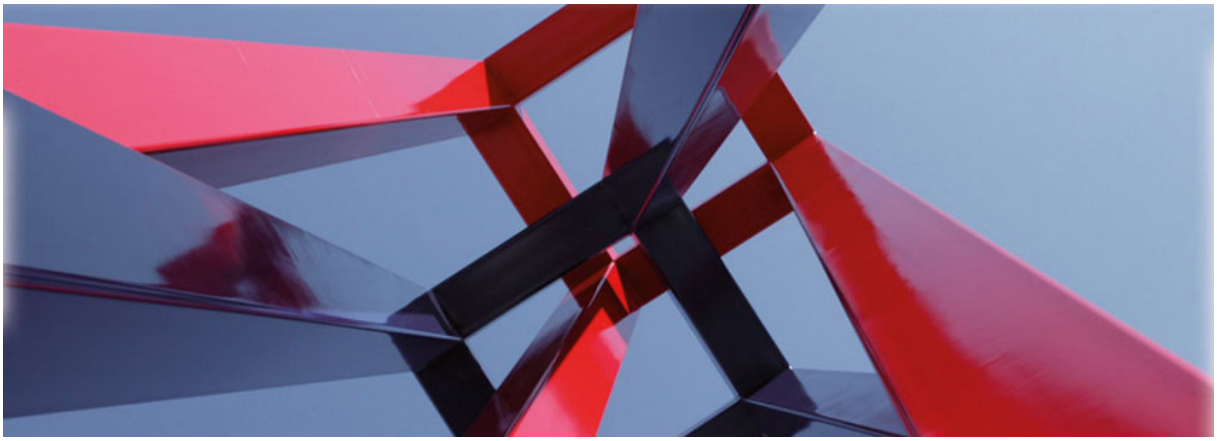
Hochschule Hof
University of Applied Sciences

Applied Robotics – Digitale Methoden WS2023



As we move deeper into the 21st century, the roles of artificial intelligence (AI) and robotics in shaping our world are becoming undeniably critical. These technologies stand at the cusp of transforming societal structures, enhancing life quality, and catalyzing human advancement in ways we've never seen before. It's essential to grasp the profound impact AI and robotics will have on our collective future.
n terms of economic development and employment, AI and robotics are redefining industry norms through the automation of monotonous tasks, boosting of productivity levels, and minimization of costs. Despite worries about the potential loss of jobs, these innovations offer a silver lining by generating new roles in nascent fields like AI engineering, data analysis, and the development of interactive technologies between humans and machines.
These are only a few examples. Therefore, the need for experts is constantly growing.
In the course 'applied robotics' students implemented various projects, where they used state of the art methods from ai and robotics.

Christian Groth (publisher)
University of applied sciences Hof
2023
DOI 10.57944/1051-173

Table of contents

Smart Firetruck

Reiner Katharina
University of Applied Sciences Hof
Hof, Germany
katharina.reiner@hof-university.de

Wilfert Luca
University of Applied Sciences Hof
Hof, Germany
luca.wilfert@hof-university.de

Raithel Tim
University of Applied Sciences Hof
Hof, Germany
tim.raithel@hof-university.de

Schlitter Moritz
University of Applied Sciences Hof
Hof, Germany
moritz.schlitter@hof-university.de

Manig Ralf
University of Applied Sciences Hof
Hof, Germany
ralf.manig@hof-university.de

***Abstract*—As part of the lecture "Applied Robotics" at the university of applied sciences Hof, an intelligent firefighting robot is built. The robot is supposed to drive autonomously to the destination, where it should recognize a picture of a flame, then put out that flame with water and lastly return to the fire station. This functionality is implemented based on the chassis of an RC car with a single board computer that uses the input of a camera to navigate. The car contains a water tank with a pump to spray the water on the flame picture. Additionally, the vehicle has working blue light and can play a siren sound.**

Keywords— intelligent, robot, extinguish, firefighters

I. STATE OF THE ART

There were already multiple projects done at the university of applied science Hof, in which the same type of electric vehicle was used. Reference [1], the practical work „Aufbau und Implementierung eines Elektrofahrzeugs mit REST API zur Steuerung und Kamerabildübertragung", written by Daniel Hanik, on which this project is based on, describes steps to build up the car. Amongst other things, the paper includes the structure and configuration of all the hardware components, the installation of the operating system and the software components, as well as the Python code that controls the servomotors. The steps of these paragraphs can be adapted to fit this project.

II. APPROACH

A. Goal

The goal of this project is to build an intelligent firefighting robot, based on the vehicle, camera, and Nvidia Jetson Xavier NX, provided by the university. The robot is supposed to look like a firefighter truck with working flashing blue light and siren sound. When the user manually activates an alarm, the vehicle autonomously follows a yellow line on the floor, by evaluating the camera input and controlling the servomotors accordingly. During that time, two blue LEDs are flashing alternately, and the siren sound file is played via a small Bluetooth speaker. Concurrently, an AI application for image recognition monitors the camera input, looking for a previously determined symbol of a flame or a fire station. When the AI application recognizes symbol of the flame, the vehicle stops and begins to put out the flame. This is accomplished by activating a small water pump inside a water tank within the body of the car for a few seconds. The water will spray out of a jet pipe mounted on the roof of the truck. After the vehicle finished this job, it will switch off the blue light and the siren. It then autonomously follows the yellow line back, until the AI application recognizes the fire station symbol. Here the vehicle will come to a hold and the process is finished.

In case the implementation of the goals stated above can be finished early in the time space of the project, there are some additional goals, which can be implemented afterwards. For example, the signal to start the whole process could be send autonomously by another image recognition software that sends an alarm when recognizing a flame symbol. The user would not have to start the alarm process manually. Furthermore, the robot's ability to navigate could be enhanced. One option is installing multiple possible destinations, that means the robot would have to decide where to go when encountering a crossing in the yellow line. Autonomous parking in a defined area or avoiding obstacles on the yellow line are additional extensions. On top of that, aiming with the water jet while putting out the flame can be enhanced. The vehicle would position itself autonomously so that the water lands in the desired container. Additionally, the image recognition software can be expanded to recognize different types of fires and put out a message containing the correct method to extinguish this kind of flame.

B. Steps

1) Preparing the hardware

a) Installing the Operating system

To install the operating system, a computer with a SD-card slot and a SD-card is needed. The Jetson operating system image is downloaded from NVIDIA and the SD-card is formatted with the "SD Memory Card Formatter" program's "quick format" option. The image is flashed onto the SD-card with the "Etcher" program. After the program has flashed the image and verified its integrity, it can be inserted into the SD-card port on the Jetson.

Now the initial setup can be conducted. A monitor via HDMI, a keyboard and a power cable are connected. The Jetson now powers on and boots. The NVIDIA Jetson software EULA must be reviewed and accepted. The system language, keyboard layout and time zone are configured. A user is created, and the hostname is setup. The partition size is set to the recommended value.

The official guide from NVIDIA was used [2].

b) Move Operating system to SSD

For Performance reasons, right after flashing the SD-card and initially setting the Jetson up, the boot process is changed so that the Jetson only uses the SD-card to boot the image and runs on the SSD afterwards.

Therefore, a guide was used [3].

c) Installung the software prerequisites

To use the camera, the corresponding software 'Pyrealsense2' is needed. Because there are no prebuilt

binaries for ARM processors like the one the Jetson has, it must be built from source. [1]

Additional required python modules were installed with the package manager pip.

d) Chassis

As in the practical work referenced above, the given chassis with the pre-mounted servomotors is used. Additionally, a wooden platform was placed on four stud bolts on top of the chassis. This platform serves as a foundation for the 3D-printed body parts, which give the vehicle the look of a firefighter truck. The body is composed of three parts that are fixated on the platform with multiple small blocks of wood that are glued on the platform. The blocks prevent the body parts from slipping horizontally, but the parts can still be removed by lifting them.

The front part is used as a mount for the camera so that the camera points downwards at a 45° angle.

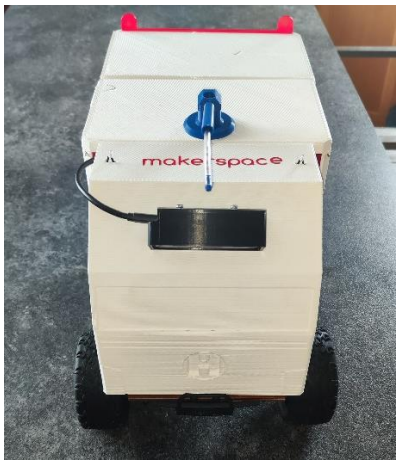


Fig. 1. Chassis front view.

The middle part contains the water tank and has lids on both sides, to grant easy access to the tank. The water tank itself is a lunch box, with the water pump inside. There are two holes in the tank to allow the cable and hose from the pump to run to the outside. From there the hose runs through the roof of the middle part, inside a 3D-printed jet pipe. In the front of the hose, there is a small 3D-printed outlet, which reduces the diameter of the hose. This allows greater range of the water jet.

The back part contains electronic devices like the Jetson, the voltage converter and the Bluetooth speaker. It is separated from the middle part to prevent water from leaking near the electronics. A trunk lid allows easy access to the electronic parts.

Fig. 2. Chassis side view.

e) Connecting electronics

Following the previously mentioned practical work [1], the Nvidia Jetson Xavier NX as well as the servomotors are powered by the given 4200mAh battery pack. Since the Jetson requires an input voltage of 18-19 volts, a voltage converter was used as described in the paper. The ground (GND), power supply (VCC) and signal cables of the servomotor used for steering are connected to the corresponding pins of row 2 on a servo driver, while the cables of the electronic speed controller are connected to the first row similarly. The servo driver itself is connected to the Jetson as described in [1]. The ground pin is connected to any ground pin of the Jetson, the SCL and SDA pins are connected to the corresponding pins of the I2C1 port (pin 5 and 3) on the Jetson and VCC is connected to one of the two 3.3V power supply pins.

The camera and the water pump are both connected via USB. It is important that the pump is connected to one of the two outputs of the USB hub closer to the power supply (port 1 or 2), while the camera is connected to either USB port 3 or 4. This is because, in order to switch off the pump, the whole USB hub is cut from power and this would affect the camera as well if connected to this hub.

The Jetson can establish Bluetooth connections, which is used to communicate with the small speaker, which plays the siren sound.

The two blue LEDs are each connected to a ground pin with their black cable. The red cable of one LED is connected to pin 11 while the other LED is connected to pin 12 with its red cable.

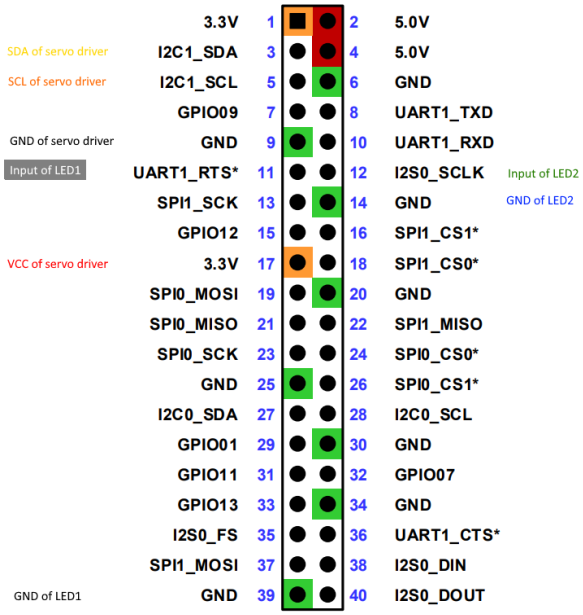


Fig. 3. Jetson Xavier pin layput [1].

2) Individual hardware control

a) Flashing light

To implement the flashing light of the firetruck, there are two blue LEDs on top of the car body. The LEDs have integrated resistors and are addressed by powering the I/O pins of the Jetson on and off. They are wired to two different I/O pins and a ground pin each.

The default state of both LEDs must be off. When powered on, they flash alternately. For this, there exists a python script

that uses the “RPi.GPIO” library, which wraps addressing the single pins.

b) Water pump

The fire extinguishing process is implemented by using a small USB-powered water pump that pumps some water from a small tank via a hose to a nozzle. For addressing the water pump, one of the USB-ports of the Jetson is used.

By default, the pump must be powered off. This is achieved by running a bash script using “uhubctl” that powers off the used USB-port before plugging in the pump.

If the Robot is commanded to release some water, there is another script that turns the respective USB-Port on. After enough water has been pumped out, the same script as in the beginning can be used again to turn the pump back off.

c) Output siren via loudspeaker

The implementation of the siren is achieved by using a small Bluetooth speaker which plays a mp3 file of a siren.

First, the Jetson must be initially paired with the speaker via Bluetooth. This is a one-time configuration task in the beginning which must be performed manually.

After the devices have been paired once, the connection-process can be automated via the command “bluetoothctl connect”. This process is supported by another script, checking regularly if the device is still connected.

For playing the mp3-file, the “playsound”-library is used in a python script.

3) Camera

a) Path guidance

The firetruck will be guided to its destination via a yellow line on the ground. To steer it accordingly, the camera mounted on the front of the firetruck will be used to detect the yellow line. This is achieved by finding the largest area of yellow pixels in the frame.

To do so, some preprocessing of the frame is necessary. First, the RGB camera stream is converted into the HSV (Hue-Saturation-Value) color space. This is done to isolate and find the yellow pixels more easily. Then, binary masks of the regions with yellow pixels are created. By applying morphological operations, noise and minor gaps are cleaned up to get smooth and solid edges of the detected yellow area for improved further processing and visual representation.

After the preprocessing, the largest contour is selected, and the center calculated. Depending on its position relative to the center of the camera frame, notifications with instructions of how the vehicle needs to steer are sent. A stop message is sent if no yellow line is detected for 1,5s.

b) Detecting start- and endpoints

For detecting the stopping points of the firetruck, an AI-object-detection model was trained using Roboflow.

The model was trained on two classes each consisting of a symbol, the first symbol being a fire, which is the destination of where the firetruck needs to go to put out the fire, and the second symbol being a fire station, where the firetruck will return to after all fires are extinguished. These symbols were printed out and various pictures were taken of them from different angles, with different lighting and from different distances.

Roboflow was utilized for the entire process of preprocessing the images and training the model. 358 pictures were labeled and normalized. Additional images were created using data augmentation and the model was trained on 500 images in total. With a precision of 97.1%, the model was deployed via the Roboflow API, which the firetruck calls. Depending on the result of the model, appropriate notifications including the information of which destination was reached are sent.

4) Integration

a) Wrapping the components with shared base class

The functionalities for controlling the individual components are summarized below. To standardize their use, all controlling elements inherit from a common base class Component. The requirement to be able to be started and stopped or interrupted, applies to all controls. For example, if the car drives off to a fire destination, the blue lights and siren must be switched on. However, as soon as the destination is reached, both must be stopped.

The base class “Component” encapsulates the usage of a thread including its start and stop process. This enables the GlobalController to manage the standardized interface according to the application.

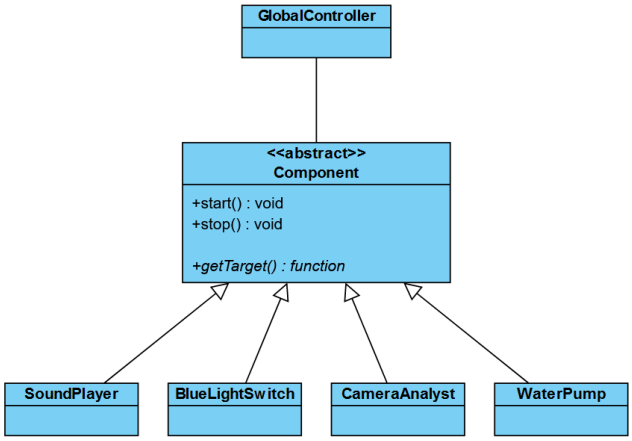


Fig. 4. UML diagram of the Component class und its subclasses.

The previously implemented processes for controlling the hardware are each integrated into an encapsulating class. The logic is transferred to a function which is used as the Component’s thread’s target. Additional cleanup code before stopping the hardware control can be linked into the stop process. For example, the flashing of the LED first ends with a LOW output before the thread is terminated.

b) Identifying the destinations

Ids are used to define the destination to be reached. To differentiate between the journey to the fire destinations and the return journey, the home ID is defined as 0. The controller can now use this information to determine whether privileges are required for a journey or not. This attribute is defined as follows:

If the destination ID is that of a fire destination, the blue lights and siren must be switched on during the journey. As soon as the destination is reached, these components stop. In return, the water pump is activated. The recognition of a successful extinguishing is simulated by a three-second wait. The pump is then no longer activated. Finally, the car reverses and the home ID is activated.

If the destination ID is the known home ID, neither privileges need to be switched on during the journey nor an extinguishing process started when the destination is reached.

c) Wrapping the REST-API

The existing REST interface [1] is controlled by an API adapter. This encapsulates the sending and configuration of the requests and only forwards the corresponding responses. In particular, the logically correct use of right, left and central steering is ensured.

d) Communication with Control-API

The car is still steered by the camera analysis. To be able to pass on current observations to the GlobalController, it offers a communication interface. A dedicated method is available for each possible event.

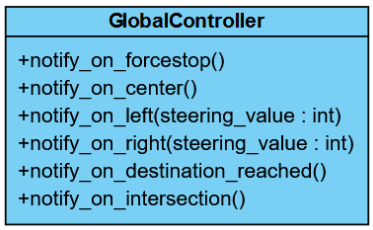


Fig. 5. UML diagram of the GlobalController methods.

These process the signal in each case. If a direct stop or steering signal is detected, corresponding POST requests are sent with the API adapter. The camera analyst continuously sends observations to the controller. To avoid overloading the REST interface, the latest messages received are cached and only new signals are forwarded to the API.

An interface for processing the detection of an intersection is also available for further development in the future. The signal is forwarded to an IntersectionGuide which would internally determine the direction to take and return a corresponding instruction to the controller.

If the destination is detected, the necessary follow-up actions are carried out as described in B.

e) Usage

In order to use the built environment, first of all the REST-Api to control the servo-motors must be started. Next, the script controlling all components must be run with sudo rights in order to have the needed access to control hardware components. This script instantiates the global controller and manually triggers the alarm. Due to python package interoperability version 3.11 must be utilized for running both scripts with the currently installed packages on the Jetson.

When the car's IP address changes, this must be applied in the 'config'-file, which is used by the ApiAdapter to communicate with the REST-Api.

III. Experiments

During the development process, multiple experiments to test the robot were conducted to make sure that every hardware and software component work both individually and together.

This included individual tests for controlling the flashing lights, enabling and disabling the USB-port to which the water pump is connected, which was visualized by using a USB-lamp, outputting the siren sound via loudspeaker, detecting the yellow line and detecting the two symbols the AI was trained on for the start- and endpoints.

After testing the components individually, everything was put together to be managed by the GlobalController and the first driving tests with the chassis were conducted. The setup for these tests were as follows: The printed-out symbols for the fire station and the fire were laid on the ground, with a curved yellow tape line connecting the two. The REST-API of the car and the GlobalController were run to start the car.

With these tests, the steering was modified for the firetruck to follow the line more accurately, and any communication errors between the individual software components and the GlobalController were ironed out.

However, the 3D-printed body of the firetruck and the water tank made the firetruck too heavy, which resulted in worse steering and the battery failing to move the vehicle continuously, therefore a ball roller was added to relief a bit of weight.

The final acceptance test is conducted in a more real environment outside. Unfortunately, the combination of the rougher ground and the weak battery resulted in the firetruck having to be supportively pushed to drive. Everything else worked well, as indicated by the following table.

Test	**Components**				
	Blue lights	***Water Pump***	***Siren***	***Path guidance***	***Start- and endpoints***
successful	✓	✓	✓	✓	✓

The final acceptance test reached all the initial goals of the project.

IV. Conclusion

This project built and implemented an intelligent firefighting robot based on a vehicle with a camera and a Nvidia Jetson Xavier NX. After being activated, the truck follows a yellow line with active blue lights and a siren, until a fire symbol is detected and puts out the fire via a water pump. Having finished, it drives back with no flashing blue lights and siren, until a fire station symbol is detected, where the process is finished.

Unfortunately, due to the many hardware complications and errors, none of the additional goals were fully implemented in time. However, because the infrastructure of the project is kept expandable, additional features can easily be added.

A future project could add the following features: Autonomously starting the process. To do so, a REST-Api connecting to the global controller was already set up. Dealing with multiple destinations and crossings. The existing interface for processing the detection of an intersection can be used. In the intersection guide a map needs to be set up manually or automatically. The existing structure expects for each destination id a list of the consecutive direction commands for each intersection. Further extensions can be: Implementing autonomous turning back after the fire is extinguished and autonomous parking after the fire station is reached. Implementing the detection of obstacles and according steering around them. Implementing better aiming of the water pump by additionally centering the firetruck in front of the fire symbol so the center of the fire is hit. And

lastly, implementing the detection and corresponding extinguishing of different types of fires.

[1] D. Hanik, “Aufbau und Implementierung eines Elektrofahrzeugs mit REST API zur Steuerung und Kamerabildübertragung,” Universtiy of Applied Sciences Hof, 2023.

[2] “Getting Started With Jetson Nano Developer Kit,” *NVIDIA Developer*, Mar. 05, 2019. https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit#intro

[3] kangalow, “Jetson Xavier NX - Run from SSD,” *JetsonHacks*, May 29, 2020. https://jetsonhacks.com/2020/05/29/jetson-xavier-nx-run-from-ssd/ (accessed Feb. 11, 2024).

Drawing Dobot

Julian Köglmeier
Fakultät Informatik
Hof University of Applied Sciences
Hof, Germany
julian.koeglmeier@hof-university.de

Julian Doberauer
Fakultät Informatik
Hof University of Applied Sciences
Hof, Germany
julian.doberauer@hof-university.de

Paul Schaller
Fakultät Informatik
Hof University of Applied Sciences
Hof, Germany
paul.schaller@hof-university.de

Yusuf Mehderoglu
Fakultät Informatik
Hof University of Applied Sciences
Hof, Germany
yusuf.mehderoglu@hof-university.de

Ronny Grebner
Fakultät Informatik
Hof University of Applied Sciences
Hof, Germany
ronny.grebner@hof-university.de

***Abstract*—This document describes how the drawing mechanisms of the Dobot Magician Arm are implemented, how to use the application and how someone could add more functionality to the program.**

***Index Terms*—Dobot Magician, Drawing, Tic-Tac-Toe, Machine Learning, Computer Vision, Robot Arm**

I. Acronyms

CP	continuous path
SVG	Scalable Vector Graphics
PNG	Portable Network Graphics
JPEG	Joint Photographic Experts Group
PTP	point to point
CV	computer vision
UI	User Interface
GUI	Graphical User Interface
CNN	Convolutional Neural Network
NN	Neural Network

II. Introduction

This paper discusses how image and text drawing functionality has been implemented using the Dobot Magician robot arm. Images can either be drawn using Scalable Vector Graphics (SVG) or other image file formats. Another functionality is to play a game of Tic-Tac-Toe using a camera and a pre-trained Convolutional Neural Network (CNN). The program is written in Java based on the Java example code from Dobot [1].

III. Quick-start

The Quick-start is intended to be used on a Linux system. JDK 17 or higher must be installed on the system.
To run the program, `sudo ./gradlew run` needs to be executed in the base directory of the project. This builds the program and then runs it. For communication with the Dobot over the USB cable, root privileges are required, thus it is important to run the program as the root user or with sudo. Otherwise the program will fail to make a connection with the Dobot.
The Dobot's default orientation after turning on and homing is opposite the side of the reset and homing buttons. From this orientation the x coordinate will increase when going forward, the y coordinate when going left and the z coordinate when going up.
The drawing area with minX, minY, maxX and maxY can be adjusted in the settings. Leaving the Dobot in the default orientation these values do not need to be changed, as they are set accordingly by default.
Before drawing however, the height the Dobot should draw at needs to be set to the height of the paper. This can also be done in the settings. The paper should be a bit elevated off the table the Dobot is standing on, as this increases the available drawing area. To find out the height of the paper, the Dobot arm should be manually moved down by pressing the lock button towards the end of the arm until the pen attached to the Dobot's arm makes contact with the paper. After moving the Dobot arm to the correct height, the "Set Height" button can be pressed. Alternatively, the height, which is the z value, can be read from the terminal and then configured manually in the settings. After setting the height, the arm should be moved up a bit so it doesn't touch the paper anymore.
The Dobot is now ready to draw images as well as play Tic-Tac-Toe if a camera is properly set up and connected to the computer running the program.

IV. Physical Setup

Our setup consists of the Dobot and a cardboard box that elevates the stack of papers for the Dobot to draw on. We placed the paper opposite of the side with the reset and homing button (see III). No additional hardware is required if the Dobot is only used for drawing SVG, Portable Network Graphics (PNG) and Joint Photographic Experts Group (JPEG) images or text.
For playing Tic-Tac-Toe, a camera is required. We use a smartphone and connect it to the computer via DroidCam. A

folded piece of cardboard is placed next to the paper and the phone is placed in a properly sized cutout in the cardboard. With this setup Tic-Tac-Toe can now be played.

V. JavaFX User Interface

The User Interface (UI) is written using the JavaFX [2] framework.

