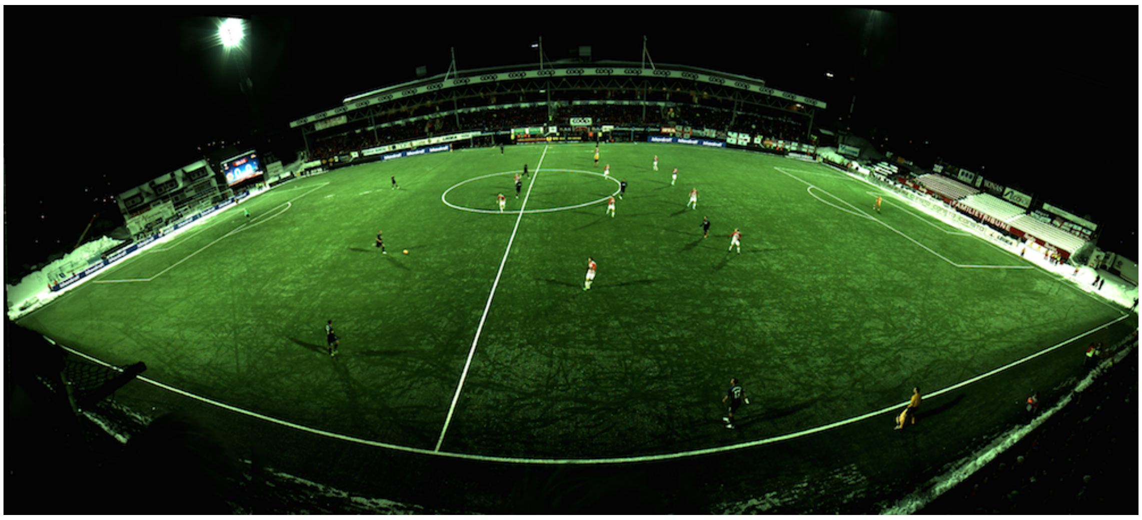


Figure 6.2: An overview of the entire field from the sequence used in the case study (5-camera panorama)

6.2 Challenges

This sequence is more challenging than the previous case studies for a number of reasons. As mentioned, there are a lot more pixels to process for the tracker, moving us towards trackers with the fastest execution speed, and with possibly lower accuracy than slower-performing ones. Furthermore, the object is a smaller part of the image, creating a smaller area of which the tracker can make e.g. an appearance model. In the previous sequences, the bounding box around the object was roughly *5000* pixels in size, while the bounding box is now only roughly *500* pixels. This pragmatically results in a less unique object and, as small parts of other regions in the image may appear similar, possibly increasing the count of false positives.

As discussed in section 3.4, a number of challenges are present, of which the most important to overcome for robust tracking are now occlusion and object motion. The players may frequently occlude the ball over longer periods of time, and the ball moves a much larger distance between frames than in the other case studies. An effect of these large motions is that the ball deforms and might look substantially different from its original appearance, as we will see.

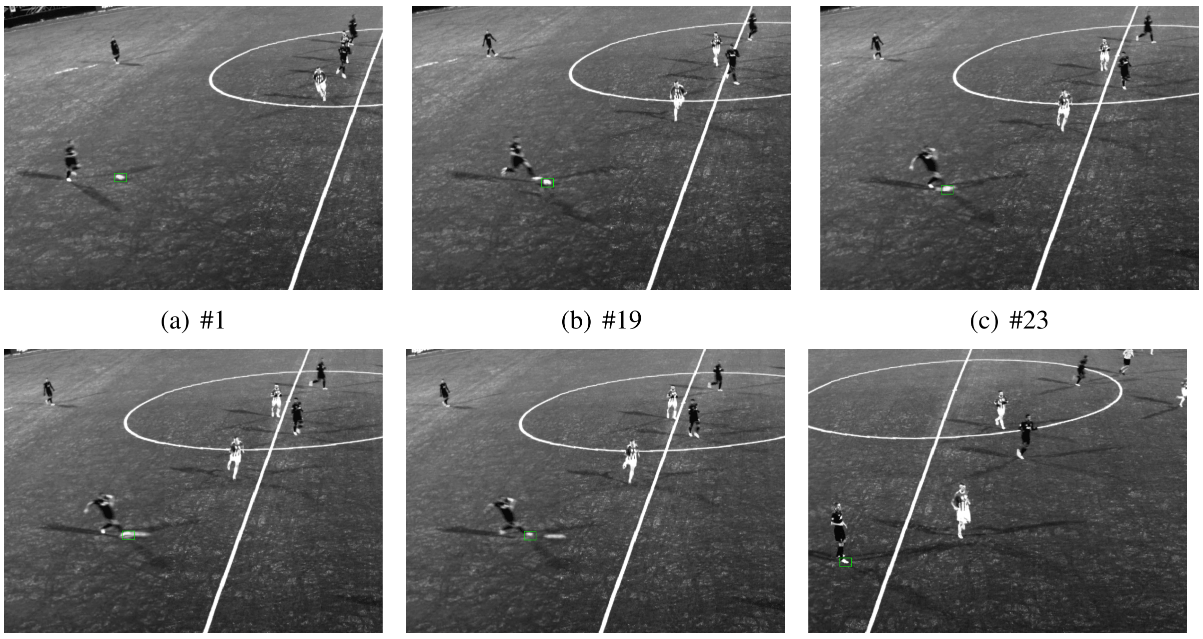
6.3 Evaluating and comparing tracking algorithms

6.3.1 Structured Output Prediction using Kernels

We can clearly see the effect of object size, as discussed in the previous section. Struck is able to accurately track the ball for the first *23* frames, until the ball is kicked. Frame #23 in figure 6.4 shows how the blurred ball and parts of the shoe is added to the positive samples during the kick. In frame #24 the entire shoe is added as well, and the tracking drifts and shifts away from the ball, tracking the shoe instead. The tracker then, ironically, robustly tracks the shoe until the near end of the sequence, and the results are clearly reflected in the support vector samples. It is noteworthy that this kick is meant as a pass, and does not demonstrate the blur which might occur during a high velocity shot to the goal. On positive note, Struck was able to track accurately in all frames leading up to the kick, meaning a useful tracking was provided when the ball was isolated and free of occlusions.

The tracking failed for two different reasons. First, it failed due to the high velocity of the ball, creating motion blur and a misleading sample to the classifier. Second, the occluding object (the shoe) was similar enough to the ball that it, too, was added to the positive samples. This demonstrates the issue of tracking an object with a lack of distinctive features, where fast motion makes the matter even worse. A possible remedy would be to eliminate the motion blur by increasing the frame rate, creating a clear distinction between the shoe and the ball. On the other hand, this also presents an even greater workload.

More importantly, we note that the average speed per frame was close to a full second (*854* ms). The tracker is overwhelmed by the sheer amount of pixels and hence is more than 800 ms past the real-time requirement of 40 ms, rendering the algorithm useless even if 100% retrieval rate was achieved.


