Universitetet i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering: Bachelor i ingeniørfag / Data	Vårsemesteret 2023 Åpen
Forfatter(e): Carl Magnus Elvevold, Huy Erik Truong og Petter Uy Hoang	
Fagansvarlig: Tormod Drengstig Veileder(e): Tormod Drengstig	
Tittel på bacheloroppgaven: Multiplayer spill med EV3 mindstorm-roboter Engelsk tittel: Multiplayer game with EV3 mindstorm-robots	
Studiepoeng: 20	
Emneord: Augmented reality, image processing, robotprogramming	Sidetall: 64 + vedlegg/annet: 85 Stavanger 29. mai 2023

Preface

In our bachelor thesis, we created something that was interesting to each of us. When we started working, we did not expect that the project would be showcased at the video-game convention SpillExpo, held in Stavanger from the 12th to the 14th of May 2023. To showcase the thesis, we created a zombie survival game which worked as a great demonstration for our thesis. We want to give special thanks to Didrik Efjestad Fjereide, Jostein Hagen Lindhom, Tom Ryen, Ståle freyer, and Per Jotun, all from the Department of Electrical Engineering and Computer Science at the University of Stavanger. They offered us extensive assistance in preparing for the convention, including providing us with the infrastructure, robots and necessary equipments.

We would also like to thank Tormod Drengstig, our advisor, for all the support and great feedback we got during the entirety of the semester.

Abstract

Our thesis is about creating an online augmented reality (AR) experience where users control physical Lego EV3 Mindstorm robots to perform virtual tasks. The robots are equipped with cameras in front to allow users to observe and interact with the environment. Image processing is used to estimate the position and orientation of the robots. The estimation are used with a virtual environment to simulate an AR experience.

Contents

Summary

In this thesis, we have created a game where users controls physical robots to survive against virtual zombies. The robots can drive around within a fixed area, and shoot the zombies. The game can be played with any number of players, each with their own robot.

Each robot is equipped with a camera, and allows the users to see where they are driving. We use *Three.js* to display the zombies and the bullets within this view, making it look like the zombies are walking towards the robots in the real world.

Each robot has marker on top. These markers are filmed by a camera which is pointing down towards the robots. We then use image processing algorithms on the video to estimate the position and orientation of each robot using those markings. This information is then used to determine where the virtual zombies should be displayed for each player, and which direction the robots are shooting.

To allow users to easily access and play the game, we created a website. The website allows users to both start and play the game, as well as registering their score after the game is finished.

The game was tested thoroughly during the video-game convention mentioned earlier, which provided us with much knowledge about what worked, and what did not work.

Chapter 1

Introduction

Over the past decades, robots have been increasingly applied to a wide range of fields including industry and medicine. These machines were originally designed to perform autonomous tasks that range from simple and repetitive to complex [12].

In recent years, augmented reality (AR) games has grown in popularity [14], which opens new worlds of possibilities when it comes to video-games and other virtual experiences. With AR, virtual objects can be placed and interacted within the physical world, and seen using a screen.

This thesis aims to combine robotics with video-games, and to use AR technologies to create a virtual game where players can control real physical robots to play the game. The game is a multiplayer game where the players have to collaborate to survive against virtual Zombies, drive around to avoid them, and shoot the zombies to get points.

The project was also showcased at a video-game convention in Stavanger, SpillExpo, from the 12th to the 14th of May 2023, providing much insight in how the game performed.

A video-demonstration of the project can be watched using this link: `https://youtu.be/RvCEhO_Czrk`.

Chapter 2

Devices and setup

The game requires robots, cameras, computers to play and servers to process information. This chapter explains all the different physical devices and how they relate to each other, as well as the game itself.

2.1 Overview of all devices and physical connections

All devices are connected to a single endpoint, the server. They are either connected using WiFi, or by using Ethernet cables. The topology diagram in figure 2.1 shows how each device is connected, as well as which domain it belongs to.

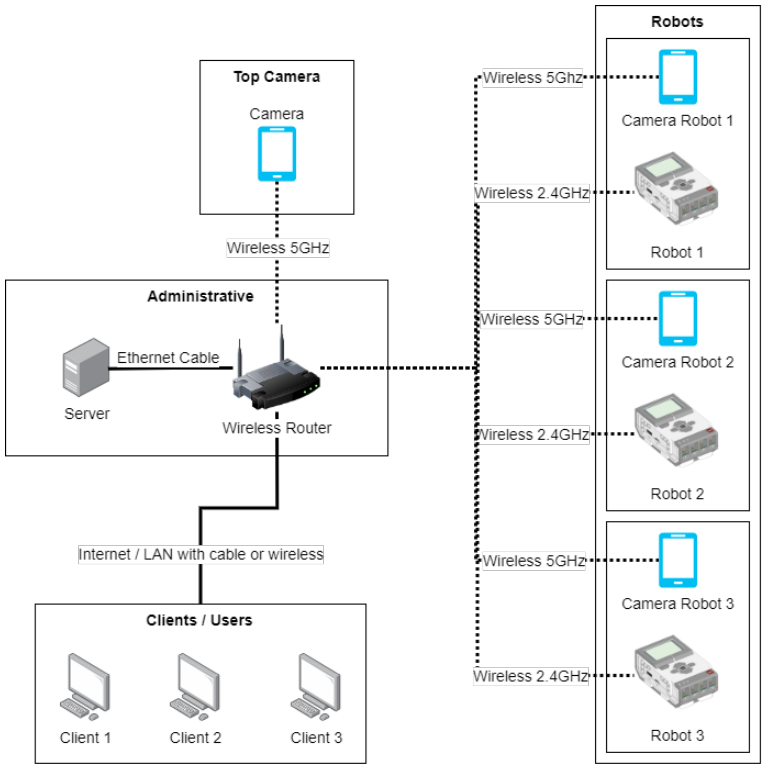


Figure 2.1: This figure shows the different domains of our project, the way it is set up for demonstration. While no end-devices are directly connected to each other, they do communicate through the server. Solid lines shows a wired connection, while dotted lines shows a wireless connection.

2.1.1 The physical devices

The robots does not have built-in cameras, so external cameras was used. All cameras, except for the top camera, was mounted in front of the robot to act as the "eyes" of the robot, which allowed the players to see the world where they were driving. Figure 2.2 shows the setup of a robot with a camera.

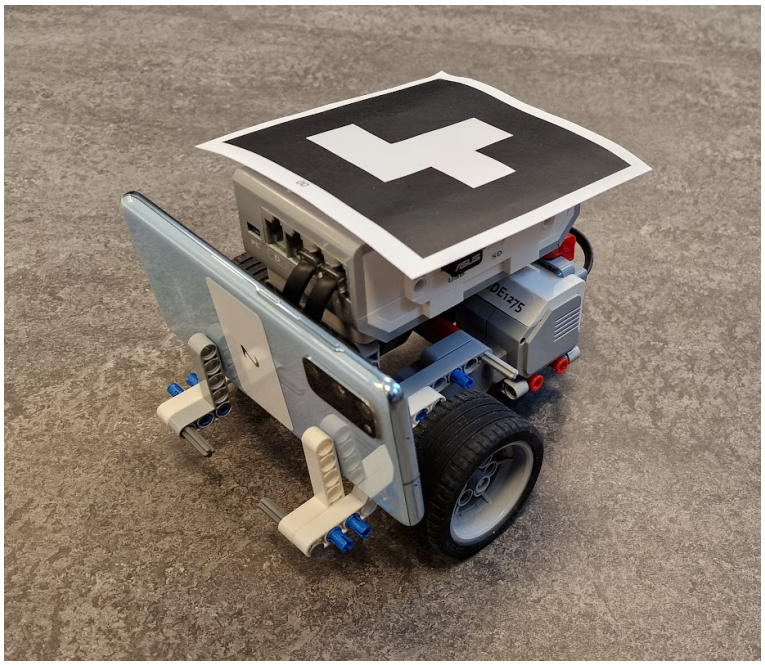


Figure 2.2: The robot are mounted with a camera in front, which in this case was a mobile phone. There is also a marker on top which contains both identification and pose-information. This marker is called ArUco, and is explained in detail later in chapter 4.1

To contain the robots within a fixed space, as well as to keep the top camera safe and steady, a structure was designed and constructed specifically for the demonstration. We measured the size of the area that the camera covered when mounted two meters above the ground. This defined the length of each side, and ensured that the entire area which the robots could move would be covered by the camera. This structure is shown below in figure 2.3, which also shows the placement of the top camera in relation to the robots.

Figure 2.3: The top camera, marked with a green circle, is mounted two meters above the robots. The camera is streaming a continuous video feed which is processed by the server to determine the position and orientation of each robot.

2.2 A brief explanation of each domain.

The physical setup is split into different domains, but can differ depending on the deployment and use-case, as more or less clients and robots may be preferred. An explanation of the different domains follows:

2.2 A brief explanation of each domain.

- Administrative
  - What goes on "behind the scenes".
  - The router sends information from the server to the devices, and from the devices to the server.
  - The server is what all devices connects to. It both sends and receives data from the different devices, as well as processing the data it receives.
- Top Camera
  - The camera sends a live video feed of a "birds-eye" view of the playing area, which contains crucial information about position and orientation of the robots. See figure 2.3
- Robots
  - Each robot consists of two different devices. One is the EV3 device itself, and the other one is a camera, which in our case is a mobile phone. See figure 2.2
  - There can be between 1 and 10 robots connected. We used two for our demonstration.
- Clients / Users
  - The clients are the computers which is used to play the game by the players. They connect and play using their web-browser of choice.
  - There must be as many clients as there are robots connected.
- Connections
  - The EV3 devices and cameras are connected to the server using WiFi to allow them to move freely without a cable.
  - For our demonstration, the clients were connected with Ethernet cables using the local network. The application can however be set up to be played over the internet.
  - The server is connected via an Ethernet cable for the best possible bandwidth and stability.

The number of clients and the number of robots (with cameras) can both increase and decrease depending on need, as long as the number of clients is equal the number of robots.

2.3 The robots

The robots we used were the Lego Mindstorm EV3 robots. They were built using the building instructions for "Driving Base" found on the Lego education website [8], with a small modification in front for a phone-holder. Figure 2.4 shows one of the robots built with the instructions.

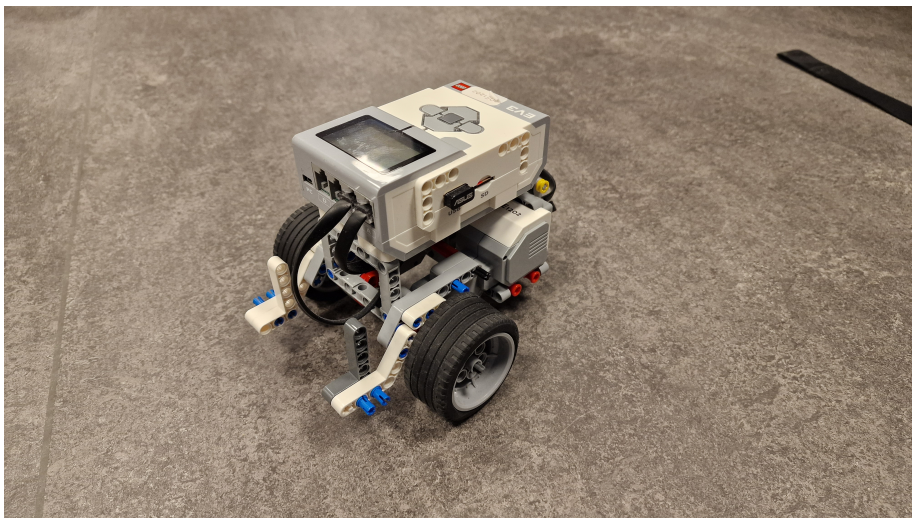


Figure 2.4: The robot when built with the instructions. The two pieces sticking up in front of the robot are for keeping the phones steady in front.

2.3.1 Robot Setup

The EV3-robots does have a built in operating system, but it is very limited. The robots was therefore equipped with the EV3Dev operating system. This is an Open Source Linux-based operating system made by EV3Dev [5], and it offers much more options when it comes to programming, connecting and controlling the robots and accessories connected to them. This only requires flashing the EV3dev image file to a memory card, which when plugged into the memory card slot on the robot, will make it boot automatically to the EV3Dev system. The robots then supports Pybricks, an open source software made for EV3Dev, which makes it easy to create and run python scripts [15] with control over the motors and other accessories connected to

the robots.

Since the EV3dev system is a Linux-based operating system, the Linux-terminal is accessible remotely using SSH, which makes modifying different aspects of our robot possible, like setting up WiFi and changing the name of the robot. This made it much easier to differentiate between the different robots when connecting to them, since they are pre-configured with the name "EV3Dev". A short guide to do this is in Appendix A.1

2.3.2 WiFi

The EV3-devices does not come with built-in WiFi. However, with a wireless network adapter, they can be configured to support it. These may or may not work out of the box, so a forum post explaining which adapters are known to work [16] was found to be really helpful. The robots were therefore equipped the Asus N10 Nano adapters, which supports the 2.4GHz radio frequency for WiFi. The adapters did not work out of the box, but fortunately, someone else had made a driver for them [23], which allowed us to use them.

WiFi allowed us to configure, connect and run the robots completely wireless. An alternative would be to use Bluetooth which is built in, but in our case, the connection was very slow and unstable.

When the wireless adapters are configured correctly, a selection on the robots screen for connecting to WiFi becomes available. This is shown in the figure below.

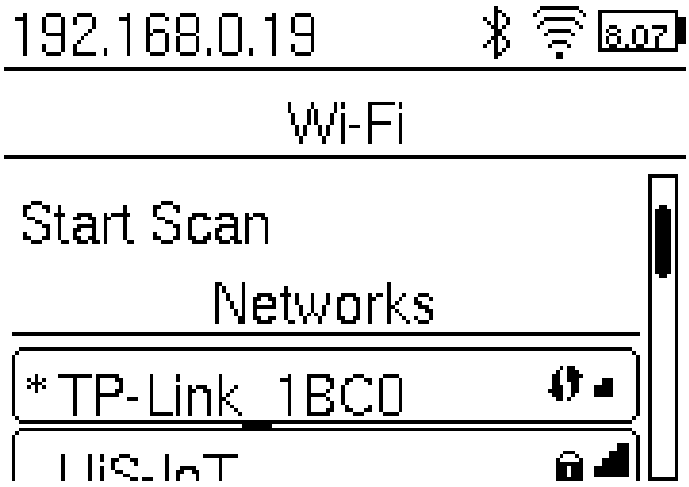


Figure 2.5: Using a WiFi-adapter allows the robots to connect to WiFi. This screenshot of the robots internal screen shows the menu on the robots which allows us to connect to different networks. The IP address is also shown at the top left corner of the screen.

2.4 The Cameras

The robots were equipped with android smartphones as we had a good enough supply of them to cover enough robots as well as the top camera, while also allowing for WiFi connections and many options for configuration. The same method was used for the top camera to keep things simplified and avoid using different methods and programs.

2.4.1 Challenges and limitations

This method came with some drawbacks. The mobile phones were bulky and heavy, which made the robots run slower. In addition, all the phones were different brands and models, so each phone had the camera in different spots. Another great difficulty was when multiple cameras sent a live video-stream at the same time over WiFi, the latency increased, causing lag and stability issues.

To deal with the issue of latency, the robot-cameras was set to stream a low resolution at 640x480. For the top camera however, it had to have a much higher resolution to be able to detect ArUco-markers, and was set to 1920x1080. All cameras were also set to only capture 10 images per second, instead of the default 30. The lower resolution did however lessen the quality of the streams for the players, and less frames from the top-camera will also cause the rendering to not be as smooth.

2.5 Clients and server(s)

The server is responsible for all server-side processing such as the top-camera image processing, as well as the connections to all devices, including the clients. The server runs three programs, which effectively counts as three different servers. The "Frontend server", the "Backend server" and the "Database server".

