RoboCupJunior Rescue Line 2023

Team Description Paper

Team BitFlip

Abstract

Our robot competes in the Rescue Line sub-league and is the result of years of iteration. The two main capabilities that set the robot apart are the use of computer vision and AI, which allow it to perform each task in the discipline. The computer vision capabilities (line following, intersection detection and rescue-kit detection) have been in development since the 2019 season. Since 2022, we use AI to detect the victims and the entrance to the evacuation zone as well. The basic configuration of this year's robot consists of four driven wheels, a single tiltable camera mounted high up at the front of the robot, an arm at the front, and victim and rescue kit storage on top. The "brain" of the robot is a Raspberry Pi which interfaces with the rest of the hardware through a custom-made PCB. The robot has seven motors which are controlled by a separate microcontroller and only two sensors, including the camera. All the homemade parts are modeled in CAD and 3D-printed. All software is written in C++ with additional tools in Python.

1. Introduction

Our team consists of Sven, Jan-Niklas and Lukas, each having a certain role within our team. Through years of collaboration, we finally found a workflow that works for us while everyone still is able to bring in their ideas to the development of our robot.

While Jan-Niklas is mainly responsible for everything related to the organization and coordination of a successful RoboCup season, Lukas developed most of the electrical and mechanical parts of the robots and currently runs our YouTube channel as well as our website.

As Sven no longer lives in the same city anymore, he has not been involved in everything related to hardware development, but even more so in the software that differentiates our robot from the competition. From neural networks to computer vision, Sven has developed most of the software-related challenges we have tackled in the past. Intensive testing as well as adjusting some parameters etc. was done by Lukas, mostly in video conferences with Sven which guaranteed rapid progress due to direct communication.

2. Project Planning

Since our team came second in last year's European Championships and this could be our last participation, our goals for this season were quite ambitious which is why we invested even more time and effort than usual. Furthermore, the experience from previous seasons increased the development speed as we all knew our roles within the team and we did not have to work out how we could contribute to the robot and split things up. Despite this, we already knew what software we wanted to use (e.g. CAD software, PCB design software, common libraries) and were therefore able to start developing a completely new robot just weeks after the previous season had ended.

One thing that we definitely wanted to use even more were neural networks which could also be described as an already-reached milestone since we drastically improved the performance and accuracy of the silver neural networks as well as the neural networks for victim detection. We also made use of a third neural network for finding the evacuation point and redesigned our whole chassis to lower the center of mass even further. Because we had some problems overcoming the speed bumps in the last season, we also decided to use four instead of two DC motors which solved the problem immediately and also allowed for more precise movement and higher torque, helping to climb the ramps and seesaws. We also use a slightly different approach for line following

which not only minimizes rapid oscillating when following straight lines but also allows for an increase in general speed.

3. Hardware

Below is a basic diagram of the robot's hardware. The Raspberry Pi runs the main program and acts as the main controller. The only input it receives is from the two sensors. It sends motor signals to the Motor Controller using UART. The Motor Controller runs a second program, designed to be very simple with as little functionality as possible. It controls the four drive motors, the arm and gate servos, the gripper motor as well as our sensors.

