Table of Contents

The Idea

We are a new team, and started late in June, so we need something that we can build that aligns with our abilities

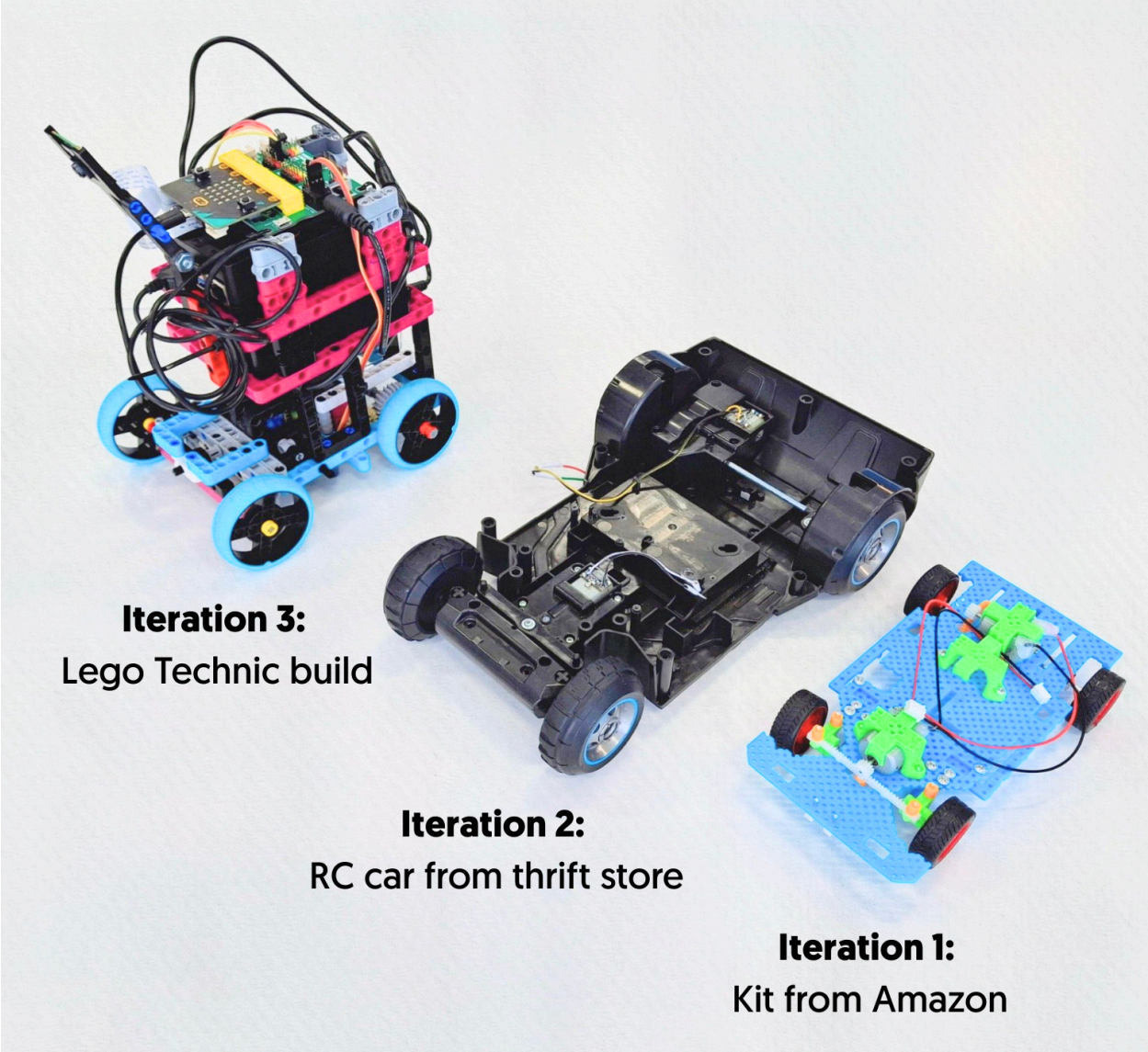
To be successful, we will need to go with:

1. A simple chassis - Either buy one, or modify an existing chassis
2. Use Raspberry Pi for , Python and a camera with OpenCV library for vision processing
3. Use Arduino or Microbit for micro controller
4. Use ultrasonic sensor, color sensor and IMU sensor as needed
5. Use simple power management setup to distribute power to micro processor, micro controller and the motor bridge

This notebook documents our journey and the decisions we made for each component as we iterated to the final working product

Chassis

The chassis underwent 3 iterations to the final product



Iteration 1

For the first iteration, we started with a simple chassis purchased from Amazon

Advantages

1. Light weight and easy to build
2. Allowed some level of customization to add 3D printed parts
3. Gained understanding of lack of fine steering control with DC motor (and its limitations)

Disadvantages

1. Too flimsy - the structure can't handle the weight of all the components, about 600 grams of it.
2. Underpowered - The small wheels with the gearing ratios with the DC motor did not allow for enough speed or enough torque to drive the weight
3. Steering was either fully to the left or right with the DC motor. Finer direction control was not possible - although we could've replaced it with a servo motor

Iteration 2

We experimented with a RC car that we got from a thrift store after removing its top and control circuitry

Advantages

1. The chassis was just available. Nothing to build. It was also very stable and not flimsy
2. Light weight
3. We could add 3D printed parts to it
4. Rubber wheels had good grip