A. The Main Window

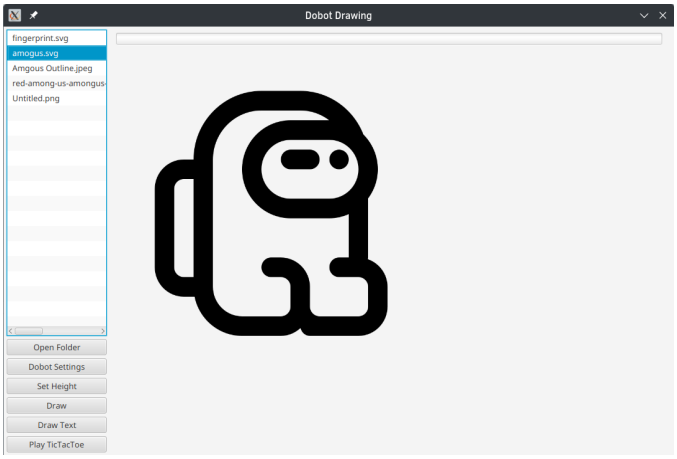


Fig. 1. Main Window

In the main window named ”Dobot Drawing” a basic overview is given. At the top of the left hand side, there is a view which lists all the available images in the selected folder. Only SVG, PNG and JPEG images can be drawn and thus only those image types are shown in this view.
Below this view there is a set of buttons which can be used to access different functionalities of the Dobot and the Dobot Drawing application. The first button labeled ”Open Folder” can be used to select the current folder. This folder is used for the view above the buttons that lists all the available images for drawing. The second button named ”Dobot Settings” is used to open a popup window. In this popup window, the user can change various settings regarding the drawing. The available settings are: Changing the height (offset from the table), the minimum and maximum x values, as well as the minimum and maximum y values of the drawing canvas. These settings can be used to specify the canvas width and length. The ”Draw” button initiates the drawing mode, where the user can select what parts are drawn for raster images. For SVG images, the drawing process will be started immediately. The ”Draw Text” button opens a small popup window, where the user can enter a text to be written using the Dobot arm and a font in which it is drawn. The final button with the label ”Play Tic-Tac-Toe” opens the Tic-Tac-Toe view, where the user can change settings regarding the game and play Tic-Tac-Toe against the Dobot.
At the top of the main window there is a progress bar to indicate how much of the final image or the text the Dobot has drawn already.
Below the progress bar, there is an image preview field, which shows the currently selected image.

B. The Image Conversion Window

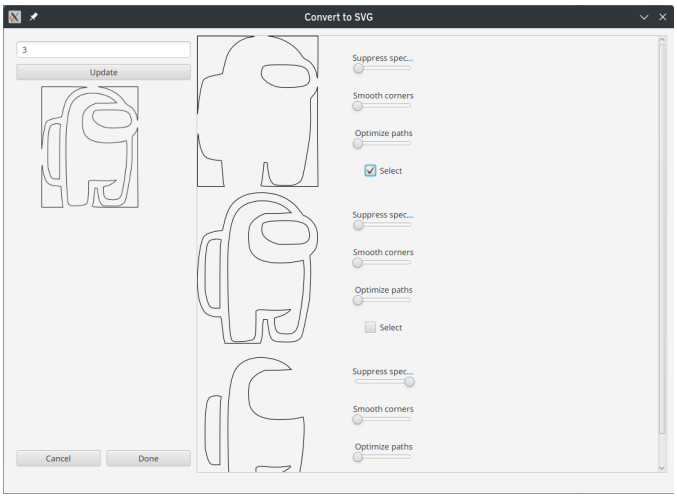


Fig. 2. Image Conversion Window

If the selected image is not an SVG image, the image conversion window is opened once the ”Draw” button is pressed.
In the image conversion window the user can specify the number of clusters the program will look for in the selected image. By pressing the ”Update” button, the program calculates the clusters.

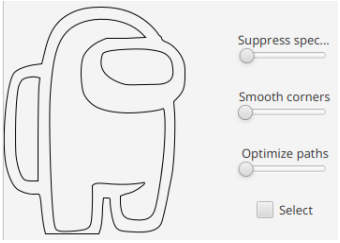


Fig. 3. Cluster

The clusters are presented on the right. The user can specify multiple settings for every cluster and can select the ”Select” checkbox to include a cluster in the final image. If the cluster algorithm finds unsatisfying clusters, the user can use the ”Suppress speckles”, ”Smooth corners” and ”Optimize paths” sliders to adjust the cluster’s appearance and shape. For further information have a look at section VI-B.
The left hand side of the image conversion window also contains a ”Cancel” button to stop the cluster selection process and a ”Done” button to start drawing the image presented in the preview section.
The preview on the left is the sum of all the selected clusters and the cluster settings. This image will be drawn by the Dobot arm.

C. The Tic-Tac-Toe Window

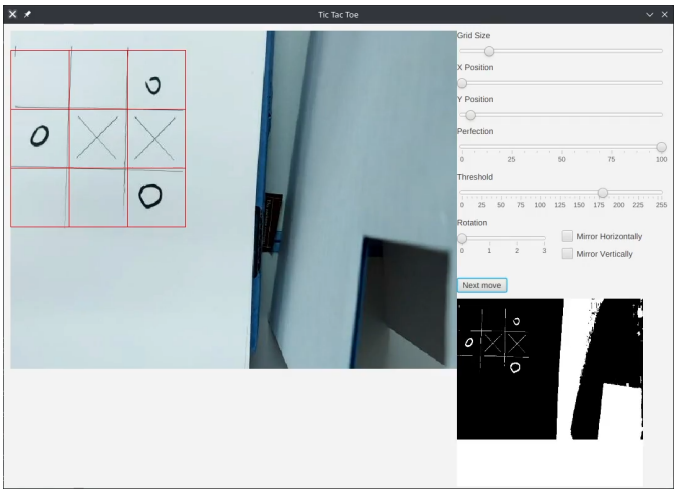


Fig. 4. Tic-Tac-Toe Window

The Tic-Tac-Toe window consists of a preview of the camera on the left and sliders to the right. With the sliders ”Grid Size”, ”X Position” and ”Y Position” the user can adjust the grid overlay that is drawn on the image. The preview image can also be mirrored using the ”Mirror Horizontally” and ”Mirror Vertically” checkboxes and rotated using the ”Rotation” slider. The preview image should be rotated and mirrored so the small extra line in one of the fields is aligned vertically in the top-left field of the Tic-Tac-Toe grid, as seen in figure 4.
Once the preview image is rotated correctly and the overlay aligns with the Tic-Tac-Toe grid drawn by the Dobot, the user should set the threshold value with the ”Threshold” slider. The Tic-Tac-Toe field in the black-and-white preview on the bottom right should be completely visible. The lines of the board should be visible and not obscured by shadows that appear white. The empty square fields of the Tic-Tac-Toe board should be completely black, except for those where either the Dobot or the User has already drawn a shape in which case the shape should be completely visible. A good example of how the threshold value should be used is seen in figure 4.
After all the settings are set, the user must make the first move of the game. With another slider ”Perfection” between 0 and 100, the user can adjust the ”skill level” of the Dobot. Zero means the Dobot will make a random valid move, while 100 means the Dobot will always make the best move possible.

VI. Drawing Images

Clicking on the ”Draw” button in the UI the program checks if the selected file is a SVG file or another image file (e.g. PNG or JPEG). Based on the file type the program either kicks off the drawing process or opens an additional window for the PNG/JPEG to SVG conversion process.

A. Drawing Vector Graphics

The `SVGDrawing` class in the project is responsible for loading and processing SVG files. It uses the Apache Batik [3] library to extract and draw SVG paths.
There are multiple methods to process SVG files, which take either a path to a file, an `InputStream`, or a byte array containing SVG data. These take care of loading the vector graphic and pass it to the main `generatePoints` method to convert it into a drawable form.
First, the SVG file is loaded as an `SVGDocument` and all of the path elements are extracted. Other types of SVG elements are not yet supported (see section X-B). The paths are first converted into a series of points using the `SVGPathSupport` class. This is done by iterating over each path in increments of `STEP_SIZE` normalized units of length. There is also a check to see if the current segment has changed to make sure that all segments are properly closed without any gaps. The points are then normalized to be in the range `0.0 - 1.0` according to the SVG file’s view box.
There is also some code to support single-line fonts (see section IX-C), which makes sure to only draw half of the points of each segment, because each letter would be drawn twice otherwise. That feature is not finished though and does not yet work as intended.
When the SVG file is drawn, the normalized coordinates are scaled to the drawing canvas and sent to the Dobot using continuous path (CP) mode commands.

B. Drawing Raster Graphics

Conversion of PNG or JPEG images is done inside the `RasterImageConverterController` along with the `RasterImageConverter` class. To turn the images into SVG paths, the svg_converter [4] library is used. It is originally a standalone application, but the relevant parts to convert raster images without the GUI code have been integrated into our project. The library uses a Java implementation of the potrace [5] utility to convert an image into multiple segments which can then be used to create an SVG file.
After the user has selected an image to convert, an instance of the `RasterImageConverter` class is created which holds all of the information about the extracted clusters.
The user then supplies an approximate number of clusters (segments) to be extracted. This number is passed to the `RasterImageConverter` using the `createClusters` method. The method then calls the `PoTraceService` class to extract the detected clusters from the source image. The clusters are then stored as a list of `DetectedCluster` objects.
The segments can afterwards be adjusted to suppress speckles, smooth corners and optimize paths, which can improve the appearance of the resulting vector graphic, especially for lower resolution images. These parameters are also contained in the `DetectedCluster` class. They are updated directly in the corresponding cluster object and afterwards, the `updatePreview` and `updateGlobalPreview` methods are called to make sure that the previews are up to date with

the user selection.
The user can then choose which of the resulting clusters to draw, and can thus exclude certain clusters from being drawn by the Dobot to speed up the drawing process or to remove duplicate lines. This is done by updating the `selected` boolean of the cluster object and then updating the previews. After the user has decided which segments to draw, they are converted into an SVG file and drawn using the SVG drawing process (see section VI-A).

VII. Drawing Text

Clicking on the ”Draw Text” button in the UI opens up an additional window. The user can write text in the empty field, select the desired font and start the drawing process with the ”Draw” button.
Text drawing is implemented in the `GUIController` of the project. First, all of the fonts installed on the user's system are listed using the `Utilities.getAllFonts` method. These are shown to the user as a selection drop-down menu.
Afterwards, the `SVGGraphics2D` class of Apache Batik [3] is used. The font is set using `setFont` and the text is drawn onto the vector graphic using the `drawString` method. There are also some calculations to make sure that the text is centered correctly inside the image and to support multiple lines of user input. The SVG file is then generated with the `textAsPath` option enabled to convert the provided text into SVG paths instead of an SVG text element. The view box of the resulting SVG document is also set to the full size of the text to make sure none of the text gets drawn outside of the drawing area and the maximum possible space is used.
The Dobot then draws the resulting SVG file using the SVG drawing process (see section VI-A).

VIII. Tic-Tac-Toe

A. JavaCV and Model

JavaCV [6] in conjunction with DeepLearning4j [7] is used for the program to be able to detect the state of the Tic-Tac-Toe playing field. In the ”Tic-Tac-Toe” window the user sees an image of the attached camera. A 3x3 grid must then be aligned over the playing field that has been drawn by the Dobot. The user can start by playing either ”X” or ”O”, the Dobot automatically plays the opposing shape.
The `TicTacToeController` is used to control the flow of the Tic-Tac-Toe game, as well as to update the video feed, by periodically fetching the frames prepared by the `TicTacToeRecognizer`. After the user verifies they have taken their turn, the `TicTacToeRecognizer` then prepares the current frame for the pre-trained CNN [8] inside the `updateFrame` method.
When the frame is grabbed from the camera feed, the image is first mirrored and rotated according to the settings the user has set. For this, the `AffineTransform` class from the Java standard library is used. Afterwards, a copy of the image is created on which the grid preview is drawn for the user interface. A third copy of the frame is also created. An inverse threshold filter is applied to convert it into a black-and-white image, according to the threshold setting the user has set. It is displayed in the user interface and also used for further processing. The image is also saved as a file called ”frame.png” for debugging purposes.
Recognizing of the shapes on the Tic-Tac-Toe field is done inside the `recognizeField` method of the `TicTacToeRecognizer` class. First, the black-and-white frame is split into nine separate cell images according to the grid specified in the UI. To improve the accuracy of the CNN, four white borders are drawn around the image (see IX-C for more details). The sub-images are saved as files called ”subimage-X.png”, where X is the index of the cell in the Tic-Tac-Toe field. Afterwards, each sub-image gets fed into the model to predict the state of the corresponding Tic-Tac-Toe cell. A cell can either be blank, contain an ”X” or an ”O”. The resulting field gets returned and the internally saved field of the `TicTacToe` class gets updated with the new move.

B. Game

We made a hard-coded version of the game Tic-Tac-Toe that uses a ”perfectness”-score between 0 and 1 which is used as a probability to choose if the next move is played randomly or the best move according to the current state of the game. To determine the best move, we use a set of simple rules and game-state patterns. This is possible, because Tic-Tac-Toe is a solved game, and the best possible move can always be quickly determined algorithmically.
When receiving the new game board after the player took their move the algorithm compares the new board and the old board and puts the shape that was supposed to be played in the field that has not been set before. This is done with the intent that if we recognize the wrong shape it is replaced with the shape that is supposed to be played. For determining what shape has to be played we have to save which shape played first. This is done after the first move when only one shape is in the playing board. The `TicTacToe` class also provides some convenience methods to get the current state of the game.
The `TicTacToe` class is also responsible for determining when the game is over and who won. In the `TicTacToeController`, after either the user or the Dobot has made a move, this is checked. If the game is determined to be over, a popup appears with the winner of the game or a message that the game has ended in a draw. If there is a winner, the Dobot also draws a line through the three winning cells.
If the game has not ended yet, the next move is chosen and the Dobot draws its shape in the corresponding cell on the board.

IX. Experiments

A. CP mode vs. point to point (PTP) mode

Initially, SVG paths were drawn using the PTP mode of the Dobot. This mode, while being more precise than the CP mode, is significantly slower. Therefore, CP mode is used for drawing the paths.

B. JPEG files

While the program supports drawing both PNG and JPEG files, PNG files are a lot higher quality because they use lossless compression. The lossy JPEG compression leads to a lot of ”speckles” (small segments) and noisy edges around segments due to the algorithm potrace [5] uses. Therefore, it is recommended to use SVG and PNG files whenever possible.

C. Template matching vs. a CNN

Before using Deeplearning4j [7] an attempt was made to use JavaCV template matching to recognize the Tic-Tac-Toe playing field. However using template matching, we were not able to reliably distinguish between ”O”s, ”X”s and blank cells of the grid.
Instead we decided to take a machine learning approach with Deeplearning4j using a pre-trained CNN. Initially when using the model the predictions of the cells were still volatile and not reliably correct. After experimentation we found out that the images used to train the model contained not only the drawn shapes, but also parts of the grid itself. Thus we adjusted the cell size to include parts of the grid drawn by the Dobot. This increased the model's ability to predict a cell correctly, however there were still frequent occasions when the model was wrong and would predict an incorrect shape. Upon further investigation we determined that the model strictly requires lines to be present in the picture of a cell, due to over-fitting. As this was not always the case after preparing the image for the model, it was still wrong at times. To make sure there are always lines present in the sub-image of a cell we draw a white rectangle around the cell. This successfully tricks the model and it now works reliably and can distinguish between blank, ”X” and ”O” cells with high confidence.

D. Single-line fonts

When implementing text drawing, we also experimented with adding support for so called ”Single-line fonts”, which are fonts that are traced along the center line of the letter rather than around it. This would allow the Dobot to draw text in a way more closely to as one would write text by hand. However, there were some issues when using these fonts which resulted in letters only being drawn halfway and we decided to abandon this in favor of the Tic-Tac-Toe implementation.

E. Select camera device

Initially the device id of the camera was hard coded. We tried to get and list all connected video devices for the user to select the appropriate camera using JavaCV. Using the `videoInput.listDevices()` method provided by JavaCV, we found out that this functionality works on Windows only [9]. Since we were on Linux and compiled the DobotDLL for Linux, we could not use this function to list all video devices. As an alternative we added an input field for the user to enter the device id before starting to play Tic-Tac-Toe. To find out the correct device id the user now has to find out the correct device number by using VLC [10] for example.

X. Conclusion

A. Implemented program

Utilizing the application provided by us, the user may now successfully draw images of the types SVG, PNG and JPEG, as well as text in a specified font and play Tic-Tac-Toe against the Dobot arm using image recognition and a camera / mobile phone connected to the computer running the program.

B. Possible expansions

One possible extension could be that more SVG elements such as text, embedded images etc. can be drawn by the Dobot. At the moment we only support paths. This could be accomplished by converting these types of elements to a point to point representation, that simply is sent to the Dobot.
Another useful feature to add could be a way to automatically detect the position of the Tic-Tac-Toe grid in the camera image. This way, the grid doesn't need to be aligned manually. A possible implementation for this could be by using the pattern-matching functionality of OpenCV to get the position of the playing field. By scaling the pattern for the pattern-matching, the size of the playing field could also be figured out.
Another extension would be the addition of a camera selector. Currently a pop-up window opens when starting the Tic-Tac-Toe game, prompting the user to enter the device id of the camera used to detect the playing field. Instead of using a prompt asking for the device id, a drop-down menu could be added, which lists all the found camera devices and allows the user to select one. To get a list of all device ids a library like ffmpeg [11] could be included.
An improved physical setup for playing Tic-Tac-Toe with the Dobot could be created. Instead of using a basic contraption made of cardboard and a phone or a webcam, a more permanent setup (e.g. a 3D printed one) is recommended to decrease the risk of accidentally moving the camera or the paper while playing. The camera could then also be placed directly over the paper instead of the side of it.

References

[1] https://www.dobot-robots.com/products/education/magician.html.
[2] https://openjfx.io/.
[3] https://xmlgraphics.apache.org/batik/.
[4] https://github.com/snow3442/professional/tree/master/svg_converter.
[5] https://potrace.sourceforge.net/.
[6] https://github.com/bytedeco/javacv.
[7] https://github.com/deeplearning4j/deeplearning4j.
[8] https://github.com/tempdata73/tic-tac-toe.
[9] https://github.com/bytedeco/javacv/issues/1650.
[10] https://www.videolan.org/vlc/.
[11] https://ffmpeg.org/.

NAO robot that imitates human movements using Mediapipe

Costilla Caballero, Salvador
Applied Robotics
Hof University of Applied Science
CDMX, Mexico
salvadorcoscab@gmail.com

***Abstract*—In this project, we programmed a NAO robot from Aldebaran to mimic human movements captured by a webcam using Mediapipe. While this type of human imitation has been done before, we aimed to approach it differently. Unlike most other projects utilizing multiple cameras or depth sensors like the LIDAR sensor, our focus was on a novel methodology. The project is still in progress, with the robot currently capable of moving its arms, albeit not with high precision. Nonetheless, we see this as a promising starting point for future developments in computer vision and robotics.**

***Index Terms*—NAO, Mediapipe, Human imitation, aldebaran, robotics**

I. State of the art

In this section, we will explore various methods for controlling the movement of a NAO robot using the Mediapipe Framework, as well as discuss related work involving human imitation with the NAO robot.

According to the NAO robot documentation [1], there are two primary methods for posing the NAO robot: using joint angles and employing inverse kinematics. While other methods exist for walking or performing predefined poses, these two are fundamental to our project's scope.

Regardless of whether inverse kinematics is employed, there are two main ways to control the effectors to achieve a pose:

- Animation methods, which operate within fixed time frames and involve blocking functions.
- Reactive methods, suitable for real-time control with non-blocking functions.

Within the *ALMotionProxy* module, numerous functions pertain to joint movement. Some notable animation methods include:

- *ALMotionProxy.angleInterpolation*, enabling movement from one specific position to another within a specified time.
- *ALMotionProxy.angleInterpolationWithSpeed*, facilitating movement between specific positions within defined time and speed parameters.

An essential method is:

- *ALMotionProxy.setAngles*, enabling the manipulation of single joint angles or sets of joint angles on the robot.

Furthermore, the robot's control extends to inverse kinematics. Here, classical IK solvers and generalized IK solvers (Whole Body control) are employed. Some pertinent animation and reactive methods related to inverse kinematics include:

- *ALMotionProxy.positionInterpolation*, facilitating movement between specific positions within a specified time frame.
- *ALMotionProxy.setPosition*, enabling precise positioning of the robot in 3D space.

BlazePose [2] is an architecture for single-person 3D pose detection based on convolutional neural networks. It is integrated into the Mediapipe framework, enabling the detection of poses from videos or images. The model is capable of detecting 33 landmarks (see Figure I), which can be interpreted as points in a 3D space.

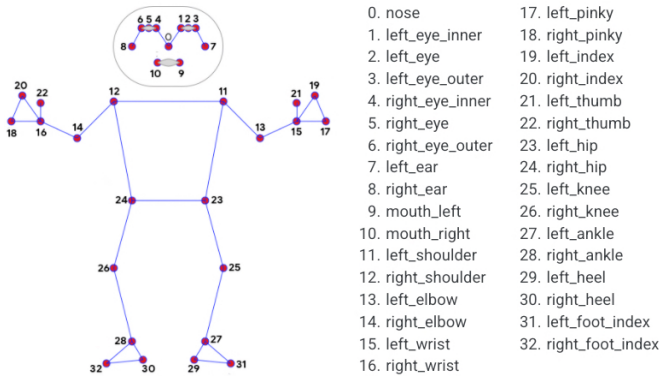


Fig. 1. Landmarks from the Mediapipe library.

The Landmarker offers various configuration options, including:

- *model_complexity*, This parameter allows selection among three different models - lite, full, and heavy. Depending on the chosen model, both accuracy and speed can vary.
- *min_detection_confidence*, It enables setting the minimum confidence level required for detecting a pose.
- *min_tracking_confidence*, This parameter allows setting the minimum confidence level required for tracking a pose.

In a thesis titled "Mimetización de movimiento por robot antropomorfo basado en imágenes de sensor de profundidad" [3], the authors utilize the NAO robot to mimic human

movements using a Kinect sensor. They adopt a client-server approach to transmit data from the Kinect sensor to the robot via the NAOqi API. In the client-side implementation, they employ the *pykinect2* library to retrieve data from the Kinect sensor, which consists of 25 landmarks representing the human body. This data is then relayed to the server-side using the *socket* library. On the server-side, the angles between the joints are computed and transmitted to the robot using the *ALMotionProxy.AngleInterpolation* method.

II. Approach

A. Objectives

During this project we had some objectives which were:

- Solve the computer vision part, that in this case was done with Mediapipe.
- Make the robot move using the NAOqi API.

The problems were that, in order to work with the NAOqi API, we had to download and setup the SDK which is written (by the moment of this project) in Python 2.7 and the Mediapipe is only availble in Python 3.7 or higher. That is why we had to add as an objective to make the two libraries work together.

B. Methodology

The methodology that we used to solve the objectives above was to first divide the problem into modules and try to solve them separately, and then join them together and solve the problems that arose from the connection between the two modules. The main two modules were the computer vision and the robot movement.

C. Computer vision

In the beginning, the computer vision part was done very easily, because Mediapipe has a lot of usage examples and good documentation, such as the video ”AI Pose Estimation with Python and MediaPipe” [4], that helped us a lot on getting the landmarks and calculate some angles. On the other hand working with the NAO robot was a much more difficult, because we did not know how to start.
For the computer vision part we used the opencv library to get images from both camera and videos and then we used the Mediapipe library to get the poses from the images. We used also the opencv library to plot the landmarks that the Mediapipe library detected, so we could see how the landmarks were being detected. This landmarks conformed an object with 33 points, which represents landmarks of the human body. (See figure I)

D. Robot movement

Then we started working with the robot movement part, to do this we used the NAOqi API. We realized, after reading the NAOqi documentation that there is two ways to control the robot for doing poses. The first one was using angles and the second one was using inverse kinematics. We decided to use the first one because we thought that with the landmarks, obtained with Mediapipe, we could easily get the angles of each join of the robot, and also the project that we found on internet used this approach. All of this joint-angle approach is written in the *ALMotion* module of the NAOqi API. In summary the robot has various joints, and each joint can be moved using a specific angle. These joints work in a similar way as the human body joints, allowing some joints to pitch, roll, and yaw. (See figure II-D)

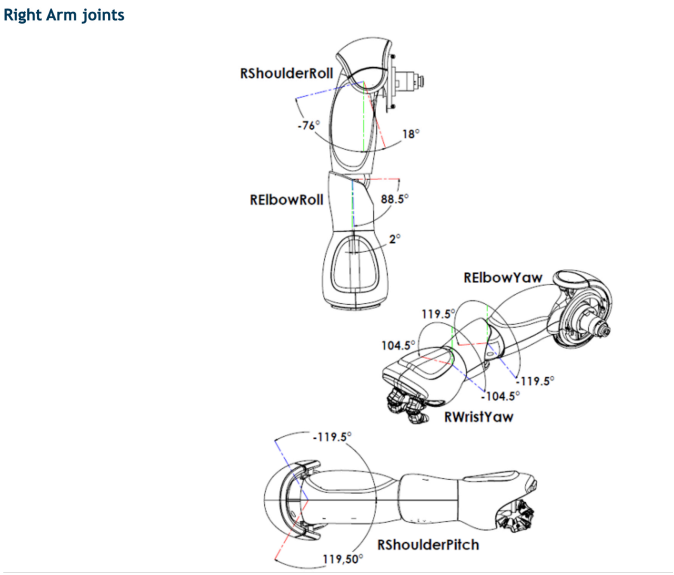


Fig. 2. Joints in the arm of the NAO robot.

E. Connection between the Computer vision and the Robot movement

After doing some testing on both parts, we had to somehow connect them. In order to do the connection between the robot and the computer vision part, we tried with *execnet*, which is a library that allows you to trigger python scripts from another python script, even with different versions of python. But we had some problems sending the data from the computer vision part to the part in which we move the robot, so we decided to try a different approach.

