NAO Vision

Final Project

Tim Grutzeck (386921)
Marcel Ebermann(382654)

February 27, 2021

1 Preface

This project is about designing an approach for the vision module of NAOs. Due to the circumstances, it was not possible to work with real robots this semester. It has a major impact on this project because we do not have access to the robot's camera, which is an important part of the hardware for the vision module. Our solution to this problem is to create a simulation that best mimics the views of the robots.

2 Introduction

One of the main tasks of the vision module is to determine the ball position.When tracking the ball, the problem may arise that one or even all of the robots do not recognize the ball at some point in time. This could be because the ball is hidden behind an obstacle, the robot is looking in the wrong direction, or the ball is too far away and therefore too small for the detection software. Our approach should offer a high level of reliability.Figure 1 shows the procedure of the project in a sequence diagram. The first step is, as mentioned, the creation of the simulation. The next step is to use the simulation to detect the ball. Multiple ball observations made by the robots will be analyzed together to track the ball more reliably. In the end, the performance of our approach is measured by comparing the estimated position of the ball with the real position in the 3D simulation.

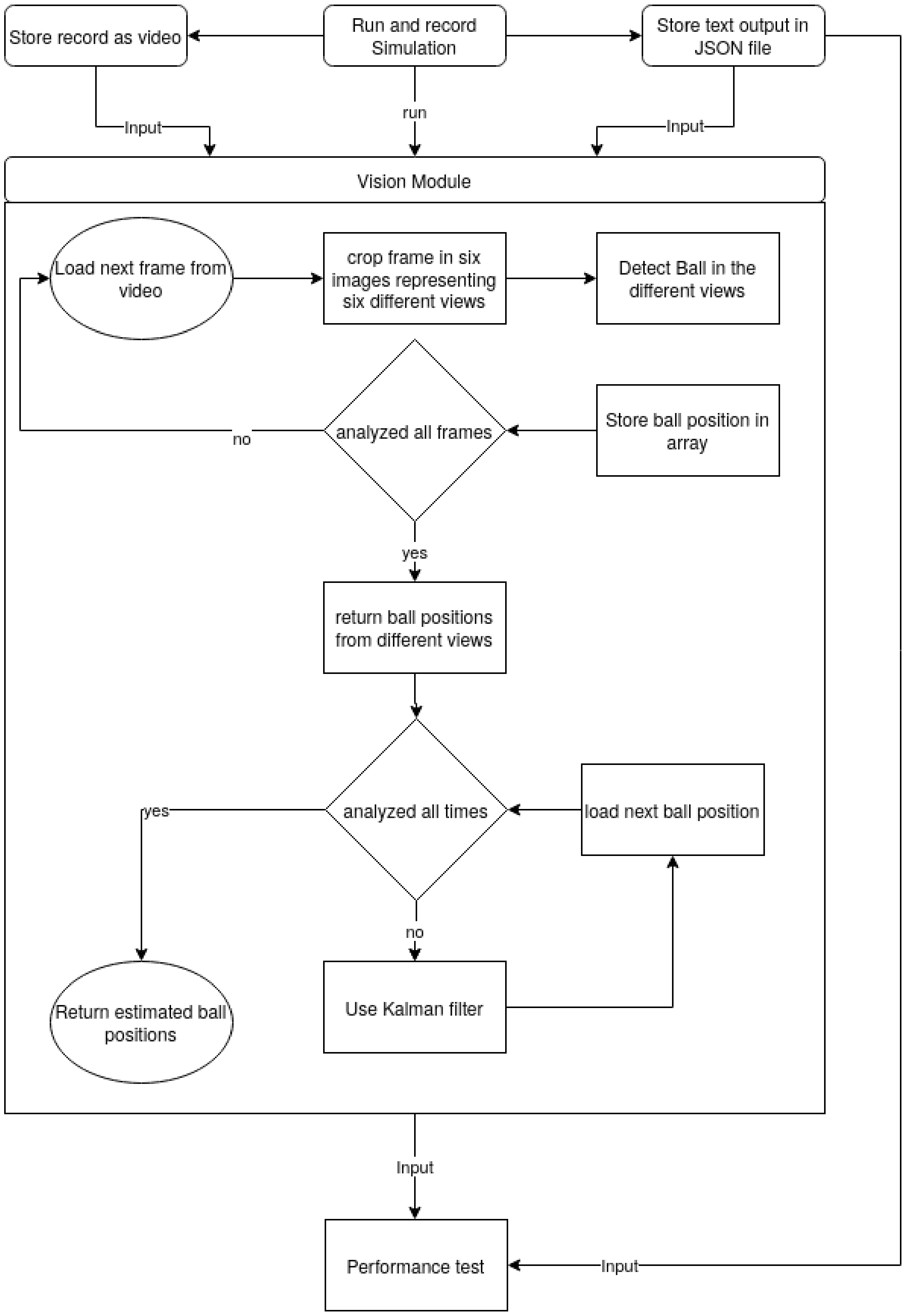
Store record as video
Run and record Simulation
Store text output in JSON file
Input
run
Input
Vision Module
Load next frame from video
crop frame in six images representing six different views
Detect Ball in the different views
analyzed all frames
no
Store ball position in array
yes
return ball positions from different views
analyzed all times
yes
load next ball position
no
Return estimated ball positions
Use Kalman filter
Input
Performance test
Input

Figure 1: Sequence Diagram

3 Requirements

Following software is required to run this project:

- gcc compiler
- OpenCV installed
- python (for performance check)
- Qt framework installed

Additional software is required, if it's desired to create the simulation yourself.

