Obstacle Management

Strategies

In Obstacle challenge, the main task to around traffic lights.

So driving just a little bit to the side from them, resulting in a consistent path around the field.

It also steers away if it suspects that it may bump into field walls or parking walls.

It counts map lines to know direction of travel and how many circles it had driven.

If it detected a line yet can't see any objects, it steers to the correct side, making corner transition smoother.

And to park consistently, it first drives on the outer wall, to calibrate its position.

In Open challenge it uses a simple wall follower to drive around.

It counts map lines to know direction of travel and how many circles it had driven.

And to finish in the center, it has a small time delay after detecting the line.

Program introduction

Both programs use C++

Due to it being fast, and us having experience working with it.

It is 720 lines of code for Obstacle challenge and 430 for Open challenge.

For organization, both are broken down into functions, that do their specific part.

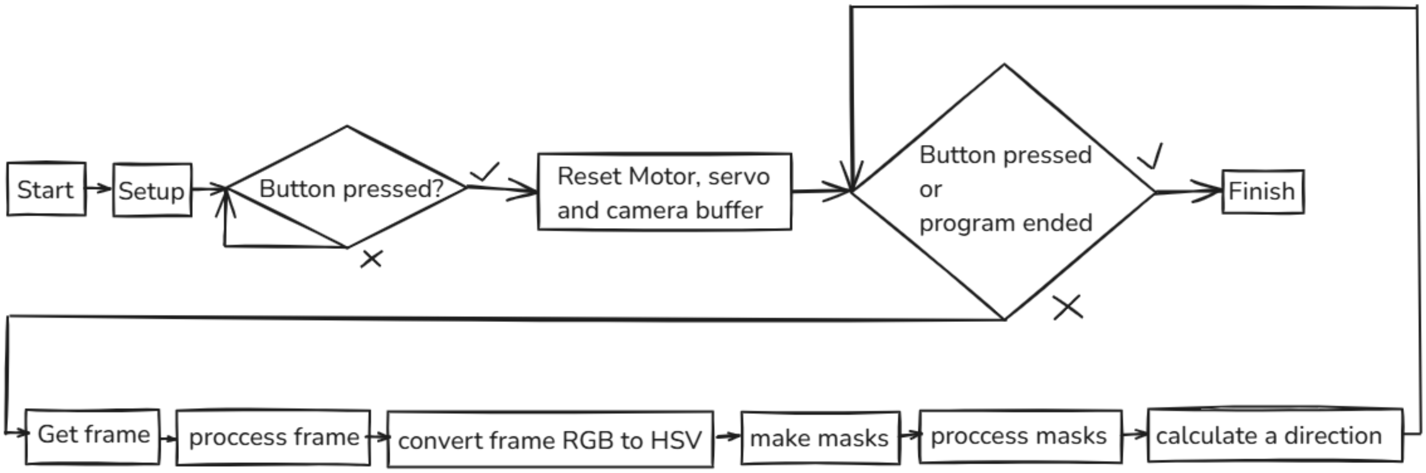
Main program statistic is its speed, measured in [ms/cycle] it is how much milliseconds it needs to do one update cycle.

In both it is 35 [ms/cycle], 33.3 of which are taken by the camera.

The detection of objects builds around an OpenCV function, that finds contours on an image.

To make it more accurate, instead of input frame it analyzes the masks of specific objects.

That reduces interference between different contour-able objects.



For map lines and walls, however, instead data is calculated by the amount of their pixels on a frame. (due to their central point position not being needed)

Both programs have 3 main sections: setup, main cycle, finish.

The setup prepares hardware and waits for button press to start the main program.