We decided to use sockets to send the data, as we got inspiration from a project that we found with a similiar approach, in which the author used sockets to send the data from a kinect sensor to the robot. This solved everything, about the connection between the scripts, because it uses the client-server approach, allowing us to use python 3.10 for the computer vision part that is the client and python 2.7 for the Robot movement part that is the server. We just had to do some encoding to send the data.

F. Testing the implemented code and solving problems

After changing the code that we found to work with Mediapipe and OpenCV instead of the kinect sensor. We did some testing on the functions that were already implemented in the server side, but they performed very bad, but at least, we saw that the robot moved.

Also the display of the images was very slow since the code had a delay to send the data, in order for the server to

process it. We solved this using a thread to get the data from the displayed image and send it to the server with the thread without messing up with the display.
As we said before after changing a lot the parameters and definition of the functions that were implemented before we realized that we was not sending the right data to the robot from the server, so we decided to change the approach and instead of sending the coordenates of the landmarks to the server, we decided to send the angles between them and send them directly to the server. This would help us to see what angles we was sending to the robot by showing them on the displayed image.

After we was able to see the angles that were being sent to the robot, we realized that the angles sent were not very good in the function definition, this is because the kinect sensor and Mediapipe have different coordenate systems, so we had to change the defintion of the functions.
This did not work very well, though the movements were more accurate, the robot was not imitating the movement of the hands perfectly. So we decided to change the function. Instead of using geometry we tried with vectors and the angles between them.

G. Vector approach

We saw that as we had the landmarks, that represents a point in the space, we could calculate a vector between two points (See equation 1), and we could use the angles between the vectors to move the robot (See equation 2). But then we realized that we could also calculate the angles between the projection of the vectors (See equation 3) in a generated plane from a normal vector (See equations 4a,4b,4c). We tried this for one joint, that was the right shoulder pitch. We got the normal vector from the left shoulder to the right shoulder to generate the plane, and then we got the vectors from the right shoulder to the right elbow and from the shoulder to the right hip. We projected these vectors in the generated plane and then we calculated the angles between them.

$$\vec{v} = \vec{p_2} - \vec{p_1} \tag{1}$$

$$\cos(\theta) = \frac{\vec{v} \cdot \vec{w}}{\|\vec{v}\|\|\vec{w}\|} \tag{2}$$

$$proy_H \vec{v} = (\vec{v} \cdot \vec{u_1})\,\vec{u_1} + (\vec{v} \cdot \vec{u_2})\,\vec{u_2} \tag{3}$$

$$\vec{n} \cdot (\vec{p} - \vec{p_0}) = 0 \tag{4a}$$

$$\vec{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4b}$$

$$\vec{p_0} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix} \tag{4c}$$

We implemented a function in python to do the calculations mentioned above and this function did not work well neither when we tested it, as we could see the angles on the displayed image, we realized that the Mediapipe library did not detect the z axis very well, even with a more complex model. Then we remembered that the kinect sensor has a LIDAR sensor, that allows him to detect, not only position in x and y, but also depth in a better way than the Mediapipe library. That is why we did not continue with this approach and as we did not have more time to finish the project, we decided to leave the functions that were already implemented.

III. Contribution

The main difference between our project and the others is that it has a different approach to solve the problem, relying only on one camera and machine learning to detect and recreate the human poses, instead of using a lot of sensors. This is a contribution to the state of the art, because it implies that if there were better machine learning models, the robot could imitate the human movements in a better way. There is also a contribution to the work done before in "Mimetización de movimiento por robot antropomorfo basado en imágenes de sensor de profundidad" [3], because we can see the angles that are being sent to the robot, which can be useful for further research.

IV. Experiments

To do the experiments and testing we used Choreographe, which has a simulator mode, in which you can see a NAO robot and move it using the localhost as the robot IP (See figure 3). We used this to test the robot movements, but we also used a real robot to see the movements in a real robot (See figure 4). We had a script to try the robot movments and another script just to try the computer vision part. The union of both scripts, as we said before, was done using sockets. So we had to run two scripts in the terminal, one for the server and one for the client.

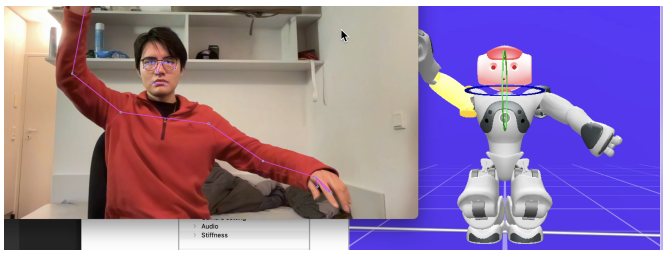


Fig. 3. NAO robot in the Choreographe simulator.

We used a webcam and videos to see the angles that are being sent to the robot. To first initaite the application you have to run the server part, using a python 2.7 interpreter. This can be done using the following command, in which you have to specify the ip address of the robot as an argument:

```
python2 server.py "ip_address"
```

Then you have to run the client part, using a python 3.10 interpreter. If you want to send the landmarks using a webcam you can use the following command:

```
python3 client.py
```

Fig. 4. NAO robot moving.

On the other hand, if you want to send the landmarks using a video you can use the following command:

```
python3 client.py "path_to_video"
```

Sometimes the robot falls and if this occurs the application might not work or would stop working, so we had to be careful about this, because sudden movements of the robot could make it fall. If the application stops working, there is no other option than to restart the application repeating the steps above.

V. Conclusion and Future Work

Even though the project is not finished, we think is a good start for just using computer vision to apply it in robotics.
There is a lot of work to do, such as improving the calculation of the angles or use both inverse kinematics and programming the angles to move the robot.
In order to use the legs of the robot is necessary to do further research, because we have to consider many other things such as the mass center of the robot, and take care that the movements are not too abrupt to make the robot fall.

Acknowledgment

We want to thank our proffesor, Dr. Christian Groth for helping us during the project and thank also to the Hof University of Applied Science for giving us the opportunity to work with the NAO robot, which was a great experience.

References

[1] NAOqi API, *http://doc.aldebaran.com/2-5/naoqi/motion/index.html*

[2] Mediapipe Pose Landmarking*https://developers.google.com/mediapipe/solutions/vision/pose_landmarker*

[3] I. Irigoyen, "Mimetización de movimiento por robot antropomorfo basado en imágenes de sensor de profundidad," Trabajo Fin de Grado, Facultad de Informática, Grado de Ingeniería Informática, Universidad del País Vasco, 2020.

[4] Nicholas Renotte - Robotics and AI, "AI Pose Estimation with Python and MediaPipe," YouTube, 2021. [Online]. Available: https://www.youtube.com/watch?v=06TE_U21FK4t=2109sab_channel=NicholasRenotte.

Painting by Numbers with the Assistance of the Dobot Magician

Applied Robotics 2023/24 Hochschule Hof, University of Applied Sciences

1st Patrick Schröder
UAS Hof

2nd Gleb Hubarevich
UAS Hof

3rd Denys Zarishniuk
UAS Hof

4th Marina Waller
UAS Hof

***Abstract*—Robotic arms such as the Dobot Magician offer countless project possibilities as they can grip, lift and position objects as well as create drawings precisely. Unfortunately, it can be challenging to realize a project with the Dobot Magician due to the lack of documentation on how to operate and program the robotic arm. To bridge this gap, we aimed to develop an application that enables the robot to automatically connect dots in the correct order as well as color specific areas on a drawing. For this, we used computer vision and machine learning to detect and recognize objects with a digital camera. Additionally, we utilized the Dobot Magician robotic arm to perform the movements and attached a pen to empower the robot to draw. With this, we were able to automatically connect dots in the correct order on sample drawings we created. However, the task of coloring areas was only partially successful, so that only simple drawings could be automatically colored.**

I. Introduction and state of the art

To introduce students and interested people to the field of robotics, performing tasks using a robotic arm, particularly the Dobot Magician, is an excellent approach. As in [1] described, the Dobot Magician robotic arm has a wide range of functionalities such as drawing, painting, 3D printing, etc. It serves as an educational tool with hands-on-experience that can be used to realize countless projects. Despite its potential, it can be challenging to realize a project due to the lack of documentation on how to operate and program the robotic arm. The Dobot Magician relies heavily on its software for control. For Python programming, users can employ a library named pydobot. Nevertheless, this designated library offers only limited control options and lacks comprehensive documentation. This limitation motivated us to implement this project to address these challenges.

The goals we aim to achieve with this project involve on the one hand automatically connecting dots in the correct order and on the other hand coloring specific areas of a drawing using the Dobot Magician. To achieve our goals, we used existing technologies. First, we integrated YOLO (You Only Look Once) [2], which is an object detection algorithm, to identify and localize objects in an image. In our case, we used it to detect numbers and dots for the first goal and shapes with numbers for the second. In addition, an OCR (Optical Character Recognition) engine named Tesseract [3] helped us to recognize each number. To integrate these technologies with the Dobot Magician in a Python environment, we used the pydobot library [4] to control the robot. By combining these technologies, we were able to fully automate the process of connecting dots in the correct order using the Dobot Magician. However, the implementation of the second goal to color certain areas of a drawing was only partially successful.

In the remainder of this paper, we present the design of our implementation for both goals (Section II). Then we describe the experiments and discuss their results (Section III). Finally, we conclude and discuss future work (Section IV).

II. Approach

A. Hardware and Software Setup

Before we implemented the actual application, we needed to figure out what hardware and software we could use and how to set them up for each use case to get accurate results. After some research and experimentation, we decided to use a digital camera to capture images, specifically the Olympus OM-D E-M10 III, instead of a webcam because of the image quality. An additional benefit of the DSLM camera was that we got the ability to directly control the brightness, focus and focal length of the image. To connect the camera to a laptop we used a mini-HDMI cable and a video capture card, so the camera works as a regular USB webcam. To automatically draw on a paper we used the Dobot Magician as the robotic arm.

To ensure consistency throughout this project, we constructed a scaffold that sets the Dobot Magician, the digital camera and the printed-out image in a fixed location (see Figure 1). The simplest and cheapest solution was to build it out of wood scraps and plywood. The base has a slot for the Dobot that helps to precisely position it in place. The platform for the drawings is made of plywood and gives a flat surface for the robot to draw. Small dots on the platform define the drawing area and help to adjust the camera's zoom. The drawing can be fixed with magnets that can be attached to the screw heads on the platform. To create a stable and adjustable mount for the camera, we used two 50cm 8mm stainless steel threaded bars, metal discs and nuts. This allows us to adjust the height of the camera, so it doesn't collide with the Dobot's arm and captures the photos with as little disturbance as possible.

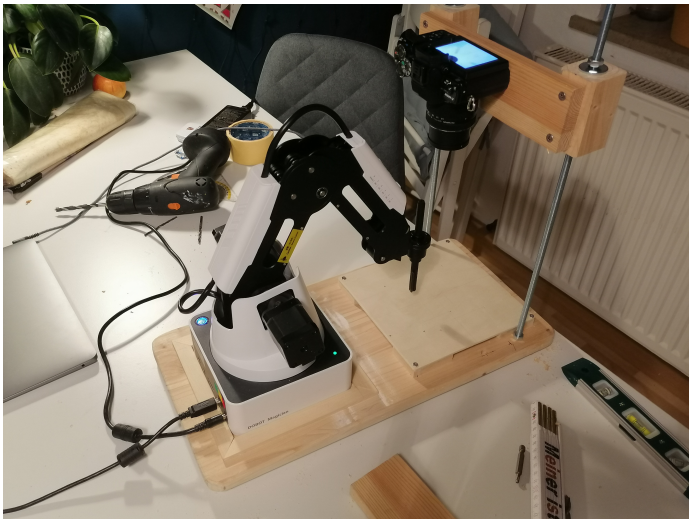


Fig. 1. Arrangement of the Dobot Magician and the digital camera on the scaffold

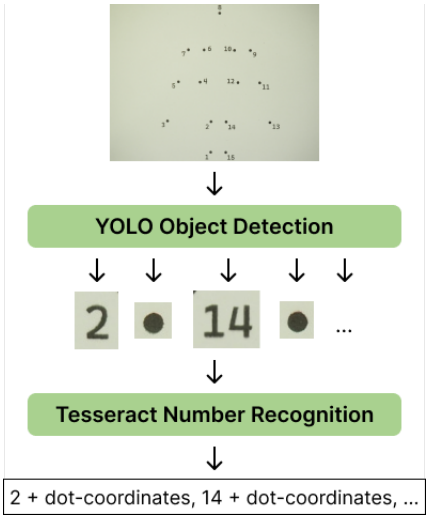


Fig. 2. Process of Object Detection and Recognition

In terms of software, we implemented our application in a Python environment. By using the OpenCV library [5] we were able to access the video feed from our camera and capture an image of it, which we saved and then used as a data source. Technologies such as YOLO and Tesseract were used to process the saved image. These tools made it possible to detect and recognize the relevant data for further procedures. For a better user experience, we implemented a simple graphical user interface (GUI) using the library customtkinter [6]. This allows the user to pause the application to review the results and return the robot to its home position with the DobotStudio software. To control the Dobot Magician in our Python environment, we used the pydobot library. This generally forms the basis for our project regarding both hardware and software.

B. Object Detection and Recognition

In order to obtain the relevant data for the first goal, we must detect and recognize each number and dot in the captured image. With this data, the Dobot Magician knows where to move and draw.

This can be accomplished using a YOLO object detection model (see Figure 2). To get the model to detect the numbers and dots in the captured image, we trained it with a dataset we made ourselves. We created a few sample images with dots and their corresponding numbers using the software Figma. These images were also augmented to expand our dataset to a total of 150 images to increase the model's accuracy. The dataset itself was created using Roboflow [7]. Each image within the dataset was annotated by marking the position of each number and dot. To ensure that the model can make accurate predictions on new, unseen images, the dataset was split into 70% training, 20% validation and 10% testing. Finally, we proceeded to train the YOLO model using a Google Colab notebook to guarantee independence from the performance of our local computers. The training was completed over 100 epochs and achieved good enough results for our project. With our trained YOLO model, we can now localize and crop each number and dot in our image to use them in further procedures.

After the numbers and dots were detected, we needed to recognize each number to sort the coordinates of the dots obtained by the localization of the YOLO model (see Figure 2). To achieve this, we used the Tesseract OCR engine. The cropped numbers we got from the object detection are now being inputted into the OCR engine. With the help of OpenCV, we processed the images to enhance the readability of the numbers. This way every number got recognized. To ensure that every dot is grouped with the correct number, the distance between the detections of the dots and numbers was calculated. This was done with the following formula:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Each dot was grouped with the number that had the shortest distance to it. At last, the data we obtained, containing each number with the coordinates of the corresponding dot, was sorted so it was in the correct order. On this basis, the data can be used to control the robot so that it moves sequentially to the coordinates of the dots.

C. Object Segmentation

To achieve our second goal of automatically coloring specific areas of a drawing, we needed to use a different technology than used in the first case of connecting dots. In this case, our model should not detect numbers and dots, but shapes containing a number. Object detection cannot obtain the precise boundaries of these shapes. This is why we used object segmentation instead of object detection.

This can be accomplished using a YOLO segmentation model. We started creating ten sample images using the Software Figma. To create the dataset, we annotated the images with Roboflow by marking each shape that contains a number. After that, we started to train the YOLO model for 100 epochs,

however, the results were inaccurate due to our limited dataset. Since it successfully detected all shapes in simple images, we decided to continue using this model because of time pressure.

After integrating the model into our application, it generated a mask for each detected shape in the captured image. As this data cannot be used to begin coloring the numbered shapes, we had to convert the masks into polygons. This means that the boundaries of the detected shapes were identified and expressed as a series of connected vertices, forming a polygon. Using the polygon points, i.e. the corner points of the shapes, we were now able to use the data to color the individual areas.

D. Dobot Magician Control

Now that we have gathered the relevant data for both use cases, we have begun programming the robot. To be able to control the Dobot Magician in our Python environment, we used the pydobot library. As this library is limited and not well documented, we also had to use the DobotStudio software. Without the software, it is not possible to control the robot's joints, which means that homing, a task that returns the robot to its starting position, is not possible. This task is necessary to continue with the further procedure. Once the robotic arm is in its homing position, it can now move to the coordinates of the dots in the given data. However, the coordinates from the detection process do not fit in the coordinate system of the Dobot Magician. For this reason, we converted the coordinates into a system that matches that of the robot.

The printed-out image was in a 1:1 aspect ratio requiring us to crop the width of the captured image since the digital camera had a 16:9 aspect ratio. That is why the left-hand side of the width of every coordinate was subtracted to achieve the desired aspect ratio. It is important to note that the zero point of the picture is located at the bottom left. For this reason, only the left-hand side needs to be subtracted. Following this, the coordinates were converted to percentages based on the image resolution to make them compatible with the coordinate system of the Dobot Magician. According to this data, the robot should draw the coordinates directly under it. However, as this is not possible, we had to add an offset. To fix this issue, we simply added the offset to the coordinates which was found through a trial-and-error process. Lastly, we noticed that the x and y coordinates of our data and the Dobot Magician were swapped so we switched their positions accordingly. Based on these calculations, the converted coordinates were now ready to be used by the robotic arm to draw.

In the first use case, the robot simply has to draw lines from one point to another. With the help of the pydobot library, the robotic arm can move to a specific coordinate. To connect the detected dots, the Dobot Magician moves downward along the z-axis at the first point, pressing the built-in pen onto the paper. After that, the robot moves to the second point and as it completes drawing the line, it moves upward along the z-axis and lifts the pen off the paper. This process is repeated until it finishes connecting all the dots. By inputting our previously sorted data the robot should accurately connect the detected dots in the correct order.

In the second use case, the Dobot Magician must colorize areas on an image. This can be done with the scanline fill algorithm, which works with lines and polygon edges. However, we decided to write a simpler algorithm that would work for our example and other simple shapes like circles, triangles or rectangles. This algorithm for area filling involves two key steps. First, we determined the key coordinates by calculating the topmost y-coordinate and identifying the farthest and the nearest x-coordinate at this topmost y-level. Next, lines should be drawn from the farthest to the nearest x-coordinate at each y-level, starting with the topmost y-coordinate and going downwards. These actions systematically fill the area of the detected shape. By inputting the previously converted data the robot should color all detected shapes.

With this, the Dobot magician automatically connected dots in the correct order (see Figure 3) and filled specific areas on a drawing.

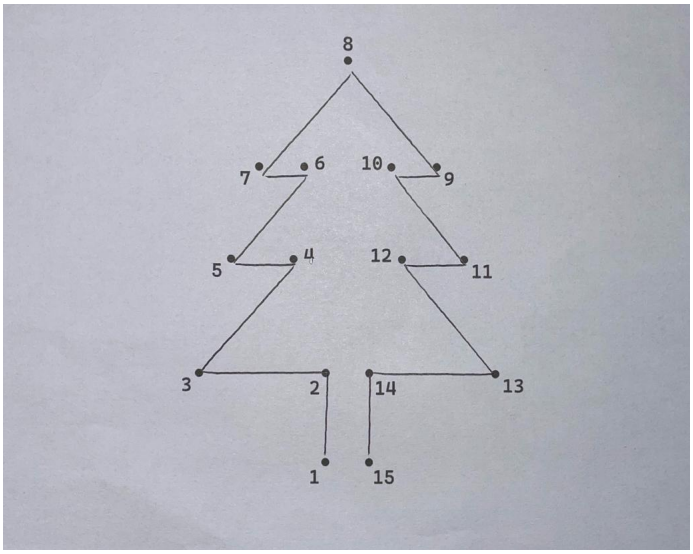


Fig. 3. Final result of the first use case to connect dots in the correct order

III. Experiments

During the development of our project, we conducted several experiments. These experiments served to test the current approach and enhance the application afterward.

1) Experiment: At a very early stage of the project, we tested if it was possible to recognize numbers and dots by using only the Tesseract OCR engine. We first tried to recognize numbers in a digital image. To visualize the output, we used OpenCV to draw boxes around the numbers and display the recognition underneath (see Figure 4).

Since this turned out accurate, we proceeded to set up our hardware and software and captured an example image. After trying to detect the numbers and dots in the image captured by the camera, the results were inaccurate (see Figure 5). With this approach, we could not advance further with the project. As a result, we had to optimize the accuracy of the detections.

We soon discovered that we could combine an object detection algorithm with optical character recognition.

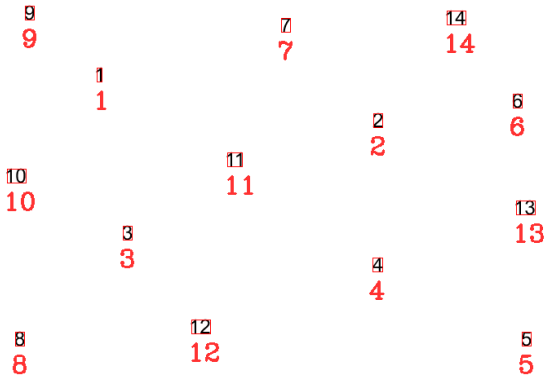


Fig. 4. Result of the detection and recognition of numbers on a digital image using Tesseract OCR engine

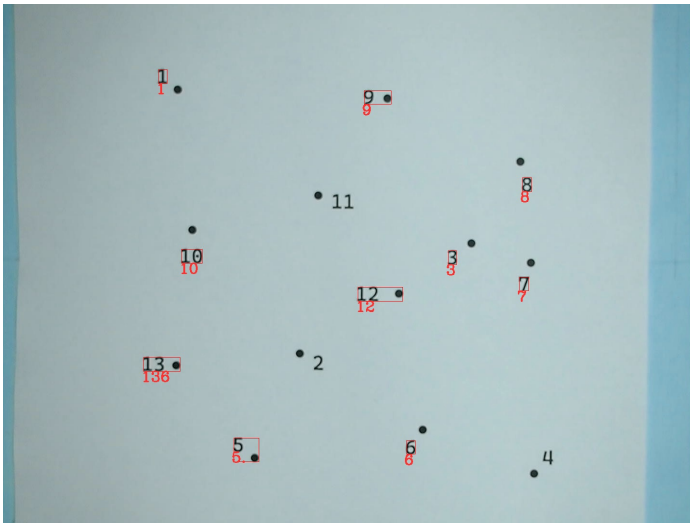


Fig. 5. Result of the detection and recognition of numbers and dots on a captured image using Tesseract OCR engine

2) Experiment: The second approach to detect and recognize dots and numbers involves separating the steps of object detection and recognition. For this reason, we proceeded to train a YOLO object detection model to increase the accuracy. To accomplish this, we created a dataset of sample images that we can use to train the model. The images were labeled, combining each dot and their corresponding number into one single object. In this procedure, the dot and the corresponding number are already grouped together and no longer need to be grouped separately. Next, we integrated the model into our Python environment to test it. We visualized the detections using the supervision library [8]. The detections of the grouped objects provided accurate results and correctly identified each pair on the captured image (see Figure 6). Each dot with its corresponding number was saved as a separate cropped image so that it could be used to recognize the numbers and dots.

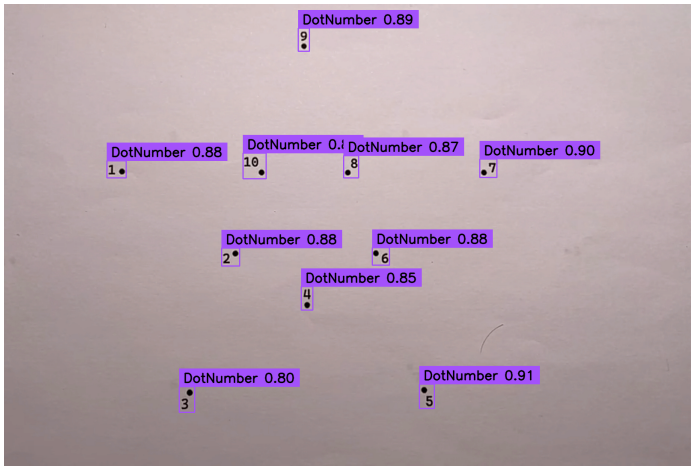


Fig. 6. Result of the object detection of numbers and dots as a group on a captured image using YOLO model

Since the object detection worked accurately, we proceeded to recognize the numbers and dots with the Tesseract OCR engine. For debugging purposes, we displayed each cropped number with the recognitions of Tesseract with OpenCV. In this analysis we observed that the recognitions were not accurate (see Figure 7). Numbers were misidentified and dots and numbers were grouped incorrectly, making it impossible to determine the exact coordinates of the dots. Therefore, we tried to train the Tesseract OCR engine on the custom font that we used in the sample images we created. However, this did not improve the accuracy of the recognition. As a result, we had to improve the accuracy of the recognitions by using a different approach.

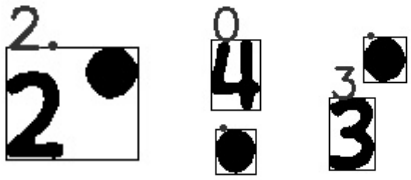


Fig. 7. Results of the recognitions of the cropped images with Tesseract OCR engine

3) Experiment: The final approach of the first use case involved treating numbers and dots as different classes within the YOLO model. To achieve this, we retrained it with images taken directly from the digital camera. We annotated the images using Roboflow by marking the dots and numbers separately. After we trained the model, we tested its performance by drawing the detections on the digital image using supervision (see Figure 8). This time, each dot and number were successfully detected.