(a) #1 (b) #19 (c) #23

(d) #24 (e) #25 (f) #69

Figure 6.3:

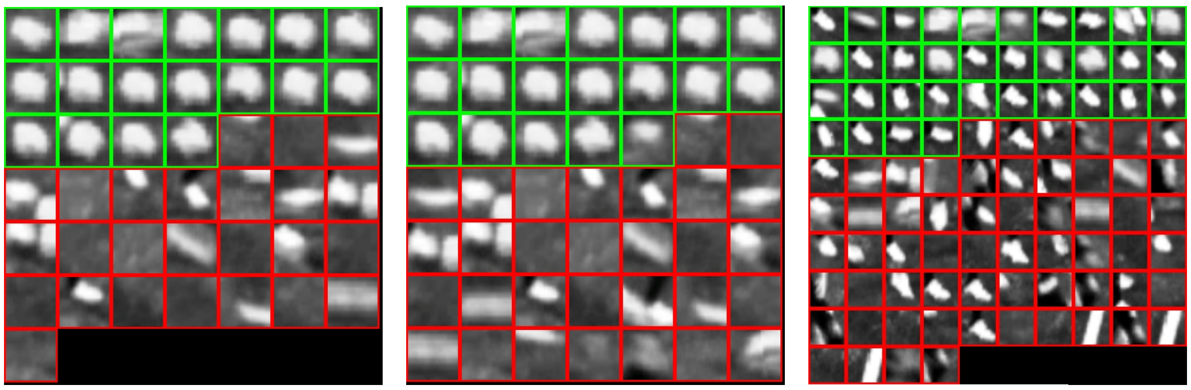
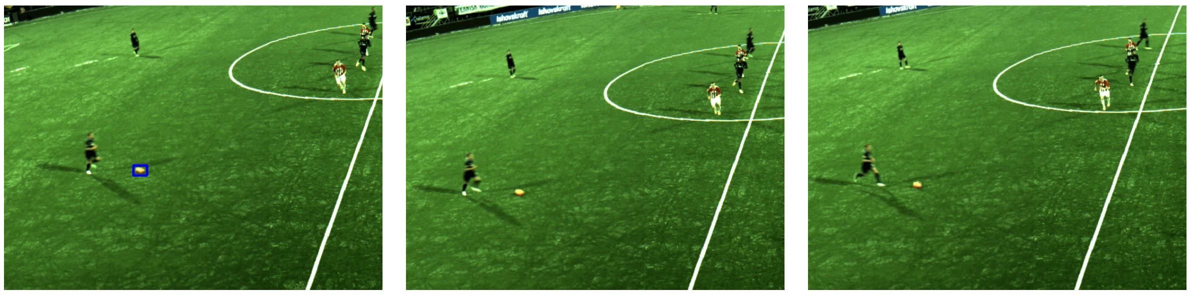

(a) #23 (b) #24 (c) #69

Figure 6.4:

6.3.2 Tracking-Learning-Detection

The TLD tracker is not able to do any proper tracking in this sequence. The tracking is lost already in the second frame of the video. The learning component is responsible for creating the object model using the first frame and initializing the detector with its results. However, the geometric transformations it is using are not enough to cover the change seen in the next frame, most likely due to the lack of features in the object's region. The detector is thus unable to correct the Median-Flow tracker when it fails. As we have seen previously, TLD often has troubles recovering after losing the object, and we see the detection mechanism fails here too (the appearance model has not yet been through enough iterations to recover from failure). After the first frame, all subsequent frames are used in its attempt to redetect the object, resulting in an extremely slow execution speed of *1346* ms on average per frame. Tracking only 1 of 75 frames, both retrieval measures stop at *1.3%*.



(a) #1 (b) #2 (c) #5

Figure 6.5: Blue rectangle shows the output position of the TLD tracker

6.3.3 Real Time Compressive Tracker

When reviewing the medium accuracy data, RTCT and Struck's performance on this sequence is nearly identical. An image timeline is provided in figure 6.6. RTCT is also able to accurately track the ball for the first *23* continuous frames, but then shift its focus to the shoe of the player kicking the ball, and tracking fails in the subsequent frames. The motion blur and the ball's similarity to the shoe confuse both algorithms and causes drift. Looking closely at frame #23, it is actually the motion blur of the kicking foot that initially blends with the slowly moving ball. When the ball gains speed as well, they merge together. There is little difference when using the higher accuracy, and we see that RTCT has only *2* more inaccurate frames and Struck has *1*.

Although both the accuracy and retrieval rates of RTCT and Struck are almost equal in both cases, RTCT is able to process each frame in only *63* ms on average, which is nearly *800* ms faster than Struck. We see that RTCT's use of a sparse measurement matrix and working in a compressed subspace is effective, and allows for much smaller amounts of computations than the structured output prediction used in Struck. Considering the size of the panorama, RTCT is fairly close to operating in real-time, as it is only *23* ms above our requirement of *40* ms per frame.

6.3.4 Optical Flow Tracker

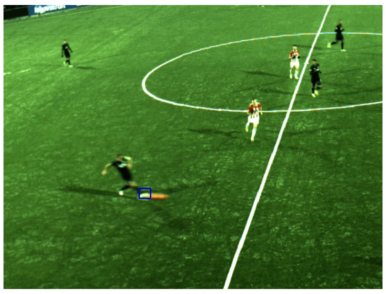
As seen in the tracking output in figure 6.7, the Optical Flow tracker performs near identical to both Struck and RTCT when using the medium accuracy measurement. The only difference is that the Optical Flow tracker is able to track the ball, inaccurately, for *1* more frame than Struck and RTCT. This increases the inaccurate retrieval rate slightly, by *1.3%*. Enforcing the higher accuracy this extra, yet inaccurate frame, is considered lost and reduces the inaccurate retrieval rate back down to *30.7%*.

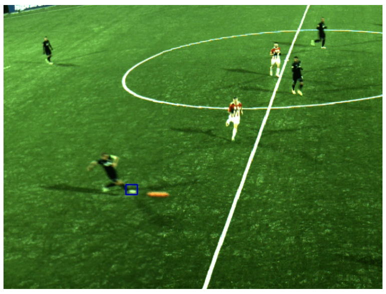
The lack of features once again leads the tracker away from the object, and shifts the focus to background clutter with more prominent features (more precisely the shoe of the player). However, in the frames that were tracked, the low complexity of its computations allows it once again to be significantly faster than Struck and TLD, with an average speed per frame of *67* ms.

(a) #1

(b) #10

(c) #23


(d) #24


(e) #25

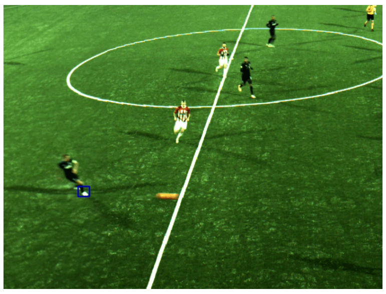
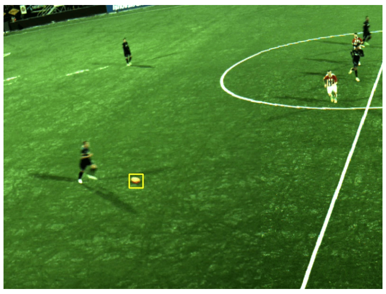

(f) #28

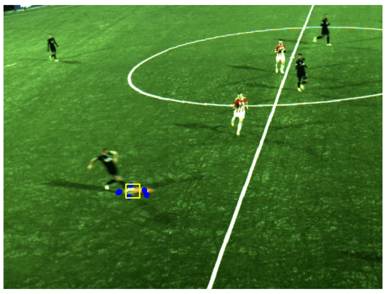
Figure 6.6: Blue rectangle shows the output position of the RTCT tracker

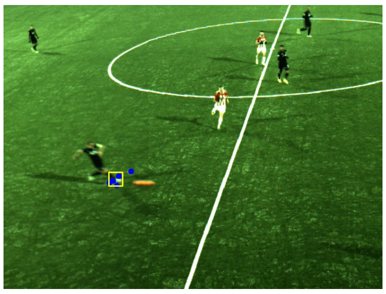
Despite its high speed, it is still *27* ms above the real-time threshold, and slightly behind RTCT's performance.