Disadvantages

1. The chassis was too wide at 15cms. It was hard to park where we needed it
2. DC drive motor based drive axle kept rolling and didn't stop where we needed it to stop, unless we reversed the motor direction.
3. Steering was either left or right with the DC motor. Finer direction control was not possible - although we could've replaced it with a servo motor

Iteration 3 - Final

Finally, we settled on a Lego technic chassis that we built ourselves and it worked for us

Advantages

1. Ability to customize as needed without having to 3D print many parts
2. 9V Lego motor provides the power and torque we need, and it easily mounts on the chassis

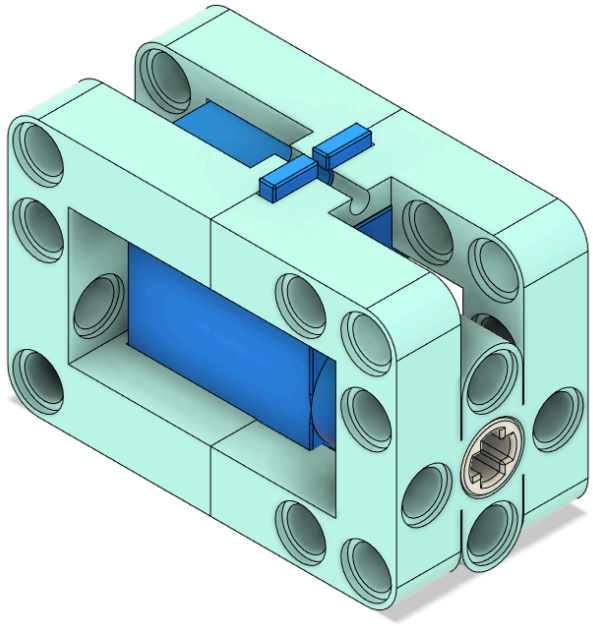
Disadvantages

1. 9v Lego motor isn't ideal. However, with a L298N motor driver we were able to handle that.
2. Couldn't get a Lego servo motor to work with the L298N motor driver. We had to build 3D printed mount for SG90 servos instead.
3. The final chassis looks bulky. There is room for optimization
4. Heavier than the other 2 chassis, however it is still within allowed weight

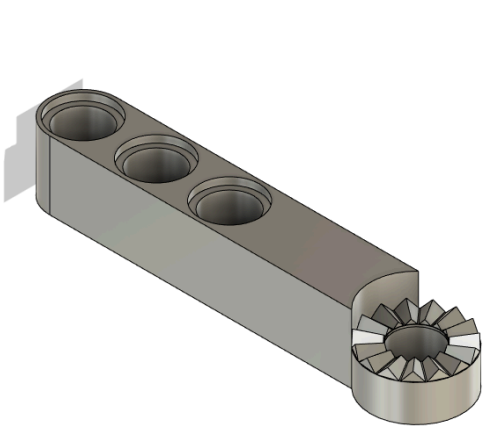
3D printed parts

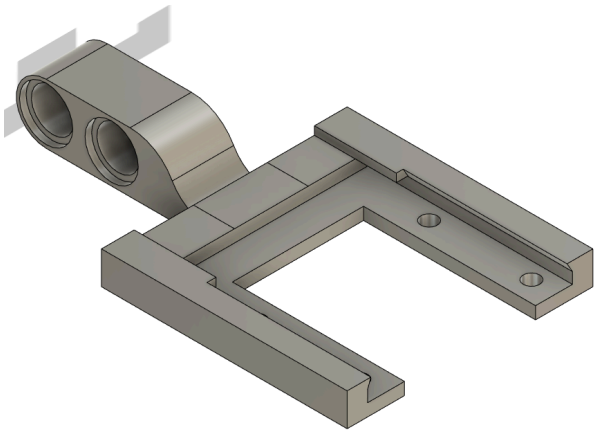
Since we used a Lego Chassis, we needed a way to mount

1. The servo motor to the front wheels for steering.
    a. Initially we tried to 3D print our own design.
    b. Later, we found a 3D file that we found on Thingiverse that worked better for our purpose