Before we used the newly trained model, there was no need to group the numbers and dots, since they were detected as

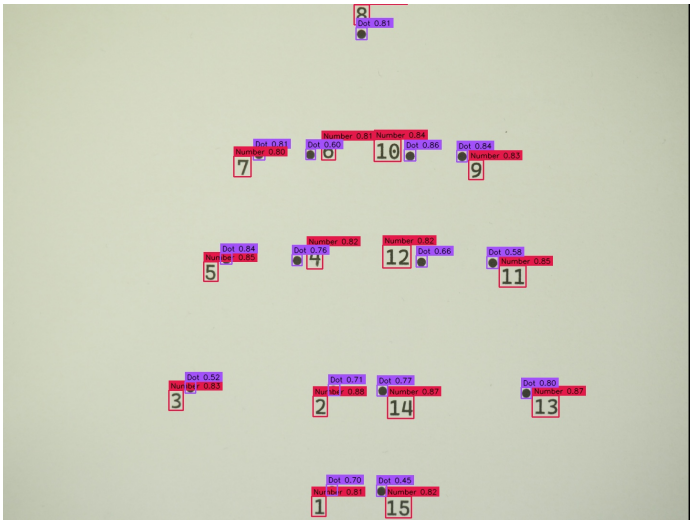


Fig. 8. Results of the detections of numbers and dots separately on a captured image using YOLO model

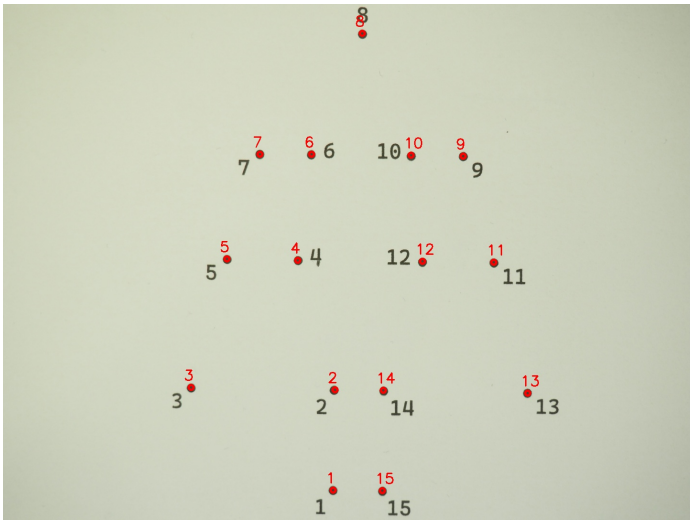


Fig. 9. Results of the located dots with their order on a captured image

one single object. As the YOLO model no longer performs this task, we also had to group the dots with their corresponding number. For this purpose, the distance between the numbers and the dots had to be calculated. After that, the dots were grouped with the number that had the shortest distance to them.

Following the grouping of each dot with its corresponding number, we proceeded to recognize the numbers with the Tesseract OCR engine. This time Tesseract only had to recognize the numbers and not locate the dots. With the help of image processing, it was possible to recognize all numbers on the captured image with Tesseract.

This approach proved good enough, allowing us to continue with the process of drawing using the Dobot Magician. We combined the functionality of the detection with the control of the robot. Before the robot used the data, we reviewed the results of the detections and recognitions by visualizing the dots and numbers on the digital image with OpenCV (see Figure 9). Once this was successful, the robot began drawing. During the process, we observed that the robotic arm was not precisely connecting the dots. The reason for that was an incorrect offset. We adjusted the offset through a trial-and-error process to enable the robot to connect the dots as accurately as possible. As soon as the offset was correct, the robot drew the drawing perfectly and connected all the dots in the correct order.

4) Experiment: Our approach to achieving the second goal of filling areas of an image involves detecting the shapes containing a number in the captured image. To detect the shapes, we created ten sample images and labeled them by marking the boundaries of the shapes. Afterward, we generated the dataset with Roboflow and trained a YOLO model. We set up the hardware and software to test the model. Following, we integrated the trained model into our Python environment and tested the results by visualizing the shapes on the digital image using supervision. There we recognized that the results were inaccurate due to the limited amount of data. Since it successfully recognized all shapes in simple images (see Figure 10), we decided to still use this model.

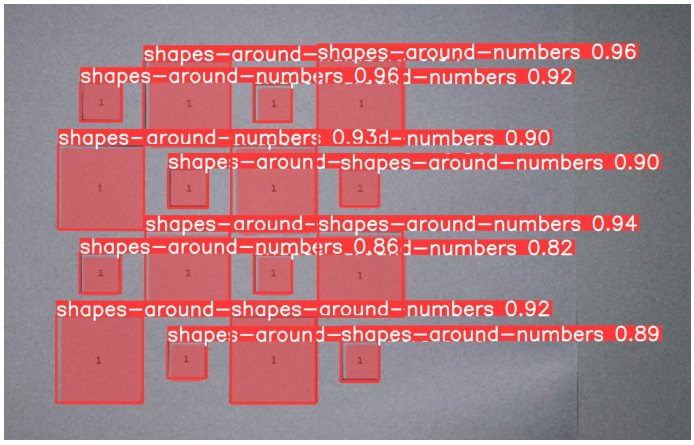


Fig. 10. Results of the object segmentation of shapes containing a number on a captured image using YOLO model

We proceeded with the task of coloring areas with the Dobot Magician. After the drawing task was performed using the detections from the YOLO model, the results were close to the shapes from the captured image. However, the robot took around 15 minutes to draw them because of redundant points (see Figure 11). This also led to the robotic arm being shut down as it took too long to draw. As a result, we tried to fix this problem by implementing a function that checks the distance between the points, identifying and eliminating redundant points that were too close to each other. This approach resulted in the robot drawing completely different shapes. Consequently, the model must be retrained to improve accuracy. This can be achieved by creating additional sample images to train the model.

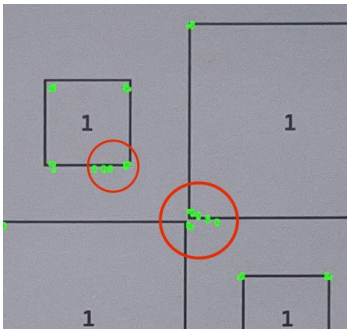


Fig. 11. Redundant points of the detected shapes

IV. Conclusion and future work

Although we had limited guidance from the Dobot Magician documentation, we managed to start this project and achieve significant successes. During our project, we learned a lot about the Dobot Magician's functionality and how to operate it. Our team developed an application that automated the "paint-by-number" experience. We integrated computer vision and machine learning, specifically a YOLO object detection model and a Tesseract OCR engine, that allow the robot to recognize and localize numbers and dots in an image and connect them. When it came to coloring specific areas in an image using the YOLO object segmentation model the application did not meet our expectations.

The experiments show the challenges that arose during the implementation of the application. We had to change our approach several times to achieve good results. When realizing a project, many problems will occur without comprehensive documentation. However, the experiments also show that there is still room for improvement for future work.

With our documentation of the approach and the experiments we aim to provide future projects with a better starting point. Looking ahead, there are a few areas that can be improved. First, the detections of our YOLO models could be enhanced by increasing the amount of training data to retrain them with a larger dataset. The recognition of the numbers could be improved by either switching the OCR engine or fine-tuning Tesseract on the characters. In addition, the coloring process could be adjusted. This can be done by choosing a more complex algorithm such as the "scanline fill algorithm" for more accurate and detailed results. To add more complexity, functionality could be implemented so that the robot can use different colors and switch them autonomously.

In summary, while we achieved our first successes, the outlined future work can address limitations and unlock the full potential of the automated Painting by Numbers with the Assistance of the Dobot Magician.

References

[1] Dobot magician - desktop robot. Accessed: February 13, 2024. [Online]. Available: https://ifdesign.com/en/winner-ranking/project/dobot-magician/236534

[2] Ultralytics yolov8 docs. Accessed: February 13, 2024. [Online]. Available: https://docs.ultralytics.com/de

[3] Tesseract documentation. Accessed: February 13, 2024. [Online]. Available: https://tesseract-ocr.github.io/

[4] Python library for dobot magician. Accessed: February 13, 2024. [Online]. Available: https://github.com/luismesas/pydobot

[5] Opencv - open computer vision library. Accessed: February 13, 2024. [Online]. Available: https://opencv.org/

[6] Customtkinter. Accessed: February 13, 2024. [Online]. Available: https://github.com/TomSchimansky/CustomTkinter

[7] Roboflow: Give your software the power to see objects in images and videos. Accessed: February 13, 2024. [Online]. Available: https://roboflow.com/

[8] Supvervision. Accessed: February 13, 2024. [Online]. Available: https://supervision.roboflow.com/

General Purpose Optics-Based Collaborative Robotic Followers

Andreas Schmidt
Department of Computer Science
University of Applied Sciences
Hof, Germany
andreas.schmidt@hof-university.de

Boris Tolgurov
Department of Computer Science
University of Applied Sciences
Hof, Germany
boris.tolgurov@hof-university.de

Given Daniel Reul
Department of Computer Science
University of Applied Sciences
Hof, Germany
daniel.reul@hof-university.de

Given Jael Bernice-Walter
Department of Computer Science
University of Applied Sciences
Hof, Germany
jael-bernice.walter@hof-university.de

***Abstract*—In this paper, we present an optics-only approach for robots to collaborate with and follow human operators. The system is designed in a modular way. As such, it can be adapted to any robot. We leverage YOLOv8 object detection in conjunction with BoT-SORT tracking and deterministic decision-making for optimal tracking accuracy. Tests are run on a common NAOv6 robot, which is controlled by a custom algorithm based on tracking data.**

***Index Terms*—robotics, YOLO, object detection, people tracking, robot companion**

I. Introduction

For a long time robots were mainly used in industry, but there is also a rising demand for social robots that assist or just befriend humans in their everyday life. Such companion robots (further abbreviated as cobots) can be used for entertainment purposes or as child toys, but they also have more serious applications. They can be used in elderly care or for therapeutic purposes. Cobots already play a significant role in modernizing and securing hazardous working environments and are used to assist workers in various business applications. Considering the demographic changes, the lack of staff in elderly care or the rising loneliness in western countries, the demand for cobots can be expected to grow in the next few years.

Building a robot that can follow and interact with a human in an unknown environment in real time has always been a difficult task, especially the persistent tracking of a specific person. Previously, a large number of sensors would have been required to achieve this goal. However, in recent years, great advances in the field of real time image detection were made. We hypothesize that due to these improvements, it is possible to build such a tracking system by only using optical image data.

To show this, we implement a system able to follow a human by using a state of the art YOLO image detection model. For testing, the robot "NAOv6" is used. Notably, the core of the detection system is designed to work on any platform. Tracking and position calculations are executed in a separate process. This makes this system highly adaptable to other situations and robots. Positional data is transferred via network. Because of our optics-only approach, no mapping techniques like SLAM are needed. Persons can be tracked by correlating subsequent detections with previous tracking information. The system is designed to work even in situations where the target is in a crowd of people or obstructed by an obstacle. It is also able to process events and react in an easy, human-understandable way through speech.

II. Approach

A. System Architecture

We use the NAOqi framework to control the robot. It provides all robot functions using a proxy mechanism via network as described in [1]. Due to the limited computational power of the in-built computer on the Nao robot itself, the decision was made to process all data remotely. The calculated motions are then transmitted back to the robot in order to control its actuators.

The system consists of two main modules. The robot control module is responsible for calculating the angles for various actuators, as well as for processing and interpreting events sent by the tracking module, converting them into robot commands if required. The tracking module handles the image detection and the tracking. Both modules run independently of each other. This makes both modules, most importantly the tracking module, reusable for other, similar applications.

The separation of both modules aids versatility. As already mentioned, the two-process approach decouples the tracking and generation of positional data from the robot in use. Any robot using any programming language can be integrated into this system merely by implementing the protocol used to transfer the positional and image data. In our concrete example, this separation was used to achieve interoperability with python2 and python3. Current versions of packages that are crucial for this project, like OpenCV, no longer support

python2. Most modern and probably future python image detection models are not expected to run with python2. As the NAOqi API is based on python2, we are coupled to this version on the controlling side of the robot.

The main loop of our application is as follows: The robot control module accesses the current image of the upper camera and sends it to the tracking module. The tracking module runs a YOLOv8-based object detection algorithm on that image and applies its tracking logic to the result. Thereafter, either the box with the new location of the target or an empty result is transmitted to the robot control module. Then the control module adjusts the cobot's movements, updates its inner state and sends the next frame. It should be noted that new frames are gathered from the robot while waiting for the tracking process to complete. This increases the speed with which images can be processed.

The protocol used for communication between the modules is based on fixed packet sizes. Images, as well as detection results are assumed to be of a fixed size. As such, there is no need for additional delimiters between transmitted packets. This makes communication relatively simple and less error-prone.

B. Robot Control Module

The `nao-follower.py` file provides both the routine for the Cobot through the `run` function of the `NaoFollower` class and the main function to start the robot control module. An interface for the tracking module is provided by the `YOLOClient` of the `interop2.py` file. The interfaces to the NAOqi library are provided in `nao.py`. In order to inform the user about the cobot's current state and state changes, `EventService` in `evt.py` was implemented. In this case the user is notified about the cobot's current state through speech and LEDs around the robot's eyes.

1) NaoFollower: The NaoFollower is initiated and run by the main method. Its `run` function contains the Cobot routine. It initializes the interfaces of `nao.py` and `interop2.py`. The Cobot is configured to have autonomous life deactivated [2] and to stand up. Using the `YOLOClient` interface, it sends the camera input and retrieves the results of the tracking module. The result is provided as a `YOLOResult` instance and contains a `transpose` function to map its coordinates to an origin in the middle of the picture for further use. Depending on the validity of the result, `NaoFollower` either adjusts the Cobot's head and movement using the `NaoMotionService`, or it stops the robot and sends a target lost event to the `EventService`. Furthermore, exceptions and keyboard interrupts are caught, causing both the Cobot and the routine to stop.

2) YOLOClient and YOLOResult: `YOLOClient` in `interop2.py` is an interface that sends an image to the tracking module and receives the results. This data is then packed and converted into a `YOLOResult` object. The `YOLOResult` class is located in the `common.py` file and provides the identified bounding box of the target through the x and y coordinates of its center, as well as the height and width of the box. Additionally, it offers a function to transpose coordinates, aligning them with a coordinate system originating in the center of the image for simplified calculations later on. This is done by subtracting half of the respective screen dimension from the position:

$$t_d = p_d - \frac{f_d}{2} \qquad \forall d \in \{x, y\} \tag{1}$$

Where:
$d =$ the respective dimension
$f =$ the frame coordinate
$p =$ the coordinate retrieved from the tracking module
$t =$ the transposed coordinate

3) NaoPictureClient: The `NaoPictureClient` class in `nao.py` provides an interface for the ALVideoDeviceService API [3] of NAOqi. It implements a function to retrieve the camera image.

4) NaoMotionService: The `NaoMotionService` class in `nao.py` serves as an interface for the ALMotion API [2], calculating movement directions based on the positional data of the `YOLOResult`.

The data of the upper camera angles [4] and the maximum angles of the two head joints [5] are used.

During initialization, the NAO's native external collision protection is enabled. This ensures that our Cobot automatically stops when it detects an obstacle.

The `head_adjust` function provides calculations for turning the head towards the target. By employing this method, the Cobot can track the target autonomously, irrespective of its own velocity. This is done by minimizing the horizontal deviation of the subject's position to the middle of the frame. Additionally, the top y-position of the bounding box is kept within the upper half of the picture. If the y-position is equal to the hight of the frame, the head is moved downwards to ensure the subject stays in view.

The `calculate_speed` function adjusts the Cobot's forward movement speed. As we expect the bounding box to grow as the Cobot approaches the target, the size of the box is used to regulate the speed. This is done by calculating the area of the bounding box with respect to the area of the frame itself.

The `move` function defines the movement configurations of the Cobot. For the forward movement, the calculated speed is used. For rotational speed, the calculated speed is multiplied with the fraction of the current yaw angle with respect to its maximal possible rotation so that the torso aims to adjust to the head orientation. Additionally, the `EventService` is notified of the action.

5) EventService: The `EventService` is located in `evt.py`. It collects information on the Cobots actions in the form of events. Through that, it manages the robots state and its transitions. Classes can subscribe to this service if they provide functions to call on certain events. Before notifying the subscribed classes, it filters which events to actually hand over.

This provides an easy way to add new states and events and reactions in subscribed classes.

6) NaoSpeechService: The `NaoSpeechService` class in `nao.py` is an interface for the ALTextToSpeech API [6]. It uses a dictionary which provides various sentences for an event. If the `EventService` hands over an event through the corresponding function, the `say` function picks a random sentence of the associated category in the dictionary. To prevent an accumulation of speech calls to the Cobot, the function checks how much time has passed since the last call to `say`. The choice of sentences are further enhanced with modes to differentiate between characters.

7) NaoLedService: The `NaoLedService` class in `nao.py` is an interface for the ALLeds API [7]. As the speech service is constrained by time and the need for clear sentences, the eye LEDs are designed to provide quicker and more easily interpretable feedback on the current state.

8) PictureClient: The `PictureClient` is an interface for receiving pictures from a `YOLOClient` instance and sending `YOLOResult` tracking information. It is used in the system's YOLO tracking module and provides easy access to the single frames sent by the cobot. It also allows for automatic reshaping of the array to a BRG-coded picture matrix.

The `YOLOResults` are packed using the python "struct" package and subsequently transmitted.

C. Tracking Module

Modern object detection systems like YOLO are able to detect people in various lighting conditions, spaces and postures. One major challenge is tracking these people reliably from frame to frame. In order to establish reliable Multi-Object Tracking (MOT), multiple pre-existing algorithms are evaluated. Specifically, MOT Accuracy (MOTA), Higher Order Tracking Accuracy (HOTA) and IDF1 metrics are evaluated to find the best possible candidate.

The MOTA score

> [...] accounts for all object configuration errors made by the tracker, false positives, misses, mismatches, over all frames. It is similar to metrics widely used in other domains (such as the Word Error Rate (W ER), commonly used in speech recognition) and gives a very intuitive measure of the tracker's performance at keeping accurate trajectories, independent of its precision in estimating object positions [8].

It enables estimations regarding the consistency of the tracking generated by the aforementioned tracking algorithms. As stated in [9], MOTA is slightly biased towards measuring detection accuracy rather than association accuracy. As such, another metric is needed for detecting errors on associations.

In order to better understand how different tracking methods will contribute to a more stable tracking, we use the IDF1 score. Reference [10] describes the IDF1 score as "[...] the ratio of correctly identified detections over the average number of ground-truth and computed detections". As already described by [9], the IDF1 metric is biased towards measuring association accuracy, rather than detection accuracy. This means that this score, in combination with the MOTA score mentioned above, gives us a relatively clear picture which tracking algorithm performs better, and in which way. According to [9], many scientific papers measuring tracker performance have often only adopted MOTA scores "[...] because detection is such an important part of tracking evaluation, and IDF1 isn't able to adequately measure it". IDF1 excels at measuring "identification". In this context, IDF1 identification, as explained in [9], tries to measure and evaluate the trajectories of objects tracked by the MOT tracker.

Finally, we are using the HOTA score to give us a better idea of the general performance of the tracker. "The HOTA is a relatively new metric, as per the description provided by [9], which states that it "[...] balances the effect of performing accurate detection, association, and localization into a single unified metric for comparing trackers". Additionally, "HOTA also incorporates measuring the localisation accuracy of tracking results which isn't present in either MOTA or IDF1" [9]. The localization aspect is especially interesting to us. Reference [9] defines localization errors as "[...] errors [which] occur when prDets are not perfectly spatially aligned with gtDets". In other words, a single localization is erroneous when a detection in the set of ground-truth detections (gtDets), or the set of actual detections in a given frame, does not spatially align with its matching predicted detection (prDet) location. This means, in addition to balancing IDF1 and MOTA biases, HOTA allows one to make assumptions about how well a given tracker will be able to correctly predict the position of a given identified object in the next expected frame.

In [11] and [12], it can be seen that SMILETrack, as well as SparseTrack, consistently achieved the highest scores in MOTA metrics. This means they are well suited for tracking scenarios requiring high detection accuracy. UCMCTrack and Deep-OC-SORT achieve the highest HOTA score in MOT17 and MOT20 evaluations, respectively. This indicates that both are good choices in tracking scenarios in which a good general performance across all metrics is favorable. Lastly, UCMC-Track and Deep-OC-SORT achieve the best IDF1 scores in MOT17 and MOT20 dataset evaluations.

For our experiments, we conclude that the use of SMILE-Track, SparseTrack or UCMCTrack would be best suited for building a tracking algorithm.

As of the time of writing this paper, no easily usable implementations of any aforementioned algorithms exist. Prototypes using BoxMot [13] yielded inconclusive results. Due to time constraints, we had to resort to trackers provided by the YOLO implementation "ultralytics" [14], which provides out-of-the-box tracking using either ByteTrack or BoT-SORT. Neither of both options ranked first in any of the aforementioned tracking metrics. However, BoT-SORT seems to consistently show relatively high scores in both MOT17 and MOT20 evaluations. Especially the consistent results across both evaluations make it a strong second tracker choice for the purposes described in this research.

Preliminary tests reveal that the use of the Track-ID from the aforementioned BoT-SORT module exhibits inaccuracies. In

particular, we are seeing the track identifier (TrackID) switch once tracking is lost for a brief moment. Due to this, an algorithm which is able to stabilize tracking in case of TrackID switches is implemented. This algorithm consists of multiple steps. A `YOLOResult` can be returned in any step.

The first three steps ensure a near instantaneous result if the detection result permits it. If no detections are found, an invalid result is returned. If exactly one person is detected, the position of this person is returned and tracked in subsequent frames. In case multiple people are detected and no previous track-point exists, the person closest to the middle of the image is tracked. All distances in each preceding and subsequent step of the algorithm are based on the L2-Norm.

The most important step in the algorithm is the association of previously tracked detections to new detections with a different TrackID and nearly equal position. In order to achieve this, a record of a configurable amount of previously tracked detections (History), in our case 40 of them, is kept. The Most Significant Detection (MSD) is the one whose TrackID appears most frequently in this list. All detections for the current frame are iterated and the best one is selected. We define a penalty for the current detection by calculating the detection's L2-Norm distance to the average position of the MSD. Positions for all previous detections are calculated using a weighted average, grouped by TrackID.

$$w(i,l) = \begin{cases} 100 & \text{if } i = 1 \\ (2 - \frac{i}{l})^6 & \text{otherwise} \end{cases} \tag{2}$$

$$W = \sum_{i=1}^{l} w(i,l) \tag{3}$$

$$p' = \frac{\sum_{i=1}^{l}(p_i * w(i,l))}{W} \qquad \forall p_i \in \{x_i, y_i\} \tag{4}$$

Where:
W = the combined weight of all points
l = the amount of detections for any given TrackID
p_i = the position of the current detection for both x and y coordinates

Equation (4) shows that recent detections have more influence on the average position than earlier detections. Additionally, the latest detection is given the highest weight of 100. If the detection's TrackID is already known, an additional penalty is calculated. The penalty is the sum of the distance of the current detection to the average of the TrackID position and the jitter value, which is defined by Equation (5):

$$j = \left| \frac{\sum_{i=2}^{l}(x_{i-1} - x_i)}{l - 1} \right| \tag{5}$$

Where:
l = the amount of detections for any given TrackID
x = the x coordinate

The jitter value j is the average of the change of x-coordinates between frames. A high jitter value indicates that the subject was either moving too quickly, the position changed rapidly or the tracking was unstable in general.

If the calculated penalty is higher than 200, the detection is not used due to the high uncertainty. In that case, if the previously tracked TrackID is found in the list of detections, this TrackID is used for the current frame.

As a last resort, the box with the lowest distance to the previously tracked box is used. Because this can lead to a high amount of TrackID switching in cases where multiple people are in view and close to the previous detection, an invalid `YOLOResult` is returned as long as the new tracking candidate was tracked in less than 10 previous frames. This leads to the robot stopping if it is unable to accurately track any given candidate due to multiple people being in view.

Tracking is done using the mechanisms defined in `yolo-client.py`. The main script uses a `PictureClient` to receive pictures from the Nao and send results to it. The function `find_best_option` implements the detection selection as described above. The `get_avg_position` function returns a map of average positions and jitter for all detections found in the detection history using the aforementioned calculation. The `get_most_sig_det` function returns the MSD based on the history. Lastly, the L2-Distance is calculated using the `dist` function, which is then used by `find_lowest_dist` to find the lowest distance to a given point for all detections supplied.

To aid human operators in understanding the choices made by the tracker, we added a visualization of the currently processed frames. Frames are annotated with symbols generated through tracking data. All detected people are identified by a gray bounding box. The tracked person itself is highlighted through a green bounding box. The distances of all people to the previously tracked positions are made visible by orange lines between the tracked point and the center of the corresponding detections. A light red circle shows the uncertainty threshold around the MSD. An orange circle indicates the uncertainty value around the currently tracked point. Finally, average positions and their corresponding jitter are indicated by dots with varying color, where blue dots mean a high jitter, green dots indicate a low jitter value. The TrackID and YOLO confidence values are shown as text in the center of the box as can be seen in Fig. 1.

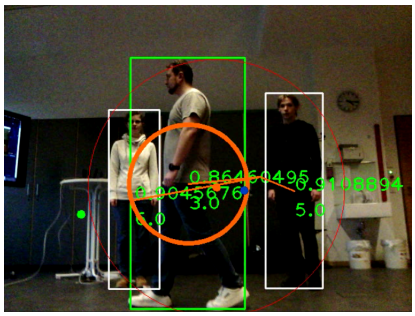


Fig. 1. An example of the tracking visualization.

III. Experiments

To test the implementation of the system described above, we used five distinct test cases:

A. Straight-line walking

The robot should be able to track the subject when simply walking in a straight line without obstructions. This test is the most basic situation. The robot was accurately able to follow the person based on the tracking module's positions. The instability induced by the robot's movements had no negative effects on the stability of the tracking. Because of the mechanism described in the "Tracking Module" subsection (II-C), the robot would have followed the person in view in any case. As such, this experiment only shows how well the tracker is able to handle the unstable camera images.