- Web browser
- screen recorder

3.1 Simulation

3.1.1 Theory

In order to simulate the robots view as best as possible, a simulation environment is created. This is not really a part of the vision module but necessary to create input data. Usually the camera provides the image which is analyzed. Therefore we use three.js [3], a cross-browser JavaScript library. With Three.js, WebGL animated 3D computer graphics can be created and displayed in a web browser. It can be directly included into an html file. The simulation shows a 3D simulation of a ball rolling across a soccer field. Some obstacles are also present on the field to simulate other robots that may occlude the ball. The ball is moved on a fixed path of which the trajectory is known. The scenery is rendered from six angles, representing the point of view of the team members.

3.1.2 Run the Simulation

To run the simulation, the file must be opened in a browser. No web server is necessary for this. To generate a video which can be used as input by the image analysis program one have to run the simulation in fullscreen mode and record the scenery with a screen recording program. When the animation is over, a text in JSON format is displayed in the browser window. This contains the ball trajectory, camera parameters and location. The text must be copied into a text file for later ball detection. This file can be found as simulation.json in the project directory.

3.2 Ball Detection

The implementation of the ball detection was realized with c++. OpenCV [2] is used for this, which is a comprehensive tool for computer vision. When dealing with real robots, the current camera capture of the robot is processed. In our application, we pass a recording of the simulation in video format to the vision module and analyze it frame by frame. Before we can start with the actual analysis, the current frame has to be edited. At the moment it shows all views at the same time. To process each individual view, the frame is cropped into six different frames, each containing a single view. In the next few steps, a selected view will be used as an example for analysis. The selected view is shown in figure 2.

Compile:

```
$ make
```

Run the program:

```
$ ./ballDetection sim.avi simulation.json 1
```

Note: The last argument is for method selection and must be a number. Pass 1 for findContours and any other number for HoughCircles method.

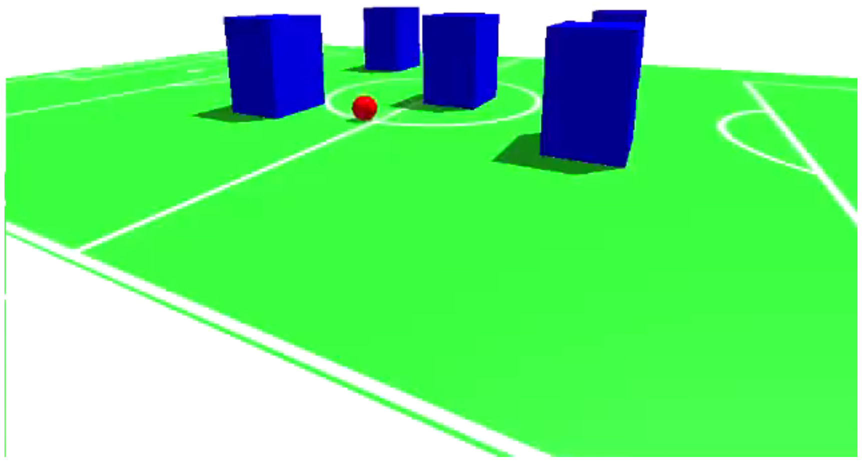
Figure 2: Original view

3.2.1 Color Conversion

For Simplification the ball has a unique color. In this example the color is chosen to be red. To identify a particular color the HSV color space works very well. The HSV color space is visualized in 3.The function cvtColor from OpenCV converts an image from one color space to another. The conversion of the original view from BGR color space to HSV is shown in figure 2 4.