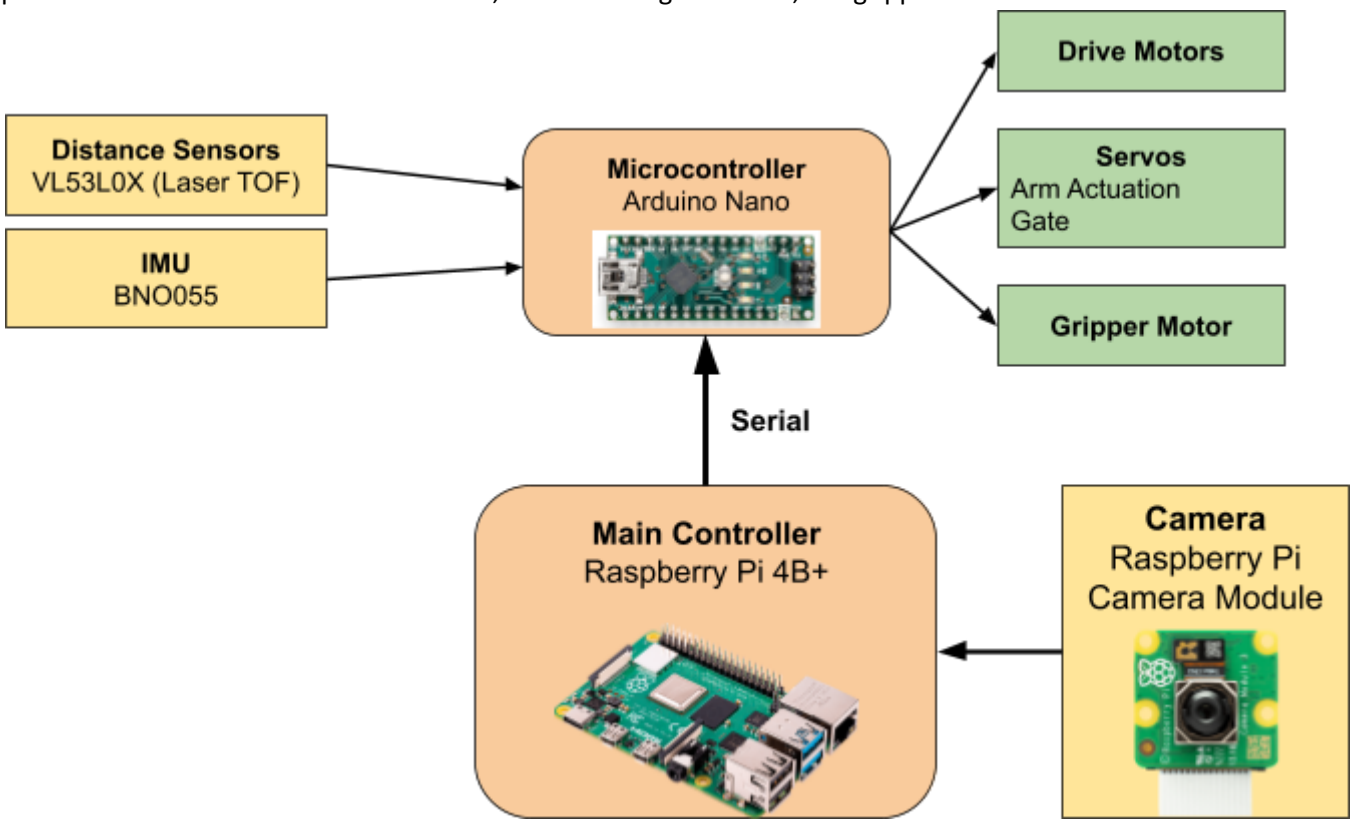

Figure 1: Basic Hardware Diagram

3a. Mechanical Design and Manufacturing

The heart of the robot is the 3D-printed chassis. Mounted on the bottom are four identical 3D-printed motor mounts, which have gone through several iterations since the first design in 2021 to solve problems such as deformation due to the robot's weight. All four wheels are powered by 12V DC motors (a big improvement over last year's front-driven robot when it comes to speed bumps). The front tires are LEGO ones, which have proven to have good traction, mounted on two 3D-printed rims, which are much narrower than the aluminum mounts we used with the Lego rims last year. The back wheels are omni wheels that can slide sideways while still driving the robot forwards. This results in a center of rotation of the robot around the front axle, enabling the robot to follow the line more accurately because its field of view is very close to the center of rotation.

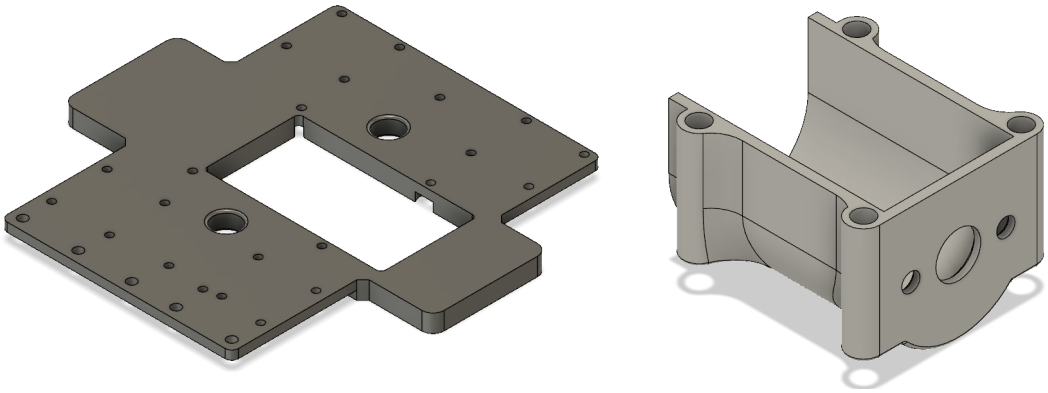

Figure 2: The chassis (left) and one of the four motor mounts (right) in Fusion 360

Our custom main PCB is mounted on top of the robot. It has mounts for the Raspberry Pi, and the Nano as well as other electrical components such as the H-Bridges, buttons and power supply. The mount for the two Li-Ion batteries is soldered to the bottom of the PCB and protrudes through a hole in the chassis. Together with the four motors this gives the robot an extremely low center of gravity (a big improvement from our 2022 robot, which would sometimes tip over on the seesaws).

The front of the robot has a mount for a powerful servo that tilts the arm. The arm of last year's robot was mounted at the back. This meant that the robot had to turn around to pick up the victims and the rescue kit, which was not always possible. The arm's plane of rotation is tilted 10° to allow the arm to clear the central camera. The grippers were originally driven by unreliable servos which were prone to failure when pushing against the walls of the evacuation zone. For the newest iterations, the gripper is actuated by the same type of electric motor used to drive the wheels, which is much more robust due to its full-metal gearbox.

Next to the servo is a single HC-SR04 ultrasonic distance sensor. On top of the arm servo mount is another servo that tilts the camera mount. Last year we used two cameras, one looking down for line following and another looking further up for victim detection. The latter camera was a USB camera that could only take about one picture per second. This, together with more flexibility when approaching victims, is the reason why we switched to a single, tiltable camera for this year's robot. Below the camera is an LED light that helps remove shadows from the image and equalizes lighting conditions.