The connections between each server and the devices is shown in the figure below:

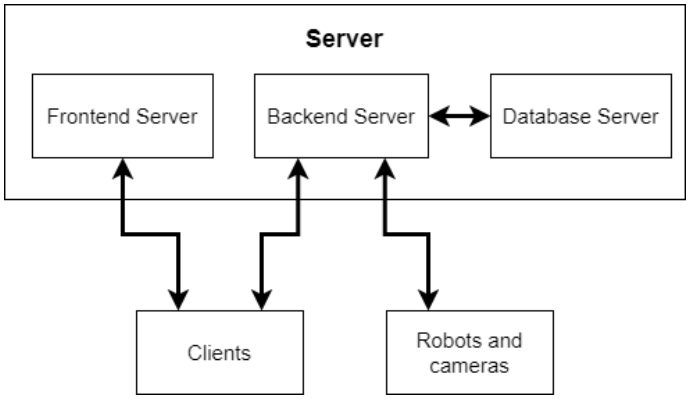


Figure 2.6: The server has three different programs running, where each has its own responsibilites. The Frontend Server gives the Clients the scripts they need to connect to the Backend server. The Backend server is also connected to the database server for storage.

The Backend server is the program that handles connections to devices, image processing, temporary storage of game-data and sending all relevant data, such as the video-feed from a specific robot, to the specific user. This server is also responsible for storing and retrieving data from the database server, which manages the databases and allows the Backend server to request and store data in them for long-term storage.

The Frontend server is a different program, and is the one that the clients connects to. The Frontend server serves the web pages to the clients, as well as serving the scripts which allows the clients to communicate with the Backend server.

Chapter 3

Backend and database Server

The Backend server handles connections, image processing, sending data to devices, and receiving data from devices. This chapter explains the Backend server, how it manages connections with the devices and how it stores data in the database.

3.1 Communication with clients using a web server

The Backend server is a web server, and connects to the devices over a network. The web server is based on Flask [22], which is a web framework for Python. With Flask, we built several *Application Programming Interfaces* (API), which sends and retrieves data from the client. These APIs are either in the form of simple *Uniform Resource Locator*, URLs, or by using sockets, which allows for real-time two-way communication between the server and the client [19]. The socket communication between the Backend server and the clients use the flask-socketIO library, which is a socket library specifically made for flask applications [7].

We also use **cookies** to identify which client is sending a request. A cookie is a small piece of information which in our case contains a unique ID for each client [18]. These are stored both at the Backend server and the client, and is sent with each request that the client makes with the Backend server.

3.2 Socket address and video capture

IP Webcam is an Android application that enables video to be streamed over WiFi within a network[13]. The video stream is accessed with a socket connection shown in figure 3.1. The socket is accessed using a combination of an Internet Protocol (IP) address and a port number. The **IP address** uniquely identifies a device (in this case the phone) connected on the network. The **port number** is used to keep track of different connections on the same device. The video stream is obtained from the resource /**Video**.

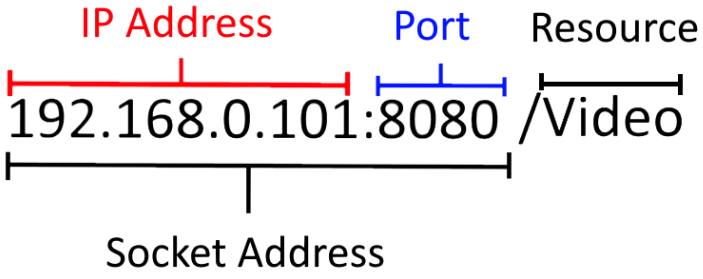


Figure 3.1: This figure illustrates a socket address for an Android phone using IP Webcam. The */Video* endpoint is used to access the video stream of the phone.

The video stream is processed with `cv2.VideoCapture()` in OpenCV shown in code 3.1. In order to process the most recent frame (an image in a video stream), we chose to use the function `grab()` (line 3) and `retrieve()` (line 4) instead of the traditional `read()`. The `read()` method combines both `grab()` and `retrieve()`. However, using `read()` blocks execution of code which makes the video stream delayed when the network is experiencing high latency. The `grab()` method has low overhead because it avoids frame decoding. The `retreieve()` method decodes the frame and returns a *numpy array* of pixels which is the preferred format for processing frames using Python in OpenCV.

Code 3.1: Code snippet for retreiving video frames in Python

```
1 video_address = "https://192.168.0.101:8080/video"
2 cap = cv2.VideoCapture(video_address)
3 grabbed = cap.grab()
4 retrieved,frame = cap.retrieve() if grabbed else ...
      (False,None)
5 self.latest_frame = frame
6 ...
```

3.3 Backend game configuration

The game configuration handles all connections to every device, as well as running the game itself.

The game configuration is a python class where all data relevant to the game is kept, and contains methods which is used to manipulate this data in order for the game to work. It keeps track of the different robots and their cameras, the clients, as well as the state of the game, for example if players are alive or not, how many points they have etc. The game configuration utilizes the other scripts within the directory, which are responsible for handling connections, rendering etc.

The game configuration is configured with a network port that allows the robots to connect to the server. We chose the port 18080 arbitrarily, as it was not in use by any other program we used. This is the main port used for

connecting to the server. We then specified the ports from 18081 to 18090 to dynamically assign to the specific robots, which allowed for a total of 10 robots connected at any given time. The ports and ranges can be changed should more robots be needed, or if the ports are already in use.

3.3.1 The robots internal configuration and scripts

The robots themselves contains scripts for connecting to the Backend server, as well as handling the instructions it receives. There are two scripts, where one of them keeps configuration data.

```
Robot Scrips/
├── Client.py
└── config.py
```

The config.py file contains the configuration data for each robot. An example is given below.

```
1     "ide1202": RobotData(
2                 robot_name="ide1202",
3                 wheel_radius=0.0275,
4                 wheel_dist=0.114,
5                 QR_code = 4,
6                 marker_size=0.11,
7
8                 camera = 1,
9                 forward=0.075,
10                right=-0.05,
11                up=-0.055
12                ).__dict__,
```

This file consists of information used to identify the robots, the size of the robots, and the ID of the camera it is assigned to. It also contains the IP address of the server, as well as the main port 18080.

Connecting robots to the server

The robots Client.py file is the main script. When it is executed, it will attempt to connect to the server. The robots will try to connect to a socket using the IP and main port number of the server. If it is successful, the server will store the robots data.

The server will then find a random available port from the range specified above in section 3.3, and send it through the socket back to the robot. The first socket is then closed, and a new one is opened using the new port number. This port number is then used for all further communication between the server and the specific robot.

3.3.2 The robots camera

As shown in figure 2.2, the camera is mounted in front of the robot. The cameras has their own unique ID which is arbitrarily chosen and saved in the Backend server. It serves multiple purposes:

- The ID allows for easy identification of the different cameras
- It allows for easy configuration when assigning a camera to a robot
- It is saved alongside the IP address of the phone to make connection easier
- If a camera requires additional files and configurations, they can be found using the phones ID.

3.3.3 The game loop

The Game Loop is initialized when the Backend server starts, and is responsible for starting the game, keeping the game running, checking if there are changes to the game state, starting new rounds if all zombies are killed, spawning new zombies, and resetting the game if all players are dead. This is illustrated as a flowchart in figure 3.2. This loop will run continuously throughout the lifetime of the program.

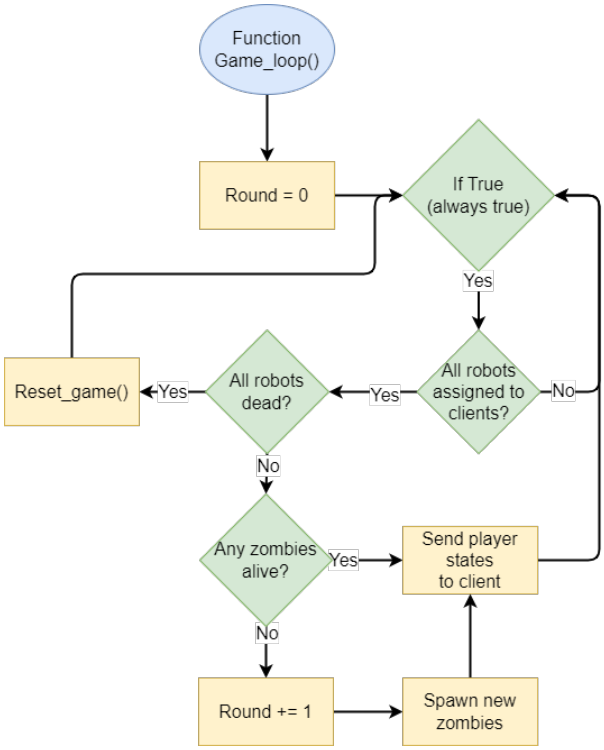


Figure 3.2: The game loop starts the game when all robots are ready, checks if players are dead, and starts new rounds when all zombies are killed.

3.4 The main application

The main application script is what is executed to start the Backend server. It initializes the flask application, the sockets, the APIs and the game configuration.

The flask application is initiated with the following code:

```
1       app = Flask(__name__)
```

The app variable contains a config which can be assigned different variables, this config is accessible anywhere in the application, and allows us to pass around the different instances, like the socket instance, to other parts of the application, like the game configuration.

Flask-socketIO, mentioned in section 3.1 allow for functions to be run concurrently, allowing us to interleave work between all robots, cameras and clients. This is achieved using Eventlet and Background tasks. Eventlet is a python library which allows and handles concurrency [1] and we configured Flask-socketIO to use it, which then allows us to run background-tasks.

The following code snippet shows how we use the Flask config and the Flask-SocketIO to initialize the different instances of for example the Top Camera, as well as how we set different functions to run concurrently.

```
1       # initializing SocketIO
2       app.config['socketio'] = SocketIO(app, ...
           async_mode='eventlet', ...
           cors_allowed_origins="*")
3
4       # initializing the game config
5       app.config["gameconfig"] = GameConfig()
6
7       #Process using top camera to estimate each ...
           robot's pose (position and orientation)
8       Process(target=TopCamera, ...
           args=(app.config["gameconfig"],True)).start()
9
10      # Starting background tasks for streaming ...
           video and pose estimation
11      app.config['socketio'] ...
           .start_background_task(game_loop,app ...
           .config["gameconfig"])
12
13      # adding socketio attribute to gameconfig
14      app.config["gameconfig"].socketio = ...
           app.config['socketio']
```

3.5 Database

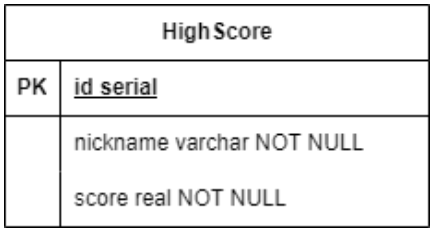
After each game, the players is prompted with the option to register their score. If they register, the score is saved in a database. Every game is saved for statistics-purposes.

For the purposes of the application, advanced database-features are not necessary, and most database-systems would suffice. We settled on PostgreSQL because of familiarity, but even a simple JSON-based database

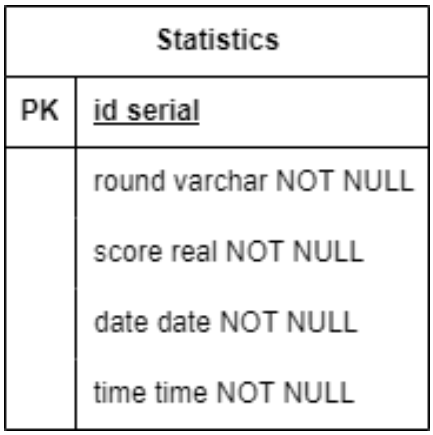
would be enough for our demo purposes. PostgreSQL does however offer many advanced features and can be a good choice should this project be expanded.

PostgreSQL runs on its own database server which the Backend server connects to, allowing the it to store information long-term, and retrieve the information when needed.

The database consists of two tables. One for those who registered, see figure 3.3a, and one for internal statistics, see figure 3.3b.



(a) The table stores the score of the game along with a nickname of the players choice.



(b) The table for statistics contained some general information about each game.

Figure 3.3: The tables for high-score and statistics

Chapter 4

Camera and image processing

In this section we describe how OpenCV library in python is used for image processing. The library includes a variety of algorithms for both computer vision and machine learning. Specifically, our focus is on detection and pose estimation of ArUco markers. To account for instances where markers are not detected due to lighting conditions, we introduce a redundancy system for pose estimation using angular speed from the wheels on the robot.

4.1 ArUco Markers

ArUco markers are used in computer vision to identify reference points in images [24]. Information from ArUco marker in our project is primarily used for;

- detection of ArUco markers in images from a video stream
- estimating position and orientation (pose estimation) of the ArUco marker on top of the robot

ArUco markers are made of black and white squares forming a specific recognisable pattern for detection. Figure 4.1 illustrates two ArUco mark-

ers of respectively 4x4 and 6x6 dimension. The black surrounding border around ArUco markers improves detection by increasing the the contrast to surrounding environment [21]. The dimension of an ArUco marker is determined by the number of white and black squares within the red bounding box illustrated in figure 4.1.

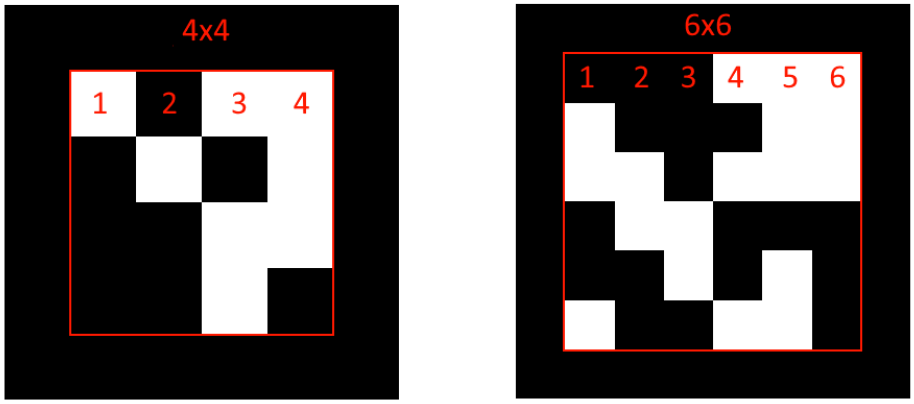


Figure 4.1: Illustration of ArUco markers with different dimensions. The red box contains a number of black and white squares which determines the dimension of the marker. The ArUco marker on the left side is a 4x4 matrix and the marker on the right is a 6x6 matrix. ArUco markers were made with arucogen [11].

In order to detect different markers of the same dimension, a data structure is needed to contain this information. A *marker dictionary* is a collection of markers with the following properties [21];

- marker dimension
- dictionary size

The **marker dimension** parameter controls the number of squares within the red box, as shown in Figure 4.1. The marker dimension is specified using grid notation, such as 4x4. By choosing a smaller marker dimension, the individual squares become larger, enhancing detection of markers at longer distances. The **dictionary size** parameter determines the number of unique markers available in the dictionary. A smaller dictionary size

reduces computational overhead by searching through fewer entries in the dictionary. In our project, we found that a single marker per robot was enough. Taking into account the limited number of markers required, we opted for a 3x3 ArUco marker, shown in figure 4.2.

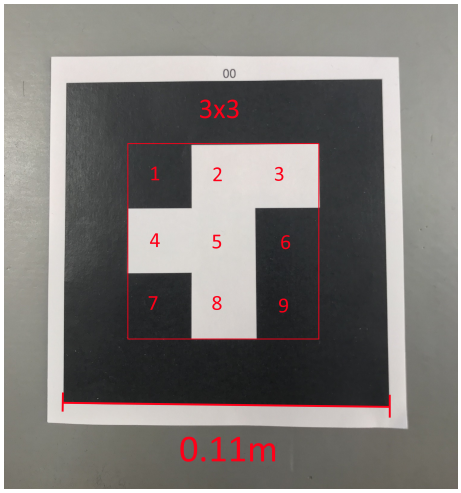


Figure 4.2: Illustration of ArUco marker with 3x3 dimension. The red box contains 9 squares of white and black color. Each individual square is larger than the traditional 4x4 ArUco marker. The physical size of the marker was measured to be 0.11 meters.

4.2 Camera calibration

Detection of ArUco marker in OpenCV using `cv2.aruco.detectMarkers()` is not dependent on camera calibration. This is because the detection algorithms looks for specific patterns in the ArUco marker. However, estimating position and orientation (pose-estimation) of the marker using `cv2.aruco.estimatePoseSingleMarkers()` requires the camera to be calibrated. This is because calibration mitigates distortion and establishes reference points for measurements of distances.

