Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Automated Visual Foosball Tracking

Bachelor's Thesis

Linus Baumberger

lbaumberger@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Till Aczel, Joël Mathys
Prof. Dr. Roger Wattenhofer

December 30, 2024

Contents

Abstract **1**

1 Introduction **2**

2 Methodology **4**

2.1 Phase 1, Hardware Phase: 5

2.2 Phase 2, System Phase: 10

2.3 Phase 3, Analysis Phase: 13

3 Results **21**

3.1 Phase 1, Hardware Phase: 21

3.2 Phase 2, System Phase: 23

3.3 Phase 3, Analysis Phase: 26

4 Future Work **29**

5 Conclusion **30**

Bibliography **31**

A Documentation **A-1**

A Hardware **1**

A.1 Camera Mount 1

A.1.1 Aluminium Frame 2

A.1.2 Connector 3

A.1.3 Angle 35° 4

A.1.4 Side Braces 5

A.1.5 Wooden Strip 6

A.2 Cameras 7

A.3 Elumination 8
A.4 Control Unit/ Raspberry Pi 9

A Software **10**
A.1 Project Structure 11
A.2 Raspberry PI 12
A.2.1 Setup 12
A.2.2 Overview 12
A.3 API Raspberry Pi 13
A.3.1 Authentication 14
A.3.2 Endpoints 14
A.3.3 Error Handling 15
A.4 VM 15
A.4.1 Setup 16
A.4.2 Camera Control 16
A.4.3 Communication RTSP / Receiving Streams 16
A.4.4 Processing 17
A.4.5 Distributing 17
A.5 API VM 17
A.5.1 Authentication 17
A.5.2 Endpoints 17
A.5.3 Error Handling 21

Abstract

This thesis will try to automate the visual tracking of a foosball. The goal is to analyse the game's state at runtime, while also providing a livestream.

In a first step we install all the necessary hardware to record a game. On top of that, we build a system that records, saves, processes/analyses, and livestreams the game.
To analyze the videostreams we compared classical visual computing algorithms and YOLO (a machine learning model, designed for object detection).

The resulat was a system that records and streams foosball games. The tested algorithms are working, but could not yet be implemented to run at runtime. The system is built in a modular way (different APIs and divided into subsystems), so that it can be easily extended or integrated into a future project.

Chapter 1
Introduction

Many of us enjoy playing a game of foosball from time to time. And often we ask ourselves if we could do better. If we just had the chance to rewatch the most capitvating situations of the last game. Maybe we just want to replay a certain scene to determine whether we scored or not. And image how cool it would be to have some statistics about our gameplay. Maybe we can even make it so far that we can predict if a team scores a goal in the next few seconds.
As a foosballplayer myself I was very motivated to be part of a project that tries to achieve that.

The goal of this project was to develop a visual tracking system for foosball.Once we have automated the process to keep track of the visual game play, we can predict and analyse many things, such as automatically keeping track of the teams' scores. Or we could analyse which team has more ball possession and we could analyze how fast some of the shots are.
To achieve all of this, we first have to overcome a few challenges.

The project divides into three phases.
We start with phase one, the hardware phase, where we design and install everything related to hardware.
After that in phase two, the system phase, we develop a system to record, save, and livestream the game.
And finally in phase three, the analysis phase, we want to collect data and create statistics from our streams.
In phase one, we first of all had to construct a cameramount and think about how many cameras we need to record the whole foosballtable. Because a foosball can be as fast as 10 metres per second, we should have a framerate high enough (around 80fps) to allow fluent replays of certain scenes. Another requirement was that the camera should record in HD. As soon as we have a hardware setup that is working and functional, we can focus on the software.

Some challenges that we have to resolve software wise are: Those recordings should be saved (for additional analysis later) and at the same time we wanted

a livestream (with a latency as low as possible, less then 10 seconds). Besides that we also have to think about algorithms that detect and track the foosball. Relying on machine learning-based or more classical computer vision methods.

Even more interesting and useful would it be, if we could use those algorithms at runtime. As soon as we want to do that, we have to think about the throughput of our alogrithms. How could we improve that without losing quality? This should include the detection of potential goal chances and goals as well in a second step include balltracking. This would allow us to do some statistics and predictions of the game.

Chapter 2

Methodology

We divided this project into 3 phases (Figure 2.1). Phase 1 is the hardware phase where we focused on building a setup that was functional and provided us with a foundation on which we can build up. We had to be careful that we built a rigid and robust setup so it would absorb all the physical stress induced on it.
In phase 2 (the system phase) our system phase we designed and developed a (software) system that provided a solution for recording, saving and streaming of the foosballgames. One thing that we have to keep in mind is that our goal is to achieve a latency as low as possible while still maintaining a high framrate and resultion (80fps@1080x720).
In the third phase, also called the analysis phase, we focused on testing algorithms to detect foosball. Our focus is on low latency and maximal throughput while still maintaining a high quality of detection. Those algorithms must be integrated into our processing pipeline. A task that will certainly be challenging.

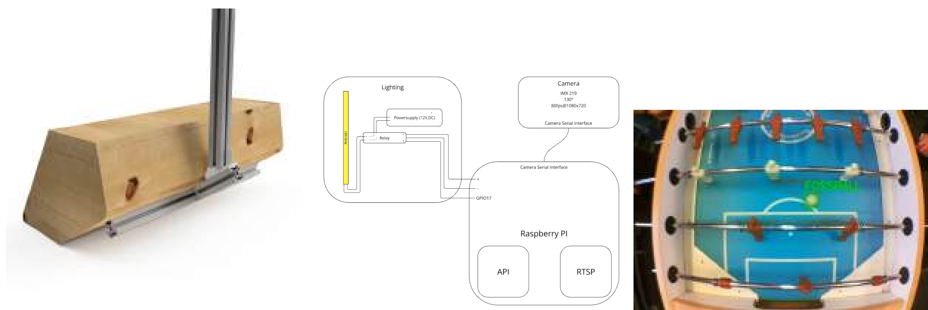

Figure 2.1: We split the project into three different phases. The first phase (left) provides the necessary hardware. In the second one (middle) we design a working system to record, save and livestream foosballgames. The third phase (right) focusses on analysing foosballgames.

2.1 Phase 1, Hardware Phase:

Here we did all the things that are hardware related. At first we just had a foosballtable. We had to install a cameramount and cameras that we can record the whole table. The cameramount had to be robust to disturbances/vibrations. Because a foosball can achieve high speeds of up to 10m/s, we have to record everything at high framerates.

This phase included planning, developing, and building the cameramount. We tested different constructions and materials to create a mount that is stable enough to fulfill its purpose but at the same time as unobtrusive as possible. That means we had to find a solution on how to mount the cameramount. We first thought about fixing it to the score counter, but we soon realized that it would be a disturbance for the players and also not be that rigid. Especially for movements along the y-axis (Figure 2.2).

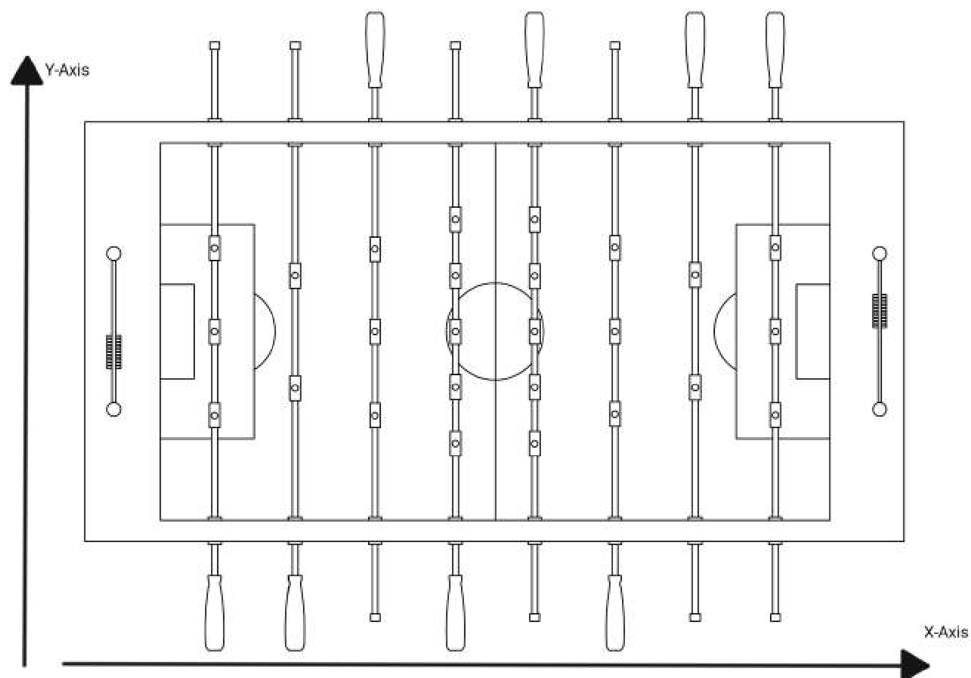

Figure 2.2: Schematic Raspberry Pi and VM

We found a better way to mount the cameramount onto the foosball table with the help of the already existing holes in the side wall of the foosballtable.

As can be seen in Figure 2.3, we had to use different kinds of materials. We had to find a compromise between the weight and the robustness of the material. While always have in mind on how we want to process/work on that material.