Above the electronics is a storage tank for victims and the rescue kit, which is also 3D-printed and bolted to the chassis. The victim storage takes up the right side of the upper part of the robot, which leaves little space for a handle. The handle is also 3D-printed and bolted to the left side of the robot, Integrating the handle while making it strong enough was a challenge. We ended up strengthening the chassis and mounting the handle to it using two large bolts. The handle also houses the power switch and start/stop button that lights up when the robot is ready to start.

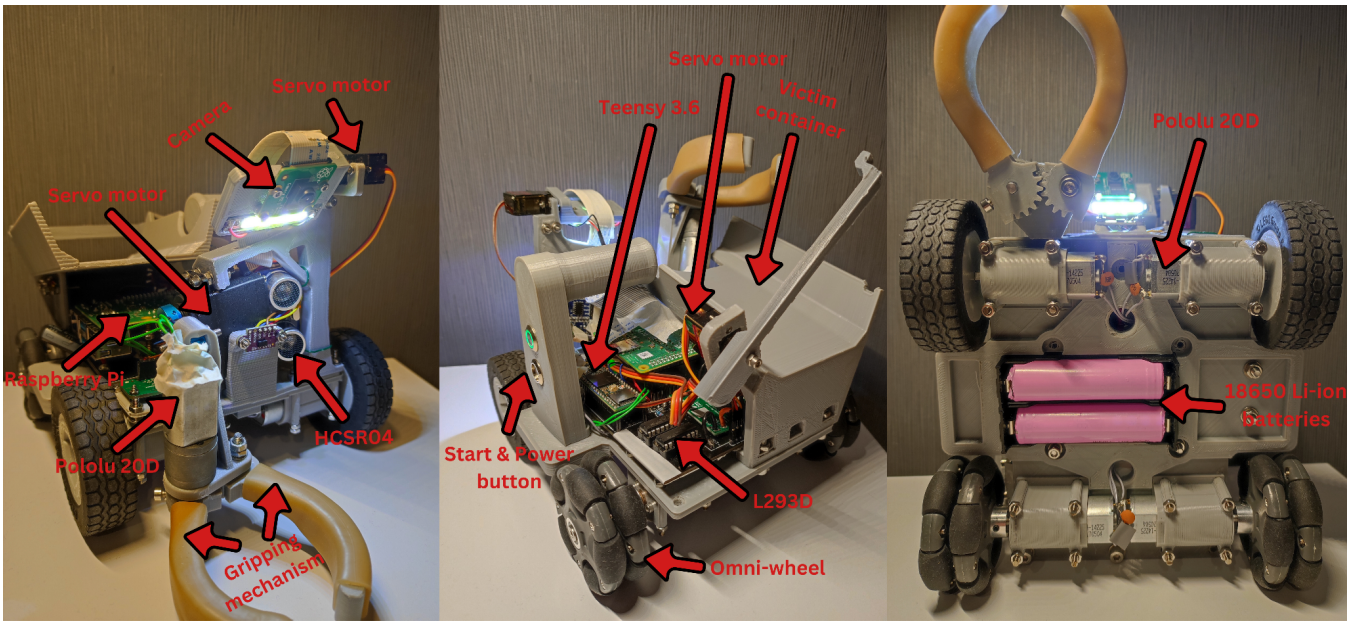

Figure 3: Overview of the robot

3b. Electronic Design and Manufacturing

Regarding the general hardware design of our robot *(Figure 1)*, we use a Raspberry Pi 4B that is responsible for image processing and general program flow and could therefore be described as the brain of our robot. The robot also has an Arduino Nano in a Master-Worker configuration using UART over USB that receives simple motor commands to control our actuators and can report sensor values back to the Pi. In previous years, we did not rely on a separate microcontroller for driving motors, but since the Raspberry Pi does not feature enough hardware PWM channels, we decided to add a separate microcontroller. The result was a large improvement compared to the software PWM we used for last year's robot. The motors can now be driven at much lower speeds and more accurately.

While we try to follow the simplistic "vision only" approach wherever possible due to more flexibility and independence from unreliable sensor measurements the robot still has two VL53L0X laser time-of-flight distance sensors to detect the obstacle and for easier navigation in the evacuation zone as well as a BNO055 IMU.

Our custom PCB we ordered from a professional manufacturer is powered by two 18650 Li-Ion cells connected in series. Their voltage is converted to 5V using a step-down module *(FEICHAO 8A UBEC)* to power the electronics and servos. To supply our four brushed DC motors *(Pololo 20D)* with constant 12V, we also use two step-up modules *(Pololu U3V70F12 Boost Regulator)*.

To control the drive motors as well as the gripper motor, we make use of H-Bridge ICs (L293D).

The development of the PCB started by noting the things we learned from the electronic design of the robots of the last two seasons. We started testing parts of the electronics on a breadboard. Then, we designed the PCB in EasyEDA and had our design critiqued by an external professional as well as with software tools. After manufacturing, we began by testing the circuits with a multimeter and started installing components. Before mounting the Raspberry Pi, we tested the power supply, the motors and the motor controller.

