System Design Project: Technical Report

Group 19

RETROSPARK

April 2018

Team members

Alexander Stewart - s1437078@sms.ed.ac.uk
Caelan Lekuse - s1530043@sms.ed.ac.uk
Callum Cheshire - s1433964@sms.ed.ac.uk
Ignat Georgiev - s1521716@ed.ac.uk
Vaida Plankyte - s1518385@sms.ed.ac.uk
Yijie Chen - s1508767@sms.ed.ac.uk

Abstract

This is a report of the project developed by Group 19 for the System Design Project. It describes the original objectives and ideas of OfficePal and how they have changed throughout the duration of the course. The document covers design decisions, implementation, testing and results. Issues and setbacks are also detailed along with the approaches used to overcome them. Lastly, the report evaluates the performance of the robot and presents future considerations.

Contents

1 Introduction

1.1 Background

The main objective given to us in this year's System Design Project was to create a robot over the course of a semester that falls into the category of 'assistive robotics'. We were granted a great amount of freedom in deciding which path we wished to pursue and were allowed a predefined budget as well as a selection of robotics hardware to get us started. This opportunity allowed us to develop our project, OfficePal.

1.2 Inspiration

OfficePal took inspiration from an autonomous delivery robot named Relay which is produced by the company Savioke[1] who specialise in service robotics. Due to budget and time constraints, it was not within the scope of the project to create such a fully autonomous robot with a similar large array of features. In spite of this, the basic idea of a user-friendly robotic assistant that can carry out delivery tasks within an office is still a concept that we have successfully attained while delivering it over a much smaller development period and at a much lower cost. Savioke at this stage is still in many ways superior to OfficePal but our team are nonetheless delighted with the functionality that we have managed to achieve.

2 Overview

OfficePal is a small differential drive robot built with Lego. It operates within an office environment where all desks are connected through black-coloured lines and can deliver items between desks using the tray attached on top of the robot. To perform these tasks it uses a vision-based line-following algorithm and can be controlled via an online web-app. Once OfficePal has multiple tasks, it can reschedule them and find the most optimal path to deal with all requests. To satisfy all of its computational needs, the robot is equipped with a Raspberry Pi 3 which enables it to execute all algorithms in real-time and host the web-app within the network it is currently connected to. Additionally, OfficePal can detect obstacles using its IR Sensors and can be controlled manually using a wireless joystick.

Figure 1: Final OfficePal design

3 Chassis

We decided to base the design of OfficePal's chassis on a lego "sumo robot" that we found online[2] which was originally designed to take part in sumo wrestling style matches. We thought that this design would be appropriate as, considering its original purpose, it was designed with robustness and stability in mind. Another benefit of this design was that most of the robot is kept relatively close to floor-level, which we decided would keep our robot suitably unobtrusive in an office environment. The sumo robot design was of course adapted numerous times to fit our hardware requirements for OfficePal yet it was a great design to start with.

Initially a castor wheel was used at the back of OfficePal to facilitate turning. This proved to be problematic. This caused traction issues which were a result of both the design of the castor wheel itself, which was clunky, as well as the way in which it was attached to the robot. The castor wheel was originally attached to the back of the robot with only 2 pins, meaning that the weight of the robot would force the axis that the castor wheel was rotating around to not face directly downwards, which was the main source of the problem. After noticing that the castor wheel was the source of the traction issues, we replaced it with a set of 3 spherical metal wheels at the back of the robot, which not only resolved the traction problems but also provided the robot with greater stability as the structure was now being supported at all four corners as well as in the centre. We also decided to have both the EV3 and the Raspberry Pi directly above our two main wheels, as having most of our weight situated there would help with traction. Initially our team wanted a 3D printed tray for our robot, but eventually decided to order an electronics enclosure and fashioned it into a tray instead. This was due to the difficulty of getting a booking at uCreate studio, as well as for budgeting reasons.

4 Hardware architecture

Our initial aim was to develop a simple and reliable platform that can be easily modified. We planned on utilising as much of the hardware available to the course as possible with the idea that the platform can be reused for different implementations in the future. Our solution as seen in Figure 2 uses a Raspberry Pi 3, Lego Mindstorm EV3 and an Arduino Uno and offers modularity and versatility at the cost of battery life.

To accommodate for the vision algorithm and the need for on-board web hosting, we chose the Raspberry Pi 3 as our main computational unit due to its low price, ease of development and connectivity options. Other options such as Arduino, Raspberry Pi Zero were considered but did not offer integrated Wi-Fi and enough processing power. On the other hand, options such as a Nvidia Jetson TX2 and Intel NUC Boards proved too expensive for the scope of the project. As a result of our choice we had to account for battery consumption and code optimisation.

The Raspberry Pi is running a modified version Raspbian Stretch Lite[3] which was stripped of unnecessary functionality. Ideally, OfficePal would be connected to an office Wi-Fi hotspot but due to the University network limitations, we chose to connect it to the Internet via eduroam and to control it via SSH through the tunneling service Dataplicity [4].

To physically control the robot, we chose the Lego Mindstorm EV3 because it was compatible with most of the motors and sensors supplied by the course and because it can power all of OfficePal. In the final platform, the EV3 controls both of the motors, powers the Raspberry Pi 3 via USB and is connected to a network with it via USB. Communication between both devices is achieved through a custom TCP library detailed in Section 6.