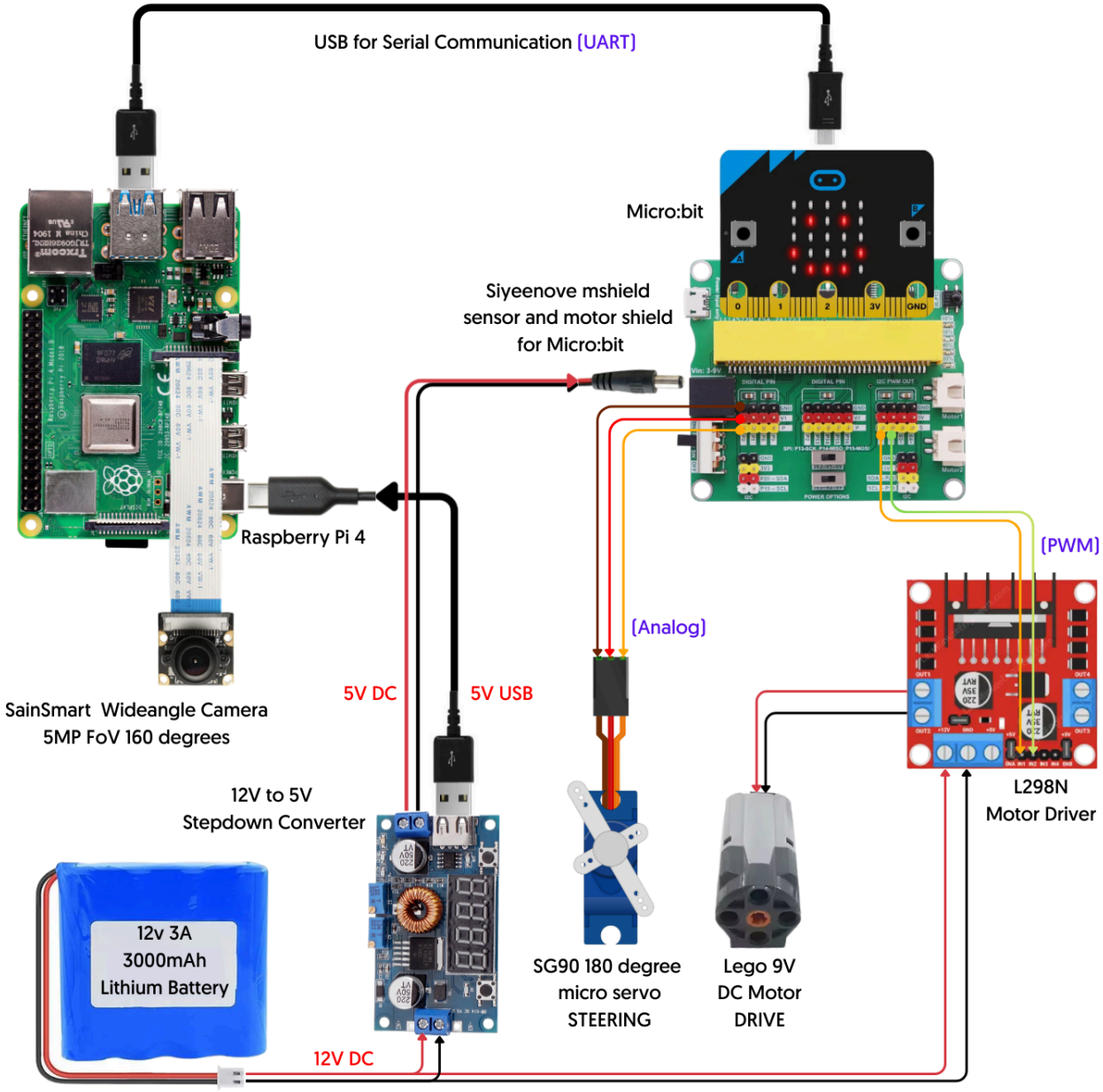
2. Mount for the wide angle camera.
    a. We designed and 3D printed this ourselves.





System control

Control system setup i.e, the raspberry pi, the camera, the microbit and the shield, the motor driver, the motors themselves



Choice of micro controller - Microbit

We experimented with an **Arduino Uno R3 and** board early on with a motor control shield for both DC motor control and Servo motor control.

Why not Arduino

We had issues getting the motor control shield to work due to an outdated library.

Why not ESP32 S3 WROOM Camera board

We could never get the camera to respond. Also the coding was to be done in C++. As we were troubleshooting that, we discovered that **Microbit** is more powerful than the Arduino Uno R3 board.

Microbit Advantages

1. More processing power than the Arduino
2. Smaller form factor compared to an Arduino
3. Similar number of I/O pinouts as the Arduino
4. Easy to learn. We used python for programming
5. Extensive library support for common sensor modules
6. Inbuilt buttons were used to start / stop the robot
7. Inbuilt LED array display and speakers were used for troubleshooting
8. Inbuilt Inertial Movement Sensor unit for precise turns. We did not need this.
9. Very low power supply requirements

Microbit Disadvantages

1. Harder to connect to the pins. We used the **Siyeenove Microbit shield** to overcome this
2. Inadequate information online to understand interfacing with Raspberry Pi over USB serial connection, but it was quite easy to figure out.

Microbit Motor Control

Siyeenove Microbit shield allows us to conveniently interface with

1. The Servo motor via the P0 pin. We use analog signal to position the servo motor
2. The DC motor control via the PWM pins connected to the L298N motor driver.

Microbit Serial connection with Raspberry Pi

The **Microbit** connects with the **Raspberry Pi** via a USB cable for serial data transfer. This serial data connection is used by the Raspberry Pi to achieve bi-directional data exchange with the Microbit

1. TX to Raspberry Pi - Control signals to start and stop the car operation
2. RX from Raspberry Pi - Steering and Servo control values

Use of Microbit buttons

- **Button A** is programmed to start and stop the robot. When robot is stopped by pressing the A button, pressing it again will commence a new run
- **Button B** is the kill command. It is programmed to stop all motor activity by signalling the Raspberry Pi to exit the Python program execution

Function Outline

- As soon as it starts, it initializes the motor speed to 0 and servo direction to 90 (which keeps the car straight)
- It awaits signal from the Raspberry Pi to confirm its program has started successfully. When it receives the confirmation, it displays an 'S' on the LED array
- When the user presses the 'A' button, it transmits a 'go' signal to the Raspberry Pi and awaits acknowledgement from the Raspberry Pi. When the acknowledgement is received, it displays a '+' on the LED array indicating that the car has started.
- As the car runs, Raspberry Pi transmits the servo steering turn data and DC motor speed data as byte stream which the Microbit will convert to numerical data, and split to integer values to be used for motor control
- The servo turn angle is sent as analog signal to the servo motor directly. The 5V input required by the Servo is supplied by the Siyeenove Mshield board.
- The speed data is sent as PWM signal to the L298N motor driver which uses the PWM signal and the power input from the 12V battery to drive the 9V DC motor.
- If the user wants to stop the car (and reset its values so the car can run again), they can simply press the 'A' button again which will signal the Raspberry Pi to stop the loop that processes vision data. This will display a minus '-' on the LED array.
- If the user wants to kill the car (completely end the program), they can press the 'B' button which will signal the Raspberry Pi to exit the Python program

Choice of SBC - Raspberry Pi

Raspberry Pi was an easy choice. We were familiar with it. Also, during our research we also noticed many other teams using it with a camera module.