4.2 Camera calibration

In order to measure the pose of an ArUco marker, a phone camera was calibrated with the checkerboard-pattern calibration. This calibration method is a standard calibration method in OpenCV [20]. The purpose of calibration is to find the **intrinsic parameters** of the camera. The standard code for camera calibration in OpenCV using Python is shown in code 4.1. The corners of the checkerboard are found in each image with `cv2.findChessboardCorners()` (line 15). To visualize the corners, `cv2.drawChessboardCorners()` (line 25) was used. Figure 4.3 illustrates the corners detected in one of the images. Calibration is done with `cv2.calibrateCamera()` (line 27) which returns the intrinsic parameters (camera matrix and distortion matrix). The distortion matrix is used to correct for distortion in images before applying pose estimation algorithms.

Code 4.1: Overview of camera calibration code in OpenCV using Python[20]

```
1  # physical size of a square (meters)
2  square_size = 0.02833
3
4  # define object points
5  objp = np.zeros((height*width, 3), np.float32)
6  ...
7  # apply physical properties
8  objp = objp * square_size
9
10 objpoints = []  # 3d point in real world space
11 imgpoints = []  # 2d points in image plane.
12
13 for i in range(images)
14     # Find the chess board corners
15     corners = cv2.findChessboardCorners(...)
16
17     # Add predefined 3d points
18     objpoints.append(objp)
19
20     # Refine corners and add these to imgpoints
21     corners2 = cv2.cornerSubPix(...)
22     imgpoints.append(corners2)
23
24     # Visualize corners on the chessboard
25     img = cv2.drawChessboardCorners(...)
```

```
26
27 mtx, dist = cv2.calibrateCamera(objpoints, ...
       imgpoints)
28 np.save("calibration_matrix", mtx)
29 np.save("distortion_coefficients", dist)
```

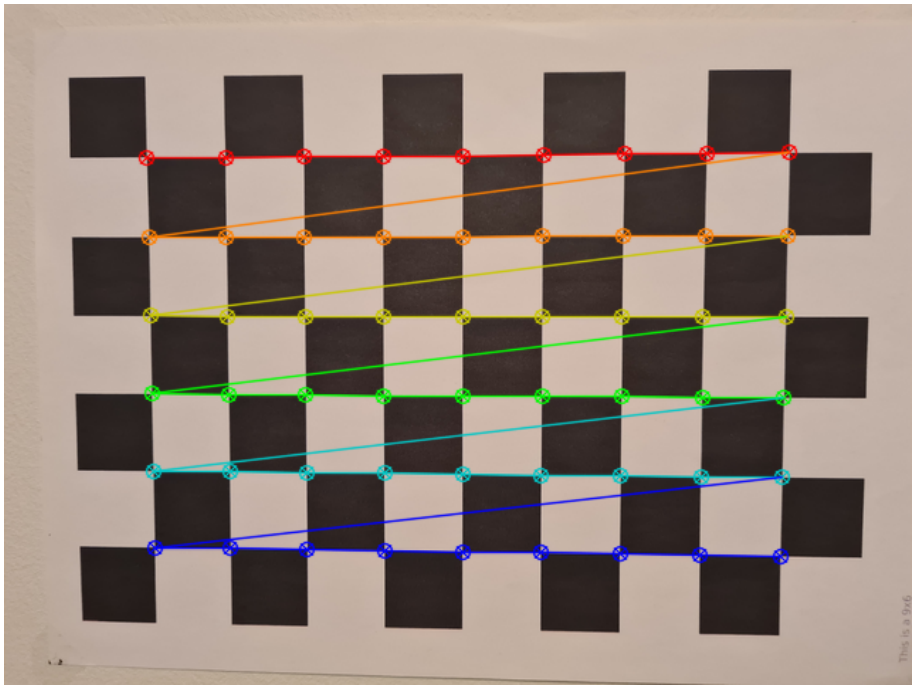


Figure 4.3: The figure illustrates the standard method of checkerboard-pattern calibration in OpenCV-Python. A phone camera was used to take images of the checkerboard provided by OpenCV. The corners are visualized with the function `cv2.drawChessboardCorners()`.

4.3 Detection and pose estimation of ArUco markers

In OpenCV, ArUco markers are detected with the function `cv2.aruco.detectMarkers()`. In order to improve detection, the image is converted to grayscale shown in figure 4.4. This reduces color information to be processed and increases

contrast between the black and white pixels. Another benefit is that it improves detection of ArUco markers in challenging lighting conditions. Code 4.2 converts the image to gray-scale (line 2) before applying the ArUco detection algorithm (line 3). `cv2.aruco.detectMarkers()` returns identifiers, `ids`, and pixel position of the corners, `corners`, for each ArUco marker detected in the image.

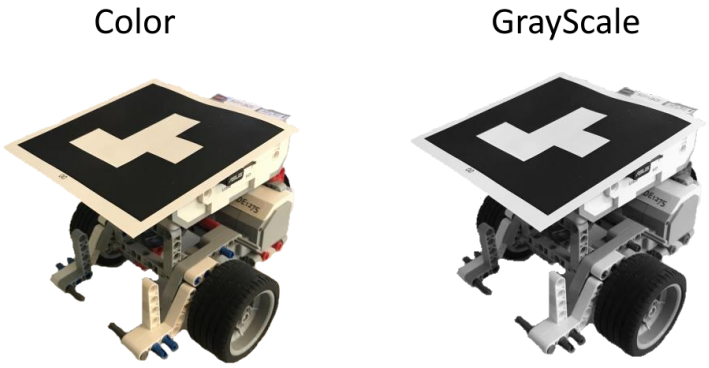


Figure 4.4: This figure illustrates image conversion to grayscale in OpenCV. The grayscaled image takes less time to process because it has less color information. The grayscale improves detection by increasing the contrast between black and white pixels.

Code 4.2: Code snippet for detecting aruco markers in Python

```
1 # If the image is valid, perform processing
2 img_gray = cv2.cvtColor(self.latest_frame, ...
      cv2.COLOR_BGR2GRAY)
3 corners, ids, _ = ...
      cv2.aruco.detectMarkers(img_gray, arucoDict, ...
      parameters=arucoParams)
```

The transformation between the image coordinate and the camera coordinate is illustrated in figure 4.5. The image coordinate uses pixels to describe the position, while the camera coordinate is calibrated to use meters. The conversion from image coordinates to camera coordinates is done with with `cv2.aruco.estimatePoseSingleMarkers()` shown in code 4.3 on line 12. This function takes in the intrinsic parameters of the phone camera (see section 4.2), the physical size and the corner coordinates of the ArUco marker.

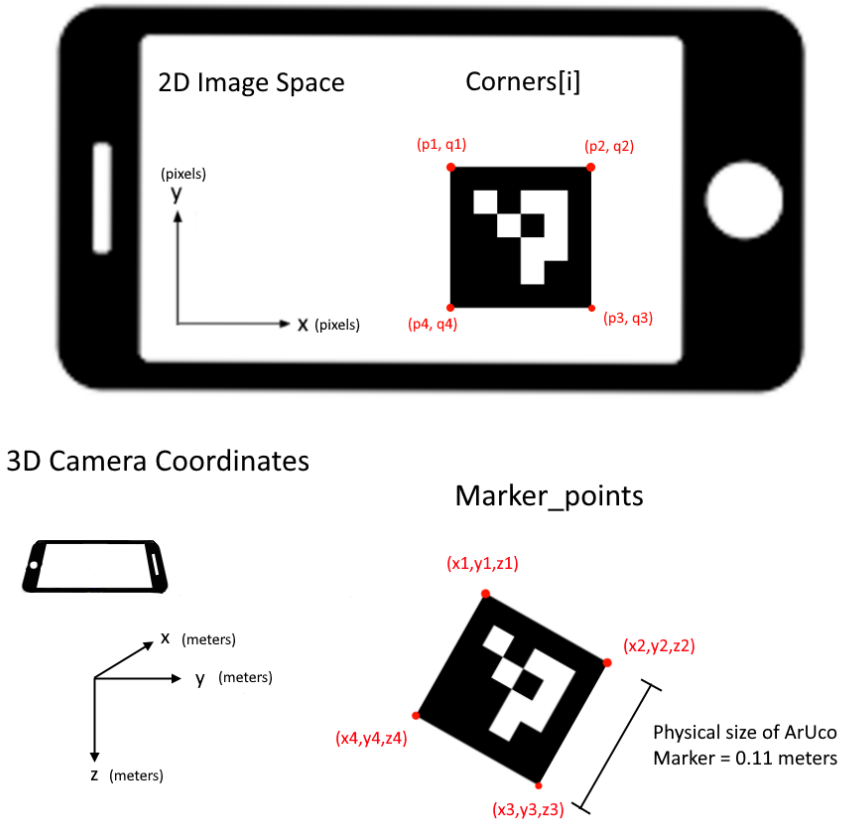


Figure 4.5: This figure illustrates mapping of corners on the ArUco marker in 2D image coordinates and 3D camera coordinates with `cv2.aruco.estimatePoseSingleMarkers()`. The z-axis in the camera coordinate is pointing down in the direction of the camera lens. The x and y axis are horizontal to the phone. The physical length of a side on the ArUco marker is measured in meters.

The **translation vector**, `tvec`, represents the 3 dimensional position for the center of the marker in camera coordinate frame. The **rotation vector**, `rvec`, is a 3 dimensional vector and is a compact representation for the orientation of the ArUco marker relative to the camera. This representation is known as a *Rodrigues vector* [3]. Unlike a 3x3 rotation matrix which contains 9 components, Rodrigues vector is a more compact representation of the orientation and saves storage space.

Code 4.3: Code snippet for detecting aruco markers in Python

```
1  # Iterate through each aruco marker detected
2  for i in range(len(ids)):
3      aruco_id = ids[i]
4
5      # args estimatePoseSingleMarkers():
6      # corners of the ArUco markers in 2D image ...
           coordinate
7      # size of the ArUco marker in meters 0.11
8      # camera matrix
9      # distortion coefficients
10
11     # Estimate the pose of the marker
12     rvec, tvec, marker_points = ...
           cv2.aruco.estimatePoseSingleMarkers(
13          corners[i],
14          self.marker_size,
15          self.mtx,
16          self.dist
17     )
```

By convention the positive z-axis in camera coordinates, denoted as *Zc*, points in the direction of the camera lens (down) shown in figure 4.6. In code 4.4 on line 1, the rotation vector, `rvec`, is converted to a rotation matrix. From the rotation matrix, the directional components `front_vector` on line 10 and `right_vector` on line 4 are extracted. To ensure that these vectors are parallel to the ground, the z-components are disregarded and set to zero (line 5 and 11). The `up_vector` points orthogonal to the ground in the opposite direction of the z-axis in camera coordinates show in line 16. The resulting vectors are normalized to *unit direction vectors* and stored

back into `rvec` on line 21. For our project, it is valid to assume that the top camera remains stationary throughout the calculations, and the camera lens is positioned to capture a downward view of the ground. By making these assumptions, we can apply the relevant calculations and algorithms to analyze the captured data in accordance with the project requirements.

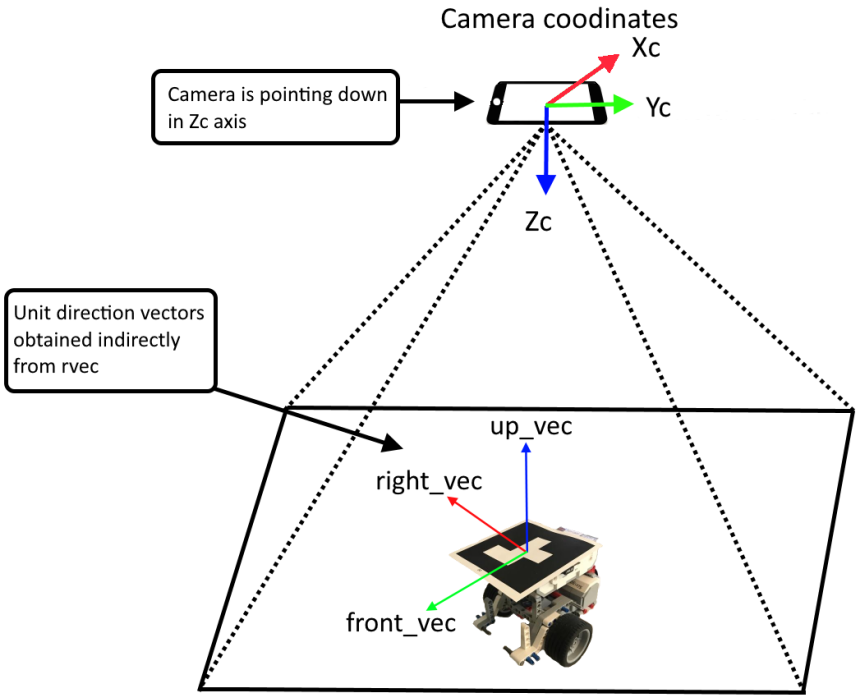


Figure 4.6: This figure illustrates the stationary camera position on top. The camera lens is capturing a downward view of the ground. `front_vec`, `right_vec` and `up_vec` is obtained by converting `rvec` to a rotation matrix and extract each component. The vertical component of `front_vec` and `rvec` are removed and normalized. The resulting components are stored back into `rvec`.

Code 4.4: Code snippet for extracting of unit vectors from rvec

```
rotation_matrix = cv2.Rodrigues(rvec)[0]

# positive x-component is right
right_vec = rotation_matrix[:,0]
right_vec[2] = 0
right_vec = right_vec/np.linalg.norm(right_vec)
rotation_matrix[:,0] = right_vec

# positive y-component is forward
front_vec = rotation_matrix[:,1]
front_vec[2] = 0
front_vec = front_vec/np.linalg.norm(front_vec)
rotation_matrix[:,1] = front_vec

# positive z-component is up
up_vec = np.array([0,0,-1])
up_vec = up_vec/np.linalg.norm(up_vec)
rotation_matrix[:,2] = up_vec

# Convert the rotation matrix back to a ...
   Rodrigues vector and flatten it
rvec = cv2.Rodrigues(rotation_matrix)[0].ravel()
```

In context of our requirements, it was sufficient to express the orientation of the robot as an angle around the vertical axis perpendicular to the ground (`yaw`). The calculation of Euler angles from the rotation matrix is done in Appendix D.1 [17]. Figure 4.7 illustrates the three Euler components.

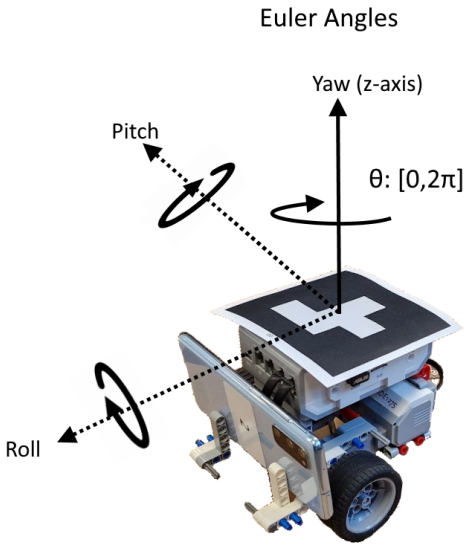


Figure 4.7: This figure illustrates the Euler angle representation of the orientation. Because the robot only rotates in the z-axis, the `yaw` component is used to describe this orientation.

4.4 Pose estimation with angular speed

The detection of ArUco markers are sensitive to variation in lighting conditions. To address this, the angular speed of the wheels were used as an alternative way to estimate the position and orientation of the robot [4]. Figure 4.8 illustrates the parameters of the robot used for pose estimation.

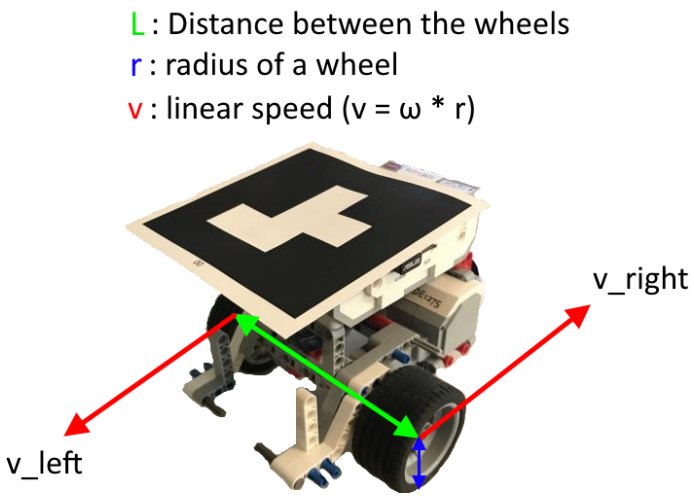


Figure 4.8: This figure illustrates the parameters used for pose estimation. The red arrow describes the linear speed, v_right and v_left for the respective wheel given in [m/s]. The green arrow represents the distance between the wheels and is given with the L parameter [m]. Radius of the wheel, r [m], is illustrated with the blue arrow.