OfficePal is equipped with 4 Sharp GP2Y0A41SK0F [5] IR sensors which enable its obstacle detection feature. To read their analogue outputs, we had to fit the robot with an Arduino Uno which then transmits the sensor data to the Raspberry Pi via UART. This option was chosen instead of an ADC Raspberry Pi shield due to its availability within the course. It offers the added benefit of modularity and additional IO but also introduces more power consumption and complexity.

For the camera, we chose the Pi Camera [6] for its speed and cost. It offers superior performance

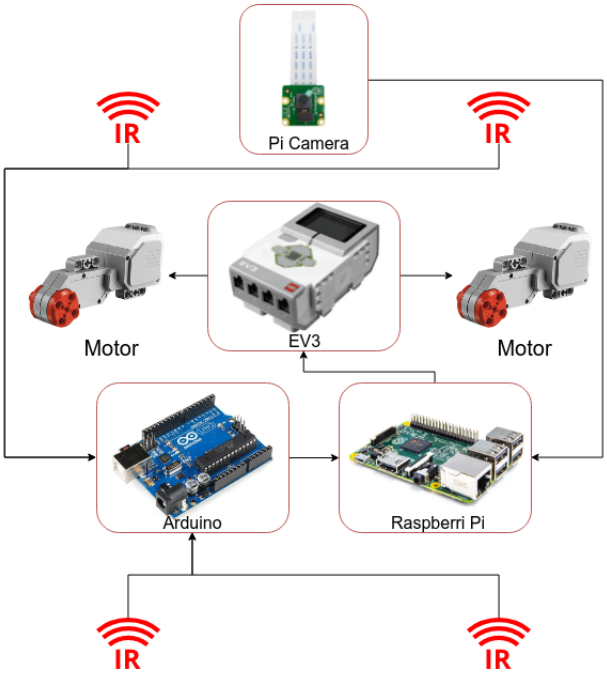

Figure 2: Hardware architecture

and resolution in comparison to the USB cameras we were supplied with but is lacking in image quality and camera accuracy. This trade-off was necessary for the robot to run in real-time. This is detailed more in Section 8.

With this hardware platform and the Raspberry Pi running in performance mode we were able to achieve more than an hour of run-time as seen in Figure 3. This would not be sufficient for real-world use but suffices for the scope of the course. It is worth noting that the data shown has been gathered with the robot fully-powering all motors and using all of its available sensors.

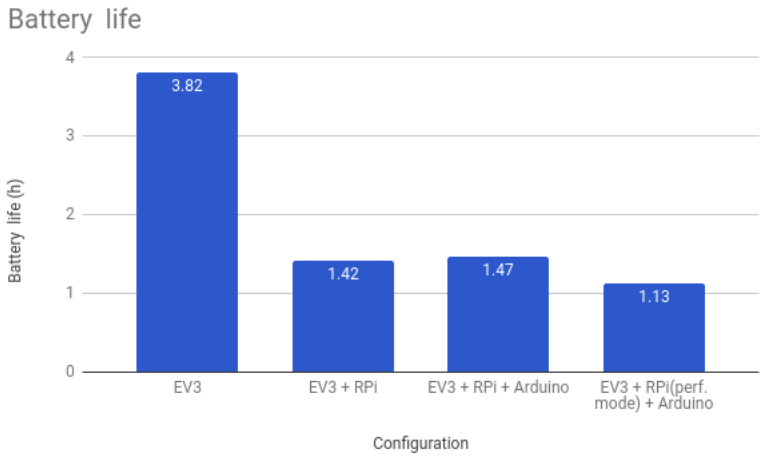

Figure 3: Battery statistics

5 Communication

To enable communication between the Lego Mindstorm EV3 and the Raspberry Pi, we chose to use TCP packets which would enable both devices to send messages to each other reliably and swiftly over USB as long as they were in the same network. This was chosen over traditional approaches (i.e. UART) because it offers extra reliability and simplicity without sacrificing speed. Additionally, this option is more modular and can be scaled to multiple devices.

The Transmission Control Protocol (TCP) is one of the main protocols of the Internet protocol suite. It originated in the initial network implementation in which it complemented the Internet Protocol (IP). Therefore, the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered, and error-checked delivery of a stream of bytes between applications running on hosts communicating by an IP network. TCP protocol operations may be divided into three phases. Connections must be properly established in a multi-step handshake process before entering the data transfer phase. After data transmission is completed, the connection termination closes established virtual circuits and releases all allocated resources[7]. Example operation of TCP is showed in Algorithm 1.

Algorithm 1 TCP Communication sequence

1: $s = connect(EV3)$
2: $send(s, MotorCommand)$
3: $s.waitAcknowledgment()$
4: $rcv = s.waitMessage()$
5: $s.sendAcknowledgment()$

After initial research, we failed to find an open-source event-based TCP library for Python 3 (required by the EV3). That forced the team to create our own solution which prolonged development time. In an attempt to limit this, we used an existing Python 2 event-based library[8] as the base of our implementation. In the end this proved to be inadequate and we had to fully develop out own solution.