Advantages

1. Familiar platform with abundant support, especially for vision processing using OpenCV libraries with a basic camera module
2. Allows programming in Python which we are comfortable with
3. Ability to connect serial interface via USB

Disadvantages

1. We used R-Pi 4B which requires a 2.5A portable power supply which was initially hard to manage.

2. We tried driving the motors with Raspberry Pi directly, but that didn't go well. The motor behaviors were unpredictable and the Raspberry Pi shutdown whenever the Servo stalled. So we decided to have the Microcontroller handle the motor control signals with the vision input parameters from the Raspberry Pi

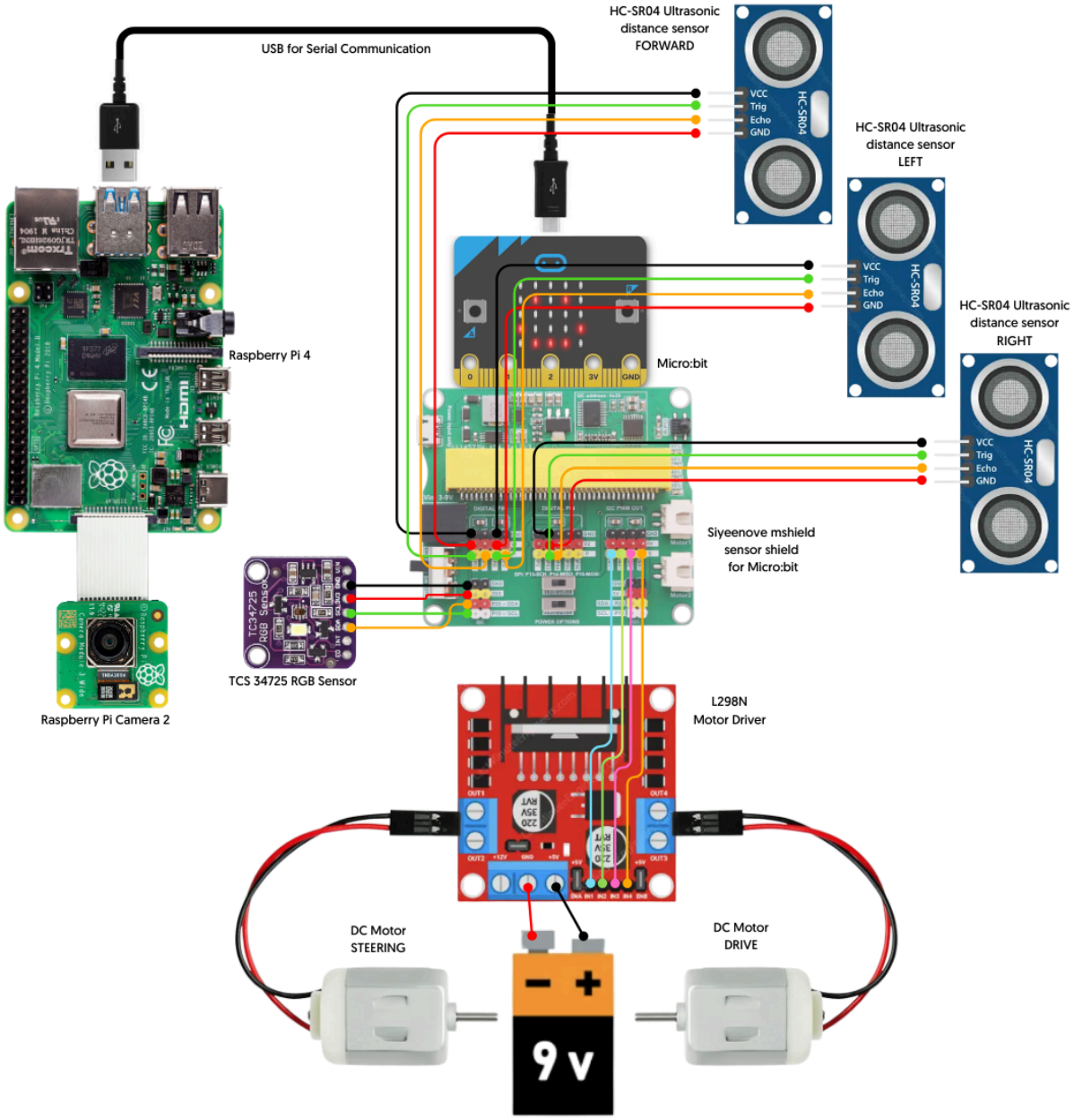
Function Outline

- Python program `obs_ch.py` is loaded from the crontab, which launches the program when the Raspberry Pi boots up
- Once it is ready, it sends a signal to the Microbit indicating that it is ready. If there is a problem with the Python Program loading, the signal to the Microbit will not be sent which will allow the user to fix the program.
- Then it listens to the Microbit's A button to be pressed to start the car which starts calculating and transmitting DC motor speed and Servo turn data to the Microbit.
- As the car runs, the program reads visual data from the camera frame by frame and processes it using proportional-derivative controller (https://en.wikipedia.org/wiki/PID_controller) to get the servo turn values, which is transmitted to the Microbit in real-time to control the cars turns
- The program has 2 distinct vision processing logic inside it
    1. For the open run, it looks at the walls and uses PD control to keep the car between the walls
    2. For the obstacle run, it looks at the color blocks and uses PD control to turn the car to avoid the blocks.
- Detailed explanation of the code is provided in the Program Logic section

To see or to sense?