We tested the electronic system with all the actuators and sensors before assembling the robot. During this phase, we also wrote and tested the parts of the software that interface with the hardware.

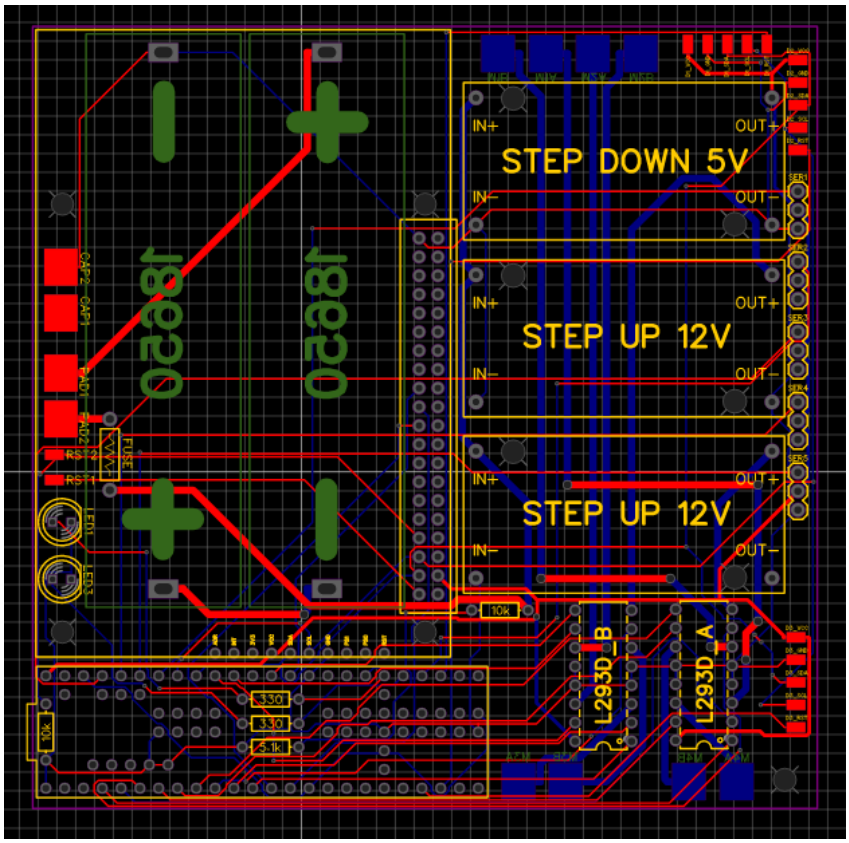

Figure 4: The main PCB in EasyEDA

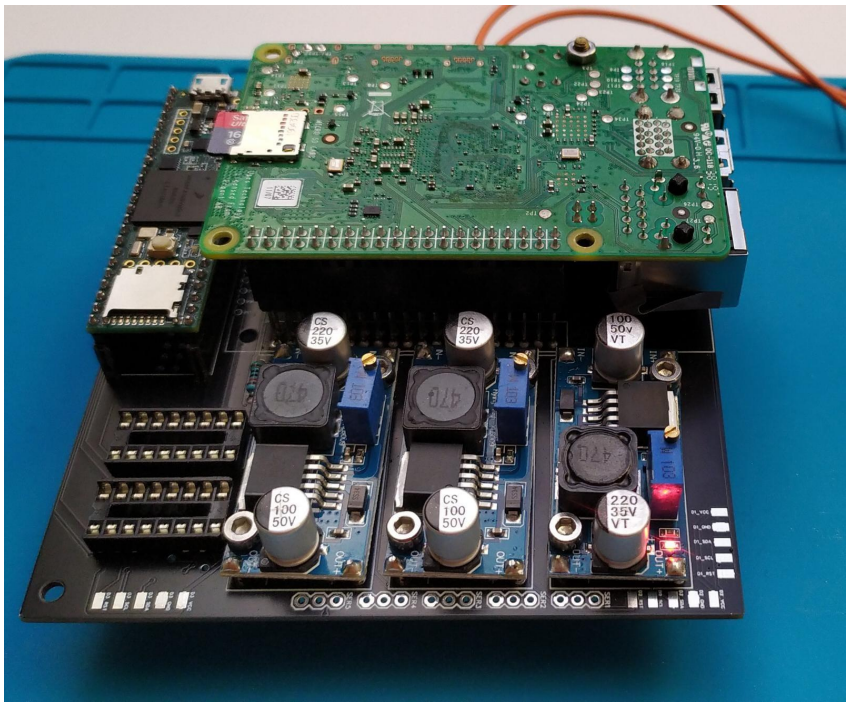

Figure 5: The main PCB during initial subsystem testing

4. Software

4a. General software architecture

Below is a basic diagram of the robot's and auxiliary software. The main program is written in C++ with a few libraries: OpenCV is used for some computer vision tasks as well as its data structures, although many algorithms are custom. TensorFlow Lite is used to run inference on the pre-trained Machine Learning models stored on the robot's SD card. WiringPI is used for the distance sensor and the buttons. The motor control program is written in C++ in the Arduino ecosystem. In addition to these two programs, we use Python (inside Google Colab notebooks) to manage datasets and train the neural networks.