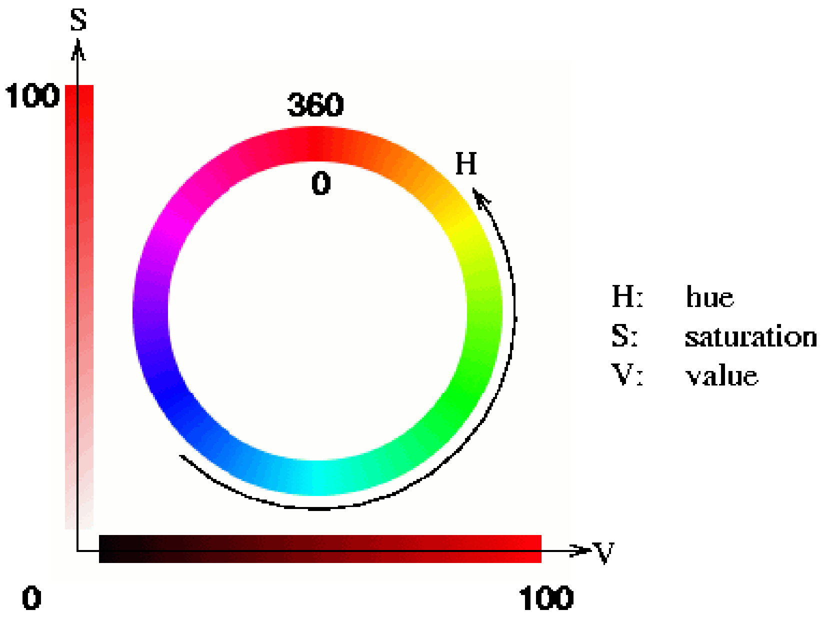

Figure 3: HSV color space

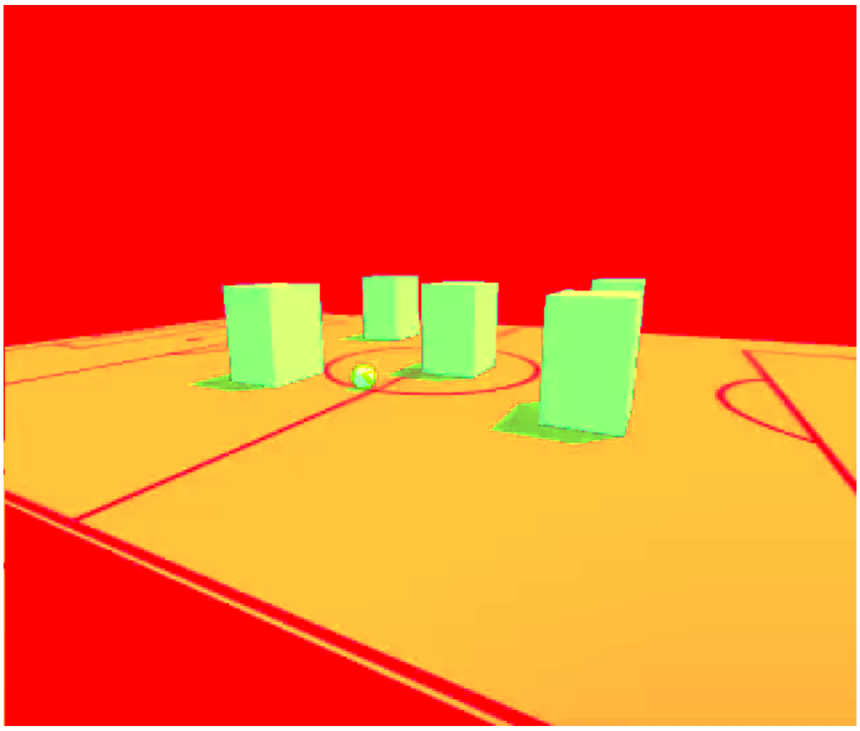

Figure 4: View in HSV color space

3.2.2 Threshold View

Instead of three values like in the RGB color space, just a single value, the hue is used to identify a color. The other parameters of the HSV color space are S the saturation and V the value, responsible for the brightness. As shown in 3 the red color, has hue values approximately in the range of 0 to 20 and 320 to 360 degrees. V color space are S the saturation and V the value, responsible for the brightness. In OpenCV H has values from 0 to 180, S and V from 0 to 255. The ranges can easily be transferred to the corresponding values in OpenCV. In OpenCV, the red color has hue values approximately in the range of 0 to 10 and 160 to 180. Because of its own shadow, the bottom of the ball appears darker. Therefore, the range for V is set to 20-255. To mask the ball, the OpenCV function inRange is used to threshold the HSV image for anything that is not red. The threshold images are shown below in figure 5 and 6. Then both masks are combined to one threshold image, shown in figure 7. In order to avoid false positives, the result is slightly blurred with a Gaussian Blur filter.

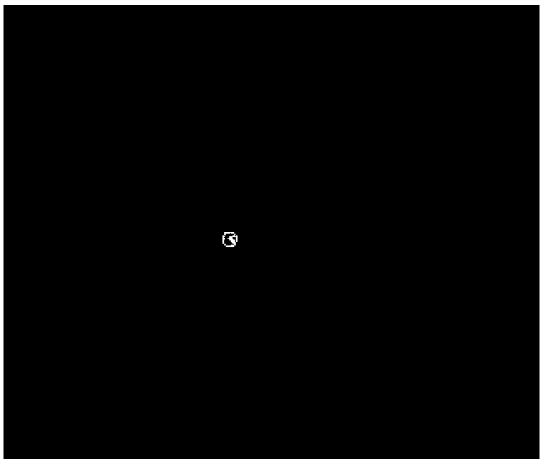
Figure 5: Mask1 H in range 0 to 20

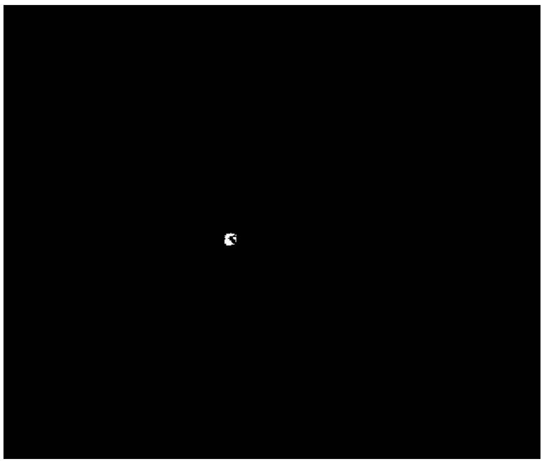
Figure 6: Mask2, H in range 160 to 180

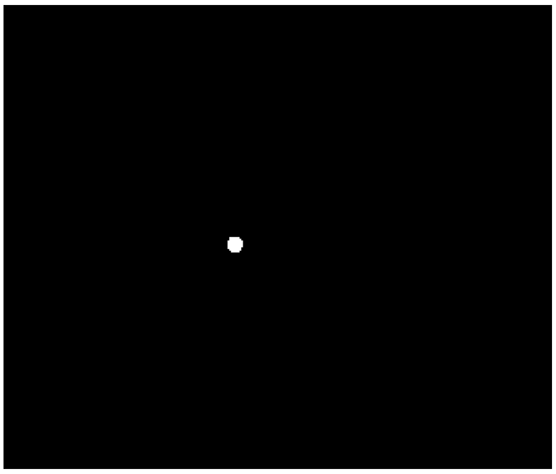
Figure 7: Threshold image, combination of both masks

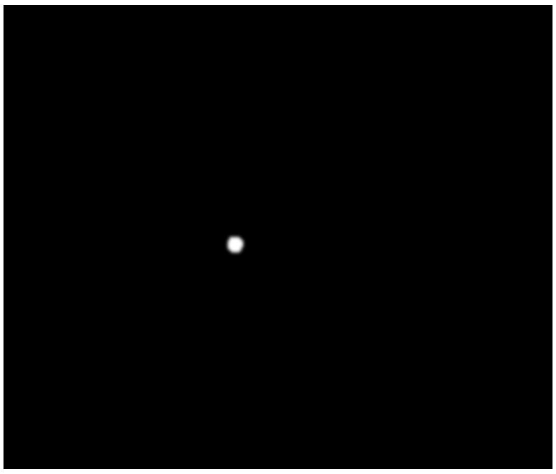
Figure 8: Threshold image with Gaussian Blur