Iteration 1

- Use of sensors connected to the Microbit
    - Ultrasonic sensors for wall detection
    - Color sensors for line detection.
    - IMU for accurate turns
- We tested with one Ultrasonic sensor and one color sensor in iteration 2 of our chassis and we were successful.



Iteration 2

- In our subsequent research we were able to find inspiration to do it all with the camera. We tested it out and found success - as explained in the program logic section
- Since we had success with using the camera alone, we removed all the sensors from the design and decided to go with only camera vision

Motor Control

Drive motor

- Used **Lego 9v DC motor** because it had the ideal torque and speed characteristics we needed
- Used **L298N motor driver** with 12v power input from the battery and PWM control input from the Microbit to control the 9V Lego DC motor
- **L293D motor driver** was considered, but was not used since it can only handle 600mA, whereas the Lego 9V motor has a stalled current draw of 850mA

Steering Servo

- Used **SG90 180 degree servo motor** with analog output from the Microbit.
    - 90 degree points the wheels straight
    - 30 degree is the farthest right steering, limited by the code
    - 150 degrees is the farthest left steering, limited by the code

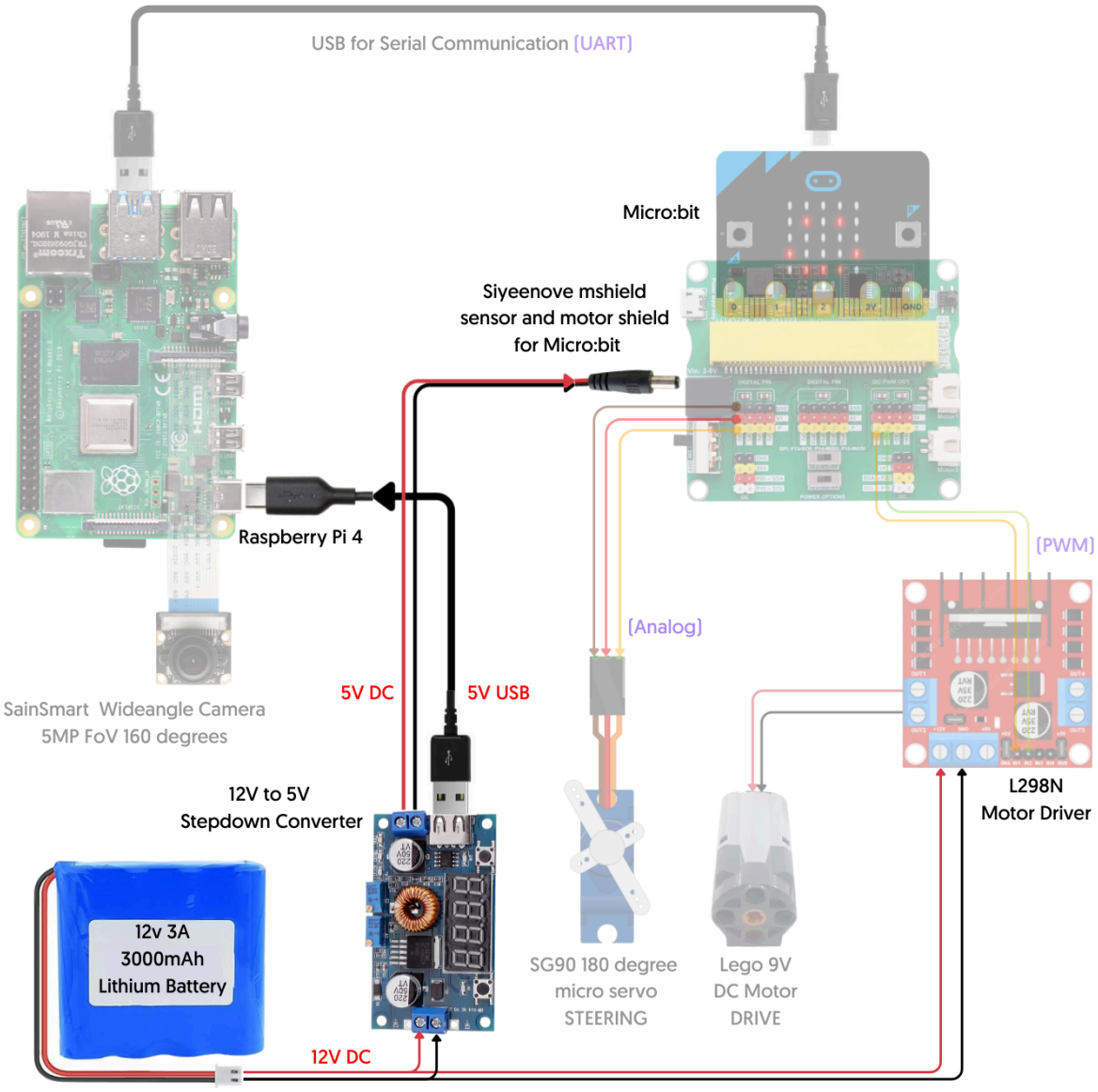
Power Management

Initially, as we were testing vision, all the components were powered with a wall outlet power source. Once we fully understood the power requirements, we found a 12V rechargeable power supply that can supply 3A of current to be able to concurrently supply power to