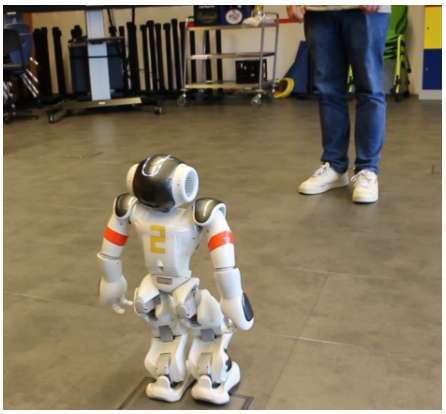


Fig. 2. The NAOv6 robot walks towards a target in a straight line.

B. Curve tracking

The robot calculates the turning radius based on the deviation of the head angle from its normal position. As such, a crucial test is whether the body actually follows the head and if the head is able to accurately follow the tracked person. The tracker needs to be able to track the person even if the background is moving and changing rapidly. In our case, this worked as expected.

Fig. 3. The NAOv6 robot tracks the subjects movement in curves and adjusts its course accordingly.

C. Tracking with multiple detections

In cases where the tracker detects multiple people, the currently tracked person needs to remain the MSD. This was tested by placing the subject in front of two other people and walking in front of them in a half-circle. The tracker is able to accurately track the subject. The robot correctly kept the person in view and adjusted its body and turning radius as expected.

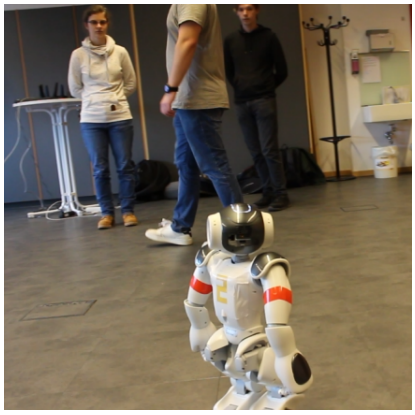


Fig. 4. The robot tracks the subject while multiple other people are in view.

D. Tracking and following with varying TrackIDs

The subject is placed in a crowded environment. The robot then needs to accurately predict the subject's path, even if the subject's TrackID changes. No abnormalities could be provoked during this test. The robot was able to correctly detect the TrackID switch and continued following the subject as desired.

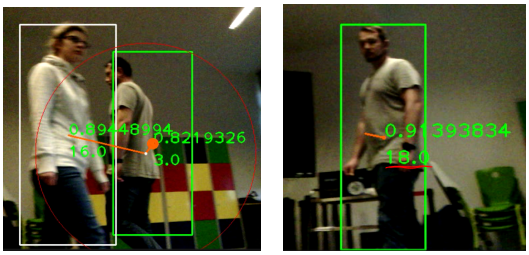


Fig. 5. The robot is able to maintain a stable tracking, even though the TrackID changes from 3 to 18.

E. Tracking and following in crowded environments

The subject was placed in a crowd of people. Due to the amount of people on our team, we were only able to test this scenario using three individual people. The algorithm shows deficits in trying to accurately follow the subject when it is overlapped by other detections. Since the algorithm is largely based on average positions, this problem was expected. We were unable to find a scenario in which this test case worked reliably. [1] An example of this problem happening can be seen in Fig. 6.

Fig. 6. The tracker shows deficits when the subject can not be reliably tracked in multiple subsequent frames while being overlapped by other people. The correct subject is TrackID 34.

IV. Conclusion

In summary, the findings in the course of this work support our initial hypothesis. A system able to detect and reliably follow human operators was implemented and successfully tested on a NAOv6 robot. The use of a pre-trained YOLOv8 object detection model significantly sped up the development process. Using data generated by the model, we are able to differentiate between object classes and effectively filter out unwanted object types. In order to reliably track subjects from frame to frame, we used the tracking algorithm "BoT-SORT". BoT-SORT is able to correctly identify the subject of a previous frame in most cases. However, in some specific cases, another tracking layer was needed. We implemented a deterministic decision-making algorithm, which is able to detect switches between TrackIDs. This is done by keeping a history of all previously detected subject positions and then calculating a weigthed average for this history. Using a mostly distance-based approach, we are able to correctly identify tracking errors and remediate them effectively.

In order to minimize the data processing duration, it was decided to transfer the computation to a remote machine. Motion calculation is done on this remote host, which then generates and sends commands to the robot. We combined the results of the tracking algorithm with the robot's motion service. One challenge was making the robot move in the target direction. As the NAO does not feature in-built camera-stabilization, the head movement speed and angle are adjusted to keep the target in focus while moving.

In-built sonar sensors are used for keeping distance between the NAO and obstacles. This ensures the robot will stop moving if it gets too close to any object. This safety system works even in circumstances where the remote controller gets disconnected due to latency or unforeseen errors. Additionally, movement speed decreases when the target's boundary box increases in size.

We also designed the `NaoSpeechService` to let the robot act more humanly. It can differentiate between situations and adapt its phrases.

The robot solved most of our test cases without any issues. However, we saw deficits when operating in crowded environments. The robot is unable to correctly identify the subject in cases where it was obstructed for more than a few frames at a time. Additional research should be done in order to investigate how an even more stable subject tracking could be achieved. Another major unsolved problem is maneuvering around hazards and obstacles. More work needs to be done on optics-only path planning for this system to be used in real-world environments.

References

[1] SoftBank Robotics Europe. "Key concepts." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/dev/naoqi/index.html%5C#naoqi-overview (visited on 02/09/2024).

[2] SoftBank Robotics Europe. "Almotion." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/naoqi/motion/almotion.html (visited on 02/09/2024).

[3] SoftBank Robotics Europe. "Alvideodevice." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/naoqi/vision/alvideodevice.html (visited on 02/09/2024).

[4] SoftBank Robotics Europe. "Nao- video camera." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-1/family/robots/video_robot.html (visited on 02/09/2024).

[5] SoftBank Robotics Europe. "Joints." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/family/nao_technical/joints_naov6.html (visited on 02/09/2024).

[6] SoftBank Robotics Europe. "Altexttospeech api." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/naoqi/audio/altexttospeech-api.html (visited on 02/09/2024).

[7] SoftBank Robotics Europe. "Alleds." (n.d.), [Online]. Available: http://doc.aldebaran.com/2-8/naoqi/sensors/alleds.html (visited on 02/09/2024).

[8] K. Bernardin, A. Elbs, and R. Stiefelhagen. "Multiple object tracking performance metrics and evaluation in a smart room environment." (n.d.), [Online]. Available: https://cvhci.anthropomatik.kit.edu/~stiefel/papers/ECCV2006WorkshopCameraReady.pdf (visited on 02/09/2024).

[9] J. Luiten, A. Ošep, P. Dendorfer, *et al.*, "Hota: A higher order metric for evaluating multi-object tracking," *International Journal of Computer Vision*, vol. 129, no. 2, pp. 548–578, Oct. 2020, ISSN: 1573-1405. DOI: 10.1007/s11263-020-01375-2. [Online]. Available: http://dx.doi.org/10.1007/s11263-020-01375-2.

[10] R. Henschel, T. von Marcard, and B. Rosenhahn, "Simultaneous identification and tracking of multiple people using video and imus," n.d. [Online]. Available: https://www.tnt.uni-hannover.de/papers/data/1359/02.pdf.

[11] PapersWithCode. "Multi-Object Tracking on MOT20." (n.d.), [Online]. Available: https://paperswithcode.com/sota/multi-object-tracking-on-mot20-1 (visited on 02/09/2024).

[12] PapersWithCode. "Multi-Object Tracking on MOT17." (n.d.), [Online]. Available: https://paperswithcode.com/sota/multi-object-tracking-on-mot17 (visited on 02/09/2024).

[13] M. Broström, *BoxMOT: pluggable SOTA tracking modules for object detection, segmentation and pose estimation models*, version 10.0.43, n.d. DOI: https://zenodo.org/record/7629840. [Online]. Available: https://github.com/mikel-brostrom/yolo_tracking.

[14] G. Jocher, A. Chaurasia, and J. Qiu, *Ultralytics YOLO*, version 8.0.0, Jan. 2023. [Online]. Available: https://github.com/ultralytics/ultralytics.

Documentation for RcpU (**R**obot that **c**an **p**lay **U**NO)

Lukas Drescher Aruzhan Turashbayeva Tomiris Ismaganbetova
Niklas Demel Mekhmetetka Turan Tim Wagner

February 9, 2024

Abstract

In an era of technological advancements, the impact of robots has significantly increased over the last decade. From robots that can mix you drinks to robots that can construct complex machines, they become important to a lot of industries. This project introduces an application in the realm of card games, specifically focusing on the Niryo Ned 2 [1] robot to autonomously play the popular card game UNO against a human in real time.

1 State of the art

Imagine you want to play UNO, but your friends and family are unavailable at the moment. Now let's consider a robot that allows you to enjoy UNO cards every day, even if you find yourself feeling completely alone. While you could purchase UNO as a computer game from Steam [2] and play it in single-player mode, we believe that's not the "real" UNO experience. The true joy lies in holding physical UNO cards in your hands and having someone to interact with you or even sharing a few curse words.

We built something for this purpose, because we got the project idea to build a robot that can play UNO cards with another human. In this way, a single person can engage in the game entirely on their own, without the need for others.

2 Approach

At first, we attempted to find a solution for detecting UNO cards. There are several approaches to achieve this, both with and without machine learning. Initially, we prioritized the non-machine learning approach to save time and ensure simplicity and clarity for the entire team. However, due to some challenges with our initial method, we used a neural network. We will focus on that later in this document.

2.1 Hardware

We requested some hardware in order to accomplish RcpU and its use cases. Our requirements included a robotic arm capable of grabbing cards and move them to another position, as well as a camera for detecting the robot's hand of cards. Professor Groth provided us with a Niryo Ned 2 robotic arm and a Logitech webcam. In addition to these components, we had to consider some additional requirements. Like what do we need in order to let the robot grab a card and what material should be used to improve the conditions of the environment? Which is why we brought some of our own materials - a black blanket to improve the card detection, a deck of UNO cards, and some rubber bands.

Why rubber bands, you may ask? Well, the robot's gripper has difficulties when attempting to grab a card due to a small gap when it's fully closed. To fix this, we added rubber bands to the gripper, allowing it to securely hold the cards.

2.2 UNO card detection with OpenCV

We searched for some approaches for detecting cards using OpenCV, and we discovered a solution that works with edge detection and color thresholds. UNO cards typically have distinct shapes and edges. We applied a contour detection algorithm to identify the outer boundaries of each card within the robot's field of view. To enhance contrast and simplify card recognition, we applied image thresholding techniques. This process allowed us to extract card features, resulting in more accurate subsequent processing. UNO cards feature specific patterns and symbols. For their recognition, we utilized template matching techniques to compare the extracted features with predefined templates, helping us in identifying card types and values.

2.3 Color detection with OpenCV

The robot's task is to place a card on the stack, ensuring it matches the previous card. Players have two options for finding a card that fits: either it should have the same color or a matching number. Let's consider the color first.

In the classic UNO game, there are four colors: Blue, Green, Red, and Yellow. While single pixels in images are typically represented in RGB (Red, Green, Blue), we found that using the HSV (Hue, Saturation, Value) color model is more efficient for our color detection logic. Here's what each component represents: Hue refers to the color itself, Saturation indicates the intensity of the color and Value represents the brightness. The HSV model allows us to define color ranges effectively. To detect a specific color, we define these ranges within the HSV color space. The program then looks at each pixel, checking for a significant presence of the desired color. Additionally, we've defined a mask that determines whether the sum of pixels for a particular color exceeds a given threshold (which is adjustable). If the sum is higher than the given threshold, the program identifies and returns the color. If no defined color presence exceeds the threshold, the program simply reports "not recognized."

However, it's worth noting that we adopted a different approach in the final version of our project.

2.3.1 Final Approach: ColorMath

ColorMath is a Python module that implements a wide range of color operations, including color detection. In our approach, we first defined the four different UNO colors to compare them with a given color.

Our program iterates through all four UNO colors and converts them from RGB to Lab Color Space. Lab is more perceptually uniform compared to RGB, making it better suited for color comparison as it aligns more closely with human vision. After this conversion, the program calculates the color difference using the CIEDE2000 formula, which serves as the corresponding color difference metric. Finally, the program returns the color that has the lowest difference with the card color to be determined.

Now, let's address how we detect the card number.

2.4 Card number detection with YOLOv8

Due to challenges in detecting the numbers on UNO cards using OpenCV, we used to train a neural network for this purpose. Our approach involved a pretrained YOLOv8 [3] model as a base and adjust it specifically for UNO card detection with transfer learning. YOLOv8 is user-friendly and easy to use, making it suitable for a wide range of object detection tasks, including UNO cards. We created a lot of UNO card images and labeled them to train our model. Notably, this labeling process consumed a significant amount of our time, but it was crucial for achieving a well-performing model.

2.5 Creating and collecting training data

We created our own dataset with our own images and combined them with an existing dataset for subsequent training. To create our custom dataset, we used an online tool called CVAT [4]. CVAT allows users to label images and export them in various formats, including YOLO formats, which suited our needs.

Of course, labeling requires images. Tim decided to capture a numerous amount of UNO card images using his phone. After that, we imported these images into the CVAT tool and labeled them.

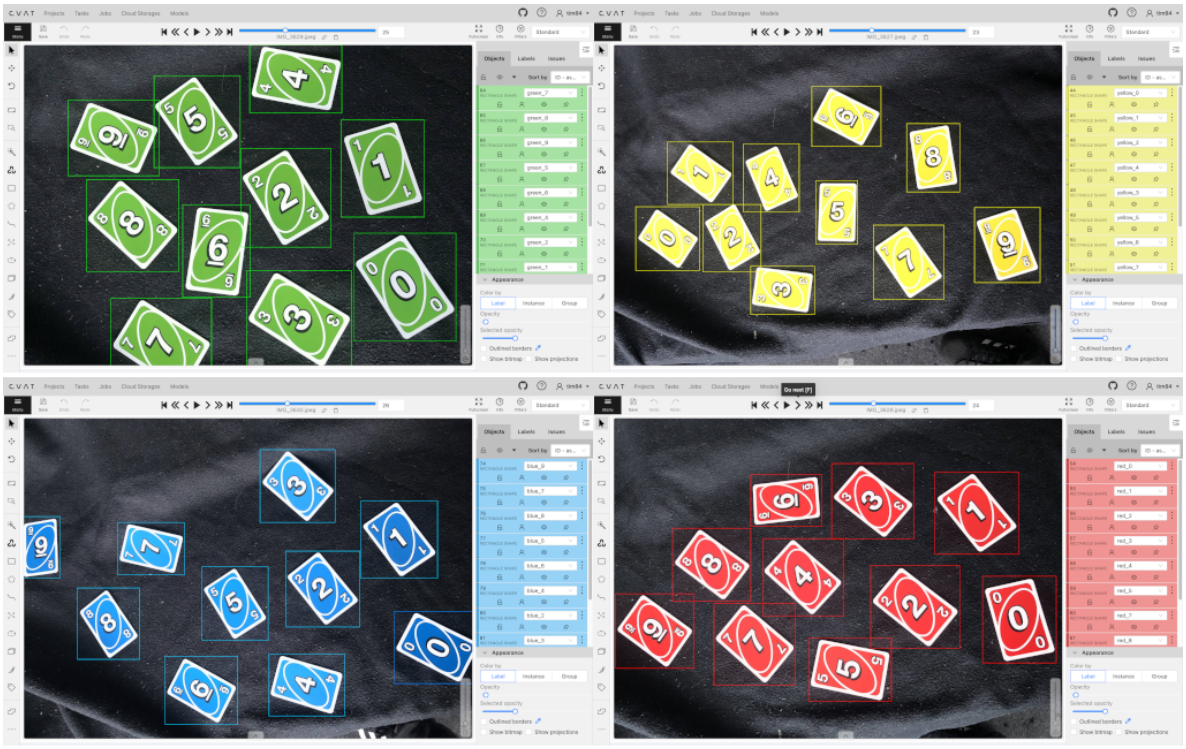


Figure 1: Labeling our own UNO card images in CVAT

However, due to the need for more training data, we looked for an additional image source. Fortunately, we discovered an UNO cards dataset [5] on Roboflow [6]. We downloaded this dataset and initiated the training process, combining images from both Tim's phone and Roboflow.

2.6 Training

We initially trained the model using UNO card images on a local computer for three epochs. However, we discovered that this wasn't enough, as the model's performance on real UNO cards was not that good. Metrics such as precision [7] and mAP50 [8] indicated that there was more potential with longer training.

As a result, we decided to retrain the model, this time for 200 epochs. Because the first training on a local computer was very slow, we used the JupyterHub hosted by the Institute for Information Systems at Hof University (iisys) to speed up the training process. Once the second training cycle was finished, we tested the updated model with real UNO cards and observed significant improvement.

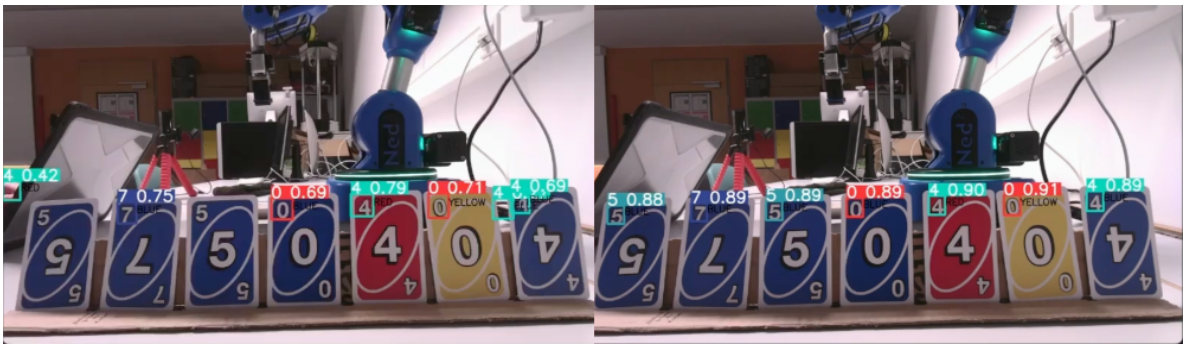


Figure 2: YOLOv8 model performance comparison

On the left side, you can see a comparison between the model trained for 3 epochs and the model trained for 200 epochs on the right side. The longer-trained model successfully detects all 7 cards in the image, whereas the previous model only identified 5 cards and had several problems, such as mistaking Tim's hand for a '4'.

2.7 Combination of OpenCV and YOLOv8

We achieved a model with a good performance, but it was limited to detecting the numbers on the cards. However, what about the color? To address this, we made slight adjustments to the color detection component of the existing Python project, enabling it to work seamlessly together with the trained model.

Here's how it works: When the model identifies numbers in a given image, it produces an array of bounding boxes, each bounding box encapsulating a single number. To also determine the color, our Python program extracts the color from a pixel located at the center-right boundary of each bounding box. We specifically chose this position because it minimizes the likelihood of including a pixel from the number itself. This approach allows us to safely detect the color of a card using the existing color detection logic.

2.8 System Design

The overall system is designed as a facade, structured into different subsystems to manage complexity. The system consists of three major subsystems: The Game Logic system implements the rules and mechanics of a typical UNO game. The Card Detection system is responsible for identifying and tracking cards and also updating the game statistics to enable the robot to play the correct cards. The Robotic Arm system's role is to manipulate cards and moving them from one position to another.

2.8.1 Game Logic

Let's take a quick look at the rules of UNO. To play properly, you'll need at least two players. The game begins with one player, and then the next player takes their turn. The game is over when a player has played all their cards.

Here's how the game flow works: The game requests the active UNO card from the card detection system. Then, the card detection system takes an image and converts it into an array of UNO cards. Next, the game determines the next player and prompts them to take their turn. After the player completes their turn, the game checks if they have won. If so, it proceeds to the cleanup phase. Otherwise, it switches to the next player. To differentiate between human and robot players, we added an interface for a "generic player." Both human and robot players can implement their own logic within this interface. The human player's turn automatically ends after a specified time, but if the system detects a change in the main stack, their turn also finishes. Meanwhile, the robot player calculates the next possible card to play and coordinates the robot's movements.

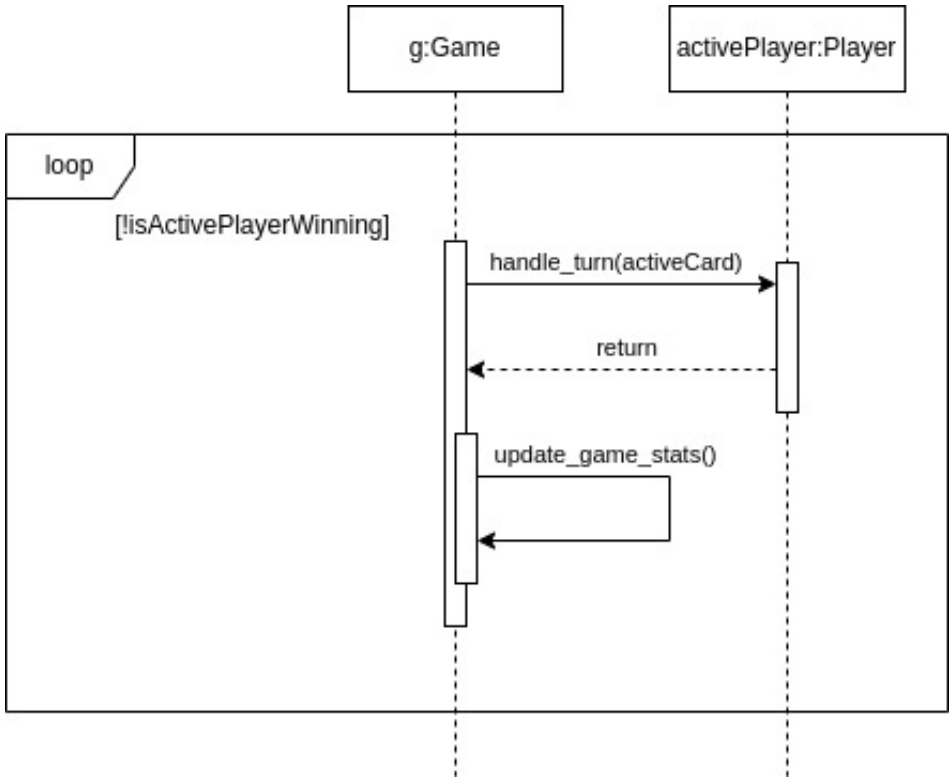


Figure 3: Workflow of the game logic

2.8.2 Card detection

The goal of card detection is to recognize both the top card on the main stack and the robot's hand of cards. This information is crucial for determining whether the robot can play a card and, more specifically, which card it should play.

To achieve this task, the card detection system provides a function that takes the requested camera index as input and returns the detected UNO cards. If you want to predict the cards on the robot's hand, you'll need the camera pointed at the cards. Otherwise, it will predict the cards from the main stack. The camera detection process automatically maps the recognized cards to their corresponding positions in the image. For example, a bottom-left yellow '4' card might be mapped to position 1, and so on.

2.8.3 Control of the robotic arm

We need to verify whether the connection to the robot remains active. Additionally, we desire additional functionality when a method of the robot is called. To address this, Lukas designed a proxy for the robot.

The proxy serves two purposes: Minimizing requests, so it reduces the direct requests made to the robot itself and permission checks, so the proxy determines whether a specific move is allowed or not. Its primary objective is to control the robot and execute physical tasks, such as grabbing the card at position 2 and moving it to the main stack. If no connection exists, the operation will fail.

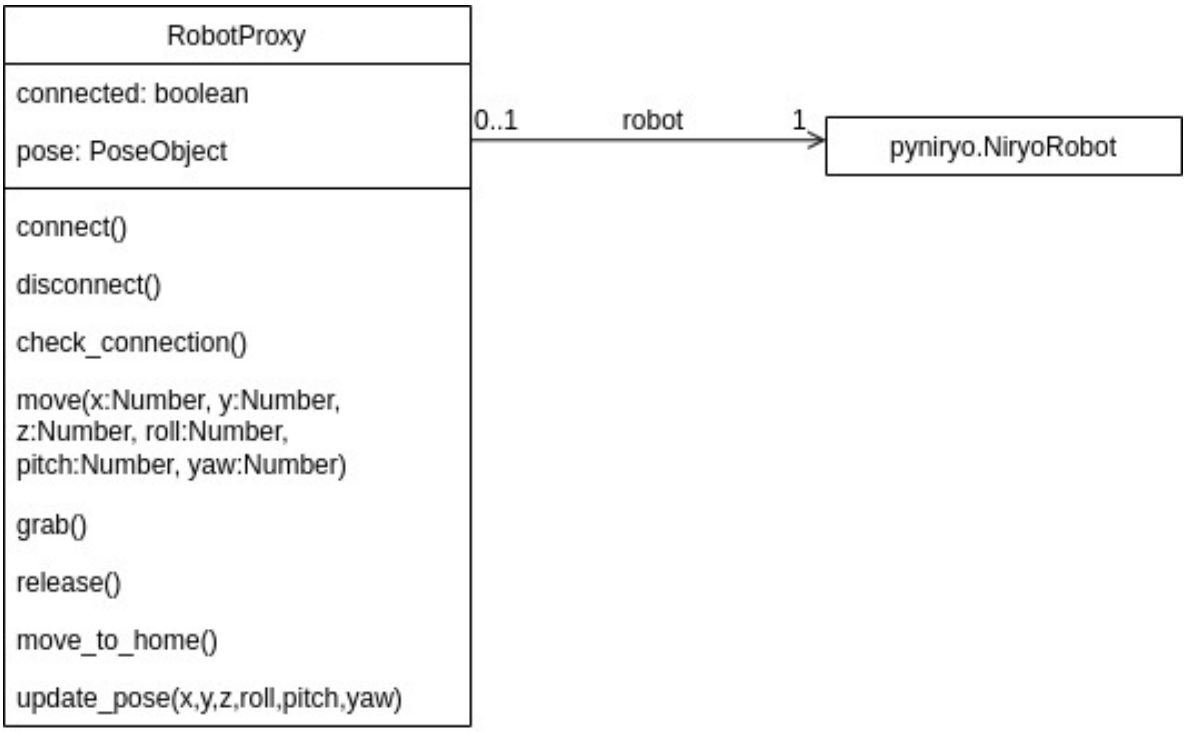


Figure 4: Class diagram of RobotProxy

3 Experiments

Throughout the development process of RcpU, we explored various approaches related to the positioning of items such as the robot, the card holder, the camera, and the overall structure of the robot's environment.