3.2.3 Ball recognition

The next step is to recognize the ball on the image. Two approaches are to be considered and then compared with one another. The first approach uses the OpenCV function HoughCircles which uses the circle Hough Transform (CHT) [9], a basic feature extraction technique is used. OpenCV provides another method, called findContours which returns all recognized contours from an image. These contours don't have to represent any geometric shapes. They can be simple blobs. Then OpenCV's approxPolyDP function is used to approximate a polygonal curve with a specified precision. This function uses the Douglas–Peucker which decimates a curve composed of line segments to a similar curve with fewer points. To find the smallest circle enclosing the found contour minEnclosingCircle is used. This circle represents the detected ball.

For the demo, a white ring is placed over the recognized ball in the original view. This is illustrated in Figure 9.

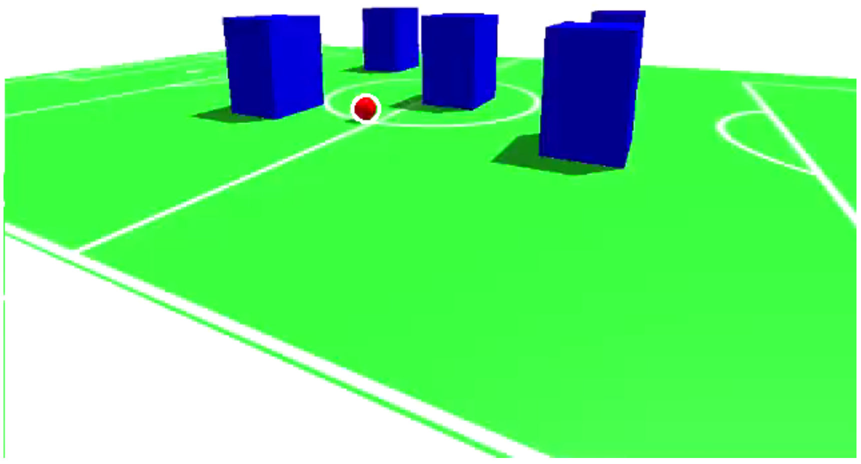

Figure 9: View with white circle around recognized ball

3.3 Ball Position Analysis

After tracking the ball in the images of multiple cameras, the position of the ball on the soccer field is calculated. This is done in a separate program. It takes the homogeneous transformation matrix of the openGL camera used in the simulation as input. That means that we assume that the camera position and orientation is given. A real robot should be able to determine its camera position and orientation relative to the soccer field by analysing the soccer field lines in its camera images.

We also assume that the focal length and the aspect ratio is given. When dealing with a real camera this parameters can be determined by calibrating the camera. Furthermore it should be possible to remove all distortions that might be present in real camera footage. Hence we assume that the camera image has no distortions which is the case in our simulation.

Our program takes also the ball positions in all camera images as input. The figure 10 shows that the origin of the 2d ball coordinate is in the center of the image and that they range from minus one to one for the x and y coordinate.

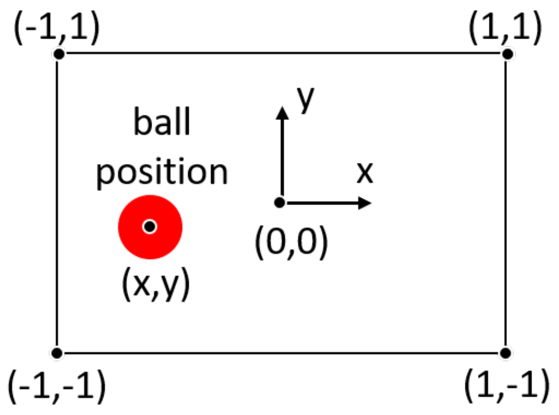

Figure 10: Ball coordinates on camera image

From our simulation we get the camera matrix of each camera which describes the position and orientation of a camera. This homogeneous matrix T transforms homogeneous coordinates from the local camera reference frame C in the world frame W. You can see that in figure 11. The z-axis of C points await from the point the camera is directly looking at. Because we use a simple pinhole camera model we can set the image plane in the local camera frame C. We place the image frame in front of the camera pinhole so that the image is not inverted. The distance between the image center and the camera pinhole point P_1 is equal to the focal length. The x-coordinate of a ball in the image can be directly used as the x-coordinate in the local camera frame of reference. The y-coordinate however needs to be scaled by dividing the aspect ratio of the image. The z-coordinate is the negative focal length. If we cast a beam of light from an object (that our camera sees) through the image plane and to the pinhole of our camera, it will intersect with the image plane exactly at the point where this object will be visible in our camera image. That means if we reverse this process and cast a ray through the image plane where we found the ball in our camera image, it will hit the ball on the soccer field. To describe this ray we can use the two points P_1 and P_2 which are visible in figure 11.