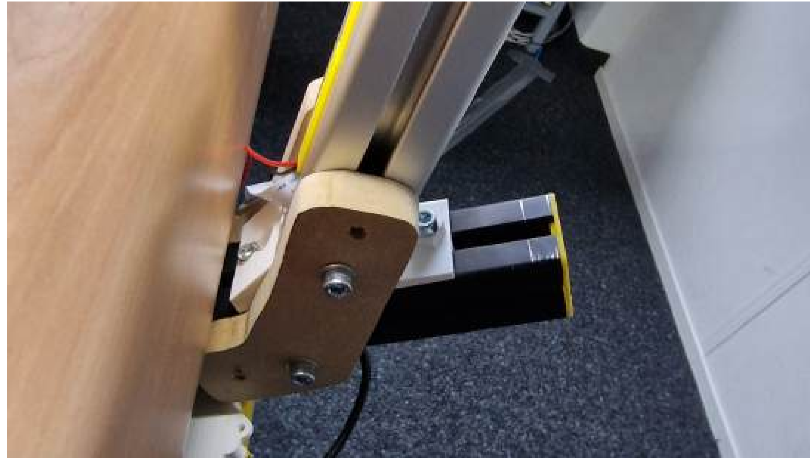

Figure 2.3: Different Materials: Plastic for the 35° angle, wood for the side braces and aluminium for the extrusion

For example, we first had plastic side braces but soon realized that plastic has the tendency to flex too much. Therefore, we switched to a wooden side brace that offered much more stability and stiffness. Another example would be the 35 ° angle that is currently made out of plastic. We know that we could improve the stiffness of the mount even more if we had a metal or wooden angle. But with our limited capabilities and possibilities to process metal and wood we just cant produce such an angle.
In addition to that, we also had to decide what kind of cameras we needed and how we wanted to control and mount those cameras. Because a foosball can have a speed of up to 10m per second, we have to record at a framerate as high as possible. Our generic cameras with an IMX219 sensor can record HD at 80 fps. To protect the cameras against foosballs, we printed small plastic cases for the cameras.
Another question that had to be resolved in this phase was about lighting. How could we minimze shadows and guarantee a uniform lighting regardless of the time of day or room lighting. We decided to use an led strip. Such a strip allows us to uniformly light up the foosball table (Figure 2.4).

The first thought was to control and power the led strip directly from the raspberry pi. We soon realized that this would potentially use too much power from the raspberry pi. So we decided to power the led strip using a separate 12v power supply and a relay to control it via the raspberry pi. The relay was placed in a 3d printed box and fixed with screws to the aluminium extrusions (Figure 2.8). All the cables coming from the wooden strip are directed into the extrusion and then covered by 3d printed PLA covers and later redirected along the side of the foosball table into the table. With the help of zipties and some 3d printed mounting brackets (screwed into the table), we provided strain relief (Figure 2.7)

Figure 2.4: LED Strip switched on and two cameras (yellow cases) mounted to the wooden strip

for all the cables. Inside the table, we stored both powersupplies (the one for the pis (Figure 2.6) and the one for the led strip (Figure 2.5)).

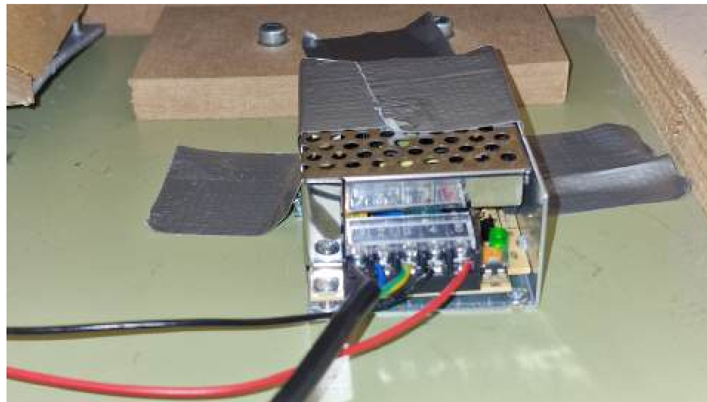

Figure 2.5: Powersuplly for the led strip, mounted inside the foosball table

Figure 2.6: Powersuplly for the raspberry pis (65W), mounted with the help of a 3d printed holder to the leg of the foosballtable

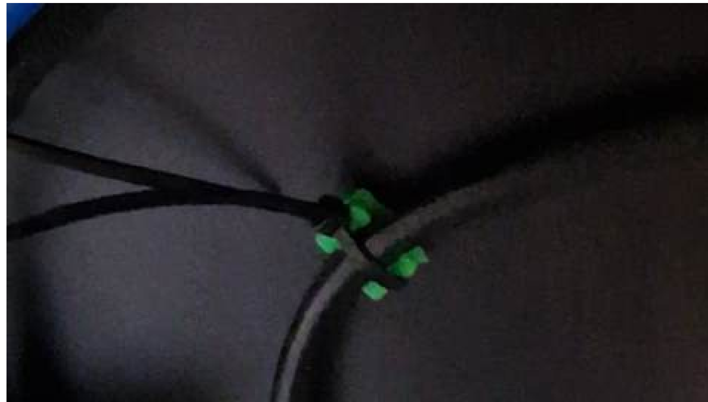

Figure 2.7: 3d printed mounting bracket, screwed into the foosball table, the cable is fixated to the mount with a zip tie, to provide strain relief for the cable

Another important decision we had to take was regarding the kind of hardware we want to use to control the cameras. Because Raspberry Pis are easily available, have already the necessary interfaces and hardware to process camerastreams and offer a big community support, we decided to use them. At the beginning we planned to use some older Raspberry Pi 3B, but we soon realized that they lack the performance to handle streams of such a high framerate. So we decided to switch to the newer and more powerful Raspberry Pi 5 (Fiugre 2.9).

Figure 2.8: Raspberry Pi 5 connected with 3 cables to the relay (yellow box below) which will control the led strip

Figure 2.9: Raspberry Pi 5 (with lan-cable, power-cable and csi-cable) mounted to the alumium extrusions

To power the whole system we used a single power cable which we taped together with the lan cables (which provide a reliable and fast internet conenction to the pis) to the ground (Figure 2.10).

Figure 2.10: Foosballtable with installed cameramount, raspberry pis are installed onto the aluminium extrusions and all the cables (one powercable, two ethernet cables) are properly taped to the ground

2.2 Phase 2, System Phase:

As soon as we had the hardware we had the foundation of our project. From now on we could focus on everything software related. To structure the development of that project we first focused on creating a minimal working system. This system should record, save, and livestream the foosballgames. On top of this system, we could later implement the analysis of the games or even further down the road we could build further projects on top of this. For this reason, we have to code as modular as possible so that in the future the system could be easily expanded.

We first wanted to simply record games with the raspberry pi and our cameras. We bought generic cameras with an IMX219 sensor and a 130° fisheyelens. This camera is capable of recording at 80 frames per second at a resolution of 1080x720. Unfortunately, it did not work out of the box. We first had to configure the "boot/Firmeware/config.txt" file with the specific information of our generic camera. Only after we did that the raspberry pi recognized those generic cameras. This was quite a tedious task which we later automated (on execution of the setup_pi.py file) to make it easier to setup and integrate a new pi. Using a raspberry pi allowed us to use the picamera2 liberary. This library is the official Python library for interfacing with the Raspberry Pi Camera Module. Because we wanted that specific fps and resolution we were dependent on a certain camera mode that was able to provide that. To use that mode we had to dive a little bit deeper to configure the camera correctly. Instead of using the default functions and configurations, we had to do our own camera configuration (Figure 2.11.

Another question that arose was on how we want to design the whole system. How should the subsystems interact with each other. For example, we first used

```
#Camera configurations
config = picam2.create_video_configuration(
    sensor={"output_size": (1640, 1232), "bit_depth": 8},
    controls={"FrameRate": 80},
    main={"size": (1080, 720)},
    lores={"size": (640, 480)}
)
```

Figure 2.11: Cameraconfiguration, we select the specific mode and further specify the framerate

a simple tcp connection to transport the stream from the Pi to the VM. Later we realized that we could improve latency and modularity if we used an rtsp server (running on the pi)[1]. This would allow us to distribute the stream not only to the vm but also to other systems.

Currently, the raspberry pi can record and distribute a stream. To be really useful for us we also need a way to control the pi and to give some kind of start and stop signal (Figure 2.12). For this purpose, we designed an API for the raspberrry pi (The API is described in the documention).

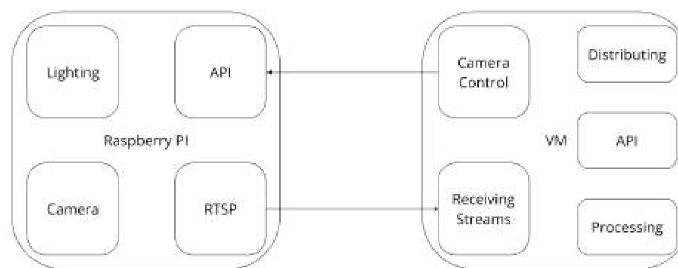

Figure 2.12: An external system for example our VM controls the Raspberry Pi via an API. The Raspberry Pi is responsible for recording and distributing the stream as an rtsp stream

The next step would be to create a system that will receive both streams and save them. To do that, we first had a really simple script using opencv [2], which receives the frames and saves them to an mp4 file. This is already nice and would give us the opportunity to analyze games retrospectively.
But initially we planned to provide some kind of livestreaming. Our system should more look something like Figure 2.13. First we tried to do that with opencv [2] but we soon realized that even though it "works" it does not provide satisfiable results. The throughput we achieved was just to low and resulted in a delayed and very jittery stream. After some research, we decided to use the ffmpeg library. Ffmpeg is an open-source mlultimedia framework for handling video, audio, and streams. It supports a wide range of formats and already includes tools for encoding and decoding, as well as for filtering. Another advantage was that it also supports complex pipelines for processing (Figure 2.14)[3]. This al-

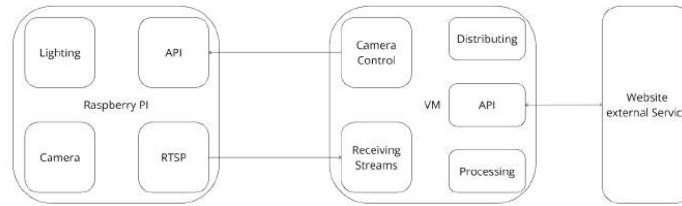

Figure 2.13: We want to receive the stream from the pis and then (after we cropped both videos together) redirect it to another service (e.g. a website) where people can watch the foosballgame live

lowed us to receive both rtsp streams and then save both original inputs while we could still crop those two inputs together and create an hls stream from that cropped together video (Figure 2.15). All this while still maintaining a framerate of 80fps and a latency of less than 10 seconds.