3.1 Picking up the cards

Our initial concept involved equipping the robot with a suction cup. The idea was that when the robot detected a card, it would position itself above it and then lower its arm until the card attaches to the suction cup. However, we encountered a challenge with this approach: While picking up the card using the suction cup could be implemented relatively quickly, releasing the card would result into new problems and significant adjustments. Consequently, we switched to an alternative solution: Utilizing the robot's gripper and attaching rubber bands to enhance its efficiency in grabbing the cards.

3.2 Structuring the environment for the robot

In the project's beginning, we encountered challenges with card detection due to changing light conditions. Our OpenCV-based detection struggled with changing backgrounds and light reflections, forcing us to explore potential solutions.

The most successful approach involved using a black blanket provided by Niklas. By placing the cards on this blanket, they became more distinct from the background. The OpenCV detection algorithm could then more precisely delimit the cards, benefiting from the high contrast between the black blanket and the card colors. Ultimately, this improved edge detection. Interestingly, as our project evolved, we found that the black blanket became unnecessary. The trained YOLOv8 model now accurately detects all types of UNO cards, regardless of the background.

4 Conclusion

Our team gained valuable insights into controlling a robot using Python, training neural networks through transfer learning, and implementing color detection. We effectively divided our project into smaller components, with each team member contributing to specific tasks. For instance, Niklas and Aruzhan took care about color detection, Lukas focused on robot movement, and Tim trained the neural network to enable the robot's visual recognition of UNO cards.

This project served as a trial for us, as all team members are in higher semesters and have experience in courses such as Object-Oriented Programming, Artificial Intelligence, and Applied Machine Learning. Combining our diverse skills into a single project was a rewarding experience!

References

[1] Niryo Ned2 User Manual: https://docs.niryo.com/product/ned2/v1.0.0/en/index.html

[2] Steam link to UNO: https://store.steampowered.com/app/470220/UNO/

[3] You Only Look Once: https://docs.ultralytics.com/

[4] Computer Vision Annotation Tool: https://www.cvat.ai/

[5] Link to UNO cards dataset: https://public.roboflow.com/object-detection/uno-cards

[6] Roboflow is a great source for datasets, tools, tutorials etc. for machine learning

[7] The accuracy of the detected objects, indicating how many detections were correct

[8] The mean average precision, a measure of the model's accuracy considering only the "easy" detections

[9] The smallest pretrained model, trained with the COCO dataset

Documentation Vovracar

Kevin Mündel
Mobile Computing

Moritz Süß
Mobile Computing

Patrick Alves
Mobile Computing

***Abstract* — This is a documentation of our project Vovracar. That contains the birth of the idea, the materials and the whole process.**

I. Introduction

At the beginning we got presented with some interesting ideas for our project. „Robopets“ was one of them. We wanted to set up some basic applications, so other groups could build upon our ideas. The basic things a Robopet has to do is to react to a users voice and follow its orders. To follow its rules, it has to move and the easiest thing to move is a model car. But this wasn't enough for us. We wanted to do something exciting. We had some prior experiences with VR applications, so it didn't take long for us to finalize our thoughts. We planned to create a model car that is voice-controlled and equipped with cameras to stream the video to a VR headset. That was the birth of Vovracar.

II. The Beginning

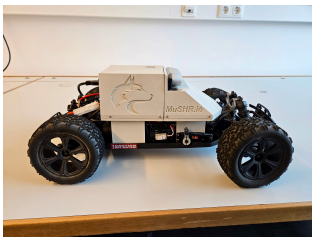
A. Idea

First of all we had to define some goals and functions we want to deliver. We aimed to build an application which can control a racing car through voice commands. The focus would be on simple commands such as „forward“, „left/ right“ and more. The car's speed should be controllable with voice inputs.

In addition we wanted to use the existing camera on the race-car to stream the video to a VR headset. This enables the user to make better and more reasonable inputs. We would use the VR headset's microphone to receive the voice commands.

These two functions were our key functions. If we had some time left, we planned to use sensors, which assist controlling the car and reducing the number of collisions and accidents. Depending on the required distance between the car and the user, there may have also been a need to create a race track. With the already mentioned sensors we could have establish checkpoints the user has to drive through.

B. Materials



As already mentioned before, we needed a model car and we got provided with a race-car. This was a MuSHR race-car, which already had a camera installed. There is a Nvidia Jetson Xavier NX built in which is used as a built in PC. There are python scripts on that PC which were used for an app before. We were allowed to use these scripts. We decided to use the Meta Quest 2 VR headset because we had some work experiences with it. We were also given a Windows PC to work with because we have reached some understanding during working with Unity and the Meta Quest 2. It was required to use a Windows PC if we wanted to use the Oculus Integration SDK and the Oculus app. The usage of sensors was a goal we set up, if we get finished way faster, but that was optimistic.

C. Concept

Before getting to work we had to decide on some concepts and design decisions. There were two ways to control the race-car, either dynamic or static. Controlling the car in a static way would mean that the user needs exact information about the distance and direction the car needs to drive. That's why we decided to use dynamic controls. That means the car uses one command and executes that command until „stop“ or another command is put in.

Another thing we had to think about is what application do we want to produce. We had two choices. We could make a web application or a unity application. After some research we decided to make a unity application because we already had some experiences working with a VR headset in unity. It's easier to create an Android application in Unity, which is getting launched on the VR headset. We need Unity to translate the input by the VR headset and send it in a way to the race-car, so the python scripts can be used. This has to work the other way too because the camera can be controlled through a python script which needs to send the video to the Unity application. The video needs to be made usable for the Quest 2, there..

III. Development

A. First Approach

Our initial plan was the installation of ROS. To do that we had to install Docker and make some initial set up. We needed a Nvidia graphics driver to be able to run ROS on the system. Unfortunately, this driver was not available for macOS, so we had to find another solution. During that time we started to realize how valuable the existing python scripts are and we cancelled the idea of using ROS from scratch. After failing the first approach we decided on testing the existing python scripts.

B. Testing the python scripts

We had a lot of scripts on the race-car and it took some time to get through them. We found two scripts which were usable for us. The first one was named „car.py“ and the second one was called „carServer.py“. These scripts contain all the controls for the race car and got used in an app developed by another student. With the first script we were able to test the controls via the command line, while with the „carServer.py„ script the controls were controllable via the API endpoints.

We encountered two problems while testing the scripts. The car didn't react to any input we made, but after changing some ports we were able to give commands via console. The other problem we had to solve is the camera stream. Firstly, we tested the camera functionality in the browser which didn't work at the beginning. After some bug fixing we got the video, but only in form of pictures and not as a video. We had to rewrite the script to get a video stream. That worked and we could get the video stream on our MacBooks.

C. Meta Quest 2

After checking the scripts, we wanted to put all the pieces together. But we didn't expect the following problems. The racecar has a Linux operating system and we wanted to install the Oculus SDK onto the racecar. This didn't work out because it was only available for Windows, but for MacOS it still had some problems.

We skipped testing the VR headset for now and split ourselves in two groups. We split the following tasks into two parts. The first task was to adjust the python scripts to be used by the Unity application and the second task was to set up a Unity project and do the basic implementations to use the VR headset in our project.

D. First problems and challenges

At this point we got access to a Windows PC because working with Unity on our MacBooks didn't seem to work without problems. We wanted to install the newest Nvidia graphics driver on the PC to be able to use SteamVR, which would get us access to the Meta Quest 2.

We also changed the camera functions on the Flask server. This had to be done because the functions only sent pictures, but no video. Setting up the Unity project went according to our plans and we could already do changes to the Unity script. This enabled us to use the camera and speech recognition.

But the next problems appeared. After installing the Oculus Integration SDK we got the info that it was deprecated since a few months. To use the Meta Quest 2 in combination with Unity we had to put it into developer mode. For this, we installed the Meta Quest App on a smartphone to connect it with the Meta Quest 2 via Bluetooth and put the headset into developer mode. We also had to use our own Meta account on the Quest 2 for this. This worked after deleting the existing accounts on the Meta Quest 2. On the PC, we installed the Meta Quest Developer Hub with which we could launch apps on the Meta Quest 2.

Now that we had installed the necessary software, we wanted to display the camera stream from the car in our Unity project. To do this, we tried to display the stream in our app, which we could retrieve using a get request via the API. Our first approach was to convert the video stream into a byte array, but this didn't work at first.

At this point in time we thought about scrapping the VR component of our project because it seemed like too much of a task to make it work.

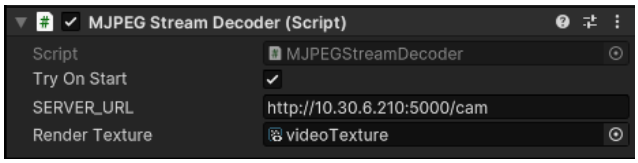
E. Access to video stream finished

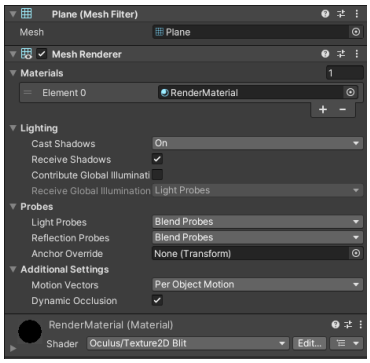
But after a short replanning we changed our approach. Firstly we completely set up our unity project in the most basic way. We didn't want to run into any problems we weren't responsible for.

We then checked the way to connect to the racecar and using its scripts again. We also confirmed our suspicion that we had a little setup problem. If we wanted to start the scripts on the racecar through our application, we would have needed a SSH connection to start the scripts or something similar. Because that wasn't that important, we decided to work on voice commands and camera stream before giving this more attention.

The camera stream was accessible through HTTP requests which worked just fine when accessing the racecar from our MacBook. But we had difficulties setting it up in unity. After some testing and debugging we managed to make it work.

With the help of a MJPEG stream decoder, which we found on GitHub, we could stream the camera from the racecar to the unity project. This decoder let us take the video-stream and converted it into 2D Texture. We set up the plane as a child of the camera, so whenever the user moves his head he would always see the screen in front of him.





We created a Render Material and selected the standard Shader at first, which didn't work until we found out that the correct Shader was the Oculus/Texture2D Blit. After that, we added this Render Material to our Mesh Renderer on our plane object. We added the MJPEG stream decoder to the parent object which contains our plane. This stream decoder streams the camera from the race-car to our video texture.

F. Voice Control

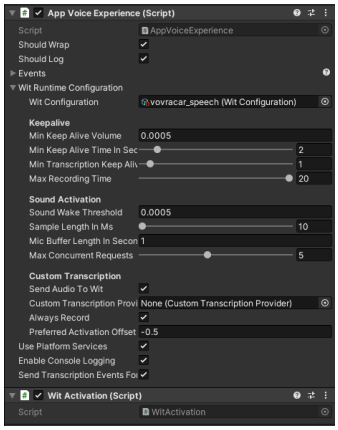
With the first task completed we shifted our attention to the voice control part. At first we tried a keyword recognizer from Unity's own Windows.Speech library. With this method, however, no POST requests were sent and we thought that this was due to the microphone. Therefore, we used Unity's microphone interface to get access to the microphone of the VR goggles. After the microphone is started, it should record for ten seconds. This recording should then be converted to a byte array and sent to Wit.ai.

But first we had to set up Wit.ai which is a NLP model from Meta that can process speech. On the Wit.ai website we created an app called vovracar_speech and set the visibility to „open“. In this Wit-app we defined a few utterances. Based on these we defined appropriate entities and intents.

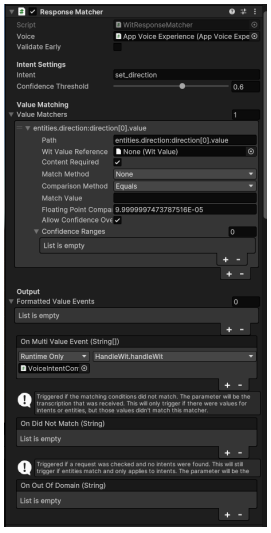
Entities are key terms or concepts that are to be identified in the texts entered by the user. They represent the important information that is to be extracted from the user input. In our case, the entities contain keywords such as "start" or "stop", i.e. the commands for controlling the car. Intents represent the intention or purpose behind the user input. They describe what the user wants to achieve with their input. By combining entities and intents, a NLP model can understand what the user wants to say and which actions should be executed based on this. When an input is sent to Wit.ai, it is checked for the keywords and a response is sent back in JSON format. This response contains the respective entities that were found in the voice input.

Entity	Roles	Intents
action	action	set_action, set_direction
direction	direction	set_direction, set_value
value	value	set_action, set_direction, set_value

Name	Entities
set_action	action:action, value:value
set_direction	value:value, direction:direction, action:action
set_value	value:value, direction:direction



The Meta Voice SDK, which is included in the Meta All-in-One SDK, has already provided us with scripts for using Wit.ai. Wit.ai first had to be configured in Unity, for example the personal server access token of our Wit-app had to be entered in order to gain access to the NLP model. To be able to receive responses from Wit.ai, an App Voice Experience is required as a game object. This contains the Wit Configuration, which is started when the Unity app is opened via a script "WitActivation.cs" created by ourselves.



Since we still had no access to our microphone, we tried to use the LipSync demo included in the Meta Voice SDK. This allowed us to check if Unity could get access to the Meta Quest microphone at all. When it worked in this demo project from Meta, we hoped to be able to implement it in our project as well. Because that didn't work either, we planned to use a self-defined speech-to-text script instead of text verification with Wit.ai. This was to convert the audio input from the microphone into text, which we could then check for the keywords.

After this didn't work either, we found out through a YouTube tutorial that the Meta Voice SDK automatically uses the correct microphone and that our error must lie elsewhere. It turned out that we were not using Wit.ai correctly. To handle the requests correctly, we needed response matchers in our Unity project to provide us with the responses from Wit.ai.

This responses are then passed to our "HandleWit.cs" script or its "handleWit" method. The method checks the passed response for its value and then executes the corresponding POST request within a coroutine to the Flask server. These POST requests can be used to control the car's API endpoints, which influence the motor and wheels.

G. Further improvements

Now that the app was basically working, we wanted to implement further functionalities. On the one hand, we wanted to be able to specify the angles for steering to the left or right and the desired driving speed as a number in the respective voice commands. Secondly, we wanted our car to be able to drive certain shapes such as circles or a figure of eight using voice commands.

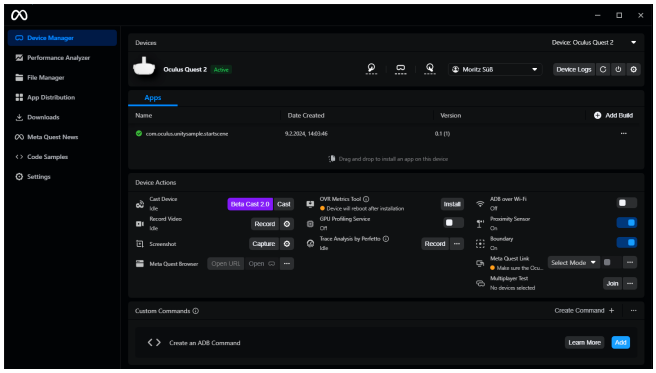
However, processing the angle or speed turned out to be difficult, as the handleWit method only ever receives one value at a time. As the number is an additional value alongside the direction, for example, it cannot be processed in the same interation of the handleWit method and therefore cannot be passed in the POST request. Instead of using request handlers, we tried to use utterance handlers in order to be able to process not only the value, but also the entire language input including the number.

In the end, the utterance handler worked, but we had to create significantly more objects in Unity and the transfer of values still didn't work for unknown reasons. As our basic implementation had also stopped working properly at this point, we stopped developing the additional functionalities.

IV. Instructions

First of all the car must be started. An SSH connection is used to access the car. The script can also be started on the car, in which case a screen, mouse and keyboard must be connected. When starting the vovracar.py script via the terminal, it is possible that the port 5000 on which the script will run must first be cleared. This will be indicated in a message in the terminal. Clearing the port (in our case 5000) is done with the command "lsof -i:[Port]". The application running on the port that is displayed must then be terminated with "kill [PID]". The script can then be executed with "python vovracar.py" in the "Desktop/Vovracar" directory.

If the app is to be started on the Meta Quest 2 via the PC, the Meta Quest Developer Hub is required on the PC. You must log in to the Meta Quest Developer Hub with a Unity Developer account, which is also used to log in to the VR goggles and the Meta Quest smartphone app. With the Meta Quest smartphone app, the VR glasses must be set to developer mode so that apps can be installed on them via USB from the PC.



If no APK file of the Unity project exists yet, it can be built under File / Build Settings in Unity. Make sure that the app is created for Android, that "Use Player Settings" is selected for Texture Compression, that the correct device is selected and that the Compression Method is set to "LZ4". The APK file created must be added as an app in the Device Manager of the Meta Quest Developer Hub. Then the app can be launched from there.

If there is already an APK file on the Meta Quest 2, it can be started in the menu under Library. To find the file easier in the library, you can filter for "unknown sources". It needs to be ensured that the Meta Quest 2 as well as the Ubuntu server on the car are in the same network. Otherwise it's not possible to communicate with the Flask server.

The voice commands "left", "right", "forward", "backward", "start" and "stop" are available for controlling the car. "Left", "right" and "forward" change the angle of the front axle, while "start" and "backward" move the car forward and backward respectively. "Stop" ends all current actions and brings the car to a standstill.

V. Testing

The first tests were for the video stream. These were easily done by checking wether we get a video or not. At one point we just got one picture, but this problem was easily fixed.

The tests regarding the voice control were surprising. First, we could only send one POST request. Every other command didn't work. After adjusting the code a bit we could make more POST requests.

During that period we saw that the servomotor wasn't well assembled. We had to be careful with the angles of the wheels.

Then we wanted to make the video on the next day. But we encountered the same problems again. The car couldn't receive any requests and our adjustments could only result in getting one post request per test run.

On the following day we tried our working application again and it worked. When we tried making the video, we found that one of the cables which was connected to one of the batteries had loose contact. And then we encountered the problems from before again.

After sending the post requests from another device, it seemed to be a problem with the Meta Quest 2 because the requests being sent from our MacBooks worked just fine. This is an assumption and we can't really prove it.

VI. Problems

There could be problems regarding the IP address. If the car doesn't connect to the application, you'll probably have to change the IP address in the Unity script.

There may also be connection problems between the car and Meta Quest 2, especially when there is a lot of network traffic. The exact cause of this is unknown, but it is most likely a network problem.

Another possible problem could be the Meta Quest 2 itself. After sending the post requests from another device we made the assumption that the VR Headset has problems regarding voice input or connection.

Voice input should be in English, commands in other languages will most likely not be processed correctly by Wit.ai. In addition, problems can occur with unclear voice commands.

It must be ensured that the microphone of the Meta Quest 2 is not muted and that the app is allowed access to the microphone.

VII. Future Projects

We made a pretty basic application, so improving the application in any way would be ideal. It doesn't have a UI. Displays for speed and battery life would be very helpful. A warning pop up when driving too far away form the user would be a good addition.

The voice commands could be trained more and new commands could be added. Our commands are set to have default values, so there could be improvements, which could led to the user being able to make specific left and right turns or decreasing and increasing the speed while driving.

VIII. Lessons Learned

At the start of the project we thought that this would be way an easier task. We have worked on an unity project with VR integration before, so we thought that adding voice commands and letting a race-car drive would be the only challenges where we needed to invest a lot of time.

But this assumption couldn't be more wrong. The biggest problem we had to face is the incompatibility of operating systems were working on. We realized pretty fast that the project is very hard to do on Macs and with the help of a separate PC, which we got later, we managed to progress more smoothly. But the most annoying thing is that Meta devices and applications don't support Linux, but our race-car only has a Linux server installed. Therefore we needed to make an unity application to connect the Meta Quest 2 with our race-car's Linux server.

This led us to have regular complication and some of these were very special interactions. We don't really know whether more planning would have changed something. The one thing we learned from this is that we are wary of using different operating systems in combination.

IV. Conclusion

To summarize, we tried to send the video stream from the race cars' camera to a VR headset. By using the microphone from the headset we wanted to send voice commands to the Flask server, which handles the requests to control the car. Technically speaking we made it work, but we encountered some unknown problems at the end. We could only make assumptions on the problems because documentation and general support for the Meta Quest 2 is lacking or deprecated.

Connect Four with a Robotic Arm: A Comprehensive Approach to Human-Robot Interaction in Gaming

Claudio Melero Martinez
Computer Science
Hof University of Applied Sciences
Hof, Germany
claudio.martinez.melero@hof-university.de

Arent Dollapaj
Computer Science
Hof University of Applied Sciences
Hof, Germany
arent.dollapaj@hof-university.de

Ali Sencer Irmak
Computer Science
Hof University of Applied Sciences
Hof, Germany
ali.irmak@hof-university.de

Oguzhan Icelliler
Computer Science
Hof University of Applied Sciences
Hof, Germany
oguzhan.iceliller@hof-university.de

Vicky Wang
Computer Science
Hof University of Applied Sciences
Hof, Germany
vicky.wang@hof-university.de

***Abstract*—This paper presents a robotic arm system capable of playing the game of Connect Four against human players. The system utilizes a combination of artificial intelligence and robotics techniques to achieve high-level strategic decision-making and precise motor control for executing moves. The AI (artificial intelligence) component employs an algorithm to evaluate potential moves and select the most advantageous ones. The camera allows the board to be visualized by the AI. The robotic arm, controlled by the AI decisions, can accurately place and manipulate game pieces, allowing it to compete against human players with a competitive level of skill. [We evaluate the performance of the system in a series of human-robot matches, demonstrating its ability to consistently defeat human opponents. The proposed system represents a novel approach to human-robot interaction in gaming, combining the cognitive capabilities of AI with the physical dexterity of a robotic arm to create an immersive and engaging experience.**

Keywords—Robotics, human-robot interaction, Connect 4, artificial intelligence, minimax algorithm.

I. Introduction

Connect Four, (also known as Four Up, Plot Four, Find Four, Captain's Mistress, four in a row, Drop Four or Gravitrips in the Soviet Union) is a two-player strategy game in which players take turn dropping colored coins into a six-row, seven-column vertical grid. The objective is to be the first to get four coins in a row, vertically, horizontally, or diagonally.

The game is almost 50 years old, (first introduced in 1974), and is still popular to this day. It is a great game against all ages and all skill levels, family, friends or even a computer.

The game is played as such, each player takes turns dropping coins in the grid, filling it up until someone wins, or the board is filled, making it a tie.

To make this game playable by computer, we separated the project into 3 components, the AI, the robot arm, and the visual recognition of the board.

The robot arm will be connected to a camera that will capture images of the game board. These images will be sent to the backend, which will use an AI to determine the best move to make.

II. The game

A. The game logic

The game logic is made in python, the player starts by choosing its difficulty, and afterwards has the honors of starting first.

Before a move, the game verifies if the given move is valid (e.g. a column is full, the column is in the range of the board) and places it if so.

After the move the game checks if a winning condition is here for the one who played or if the board is full, making it a tie, otherwise it proceeds to the other player's turn.

B. The artificial intelligence

When choosing the difficulty, the player is choosing the depth of our AI, ranging from 3 to 6. The calculation for 3 is instant, whereas for 5 can take up to a second. The last difficulty can be challenging, having to see at least 6 moves in the future to match the AI.

The AI works as such, depending on the depth, it will see its amount ahead. So, a 4-depth AI will look 4 moves ahead. Each move it makes at that depth is evaluated to see if a move on that column will create a win condition. If it does, it places it immediately as it is the best move. Otherwise, it either maximizes his chances of winning, or minimizes the player's chances of winning by blocking his winning line.

C. The robotic arm

We use the Niryo's Ned2 6 axis robot in the project. We control this robot using the python API "pyniryo" and "pyniryo2". This robot is used to pick up the pieces from the conveyor belt and place them into the relevant row on the Connect Four game board.

Before the game starts the robot must be calibrated then set up all of the 21 pieces on the conveyor belt. We assume that the robot is placed on the right side of the conveyor belt, assuming the conveyor belt's motor is on the left and facing away from the view point described here. 21 pieces should be placed onto the magazine where the robot will pick them up and place them on the belt.

What we describe as calibration is the process of robot calibrating its motors if they are not already calibrated, set up the magazine position and calibrate the game board position.

For the magazine while there is a pre-coded draft magazine position the user asked to show the robot the exact position of the magazine's mouth where the robot is going to pick them up. User shows this position using the "Freemotion" button on the robot (or the "Freemotion" button from the Niryo Studio software) and moves the robot to the relevant position while holding the button. After that user should confirm by pressing enter where they are prompted from the terminal. It should be also noted that for precise movement the robot arm will slow down while approaching, and moving away from the magazine.