(a) #1

(b) #19

(c) #23


(d) #24


(e) #25

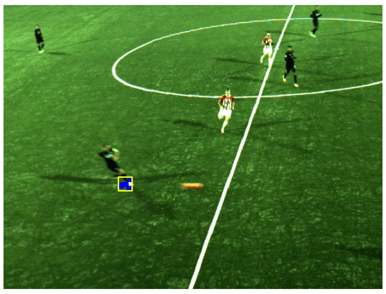

(f) #27

Figure 6.7: Yellow rectangle marks the feature search area, and the blue dots represent each feature found by the OFT

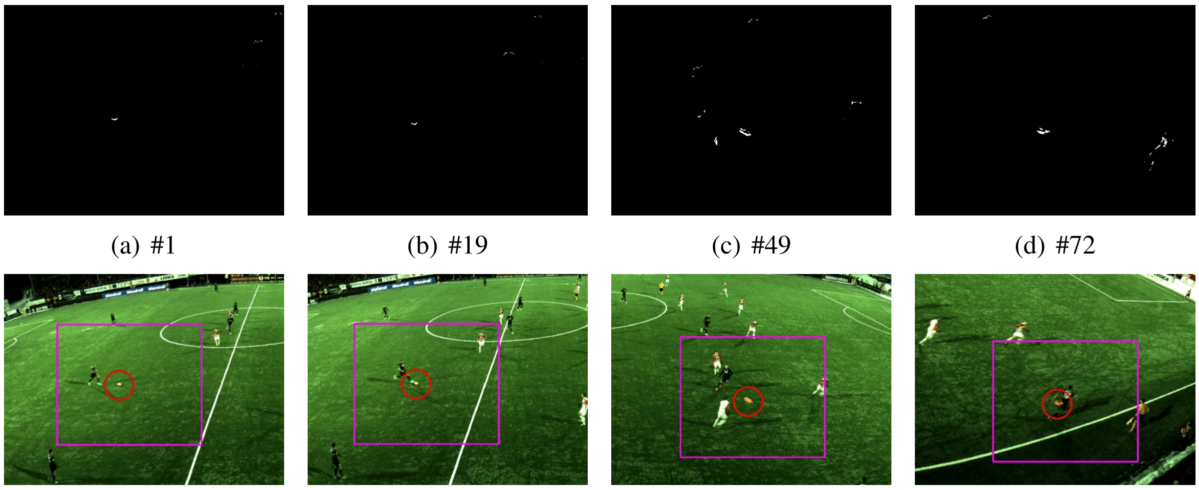
6.3.5 Color thresholding

This sequence once again illustrates a terrific opportuniy to use color as a feature to facilitate tracking. The ball is orange, with few other elements in the image of the same color, making it sufficiently unique. Like the case study in chapter 4, the color range is defined as a set of two

HSV vectors, describing the permitted range for the pixel values that should pass through the filter. The same filtering operation that was shown in section 4.3.4 is applied, using the new color vectors [10,200,125] and [22,255,255] to describe the lower and upper bound, respectively.

In the case study in chapter 4, the detection mechanism used the entire first frame to determine the initial position of the object, before creating a ROI in subsequent frames (reducing the search space). Due to the panorama's huge coverage and to avoid any false positives during the initialization, we instead setup an ROI on the first frame, similar to how the adaptive trackers set up a bounding box. The image is run through the filter and the ROI is scrutinized to locate any blobs in the binary image. We can see the ROI marked as a purple rectangle in figure 6.8, and the binary output after the color filtering is shown in the top row images.

This approach is able to successfully and accurately locate the object in *97.3%* of the frames, using medium accuracy. With only *2* lost frames and *73* tracked frames, this tracker is able to retrieve more frames than all the other trackers comfortably. Lowering the permitted drift down to *15* pixels (high accuracy), it is still able to track the same number of frames, whereas only *1* more frame is considered inaccurate. Naturally this decreases the accurate retrieval rate slightly, by *1.3%*, and leaves the inaccurate retrieval rate at *97.3%*. Furthermore, it is also by far the fastest, with only *21* ms on average per frame. This puts our Color Threshold tracker well within our real-time requirement of 40 ms, unlike all the other tracking algorithms.



(a) #1 (b) #19 (c) #49 (d) #72

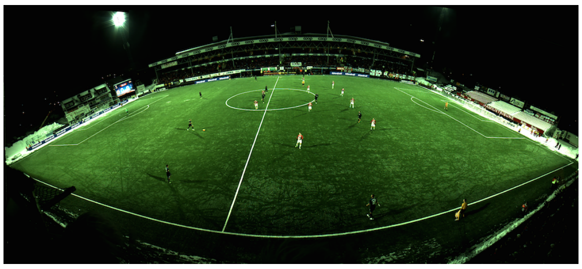
(e) #1 (f) #19 (g) #49 (h) #72

Figure 6.8: Top row is the output images after a color threshold on orange. Bottom row is the tracker output represented by a red circle, along with a purple rectangle showing the ROI.

6.3.6 Background Subtraction Tracker

In the earlier versions of Bagadus, background subtraction was used to aid the panorama stitcher avoid creating its seam through players, as detailed in section 2.1. As we have seen in the previous case studies, background subtraction can provide very useful information for a tracker as well. Object tracking is usually desirable in order to track some kind of movement, and background subtraction can be useful for pruning the search space by locating moving entities quickly. However, when multiple objects are moving, a binary image of the moving objects is not enough to uniquely identify and track a single object of interest. On a soccer field there are a plethora of blobs in the binary output image: players, referees, the ball and potential movement in the background (see figure 6.9). This leads to the same results as in the case study from chapter 4, where the tracking fails due to multiple blobs. This, however, can be remedied by including other features than the object's movement in order to establish more uniqueness in each object, and thus more easily distinguish between them.

In the old pipeline, the background subtraction was used to determine which pixels were occupied by players by looking at the players' positioning data provided by ZXY. Then, the stitcher could use the look-up produced by the background subtraction to know which pixels to avoid. In an object tracking scenario, one could do the opposite by using the player positioning data to remove each player's pixels. In an ideal world this would leave only the ball in the binary image output from the background subtraction, given that it is moving (which is almost always the case in soccer). An obvious challenge here would be to separate the player pixels from the ball pixels since the ball is more than often in the feet of a player. Still, background subtraction could prove an important tool in providing object tracking in Bagadus, especially in combination with player positioning data.


(a) The original image

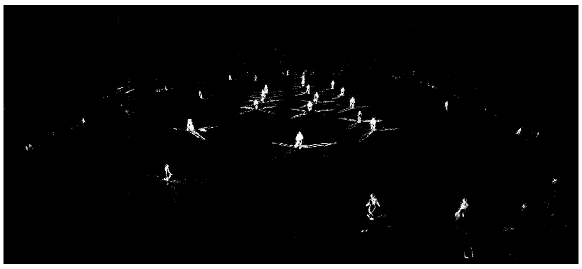

(b) Results after background subtraction

Figure 6.9: An example of background subtraction

6.4 Performance

	Struck	TLD	RTCT	CT	OFT	BGST
Average speed (ms)	854	1346	63	21	67	318
Accurate frames	23	1	23	73	23	0
Inaccurate frames	0	0	0	0	1	0
Continuous frames	23	1	23	72	24	0
Lost frames	52	74	52	2	51	75
Accurate retrieval rate	30.7%	1.3%	30.7%	97.3%	30.7%	0%
Inaccurate retrieval rate	30.7%	1.3%	30.7%	97.3%	32%	0%

Table 6.1: Medium accuracy. Green color indicate the best performance while blue color indicate the second best.