To display that livestream, we developed a webpage (Figure 3.6 to watch the livestream. In addition, we develop a simple web interface for debugging and testing (Figure 3.5). To control the VM from this web interface we introduced an API for the VM (described in the documentation). The API and that livestream would also allow us to link this system with another external project.

To automate everything and make it easy to use, we created scripts to setup the environments and the corresponding scripts on the VM and Raspberry Pis. This also created services on the VM and the Raspberry Pis, which are responsible to automatically pull the newest version of the "build" branch from our gitlab repository and start the systems on reboot or restart the systems in case of an error at runtime.

The system was complete and functioning. At least that is what we thought. After a few weeks we got the warning that we are soon running out of storage on the VM. To fix that we installed an external HDD and created a cron job for the "move_old_files.sh" script. Once every day we run this script and the script moves all recordings and streams older than seven days to the external HDD.

Another problem that occured in the system phase but is rather related to the hardware phase were random crashes of our raspberry pis.The logs did not give us any reasonable error. So we had to dig deeper. In the end, we discovered that our first powersupply for the pis had voltage drops (probably because we pulled to much power). Those voltage drops caused the random crashes of the pi. After replacing the powersupply with a more powerful one it worked without any problems.

```
100     # Define the command as a list of arguments
101     cmd = [
102     "ffmpeg",
103     "-i",
104     rtsp_url1,                      # Input file 1 (RTSP stream)
105     #filename1,
106     "-itsoffset", "0.59",
107     "-i",
108     #filename1,
109     rtsp_url2,                      # Input file 2 (RTSP stream)
110     "-filter_complex",
111     (
112         "[0:v]lenscorrection=k1=-0.2:k2=0.0,transpose=1[v0];"
113         "[1:v]lenscorrection=cx=0.55:cy=0.5:k1=-0.22:k2=0.0,transpose=2[v1];"
114         "[v0][v1]hstack=inputs=2:shortest=1[v]"
115     ),
116     "-map", "[v]",                        # Map the filtered output for HLS
117     "-f", "hls",                          # HLS format
118     "-hls_time", "1",                     # Segment duration
119     "-hls_list_size", "0",                # Keep all segments
120     "-start_number", "0",                 # Start segment numbering at 0
121     "-vcodec", "libx264",                 # Use H.264 codec
122     "-preset", "ultrafast",               # Fast encoding
123     "-tune", "zerolatency",               # Low latency tuning
124     "-g", "160",                           # GOP size
125     output_file,                          # Output HLS file
126     #output,
127
128     # Save a copy of the first input stream
129     "-map", "0:v",                        # Map video from input 1
130     "-c:v", "copy",                       # Do not re-encode
131     output1,                              # Save copy of input 1
132
133     # Save a copy of the second input stream
134     "-map", "1:v",                        # Map video from input 2
135     "-c:v", "copy",                       # Do not re-encode
136     output2                               # Save copy of input 2
137  ]
```

Figure 2.14: Ffmpeg command to receive two rtsp stream, those orignal streams are saved as output1 and output2. At the same time we take those orignal streams correct the fisheyelenseffect, crop them together and save it as output_file (an hls stream)

2.3 Phase 3, Analysis Phase:

This phase introduced analysis of the game's state to this project.

With completing the system phase we now have a system into which we we can build further functionality. In this phase it is the goal to develop a system capable of trakcing a foosball, calling out (potential) goals and do some simple statistics with the data collected. In a further phase/different project, this functionality could be extended even more.

Before we even started thinking about detecting the ball everywhere, we wanted to implement a system that calls out goals and goal chances. To do that, we thought about doing some kind of motion detection in the goal zone. We ended up with two zones. The outer and bigger one for potential goals, the smaller one for actual goals (Figure 2.16). For each zone we do a background subtraction over the average of the last 500 frames (at 80fp, that is, around the last 6 seconds).

Figure 2.15: All our three outputs, from the left to the right we save the orignial input of the camera stream, we do the same for the second input and we do a fisheyelenscorrection and crop those videos togehter. The cropped together version gets distributed as an hls stream

After that background subtraction, we do some postprocessing. An elliptical 5x5 kernel is created for morphological operations. The mask is cleaned of noise using opening and dilation fills the gaps to make the object more cohesive. We then count the pixel change, if the change is over a certeain threshold we assume that a (potential) goal happened. At the beginning, we got to many false positives because of the goalkeeper moving very fast. After adjusting the threshold, we could reduce the number of false positives drastically. The number of false negatives remained small.

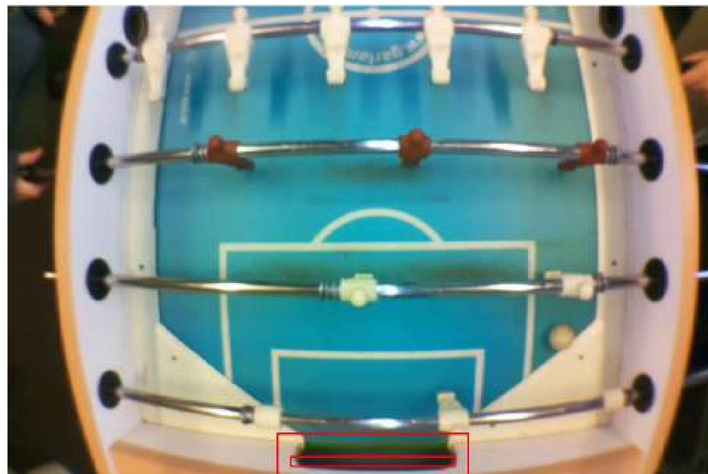

Figure 2.16: The outer and bigger zone checks for potential goals, the smaller one for actual goals.

After that we started to test different algorithms for balltracking. Initially we focussed on classical computer vision algorithms. One of our first ideas was to use template matching. This required us to take some example pictures of a foosball and match them against every frame. With this approach, there were a few problems. It was computationally quite expensive and even worse was that due to the high speed of the foosball (the foosball became "stretched" on the

videos) it often failed exactly when there was a (potential) goal (Figure 2.17).

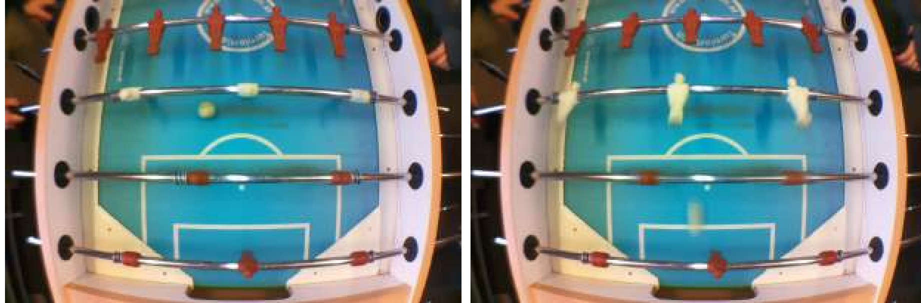

Figure 2.17: On the left side there is a still foosball. It is quite round and would be detectable with template matching. On the right we have a very fast foosball. It is so fast that it appears stretched on the image. To detect a foosball on such an image is much more difficult

Another algorithm that we tried but later discarded calculated the optical flow. One of the flaws there was that in order that the algorithm worked, we had to define the foosball's initial position. A task that generally does not sound that difficult. We could assume that every game starts with the foosball in the middle. This would certainly be possible to implement. But how would we handle a situation where we lost track of the foosball? How could we find his position again? This would require us to have a secondary algorithm that is able to detect the foosball in such a situation. But why should we still use an optical flow algorithm if we have a different algorithm that allows us to track the foosball at every possible point of time.
While thinking about such an algorithm, we remembered how we do the (potential) goal detection. We could just subtract the average background, and this would leave us with a selection of objects that moved in the last few seconds. As we did with the goal detection, we first do a background subtraction over the average of the last 500 frames and then add some postprocessing. Again we use a 5x5 elliptical kernel is created for morphological operations. The mask is cleaned of noise using opening and dilation fills the gaps to make the object more cohesive.

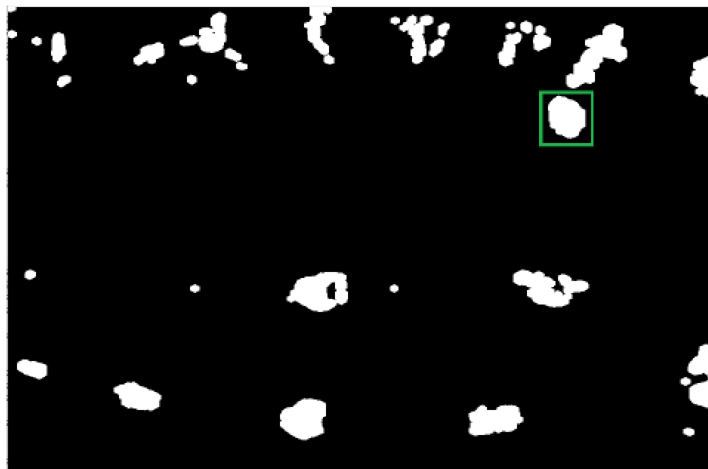

Figure 2.18: This is the frame after background subtraction and postprocessing (5x5 elliptical kernel, opening and dilation). We see all the objects that moved in the last few seconds. All the foosballplayers and also the foosball(green rectangel)

Now with the help of that background subtraction we reduced our possibilities down to a handfull, but how could we differentiate between a player and the foosball. What we certainly can do is ignore all detected objects with an area too small or too large to be a foosball. Furthermore, when we look briefly at Figure 2.18 we see that even though the foosball is not perfectly round, it is still "rounder" than the rest of the objects. So our next step was to check all our remaining possibilities for their cirularity. For that purpose, we used a helper function (Figure 2.19) which calculates the circularity of a contour using the formula $circularity = \frac{4\pi * area}{perimeter^2}$ (where area is the area of the object and perimeter is the total length of the boundary of the shape), which measures how close the shape is to a perfect circle. The function returns True if the circularity is within the range 0.7 to 1.2, indicating that the object is roughly circular. If we had a perfect circle, the formula would give us 1. If we have too many false positives, we could increase the lower bound of that range.
If we use that additional helper function, we can drastically reduce the amount of false positives. But we still have them from time to time. For example, if a foosballplayer looks a little bit too much circular after the background subtraction. Another problem that occurs is that if the ball is close to a foosballplayer, we have sometime difficulties to detect it properly (Figure 2.20). Generally speaking, it provides good results and also is accurate regarding counting goals.