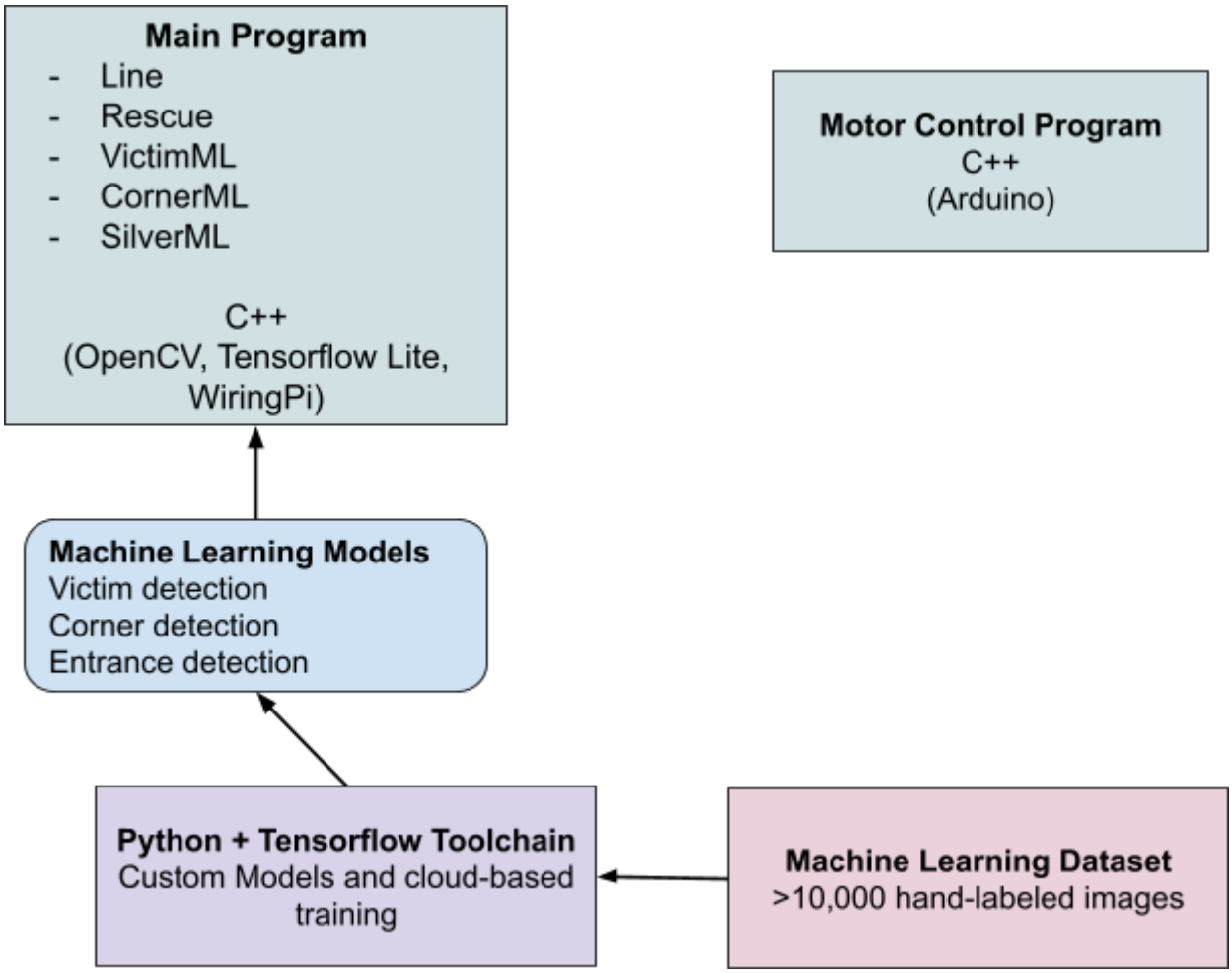

Figure 6: Basic Software Diagram

The main program is made up of several modules (classes), which are described below.

Line

This class handles everything except the rescue operation. The obstacle detection is run in a separate thread because of the slow ultrasonic sensor. By exploiting the Pi's multithreading capabilities and using efficient algorithms, the main *line* method runs up to 120 times per second. The things it does are illustrated in the following diagram:

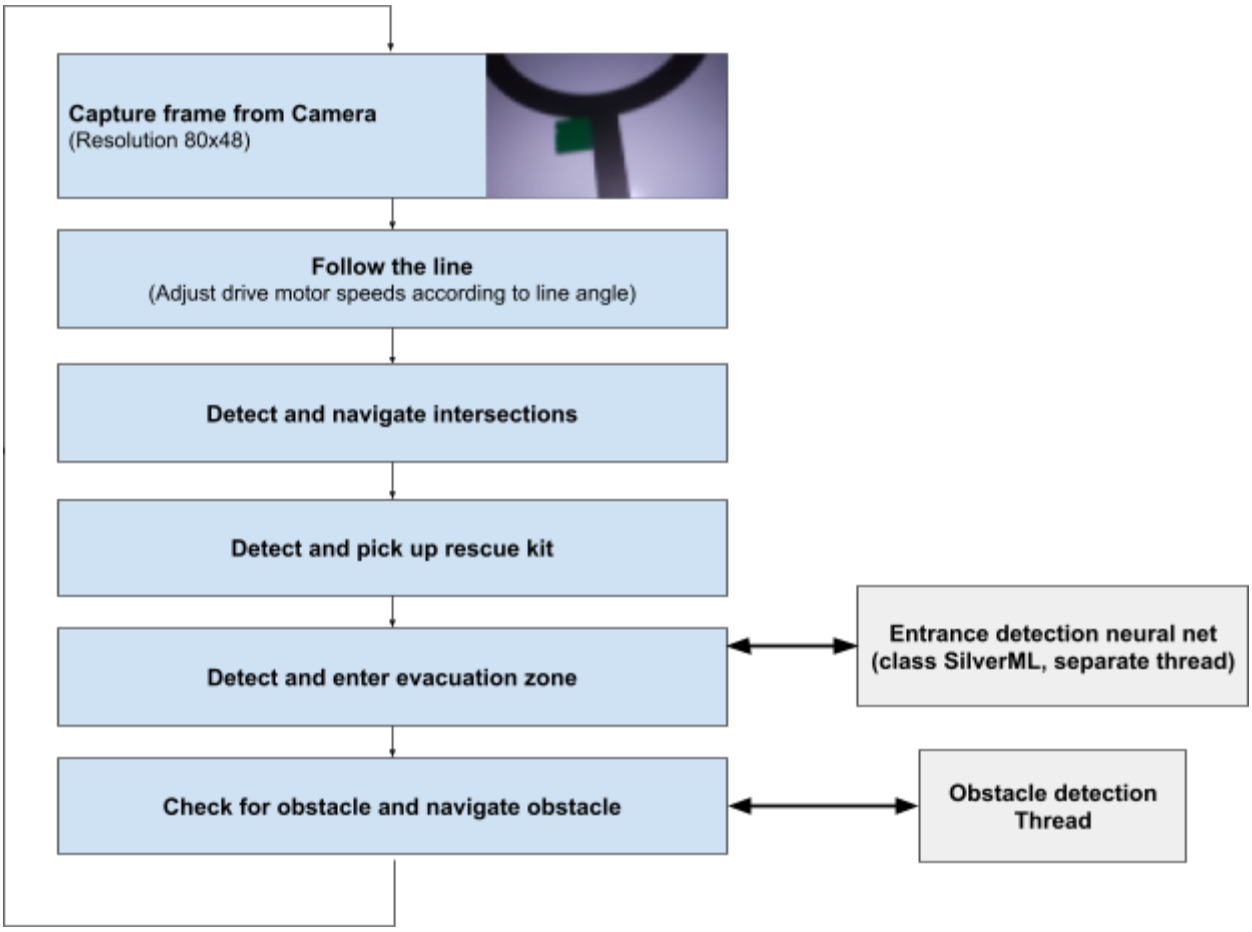

Figure 7: The main line-following routine

The individual subroutines are explained in more detail under section 4b.

SilverML

This is a simple abstraction class for initializing and running inference on the (comparatively simple) entrance detection neural net. The neural net outputs two numbers that represent the probabilities that the given image (directly captured during line following) contains the reflective tape or not. It runs in a separate thread because the neural net takes several milliseconds to run which would lower line following update rates to unacceptable levels.

Rescue

This is the class that takes over control of the robot once it has entered the evacuation zone, and until it has found the exit. The robot mostly operates from the center, where its camera can see all parts of the zone. Initially, the robot drives to the center and turns to find the corner using the corner neural net, which is explained in detail below. Then, the robot approaches the corner, turns around and unloads the rescue kit. Once back in the center, the robot repeatedly turns to search for victims. When it has found one, it incrementally approaches the victim until it is close enough, picks it up and returns to the center. From there, the robot can find the corner and repeat the process until all the victims are rescued. Then, the robot searches for the exit and returns to following the line. This class calls methods in the VictimML and CornerML classes but handles control flow, image capturing and contains the algorithms.

VictimML
This is the class that invokes the neural net for finding victims. The neural network's operating principle is described in section 4b.

CornerML
This class invokes the corner detection neural net. The neural network works very similarly to the victim detection neural net, however, it has a lower resolution.

4b. Innovative solutions

Efficient line following using computer vision

The robot follows the line using a custom algorithm that operates on the captured frame: First, we apply a thresholding operation to separate the image into black and non-black. Now we iterate over each pixel of the image. For each black pixel, we determine the angle between a line from the bottom center of the image to the pixel and the vertical. The angles of the individual pixels are then combined with a weighted average to get an estimate of the angle of the line. Pixels that are very close to the bottom center or very far away get low weights. These weights are constant, so they are calculated once at the beginning of the program, and then stored in an image *(Figure 8)*.