For our final implementation, we have 4 separate classes $TCPServer$, $ServerHandler$, $TCPClient$ and $CliendHandler$. In this report we are going to explore the TCPServer in particular. As shown below, the server is initialised with:

```
import TCPServer
server = TCPServer(port, stateChanged=stateTrans)
def stateTrans(state, msg):
global isConnected
    global reqSensorReceived
    global sensorData
    if state == "LISTENING":
        wait for client to connect
    elif state == "CONNECTED":
     isConnected = True
    elif state == "MESSAGE":
        parse newly received message
```

[1]

Where it is left for the user to define what actions to take on state changes and the user does not need to take action on every change. This allows the user to flexibly add needed messages to both the sender or the parser. All messages used in our implementation are shown in Table 1. Messages have the following structure `message_type:comma,seperated,values` An example message history from the `TCPServer` used on OfficePal is detailed below.

Sender	Message	Description	Example
Raspberry Pi	Motor command	Controls both left and right motors of the robot	CMD:100,100
Raspberry Pi	Turn	Turn command used for junctions; value is in degrees	TRN:90
Raspberry Pi	Speak	Sends message for the EV3 to speak out	SPK:speak
Raspberry Pi	Request	Request the EV3 to send sensor data back	RQT
EV3	Sensor data	Send back data from ultrasonic, left colour and right colour sensor; the 2 last arguments are colours	SNR:45,Y,N

Table 1: TCP Messages

```
sent: CMD:50,50
sent: CMD:32,67
sent: TRN:90
sent: SPK:speak command
sent: RQT
received: SNR:50,
```
[2]

In further detail, the `TCPServer` itself handles all interactions with the Clients. The *TCPServer* class consists of the state control and connection establishment, whereas *ServerHandler* parses received messages and deals with pending messages that need to be sent. This is shown in Figure 4 along with the states used in our final implementation. *TCPClient* is implemented in a similar manner but instead attempts to find a server with a predefined IP, instead of waiting for a client to establish a connection.

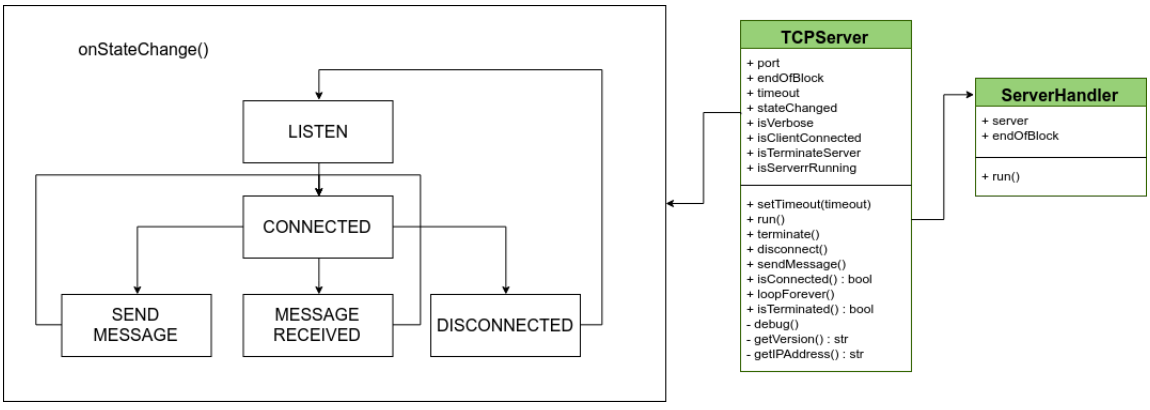

Figure 4: TCP Server Diagram

To test the speed and reliability of the protocol we used our manual control feature described in Section 12, along with the Python profile module[9]. The reliability of the system was tested through prototyping other modules of the robot and proved to be sufficiently reliable and caused only one issue during the final 3 weeks. Regarding speed, we managed to achieve a 120 µs response time from the Raspberry Pi sending a message to it receiving an acknowledgement, which proved to be sufficient to control the robot in real-time. In comparison, 40 µs was accomplished with UDP but reliability was not ensured.

A setback during the early stages of development of the TCP Library, was the unreliability of the Lego Mindstorm EV3. It would randomly allocate less resources to its `TCPClient` instance which resulted in bad overall performance. In some instances the process was not allocated memory at all and motors would be actuated with a large delay. To overcome this issue we had to convert the client to an operating system service which allocated high priority to the process and would not allow its resources to be reallocated. This also had the added benefit of launching the client on boot.