We thought that we could reduce the faulty detection of foosballplayers if we additionaly added some colortesting. Theoretically, this should help us to determine whether we detect a player or a foosball. While we had some minor improvement, we also had to realize that that slowed down our code quite a bit. If we wanna use that algorithm at runtime, that would make quite a difference

```python
# Helper function to calculate circularity
def is_circular(contour):
    perimeter = cv2.arcLength(contour, True)
    area = cv2.contourArea(contour)
    if perimeter == 0:
        return False
    circularity = 4 * np.pi * (area / (perimeter * perimeter))
    return 0.7 < circularity < 1.2  # Range for circular objects
```

Figure 2.19: This function calculates the circularity of a contour using the formula $circularity = \frac{4\pi * area}{perimeter^2}$ (where area is the area of the object and perimeter is he total length of the boundary of the shape), which measures how close the shape is to a perfect circle. The function returns True if the circularity is within the range 0.7 to 1.2, indicating the object is roughly circular.

Figure 2.20: As on the left we normally detect the foosball , it can happen sporadically that we mistake a foosballplayer for the foosball (middle) or as on the right we do not detect the foosball because it is to close to a foosballplayer

whether we could process 30 or 50fps.

With the idea to further improve the accuracy of our balldetection, we looked at YOLO. YOLO is a state-of-the-art object detection algorithm that detects and classifies multiple objects in an image or video in a single forward pass. YOLO divides the image into a grid and predicts bounding boxes and class probabilities simultaneously for each grid cell, making it extremely fast and suitable for real-time applications.

We first tried the pretrained models from ultralytics. YOLO has different versions, we tested YOLOv8n and YOLO11n [4, 5]. This is the 8th and 11th generation. The letter n means that we use the "nano" model (there are also s (small), m (medium), l (large) and x (extra large)). The nano model is design for fast interference (which is exactly what we need if we want to do the analysis at runtime). All of those models include a class called sportsball. Our hopes were shattered, we had absolutly no detections at all with the class sportsball. So we had to take a few more extra steps to make YOLO work.
We decided to create our own set of training data to fine-tune an already-pre-trained model. Using cvat.ai, a free tool to annotate pictures, we annotated a total of 430 pictures (Figure 2.21). Those pictures where selected with a focus on edgecases (e.g., the foosball is partially hidden behind the player).

Figure 2.21: CVAT.ai is a free to use tool to annotate pictures efficently

To train our model, we used all those annotated pictures as training and validation data. First, we wanted to train the model, locally on a CPU. Running YOLO on a CPU is possible, but unfortunatly not that fast. We were able to process/ analyze around 10frames per second. Training those models would have taken hours. That is why we decided to create a Jupyter Notebook on Google colab, which allows us to run that training on a gpu. Running it on a GPU was more than 10 times faster. We trained our models for different amount of epochs (10,50,100 and 200)(Figure 2.22, Figure 2.23).

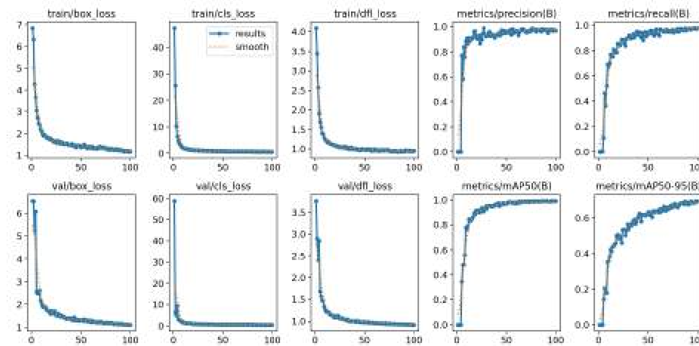

Figure 2.22: Results of YOLOv8n trained for 100 epochs on 430 pictures (used as training and validation)

The only drawback is the slow speed. If we run it on a cpu, we get not much more than 10 frames per second. We get almost 100 fps if we run it on a gpu (we used an L4 Tensor Core GPU provided by Google colab). So, theoretically it would be possible to use YOLO to analyze the frames at runtime if we have access to a gpu. To analyze prerecorded video files we also suggest doing it on a gpu. For that reason, we provide in the gitlab repo another Jupyter Notebook (which can be run on the google colab) to analyze the videos (Figure 2.24.

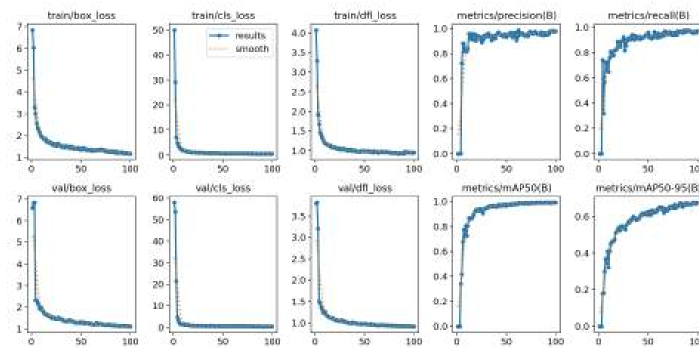

Figure 2.23: Results of YOLO11n trained for 100 epochs on 430 pictures (used as training and validation)

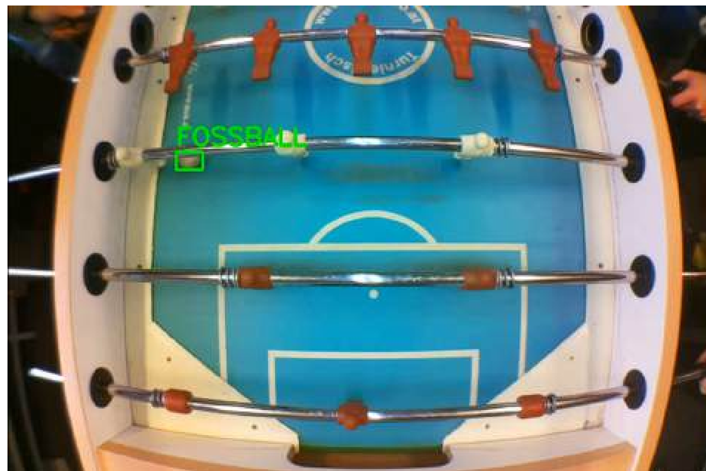

Figure 2.24: The balldetection with YOLO was more accurate and fossbals were detected even if they are fast or partially obscured

After having satisfiable results with the goal detection and the detection of the foosball, we wanted to move on and tried to integrate those algorithms into our processing pipeline.
There are a few possibilitis. One thing that we tried was writing all the frames to a pipe, processing all the frames, writing it to another pipe, and then creating the hls stream. Because we have a lot of writing and reading here it was just too slow.
Another approach would be to write our own ffmpeg filter. This would allow us to do the processing in the ffmpeg framework/pipeline. Writing an own ffmpeg filter includes that we first download the ffmpeg github repository. We can now add our own (filter) files and have to register them correctly so they are compiled when we build ffmpeg. The following document was of great help: https://github.com/FFmpeg/FFmpeg/blob/master/doc/writing_filters.txt

It was possible to create new filters, but as soon as we wanted to enable the use of opencv we got issues while compiling (opencv is in c++, while ffmpeg is in c)(Figure 2.25, Figure 2.26. After much trial and error this error is still not resolved.

```
lbaumberger@ee-tik-vm062:~/FFmpeg$ ./configure --enable-libopencv
ERROR: opencv4 not found using pkg-config

If you think configure made a mistake, make sure you are using the latest
version from Git.  If the latest version fails, report the problem to the
ffmpeg-user@ffmpeg.org mailing list or IRC #ffmpeg on irc.libera.chat.
Include the log file "ffbuild/config.log" produced by configure as this will help
```

Figure 2.25: To configure FFMPEG to enable opencv does not work. For further information we have to look at the log files

```
/usr/include/opencv2/core/cvdef.h:773:4: error: #error "OpenCV 4.x+ requires enabled C++1
1 support"
  773 | #  error "OpenCV 4.x+ requires enabled C++11 support"
      |     ^~~~~
/usr/include/opencv2/core/cvdef.h:779:10: fatal error: array: No such file or directory
  779 | #include <array>
      |          ^~~~~~~
```

Figure 2.26: In the logs we can see that there is problem while compiling because some code (opencv) is written in c++ while the rest of ffmpeg is written in c

A third approach would be to use opencv to receive and process the streams.While trying that out, we had to realize that opencv is overwhelmed with the high framerate of 80fps.

In the future, it should be the priority number one to find a way to use those algorithms at runtime. This would allow us to start creating some statistics about the game's state and maybe even make some predictions about the future state.