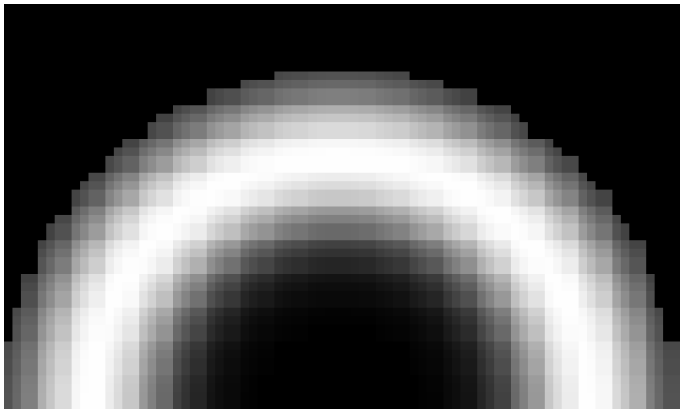

Figure 8: The distance-based pixel weights (white is 1, black is 0)

Sometimes with complex tiles, a different part of the line comes into view which we do not want to follow. To minimize their influence the pixel angles are also weighed according to how close they are to the last line angle. This line angle is then multiplied with a sensitivity factor and added and subtracted to the left and right motor speeds. The sensitivity factor is larger for large line angles and smaller for small angles. This smooths out following straight lines while retaining quick reaction times for sharp corners.

You can find simplified pseudocode for this algorithm in the appendix.

During line following, we constantly scan for green dots. Colors are detected by calculating the ratio of one component of color to the other two and comparing it to a threshold. We use a recursive algorithm to find groups of green pixels. If there is a green dot (or multiple), we cut out a part of the image around the green dot. Now we calculate the average position of the black pixels in that cutout. The quadrant of this average pixel now tells us whether to ignore this green dot and continue forward, or turn right or left (Figure 9).

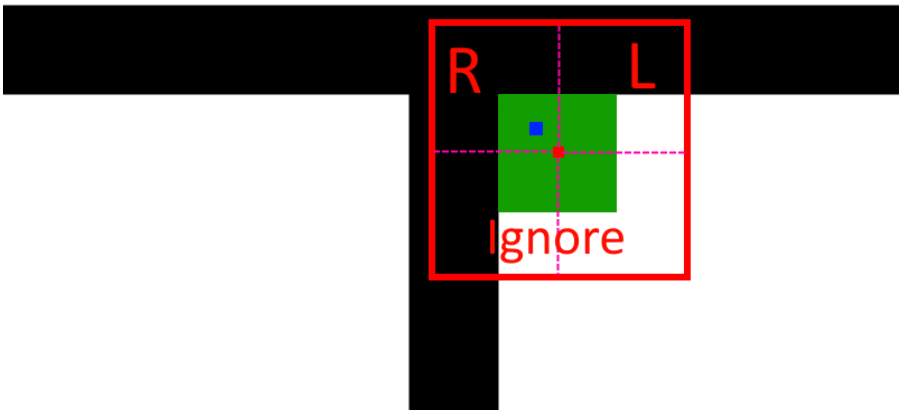

Figure 9: Determining the correct direction at an intersection (average pixel in blue)

The rescue-kit detection works very similarly to detecting the individual green dots. Once there are enough blue pixels the robot turns towards the rescue kit based on the horizontal position of the pixels, picks up the rescue kit and turns back to the line.

Victim detection neural net

The neural net was mainly inspired by the popular YOLO model for object detection, but it is greatly simplified in that it does not detect the bounding boxes of the victims directly but creates a "heat map" which can be processed further to obtain the actual victim positions.

It takes in a 160x120 black-and-white image and outputs a probability map of the two different classes of victims: dead and alive. Reflective and black spheres are easy for a neural net to find, so the model is very small (Only two convolutional layers). Figure 10 shows four different inputs and their corresponding outputs. The two images on the left with their outputs are taken from training data. The two inputs on the right are images unknown to the neural net. Before deploying the neural networks to the robot, they were extensively tested using pre-captured images on our PCs.

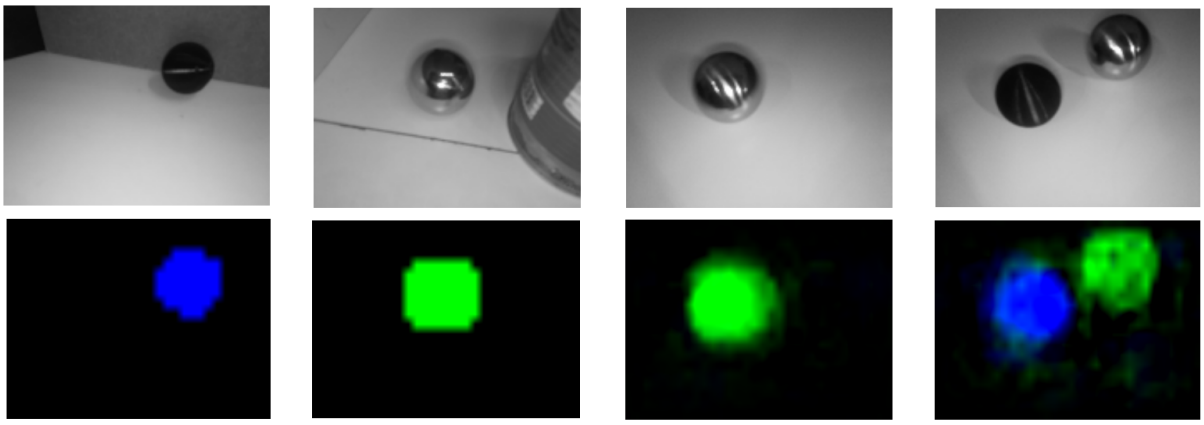

Figure 10: Two training images (left) and two real images (right) and their corresponding neural net outputs

Using simple thresholding operations and contour finding, we can now extract the victim positions from the neural net outputs. The neural net can run at about 5fps on the Raspberry Pi.