6 Environment

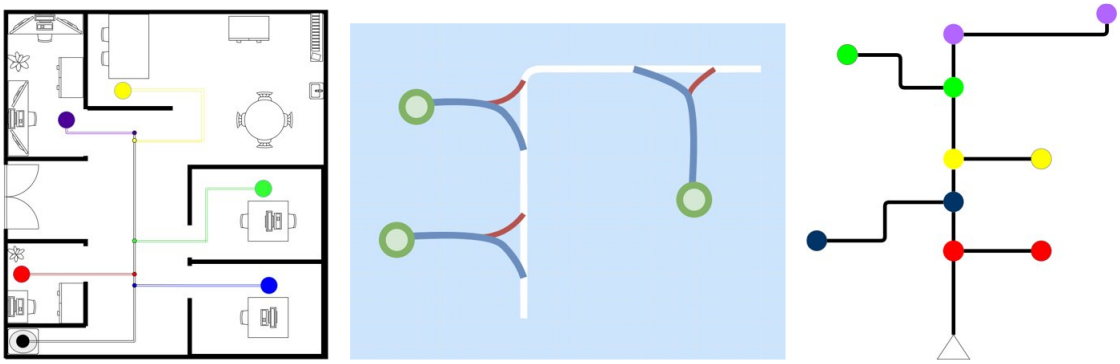

Figure 5: Previous environments

Our initial environment design (left, figure 5) was designed with colour-sensor base line following in mind, but our initial traction issues caused our robot to struggle with 90 degree turns, which brought use to our second design (middle, figure 5), featuring smoother curves which our line-following algorithm at the time had an easier time dealing with. After we decided to switch to camera based line following, we reverted to a design similar to our initial one (right, figure 5).

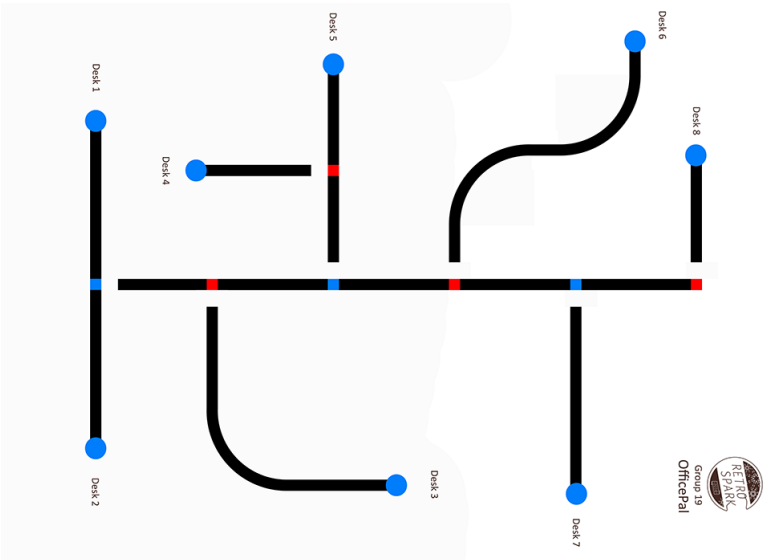

Figure 6: Final environment

We ended up with our final design (figure 6) as an environment featuring junctions of only two different colours was more suitable for our camera based line following. Multiple colours were unsuitable as we were unable to find enough colours with non-overlapping HSV values, which meant that the robot would frequently misidentify junctions due to the overlaps. We also decided that a design based on two colours was most suitable as it would be scalable, which a design that features different colours for each junction obviously wasn't.

7 Software architecture

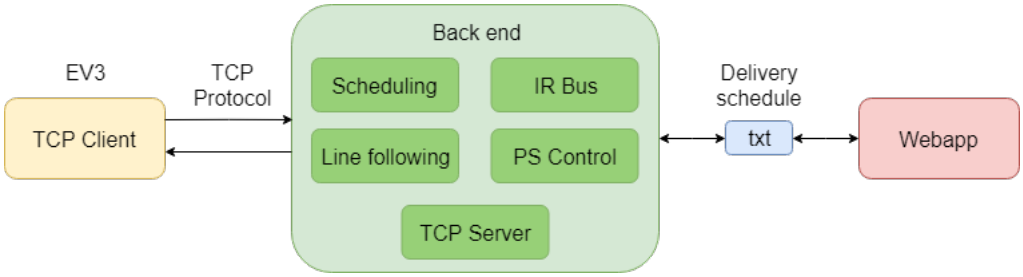

Figure 7: Independent subsystems and their interactions

In figure 7, the 3 blocks `TCP Client`, `Back-end` and `Webapp` were all independent subsystems running as different processes that required communications. The TCP client ran on the EV3 and controlled the wheel speeds based on TCP messages from sent from the Raspberry Pi. The `PS Control`, , etc. are implemented as separate modules running within the `Back-end` process on the Raspberry Pi. The `Webapp` process wrote to a file that is read by the `Back-end`

We chose to implement features in different modules to isolate each component, allowing them to be worked on independently. The main logic of the application used stub implementations of the modules while they were being developed, to allow the main logic to be tested regardless of the state of the other components.