In general, we see the same effect we saw in our first case study in chapter 4. A lack of distinctive features in the object made it difficult for the trackers. However, besides TLD, the trackers were able to track the ball in the first roughly 20 frames, when the ball was moving freely and without occlusions. We saw that despite the relative size of the ball, some level of tracking was indeed provided. As expected, we also observed that the fast motion of the ball confused all trackers except our Color Tracker. Due to its use of colors as a feature, which are naturally the same regardless of the object's speed, it was able to withstand this large change

	Struck	TLD	RTCT	CT	OFT	BGST
Average speed (ms)	854	1346	63	21	67	318
Accurate frames	21	1	21	72	23	0
Inaccurate frames	1	0	2	1	0	0
Continuous frames	23	0	23	72	23	0
Lost frames	52	74	52	2	52	75
Accurate retrieval rate	29.3%	1.3%	28%	96%	30.7%	0%
Inaccurate retrieval rate	30.7%	1.3%	30.7%	97.3%	30.7%	0%

Table 6.2: High accuracy. Green color indicate the best performance while blue color indicate the second best.

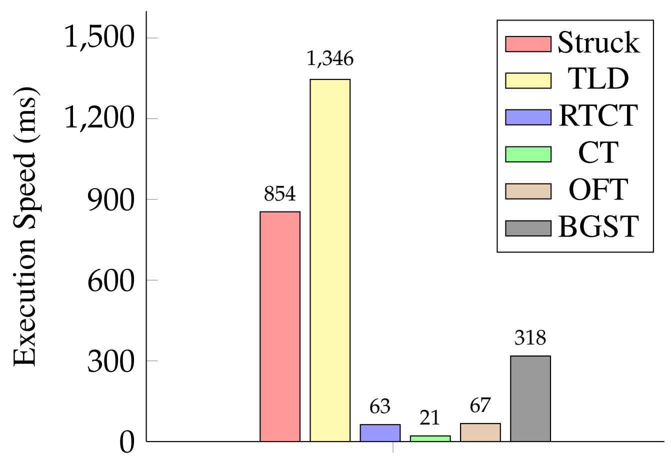
in object appearance. Furthermore, the Color Tracker was able to track the ball in almost all the frames in the sequence, resulting in a accurate retrieval rate of *97.3%*, due to its natural resistance to occlusions and fast motion. It does, however, require predetermined values for its color range. Furthermore, when the ball's colors are no longer unique to the scene, like when players wear jerseys with the same color as the ball, this can become practically infeasible (this effect was observed in chapter 5).

In the previous case studies, we have seen a large effect when moving from medium to high accuracy. This effect is diminished in this use case, as we see the trackers are quite precise when they are actually tracking the ball. However, the difficulty of this sequence is the motion blur, and is the reason for almost all trackers' lack of ability to maintain tracking. Looking at this final case study, we also see that only our simplistic Color Tracker is able to stay within our real-time threshold, executing at *21* ms on average per frame. RTCT is the second fastest, with *63* ms on average per frame, while OFT is at *67* ms. It is important to note that all of these implementations are running exclusively on a CPU. When incorporating tracking into a module of the pipeline, a GPU port of each tracker, such as the other modules (Dynamic Stitcher, Bayer Converter etc), would likely cause a considerable speedup.

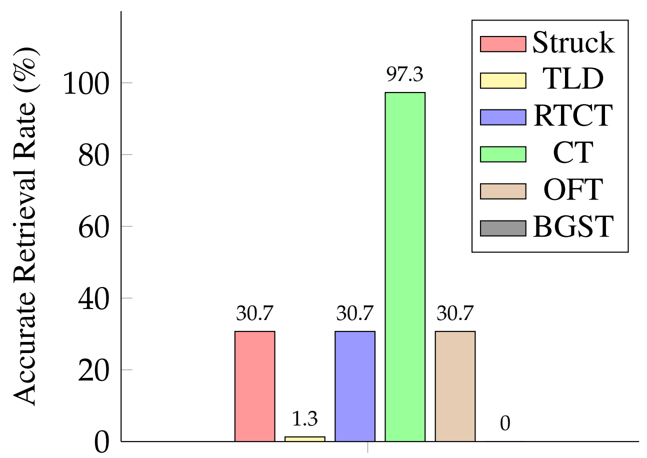
6.5 Summary

In this chapter, we have presented our final case study and evaluated object tracking in a sequence produced by the Bagadus system. Here, we have used the full setup of 5 cameras, analyzing images of 4450×2000 pixels. We have seen that most trackers tested were unable to provide the level of overall performance expected by an analytical system such as Bagadus. The huge amount of pixels in the panorama was difficult to handle for the adaptive trackers, except for the efficient implementation of RTCT. Only our specialized Color Tracker was able to track for the majority of the frames and, for this particular case, produced data that can be used to provide new services (such as automatic camera guidance). We have also seen that the Background Subtraction Tracker was useless on its own in this sequence, but we recognize that as part of a larger algorithm, it could be of great use in an object tracking module.

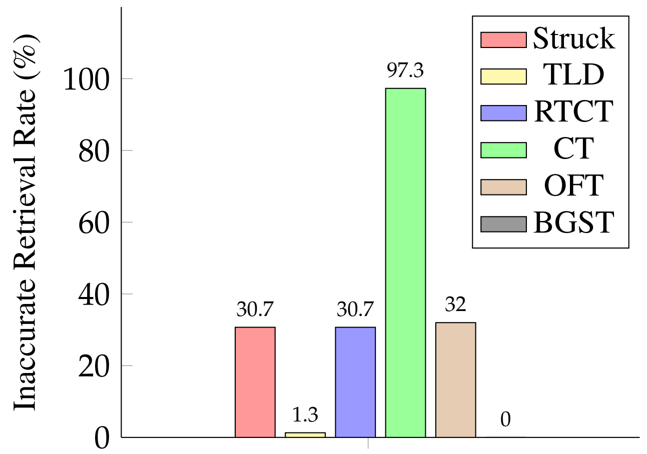
We see that ball tracking is still very difficult to achieve, even using the current state-of-the-art single object trackers. It is clear that exploiting a priori information about the scene, such as object color and its relation to the scene, is necessary to be able to reach the needed level of performance.