1. Raspberry Pi 4B - Needs 5V 2.5A
2. Siyeenove shield for the Microbit - Needs 5V 0.5A
3. L298N motor driver (which in turn powers the 9V lego motor) - Needs 12V 0.5A

We used a 12V to 5V step-down power converter to supply 5V power to the Raspberry Pi and the Microbit shield.

The Microbit itself did not require separate power since it was powered by the shield.



Program Logic

Microbit - pi2mb_motors_control.py

Summary

- Microbit powers up, then plays a tone, initializes motors & serial.
- **Button A** sends "go" to R-Pi (start/stop car).
- **Button B** sends "quit" to R-Pi (exit program).
- **Raspberry Pi** to **Microbit** Serial Connection data transmission via USB**:**
    - Sends **status codes ("startt", "stoppp", etc.)** → Microbit shows icons.
    - Sends **numeric codes (e.g., "120250")** → Microbit sets servo angle + motor speed.
- Servo Motor is controlled using analog signals via the **Siyeenove mShield**
- Drive Motor is controlled using PWM signals sent to the **L298N motor driver**.

Imports and Setup

```
from microbit import *
import music
import mShield
```

- `microbit`: gives access to Microbit's hardware (pins, buttons, display, etc.).
- `music`: allows sounds to be played.
- `mShield`: library for controlling the **Siyeenove motor shield** (PWM outputs for motors/servos).

Startup Signal

```
music.play(music.tone_playable(Note.C,
music.beat(BeatFraction.WHOLE)), music.PlaybackMode.UNTIL_DONE)
```

- As soon as the Microbit is started or reset, it plays a tone to indicate the Microbit is active.

Serial Communication Setup

```
serial.redirect_to_usb()
serial.set_baud_rate(BaudRate.BAUD_RATE115200)
serial.set_rx_buffer_size(7)
```

- Setting up serial connection parameters. This is how the Microbit exchanges commands/data with the Raspberry Pi.
- Notice the buffer size of 7. This means all the inputs received from the Raspberry Pi should be 7 characters including the newline character used a delimiter

---

Motor Shield Setup

```
mShield.set_s1_to_s4_type(mShield.S1ToS4Type.PWM)
```

- Configures ports **S1–S4** on the Siyeenove Mshield to output **PWM** (used for motor speed/servo control).

Default Values

```
deg = 90
speed = 0
pins.servo_write_pin(AnalogPin.P0, deg)   # Servo straight
mShield.extend_pwm_control(mShield.PwmIndex.S3, 0)  # Stop motor
mShield.extend_pwm_control(mShield.PwmIndex.S4, speed)
```

- Servo is set to **90° (straight)**.
- Motor PWM is set to **0 (stopped)**.

Button Controls

```
def on_button_pressed_a():
    basic.pause(100)
    serial.write_string("go")

input.on_button_pressed(Button.A, on_button_pressed_a)
```

- When **Button A** is pressed → Microbit sends **"go"** over serial to R-Pi.
- R-Pi decides whether to start/stop the car.

```
def on_button_pressed_b():
    basic.pause(100)
    serial.write_string("quit")
    basic.show_leds("""
        # . . . #
        . # . # .
        . . # . .
        . # . # .
        # . . . #
    """)

input.on_button_pressed(Button.B, on_button_pressed_b)
```

- When **Button B** is pressed → Microbit sends **"quit"** and shows a pattern on its LED display.

Main Loop

```
while True:
    try:
        data_in =
serial.read_until(serial.delimiters(Delimiters.NEW_LINE))
```

- Continuously listens for messages from the R-Pi.

Status Messages from R-Pi

```
if (data_in[0] == "s"):
    if data_in == "startt":   # Show "start" icon
    elif data_in == "stoppp": # Show "stop" icon
    elif data_in == "ssssss": # Show another status pattern
```

- If the incoming data starts with `"s"`, it's a **status message**.
- Different LED patterns indicate whether the car is:
    - **S** indicates **Raspberry Pi Ready** (also a tribute to the initials of the team member names, Swara and Shadya)
    - **+** indicates **Starting**
    - **-** indicates **Stopping**

Motor/Servo Control Commands

```
else:
    deg, speed = int(data_in[:3])-100, int(data_in[3:6])-200
    pins.servo_write_pin(AnalogPin.P0, deg)
    mShield.extend_pwm_control(mShield.PwmIndex.S4, speed)
```

- If it's not a status message, it's a **control command** (numbers only).
- The first **3 characters** → servo angle (shifted by 100).
- The next **3 characters** → motor speed (shifted by 200).
- Sample:
    - `"150250"` → servo = 150-100 = **50°**, motor speed = 250-200 = **50**.

Error Handling

```
except:
    basic.show_leds("""
        # . . . #
        . . . . .
        . . . . .
        . . . . .
        # . . . #
    """)
    pass
```

- If something goes wrong (bad serial data, etc.), shows an **error pattern** on the LED grid.