8 Line following algorithm

The line following algorithm is implemented by Python's OpenCV library[9], and the pseudo code of this algorithm is:

Algorithm 2 Line following algorithm

procedure MAIN lower: lower HSV boundary of black;
upper: upper HSV boundary of black;
biasDistance: an empty list for store bias;
weight: a list of weight for bias distance
biasThreshold: a threshold which controls whether robot should turn or not;
height: height of input image;
width: width of input image;
image: input image from picamera;
    while get image from picamera **do**
        $sl \leftarrow height/4$
        for i less then 4 **do**
            $crop\_img \leftarrow im[sl * i : sl * i + sl, 0 : width]$
            $mask \leftarrow cv2.inRange(crop\_img, lower, upper)$
            $crop\_img \leftarrow cv2.erode(cv2.dilate(cv2.bitwise\_and(crop\_img, crop\_img, mask)))$
            $imgray \leftarrow cv2.dilate(cv2.erode(cv2.cvtColor(crop\_img, cv2.COLOR\_BGR2GRAY)))$
            $binaryImage \leftarrow cv2.threshold(imgray)$
            $contour \leftarrow$ **removeSmallContour**$(cv2.findContours(binaryImage))$
            $centerOfLine \leftarrow$ **cv2.Moment**$(contour)$
            $bias \leftarrow \frac{width}{2} - centerOfLine$
            $biasDistance.append(bias)$
            $sumOfbias \leftarrow sum(weight * biasDistance)$
            if $sumOfbias > biasThreshold$ **then**
                $leftmotorSpeed \leftarrow fixedspeed - bias/factor$
                $rightmotorSpeed \leftarrow fixedspeed + bias/factor$
            else
                $leftmotorSpeed \leftarrow 30$
                $rightmotorSpeed \leftarrow 30$

First, we use the **inRange** function with upper and lower HSV boundaries of a specific colour to generate a colour mask. This mask is then used to put the original frame into the **bitwise_and** function. Next, the closing algorithm **cv2.erode(cv2.dilate(image))** is used to delete the noise(from *figure 8* to *figure 9*). Through these functions, It will generate a diagram which only contains one specific color. Now, **cv2.threshold** is used to change the frame to binary and get the contour of the line, however the frame contains a lot of noise. This influences the quality of contour vastly, which may affect the bias calculation in the next step. Thus, we use the opening algorithm **cv2.dilate(cv2.erode(image))** then use **removeSmallContour**(small contour's area is less than 5% of total area). For the next step, 2 points are set in the frame. One is in the middle of robot vision and the other point is the center of the line using the **Moment** function. Therefore, the difference between these 2 points represents how large the offset will be. Finally, the bias is added to a distance list for future usage.*(figure10)*

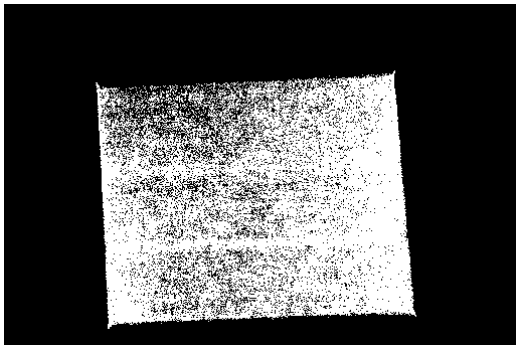

Figure 8: frame with noise

Figure 9: frame after using closing algorithm

Moving onto the bias distance list, every bias distance will have a reasonable weight, for example, the first and last bias distance will have a smaller weight to ensure the robot does not turn too early or too late. The result of the bias will be $bias = \sum_{i=1}^{4} bias\ distance_i\ *\ weight_i$ a threshold will then be set. If the absolute bias is larger than the threshold, then it means that the robot has an unacceptable bias and OfficePal should adjust its position and moving direction. The algorithm will renew the left and right motor speeds and send it to EV3. If the bias is larger than 0, then the left motor should be $fixed\ speed - \frac{bias}{factor}$ and the right motor should be $fixed\ speed + \frac{bias}{factor}$. This will reverse when there is a negative bias. Moreover, if the black line cannot be detected, OfficePal will output *-previous right motor speed* for the left motor and *-previous left motor speed* for the right motor until it can find the black line again.

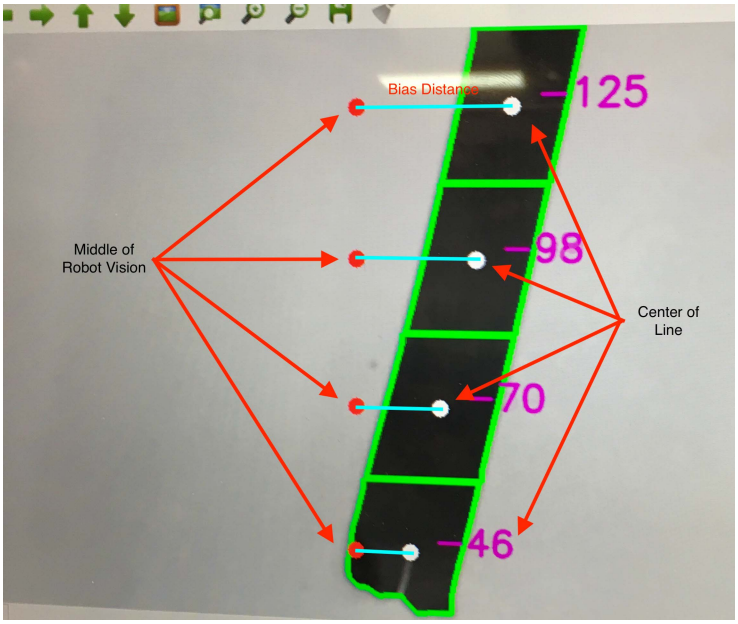

Figure 10: Visualised Line-Following

An issue we came across when implementing the vision line following was the IO buffer delay. We tried 2 different kinds of camera to test which one was the best suited. The first camera, the Logitech c270 webcam, proved to be unsuitable for our line following needs for the following reason: When the FPS is set to 60, for every loop it takes 0.3s to process the image, the camera will take around 0.25s to read the image and consequently the robot will have a delay of almost 1.5s when receiving motor commands. The result of delayed motor commands was inaccurate line following. As this was unsuitable, the decision was made to change the camera from the Logitech c270 to