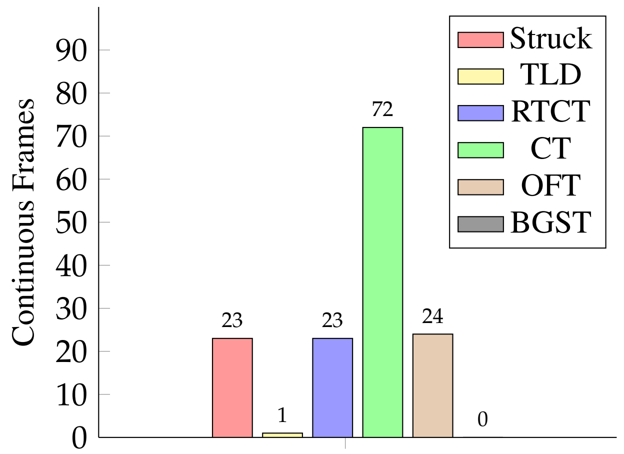
(a) Execution speed - *Medium accuracy*



(b) Accurate retrieval rate - *Medium accuracy*

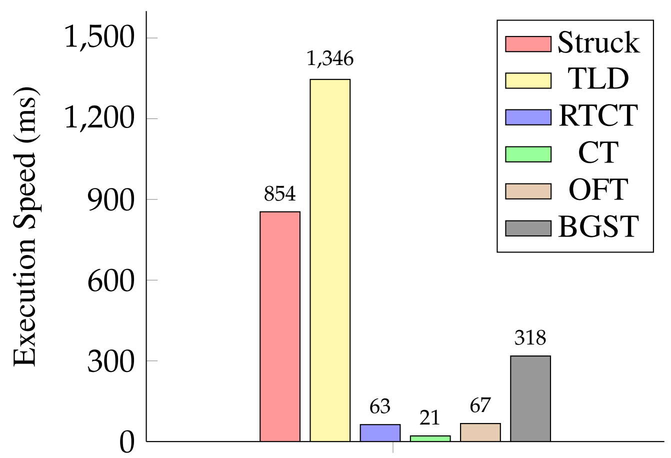


(c) Inaccurate retrieval rate - *Medium accuracy*

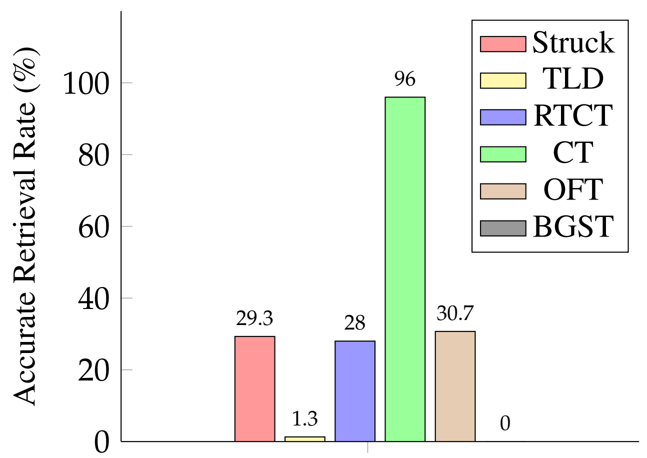


(d) Number of Continuous Frames - *Medium accuracy*

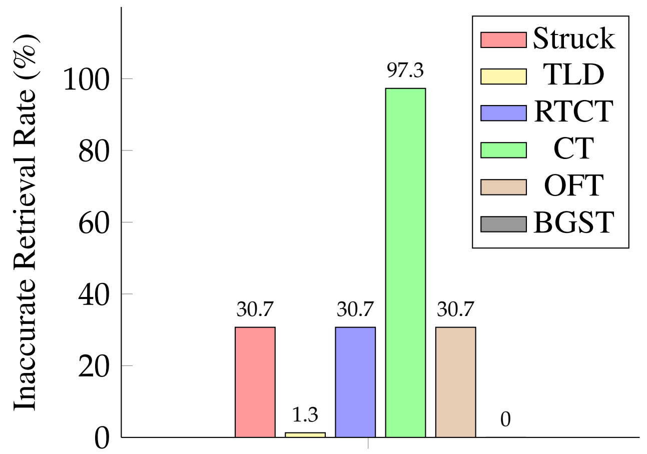
Figure 6.10: Performance plots for Execution Speed, Accurate Retrieval Rate, Inaccurate Retrieval Rate and Continuous Frames for each tracker. Medium accuracy.



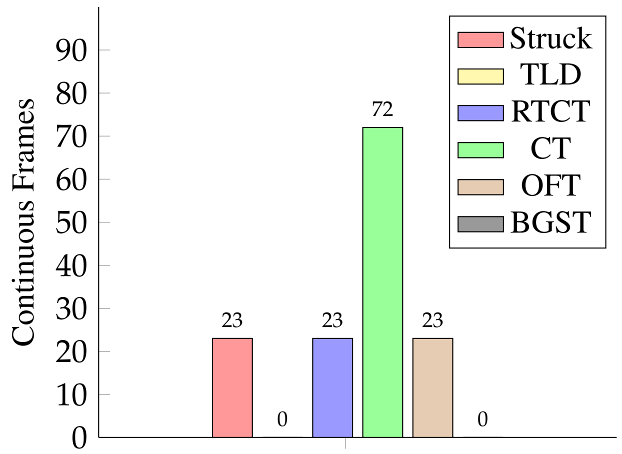
(a) Execution speed - *High accuracy*



(b) Accurate retrieval rate - *High accuracy*



(c) Inaccurate retrieval rate - *High accuracy*



(d) Number of Continuous Frames - *High accuracy*

Figure 6.11: Performance plots for Execution Speed, Accurate Retrieval Rate, Inaccurate Retrieval Rate and Continuous Frames for each tracker. High accuracy.

Chapter 7

Conclusion

This final chapter first provide a summary of the work done in this thesis. Next, we go on to review our main contributions and finally close out the chapter with a section on future work.

7.1 Summary

In this thesis, we have evaluated the feasibility and performance of object tracking in high resolution panorama images produced by a real-time distributed video capturing system. A previous prototype of the system was presented in chapter 2, where we described the overall architecture as well as a short review of each module in the pipeline. Later in the chapter, we reviewed important changes made to the system in order to allow for much higher scalability. Instead of keeping one machine in charge of all capturing, we saw that by distributing the process onto several machines we were able to not only greatly increase bandwidth, but also the speed of transfer. Furthermore, this allowed for use of much higher quality cameras. Combined with an HDR module and a cylindrical panorama, we were able to produce a much more visually pleasing panorama stitch than before.

In chapter 3, we looked into what forms the basis of a tracker, and saw that object representation, feature extraction and object detection are important components in a tracking algorithm. We reviewed our goals and requirements for a tracker, and pointed out the importance of accuracy and retrieval rate. Then, related work in the area of object tracking was presented, where we saw that object tracking in sports is a highly complex problem. Hawk Eye [22] and [61] showed promise, but made use of multiple highly expensive cameras and did not provide accurate ball tracking in all the different situations that can arise in team sports. Later in the chapter, we reviewed the current state-of-the-art in single object tracking, and selected three candidates to serve as representatives for current research.

Our first case study was presented in chapter 4. Here, 2 cameras were used in a test setup in an indoor environment as an initial step towards Bagadus. We observed that the object's appearance was difficult to handle for many of the trackers due to its lack of features, and only the specialized color tracker was able to provide usable results. Despite its good results, we also discussed the difficulty of porting such an approach to other sequences, which was an effect we saw in chapter 5. In this next case study, we investigated the six tracking algorithms on a setup using 3 cameras, with video captured outdoors during midday. Here, the trackers were generally able to handle quite a lot of occlusions, but the increased workload led to few trackers staying real-time.

In our last case study in chapter 6, we saw that the state-of-the-art adaptive trackers were unable to follow the fast movement off the ball coupled with the extreme resolution. Only the color tracking algorithm, which was tailored to the footage, was able to provide a level of overall performance that can be of further use to the system.