Chapter 3

Results

3.1 Phase 1, Hardware Phase:

We achieved a rigid and robust cameramount (Figure 3.1, Figure 3.2). The mount is unobtrusive and does not bother players while playing. The aluminium extrusions and the wooden strip on top allow for great modularity in case of future upgrades or projects. The whole electronic is properly secured and stuffed away under the table. There are only the two ethernet cables and one power cable. Those cables are secured with tape to the ground so no one trips over it. In addition, we installed strain relief for all cables.

Figure 3.1: Cameramount installed onto the foosballtable

Figure 3.2: CAD Drawings of the Cameramount and Foosballtable

We have two generic Cameras with an IMX219 sensor (80fps at 1080x720). Each camera is controlled by a Raspberry Pi 5. Those two Raspberry Pi 5 are connected via ethernet cable to the ETHZ network and are powered by a single 65W powersupply. One of the Raspberry Pis controls a relay (GPIO17). This relay switches the led strip on and off. The led strip (12V) is powered by an additional powersupply (Figure 3.3.

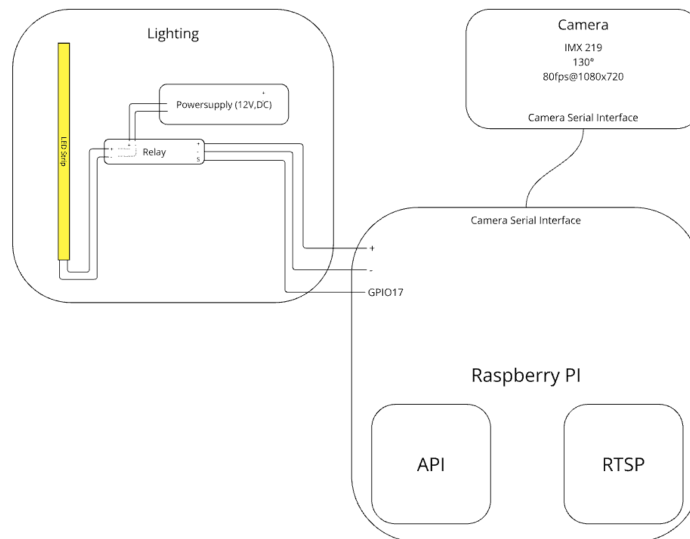

Figure 3.3: Schematic on how the Raspberry Pi controls the LED Strip and the Camera

3.2 Phase 2, System Phase:

We achieved a system that reliably records and streams a foosball game with a latency of less than 10 seconds.

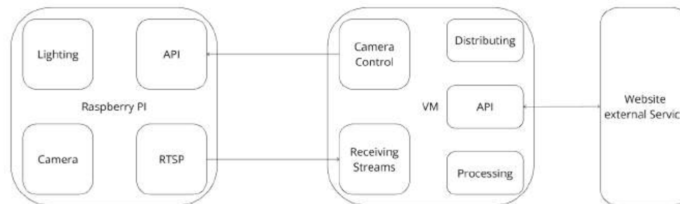

Figure 3.4: Schematic on how the Raspberry Pi the VM and the Website interacti with each other

The Raspberry Pi provides a rtsp stream and is controlled by the VM over an API[1]. The VM receives the stream and does some minimal processing (fisheye-lenscorrection and cropping the two streams together) before distributing it as an hls stream. The VM itself is controlled via an API. The stream is received by a website (Figure 3.4).
The API of both the Pi and the VM allows one to integrate this project into other projects.
The system is to a large part automated. It automatically pulls the newest version of the code from gitlab and automatically starts on reboot. In case something crashes, the system should automatically be restarted. If this does not work properly for some reason, we have a simple restart button on the debug page to manually restart everything.

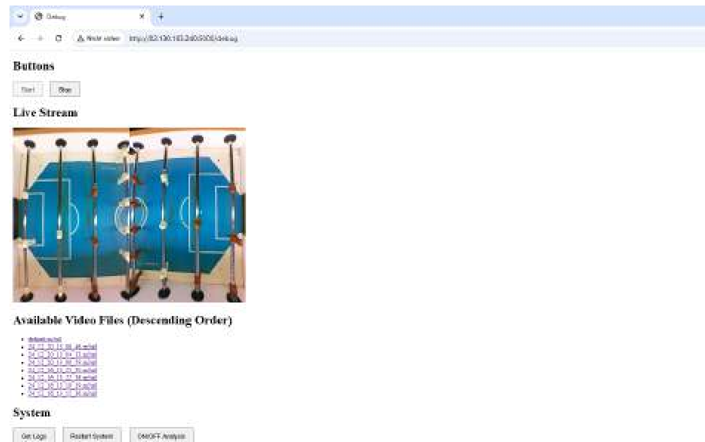

Figure 3.5: Debugging webpage, to start and stop streams, the current livestream is displayed and there is the possibility to watch already recorded games (from the last week). At the bottom, there are additional buttons to get the logs, restart the whole system or switch on and off the analysis

This webinterface has two parts. Most important for the user is the livestream (http://82.130.103.240:5000/)(Figure 3.6), which we provide. For development and debugging, there also exists a debug page (http://82.130.103.240:5000/debug) (Figure 3.5).

Figure 3.6: Webpage to watch the livestream (latency is less then 10 seconds)

Our system uses the following resources. If there is no livestream active, we use around 1.2G out of 19.5G memory available and the CPU has an average load of arround 25%. If a livestream is active, we use around 1.3G of memory and the average load on the CPU is 66% (Figure 3.7). The latency of our stream is as followed. From the raspberry pi to the vm we hava a latency of roughly 2 seconds. From the VM to the website (includes minimal processing) we have a latency of around 5 seconds. So the latency in total is around 6 to 8 seconds.

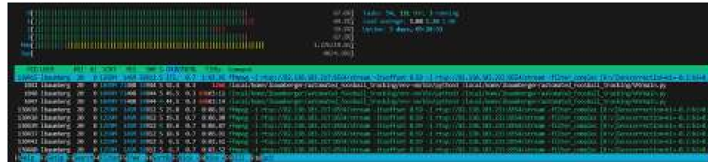

Figure 3.7: CPU and Memory usage of the VM while livestreaming

3.3 Phase 3, Analysis Phase:

We have three working algorithms. The first one is for (potential) goal detection. We have two zones. The outer and bigger one for potential goals, the smaller one for actual goals (Figure 3.8). For each zone we do a background subtraction over the average of the last 500 frames (at 80fp, that are a little bit more than the last 6 seconds). After that background subtraction we do some postprocessing. A 5x5 elliptical kernel is created for morphological operations. The mask is cleaned of noise using opening, and dilation fills the gaps to make the object more cohesive. We then count the pixel change, if the change is over a certeain threshold we assume that a (potential) goal happened.

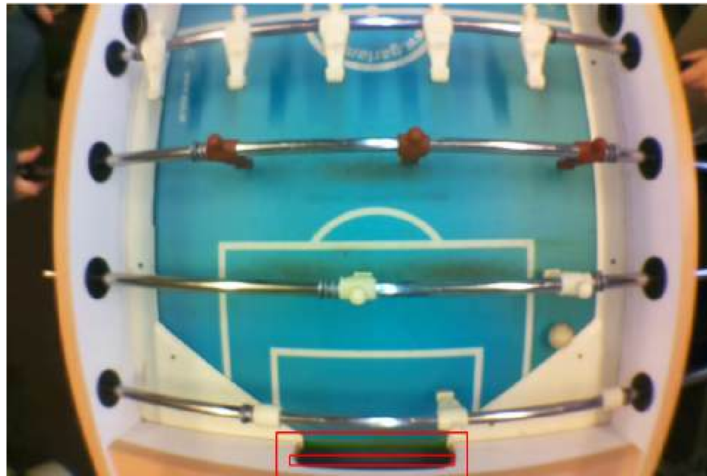

Figure 3.8: The outer and bigger zone checks for potential goals, the smaller one for actual goals.

The second algorithm uses background subtraction and checks for circularity of detected objects. We subtract the average background and this leaves us with a selection of objects that moved in the last few seconds. As we did with the goal detection we first do a background subtraction over the average of the last 500 frames and then add some postprocessing (Figure 3.9). Again we use a 5x5 elliptical kernel that is created for morphological operations. The mask is cleaned of noise using opening, and dilation fills gaps to make the object more cohesive. We ignore all detected objects with an area to small or to big to be a foosball. Then we check our remaining possibilities for their cirularity. For that purpose we used a helper function which calculates the circularity of a contour using the formula $circularity = \frac{4\pi * area}{perimeter^2}$ (where area is the area of the object and perimeter is he total length of the boundary of the shape), which measures how close the shape is to a perfect circle. The function returns True if the circularity is within the range 0.7 to 1.2, indicating the object is roughly circular.