a Pi camera. Using the Pi camera (with its own library) caused the delay to become acceptable (at about 0.5s). Now every loop only took around 0.1s and enabled line following to function properly with acceptable accuracy. Cause the Pi camera connects directly to the GPU. Due to being attach to the GPU directly, there is only a small impact on the CPU, leaving it available for other processing. Thus the processing time for every loop is decreased compared to the Logitech. Regular webcams are unlikely to get the same performance as they also use a lot more CPU.

9 Web application

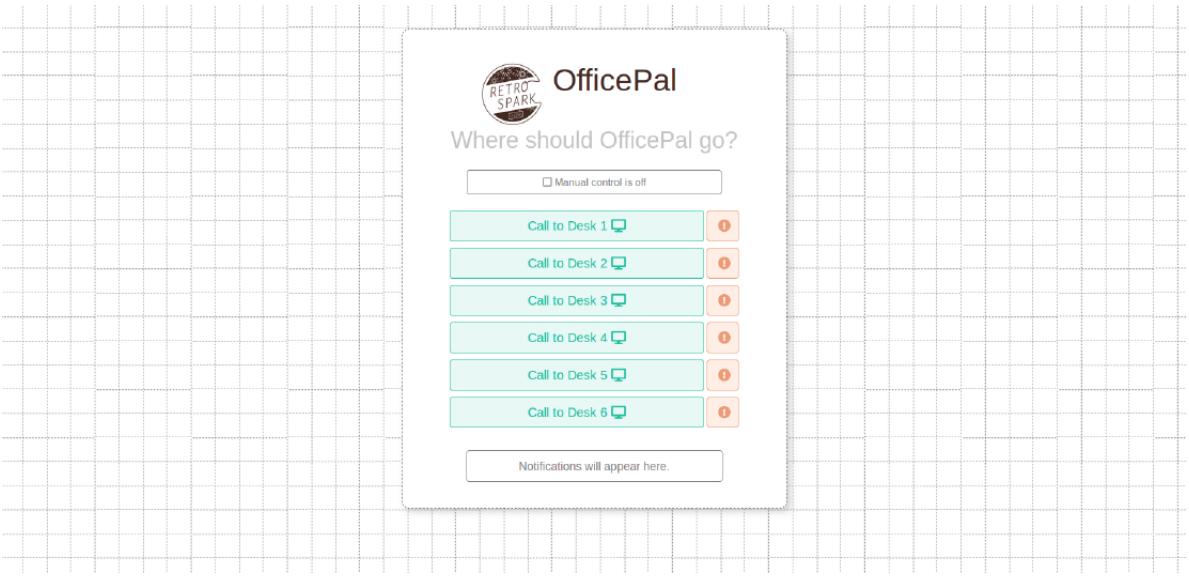

Figure 11: Webapp

The web application (see Figure 11) is written in Python using the Flask web framework for serving the application. Flask was chosen as it is intuitive and doesn't come with unnecessary libraries, and Python was being used for vision and communication with the EV3 already. A few additional libraries were used, such as `pygame` for joystick implementation and `numpy` for easier array manipulation.

The web application stores the requested destinations in a queue. A task ordering algorithm is implemented, based on the (x,y) locations of desks, which places the desk closest to OfficePal's current location to the front of the queue. This allows desks that are closer to OfficePal's current location to be fulfilled first, making the robot more efficient. It would be easy to extend the code to support different distance calculations, as the functionality is separated into clear functions.

The application also supports priority scheduling, which bumps priority requests to the front of the task queue. A starvation mechanism is implemented to prevent normal requests from being ignored due to an overuse of priority requests: after a set number of priority requests have been executed, a normal request in the queue will get promoted to a priority request itself. Priority tasks ignore the task ordering algorithm, as we need to ensure that the earliest priority call gets executed as soon as possible.

The previously mentioned functionality is summarised at a high-level in the pseudocode below.

Main functionality and request handling:

```
request_desk(desk)
check whether desk exists
    check whether desk has already been called
    if normal call:
     add desk to job_queue
    else:
     add desk at the front of job_queue, behind any other priority calls
    reorder_jobs()
    write_job()
```
[3]

```
reorder_jobs()
if there are priority calls:
     ignore task ordering
    else:
     find desk with min distance from current desk
        move this desk to the front of job_queue
        update_priorities()
```
[4]

```
update_priorities():
update the priority of each normal desk call in job_queue
    for each desk:
     if priority count of desk reaches a certain number:
         update desk to priority call
```
[5]

```
write_job()
check text file
if file contains a destination:
     write next job to file (overwriting file)
    else:
     remove the previous job from job_queue
         write next job to file
         reorder_jobs()
```
[6]

Regular checking of text file in separate thread:

```
every 2 seconds:
check text file
    if file is empty:
     remove previous job from job_queue
         reorder_jobs()
         write_job()
```
[7]

The application (`app.py`) is integrated with the vision line-following and path planning (`logic.py`) using a text file. `app.py` writes a number, that corresponds to the next destination that needs to be reached, into the text file. `logic.py` checks the file at regular intervals, and when it receives a destination, it empties the file and executes the path planning and camera code. After it is done processing a destination, it will start checking the file again. Meanwhile, when `app.py` sees that the file has been emptied, it then knows which location OfficePal is attempting to reach, and is

able to execute its task ordering algorithm and fill the file with the optimal location. The task ordering algorithm also overwrites the file if a new, closer location is requested by the user.