7.2 Main Contributions

In our research, we have given insight into what object tracking is able to do today. Through the presence of three case studies, we have shown the overall performance and feasibility of object tracking algorithms within the context of a high resolution cylindrical panorama image. By testing and evaluating both the current state-of-the-art as well as 3 tracker algorithms we have made, we have seen that this is no trivial task. In our first case study, detailed in chapter 4, we learned that an object with little texture and internal contrasts is difficult to track. The adaptive trackers, namely Struck, TLD and RTCT, had problems establishing a good object appearance model, and was thus highly sensitive when strong features appeared in the background. This caused drift in the object's appearance model and the tracking shifted to background regions. We also saw that we were unable to utilize the Background Subtraction Tracker due to multiple moving objects and no way to distinguish between regions in the detected foreground. This first case study was mainly a lab setup to investigate how well we could utilize the strong color of the beanie, and we saw that good results can be achieved using our Color Tracker algorithm.

Increased amount of object features enabled the trackers to perform better, where especially the adaptive trackers could more easily establish an appearance model. In turn, the classifier's were thus able to discriminative better between background and foreground. In particular, we saw Struck doing well in chapter 5, but at a high cost. Even though Struck uses a budgeting mechanism to allow for real-time operation, the workload is too great, and the gain in robustness is too expensive to justify when the execution speed is more than *400* ms above our real-time threshold. In general we have seen a trade-off between tracking robustness and execution speed. More computational complexity can be introduced to handle more cases, but this moves the tracking algorithm further away from the real-time threshold. Our own tracker implementations (OFT, CT and BGST) are specialized algorithms that are either unable to give proper results, or able to exploit the use case at hand to perform well. In the latter case, their simplicity allowed them to perform fast as well as provide a usable retrieval rate. In the former case, however, portability is sacrificed to achieve good performance. Higher overall performance across case studies would likely be achieved by combining several approaches.

We were able to implement a Color Tracker that tracked the ball in almost all the frames in the footage from Bagadus, with a satisfying level of accuracy, but we note that the case study presented is an ideal situation. If the tracking should ever lose the ball, or if the ball exits the frame and reenters, the ROI used to locate the ball would need to expand to include the whole image, likely introducing false positives. More importantly, different teams have different colored jerseys, and the ball is not always uniquely colored either.

7.3 Future work

In chapter 4, we observed that useful information for tracking can be extracted through background subtraction, similar to color filtering and the features used in the Optical Flow Tracker. These algorithms are simplistic and primitive on their own, but could easily be combined to enable higher robustness and better overall tracking. For example, one could construct a background mask to color filter only the foreground, and then combine the results with other features found in the resulting region.

All of the tracker implementations presented in this thesis were pure CPU implementations. Due to the nature of image processing, GPUs can be excellent tools to speed up the execution of an object tracker using the massive parallelism enabled by the GPUs' architecture. Additionally, we note that camera manufacturers keep on creating cameras with increasing resolutions, while the CPUs do not scale fast enough to follow this development. Additionally, proper occlusion handling has shown to be central to object tracking in challenging environments, and a solid solution to the problem is combining information from multiple cameras. This produces even more data and puts further stress on the tracker algorithms. NVIDIA's Visionworks [15] is a brand new and exciting alternative to CPUs, where an SDK is combined with the NVIDIA Tegra K1 [14] to enable the creation of fast, scalable and easy-to-use Computer Vision algorithms. This toolkit could help researchers in object tracking utilize the power of GPUs in a convenient framework, and might be a viable option for more efficient tracker implementations.

It is also important to note that the trackers have only been tested using three sequences. In the future, more match data from Bagadus can be extracted, or the system can be installed at additional locations to produce datasets that cover other use cases. Computer vision and object tracking are also very hot research topics that constantly spawn new work, meaning more trackers can be tested to gain further knowledge.

Through our work, we have seen that single object trackers can easily fail, especially due to difficult single frames that confuse the appearance model. For a tracker to work properly in Bagadus, exploiting contextual information is key. Leveraging information from ZXY such as player positioning and running direction can greatly aid the tracking. One could for example analyze the movement patterns of the players to reason about the location of the ball. Also, more cameras that cover several viewpoints can help maintain tracking during partial or full occlusions, as additional computational complexity can only do so much. This is an expensive but realistic option in Bagadus, which is a highly scalable distributed system.

A central future goal is to introduce an object tracking module into the pipeline. However, there is little value in integrating a tracker if the tracker's output is inadequate. First and foremost, future work should focus on overall tracking performance. If the needed performance is achieved, a workable automatic guidance of the virtual viewer is trivial. Also, the tracking data could be added to the match database along with the player positioning data, enabling video extraction based on the ball's movements. The system could then easily generate a number of ball statistics, such as moving patterns, ball possession percentages, average shot speed and many more. There is a plethora of useful data that can be extracted and utilized not only for the system and its viewers, but for analyzing and improving any sport where such a system can be installed. In the meantime, we acknowledge that real-time ball tracking in team sports still remains a daunting task.

Appendix A

Code

A.1 Color Tracker algorithm (for beanie tracking)

Algorithm 2 Color thresholding with ROI brightness boosting

```
while there are frames available do
    frame ← read frame from disk
    img ← yuv2rgb(frame)
    bin_img ← color threshold(img)
    blobs ← get the blobs in bin_img

    if blobs is non empty  then                          ▷ tracker has found a candidate
        DrawBlobs(bin_img,blobs)
    else                                                 ▷ try brightness boost
        roi ← 200x350 pixel rectangle around last known position of beanie
        roi_threshold ← color threshold(roi)
        blobs ← get the blobs in roi_threshold

        if blobs is non empty then                       ▷ tracker has found a candidate
            DrawBlobs(roi_threshold,blobs)
        else
            error: tracking failed for current frame
        end if
    end if
end while

function DrawBlobs(img,blobs)
    position ← GetBlobs(img)
    if position is valid then
        mark object position in the video
    else
        error: tracking failed for current frame
    end if
end function

function GetBlobs(bin_img)
    biggest ← get the biggest blob in bin_img
    if biggest is non empty then                         ▷ We found the beanie
        position ← compute average of blob's coordinates
        return position
    else
```

A.2 Background Subtraction Tracker algorithm

Algorithm 3 Background Subtraction Tracker

```
 1: while there are frames available do
 2:     frame ← read frame from disk
 3:     img ← yuv2rgb(frame)
 4:     bin_img ← backgroundSubtraction(img)
 5:     blobs ← get the blobs in bin_img
 6:
 7:     if blobs is non empty then                     ▷ tracker has found a candidate
 8:         DRAWBLOBS(bin_img,blobs)
 9:     else
10:         error: tracking failed for current frame
11:     end if
12: end while
13:
14: function DRAWBLOBS(img,blobs)
15:     position ← GETBLOBS(img)
16:     if position is valid then
17:         mark object position in the video
18:     else
19:         error: tracking failed for current frame
20:     end if
21: end function
22:
23: function GETBLOBS(bin_img)
24:     biggest ← get the biggest blob in bin_img
25:     if biggest is non empty then                   ▷ We found the beanie
26:         position ← compute average of blob's coordinates
27:         return position
28:     else
29:         return error
30:     end if
31: end function
```

A.3 Accessing the source code

Access to source code will be given upon request.

Bibliography

[1] Basler - ACE Series - ACA1300-30GC. 2014. URL: `http://www.baslerweb.com/products/ace.html?model=167&langage=en` (visited on 03/19/2014).