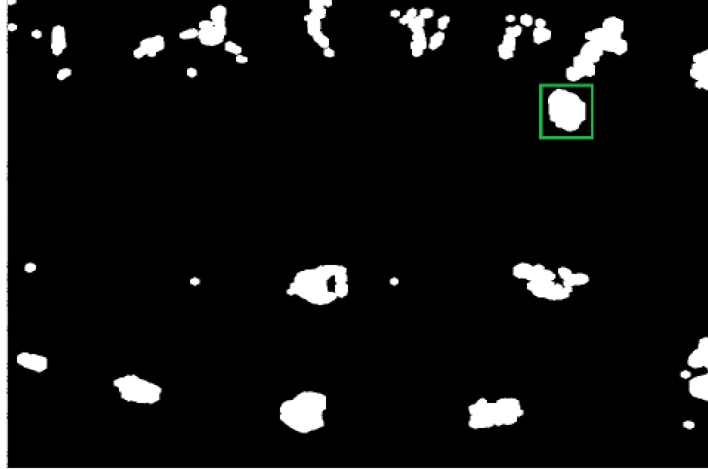

Figure 3.9: This is the frame after background subtraction and postprocessing (5x5 elliptical kernel, opening and dilation). We see all the objects that moved in the last few seconds. All the foosballplayers and also the foosball(green rectangel)

The third algorithm uses YOLO. YOLO is a state-of-the-art object detection algorithm that detects and classifies multiple objects in an image or video in a single forward pass [4, 5]. YOLO divides the image into a grid and predicts bounding boxes and class probabilities simultaneously for each grid cell, making it extremely fast and suitable for real-time applications. We used pretrained models and finetuned them with our own data. We tested the models on fresh and previously not seen data and it performed way better than the classical computer vision algorithm which uses backgroung subtraction. When tested on edegcases (for example partially covered foosballs or foosball on white background) both the precision and the recall was more than doubled (Table 3.1). Especially it was more accurate in tracking partially obscured foosballs and in tracking really fast foosballs (they appear stretched on the video)(Figure 3.10). The downside is the lower throughput (of only 10 frames per second) while running it on a cpu.

Algorithm	True Positives (TP)	False Positives (FP)	Precision (%)	Recall (%)
Opencv	32	14	69.5	33.3
YOLO8n	87	0	100	89.6
YOLO11n	88	2	97.7	91.6

Table 3.1: Results of the more classical computervision algorithm using opencv, YOLO8n and YOLO11n. Tested on 100 pictures (including three true negatives), mainly edgecases (foosball partially covered, on white background)

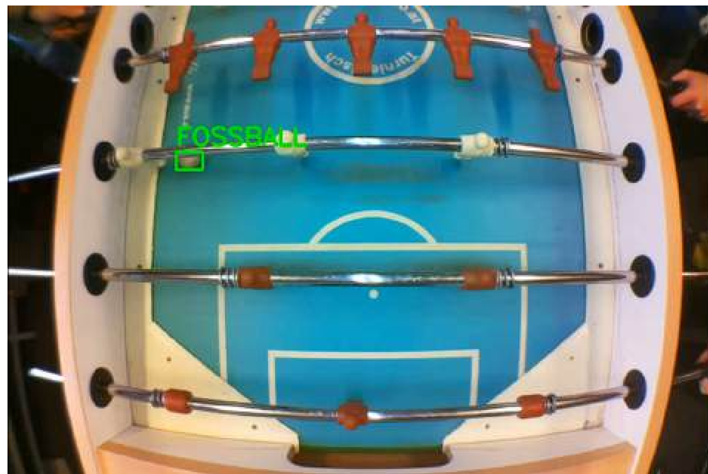

Figure 3.10: The balldetection with YOLO was more accurate and foosballs were detected even if they are fast or partially obscured

Chapter 4

Future Work

Currently we have the alogrithms to do the analysis but we are not yet able to run them at runtime. It should have high priority to find a solution to that problem. Those problems are related to the analysis phase (phase 3) and the following things should be worked on.

1. Implement an ffmpeg filter with opencv. Currently, there is a problem while compiling our own version of ffmpeg(that supports opencv). Because opencv is written in c++ while ffmpeg is mostly written in c. If we are able to find a solution that would automatically allow us to use our classical computer vision alogrithms as well as YOLO.
2. Analysis at runtime (one possibility would be our own ffmpeg filter). The challenges are certainly the high framerate we have to process. Reducing the framerate is not really an option because the foosball moves so fast that we would miss certain scenes/situations if we would for example only analyze every third frame.
3. Another interesting idea would be to automatically replay (potential) goals in slow motion. This would give the players and specatators and interesting insight into the crucial situations of the game.
4. As soon as we are able to perform an analysis at runtime we can collect data and produce statistics about the current game state. With those statistics we could maybe even do predictions about the future state of the game.

After finishing phase 3 in the future there is room open for additional phases or projects. The system is built in a modular way that allows future projects to integrate well.

Chapter 5
Conclusion

As a foosballplayer myself it was really fun to work at that project. To combine the passion for hardware projects, computer vision and software. The goal was to develop a visual tracking system for foosball. Such a visual tracking system for foosball would allow us to collect data and create statistics on the current state of the game. With the help of all those data one could for example create an automated virtual assisted referee. Or we could analyze which team has more ball possession and we could analyze how fast some of the shots are. We first designed and installed a cameramount and all the necessary hardware (Raspberry Pi and the led strip). This cameramount is built in a way that would allow one to mount even more hardware and sensors for a future project. We then moved on to provide a system that reliably records and streams a foosball game with a latency of less than 10 seconds. In a third step we started developing algorithms to detect goals and the foosball.
A further step would be to integrate those algorithms into our existing system. Introduce runtime analysis and do some statistics with data collected. Really interesting would also be the if we could predict how likely it is that a certain team scores a goal in the next few seconds.

Bibliography

[1] "https://github.com/bluenviron/mediamtx."

[2] "https://github.com/opencv/opencv."

[3] "https://www.ffmpeg.org."

[4] R. Khanam and M. Hassain, "Yolov11: An overview of the key architectural enhancements," Oct. 2024.

[5] "https://github.com/ultralytics/ultralytics."

Appendix A

Documentation

Appendix A
Hardware

For all the parts used and described in this chapter there are CAD-drawings which can be downloaded from the gitlab repository.

A.1 Camera Mount

The camera mount is an integral part of this project. On the one hand it holds both cameras on the other hand it also holds the light strip. To construct the mount, we used a combination of wooden, plastic, and metal parts, which will be explained in more detail later.

The following difficulties/problems had to be addressed during the construction of the Mount. One major challenge was to reduce the movement/vibration along the axis, especially along the y-axis (Figure A.1).

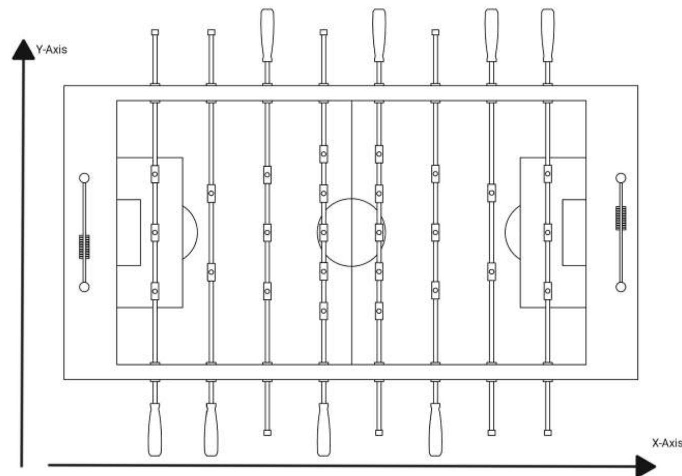

Figure A.1: Schematic Foosball Table

Our general approach to that problem was to increase the stiffness of our construction and, at the same time, to make the connection to the foosball table as rigid and movementless as possible. The details on how we achieved this will be explained later when we discuss the specific parts.

Another challenge we had was that our mount has to be designed in a way that we do not block (or at least minimize the impact on) the field of view or the movement of the players.

The mount is designed in such a way that it could be modified and upgraded in the future. As a result of our use of standard aluminum profiles it is very easy to add further sensors and equipment in the future. The same holds for the wooden strip (Figure A.2).

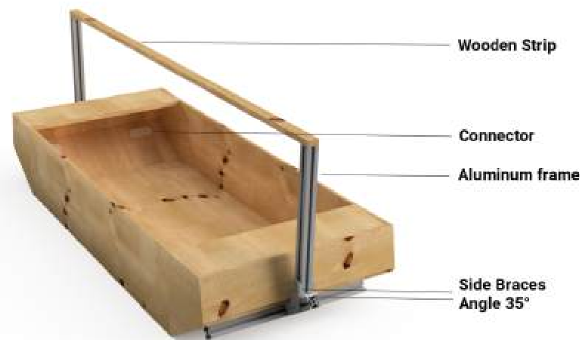

Figure A.2: Overview Foosball Table

A.1.1 Aluminium Frame

We decided to use 40mm by 40mm aluminium extrusions (Figure A.3). This allowed us to use mostly off the shelf components, screws and connectors. Aluminium is lightweight, stiff, and offers a bunch of possibilities to add our own components, therefore this is the optimal choice for a mount like this. For each side we needed 3 pieces, with the following lengths: 250mm, 500mm, 600mm (Figure A.4).

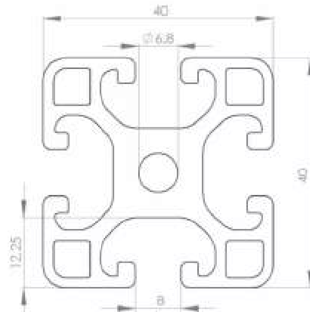

Figure A.3: Dimensions Aluminium Extrusion

Figure A.4: Aluminium extrusions overview, and where they are located on the foosballtable

A.1.2 Connector

This part connects the cameramount to the foosballtable. The foosballtable already has two existing holes on each side, which we use as mountpoints. The connector is split into two parts. On the one hand we have a wooden/plastic plate (wood is prefered, because of the higher strength). This plate is placed on the inside of the hole. On the outside we have the aluminium extrusion and in the hole itself we have a plastic spacer, which helps us to reduce movements along the y-axis (Figure A.5, Figure A.6).

Figure A.5: Wooden Plate and Plastic (PLA) Spacer

Figure A.6: Installation and location of the connector

A.1.3 Angle 35°

To adress the custom angle of our mountpoints on the foosballtable, we used a plastic 35° angle (Figure A.7). While it would be more rigid and stiff if we would use metal, it would be also be more difficult to produce. Therefore we settled here for a plastic part. For additional support of the angled connector we added side braces (see next subsection) The angle is mounted with T-Nuts and M6 screws to the angled 250mm long extrusions. To allow a screwed connection to the vertical 500mm long extrusion we threaded an M8 thread into the extrusion (Figure A.8).