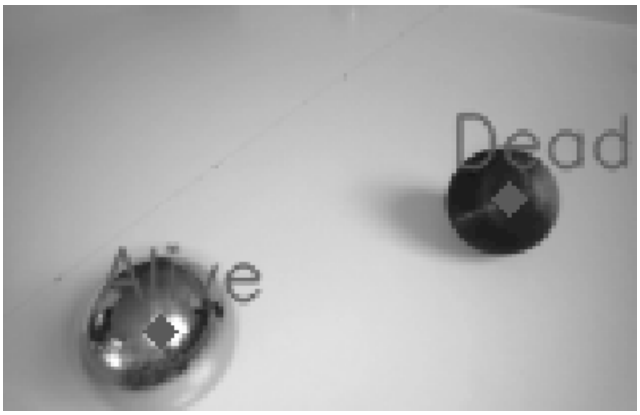

Figure 11: Two victims located and correctly labeled by the robot

In previous years, we used OpenCV's *HoughCircles* method to detect the victims. This method was very unreliable and rarely worked under different lighting conditions. At one point, the robot repeatedly detected debris which happened to be placed in a circular pattern. This led to the development of this neural network which now works reliably under different lighting conditions and is resistant to noise.

5. Performance evaluation

We were able to significantly improve the performance of this year's robot compared to our previous ones which is not only a result of experience but also intensive testing of both hardware and software. While most software subroutines were first tested in small test scripts before being implemented into our main program, we still conducted deliberate practice sessions for the robot after major changes to the program where we let it maneuver through the whole field (both line following and evacuation area) while monitoring and logging everything the robot was doing. We also always use a live preview in which the robot shows the current frame it is processing as well as further information (e.g. marking detected green turning signals or showing the current angle relative to the line). But even more importantly, we always store essential frames on the SD-Card whenever the robot detects an intersection, the rescue kit, the entrance to the evacuation zone, or a victim. This allows for post-evaluation and gives us further training data for the neural networks, in case of false positives.

As performance evaluation of neural networks in real-time can be challenging, we used a sufficient training/validation split and testing scripts to test the performance of newly trained neural networks before using them on the robot itself. To further test the resilience under competition conditions we often make use of debris or speed bumps placed all over the field to test the design. Experience tells us that the best performance is achieved through the harshest test conditions. Debris in the evacuation zone in particular is one of the major reasons we are using a neural net instead of standard circle-detection algorithms in the first place.

6. Conclusion

All in all, after a year of developing this robot, we can say that we have improved in many areas. We reached many of the goals we set ourselves after last year's season: Our development process was much more elaborate, and we planned, prototyped and tested more. We switched to mostly cloud-based services for better collaboration (Git, Fusion 360, Google Colab) and our communication became more frequent. We also made sure to document every aspect of the development, hardware and software, along with gathering more testing data.

The overall design of the robot is also much more mature and reliable. It is now easier to work on the robot's hardware and iterate due to its more modular design.

We are looking forward to this year's RoboCup season to not only be able to test the performance of our robot during actual competition while also talking to other teams as well as discussing their technical implementation and software solutions.

Appendix

If you would like to learn more about our robot, feel free to check out:

- Our YouTube channel
- Our website
- Our GitHub repository

If you have suggestions, comments, or questions about our development you can also write us an email at: robocup.evb@gmail.com

Simplified pseudocode for following the line using a weighted sum:

```
// Difference weight based on difference to last line angle
function diff_weight(x):
  // x in radians
  x *= 2 / pi
  return 0.25 + 0.75 * exp(-16 * x^2)


// Distance weight based on distance to bottom center
function dist_weight(x):
  e = (3.25 * x - 2)
  f = exp(-e^2) - 0.1
  if f > 0: return f
  else: return 0


sum = 0
sum_weights = 0

// Note: Image coordinates are from top left to bottom right
bottom_center_x = image_width / 2
bottom_center_y = image_height

for every pixel x, y:
  if pixel is black:
    distance = sqrt((x - bottom center x)^2 + (y - bottom center y)^2)
    angle = atan2(y - bottom center y, x - bottom center x) + 90°

    weight = diff_weight(angle - last_angle) * dist_weight(distance)

    sum += weight * angle
    sum_weights += weight

line_angle = sum / sum_weights

motor_speed_left = base_motor_speed + sensitivity * line_angle
motor_speed_right = base_motor_speed - sensitivity * line_angle
```

References

Software and platforms:

Google Colab

GitHub

Fusion 360

Libraries:

TensorFlow

OpenCV

WiringPi

Hardware:

Arduino Nano

Raspberry Pi

BNO055 library

Pololu - 20D Metal Gearmotors

Anycubic i3 Mega S 3D printer