After this robot will pick up the first piece from the magazine and will calibrate the game board using this. We assume that the game board is placed parallel to the robot piece holes facing the robot and the other side facing the camera. Similar to magazine calibration there is pre-coded draft game board position but the user is asked to show the robot the first row using the "Freemotion" function again. After this the robot saves the position of the relatively moved 0.05 units moved on the Z axis according to the first row. Then the robot will show and ask the user to confirm each row one by one by moving relatively to the first one.

The reason we are using the "Freemotion" button to get input from the user is that during our testing we have realized that the robot is not really accurate in terms of positioning between different runs and different calibrations. So setting up a constant position for the game board and magazine which are crucial parts for the robot caused us more issues compared to implementing the "Freemotion" method.

The belt placement works where 2 pieces are placed next to each other where one is close to the robot and the other one

is far from the robot. After 2 pieces are placed they are moved back in the conveyor belt making space for 2 more pieces. We refer to these 2 pieces next to each other as a stack. According to our calculations, the conveyor belt moving at 15% speed for 4.3 seconds for each stack lets us fit exactly 22 pieces on the conveyor belt, this is just enough for our 21 pieces. The belt is moved asynchronously using threading where the thread will start the belt and stop it after 4.3 seconds. This allows us to operate the robot arm while the belt is moving.

To speed up the belt placement where lets say from a previous game "n" amount pieces were placed and there currently "s" amount of pieces on stack (the stack can have 0, 1, 2 pieces). The user can give these parameters to the robot and calculation will start from the leftover piece therefore the user does not have to place 21 pieces if there are some set up from previous games already.

After all of these operations the robot will wait for the algorithm to respond with an answer. The answer will be a value 0 to including 6 where each number represents a game board row. The robot will pick up the next piece, which is the far side from the robot first, and then it will place it into the requested game board row by the algorithm. To avoid colliding with the game board the robot will move its joint 1 then switch back to the idle position. The robot while approaching the game board will also slow down its arm like the magazine for precise movements.

To pick up and drop the pieces we are using the robots gripper module. While this module seems very easy to control from the API it has a software when it is used for long periods of time, the software will report it as overheated, rendering the module inoperable. To counteract this we can restart the gripper using a function from the "pyniryo" python library. While for all other things we are using the newer "pyniryo2" library for only just this function we need to use the older version of the library because this functionality is missing from the new library. However our robot's firmware was also out of date which in the future versions of the firmware this might be fixed so we cannot entirely blame this issue for Niryo.

To fix this issue we check if the gripper is in an error state or not before operating it. If it is an error state we call the error handling routine for the gripper. In this routine we reboot the tool and wait for 5 seconds and check again if the state is cleared, if not cleared we repeat this process until it clears. While this in most cases fixes the issue there are still some edge cases where it fixes the issue during the routine but just as the routine finishes the gripper fails again and causes the requested gripper action whether it should open or close to fail and not report anything or raise any exceptions during the code resulting in the robot skipping critical actions. To detect this we also check for error state after the requested action and if another failure is detected we ask the user whether they want to repeat the action or not. This way the user can repeat the action if the robot faces this weird edge case.

We also faced issues with the "pyniryo2" API where calling a function will throw a "RosTimeoutError" exception where the API fails internally. To counteract this issue we wrap every "pyniryo2" API call with a function where it repeats the call until no exceptions are thrown. While this might seem like a bad programming practice there is no way to detect this beforehand and it is completely random with regards when it is going to happen. Sometimes we never face this issue. Other times we face this issue upwards of 10 times. This issue also caused us to not plug the robot's camera to back of the robot but rather plug directly to the computer which runs the python program. This causes unnecessary slow downs during the execution of our program where this is not our fault. Like we said before we cannot blame Niryo here because our firmware was out of date and maybe this was fixed in a future version of the firmware.

D. The camera

We use Niryo's Ned2 default camera in this project. However, the camera has a distortion issue. In order to solve this issue, we use Intrinsic Parameters and Distortion Coefficients. The aim of using a camera is detecting the pieces on the game board and distinguishing each piece based on its color, whether it be red or green. To achieve this goal, we leverage the Python OpenCV library. Following the detection and differentiation, we put numbers to an array to represent the game board. Specifically, we assign 0 for empty spaces, 1 for red-colored pieces and 2 for green-colored pieces. To facilitate this classification, we establish lower and upper boundaries for the red and green colors. We create masks to isolate pixels falling within these color ranges. We proceed to identify contours within these masks. Finally, we update the array to represent the colors.

E. The board

Board and Coin Material and Manufacturing:

The game board and coins were fabricated using polylactic acid (PLA), a biodegradable and compostable polymer derived from sustainable resources like corn starch. All components were designed and 3D printed at the University of Hof's MakerSpaces, leveraging open-source software for both the design and printing processes.

Blender: This open-source 3D creation suite was employed for the design and modeling of the game board and coins.
Cura: This open-source lamination software served as the platform for preparing the 3D models for optimal printing on the Ultimaker S5 & Ultimaker 3 printers.

III. CONCLUSION

This project successfully explored the integration of a robotic arm into the classic game of Connect Four, creating a unique and engaging human-robot interaction experience.

Key Achievements:

Developed a multi-component system: The project combined an AI for strategic decision-making, a robotic arm for physical interaction, and a camera system for visual recognition of the game board.

Implemented a depth-based AI: The AI, with adjustable difficulty levels, evaluated future moves and optimized its choices based on the chosen depth.

Integrated a robotic arm: The robotic arm, controlled through Python libraries, picked up and placed game pieces according to the AI's instructions.

Established camera-based recognition: Using OpenCV, the camera identified and differentiated pieces based on color, mapping the game board state for the AI.

Employed sustainable materials: The game board and coins were 3D printed from eco-friendly PLA plastic.

Challenges and Solutions:

Robot arm calibration: The project addressed robot positioning inconsistencies by utilizing user input during calibration through the "Freemotion" button.

Conveyor belt speed optimization: Calculations determined the optimal belt speed and placement strategy for efficient piece distribution.

Gripper overheating: The system implemented error handling and user interaction to address occasional gripper overheating issues.

API limitations: Workarounds were developed to handle API-related exceptions and ensure smooth program execution.

Camera distortion: Intrinsic parameters and distortion coefficients were used to compensate for camera distortion and improve image processing.

Overall, this project demonstrates the potential of robotic integration in game playing, offering a novel and interactive experience while overcoming technical challenges.

ACKNOWLEDGMENT

FixMix

Julius Bauer
Applied Robotics WS24
Hochschule Hof
Hof, Deutschland
jbauer4@hof-university.de

Carsten Philipp Duttiné
Applied Robotics WS24
Hochschule Hof
Hof, Deutschland
cduttine@hof-university.de

Niklas Daniel Jenisch
Applied Robotics WS24
Hochschule Hof
Hof, Detuschland
njenisch@hof-university.de

***Abstract*—This paper presents the development and construction of an automated cocktail mixer, named "FixMix", designed for private use. There already are many household appliances that automate tasks like brewing coffee and preparing food, but there are almost no solutions to the preparation of refreshing beverage blends. Our motivation was to create an automatic cocktail mixer, which provides a seamless blend of convenience and consistency, offering users a quick and effortless way to enjoy mixed drinks. Existing solutions accomplish their purpose but use deprecated technologies and packages or provide only a single way to interact with the user interface. While the "Smart Bartender" by the "Hacker Shack" had buttons and an OLED display, "nebhead" used a web application to control his "PiTender". Since there was no combination of the two forms, we did it ourselves and while we were at it, we used more recent packages and made some improvements to code and design. The "FixMix'" has been successfully tested and succeeded in supplying an entire household with Cocktails.**

Keywords— smart cocktail mixer, mixed drinks, cocktails, automation, FixMix

I. Introduction

In recent years, advancements in technology have made various aspects of our daily life more comfortable, through transforming routine tasks into automated processes. While smart solutions in areas such as brewing coffee and preparing food are already widespread, the world of mixology has remained nearly untouched by the trend of automation. This paper describes the construction and upgrade of an already existing but lacking blueprint called "Smart Bartender" from the "Hacker Shack" team. It is common knowledge that, just like many other things, preparing drinks at home is much cheaper than paying a professional to do it. Unfortunately, there are quite a lot of mistakes amateurs constantly make. The right mixing ratio is especially difficult to achieve and basic recipes are less widespread as one would expect. Aside from that, the task of mixing drinks can be a very time intensive chore. With our project, we aim to achieve a comprehensive solution that not only mixes drinks more consistently than a barkeeper ever could, but also does it in a very efficient way, saving time and resources simultaneously. The accessibility and user-friendliness should be regarded as well, so that anyone can use our product without any former knowledge or introduction. All of this provides a seamless and consistent cocktail crafting experience.

II. State of the ART

There is an existing project by the group "Hacker Shack", that implemented an automated cocktail mixer. It is called the "Smart Bartender" and blends drinks at the push of a button. Their machine is fully functional and can be extended, so more ingredients get blended. There is also the possibility to link with Alexa or other smart home technology. Furthermore, the drinks can be selected via two buttons and an OLED display. The database of drinks can be freely customized and expanded. They built a solid foundation we can expand upon. One thing that should be noted is the fact that their code is about five years old and uses many deprecated libraries.

III. Our Goal

The goal we envisioned was to have a fully working drink pouring and mixing machine with a large integrated beverage library as another addition to the appliances commonly found in every household. Its appearance should fit seamlessly into every kitchen, right next to coffee machines and microwaves, which also serve as size references for this project. Even though it is able to act as an impressive attraction at parties, its strength lies in the everyday use at home, bringing innovation and convenience to every kitchen, bar or social gathering. The aspect of comfort is at the center, on which it delivers through its remote capability. The code and libraries of the projects it is based on will be revised and brought up to date. Additionally, quality of life improvements like adding switches to the power plug, better pumps and a panel for partitioning will be made as well as many other enhancements. With its clean, white case, which is built from wood, the most renewable resource, it will fit into any modern household. But its appearance can be altered at any time, due to the mounting of all electrical components on one easily removable plate, which can quickly be taken into a new customized case. The design features easy access for maintenance purposes and the choice of components allow for infinite upgrade possibilities, which makes it a long-lasting and future-proof companion.

IV. Components

Our construct consists of a lot of different components. Some of these are technical parts related to the function and others are just for the case. The wooden box was constructed using twelve wooden laths. Four of them were 40 centimeters long and the other twelve were 30 centimeters long. Besides that, we used eight wooden plates, two were 40 centimeters times 40 centimeters, six were 36 centimeters times 40 centimeters, one was 36 centimeters times 24 centimeters and the last one was 26 centimeters times 24 centimeters. On the bottom side of the top plate is a 3d printed turbo funnel mounted. On the bottom plate of the case, we attached four rubber feet.

Of course, there are also many technical components directly linked to the functionality. The most important piece was the raspberry pi, serving as the heart of our creation. Other parts were an eight-channel relay, a twelve volt switching power supply, a five volt regulator, different jumper wires (male to female, female to female and male to male), three peristaltic pumps with food grade silicone tubing, a momentary push button switch, an OLED display, some high voltage, high current rated diodes, a power socket inlet module plug, a power distribution board and a few more wires.

We adapted certain elements from prior projects to optimize our current work. As a consequence we did not use any LED stripes, since they do not have a real benefit besides a more pleasing appearance. Initially, we opted to use only three pumps in our build to assess functionality. However, we designed our setup with expansion in mind. An example for this is our relay because it has eight possible slots. Other major differences were the addition of a power socket inlet module plug, which allowed us to have a power switch on the backside. The final modification was the exchange of a breadboard for a power distribution board because breadboards tend to get loose and are not optimal for permanent solutions.

V. Design

The appearance and functionality of the case shown in the template was altered and upgraded quite considerably. To begin with, all pumps were relocated to the inside instead of the back, resulting not only in a cleaner outer surface, but also an enhancement in safety. This design change eliminates external wires, reducing potential safety risks and offers protection for the components. In addition to that, the hoses and wires are covered by a wooden board, while the pumps themselves are still visible and can be checked for functionality. All electrical components are mounted on a single wooden panel, which is placed in the electronics compartment under the bottom floor of the dispensing area, which can be flipped open like the hood of a car. This allows for easy access to maintain and upgrade the system, as well as switching out the case if preferred. The input of the power supply has been expanded to include a cold appliance plug and a power switch. The template recommended the use and implementation of led stripes for decorative purposes, which we decided not to include, in order to keep the costs low and the wiring less cluttered. A flowerpot coaster serves as an indicator for the glass, while a 3d printed turbo funnel gathers and bundles the tubes dispensing liquid to prevent spillage.

VI. Chasis

We built the case out of wooden laths, encountering a minor challenge as we needed to strategically position the screws for stability while avoiding interference between them. The panels were added much later to the framework since we wanted to leave as many sides open as possible to make it easier to work on the technical parts inside. To accommodate the display cables and ensure the display was centered, we drilled a hole in the wooden bar at the front. Moreover, there was a knothole in the front, which we had to remove since the front panel would not close completely. Later on, we encountered a problem that required us to cut out a piece of a wooden lath in the back due to the dimensions of our power socket inlet module plug. Before attaching certain panels, we needed to prepare them by drilling holes to accommodate cables, tubing, buttons, and pumps. Others had to be cut into shape, such as the one we used as a second floor. We removed the corners to seamlessly fit into the framework and adjusted the backside, so the cables connected to the pumps could reach the electronics and were not visible to the user. Additionally, we added some rubber feet to the bottom for stability and noise reduction.

VII. Hardware

Initially, all the components were connected and laid out loosely to be checked for functionality. Most were easy to connect with a variety of cables, which we had to figure out ourselves and could be stripped and plugged in or clamped. Some components had to be soldered, for instance the cables leading to the pumps and the ones that had to be soldered to the circuit board of the voltage regulator for the Raspberry Pi. The high voltage, high current rated diodes, bridging the power cables leading to the pumps, had to be soldered on as well. This process significantly improved the team's soldering abilities. In most cases, cable colors of the individual cables were matched to the ones shown on the circuit diagram. Only the voltage regulator which is supposed to supply the Raspberry Pi with 5 Volt instead of 12 had to be adjusted manually by turning the potentiometer screw clockwise. According to the circuit diagram, one cable required splitting after the voltage regulator, but we were able to solder it right to the circuit where the other cable is connected, which worked just fine. Connections were frequently tested using a circuit analyzer.

VIII. Software

Initially, we planned to build an automated cocktail mixer, based on the "Smart Bartender" with manual input. However, as we encountered some issues with the OLED display, we explored alternative options and came across a web interface called "PiTender", which we modified to our purposes. Fortunately, we successfully resolved the issue with the display. As a result, the FixMix can now provide users with the flexibility of choosing between manual input via display and buttons or input through our easy-to-use web interface. The following describes implementation of the manual input, the web interface, and the challenges we encountered during this process. While we used JSON files for configuration, python serves as the primary language for the main aspect of the project.

A. Manuel Input

For our manual input solution, as already mentioned, we adapted the project “Smart Bartender”. At the beginning, we had to enable SPI in the Raspberry Pi configuration, for the OLED display to function properly. After this step, we also had to add the following 2 lines of code, “i2c-bcm2708” and “i2c-dev” in the i2c config to ensure that i2c will work without any issues. Afterwards we need to set up the library for the OLED screen, but since the project relies on a deprecated library, we are utilizing the “Adafruit Python SSD1306” package. Consequently, we had to update the function calls according to the new module.

Continuing with the installation of the “requirements.txt” file through the command line with “sudo pip install -r requirements.txt”. This results in an outdated installation of “RPi.GPIO”, we proceeded to update it to the latest version 0.7.1. This Adjustment was because the older version led to errors during the executing of the code. In addition, as mentioned earlier, we completely cut the code for controlling the LED strip, because we decided to cut the whole LED part.

Here is a brief overview of the files within the FixMix:

1. bartender.py
    - The file “bartender.py” is the heart of the program by seamlessly interacting with all other files. It imports the pump settings from the “pump_config.json” and, if necessary, writes changes into it, displays

the “FixMix” menu on the OLED display, handles the input of the two buttons, determines, based on the pump configuration, which cocktails can be mixed, and subsequently activates the corresponding pumps. Most of the changes were made in this file.

2. drinks.py
    - The “drinks.py” file contains two object arrays, the “drinks_list” and the “drink_options”. The “drinks_list” array stores values representing cocktails, specifically their “name” and “ingredients”. In this file we included our preferred drinks along with the required ingredients.
3. menu.py
    - The “menu.py” handles everything related to the menu we display on the OLED screen. No changes were made in this file as it worked out of the box and met our expectations.
4. pump_config.json
    - The “pump_config.json” file stores information about each pump and its associated pin, as well the current active ingredient. We trimmed down the JSON file to include information for only 3 pumps, because our FixMix only uses this quantity.
5. requirements.txt
    - The “requirements.txt” file installs the latest "RPi.GPIO" version.
6. read.txt
    - The “readme.txt” includes helpful information about the project. We have added explanations regarding the new OLED library and a quick guide on how to use the FixMix.

To launch the manual input of the FixMix, run the following command “sudo python bartender.py” within the FixMix directory. To immediately stop the process use “ctrl + c”.

B. *Input via web interface*

For the input via web interface, as already mentioned, we adapted the project “PiTender” from “nebhead” on GitHub. Due to the OLED display issues we encountered, we were looking for an alternative to test our entire setup and came across the “PiTender”. This version removed the buttons and display completely, aligning perfectly with our requirements. It uses a flask web interface, with gunicorn and nginx to proxy the web requests. The project runs two processes simultaneously, the app.py handles the web routing and the control.py handles the GPIO inputs. They communicate with each other through JSON files. The installation was straightforward using an auto-installation file but comes along with an issue we will address in detail later.

What have we changed? Firstly, we changed the UI into a Dark Mode, as the FixMix is likely to be used most often in the evening or at night, providing eye comfort. Additionally, we replaced the carousel that displayed mixable drinks sequentially with individual cards presented in a vertical layout. This allows users to quickly see the variety of drinks the FixMix can provide. Furthermore, we replaced the old simple cocktail images with appealing photorealistic ones. All menus were also translated into German. Another change we made is reducing the number of pumps in the source code to three. The web interface launches automatically upon starting FixMix. To modify its appearance, you'll need to edit the HTML files in the templates directory and restart the system afterwards.

C. *Best of both worlds*

After successfully testing our setup with the web interface and resolving the display issue with manual input, we decided to combine both projects, so we can provide our FixMix users the best user experience possible. This integration enhances the overall functionality and convenience of the FixMix, ensuring a seamless and efficient cocktail mixing experience for our users. Due to the fact that both input methods use JSON files for their pump configurations, a configuration file could be implemented which synchronizes the configuration between both systems. This approach would enhance consistency and streamline the management of pump configurations across both the manual and web-based input methods.

IX. Functionality

The main functionality of this device consists of dispensing liquids from containers into a glass and mixing them in the process. This action is carried out by peristaltic pumps, controlled by a Raspberry Pi and relay module, either via the web application or the buttons on the device itself. The user is able to track the progress of dispensing by observing the progress bar either on his phone or the machine itself and plan accordingly. Besides mixing cocktails, the FixMix is capable of cleaning itself by pumping water through all its tubing. This can be executed by placing the intake ends of the hoses into a container filled with water and selecting the provided cleaning process. Due to excess space on the relay module and power distribution board, FixMix can be expanded to include up to five more pumps. Furthermore, users have the flexibility to change the configuration of assigned beverages directly on the device itself, as well as using the web application to do so. The user can always add more drinks to the integrated recipe book. Dispensing non-alcoholic beverages and shots of liquor are also possible. Though, FixMix will show only the recipes it is able to prepare with the current number of pumps attached and their configuration.

X. Issues we faced

During the project we faced some significant challenges, with the majority of time devoted to addressing display issues. The first OLED display we used had an incompatible hardware chip. While we could have potentially worked around the issue by rewriting parts of the code, it would have been very complex and would have taken a lot of time. Instead, we purchased another display which had the correct chip but used the i2c technology instead of SPI. We considered fixing the issue by resoldering parts of the display. But instead, we decided to get another display, which would have been the right one, but it got damaged pretty badly via shipping. Subsequently the fourth display finally had all the correct requirements and still didn’t work as intended. Upon investigation we managed to locate the error: one jumper wire

was defective. After changing the corrupted wire, the display finally worked as intended.

Another small issue occurred after we installed the pumps, one of them did not work at all and forced an error which shut down our whole system. Later we found out that the wires were attached to the wrong ends on one pump. After rewiring, the problem did not occur again and we could proceed.

The web interface also brought its own set of inconveniences. The changes we made in the template files did not show in our browsers, because the debug mode, which should enable previews by displaying changes in real time, did not work. We managed to find out that we were working in the wrong directory, because the installation file copied the project it had already downloaded to a completely different folder instead of using the one it already provided. After that we noticed, this folder was protected and every change had to be made by console commands as an administrator, which disrupted the workflow considerably.

XI. How to use

We made it very easy to use the FixMix. You can decide whether you want to use it manually via the buttons and the display or through the web interface. Regardless of your choice, always use a glass with a capacity of 250ml, otherwise it may overflow.

A. Manually

By pressing the left button, you can cycle through the available drinks. Once you find your desired drink, you can confirm the input with the right button and FixMix will mix your selected drink. Additionally, you can navigate through the drinks and will see the possibility to configure, by pressing the right button you get to the configuration menu. In that menu layer, you have the option to modify the ingredients linked to the pumps or initiate a cleaning process.

B. Web application

In the web application, you can perform the same actions by navigating through a website instead of pressing a physical button. At the top right of the interface is a burger menu that presents the three main pages: the “Rezeptbuch”, where you can add new drinks, edit existing ones and manage the ingredients. The “Cocktailkarte”: here you have the option to have your desired drink mixed. And the “Einstellungen” page, used for configuring the pumps, setting the pump flow rates and starting a cleaning procedure for the tubing and the funnel.

XII. Tests

During the testing phase with water and orange juice, everything worked as intended. The result had the correct amount of the ingredients. A problem occurred in the second testing phase with water and a carbonated energy drink. The glass was not as filled as expected and after repeating with the same ingredients we saw bubbles in the tubing that came from the energy drink. This led us to the conclusion that carbonic acid was the cause. We matched the carbonated energy and water to determine the difference and calculated the conversion factor of 1.6 based on the results. We adjusted the amount of energy drink in the recipe accordingly by 1.6 times and it worked as intended. Therefore, we corrected all the recipes that involved any sparkling ingredients like lemonade for example. After these adjustments, we tried a recipe that had only coke in it and the glass was correctly filled.

After using the FixMix we also tested the cleaning program. There we put water into all the tubing and started the cleaning process. After it was done, we removed the water and started it again so all the water would get out. The result was incredible, because no visible remarks of orange juice or energy drink were visible and almost nearly no water was left in the tubing.

XIII. Experiment

Since we were using stronger pumps than the original project, we had to recalibrate the flow rate in the code to make sure that exactly the required amount of liquid will be dispensed. We determined our new value by experimenting with different values. We measured the results using a setup consisting of two containers, a scale and some water.

Our goal was to achieve a liquid capacity of 200ml with marginal fluctuations.

TABLE I. Flow rate adjustment

Flow rate value	Result in milliliters
60/100	290
85/100	410
50/100	327
50/150	157
85/150	273
60/150	200

The flowrate value consists of 2 values. The initial entry was "60/100". First, we began testing changes in the first value: 60/100, 85/100 and 50/100 all exceeded the 200ml mark.

After closer inspection of the setup provided in the template, we discovered that the original pumps are rated at a maximum speed of 100ml per minute, which is represented by the second value. Consequentially we had to change this value to 150, since our pumps operate at a maximum speed of 150ml per minute. We tested the same values with the new parameter in place and received results much closer to our objective.

At last, we determined that a value of "60/150" achieved the desired results.

After running this setup multiple times, we obtained results with negligible differences, within a range of ±5ml at an operating time of 60 seconds.

XIV. Future revisions and Conclusion

We managed to achieve most of the goals we set ourselves and produced a working automated bartender. The possibility to control it manually and via a web interface has been realized. The design got enhanced with the addition of a power switch and new positions for the pumps, to contribute to an improved overall aesthetic appearance. Furthermore, we built a database with many drinks and updated deprecated code libraries, to ensure future proofing. The FixMix was a success and is ready for parties or private use.

Looking ahead, there is potential for expanding this project. This is shown by the pumps for example, since there are open slots in the relay to attach five more. Introducing an LED stripe could enhance the aesthetic look and provide better lighting inside the mixer. On the software side, refining the user interface or resolving the issue, where the manual input and the web interface use two separate JSON files with their own pump configuration as well as syntax, leading to no synchronization between the configurations. To address this problem, both 'control.py' and 'bartender.py,' which manage the JSON input, require adjustments to ensure they read and write using the same objects and values. The wooden case is not perfect as well, considering alternative materials with better water resistance than wood could be an upgrade.

XV. Sources and attachments

PiTender:

https://github.com/nebhead/PiTender

Smart Bartender:
https://github.com/HackerShackOfficial/Smart-Bartender

Other attachments like circuit diagram and screenshots are at the document folder:

https://github.com/Schnigglas/fixmix/document

Dancing Nao Robot

Andreas Popp
Medienifnormatik
Hof University of Applied Science
Hof, Germany
andreas.popp.2@hof-university.de

Emir Mervan Kanun
Medieninformatik
Hof University of Applied Science
Hof, Germany
emir.kanun@hof-university.de

Ercüment Zorlu
Medieninformatik
Hof University of Applied Science
Hof, Germany
ercuement.zorlu@hof-university.de

Kevin Pertek
Medieninformatik
Hof University of Applied Science
Hof, Germany
kevin.pertek@hof-university.de

The following Project revolves around a Nao Robot that is supposed to use video input to assume poses of people in said video.