The $linear\_speed$ [m/s] of the robot is calculated in equation 4.1. The variables r [m], ω [rad/s] are respectively radius and angular speed of the wheels.

$$linear\_speed = \frac{\mathrm{r}}{2} \cdot (\omega_{Right} + \omega_{Left}) \tag{4.1}$$

As mentioned earlier, `tvec` represents the center position of the ArUco marker in camera coordinates. Figure 4.9 illustrates how the new value for `tvec` is calculated. The blue vector in the figure scales the `front_vec` (green vector) with the scalar, $linear\_speed$. The `new tvec` is calculated using Euler Forward method by adding the contribution of the distance traveled during the time-step Δt to the previous position `tvec`.

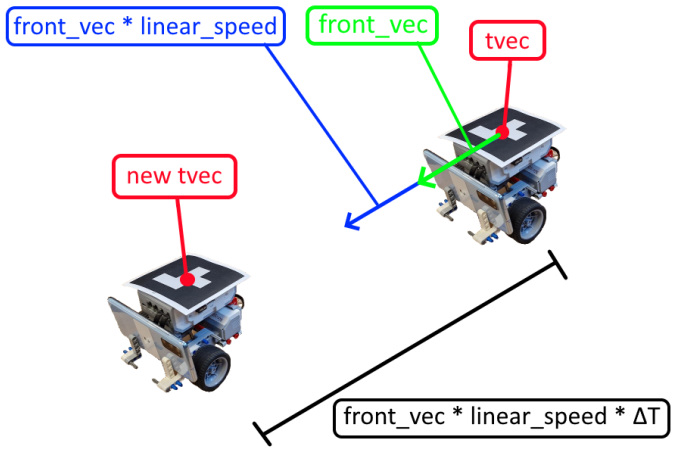


Figure 4.9: This figure illustrates the calculations for updating the robots new position `new tvec`. The distance traveled during a timestep Δt is added to the previous `tvec` in order to calculate the new position of the robot. The green vector `front_vec` is a unit vector that indicates the direction the robot is facing.

The equation for calculating the change in angle (θ [rad]) of the robot is given by equation 4.2. The variable L represents the distance between the wheels, as depicted by the green line in figure 4.8.

$$\theta = \frac{\mathrm{r}}{L} \cdot (\omega_R - \omega_L) \cdot \Delta t \tag{4.2}$$

In this project, the rotation axis always points perpendicular to the plane illustrated as $\hat{k}$ in figure 4.10. The angle θ obtained in equation 4.2 is used to rotate vector $\vec{v}$. The figure demonstrates rotation applied to the vector $\vec{v}$, represented by the green vector, around the axis $\hat{k}$ denoted by the black vector. As a consequence of this rotation, the vector $\vec{v_{rot}}$ is obtained, depicted by the blue vector. The cross product, $\hat{k}$ x $\vec{v}$ results in a vector perpendicular to both vectors illustrated with the red arrow. $\vec{v_{rot}}$ is obtained by adding each vector component in the respective directions.

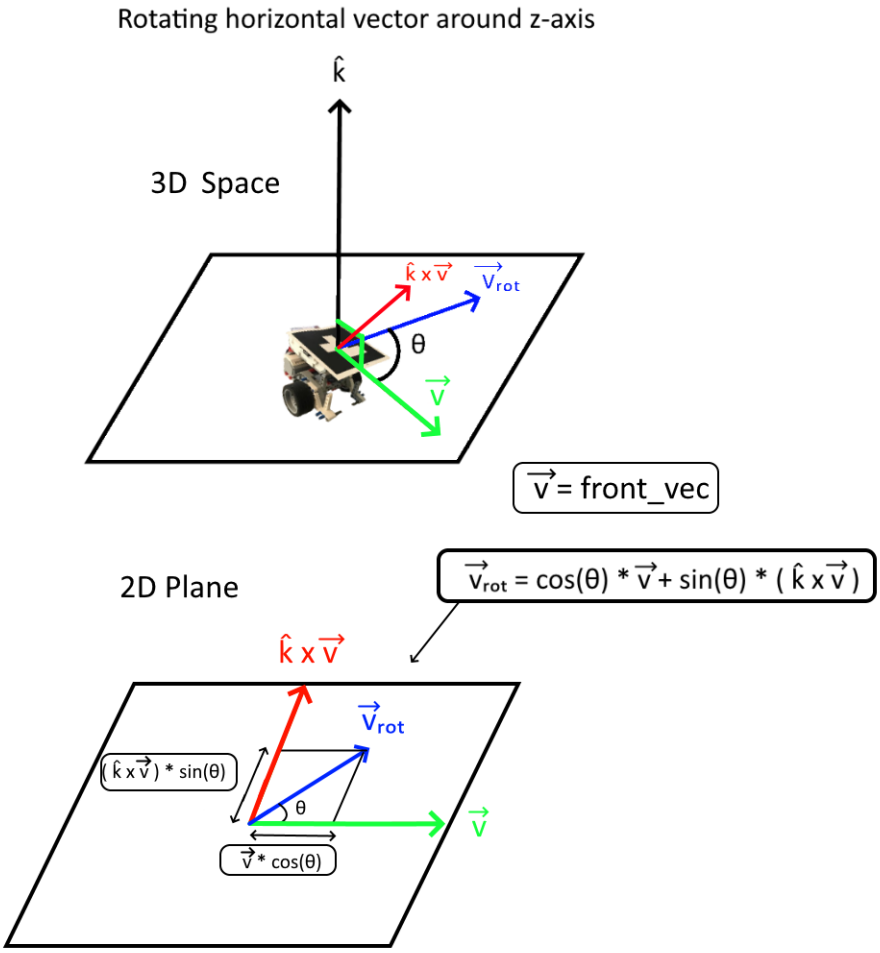


Figure 4.10: This figure illustrates rotation of vector $\vec{v}$ (green vector) around the axis $\hat{k}$. The outcome is $\vec{v_{rot}}$ depicted as the blue vector. The equation is derived with vector addition of each component along the directions of $\vec{v}$ and $\hat{k}$ x $\vec{v}$. This equation is only correct under the assumption that the axis of rotation $\hat{k}$ is perpendicular to the plane.

Code 4.5 shows the implementation of rotation formula in code. The change in angle, `theta`, is calculated on line 2. The axis of rotation `k` is defined on line 5. The rotation of `front_vec` is calculated on line 8 using the equation derived in figure 4.10.

Code 4.5: Code snippet for rotating the unit direction vectors with an angle θ around the axis $\hat{k}$.

```
1  # change in angle [rad]
2  theta = r/L * (w_R - w_L) * dt
3
4  # Axis of rotation
5  k = np.array([0,0,-1])
6
7  # Rotate front vector
8  front_vec = np.cos(theta) * front_vec + ...
       np.sin(theta) * np.cross(k, front_vec)
9
10 # Store the vectors back into the rotation matrix
11 rotmat[:,1] = front_vec
12
13 # Convert the rotation matrix back to rotation ...
       vector
14 rvec = cv2.Rodrigues(rotmat)[0].ravel()
```

In order to verify that the that calculations were done correctly, the pose estimation algorithm using OpenCV in Python (hereafter *camera estimation method*) was compared with the above calculations (hereafter *wheel estimation method*). Figure 4.11 illustrates how the experiment was performed. The camera estimation method continuously calculated `tvec` and `rvec` from the new frames obtained in the video stream. In contrast, the wheel estimation method only received the first `tvec` and `rvec` and used the angular speed of both wheels to estimate the new `tvec` and `rvec`.

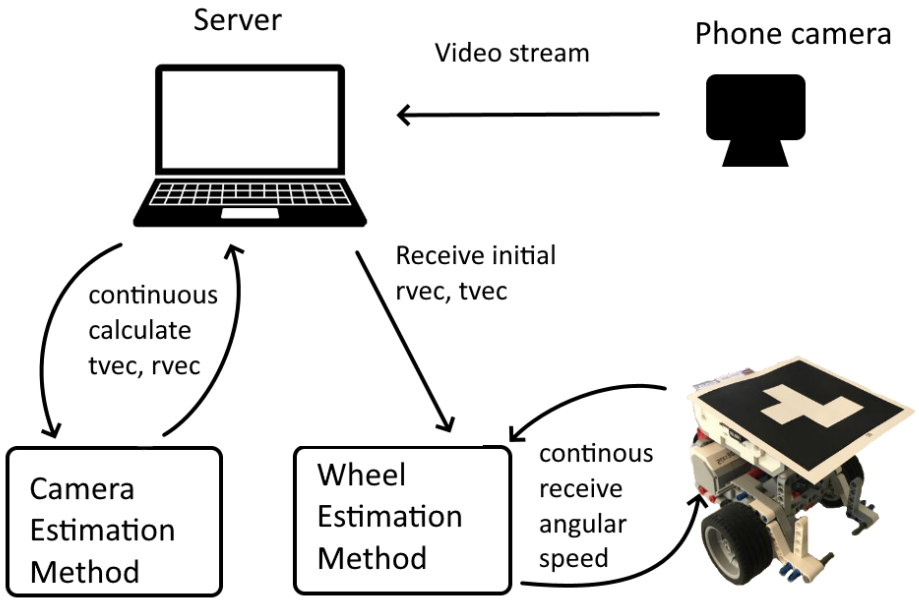


Figure 4.11: This figure illustrates the flow of data between two pose estimation methods. *Camera estimation method* uses OpenCV to continuously calculate `tvec` and `rvec`. *Wheel estimation method* uses the initial `tvec` and `rvec`. However, the angular speed of the wheels is constantly being received to calculate the new position and orientation.

Figure 4.12 illustrates the position (x and y component of `tvec`) for the robot driving in a straight line. The red line shows the calculation done with the *camera estimation method* by OpenCV. The blue line shows calculation with the *wheel estimation method* using angular speed of the wheels. The observation shows significant noise present in the red curve estimated with OpenCV. The main reasons is the distance between the markers and

the camera. As the distance between the camera and the ArUco markers increases, the physical size of the markers in the picture becomes smaller. This makes pose estimation become unstable. The blue curve is not affected by this noise because it only uses the first initial `tvec` and translates this position based on the angular speed of the wheels. Another observation is that there are more calculated points for the camera estimation method. This is because of a design choice. The angular speeds from the robot is sent to the server at a lower frequency to avoid impacting the bandwidth.

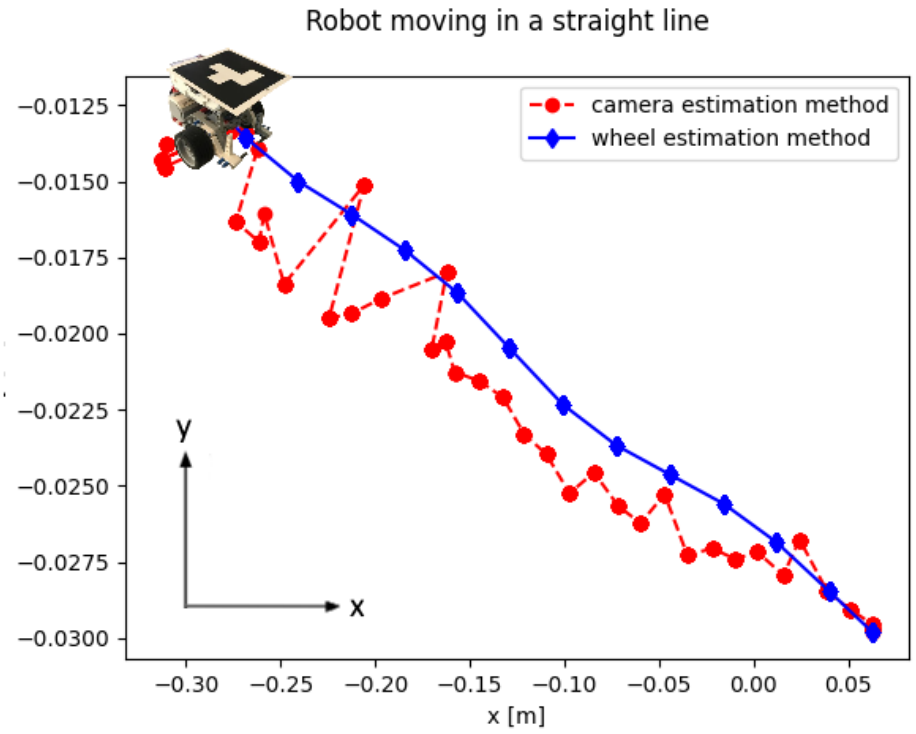


Figure 4.12: This figure illustrates estimation of position for the robot relative to the top camera. The red curve shows the translation vector, `tvec`, calculated from the aruco estimation algorithm. The blue curve shows the estimated tvec based on angular speed of the wheels.

For demonstration purpose, the angle is converted from radians to degrees. Figure 4.13 illustrates a robot rotating clockwise around the rotation axis (z-axis). The curves represents the angle in degrees ranging from [0, 360°]. The red curve was obtained from *camera estimation method*. The blue curve shows the calculated angle based on the *wheel estimation method*. Due to the design choice aimed at minimizing bandwidth utilization, a reduced number of calculated points is present on the blue curve. An important observation reveals that both curves exhibit a step-wise behavior. This design was implemented to mitigate the effects of noise by rounding the angle to the nearest 3-degree increment. This approach was necessary because the naturally observed noise when the robot was stationary was fluctuating between 1-2 degrees.

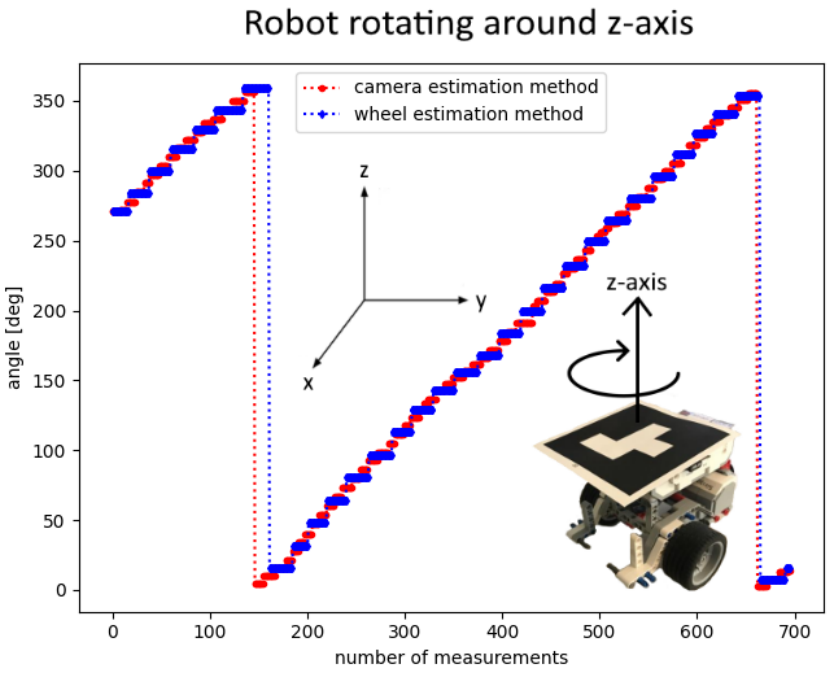


Figure 4.13: This figure illustrates the angle estimation for a robot rotating clockwise around the z-axis. The red curve shows the angle calculated from the ArUco estimation algorithm. The blue curve was calculated by using angular speed on both wheels to estimate the orientation of the robot.

Figure 4.14 illustrates the robot turning 90° to the right. The figure combines both the calculation of position in the top figure and angle in the bottom figure. The right turn is observed in the top figure as the position changes to the right. This right turn is verified by the change in the angle on the bottom figure from 90° to 180°.