Raspberry Pi - obs_ch.py

Defining global constants and variables

Library imports

```
from picamera2 import Picamera2
import cv2
import numpy as np
import serial, time
```

- `Picamera2` → controls the Raspberry Pi camera.
- `cv2` (OpenCV) → image processing (drawing, masks, contours, etc.).
- `numpy` → used for arrays (e.g. HSV color ranges).
- `serial` → to transmit to Microbit over USB.
- `time` → for delays.

Constants

```
DRAW = True
DEFAULT_SPEED = 0
DEFAULT_STEER = 90
MAX_R_STEER = 15
MAX_L_STEER = 165
STEER_ADJ = 25
```

- Flag: if `True`, draw debug rectangles/contours on the image so we can see what the robot sees.
- The other constants are used to define default values for the motor and servo control

---

PD controller gains

```
KP = 0.025
KD = 0.1
```

These tune the robot's steering smoothness.

- `KP` (proportional gain) → how strongly to steer based on error.
- `KD` (derivative gain) → how much to react to changes in rate of error.

Block detection scaling

```
K_X = 0.12
K_Y = 0.2
```

- K_X is used to control the steering based on the blocks position on the X axis.
- K_Y is used to control the steering based on the blocks position on the Y axis. Closer the object is, the greater the influence will be.

HSV color thresholds

These define which pixels belong to a given color range. HSV is used because it's more reliable than RGB in challenging light conditions where the Hue of a color falls within a specific range. An upper range and lower range are defined to create a mask which filters these colors

- Black - for walls and lines.
- Green and Red - for block detection.
- Blue and Orange - for line detection

We used a HSV color palette and converted the actual HSV values to OpenCV values (to be contained within the 0-255 range) using the below formula

- H (Hue) - 0 to 360 values converted to 0-180 values by dividing by 2
- S (Saturation - 0-100 values converted to 0-255 values by multiplying by 2.55
- V (Value) - 0-100 values converted to 0-255 values by multiplying by 2.55

Camera image dimensions

```
FRAME_WIDTH = 640
FRAME_HEIGHT = 480
```

Scaled down resolution from the OV5647 sensor for faster image processing. We are able to achieve up to 25 frames per second because of this.

All the rectangle coordinates below use this.

Regions of interest

We split the camera view into smaller zones, each with a specific purpose.

<u>For block detection</u>

```
X1_CUB = 40
```

```
X2_CUB = 600
Y1_CUB = 220
Y2_CUB = 440
```

- Rectangle in the **center** of the image.
- Used to look for green/red blocks.

For wall following (PD controller)

```
X1_1_PD = 610
X2_1_PD = 640

X1_2_PD = 0
X2_2_PD = 30

Y1_PD = 220
Y2_PD = 480
```

- Used to make two rectangles:
    - Left zone (`pd_l`) = from 0 to 30 along the X axis on the far left.
    - Right zone (`pd_r`) = from 610 to 640 along the X axis on the far right
    - Used to measure wall area left vs right.
- Both at the same height (220–480).

For line counting

```
X1_LINE = 280
X2_LINE = 360
Y1_LINE = 400
Y2_LINE = 480
```

- Small horizontal strip at the **bottom** of the frame.
- Used to detect black crossing lines (to count laps/checkpoints).

Frames class

Summary

This is a **helper class** for working with **specific regions of the camera image** by dividing its camera view into regions (Frames), and each region specializes in finding a specific thing (wall, line, block).

- A `Frames` object = “a rectangular window” into one part of the camera image.
- It knows:
    - **Where** to look (rectangle coordinates).
      What color to look for (HSV range).
- It can:
    - Crop & process that rectangular window.
    - Detect contours (outlined regions) of desired colors.

Initialization (`__init__`)

```
def __init__(self, img, x_1, x_2, y_1, y_2, low, up):
```

- Takes:
    - `img`: the current camera image
    - `x_1, x_2, y_1, y_2`: rectangle coordinates (the window of the image to look at)
    - `low, up`: HSV color boundaries (of the colors to be processed in the image)

It stores those values and then calls `self.update(img)` to process the image.

Update function (`update`)

```
def update(self, img):
    cv2.rectangle(img, (self.x_1, self.y_1), (self.x_2, self.y_2),
(0,255,220), 1)
    self.frame = img[self.y_1:self.y_2, self.x_1:self.x_2]
    self.frame_gaussed = cv2.GaussianBlur(self.frame, (1,1),
cv2.BORDER_DEFAULT)
    self.hsv = cv2.cvtColor(self.frame_gaussed, cv2.COLOR_BGR2HSV)
```

- Draws a rectangle on the original image (for debugging/visualization).
- Crops out just that rectangle → `self.frame`.
- Applies a tiny blur (to reduce noise).
- Converts it from **BGR** (camera colors) to **HSV** (better for color detection).

So now this `Frames` object has a **pre-processed image region** ready for analysis.

Find contours (`find_contours`)

```
def find_contours(self, n=0, to_draw=True, color=(0,0,255),
min_area=50, red_dop=0):
    self.mask = cv2.inRange(self.hsv, self.low[n], self.up[n])
    if red_dop == 1:
        mask_1 = cv2.inRange(self.hsv, self.low[n+1], self.up[n+1])
        self.mask = cv2.bitwise_or(self.mask, mask_1)
```

- Creates a **mask**: keeps only the pixels in the HSV range (`low[n]`–`up[n]`), everything else becomes black.
    - Example: if `low, up` are for green → only green things show up.
- If `red_dop == 1`: combines two red ranges (since red in HSV wraps around 0°/180°).
- Finds the **contours** (shapes) of the detected areas.
- Filters out very small ones (`min_area`).
- Optionally draws them on the image (if `to_draw=True`).
- Returns the list of "good" contours.