Figure A.7: 35° Angle

Figure A.8: Installation and location of the angle

A.1.4 Side Braces

We have two wooden (plastic would work, but is less effectiv) braces to stabilize the connection between the aluminium extrusions. It maximises the stiffness and minimizes vibration and movement along the y-axis (Figure A.9, Figure A.10).

Figure A.9: Side Brace

Figure A.10: Installation and location of the side braces

A.1.5 Wooden Strip

The wooden strip is mounted on top of the aluminium frame. It is screwed to the extrusions and allows us to not only mount the cameras and the led strip but also leaves us plenty of space to install further sensors and equipment for a further project (Figure A.11).

Figure A.11: The wooden strip is mounted on top of the aluminium frame.

A.2 Cameras

As cameras we used IMX219 generic cameras with a fisheyelens. Both cameras have a field of view of 130° (Figure A.12). Further specs which are relevant for us are the resolution and fps we can achieve with those cameras. A framerate of 80 fps at 1080x720 allows us to efficently and correctly track the foosball (under the asumption that a foosball has a velocity of at most 10m/s). In case if those 80fps should be to slow to create reliable results one could increase the framerate up to 200fps as long as we are okay with a reduction in the pictureresolution (down to 640x480)

Those cameras are mounted with screws to a plastic case for protection against flying foosballs. Those plasticcases are 3d printed and themself screwed to the wooden strip (Figure A.13. The Camera Serial Interface (CSI) cable which connects the cameras with the raspberry pis is glued to the wooden bar for an unobtrusive appearance.

As mentioned before, there is the possiblity to add more/different cameras to the wooden bar to improve the results or get additional data.

Figure A.12: 130° Camera

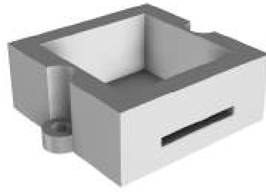

Figure A.13: Casing

A.3 Elumination

To garantee lighting that is consistent and to minimize shadows, there is a controllable led-strip (glued to the wooden bar). The LED-Strip ist powered by an external 12V DC powersupply and controlled by one of the pis (Figure A.14).

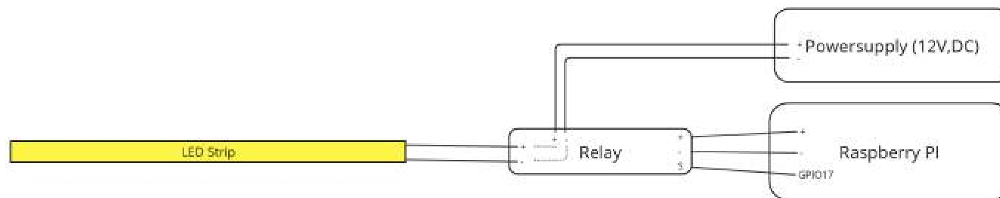

Figure A.14: Schematic LED Strip

A.4 Control Unit/ Raspberry Pi

We use the Raspberry Pi 5 to fullfill four main duties. First and most important the act as a server and offer a small but functional API for all the functionality explained in the next few paragraphs

Secondly to controll the camera, to start and end the recordings and redirect the recordings to a mediaserver.

Third: As mentioned before the streams get sent to an RTSP-Mediaserver which is also running on the raspberry pi. This RTSP-Server is responsible that we can access the footage via the VM as fast as possible

And the fourth thing that is managed by the Raspberry Pi is the lighting.

Below there is a schematic drawing on how erverything is working together and how it interacts (Figure A.15).

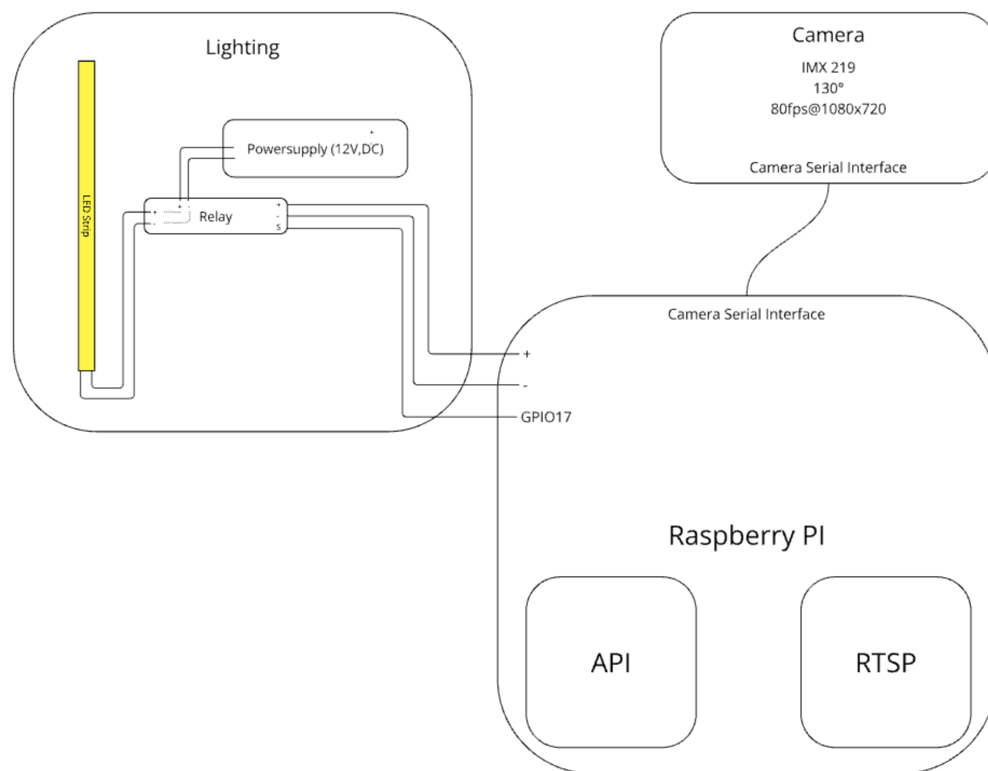

Figure A.15: Schematic Raspberry Pi

Appendix A

Software

When we look at the software we divide it into the VM side and the Rasperry Pi side. The schematic below gives us a better understanding of the whole system (Figure A.1).

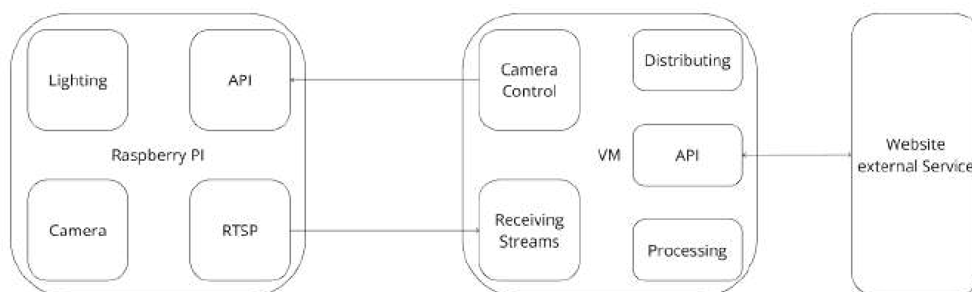

Figure A.1: Schematic Raspberry Pi and VM

The communnication between the VM and the PI goes bidirectional. On the one hand we have an API on the PI which we can use to controll it from the VM. On the other hand we have an RTSP-Server running on the Pis which allows the VM to access the recordings/streams in an effficent and timely manner.

Additionally we provide an API on the VM. This is used to feed the website with up to date information and allows external systems and users to take control over the VM.

All the details on how the API works will be explained in the corresponding subsections

A.1 Project Structure

```
project/
|
|-- env/                        % python environement (will be automatically
|                               % created on setup)
|
|-- CAD/                        % CAD files for the 3D printed parts
|
|-- HLS/                        % Folder where we save all the HLS Files
|
|-- LOGS/                       % Folder where we save all the LOG Files
|                               % created from the API
|
|-- Maintenance/
|   |-- move_old_files.sh       % Bash script to move files older than 7
|                               % days to the external HDD
|
|-- Mediamtx/                   % All files to run the RTSP/Media-Server
|
|-- PI/
|   |-- P1.py                   % Script for Pi 1 (responsible for the lighting)
|   |-- P2.py                   % Script for Pi 2
|
|-- Recordings/                 % Folder where we save alle the Recordings
|
|-- Setup/
|   |-- P1_setup_service.py
|   |-- P2_setup_service.py
|   |-- VM_setup_service.py
|   |-- setup_pi.py
|   |-- setup_vm.py
|   |-- requirements_pi.txt     % requirements for the PI
|   |-- requirements_vm.txt     % requirements for the VM
|
|-- Start/
|   |-- reboot_system.sh        % reboots the PIs and restarts the service
|                               % on the VM
|
|-- Status/
|   |-- check_service.py        % allows to check for the services and
|                               % their logs
|
```

```
|-- VM/                          % all the scripts that run on the VM
|    |-- main.py
|    |-- app.py
|    |-- control.py
|    |-- hls.py
|    |-- logs.py
|    |-- variables.py
|
|-- VM.log                       % all the logs of the VM
|-- P1.log                       % all the logs of Pi 1
|-- P2.log                       % all the logs of Pi 2
|
|-- Tracking/                    % Folder with all the tracking algorithms
|    |-- gnerate_trainingsdata/  % Folder with all the scripts to generate
|    |                           % the training data
|    |-- Predict_Video_Yolo/     % Videoanalysis with YOLO
|    |-- Train_Yolov8_Foosball/  % Training of the YOLOv8n and Yolo11n
|    |                           % model
|    |-- results_yolo_training   % detailed reports of the training
|    |-- weights/                % Folder with the weights of the trained
|    |                           % models
|    |-- background_subtraction.py
|    |-- goal_detection.py
```

A.2 Raspberry PI

A.2.1 Setup

To setup a new Raspberry Pi (Figure A.15) one can install the newest release of Raspberry Pi OS onto the Pi. Connect the Pi to the gitlab repo and then first execute the Setup/setup_pi.py, this script downloads all the necessary packages and creates a virtual pyhton environment. Secondly one should execute the Setup/P1_setup_service.py or Setup/P2_setup_service.py to setup all the services.