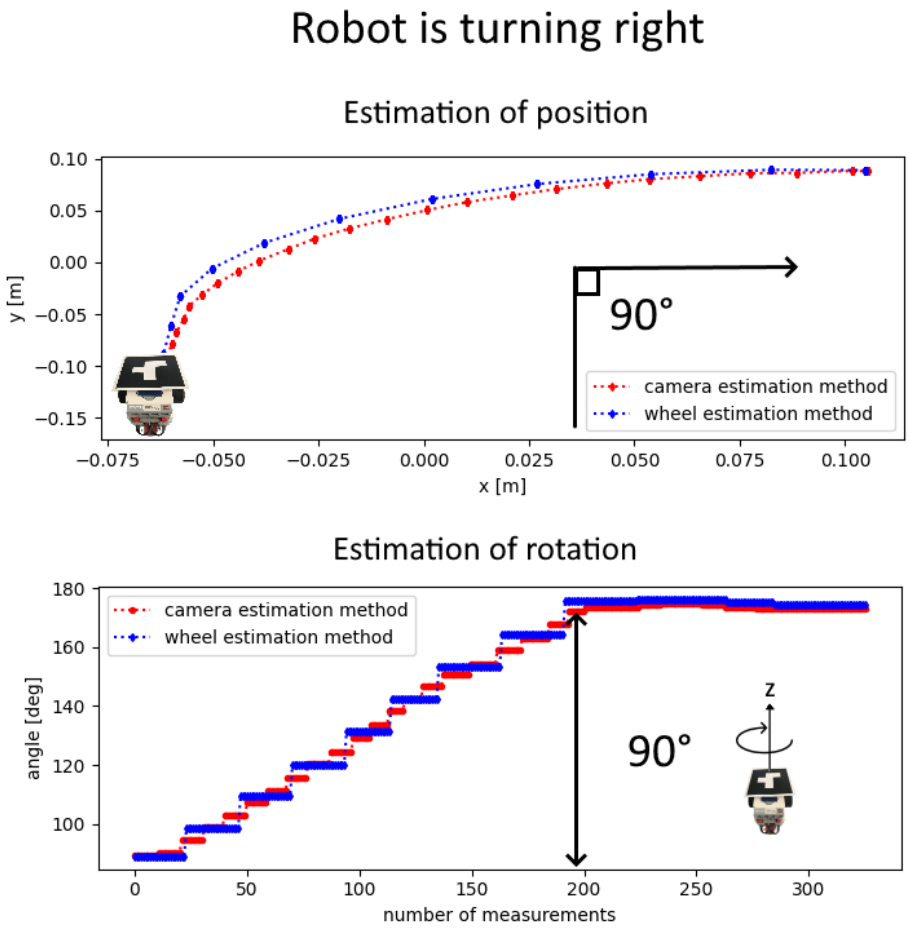


Figure 4.14: The figure illustrates a robot turning 90° to the right. The top figure shows the 2 dimensional position (x,y) of the robot. The bottom figure shows the rotation of the robot. The red curve illustrates the *camera estimation method* using OpenCV. The blue curve shows the *wheel estimation method*.

Despite the close overlap between the red and blue curves, using the *wheel estimation method* exclusively presents certain drawbacks such as;

- drifting (cumulative errors)
- slip (wheels are rotating when not touching the ground)
- increased overhead retrieving data from the robot
- increased use of bandwidth (sending data from robot)
- inaccurate measurement of angular speed and radius

Drifting is a significant concern, particularly due to its cumulative nature that lead to errors adding up over time. Figure 4.15 demonstrates the impact of drifting. As the robot moves, a small deviation in angle impacts calculation of `front_vec`. This results in a gradual increase of error in the position. While these experiments were conducted using each method independently, in the actual application, both methods were deployed to complement each other.

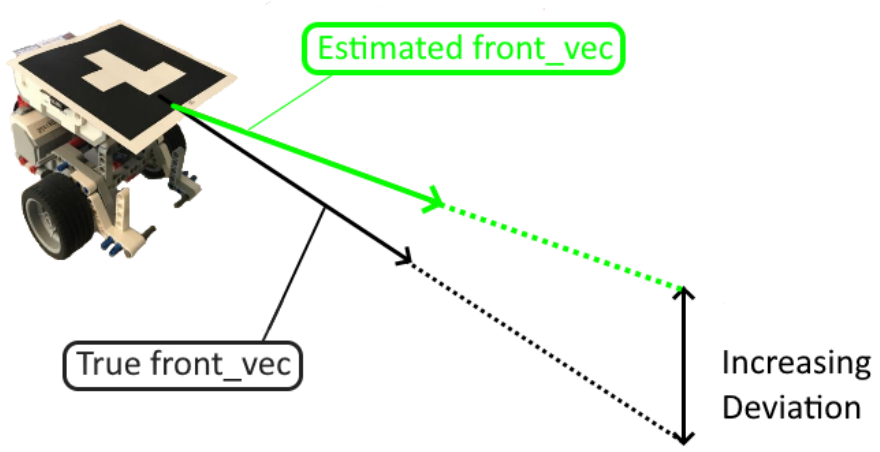


Figure 4.15: This figure illustrates how a small deviation between the unknown true `front_vec` and estimated `front_vec` leads to a gradual increase in the positional error using the *wheel estimation method* exclusively.

Chapter 5

Augmented reality using Three.js

In this chapter, we explore the use of the JavaScript library, *Three.js*, to create an augmented reality (AR) experience. We focus on constructing a virtual scene that are overlaid on a live video stream. Additionally, we examine the synchronization of the virtual camera with pose estimation data.

5.1 Creating virtual scene, camera and 3D models

Three.js is a JavaScript library for 3D computer graphics on the web-browser [25]. Figure 5.1 illustrates a web-browser rendering 3D objects within a virtual scene. In Three.js, the scene acts as a container that holds virtual objects and serves as the fundamental framework for manipulating their properties, including position and orientation.

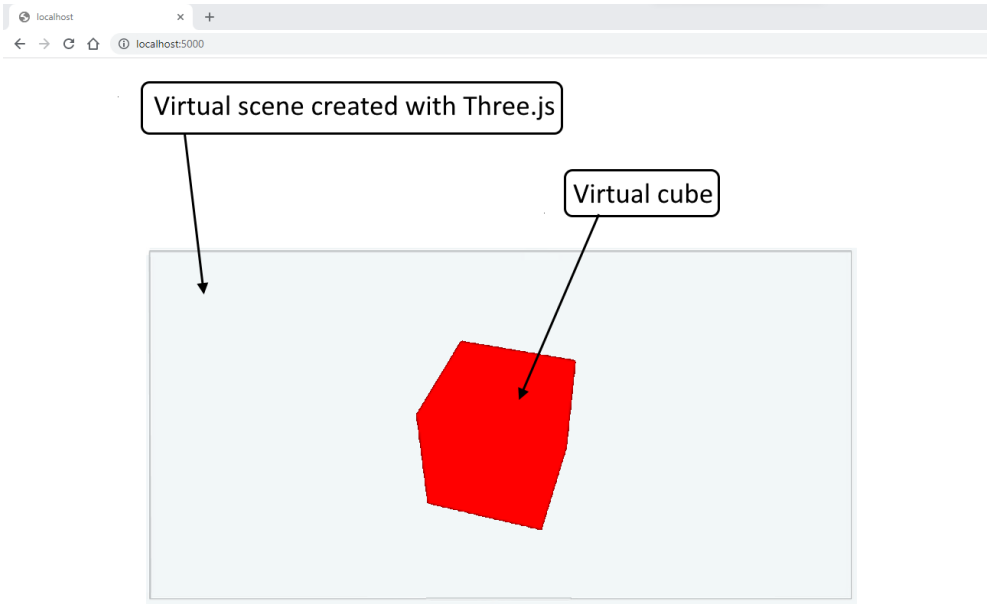


Figure 5.1: This figure illustrates a web-browser rendering virtual objects with the library Three.js. The gray canvas represents the virtual scene where 3D objects are added. Notably, the figure presents a red cube created and placed within the scene.

A scene is created in the JavaScript code 5.1 on line 2. To visualize the objects within the scene, a virtual camera is created, shown in line 5. The camera defines the viewpoint and perspective from which the scene is rendered and observed.

Code 5.1: Code snippet from index.js. Creating virtual scene and camera.

```
1 // create a scene
2 const scene = new THREE.Scene();
3
4 // create camera
5 const camera = new THREE.PerspectiveCamera(fov, aspect, ...
      near, far);
```

5.1 Creating virtual scene, camera and 3D models

The 3D models and animations used in this project were obtained from Mixamo. Mixamo is an online platform from Adobe which offers free premade 3D models and animations [9]. The models were stored in **gltf**-format, which is a suitable 3D-format used within AR applications [10]. Figure 5.2 illustrates a 3D model imported into the virtual scene next to the red cube.

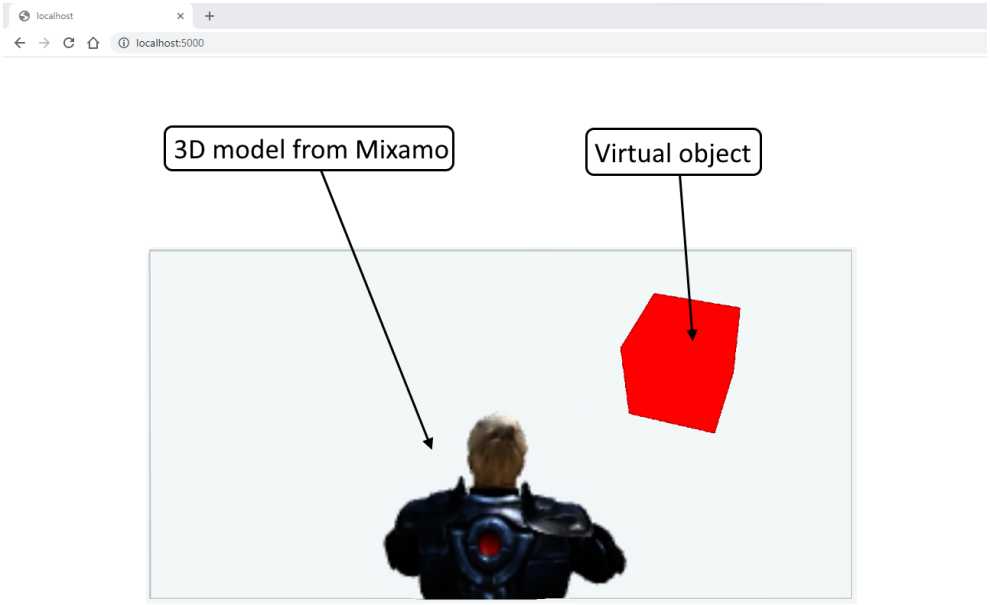


Figure 5.2: This figure illustrates a 3D model from Mixamo imported into the virtual scene. The model contains animations used for various actions performed by the player.

5.2 Combining video stream and the virtual scene

The video-feed is sent from the server as a continuous stream of images explained in section 3.2. The HTML code 5.2 on line 1 shows a HTML image tag used to display images in a video stream to the website. An identifier `id="videoFeed"` is attached to the HTML image tag in order to manipulate the tag in JavaScript.

Code 5.2: The image tag is responsible for displaying the video stream from the cameras in front of the robot.

```
1      <img id="videoFeed" src=""/>
```

The JavaScript code 5.3 on line 2 receives data from the server through a socket connection and updates the `src` attribute of the HTML tag. The `data` variable represents the encoded image retrieved from the video stream captured by the camera on the robot. By updating the `src` attribute of the HTML image tag with new images, the illusion of a video stream is created.

Code 5.3: The `src` attribute of the HTML image tag is updated with data received from the server. This data represents a single image in the video stream captured by the camera on the robot.

```
1 socket.on('video_feed', (data) => {
2     document.getElementById('videoFeed').src = data;
3 });
```

Figure 5.3 illustrates the integration of the virtual scene with images obtained from the video stream. The top left image shows the website with a virtual scene (gray canvas). The top right image shows a picture captured by the camera in front of the robot. The virtual scene is stacked on top of the picture seen in the image below.

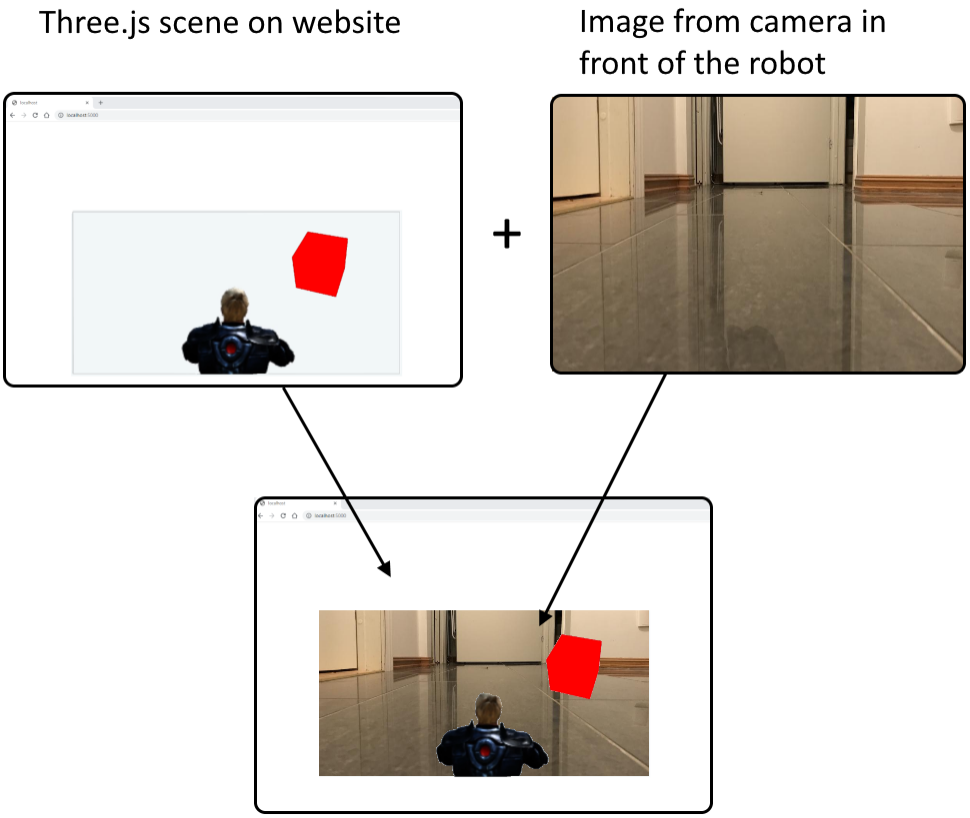


Figure 5.3: This figure illustrates the combination of the virtual scene and images from the video stream captured by the phone camera.

The x and y component of the variable `tvec` (see figure 4.6 in section 4.3) is used to translate the camera's position shown on line 7 in the JavaScript code 5.4. Because the robot only rotates around the z-axis (see figure 4.7 in section 4.3), the `yaw` angle is used to describe the orientation of the robot. Line 8 shows how the camera is rotated based on the `yaw` angle obtained from pose estimation.

Code 5.4: JavaScript code using pose estimation data to manipulate the virtual camera position and the orientation.

```
1 const camera_height = data.camera_height;
2 socket.on('update_pose', function (data) {
3     const pos = data.position;
4     const position = new THREE.Vector3(pos[0], ...
          camera_height, pos[1]);
5     const theta = data.yaw;
6
7     camera.position.copy(position);
8     camera.rotation.set(0, theta, 0);
9 });
```

Figure 5.4 illustrates the result of combining Three.js with pose estimation data. The left image shows one frame from the video stream captured by the top camera. The green arrow, represented by `front_vec`, indicates the direction that the robot's camera and character model is facing. The white coordinate system is added for clarification and represents polar coordinates ranging from 0 to 360° in clockwise rotation. The white dotted arrow from the center of the ArUco marker to the red cube in the left image creates an angle of 0° in the polar coordinate system. The green vector, `front_vec`, in the left image creates an angle of 336°. These two vectors are observed in the right image.