[2] Basler - ACE series - ACA2000-50GC. 2014. URL: `http://www.baslerweb.com/products/ace.html?model=173.` (visited on 03/19/2014).

[3] Boris Babenko, Ming-Hsuan Yang, and Serge Belongie. “Robust object tracking with online multiple instance learning”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 33.8 (2011), pp. 1619–1632.

[4] Xuefeng BAI, Qiong LI, Tiejun ZHANG, Xianhua Song, and Xiamu Niu. “A Novel Ball Tracking Method using Dynamic Kalman Filter with Two-stage Ball Search”. In: *Journal of Computational Information Systems* 9.9 (2013), pp. 3373–3381.

[5] Visual Tracker Benchmark. *Tracker Benchmark v1.0*. 2013. URL: `https://sites.google.com/site/trackerbenchmark/benchmarks/v10` (visited on 07/13/2014).

[6] Ella Bingham and Heikki Mannbjecila. “Random projection in dimensionality reduction: applications to image and text data”. In: *Proceedings of the seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2001, pp. 245–250.

[7] Antoine Bordes, Léon Bottou, Patrick Gallinari, and Jason Weston. “Solving multiclass support vector machines with LaRank”. In: *Proceedings of the 24th international conference on Machine learning*. ACM. 2007, pp. 89–96.

[8] Antoine Bordes, Nicolas Usunier, and Léon Bottou. “Sequence labelling SVMs trained in one pass”. In: *Machine Learning and Knowledge Discovery in Databases*. Springer, 2008, pp. 146–161.

[9] Michael D Breitenstein, Fabian Reichlin, Bastian Leibe, Esther Koller-Meier, and Luc Van Gool. “Robust tracking-by-detection using a detector confidence particle filter”. In: *2009 IEEE 12th International Conference on Computer Vision*. IEEE. 2009, pp. 1515–1522.

[10] Dorin Comaniciu, Visvanathan Ramesh, and Peter Meer. “Kernel-based Object Tracking”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 25.5 (2003), pp. 564–577.

[11] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. “Computing As a Discipline”. In: *Commun. ACM* 32.1 (Jan. 1989). Ed. by Peter J. Denning, pp. 9–23. ISSN: 0001-0782.

[12] Intel Corporation. *Open Source Computer Vision*. 2014. URL: `http://opencv.org/` (visited on 03/19/2014).

[13] NVIDIA Corporation. *Compute Unified Device Architecture (CUDA)*. 2014. URL: `http://www.nvidia.com/object/cuda_home_new.html` (visited on 07/23/2014).

[14] NVIDIA Corporation. *NVIDIA Tegra K1*. 2014. URL: `http://www.nvidia.com/object/tegra-k1-processor.html` (visited on 08/13/2014).

[15] NVIDIA Corporation. *NVIDIA Visionworks Computer Vision Toolkit*. 2014. URL: `http://www.nvidia.com/object/visionworks-cv-toolkit.html` (visited on 08/13/2014).

[16] Edsger. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1 (1, 1959), pp. 269–271.

[17] Vamsidhar Reddy Gaddam, Ragnar Langseth, Sigurd Ljødal, Pierre Gurdjos, Vincent Charvillat, Carsten Griwodz, and Pål Halvorsen Halvorsen. “Interactive zoom and panning from live panoramic video”. In: *Proceedings of ACM NOSSDAV*. 2014.

[18] 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 ACM Multimedia Systems Conference*. ACM. 2014, pp. 168–171.

[19] Vamsidhar Reddy Gaddam, Ragnar Langseth, Håkon Stensland, Carsten Griwodz, Pål Halvorsen, and øystein Landsverk. “Automatic real-time zooming and panning on salient objects from a panoramic video”. In: *Submitted and accepted to proceedings of the International Conference on Multimedia Systems (MMsys)*. ACM. 2014.

[20] Pål Halvorsen, Simen Sægrov, Asgeir Mortensen, David K. C. Kristensen, Alexander Eichhorn, Magnus Stenhaug, Stian Dahl, Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Carsten Griwodz, and Dag Johansen. “Bagadus: An Integrated System for Arena Sports Analytics: A Soccer Case Study”. In: *Proceedings of the 4th ACM Multimedia Systems Conference*. ACM, 2013, pp. 48–59.

[21] Sam Hare, Amir Saffari, and Philip HS Torr. “Struck: Structured output tracking with kernels”. In: *2011 IEEE International Conference on Computer Vision (ICCV)*. IEEE. 2011, pp. 263–270.

[22] Dr. Paul Hawkins. *Hawk-Eye Innovations Ltd*. 2014. URL: `http://www.hawkeyeinnovations.co.uk/page/sports-officiating/football` (visited on 05/19/2014).

[23] Espen Oldeide Helgedagsrud. “Efficient implementation and processing of a real-time panorama video pipeline with emphasis on dynamic stitching”. MA thesis. University of Oslo, 2013.

[24] Google Inc. *Google driverless car*. URL: `http://en.wikipedia.org/wiki/Google_driverless_car` (visited on 08/01/2014).

[25] International Telecommunication Union (ITU-T). *H.264*. URL: `https://www.itu.int/rec/T-REC-H.264` (visited on 07/27/2014).

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

[27] Ian Jolliffe. *Principal Component Analysis*. Wiley Online Library, 2005.

[28] Z. Kalal, K. Mikolajczyk, and J. Matas. *Tracking Learning Detection*. 2012. URL: `http://personal.ee.surrey.ac.uk/Personal/Z.Kalal/tld.html` (visited on 05/19/2014).

[29] Z. Kalal, K. Mikolajczyk, and J. Matas. “Tracking Learning Detection”. In: vol. 34. July 2012, pp. 1409–1422.

[30] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. “Forward-backward error: Automatic detection of tracking failures”. In: *2010 20th International Conference on Pattern Recognition (ICPR)*. IEEE. 2010, pp. 2756–2759.

[31] 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*. ACM. 2014.

[32] Lorenz Kellerer-Pirklbauer. “Survey of Real time Video HDR Algorithms and their implementation in CUDA”. MA thesis. Graz University of Technology, 2014.

[33] Pavel Korshunov and Wei Tsang Ooi. “Reducing Frame Rate for Object Tracking”. In: *Advances in Multimedia Modeling*. Springer, 2010, pp. 454–464.

[34] Martin Stensgård (Simula Research Labratory. *Micro trigger box*. URL: `https://bitbucket.org/mpg_code/micro-trigger-box` (visited on 07/27/2014).

[35] Ragnar Langseth. “Implementation of a distributed real-time video panorama pipeline for creating high quality virtual views”. MA thesis. University of Oslo, 2014.

[36] Gregory Ward Larson, Holly Rushmeier, and Christine Piatko. “A visibility matching tone reproduction operator for high dynamic range scenes”. In: *IEEE Transactions on Visualization and Computer Graphics* 3.4 (1997), pp. 291–306.

[37] Azure - Azure-0814M5M lens. 2014. URL: `http://www.azurephotonicsus.com/products/azure-0814M5M.html` (visited on 03/19/2014).

[38] Marco Leo, Nicola Mosca, Paolo Spagnolo, Pier Luigi Mazzeo, Tiziana D’Orazio, and Arcangelo Distante. “Real-time multiview analysis of soccer matches for understanding interactions between ball and players”. In: *Proceedings of the International Conference on Content-based image and Video retrieval*. ACM. 2008, pp. 525–534.