A.2.2 Overview

If you execute the Setup/P1_setup_service.py or the Setup/P2_setup_service.py script you will set up two independent services. First you will install a service that automatically start the mediamtx server. We will use this server as a rtsp server. That guarantees us that we are always ready to stream.

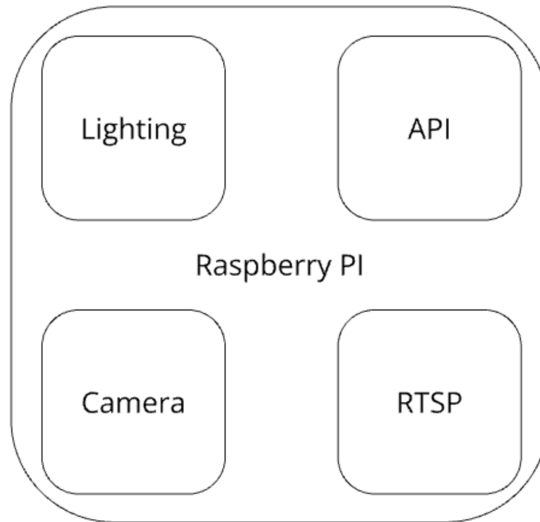

Figure A.2: Each raspberry pi has for main function. Controlling the Camera, controlling the lighting, providing an API (to be controlled by the VM) and providing a stream via rtsp server

Secondly it will install a service that automatically pulls the newest version of the "build" branch from the gitlab repository and then executes the PI/P1.py or PI/P2.py file.

Those files do the following:

1. We start a flask server. This provides the API.
2. We wait for an API request. As soon as we have a request to start the Stream we will create a new thread which handles the whole recording and redirecting to our RTSP server. We then also call a function that activates the lighting.
3. After that our server continues to wait for more requests. If we receive a request to stop the stream, then we end the recording and switch of the lighting.

A.3 API Raspberry Pi

This API allows us to control the Raspberry Pis from the VM. Currently the functionality is limited to the basics. In the future this could be extended.

A.3.1 Authentication

Currently there is no authentication necessary.

A.3.2 Endpoints

GET /start

Description: Starts the recording and automatically redirects the stream to the RTSP server **Request**:

```
GET /start HTTP/1.1
Host: IP-Adress of the Raspberry Pi
```

Response:

```
# if the stream/recording can be started: 200
{
    "message": "Stream started"
}
# if the stream/recording is already running:
   400
{
    "message": "Stream is already running"
}
# if there is a server side error: 500
{
    "error": "Error description"
}
```

GET /stop

Description: Stop the recording

Request:

```
GET /stop HTTP/1.1
Host: IP-Adress of the Raspberry Pi
```

Response:

```
# if we were able to stop the stream: 200
{
    "message": "State set to Streaming stopped"
}
# if there is a server side error: 500
{
    "error": "Error description"
}
```

A.3.3 Error Handling

- **404 Not Found**: The requested Endpoint does not exist
- **500 Not Found**: Serverside Error

A.4 VM

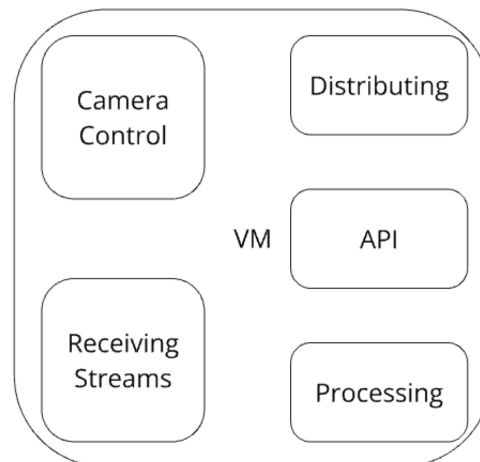

Figure A.3: Schematic VM Overview

As soon as that service is running our VM provides 5 different main tasks (Figure A.3) The functionality is split up into different files and provided by different threads (Figure A.4).

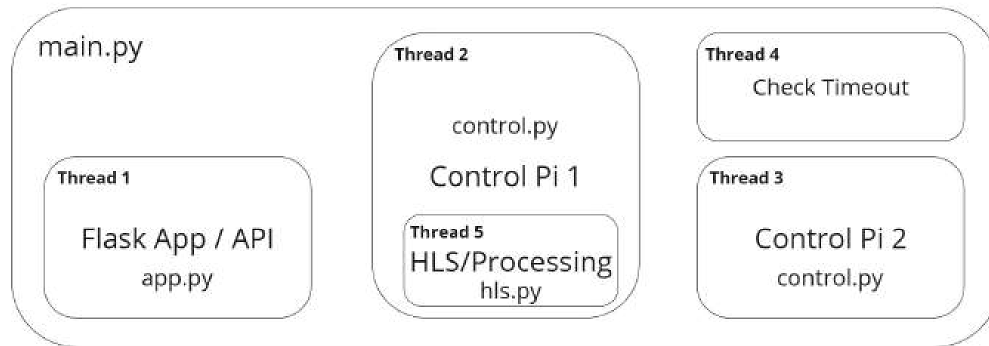

Figure A.4: VM Software Threads

A.4.1 Setup

To install all the necessary packages and libraries and to also create a fitting virtual environement one can execute the following script: Setup/setup_vm.py If we execute the Setup/VM_setup_service.py we install a service on the VM, which automatically pulls the newest version of the "build" branch everytime we restart the VM. Besides pulling the newest version it takes care of automatically starting/restarting the VM/main.py file.

A.4.2 Camera Control

To control both our Cameras we take advantage of the API on the Raspberry PI. Because we have two Raspberry Pis we also have two seperate hreads (**Thread 2** and **Thread 3**) which are both responsible to control one of the Pis.

A.4.3 Communication RTSP / Receiving Streams

As soon as we make the API calls to start a stream we call from **Thread 2** a new thread (**Thread 5**), whose sole purpose is to receive both streams from the RTSP servers and proccess them. Processing includes the following steps:

1. Save input1 as well as input2 as an individual mp4 file
2. (Analyse both inputs for potential goals and do balltracking)
3. Crop both inputs together and correct the fisheyelens
4. Save the stream which is now cropped together as an HTTP-Livestream (HLS)

A.4.4 Processing

All the processing is defined in the following file: VM/hls.py Before we crop the inputs together to one stream we do the following steps:

1. Correction of the fisheyelens
2. Transpose both inputs
3. Stack them together

A.4.5 Distributing

As soon as we receive and save our files the flask server (**Thread 1**) running on our VM (provides the API and the Website) takes over. It distributes the livestream and on request also allready recorded games. The API is described down below.

A.5 API VM

This API allows us to control the VM from our (debug)website or from any external system. This allows us to integrate the functionality of recording, livestreaming, and analyzing at runtime into other projects.

A.5.1 Authentication

Currently there is no authentication necessary.

A.5.2 Endpoints

GET /start

Description: Starts the recording and automatically redirects the stream to the RTSP server. **Request**:

```
GET /start HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response:

```
# if the stream/recording can be started: 200
{
```

```
        "status": "success",
        "message": "Streaming started successfully",
        "streaming_status": [true, "filename"]
    }
    # if there is a server-side error: 500
    {
        "status": "error",
        "message": "Failed to start streaming: <
           Error description>"
    }
```

GET /stop

Description: Stops the recording.

Request:

```
GET /stop HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response:

```
    # if we were able to stop the stream: 200
    {
        "status": "success",
        "message": "Streaming stopped successfully",
        "streaming_status": [false, "filename"]
    }
    # if there is a server-side error: 500
    {
        "status": "error",
        "message": "Failed to stop streaming: <Error
            description>"
    }
```

GET /check_status

Description: Returns the current streaming status.

Request:

```
GET /check_status HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response:

```
# Current streaming status: 200
{
    "shouldBlink": <true or false>
}
```

POST /start_stop

Description: Toggles the state of the recording (start/stop).

Request:

```
POST /start_stop HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Redirects to the index page.

GET /get_logs

Description: Downloads the latest formatted log file.

Request:

```
GET /get_logs HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response:

```
# If the log file is available: 200
<File download as an attachment>

# If the log file is missing: 404
{
    "error": "Log file not found"
}
```

GET /debug

Description: Displays a debug page listing all available HLS (.m3u8) files in the directory.

Request:

```
GET /debug HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Renders an HTML page displaying the list of files.

GET /livestream

Description: Streams the currently recording video, or a default file if none is being recorded.

Request:

```
GET /livestream HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Streams the appropriate HLS (.m3u8) file.

GET /<filename>

Description: Serves an HLS (.m3u8 or .ts) file from the directory.

Request:

```
GET /<filename> HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Sends the requested file.

GET /play/<filename>

Description: Plays a specified video file directly.

Request:

```
GET /play/<filename> HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Sends the requested file to be played.

GET /analysis

Description: Toggles or sets the analysis state.

Request:

```
GET /analysis?analysis=<true|false>
Host: <IP-Address of the Raspberry Pi>
```

Response: Redirects to the debug page.

POST /restart

Description: Restarts the system.

Request:

```
POST /restart HTTP/1.1
Host: <IP-Address of the Raspberry Pi>
```

Response: Redirects to the index page.

A.5.3 Error Handling

- **404 Not Found**: The requested endpoint or file does not exist.
- **500 Internal Server Error**: A server-side error occurred.