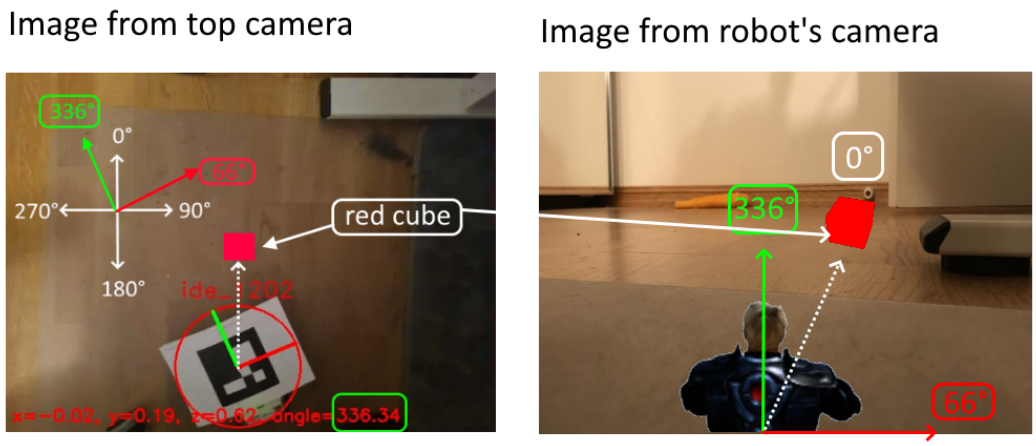


Figure 5.4: This figure illustrates the use of yaw angle from pose estimation to rotate the virtual camera in Three.js. The left image shows a red cube at 0°and the green vector `front_vec` at 336° in polar coordinates. These angles are observed in the right image from the robot's camera.

It is important to note that the `front_vec` (green vector) starts at the position of the robot's camera. If the phone camera is not positioned at the center of the ArUco marker, a linear combination of the unit direction vectors is performed to translate the starting position `tvec` to `camera_tvec`. This is shown in figure 5.5. The red dot illustrates the variable `tvec` which is the position in the center of the ArUco marker with respect to camera coordinates. Because the camera lens on the phone is offset from the center of the ArUco marker (illustrated with the blue dot), a linear combination using the unit direction vectors (`front_vec`, `right_vec` and `up_vec`) is used to translate `tvec` to the new position `camera_tvec`. The variables `a`, `b` and `c` are scalars measured in each unit direction from the center of the ArUco marker to the camera lens.

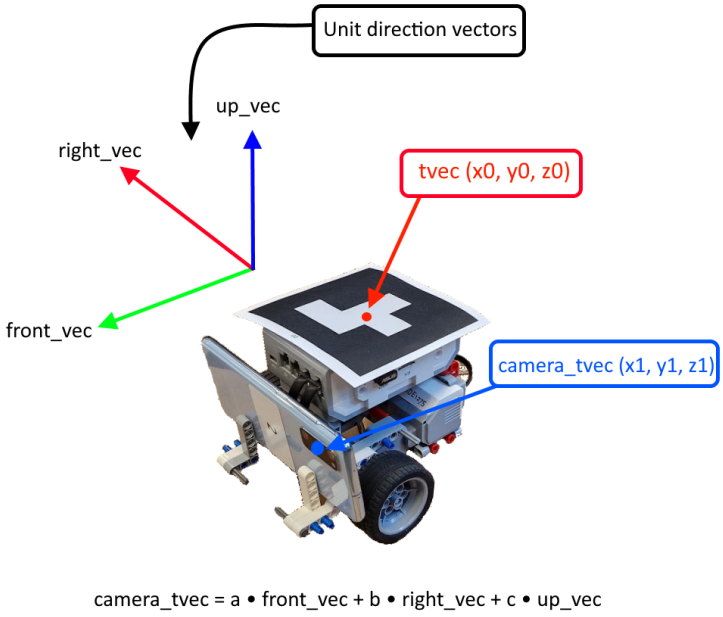


Figure 5.5: This figure illustrates translation of `tvec` to `camera_tvec` which is positioned at the center of the camera lens on the phone. This is achieved by linear combination of each unit vector derived from `rvec`.

Chapter 6

Frontend Server: Serving the web page to the clients

The Frontend server contains the web-site, as well as client-side scripts. This server runs on Node.js [6] and uses the vue.js javascript framework [2] for building user interfaces. This chapter explains the general setup of the Frontend server.

6.1 Overview of the Frontend server

The website is designed to be used as a demonstration for the video-game convention. Users can start a game, play, and when finished, register a nickname to go with their score.

The landing-page of the website is the homepage. From there, they can start the game, which redirects them to the game page. After the game is finished, they are redirected to a register page, where they can register their score. The high-score page is not accessible from any of these web pages, and can only be accessed using a URL. Each page is explained in detail throughout this chapter, and a simple user-flow diagram is shown below in figure 6.1.

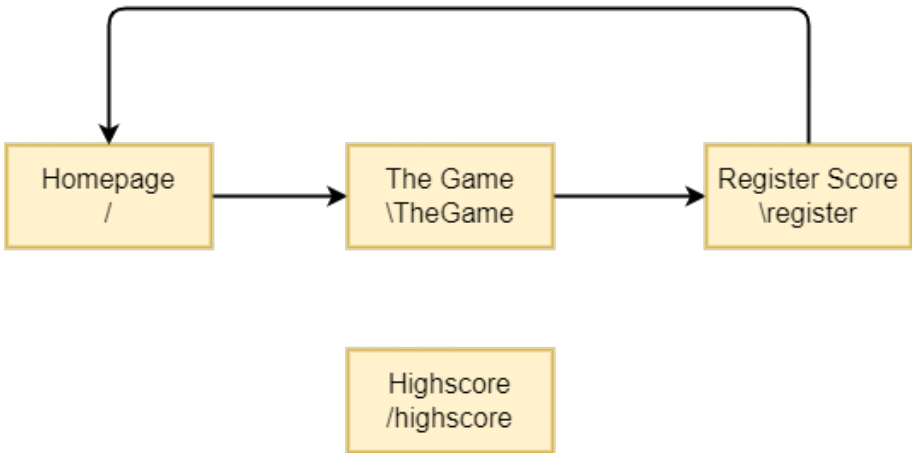


Figure 6.1: This diagram illustrates the general user-flow of the website. The user starts at the homepage, plays the game, and when finished can register their score. The high score page is not directly linked from any site, but can be accessed via a URL.

6.2 Frontend Server Configuration

The server itself runs on Node, which is a JavaScript run-time environment. It has many options for configuration, and includes its own package manager for installing libraries.

6.2.1 When a client connects to the website

When a client connects to the website, it receives the some general code it needs to render the web page. In addition, it will connect to the Backend server to receive the cookie mentioned in 3.3.

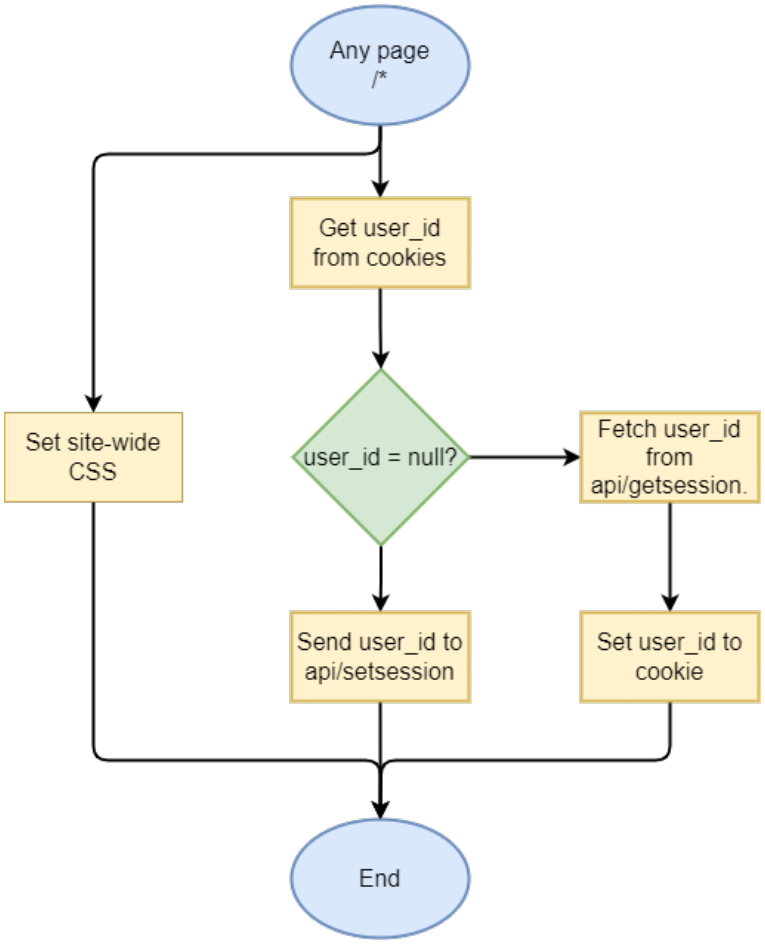


Figure 6.2: The Backend server identifies clients using cookies.

The individual web-pages are written as Vue-Components. The components consists of a template, scripts and a style-sheet. The template is similar to HTML, but can access variables which can be changed dynamically in real-time throughout the lifespan of the page. The script is written in JavaScript.

6.2.2 The Home Page

The homepage is simple and consists of a button to start the game. The home page is illustrated in figure 6.3.

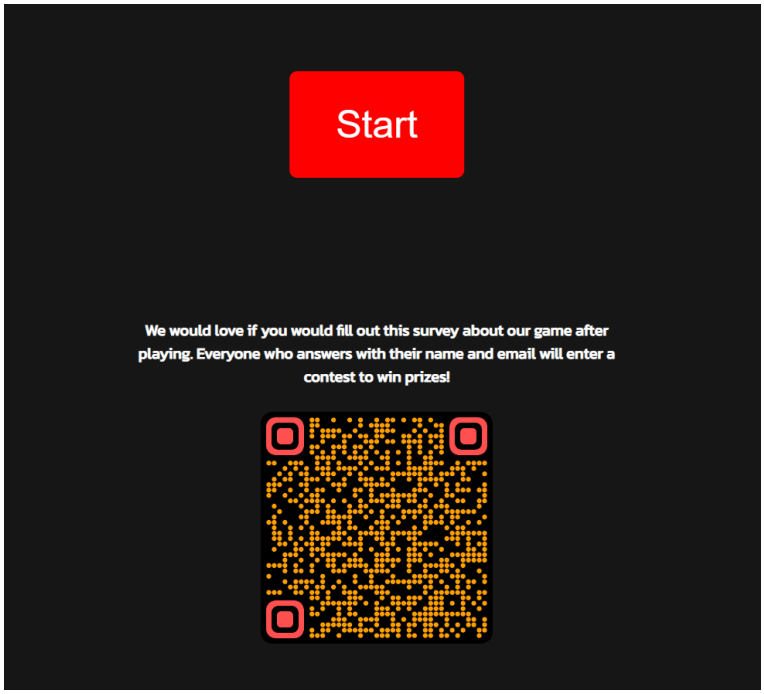


Figure 6.3: The home page is simple with a button to start the game. The QR-code was linked to a survey used during the video-game convention.

The functions of the homepage is illustrated with the flowchart below in figure 6.4.

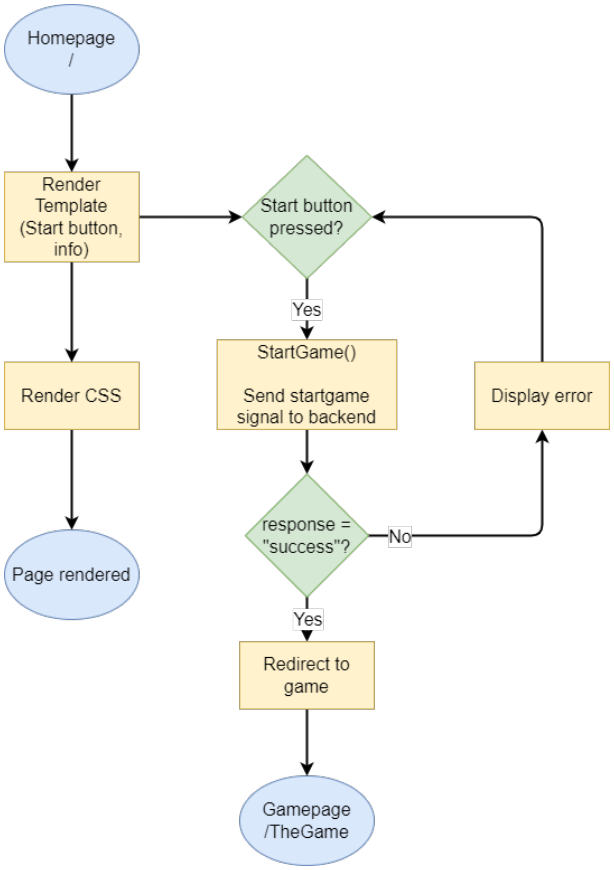


Figure 6.4: The start button runs the StartGame() method which sends a signal to the Backend server to check for available robots, and if it is, it will redirect to the game.

The StartGame() method emits a "StartGame" signal through the socket. If the Backend server successfully assigns the client to a robot, it will respond with "success", and the client will be redirected to the game page. If it does not receive "success", an error will show.

6.2.3 The Game page

The game page is the landing page after the client has pressed the start button. This is where the game is played, and the functions of this page

are what was explained in chapter 5.

When the game is finished, the players are redirected to the register page.

6.2.4 Register Page

The register page is landing page after the game is over. It allows the players to register their score with a nickname to appear on the high-score list.

An example screenshot of the page is shown below in figure 6.5

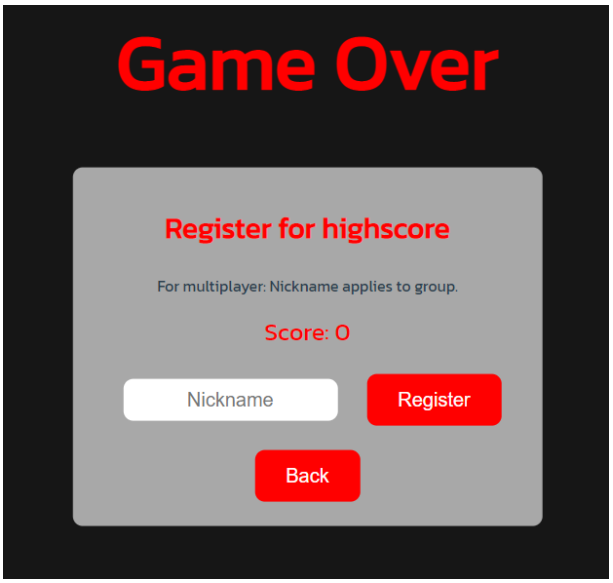


Figure 6.5: The register page informs that the game is over, and contains a form for registering a nickname with their score.

The functions of the register page is shown in the flowchart below in figure 6.6.

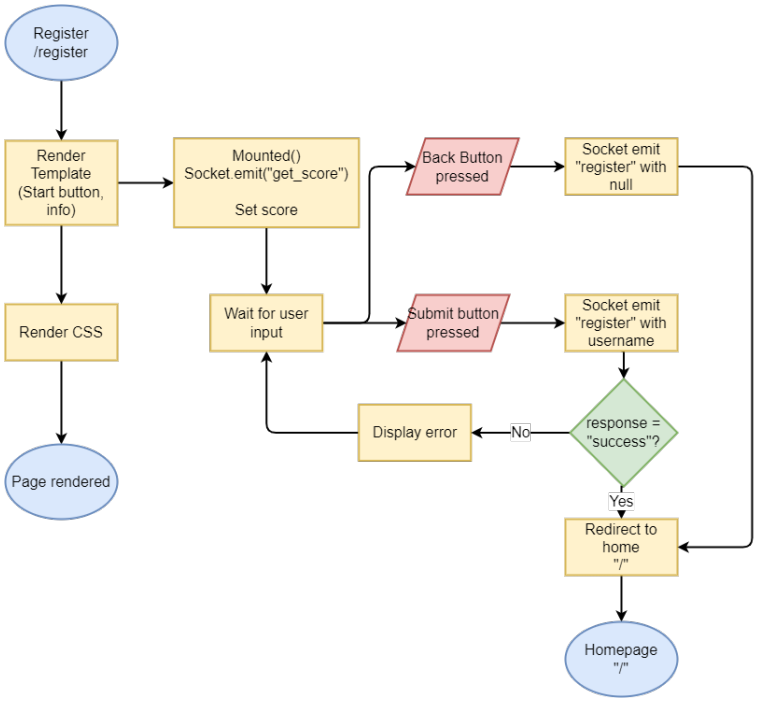


Figure 6.6: After rendering, it request the score from the Backend server. When the Back button is pressed, it signals to the Backend server that you do not want to register. When submit is pressed, it sends the typed nickname to the Backend server. Both buttons redirects to the homepage.

When the page has loaded, it will ask the Backend server for the score of the game through the socket.

```
1  mounted() {
2      socket.emit('get_score', (response) => {
3        this.Score = response
4      })
5    },
```

The register() method checks that the username-field is not empty, and sends the username through a socket to the Backend server. If it received

"success", it will redirect to the home page. If it does not receive "success", an error will show.