---

pd() function

Summary

Uses a **PD controller** to make the steering smooth, not jerky. The robot looks at how much wall is on the **left** vs **right**.

- If the right wall looks closer → steer left
- If the left wall looks closer → steer right.
- If both are equal → go straight.

Measure wall area on the right side

```
contours = pd_r.find_contours(...)
if contours:
    area_r = max(map(cv2.contourArea, contours))
else:
    area_r = 0
```

- Looks in the **right region** of the camera view.
- Finds black areas (walls).
- Keeps the **biggest contour's area** (so ignores tiny specks).

Measure wall area on the left side

```
contours = pd_l.find_contours(...)
if contours:
    area_l = max(map(cv2.contourArea, contours))
else:
    area_l = 0
```

- Same as above, but on the **left side**.

Calculate the error

```
err = area_r - area_l
```

- If the **right wall looks bigge**r (closer), error is positive → robot should steer left.
- If the **left wall looks bigger**, error is negative → robot should steer right.
- If both areas are **equal** → robot is centered.

Apply PD control

```
steer = int(err * KP + ((err - err_old) / 10) * KD + 90 + steer_adj)
err_old = err
```

- **Proportional term (err * KP):** steers proportionally to how off-center it is.
- **Derivative term ((err - err_old) * KD):** reacts to how fast the error is changing (to avoid overshoot/wobble).
- **+90:** centers steering (90 = straight).
- **+ steer_adj:** adds camera correction.
- Updates `err_old` for next time.

Limit the steering range and return the final value

```
if steer > 150:
    steer = 150
if steer < 30:
    steer = 30

return steer
```

- Prevents steering from going too far left or right (to protect servo).
- Returns the final steering value

---

pd_block() function

Summary

- Looks for either a green block or a red block in a predefined region of the camera frame.
- If a block is found, it calculates how far the block is from the desired position.
- It then produces a control signal (e_block) that tells the robot how much to adjust its steering when approaching the block.

Choose color filter

- If `color == 'green'`, it looks for green contours in the image.
- If `color == 'red'`, it looks for red contours (red needs two HSV ranges, so it combines them).
- If no valid color is given, it just returns `-1` (error).

Find the largest block

- If any contours (shapes) are detected, it picks the **biggest one** (to ignore noise or small specks).
- From this contour, it calculates the **bounding rectangle** and extracts its center **x** (horizontal) and bottom **y** (vertical).

Set the "target" position

- For **red blocks**, the "target x" is the **left side** of the detection zone.
- For **green blocks**, the "target x" is the **right side** of the detection zone.
- This means the robot is supposed to go toward the left for red blocks and toward the right for green blocks.

Calculate error

- `e_x`: how far the block's x-position is from where it *should* be.
- `e_y`: how far down the block is (so closer blocks give higher y).
- These are scaled by tuning constants `K_X` and `K_Y`.
- Both are combined into one error signal `e_block`. This makes the robot steer differently depending on the block's color and the robot's driving direction.

Show debug info (optional)

- Writes the error value (`e_block`) on the video feed in either green (for green blocks) or red (for red blocks).

Return the control value

- If a block is found → returns the steering correction (`e_block`).
- If no block is found → returns `-1`.

Bill of Materials

We experimented with many boards and components before settling on our final design. The below list is for the final design only.

Part	Reference Cost (As of Sep 1, 2025 - Taxes not included)	Purchase Link (for Future Reference)
Micro:Bit		
Micro:Bit	$48.59	https://www.amazon.ca/KEYESTUDIO-MICROBIT-V2-2-KIT-Beginners/dp/B0BNKTGGB7
Micro:Bit Motor Shield (DC Motor and Servo)	$16.00	https://www.amazon.ca/dp/B0F24V8X31
Raspberry Pi		
Raspberry Pi 4	$62.95	https://www.pishop.ca/product/raspberry-pi-4-model-b-2gb/
Sainsmart Wideangle 5MP Camera 160 degree FoV	$16.99	https://www.amazon.ca/SainSmart-Fish-Eye-Camera-Raspberry-Arduino/dp/B00N1YJKFS
Other Accessories to build the car		
9v-12 Battery bank	$41.79	https://www.amazon.ca/dp/B01M7Z9Z1N
12V to 5V Step-down converter	$5.28	https://www.aliexpress.com/item/1005006655110785.html
L298N motor driver	$11.99	https://www.amazon.ca/Controller-Module-Bridge-Stepper-Arduino/dp/B0D8G2PZBB
SG90 Servo motors	$19.99	https://www.amazon.ca/dp/B0CP98TZJ2
9V Lego DC 8883M Medium motor	$7.88	https://www.aliexpress.com/item/1005006112152056.html
Short USB to USB-C cable	$10.99	https://www.amazon.ca/etguuds-Charging-Charger-Braided-Compatible/dp/B08933P982
Short USB to Micro-USB cable	$8.99	https://www.amazon.ca/Micro-CableCreation-High-Speed-Triple-Shielded/dp/B013G4D2OM
Lego Parts to build the chassis		
3D printed parts to complete the chassis		
Total	**$251.44**	