Using the in-between file allows for modularity, as `app.py` could be completely rewritten (even in a different language), and as long as its functionality included writing to a file, it would be able to function with the rest of the code. Multi-threading was another possible approach, which could have allowed us to have the application and the logic in a single file. However, it would have introduced issues in the form of race conditions, and preventing the web application from being an independent component. As a result, `app.py` and `logic.py` were kept independent, with minimal multi-threading used in `app.py` for checking the text file at regular intervals.

10 Obstacle detection

To enable OfficePal to safely accomplish its tasks without impacting the office workflow, it was equipped with 4 Sharp GP2Y0A41SK0F IR Sensors [*]. This allows our robot to accurately detect obstacles 4-30cm in a 7°field of view forwards and backwards as seen in Figure *. The sensors were chosen for their simplicity and availability within the course. Their configuration was chosen on the assumption that the robot would not move sideways and were found to be sufficient for our use. On the other hand, it also introduces large blind spots especially at the sides of the robot. We attempted to overcome this issue by adding complimentary ultrasonic sensors which provide a larger field of view but were found to produce noisy and inaccurate outputs.

As mentioned in Section 4, the sensor data is fed to the Raspberry Pi through an Arduino Uno. The Arduino reads the data at 1.78 MHz and transmits it as comma-separated ASCII characters via UART at 9600 baud rate. The microcontroller has been programmed directly in C using Atmel Studio as this offers superior speed in comparison the Arduino IDE.

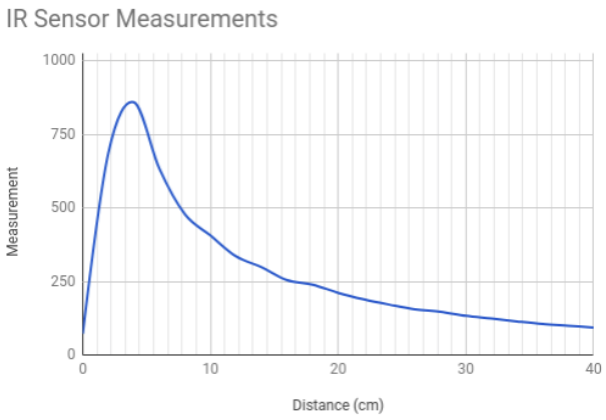

Figure 12: IR Sensor Measurements

To enable the Raspberry Pi to read the IR sensor data, a special class `IR_Bus` has been created which simply reads the serial data, detects potential errors and stores it. To guarantee realtime reads, data is read and stored in a separate thread which also tackles any I/O buffer overload issues. IR output data has been measured with a low-reflective obstacle for each of the 4 sensors and their average values are presented in Figure *. Due to the limitations of line following the robot simply stops as soon as it detects an obstacle and waits for its path to clear before proceeding with its task.

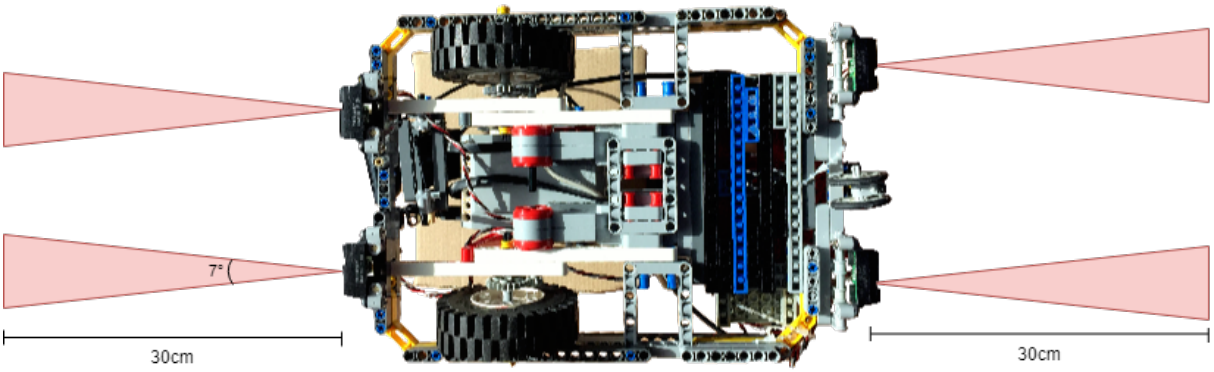

Figure 13: IR Sensor Field of View

11 Manual Control

To test the mobility and validate communication between different sub-components of OfficePal, we decided to add a manual control override which allows the user to control the robot via any wireless controller. We based our solution on the code by bythew3i [*] and we use the pygame library [*] to connect to the joystick to the Raspberry Pi. We opted to implement it as a standalone threaded class. This improves modularity and ease of use. Whenever an PSContoller object is created and ran, it automatically searches for a joystick and once found allows control.