[39] Libevent. *Libevent - an event notification library*. URL: `http://libevent.org/` (visited on 07/27/2014).

[40] B. D. Lucas and T. Kanade. “An iterative image registration technique with an application to stereo vision”. In: *In International Joint Conference on Artificial Intelligence*. 1981.

[41] David Mills, Jim Martin, Jack Burbank, and William Kasch. “Network time protocol version 4: Protocol and algorithms specification”. In: *IETF RFC5905, June* (2010).

[42] Katja Nummiaro, Esther Koller-Meier, and Luc Van Gool. “An adaptive color-based particle filter”. In: *Image and Vision Computing* 21.1 (2003), pp. 99–110.

[43] Yoshinori Ohno, Jun Miura, and Yoshiaki Shirai. “Tracking players and estimation of the 3D position of a ball in soccer games”. In: *15th International Conference on Pattern Recognition*. Vol. 1. IEEE. 2000, pp. 145–148.

[44] VideoLAN Organization. *x264 - freesoftware library and application*. URL: `http://www.videolan.org/developers/x264.html` (visited on 07/27/2014).

[45] Svein Arne Pettersen, Dag Johansen, Håvard Johansen, Vegard Berg-Johansen Gaddam, Vamsidhar Reddy, Asgeir Mortensen, Ragnar Langseth, Carsten Griwodz, Håkon Kvale Stensland, and Pål Halvorsen. “"Soccer Video and Player Position Dataset"”. In: *Proceedings of the International Conference on Multimedia Systems (MMSys)*. 2014, pp. 18–23.

[46] Erik Reinhard, Wolfgang Heidrich, Paul Debevec, Sumanta Pattanaik, Greg Ward, and Karol Myszkowski. *High dynamic range imaging: acquisition, display, and image-based lighting*. Morgan Kaufmann, 2010.

[47] Jinchang Ren, James Orwell, Graeme A Jones, and Ming Xu. “Real-Time Modeling of 3-D soccer ball trajectories from multiple fixed cameras”. In: *IEEE Transactions on Circuits and Systems for Video Technology* 18.3 (2008), pp. 350–362.

[48] S. Saegrov, A. Eichhorn, J. Emerslund, H.K. Stensland, C. Griwodz, D. Johansen, and P. Halvorsen. “Demo: Bagadus an integrated system for soccer analysis”. In: *2012 Sixth International Conference on Distributed Smart Cameras (ICDSC)*. Oct. 2012, pp. 1–2.

[49] Koichi Sato and Jake K Aggarwal. “Temporal spatio-velocity transform and its application to tracking and interaction”. In: *Computer Vision and Image Understanding* 96.2 (2004), pp. 100–128.

[50] Basler - Basler Pylon SDK. 2014. URL: `https://www.baslerweb.com/Downloads-Software-43868.html?type=20&series=1&model=173` (visited on 03/19/2014).

[51] Jianbo Shi and Carlo Tomasi. “Good Features to Track”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. 1994, pp. 593–600.

[52] Dolphin Interconnect Solutions. *IXH610 Host Adapter*. 2014. URL: `http://www.dolphinics.com/products/IXH610.html` (visited on 03/19/2014).

[53] Dolphin Interconnect Solutions. *IXS600 Switch*. 2014. URL: `http://www.dolphinics.com/products/ISX600.html` (visited on 03/19/2014).

[54] H. K. Stensland, V. R. Gaddam, M. TennÃ¸e, E. Helgedagsrud, M. NÃ¦ss, H. K. Alstad, C. Griwodz, P. Halvorsen, and D. Johansen. “Processing Panorama Video in Real-Time”. In: *International Journal of Semantic Computing (IJSC)* 8.2 (2014), pp. 1–19.

[55] Håkon Kvale Stensland, Vamsidhar Reddy Gaddam, Marius Tennøe, Espen O Helgedagsrud, Mikkel Næss, Henrik Kjus Alstad, Asgeir Mortensen, Ragnar Langseth, Sigurd Ljødal, øystein Landsverk, Carsten Griwodz, and Pål Halvorsen. “Bagadus: An integrated real-time system for soccer analytics”. In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMCCAP)* 10.1s (2014), p. 14.

[56] Marius Tennøe. “Efficient implementation and processing of a real-time panorama video pipeline with emphasis on background subtraction”. MA thesis. University of Oslo, 2013.

[57] Marius Tennøe, Espen O 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: *International Symposium on Multimedia*. IEEE, 2013, pp. 76–83.

[58] ZXY Sport Tracking. 2014. URL: `http://www.zxy.no/` (visited on 03/19/2014).

[59] Ioannis Tsochantaridis, Thorsten Joachims, Thomas Hofmann, and Yasemin Altun. “Large margin methods for structured and interdependent output variables”. In: *Journal of Machine Learning Research*. 2005, pp. 1453–1484.

[60] Cor J Veenman, Marcel JT Reinders, and Eric Backer. “Resolving motion correspondence for densely moving points”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23.1 (2001), pp. 54–72.

[61] Xinchao Wang, Vitaly Ablavsky, and Pascal Fua Horesh Ben Shitrit. *Take Your Eyes off the Ball. Improving Ball-Tracking by Focusing on Team Play*. Tech. rep. Computer Vision Laboratory - School of Computer and Communication Sciences: Swiss Federal Institute of Technology, Lausanne (EPFL), Mar. 20, 2013.

[62] Greg Welch and Gary Bishop. *An Introduction to the Kalman Filter*. Department of Computer Science, University of North Carolina, 1995.

[63] Wikipedia. *HSL and HSV*. 2014. URL: `http://en.wikipedia.org/wiki/HSL_and_HSV` (visited on 05/21/2014).

[64] Kesheng Wu, Ekow Otoo, and Kenji Suzuki. "Two strategies to speed up connected component labeling algorithms". In: (2008).

[65] Yi Wu, Jongwoo Lim, and Ming-Hsuan Yang. "Online Object Tracking: A Benchmark". In: *2013 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2013, pp. 2411–2418.

[66] Alper Yilmaz, Omar Javed, and Mubarak Shah. "Object Tracking: A Survey". In: *ACM Comput. Surv.* 38.4 (Dec. 2006).

[67] Wikipedia - YUV. 2014. URL: `http://en.wikipedia.org/wiki/YUV` (visited on 01/16/2014).

[68] Kaihua Zhang, Lei Zhang, and Ming-Hsuan Yang. "Real-Time Compressive Tracking". In: *Computer Vision - ECCV 2012*. Ed. by Andrew Fitzgibbon, Svetlana Lazebnik, Pietro Perona, Yoichi Sato, and Schmid. Vol. 7574. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 864–877.

[69] Yi Zhang, Hanqing Lu, and Changsheng Xu. "Collaborate ball and player trajectory extraction in broadcast soccer video". In: *19th International Conference on Pattern Recognition, 2008. ICPR 2008.* IEEE. 2008, pp. 1–4.

[70] Wei Zhong, Huchuan Lu, and Ming-Hsuan Yang. "Robust object tracking via sparsity-based collaborative model". In: *2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. 2012, pp. 1838–1845.

[71] Zoran Zivkovic. "Improved Adaptive Gaussian Mixture Model for Background Subtraction". In: *Proceedings of the 17th International Conference on Pattern Recognition (ICPR)*. Vol. 2. IEEE. 2004, pp. 28–31.