6.2.5 The high-score page

The high-score page was displayed publicly during the entirety of the video-game convention. Figure 6.7 shows the high-score page with one list for games using one robot, and one list for games using two robots. The lists were refreshed every second and accessed the data using an API to the Backend server.

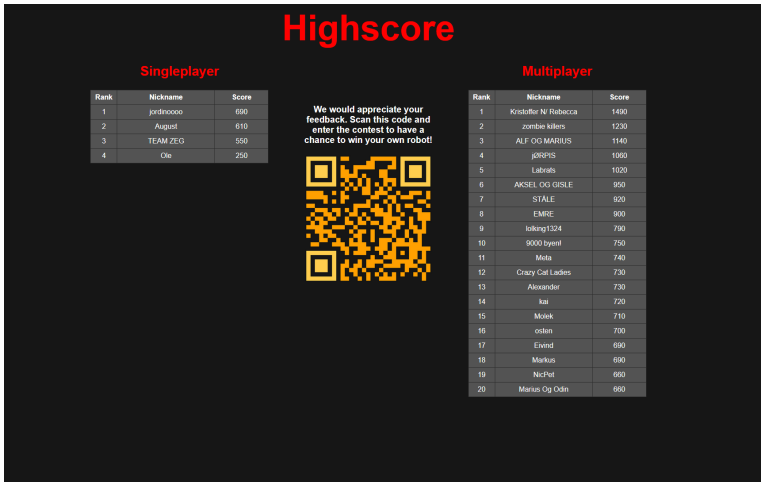


Figure 6.7: The high-score list as it was displayed during the video-game convention.

The high-score data was received as JSON format, and we used the `setInterval()` function to refresh the data every second. The following code snippet shows the set up of one high-score list:

```
1      setInterval(async () => {
2        await ...
             fetch(`${backendURL}/api/gethighscore/singleplayer`)
3          .then((res) => res.json())
4          .then((data) => {
5            this.highscoresingle = data;
6          })
7      }, 1000, false);
```

Chapter 7

Demonstration, User feedback and statistics

During the video-game convention, the game was tested in a real world scenario with people of all ages. We held a survey about the game, as well as collecting some anonymous data about each game. This chapter explains the problems we faced, and the feedback we got through the survey.

7.1 Setup

7.2 Stress-test, problems and solutions

The game was tested the day before the convention, and everything ran smoothly, but when we arrived at the day of the convention, the game was so slow and unstable that it was not playable. Our main suspicion is interference with the WiFi network, as there were thousands of people, and many of the stands had their own WiFi networks. The robots were especially prone to this, as they used an older WiFi-protocol on the 2.4GHz band.

We counteracted this by moving the router as close as possible to the robots themselves, and while this didn't eliminate the issue, it alone made a huge difference in latency.

During the three days of the convention, there were still periodic issues with latency and disconnection. The robots would disconnect from the server frequently seemingly without reason, which we also suspect was due to interference.

7.3 User feedback

The survey could be answered by everyone, even if they didn't play themselves. During the entirety of the convention, 79 people answered the survey. We first asked if they had tried the demo.

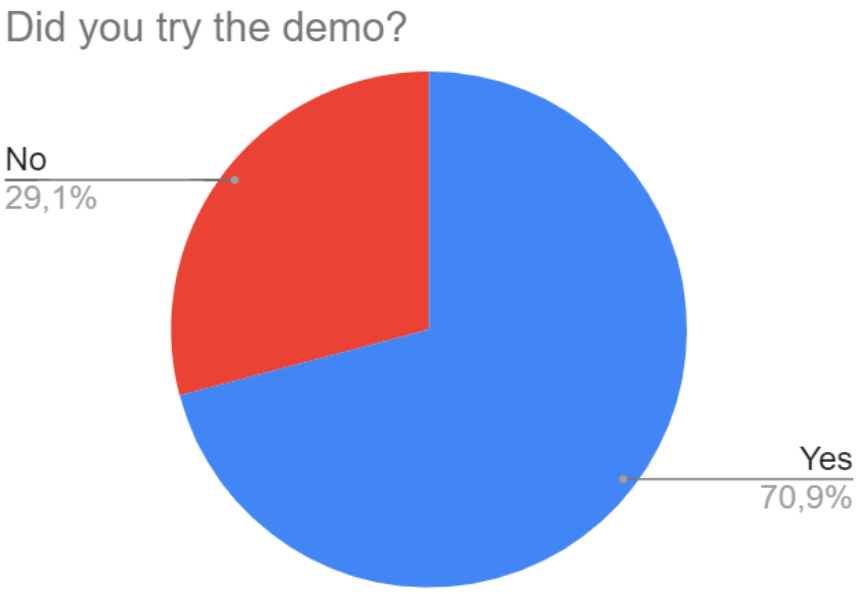


Figure 7.1: Out of 79 participants, 71% played the game and 29% did not.

7.3.1 Survey answers

The remaining questions was about the game itself. They could answer "Very good", "Good", "Ok", "A little bad", and "Very Bad" on each question, and also write a comment to each of these questions.

The answers to the question "What do you think about the concept of the project?" is shown in the diagram below

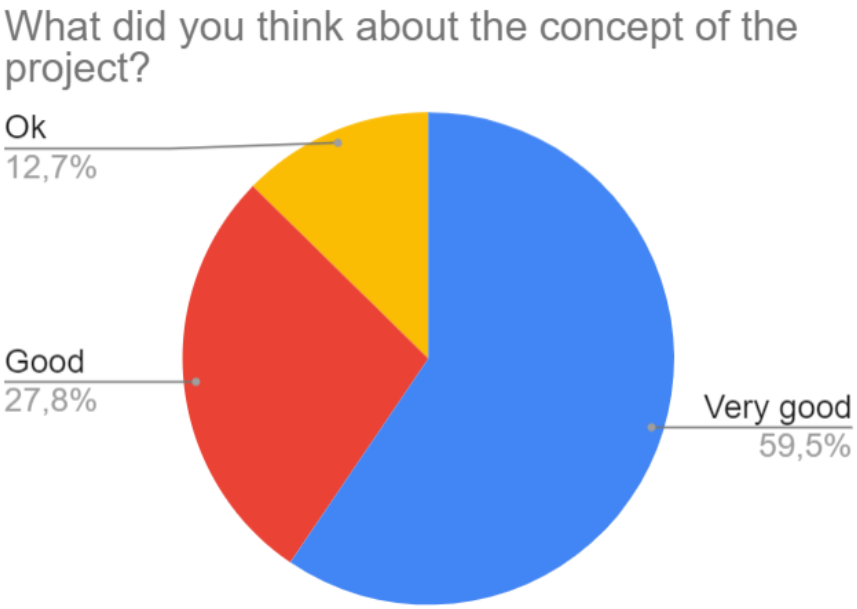


Figure 7.2: There was a very clear positive trend on how people felt about the concept of our project. 87% answered positively.

People commented that they liked the fact that they could see the physical robots in action while playing, and that it had much potential for performing tasks remotely. People also found the project interesting and educational.

Most of those who answered "Ok" did not leave a comment, but one person commented that the robots could have been more involved in other ways than just driving around.

People were also very positive to the concept of the game itself. The answers to the question "What do you think about the concept of the game?" is shown in the diagram below

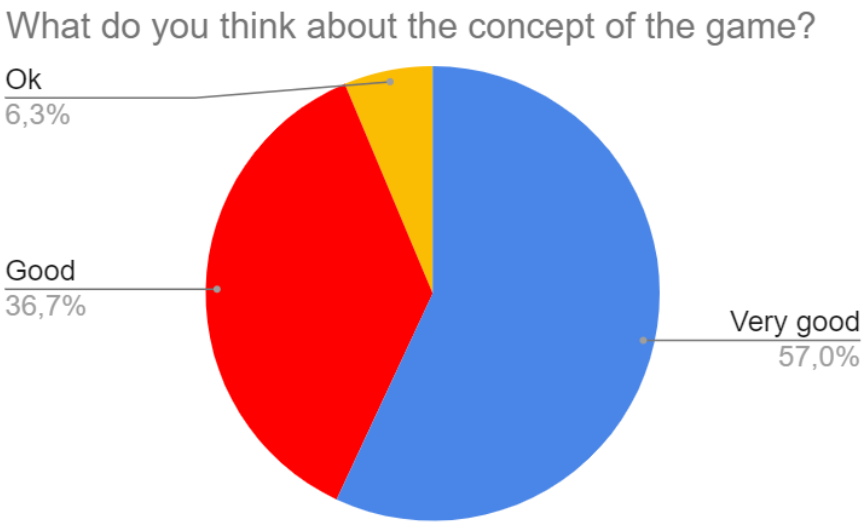


Figure 7.3: There was a very clear positive trend on how people felt about the concept of our game.

People in general liked the idea, but some answered that it got a bit boring after a while. Some people wanted more design and obstacles within the game.

When asked "What do you think about the execution of the game?", the answers shifted more towards "good", rather than "very good", with a still very positive trend.

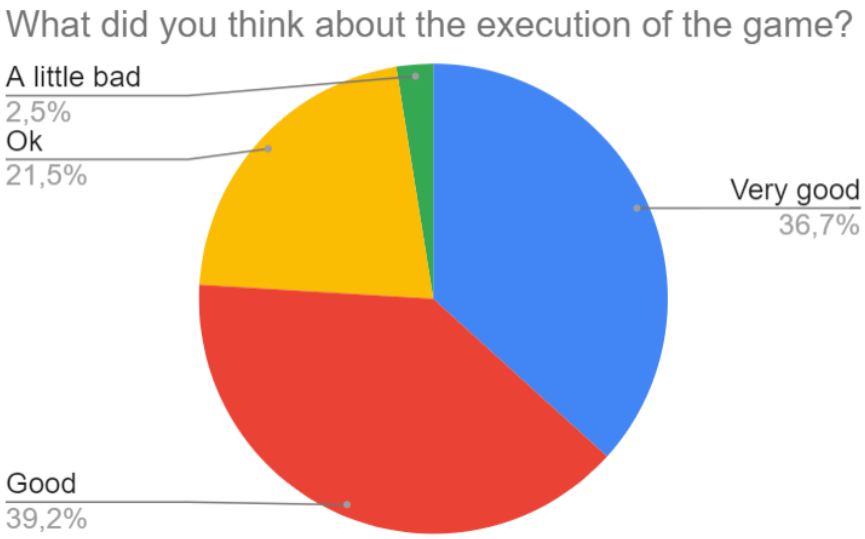


Figure 7.4: There was a very clear positive trend on how people felt about the execution, but it was more shifted towards "good", and a small bit answered negatively.

Most of the critical comments was related to latency and controlling. Unfortunately, we could not fix the issue with latency during the convention, and most of the controller issues was due to latency. However, we made some adjustments on the controllers, with some alternative keys to control the robot, which seemed to make it easier for some.

Some comments also mentioned that zombies could spawn right where the player was located, making it impossible not to lose life-points.

7.4 Statistics

During the convention, we gathered some anonymous data about each game. Unfortunately, we did not gather the data during the first four hours of the convention due to a mistake in the code. The data does not include games where the game did not advance beyond the first round to account for our own testing.

Statistics for the whole convention:

- **Total number of games played:** 80
- **Average number of rounds reached:** 7.7
- **Highest round reached:** 15
- **Average score:** 490
- **Highest score:** 1490

Chapter 8

Conclusion

In this thesis, we aimed to explore the development of an augmented reality (AR) multiplayer game that involved physical robots interacting with the virtual environment. In this study we presented the physical devices and their connections from a topological view. To structure the application, we divided the code into sections, namely the back-end and front-end components. The back-end component of the application is responsible for image processing, maintaining game states, establishing connections with physical devices and storage of data in a database. The front-end component serves web pages and formats the data from the back-end component in a presentable way to the players. In context of image processing, ArUco markers were used to estimate the position and orientation of the robots. Pose estimation data was used to synchronize the the video stream from the robots camera with the virtual scene. As a result, players were able to interact with the virtual environment through the robots, and play the game as intended.

Bibliography

[1] Travis CI. Eventlet. URL: https://eventlet.net/.

[2] Collaboration. Vue.js. URL: https://vuejs.org/.

[3] Jian S Dai. Euler–rodrigues formula variations, quaternion conjugation and intrinsic connections. *Mechanism and Machine Theory*, 92:144–152, 2015. URL: https://www.sciencedirect.com/science/article/pii/S0094114X15000415, doi:10.1016/j.mechmachtheory.2015.03.004.

[4] Frank Dellaert and Seth Hutchinson. Motion model for the differential drive robot. URL: https://www.roboticsbook.org/S52_diffdrive_actions.html.

[5] dfdsf sdfdsf dfsdfs. Ev3dev. https://www.ev3dev.org/.

[6] OpenJS Foundation. Node.js. URL: https://nodejs.org/en.

[7] Miguel Grinberg. Flask-socketio. URL: https://flask-socketio.readthedocs.io/en/latest/.

[8] The Lego Group. Lego mindstorms ev3 building instructions. URL: https://education.lego.com/en-us/product-resources/mindstorms-ev3/downloads/building-instructions.

[9] Adobe Systems Incorporated. Three.js github repository. https://www.mixamo.com/#/, Accessed 2023.

[10] James. gltf, glb, and usdz. URL: https://help.sketchfab.com/hc/en-us/articles/360046421631-glTF-GLB-and-USDZ.

[11] Oleg Kalachev. *Online ArUco markers generator*. URL: https://github.com/okalachev/arucogen.

[12] W. Khalil, E. Dombre, and E. Dombre. *Modeling, Identification and Control of Robots*. Elsevier Science & Technology, Oxford, UK, 2004. URL: https://ebookcentral-proquest-com.ezproxy.uis.no/lib/uisbib/detail.action?docID=413853.

[13] Pavel Khlebovich. Ip webcam android app. https://play.google.com/store/apps/details?id=com.pas.webcam.

[14] Chris Kolmar. 25+ amazing virtual reality statistics [2023]: The future of vr + ar. *Zippia*, 2023. URL: https://www.zippia.com/advice/virtual-reality-statistics/.

[15] Valk Lauren and David Lechner. Pybricks. https://pybricks.com/.

[16] David Lechner. Usb wi fi dongles. URL: https://github.com/ev3dev/ev3dev/wiki/USB-Wi-Fi-Dongles.

[17] Satya Mallick. Rotation matrix to euler angles. 2016. URL: https://learnopencv.com/rotation-matrix-to-euler-angles/.

[18] mdn web docs. Using http cookies. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies.

[19] mdn web docs. The websocket api (websockets). 2023. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API.

[20] OpenCV. *Camera Calibration*. URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html.

[21] OpenCV. *Detection of ArUco Markers*. URL: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html.

[22] The Pallets Projects. Flask. URL: https://flask.palletsprojects.com/en/2.3.x/#.

[23] Dmitry Slinkin. URL: https://github.com/ev3dev/ev3dev/issues/1442#issuecomment-757471893.

[24] Fernando Souza. What is a fiducial marker. URL: https://medium.com/vacatronics/what-is-a-fiducial-marker-865799bc7266.

[25] three.js authors. Three.js github repository. https://github.com/mrdoob/three.js, Accessed 2023.

Attachments A

Guides

A.1 Changing EV3 Names

To change the EV3 names, we ran the command: `sudo ev3dev-config`

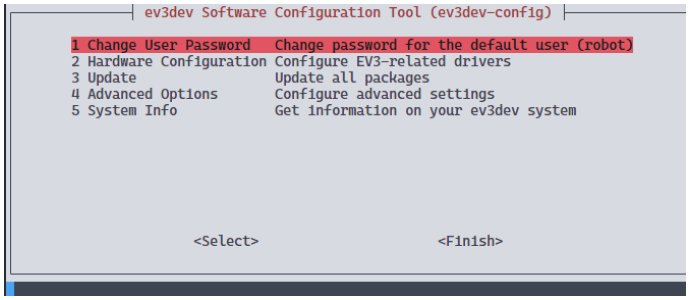


Figure A.1: The command opens a configuration menu. Changing hostname is in "4 Advanced options"

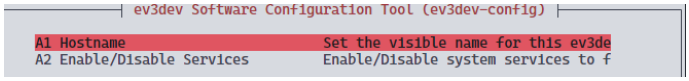


Figure A.2: Choose hostname. Some name-rules will show up, click ok.