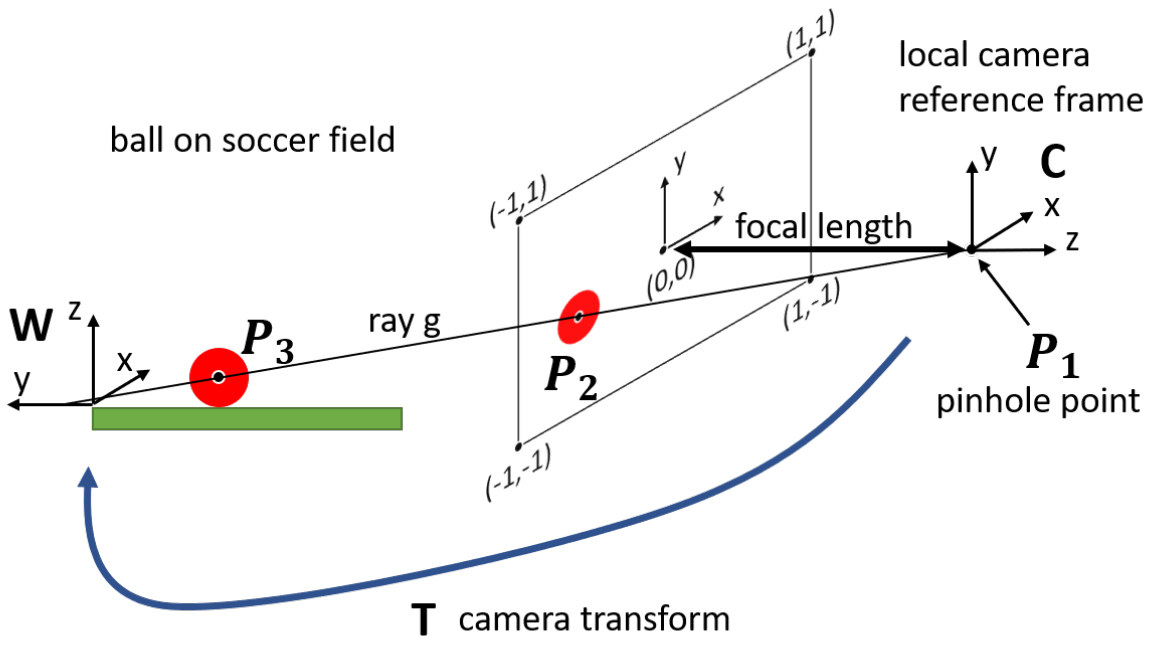

Figure 11: Calculation of the ball position

The equation to calculate P_1 and P_2 in the coordinates belonging to the global frame of reference W is as follows.

$$P_1 = T \cdot \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$P_2 = T \cdot \begin{bmatrix} x_i \\ \frac{y_i}{a} \\ -f \\ 1 \end{bmatrix}$$

Here T is the homogeneous camera matrix. x_i and y_i are the coordinates of the ball in the camera image, f is the focal length and $a = width/height$ is the aspect ratio of the camera image.

Now we can describe the ray g going through the pinhole point P_1 and the ball center in the camera image P_2.

$$g : x(t) = P_1 + (P2 - P1) \cdot t \mid t \in [0, \infty]$$

We assume that the ball can't jump and touches always the ground. Now imagine a plane E which is parallel to our soccer field and hovers over the ground with the distance of the ball radius.

$$E : x(u, v) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \cdot u + \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \cdot v + \begin{bmatrix} 0 \\ 0 \\ r \\ 1 \end{bmatrix} \mid u, v \in \mathbb{R}$$

In this equation r is the ball radius. The ray g will now intersect the plane E in the center of the ball. See figure 12.

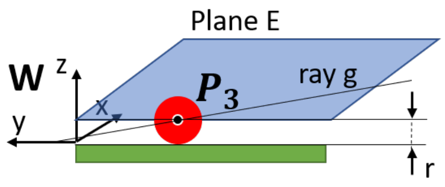

Figure 12: The ray g will intersect plane E at the balls center

Now the ball center can be calculated by setting the equations of g and E equal and solving the system of linear equations for t, u, v. After that u and v are the x and y ball coordinates in the reference frame W. We then add a third component (the height of the ball) and set it to the radius r. Now we have successfully calculated the point P_3 (from figure 11) where the ray g goes through the center of the ball.

$$P_3 = \begin{bmatrix} u \\ v \\ r \end{bmatrix}$$

This process is repeated for every camera so that we get a set of measured ball locations per time point. Those measurements must be combined. This will be explained in the next section.

3.4 Kalman Filter

A discrete Kalman Filter works by updating two main variables. The first one is a vector x which describes the current state of the system. The second one is a covariance matrix which describes how much the measured state of the system can probably deviate from the real one. This is necessary because the measurement noise and the process noise which are random vectors have an impact on the measured state. The system state x and the coressponding covariance matrix are first initialized. After that, a predict and a correct function are repeated alternately. First the predict function updates the state x by estimating the new state x_t from the parameters given in the previous state x_{t-1}. Likewise, the covariance matrix is updated.

In the following equations we use the index t for the current state and $t-1$ for the previous state.

In our implementation we have the state vector:

$$x = \begin{bmatrix} p_x \\ p_y \\ v_x \\ v_y \end{bmatrix}$$

Here p_x and p_y are the position of the ball in the x and y direction. v_x and v_y are the velocities of our ball in the x and y direction. In the predict function we calculate the current state x_t by multiplying the previous state x_{t-1} with our model matrix A. The model matrix describes the physical behaviour of the ball.

$$x_t = A \cdot x_{t-1} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} x_{t-1}$$

This results in the following equations.

$$\begin{aligned} p_{xt} &= p_{xt-1} + v_x \\ p_{yt} &= p_{yt-1} + v_x \\ v_{xt} &= v_{xt-1} \\ v_{yt} &= v_{yt-1} \end{aligned}$$

It is important to note that in this equations we assume that the Kalman Filter gets updated at a constant rate. Hence we can multiply the velocity v by one to get a distance. In this equation you can see, that we assume that the velocity of the ball does not change. We do this because we can not predict, when the ball gets accelerated and when friction slows the ball down.