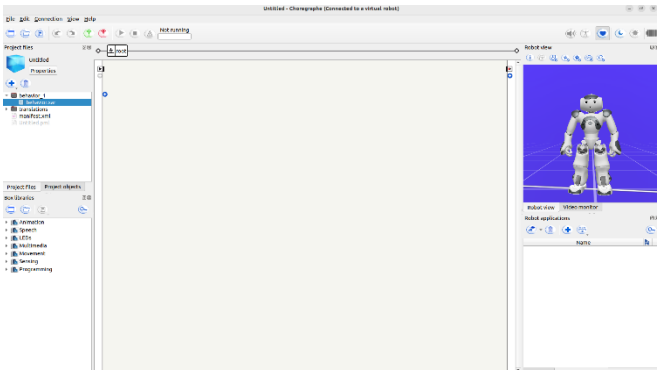
I. Introduction

Our goal was to get the Nao robot to recognize specific songs which lead him to dance a specific choreography corresponding to given song. To reach this goal we set five different milestones. Firstly, we wanted the Nao robot to imitate poses from a picture. Secondly, the robot should imitate poses using a video. Thirdly, save certain dances and perform them on command. Fourthly, the robot recognizes specific songs and dances the according choreography if he doesn't recognize the song, he will make up his own choreography. Then lastly, we had an optional goal where Nao is supposed to make up his own dance synchronized with the beat of the song that is being played.

II. Approach

A. Nao Software

We installed the Nao software on Windows and Mac which is composed of Choreographe, Robot Settings and the NAOqi SDK from the Aldebaran website [1]. We got ourselves accustomed to the Nao Software and tried communicating with it. We also tried used Choreographe to make Nao assume poses. We noticed that the Software was old using Python 2.7 and Choreographe didn't even work on Mac OS with M1 chips at first. After a while Aldebaran rolled out an update to support the M1 Mac OS platform.



B. Working with ROS

We decided to use ROS version 1 which is a Robot Operating System. To use ROS, we needed an old version of Ubuntu, we used version 20.04.02[2] and installed ROS [3]. We first tried to get an understanding of ROS through research. Using ROS came with a lot of problems. Firstly, finding the right versions of Ubuntu and ROS was hard and packages used by ROS were too old and also hard to find. Secondly, the Ubuntu made automatic updates and updated software while installing other software rendering our distribution of Ubuntu useless. We also realized that our NAO robot does not belong to version 3 but version 6 which meant that our version of the NAOqi SDK was wrong because we needed version 2.8.7.4[1].

C. Working with Python

After these issues with ROS, we decided to drop our ideas and use Python instead. Python as a lot of powerful libraries and is a very well-known language for AI and Robotic Applications. With a hint of another group which was working with the Nao. We installed an even older version of Ubuntu, that had Pyton 2.7 preinstalled. All the Group members weren't too comfortable in using Python so we had to refresh our knowledge. After that we had to install NaoQi to be able to program the robot. The installation was quite tricky, due to the very old and flimsy documentations. We managed to get it working after setting the Environment Variables of our Python Version, so that it can load the Qi.
After some trial and error, we managed to connect to the robot and run our first Hello World Program. Thrilled by this achievement after so many frustrating fails, we had some fun with the TextToSpeach function of the library.

D. Computational Vision and Pose Recognition

After our first successes with the robot, we now had to research different libraries for CV (Computational Vision) and Pose Recognition. There was just one real option in this topic, as we used such an old version of Python. We installed OpenCV and tried to get it to work. After several failed attempts and a hint of the professor we dropped this Idea again.

We decided to use the NAOqi on an old Python version but use a current version of Python to get the Pose Recognition running. We reinstalled our Ubuntu to be the most recent Version with Python3 preinstalled and install Python2 separately. Reinstalling the NAOqi and setting the Environmental Variables. After further researches we installed Google Mediapipe. An easy to use and very popular CV and Pose Recognition Tool. After some experimenting,

we managed to run Mediapipe and got our first rigged image and the landmarks of a human skeleton as an output.

After a short break we returned to our project, but nothing seemed to work anymore. We realized that another software called OpenRoberta was running on the Robot. It was installed by a professor to use it on a Project with pupils. We tried to uninstall OpenRoberta but we couldn't manage to do it. After several tries to reach out to the professor and trying to find other ways to uninstall it, we finally got a reply on how to uninstall it.
So, we finally got rid of the program and our Code was working again. We worked on a Jason Parser to save our landmarks in a Json file and plotted them in a 3-dimensional coordinate system with the use of MathPlotLib. We finally had a way to see our Mediapipe result.

E. Calculate corresponding Angles

After that, we now had to translate our Google Mediapipe results to be able to use them on our Nao robot. Our first idea was to calculate the angles of the skeletal joints to pass them to the robot. For that, we had to read some Documentation to understand the complex math behind it [5] [6]. We managed to calculate some easy joints like the knees. But we couldn't implement the other joints as they had more degrees of freedom. The math behind these more complex joints was too hard for us. Even after reading documentations and papers on how it should work the problem was still too complex for us to understand. We then tried to get more examples by using different pictures with varying poses to get a better understanding.

F. Using inverse kinematics

We could After all, we tried using inverse kinematics. First, we read the documentation from SoftBank Robotics [7] and used this information for our project, especially the part about Cartesian control [8]. In this chapter they describe the control of the Effectors of the Nao robot in a cartesian space using inverse kinematics. To gain knowledge about it, we used the example code and tried it with the robot. After that we edited the code for our purposes. While exploring the code we found out that Nao is working with meters and has a maximum range where he can perform his movements. Following that we gave the Robot a T-Pose from one of our team members which he should perform but he did everything but the T-Pose. To test the function, we gave him absolute variables to look how Nao is making certain poses. But changing even the numbers a little bit, Nao went crazy again. Further we thought that not using normalized coordinates for inverse kinematics could be a problem after all too. So, we tried calculations to get the Mediapipe landmarks to fit on the Nao movement. We changed the rotation matrices of our Mediapipe landmarks, normalized them and moving the coordinates to get a result to work with.

III. EXPERIMENTS

The first tests we conducted were on the result of google media pipe. To do this, we used some code that draws the found landmarks on the input picture. This went quite well and we left this in the final code. The "rigged" picture can be found under "*Landmarks/pictures/TPoseRigged.jpg*". This test showed, that media pipe was quite apt in finding the right landmarks in 2d. However, when looking through the output, we quickly realized, that the 3rd Dimension wasn't as easy for media pipe, which is why we only used a constant for this dimension when posing NAO.

The second batch of tests revolved around getting a feeling for NAOs capabilities when using carthesian control. To do this we fed him many different absolute values for each hand and tried to assess where he moved them. After some confusion, we found out that every value was measured in metres and decided to use a ruler to find some example points. Even then we had to try many similar values to get NAO into a T-Pose, because as soon as NAO couldn't move his hands exactly where we told him to, he didn't even try to approximate the pose and instead did something completely different. This will still happen in the final result, which is the main reason, why we would recommend using the angle calculation approach instead of carthesian control. However, as we found some usable values in this experiment, we were able to use them to calculate the scale we had to apply to the normalized landmarks we got from our picture to convert them into NAO's coordinate system.

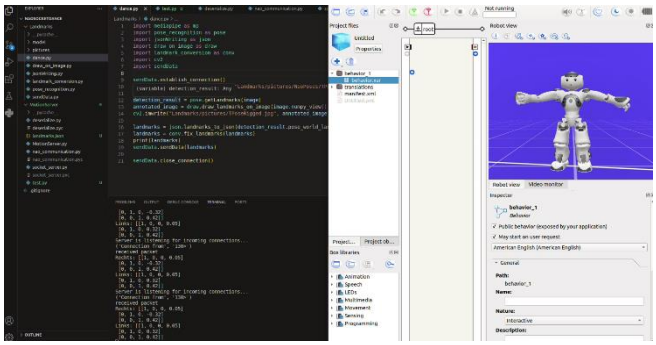
One of the last experiments we did went back to one of our first major problems: The python version mismatch between Mediapipe (python3) and NAOqi (python2.7). As using choreographe for more than it's virtual robot for testing quickly proved to be a major problem, we couldn't use the suggested workaround and instead decided on a server-client approach with the server running on python2.7 and waiting for values on a Unix-Socket, while a clients could be run in python3 to send values to the server. This approach required some serialization to be able to send the data but didn't cause any major problems otherwise.

This means the program can be used in the following steps:

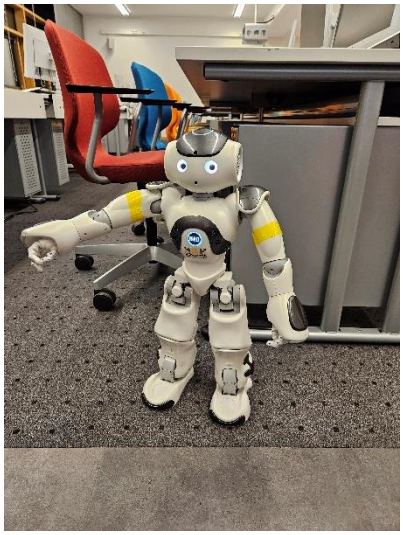
1. Adjust the port and IP-Address in *MotionServer/nao_communication* in the *get_session* Method if needed (Or use a virtual robot from choreographe as per default)
2. Run *MotionServer/MotionServer.py* in python2.7
3. Adjust the image path in *Landmarks/dance.py* to point to the intended input image if needed
4. Run *Landmarks/dance.py* in python3

One of the last problems we had to solve was the fact, that the virtual NAO did the right pose after everything was done, but the actual NAO Robot had problems with it's left arm. We still aren't quite sure what causes this but can only assume that our

NAO actually has a bug. We recorded a video of the virtual robot doing what it was supposed to from which the following screenshot is taken:



In comparison, this is the pose the physical NAO did with the same program:



IV. Conculsion

We first installed Nao software from Aldebaran, then tried using ROS after which we tried communicating with NAO using only Python 2.7 and NAOqi. After that we used Google Mediapipe and CV Pose Recognition to get landmarks from people on pictures and had to calculate the corresponding angles using the given landmarks. In the end we used inverse kinematics using code provided by SoftBank editing it for our purpose.

In conclusion we tried to get a Nao robot to dance but weren't able to. Sadly, there was a lot to figure out and the only goal we were able to accomplish in the end was getting Nao to imitate poses from a picture at least virtually in Choreographe. We accomplished our first milestone. That means that ultimately, we couldn't get Nao to dance or even use video input in the time that this project was concluded. This means that there is still a lot to accomplish for future courses.

For the students that want to approach this project we recommend trying to get behind the math that is required to calculate angles from landmarks which we couldn't do in the time given. Use Mediapipe and don't bother using ROS, you should use the tools provided by Aldebaran such as NAOqi to communicate with Nao. Also our Nao robot might have been broken.

V. References

Sources:

[1] https://www.aldebaran.com/en/support/nao-6/downloads-softwares

[2] https://releases.ubuntu.com

[3] https://www.ros.org

[4] https://developers.google.com/mediapipe

[5] https://developers.google.com/mediapipe/solutions/vision/pose_landmarker

[6] https://temugeb.github.io/python/motion_capture/2021/09/16/joint_rotations.html

[7] http://doc.aldebaran.com/2-8/index_dev_guide.html

[8] http://doc.aldebaran.com/2-8/naoqi/motion/control-cartesian.html

Racecar

Akim Zholdoshbek
Computer Science
Hochschule Hof
Hof, Germany
akim.zholdoshbek@hof-university.de

Yedil Yelkhan
Computer Science
Hochschule Hof
Hof, Germany
yedil.yelkhan@hof-university.de

Aidar Chakiev
Computer Science
Hochschule Hof
Hof, Germany
aidar.chakiev@hof-university.de

Jamilya Tankimanova
Computer Science
Hochschule Hof
Hof, Germany
jamilya.tankimanova@hof-university.de

Albina Zekenova
Computer Science
Hochschule Hof
Hof, Germany
albina.zekenova@hof-university.de

***Abstract*— The paper includes three outlines for research on an autonomous shipment vehicle driven by a Raspberry Pi 4, with a particular emphasis on steering and control in a preset environment. Each iteration highlights the system's capacity to stay on a line, make exact turns, and move utilizing visual processing and a Flask server for remote manipulation and monitoring, regardless the constraints of the hardware. The techniques varied in detail, but they all emphasize the creativity and possible utilization of affordable technology in challenging automation challenges inside unmanned delivery networks.**

Keywords*— *Raspberry Pi 4, Flask server, Vehicle, Jetson Xavier, RESTful , Nvidia , APIs, RealSense D435 depth camera, GPIO , I2C, HTTP, Python, JSON

I. Introduction

Vehicle manufacturers and scientists identically are becoming more and more interested in self-driving technologies. We go into further depth about our completed project, its primary elements, the challenges we had and how we overcame them, and the final product based on the interest in this vital topic and the issues listed below.

II. Problem Explanation

A. Autonomous Vehicle Navigation Challenges

The development of new algorithms and improving the performance of current ones used for automation by vehicles are relevant because testing scenarios for autonomous vehicles are continuously becoming more complex, the expectations for operating such machines in an industrial environment are becoming stricter, and the requirements for systems are becoming tighter. Even if a lot of self-driving car experiments currently are performed out in ideal circumstances and natural settings, under human supervision, there are still several issues to be resolved before establishing a completely autonomous vehicle:

1. For instance, autonomous vehicles navigate using radar and lasers. The radar's basic method of operation is to find radio wave reflections from nearby objects. The car will constantly generate radio frequency waves while it is on the road, and these waves will be reflected off of other vehicles and other objects. The distance between the vehicle and the object is determined by measuring the amount of time needed for reflection. Will an automobile be able to tell its reflected signal from another car's when hundreds of cars are using this technology on the road? For these kinds of devices, global industrialization presents a challenge. In our example, we used the camera - as an advanced kind of radar - to get an image from the camera and calculate the angle of inclination of the line along which the car will travel. And depending on the angle, he sent requests regulating the movement of the car along the line. While computer vision algorithms can analyze images to detect lines and estimate their angles, despite the ideal operating condition of the algorithm, factors such as perspective distortion, lighting and shading conditions can introduce errors into the angle estimation process. Using camera data exclusively to determine the angle of inclination can lead to unreliable navigation solutions, especially in dynamic and unpredictable conditions, when road markings may be blurred or ambiguous. Therefore, it is better to check in the ideal conditions created. However, as it was tested on the yellow line - the path of the machine - on a white background, since there is no particularly bright contrast, we can confidently say that our machine can correct its work despite such an obstacle in image processing.
2. To execute maneuvers without overshooting or understeering, precise control algorithms are essential. Dynamic aspects including tire grip, road conditions, and vehicle speed must be taken into consideration by these algorithms. To maintain stability and maneuver through curves with ease, algorithms that can adaptively modify steering inputs based on real-time feedback from sensors must be developed. Precise turns are necessary in industrial settings to efficiently navigate vehicles around obstructions and confined areas, optimize output, and reduce downtime—all of which contribute to a competitive advantage in the marketplace. Our machine is programmed so that, depending on the angle of inclination of the line from 0 degrees relative to the camera in the vertical plane, the robot turns the wheels in the right direction - to the right or to the left.
3. Driving a car over a pre-made map is a complex task that requires both the ability to make judgments in real time to go to a destination and a deep awareness of the vehicle's position within a bigger area. This task involves a number of complex elements, including as path planning, spatial awareness, and flexibility in the face of route variations. The vehicle is ready to explore a square map with four locations that has been prepared, remembering its current location.

4. Handling hardware restrictions is a major problem when navigating autonomous vehicles, and this challenge is exacerbated in projects where initial hardware selections are met with obstacles that require modifications. Due to hardware problems with the Jetson Xavier, we had to switch to the Raspberry Pi 4 for our project. These incidents highlight how crucially hardware capabilities and the overall system design interact. The complexity of algorithms that can be implemented and the speed at which computations can be completed are directly influenced by the hardware platform's processing capability. In our instance, the Jetson Xavier's processing power constraints forced us to reevaluate our computing needs and the necessity for a more scalable and affordable substitute.

B. *Previous works*

Daniel Hanik's paper "Aufbau und Implementierung eines Elektrofahrzeugs mit REST API zur Steuerung und Kamerabildübertragung" offers a thorough walkthrough of building an electric car that has a camera for real-time picture transmission. Here is a thorough synopsis based on the material that makes up 74% of the document:

The project's goal is to create a camera-equipped remote-controlled car, with an emphasis on component assembly and setup for vehicle control and camera connectivity. The objective is to build a vehicle equipped with a camera, combine the camera, and set up a server for RESTful control.

Using Nvidia's SDK Manager for setting up the operating system simplifies setup by installing all required software elements. It involves choosing the desired both software and hardware elements, setting up the operating system, libraries, hardware function APIs, additional tools, and development assistance samples. To operate the car, a Python script was developed that makes use of the Adafruit_Servokit library to modify the servo motor degrees while controlling the speed of the Microcontroller. It details the values for forward and backward movement as well as the tilt settings for turning.

The task at hand in this document serves as an illustration of prior work and focuses on setting up the Jetson Xavier NX to function as a vehicle's primary control unit. Emphasize the incorporation of a camera, the construction of mechanical parts, including the vehicle and a 3D-printed case, and the development of a Flask server to allow control of the vehicle and live picture signal extraction.

III. CONTRIBUTION

Our project is distinguished by a strategic shift to the Raspberry Pi 4 as the cornerstone of our autonomous racecar design, which is illustrated by its exceptional adaptability and cost effectiveness. The desire to build a system that balances affordability with strong computational capacity, which will enable autonomous technologies to be made available to the wider public motivated this choice. The Raspberry Pi 4, with its high computing power, meets the needs of real time image processing and vehicle control well.

Our project includes significant algorithmic developments in line following and turn execution, pushing the limits of what is possible in autonomous navigation.Our navigation system is built on a sophisticated algorithm capable of interpreting live video feeds and determining the racecar's route with remarkable precision. The vehicle maintains a precise course by estimating the slope of the line and dynamically modifying the steering. Turn execution is also upgraded; the system recognizes specified markers or orthogonal lines and executes sharp 90-degree turns with the reliability and precision that set a new bar for autonomous vehicle navigation.

A. *Software Architecture and Remote Control.*

The autonomous racecar project's software design is built around a Flask server running on the Raspberry Pi 4. This server is carefully setup to handle HTTP requests for two key functions: vehicle control and image retrieval from the onboard camera. The Flask framework was chosen for its lightweight design and ease of deployment, resulting in a responsive and efficient interface for real-time interactions with the racecar.

The server configuration entails setting specific endpoints for command receiving (e.g., moving ahead, turning, and stopping) as well as broadcasting live video feed from the camera. This architecture allows for a modular approach, in which the Raspberry Pi 4 works as a direct interface to the car's electronics while also serving as a gateway for external commands and data processing.

One of the most significant advantages of this Flask server integration is the ability to perform remote control and compute offloading. By detaching the computational heavy lifting from the car's onboard system, the concept takes advantage of external computers' more powerful processing capabilities. This design choice improves the system's overall efficiency and capabilities, allowing for real-time image analysis and complicated decision-making processes without taxing the Raspberry Pi's resources.

The strategic decision to move computing work to an external laptop significantly improves the project's navigation capabilities. This method overcomes the Raspberry Pi 4's computing limits, allowing for the implementation of more advanced image processing and decision-making algorithms. For example, the external laptop can do complex computations such as evaluating the video stream to detect lines, interpret signs, and make navigation decisions, which are then transmitted back to the Raspberry Pi for execution.

B. *Hypothesis and Goal*

- Our goal is to create an adaptable system that can be readily adjusted or expanded to satisfy diverse demands or to investigate different elements of autonomous navigation. Our project's hardware and software components underscore the need of adaptability. We ensure that different sensors and components may be easily integrated, as well as that programming languages, frameworks, and algorithms are flexible.
- One specific goal is to create algorithms that allow the racecar to detect and follow a course with high accuracy, regardless of external factors such as lighting changes or surface textures.
- Another important requirement is to make exact turns, particularly 90-degree turns, which are necessary for navigating specified courses or surroundings. The goal

is to precisely detect indications for these turns and ensure precision in control mechanisms so that the vehicle can navigate effectively without human assistance.

C. Methodology

System Architecture Overview:

- Raspberry Pi 4: The Raspberry Pi 4 is the core of our self-driving racecar, acting as the central control unit that interfaces with the vehicle's electronics. This includes controlling motors and sensors through GPIO pins and the I2C interface. Raspberry Pi 4 was chosen because of its good blend of processing power, cost-effectiveness, and strong community support, making it a perfect platform for creating and experimenting self-navigating solutions.
- Flask Server: A Flask server is installed on the Raspberry Pi 4, serving as the primary communication gateway between the racecar and external computing resources. The major functions of this server include receiving picture data from the onboard camera and processing HTTP requests for vehicle control directives such as steering and acceleration control.

Integration with External Computing Resources:

- To improve the racecar's navigation skills without sacrificing operating efficiency, complex computational duties are delegated to external computing resources such as a laptop or desktop computer. This deliberate move allows for more advanced image processing and decision-making algorithms that outperform the Raspberry Pi 4's computational capabilities.
- The Raspberry Pi 4's established communication protocol with the external computer provides rapid and secure data exchange. Commands are transmitted and received in a structured format, often JSON for commands and video streaming protocols, with required security mechanisms such as authentication included to protect command requests.

Image Processing and Navigation Logic

- The camera's integration with the Raspberry Pi 4 is essential for recording real-time video, which is required for navigation. The video data is sent to an external computer and processed using particular algorithms to recognize lines, understand visual cues for turns, and aid in map navigation.
- The algorithms created to process the picture data form the basis of our navigation logic. These algorithms are intended to precisely compute the slope of a line for line following, identify specific markers or configurations for turning, and determine the vehicle's position on a map for effective navigation.

System Operation Flow

- The operational workflow of the system is streamlined, beginning with image capture and ending with command execution. The Flask server is important for receiving processed commands from the external computer, which are then translated into actual actions by the Raspberry Pi 4, effectively driving the racecar's motors and steering mechanism.
- Our system uses feedback mechanisms and dynamic adjustments to provide the highest level of navigation accuracy. These may include real-time steering corrections based on continuous image analysis or changes to navigation algorithms in reaction to performance data, guaranteeing that our self-driving racecar can negotiate difficult situations with precision and reliability.

D. Implementation Steps

1. Switch from Jetson Xavier to Raspberry Pi 4.

The Nvidia Jetson Xavier was initially used in the project due to its superior processing capabilities, which were appropriate for challenging image processing jobs in autonomous navigation. However, we ran into substantial hardware obstacles, such as car communication issues, power efficiency constraints, and integration challenges. These challenges prompted looking for a more appropriate and dependable solution.

After considering several options, we chose the Raspberry Pi 4 for its low cost, adequate processing capabilities, huge community support, and hardware compatibility. To fit the Raspberry Pi 4's design, we needed to make significant adjustments to our system setup and software configurations. We also had to change peripheral connections to enable flawless integration with the vehicle's existing components.

Switching to the Raspberry Pi 4 successfully cured our original hardware concerns, improving the project's overall dependability and performance. The system became more stable, with increased connectivity and power efficiency. However, this transfer demanded certain changes, such as adjusting processing power assumptions and changing the development environment to meet the Raspberry Pi 4's specifications.

2. Development of the Flask Server

A lightweight, efficient server was required for real-time control commands and picture data coming from the car's camera. The Flask server was chosen because to its ease of use and efficacy in developing RESTful web services, allowing for easy communication between the automobile and external computing resources.

On the Raspberry Pi 4, we installed the Flask server and used tools like Flask-RESTful to easily create APIs. Endpoints were configured on the server to receive control commands (e.g., steering, acceleration) and broadcast camera images. This configuration used JSON for command data formats and established protocols to ensure secure and efficient data transport.

The Flask server greatly improved communication between the external computing resources and the Raspberry Pi 4. It enabled more responsive and dependable racecar control, as evidenced by real-time picture processing and command execution, confirming the system's increased operating efficiency.

3. Algorithm Design for Navigation.

Developing strong algorithms for line following, turn execution, and map navigation was important for attaining precise and autonomous racecar control. These necessary navigation functions are critical to the vehicle's capacity to navigate complex settings independently.

- Line Following: We created algorithms that used images from the onboard camera to detect lines and determine their slopes. Based on these estimates, the system modified the steering to keep the automobile on course.
- Turn Execution: The system is programmed to recognize specific markers or configurations that indicate the necessity for a 90-degree turn. When detected, it properly regulated the vehicle's turn.
- Map Navigation: Algorithms were created to understand the car's position inside a predefined map and make real-time navigation decisions to reach specified places while taking into consideration the vehicle's current location and the layout of the environment.

These algorithms greatly enhanced the racecar's navigation accuracy and dependability. Tests revealed improved precision in line following, successful execution of 90-degree turns, and good navigation via predetermined routes, demonstrating the algorithmic approach's usefulness.

IV. Conclusion

Our research paper summarizes the experiment's approach, methods, and important findings, making judgments about how well it performed in fulfilling its goals, its consequences for navigational autonomy, and its potential effect on the field. Furthermore, it advocates determining future research areas, implying possible enhancements, and developing new features such as broadened directions algorithms or cooperation with other technologies, emphasizing the document's goal of contributing totally to the field of robotics and autonomous vehicle navigation.

References

[1] Daniel Hanik, “Aufbau und Implementierung eines Elektrofahrzeugs mit REST API zur Steuerung und Kamerabildübertragung”, Hochschule für Angewandte Wissenschaften Hof, 15.09.23.

[2] Daniel Hanik, “Entwurf und Umsetzung einer Smartphone-App zur Steuerung eines Fahrzeugs mit Echtzeit-Kamerabildübertragung unter Einsatz von KI”, Hochschule für Angewandte Wissenschaften Hof, 15.09.23.