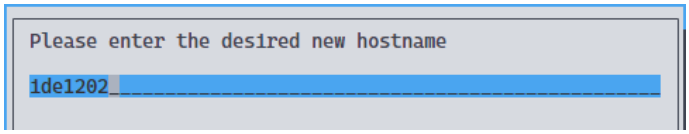


Figure A.3: Write new name.

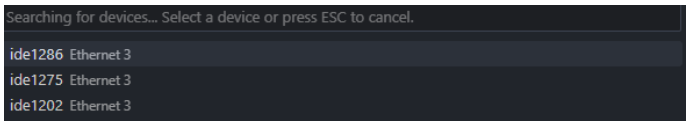


Figure A.4: When working on a robot when multiple are connected, choosing the right one is important. Changing the name of the robots makes it much easier to know which robot to choose.

Attachments B

Additional Tools

We made some tools to streamline manual tasks that took much time, especially during setup of devices. While not really necessary, they saved us much time in the long run.

B.1 Autocalibration

B.1.1 Problem

When we needed to change either the camera, or try a new resolution for a camera, it had to be re-calibrated for the pose-estimation to be accurate. This required us to grab multiple photos and manually set up each photo for calibration. This was time-consuming and inefficient. What we wanted was a program that could allow us to:

- Easily grab multiple photos
- Save and order them by phone and resolution
- Use an ID and an IP address to identify and connect to phones
- Automatically name them

- Automatically calibrate using the grabbed photos

B.1.2 Solution

We used Tkinter, a python interface package, as our **gui**.

When the program starts, the user is prompted with a window for choosing which phone to calibrate.

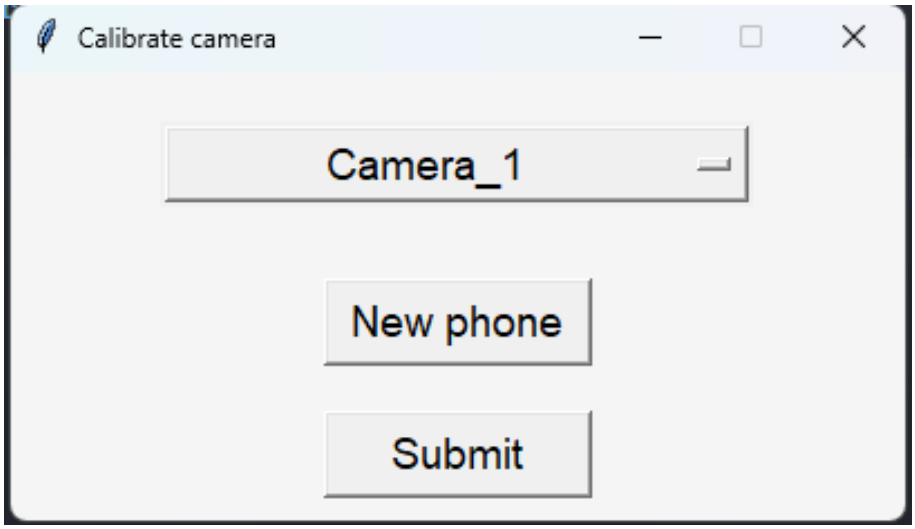


Figure B.1: You can choose an already existing phone, or register a new phone.

The folders for the calibrations are structured as following:

```
Calibrations/
└── Camera_1/
    ├── 1920x1080/
    │   ├── 0.jpg
    │   ├── 1.jpg
    │   ├── ...
    │   ├── calibration_matrix.npy
    │   └── distortion_coefficients.npy
    ├── 3840x2160/
    └── .../
```

```
├── Camera_2/
├── .../
├── autocalibration.py
└── device_info.txt
```

device_info.txt

The device_info.txt file contains all registered devices. It follows a simple convention. ID <space> IP.

An example follows:

```
1 192.168.0.101
2 192.168.0.102
3 192.168.0.103
4 192.168.0.104
5 192.168.0.105
```

The ID is for identifying the physical device, as we have given each of our phones a unique ID. The IP is the ip-address registered with the phone.

If the IP should change, you can either calibrate again by registering a new device with the same ID, or you can manually change the device_info.txt file.

autocalibration.py

Code for choosing phone window:

```
1 ...
2 # Function for choosing phone
3 def create_choose_ui():
4     ...
```

```
    # Finds all cameras currently calibrated and adds them to list
    options = [i for i in os.listdir(<calibration_dir>) if os.path.isdir(os.path.join(<calibration_dir>, i))]

    ...
    # Create option menu
    option_menu = tk.OptionMenu(window, selected_option, *options)
    ...

    # Submit button if pressed gets the selected option and passes them to the function save_images()
    # Since the options will be destroyed when the window gets destroyed, we must pass the window to the function and destroy it there.
    submit_button = tk.Button(window, text="Submit", command=lambda: [save_images(selected_option.get()[7:], window)])

    ...

    # New button if pressed destroys current window and runs the function for registering new phone create_new_ui()
    new_button = tk.Button(window, text="New phone", command=lambda: [window.destroy(), create_new_ui()])
```

Code for registering phone window:

```
def create_new_ui():
    ...
```

```
    # Get user inputs for ID and IP
    id_label = tk.Label(window, text="ID: ")
    id_input = tk.Entry(window)
    ...
    ip_label = tk.Label(window, text="IP: ")
    ip_input = tk.Entry(window)
    ...

    # submit button runs the function ...
       save_images() and passes the id, ip and ...
       the window to it.
    submit_button = tk.Button(window, ...
       text="Submit", command=lambda: ...
       [save_images(id_input.get(), window, ...
       ip_input.get())])
```

Code for showing and capturing images:

```
def save_images(id, old_window, ip=None):

    # Finds IP if phone is registered.
    if ip is None:
        with open(os.path.join(calibration_dir, ...
           'device_info.txt'), "r") as f:
            for line in f:
                line = line.split(' ')
                if id == line[0].strip():
                    ip = line[1].strip()
                    break

    # Use cv2 for displaying images and store ...
       the resolution
    cap = ...
       cv2.VideoCapture(f"http://{ip}:8080/video")
    res = ...
       f"{int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))}x{int(cap.get(cv2.CAP_P

    def video_stream():
```

```
17          _, frame = cap.read()
18          frame = cv2.resize(frame, (800, 500), 0, 0)
19          cv2image = cv2.cvtColor(frame, ...
                cv2.COLOR_BGR2RGBA)
20          img = Image.fromarray(cv2image)
21          imgtk = ImageTk.PhotoImage(image=img)
22          lmain.imgtk = imgtk
23          lmain.configure(image=imgtk)
24          lmain.after(1, video_stream)
25
26      video_stream()
27
28      # Create take button
29      take_button = tk.Button(window, text="Take ...
            photo", command=lambda: [save_image(id, ...
            ip, res)])
30
31      # Create done button
32      done_button = tk.Button(window, text="Done", ...
            command=lambda: ...
            [calibrate(os.path.join(calibration_dir, ...
            f"Camera_{id}", res)), window.destroy()])
33
34      def on_enter(event):
35          save_image(id, ip, res)
36
37      window.bind('<Return>', on_enter)
38
39      # Create new directory for ID if ID does not ...
            exist
40      id_path = os.path.join(calibration_dir, ...
            f"Camera_{id}")
41
42      if os.path.exists(id_path):
43          with open(os.path.join(calibration_dir, ...
                'device_info.txt'), "r") as info:
44              found = False
45              for line in info:
46                  if line == f"{id} {ip}\n":
```

```
47                          found = True
48                  if not found:
49                      if messagebox.askyesno("Alert", ...
                           f"Phone with ID {id} is ...
                           configured with IP ...
                           {line.split(' ')[1]}!\nAre ...
                           you sure you want to change ...
                           to IP {ip}?\nThis will erase ...
                           all calibrations for this ...
                           ID.\nYou can change IP ...
                           manually in the file named ...
                           \"device_info.txt\" if you ...
                           want to change IP but keep ...
                           calibrations." ):
50                          for file in ...
                               os.listdir(id_path):
51                              file_path = ...
                                   os.path.join(id_path, ...
                                   file
52                                                      )
53                              try:
54                                  os.unlink(file_path)
55                              except Exception as e:
56                                  print('Failed to ...
                                       delete %s. ...
                                       Reason: %s' % ...
                                       (file_path, e))
57          else:
58              os.makedirs(id_path)
59              with open(os.path.join(calibration_dir, ...
                   'device_info.txt'), "a") as info:
60                  info.write(f"{id} {ip}\n")
61
62
63          # Sort for easy modification
64          with open(os.path.join(calibration_dir, ...
                 'device_info.txt'), 'r+') as f:
65              sorted_contents =  ...
                   ''.join(sorted(f.readlines(), key = ...
```

```
            lambda x: int(x.split(' ')[0])))
66          f.seek(0)
67          f.truncate()
68          f.write(sorted_contents)
69
70      # Create new directory for resolution
71      res_path = os.path.join(id_path, f"{res}")
72      if not os.path.exists(res_path):
73          os.makedirs(res_path)
74      elif messagebox.askyesno("Alert", "A phone ...
           with this ID and Resolution already ...
           exists!\nAre you sure you want to ...
           calibrate again?"):
75          for file in os.listdir(res_path):
76              file_path = os.path.join(res_path, ...
                   file)
77              try:
78                  if os.path.isfile(file_path) or ...
                       os.path.islink(file_path):
79                      os.unlink(file_path)
80              except Exception as e:
81                  print('Failed to delete %s. ...
                       Reason: %s' % (file_path, e))
```

Code for saving image:

```
1  def save_image(id, ip, res):
2      # image number
3      i = ...
           len(os.listdir(os.path.join(calibration_dir, ...
           f"Camera_{id}/{res}")))
4
5      # Retrieve image from phone and save to ...
           directory
6      ul.urlretrieve(f"http://{ip}:8080/photo.jpg", ...
           os.path.join(calibration_dir, ...
           f"Camera_{id}/{res}/{i}.jpg")
```

B.2 Arucogenerator

B.2.1 Problem

We used a great deal of aruco-markers. Different sizes, setups and paper-types made the process of testing both time-consuming and expensive, as we had to print multiple sheets of paper, and some paper we used was of higher qualtity. There are not many aruco-generators available online, at least from what we could see. The one we found was very limited, and we had to use a great bit of time setting up our markers for printing. In addition to this, it could not generate 3x3 aruco-markers, which is what we wanted to use.

What we wanted was a program that could allow us to:

- Generate every aruco-marker of our choosing
- Save them individually and as a ready-to-print pdf
- Give us full control over sizes and number of markers, as well as the margin around the markers
- The pdf should be as efficiently packed with markers as possible, for an A4 size paper, so we can print multiple markers on one sheet

B.2.2 Solution

We created a python script for creating any number of aruco-markers we wanted to.

The folder-structure for the aruco-generator is as following (with example data):

```
ArucoGenerator/
└── arucomarkers/
    ├── 3x3_32_00.png
    ├── 3x3_32_01.png
    └── ...
```

```
    aruco.pdf
generate_aruco.py
```

Libraries used:

- **cv2** for aruco generation
- **numpy** for image matrix
- **os** for paths
- **reportlab** for pdf generation

Code for setup:

```
    marker_id = None # None if all markers up to ...
       "num_of_markers" should be created
    marker_size = 8 # Markersize in cm
    marker_dim = 3 # Dimentions of marker
    total_markers_max = 32 # Total number of ...
       markers for the set of markers
    num_of_markers = 10 # Number of markers needed
    margin = 0 # Margin around markers in cm
```

Code for generating aruco markers:

```
def generate_aruco_markers(marker_id, ...
   marker_size, marker_dim, total_markers):
    try:
        # Get dictionary for marker dimentions
        key = getattr(cv2.aruco, ...
           f'DICT_{marker_dim}X{marker_dim}_{total_markers}')
        arucoDict = ...
           cv2.aruco.getPredefinedDictionary(key)
    except AttributeError:
        # Define our own dictionary if not ...
           present in predefined dictionary
```

```
8           arucoDict = ...
                cv2.aruco.extendDictionary(total_markers, ...
                marker_dim)
9
10      # Generate the marker image with the ...
            included generator of the cv2 library
11      marker_img = np.zeros((marker_size, ...
            marker_size), dtype=np.uint8)
12      marker_img = ...
            cv2.aruco.generateImageMarker(arucoDict, ...
            marker_id, marker_size, marker_img, ...
            borderBits=1)
13
14      return marker_img
```

Code for creating pdf:

```
1  c = ...
       canvas.Canvas(os.path.join(os.path.dirname(os.path.abspath(__file__))
       "arucomarkers", "aruco.pdf"), pagesize=A4)
2
3       one_cm = A4[0] / 21 # 1 cm in units used by ...
            reportlab
4
5       marker_size = marker_size * one_cm # Convert ...
            marker_size to units used by reportlab
6       margin = margin * one_cm # Convert margin to ...
            units used by reportlab
7       size = marker_size + margin * 2 # Add ...
            margins to marker_size for total size
8
9       num_x = int(A4[0] // size) # Number of ...
            markers in x direction
10      num_y = int(A4[1] // size) # Number of ...
            markers in y direction
11
12
13      print(num_x, num_y)
```

```
14
15      if num_y == 0 or num_x == 0:
16          raise ValueError ("Size is too big, ...
               total size must be under 21 ...
               (marker_size + 2*margin)")
17
18
19      # Draw all markers on separate pages in the pdf
20      for image in ...
           os.listdir(os.path.join(os.path.dirname(os.path.abspath(__file__)),
           "arucomarkers")):
21          c.setStrokeGray(0.7)
22          c.setDash([4, 4])
23          c.grid(range(0, int(A4[0]), int(size)), ...
               range(0, int(A4[1]), int(size)))
24          # Draw each marker x*y times
25          for x in range(num_x):
26              for y in range(num_y):
27                  x_pos = x * size + margin
28                  x_label_pos = x_pos + marker_size/2
29                  y_pos = y * size + margin
30                  y_label_pos = y_pos - margin + ...
                       marker_size + 1.2*one_cm
31                  # Labeling the markers with ...
                       their id
32                  c.drawCentredString(x_label_pos, ...
                       y_label_pos, ...
                       image[:-4].split("_")[-1])
33                  # Draw the marker
34                  c.drawImage(os.path.join(os.path.dirname(os.path.abspath
                       "arucomarkers", image), ...
                       x_pos, y_pos, marker_size, ...
                       marker_size)
35          # Move to the next page
36          c.showPage()
37
38      c.save()
```

To calculate placement of the markers, we derived a formula:

$$\text{image\_x, image\_y} = \text{Image index for x and y} \tag{B.1}$$

$$\text{X} = image\_x * image\_width + margin \tag{B.2}$$

$$\text{Y} = image\_y * image\_height + margin \tag{B.3}$$

$$\text{Label X} = X + \frac{image\_width}{2} \tag{B.4}$$

$$\text{Label Y} = Y + \frac{image\_width}{2} \tag{B.5}$$

This will place as many aruco-markers (with margins) as can fit within an A4 paper

Attachments C

Programlisting

The entire source code is attatched to this PDF:

Attachments D

Code listings

D.1 Converting `rvec` to Euler angles

Code D.1: Code for converting rotation matrix to Euler angles[17].

```
# Checks if a matrix is a valid rotation matrix.
def isRotationMatrix(R):
    Rt = np.transpose(R)
    shouldBeIdentity = np.dot(Rt, R)
    I = np.identity(3, dtype = R.dtype)
    n = np.linalg.norm(I - shouldBeIdentity)
    return n < 1e-6

# Calculates rotation matrix to euler angles
# The result is the same as MATLAB except the order
# of the euler angles (x and z are swapped ).
def rotationMatrixToEulerAngles(R):
    assert(isRotationMatrix(R))
    sy = np.sqrt(R[0,0] * R[0,0] +  R[1,0] * ...
       R[1,0])
    singular = sy < 1e-6
    if  not singular :
        x = np.arctan2(R[2,1] , R[2,2])
```

```
18          y = np.arctan2(-R[2,0], sy)
19          z = np.arctan2(R[1,0], R[0,0])
20      else :
21          x = np.arctan2(-R[1,2], R[1,1])
22          y = np.arctan2(-R[2,0], sy)
23          z = 0
24      return np.array([x, y, z])
25
26  # Convert rotation matrix to euler angles
27  rotation_matrix = cv2.Rodrigues(rvec)[0]
28  euler_angles = ...
        rotationMatrixToEulerAngles(rotation_matrix)
29  theta = euler_angles[2]
```