The second variable that we need to update is the covariance matrix. This is done by the OpenCV implementation of the Kalman Filter [5].

In the update stage we just discussed the state and the corresponding covariance matrix are estimated. Now we know where the ball probably is based on the previous observation. If the position of the ball can not be measured again the predict function will assure that the ball positions gets estimated based on the last measurement of the ball position.

But now we want to integrate our measurements in our calculations to correct the our guess for the state we received from the predict function. To do that we use the OpenCV implementation of the correct function of the Kalman Filter. Here the Kalman Filter takes multiple ball position measurements as input. The math that is used internally is very difficult and hard to explain but can be explained in the following way by making some simplifications: All measurements and the predicted position from the predict function are weighted according to how reliable they are. This is done by using the measurement covariance matrix which describes the accuracy of the ball position measurement and the covariance matrix that we get from the predict function and describes how accurate our prediction is. The weighted positions from the measurements and the prediction are then combined to get the location where the ball probably is. This position is then set to the state vector. Likewise, also the velocity in the state vector is calculated by putting more weight on the predicted or the measured velocity.

Finally we can extract the positional elements of the state vector we get after each iteration to get a good estimate for the actual ball position.

The process described here is implemented in a seperate program. It takes the JSON files from the simulation and the image analysis programm as input. It's output is a JSON file which contains the ballposition estimate from the kalman filter and the ball generated by the simulation. We get also the timestamps for each ball position. We use this data to evaluate our result which is described in the next section.

3.5 Performance Check

To test the outcome of the vision module, the calculated ball positions are compared with the real ball position data from the simulation. This is done by a separate program, which takes the generated JSON from the simulation and the calculated ball positions from the vision module. This program is written in python. Matplotlib is used to plot the real and calculated ball positions in 2D and the distance between these points 15. Additionally the minimum, maximum and average distance is calculated.

Open a terminal and run the simulation:

```
$ python visualize_performance.py
    ballPositionsOutHoughCircles.json
    ballPositionsOutFindContours.json
```

Depending on the used method from chapter 3.2.3, HoughCircles or findContours, different result are obtained. The ball positions are plotted in Figure 13 and 14. The distance between 2D coordinates are calculated with the following formula.The results are summarized in the table 1.

$$d = \sqrt{x^2 + y^2}$$

Table 1: Distance between real and calculated ball position

Method	Maximum	Minimum	Average	Median
HoughCircles	0.4072	0.0012	0.0655	0.03716
FindContours	1.3636	0.0009	0.0575	0.03582