The manual control setting is enabled through the web application, which then disables desk-calling and passes control to the wireless controller. The controller is constantly polled for output data via Bluetooth at 15200 baud rate. This allows for fast response time but also increases data throughput which can cause performance issues. Unlike buttons, controller joystick values are constantly being sent which can result in jerky movements in the case where one value is updated but the other one is not. To avoid this we have implemented a system which ensures that both X and Y joysticks are polled before a command is sent to the robot. This introduces a delay but has proved to be insignificant for our needs. During manual mode, the user can control the movements of the robot and can send speak commands. Controls are depicted in Figure * but it should be noted that these are highly configurable and can be changes by modifying a configuration file. To disable manual control, the user must press a button on the controller which breaks out of its loop and allows the line-following algorithm to gain back control.

Figure 14: PS4 control

12 Conclusion

12.1 Changes

From the very start an office delivery assistant robot named OfficePal was the general idea. The way in which we planned on implementing it has multiple differences when compared to the final version of OfficePal:

- Paths between desks were to be mapped on the floor using colour coordinated tape which would represent the route to each respective desk. Now, the paths between desks are all black lines with each junction to a desk having a squares in a red, blue, red, blue... pattern.
- Two colour sensors situated at the front of OfficePal would be aligned on each side of the coloured line it was intending to follow. OfficePal would stop the wheel on the respective side of the colour sensor if the line colour was detected. This would realign OfficePal back onto the line. Camera vision line following would also be employed by using OpenCV Python libraries with a USB webcam. The plan was to enable more accurate line following through the fusion of data between the vision and colour sensors using machine learning. For the current version of OfficePal colour sensors were discontinued and the vision line following was used as the sole line following method.
- We initially decided that OfficePal would feature a 3D printed tray in which users could place items and afterwards instruct it to deliver them to another desk within the office. We instead have a tray fashioned from an electronics enclosure.
- Commands were were to be issued through a web application or through a mobile device using Google Assistant's voice control. The final version of OfficePal is commanded through a web-app however using Google Assistant's voice control was an feature we planned on implementing if we had extra time. Unfortunately these features were not implemented due to time constraints.
- Odometry using camera vision would be implemented to allow obstacle avoidance and the ability to correct its path if it lost track of the line. Odometry was not implemented as further down the line in the development of OfficePal the idea became more unrealistic due to the complexity and time it would require to implement.
- An additional feature that was not originally planned was manual control.

12.2 Final performance

On the final demo day OfficePal performed mostly the way we had intended and generally we were happy with it, apart from one unfortunate instance during a call to a desk where it stopped

in the middle of a path under the impression that it was at its destination. At this point the error rate of OfficePal's usage was very low and had performed perfectly in the final client demo on the previous day, so were slightly disappointed about the mishap. We believe this malfunction was caused by an error in the colour detection in the vision line following. The HSV thresholds may have been too broad to accommodate to the lighting levels in the room we were presenting in. As a result of this, OfficePal did not recognise that it had passed over a junction which caused it to misunderstand where its destination was located.

12.3 Future considerations

1. For OfficePal to be sold to customers, it needs to be more visually appealing. For this we would need to create bodywork for the robot similar to Figure 15 which can be made out of ABS plastic in order to limit costs.
2. To limit hardware complexity, it would be ideal to use an Raspberry Pi ADC shield[12] instead of the Arduino. This would allow the Raspberry Pi to read the IR sensor values directly and would reduce the overall cost of the project.
3. The EV3 proved troublesome for the duration of the project and would best be replaced with another embedded microcontroller. An easy solution would be to use a BrickPi3 [13] which combines the functionality of a RaspberryPi 3 and a Lego Mindstorm EV3 into one piece. This would reduce the hardware complexity of the robot and would decrease the delays in controlling the motor.
4. We realise that if we were to roll out OfficePal to the general public, lines within an office would be frowned upon. To avoid that we might want to move from line-following to a VSLAM algorithm with multiple cameras. This would make the robot more versatile and widely applicable but would also increase the cost of the robot and the development time significantly.
5. To localise OfficePal within any environment, it will be worthwhile to research into RFID floors which would easily and reliably define the location of the robot at any time. Although, simple and easy for implementation this would also require customers to radically remodel their office environment which would prove costly and might be frowned upon.

Figure 15: Example bodywork

Figure 16: RFID floor

13 References

1. Savioke hotel robot

2. Sumo robot
3. Raspbian Stretch Lite
4. Dataplicity
5. Sharp IR Sensors
6. Pi Camera - https://www.raspberrypi.org/products/camera-module-v2/
7. Wiki page on TCP
8. Python profiling module
9. OpenCV library - https://opencv.org
10. Pygame library - https://www.pygame.org/news
11. Joystick code - https://by-the-w3i.github.io/2018/01/03/EV3-PS4-controller/
12. Raspberry Pi ADC shield
13. BrickPi3