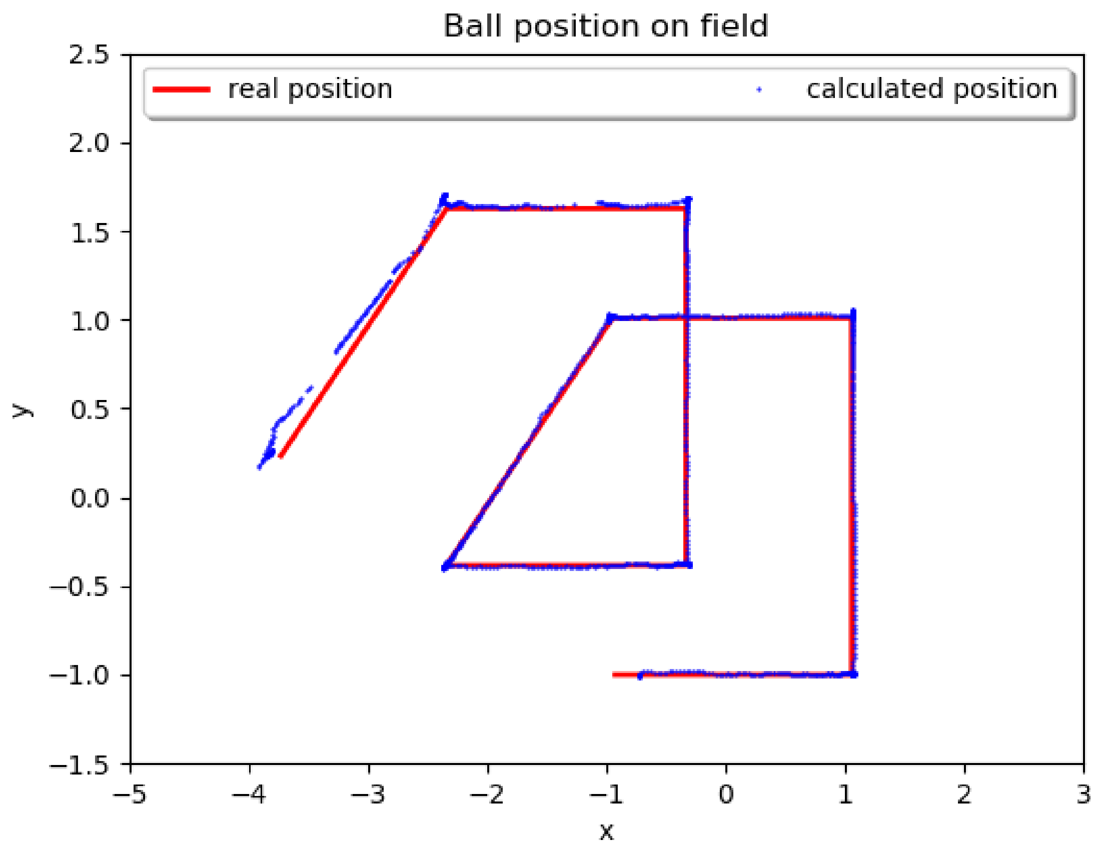

Figure 13: Ball position in 2D using HoughCircles

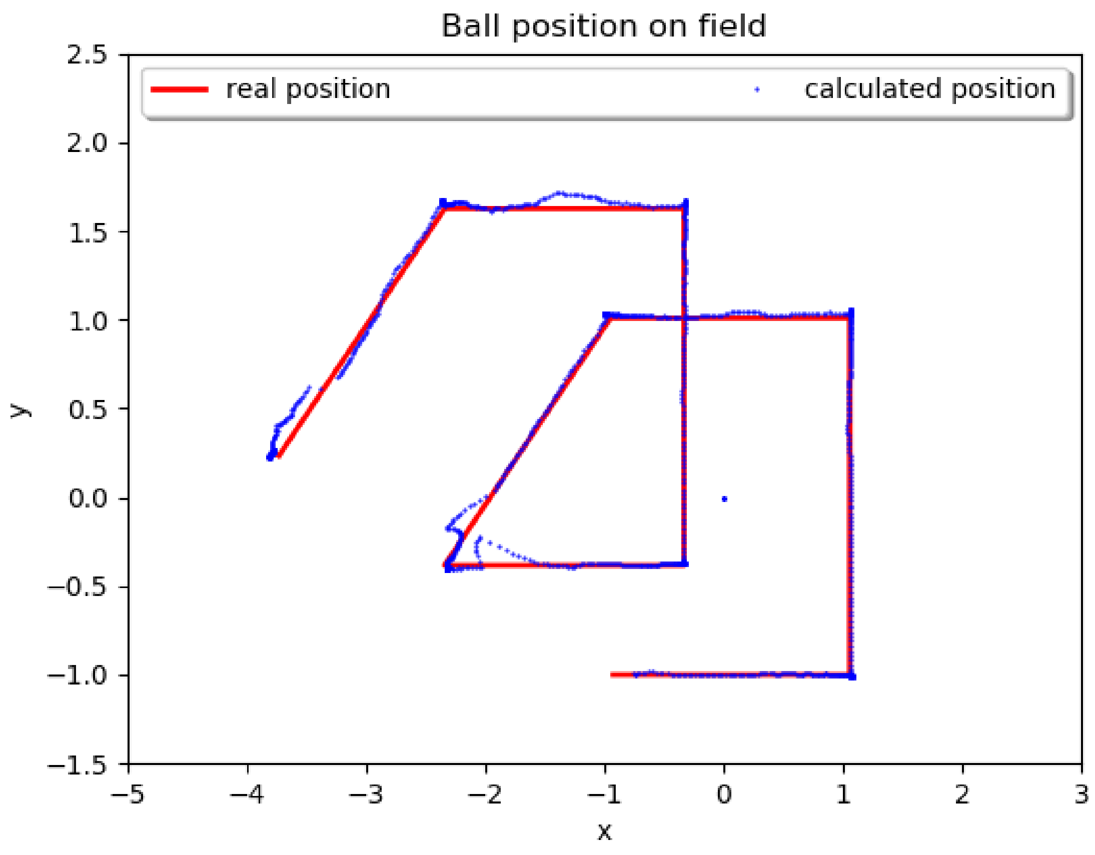

Figure 14: Ball position in 2D using findContours

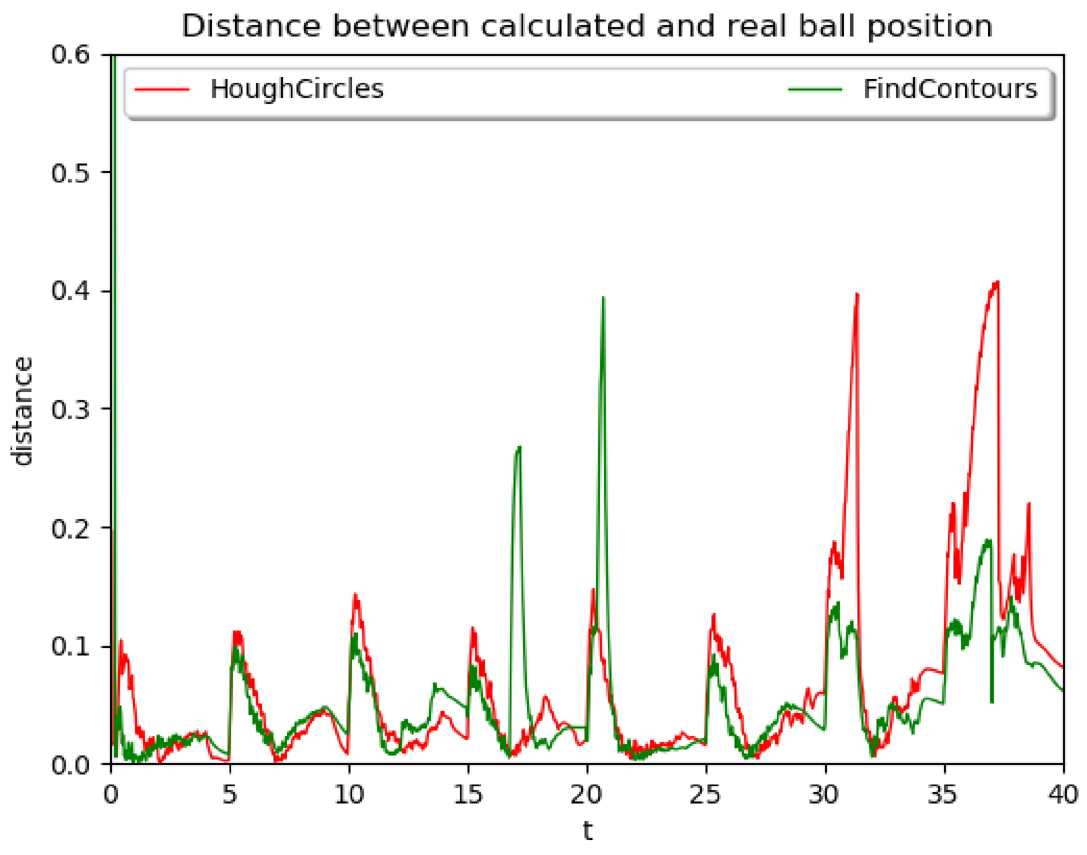

Figure 15: Distance between real position and calculated position

On average, the ball positions were calculated more precisely with the findContours method. The advantage of this method is, that the ball is recognized even if it is almost completely covered by an obstacle, as shown in 16. It should be take into account that with this method, objects with the same color could easily be mistakenly recognized as the ball.Also the circle enclosing the contour does not necessarily have the same center as the ball. This could lead to inaccuracies. The advantage of the HoughCircles method is, that objects with the same color as the ball are only considered when they are round. Becaus the ball is detected less often with this method, the Kalman filter has to work with fewer measurements and therefore less precise.

The reason for the spikes which occur every 5 seconds is that the ball there gets accelerated. This causes a rapid change in the ball velocity to which the Kalman Filter must adapt. After that the ball slows down. But since we can not predict when the ball gets accelerated and slows down we assume in the Kalman Filter that the ball moves with a constant velocity.

That does also explain the inaccuracies when the ball slows down.

The very large spikes occur when the ball is barely visible or not visible at all. There the position measurement gets inaccurate or the Kalman Filter can only use previous observations to estimate the current position.

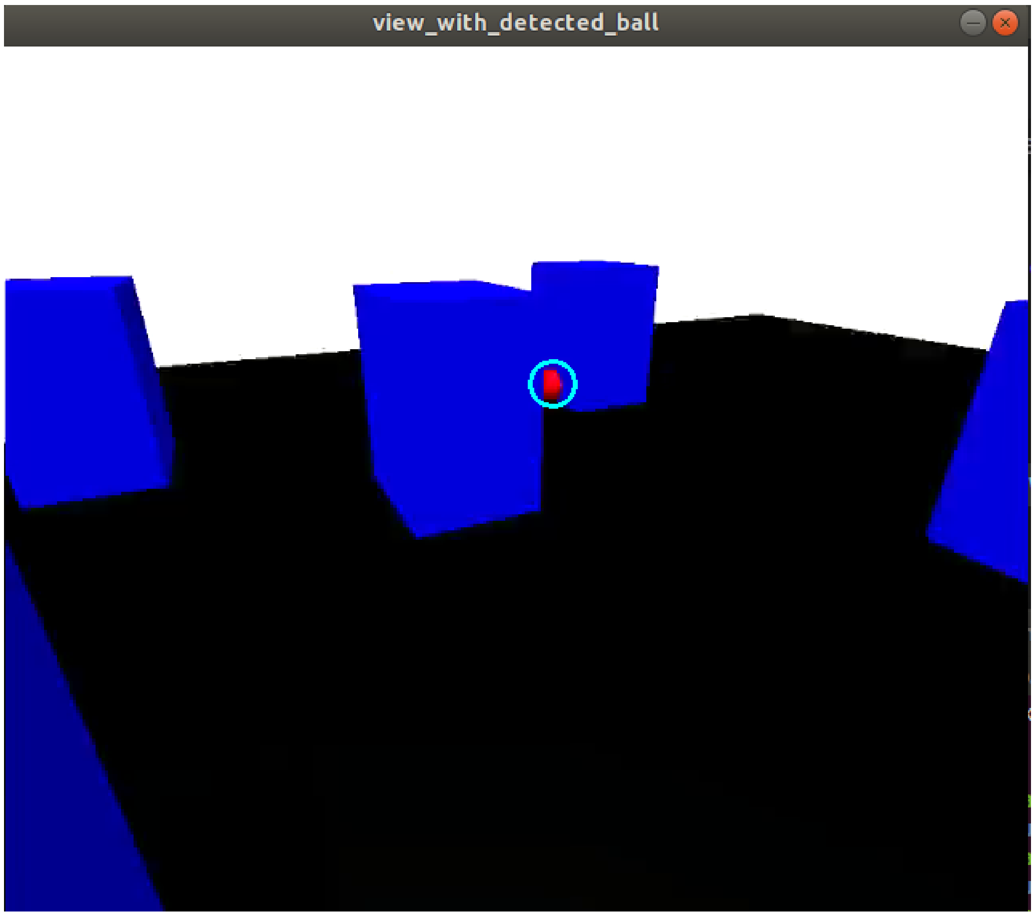

Figure 16: Ball detected behind an obstacle

References

[1] DAInamite Team Description 2016, Yuan Xu, Martin Berger, Qian Qian and Erdene-Ochir Tuguldur, 2016

[2] https://opencv.org/
`(accessed at 27.2.2021)`

[3] https://threejs.org/
(accessed at 27.2.2021)

[4] https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html

[5] https://docs.opencv.org/master/dd/d6a/classcv_1_1KalmanFilter.html
(accessed at 27.2.2021)

[6] Greg Welch and Gary Bishop. An introduction to the kalman filter. 2006
https://www.cs.unc.edu/welch/media/pdf/kalman_intro.pdf
(accessed at 27.2.2021)

[7] https://solarianprogrammer.com/2015/05/08/detect-red-circles-image-using-opencv
(accessed at 27.2.2021)

[8] https://docs.opencv.org/3.4/df/d0d/tutorial_find_contours.html
(accessed at 27.2.2021)

[9] https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html
(accessed at 27.2.2021)

[10] https://docs.opencv.org/3.4/d4/d70/tutorial_hough_circle.html
(accessed at 27.2.2021) https://docs.opencv.org/3.4/dc/dcf/tutorial$_js_contour_features.html$