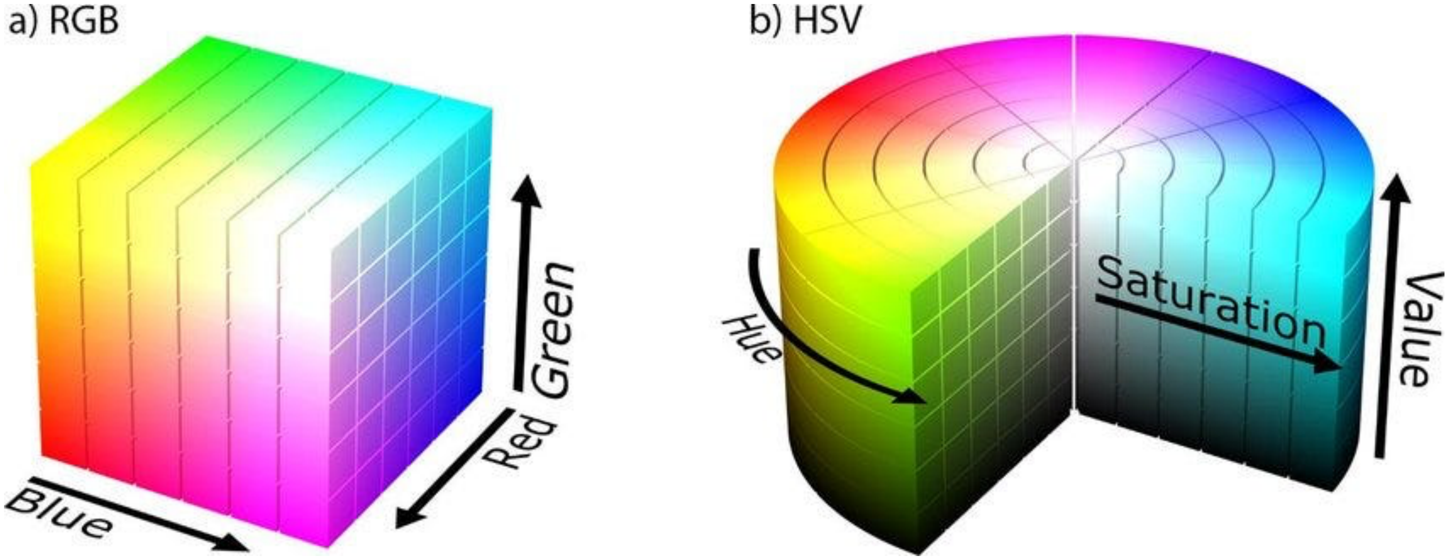
The main cycle is a loop with the main program that controls the robot.

And the finish saves data and outputs program time and speed, it also parks the robot in the obstacle challenge.

To organize the programs, they are divided into functions that prepare and process frame into usable data.

Then it uses this data to calculate the best path in the current frame.

Processing input image



To create masks the frame is converted from RGB to HSV color format.

That is to separate colors easier, using Hue component.

Also Saturation and Value components are used to separate colors with no difference from Hue – any shade of gray.

The frame itself is processed.

It is cut in half to remove the outside view of the field.

And then scaled in half to make program faster, using square cube law.

Walls are converted into percentages of the wall pixels on the left and right sides of the frame.

Map lines are converted into their respective pixel amount.

And for traffic lights, program utilizes ‘divide and conquer’ approach by only dealing with the biggest traffic light.

Others are smaller, meaning they are farther from the robot, so they can be dealt with later.

The data is converted into an array {x, y, area, type}.

The only data needed for a P algorithm. (only P from PID, since the target isn’t continuous so I and D would confuse it).

The OpenCV function returns an array of bounding points.

While sorting them out, program also removes noise.

Firstly, they are converted into bounding boxes to calculate area.

Secondly, if the area is smaller than minimum allowed (meaning it is noise) they are ignored.

Thirdly, if the width is greater than height they are also ignored (that means that it is a big noise group, as traffic objects are tall).

Finally, if it is bigger by visible area than other detected traffic lights (possibly none) it is selected as the target.

To communicate data found and program state (for debug and presentation) it uses a RGB LED. The signals are:

Black	Gray	Red	Green	Orange	Blue	Magenta
Program is off	Nothing detected.	Red object is targeted	Green object is targeted	Orange line detected	Blue line detected Or program start / finish	It is parking

They have a display priority of:

Gray < Red/Green < Orange/Blue < Magenta < Black

Obstacle challenge main cycle

After processing input image, the program needs to decide the direction from it.

The data it has:

- Biggest traffic light, as an array {x, y, area, type}
- Parking walls, as an array {x, y, area}
- Walls, as percentages of their pixels on left and right sides of the frame.
- Map lines as the amount of their pixels.

First the program finds the best direction for avoiding the traffic objects.

It steers just a bit to the left or right of the object.

That results in a path that generally doesn't collide with the walls of the field.

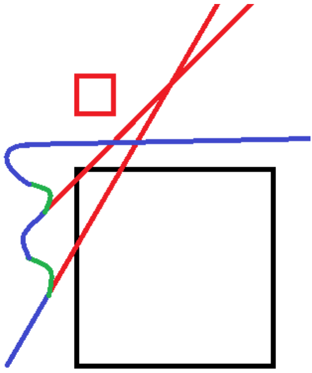
It uses a P algorithm to do that.

If no traffic object is detected, it checks the parking walls to avoid hitting them using a similar P algorithm.

It only steers away from them, since robot doesn't need to be as close as possible to them.

By the strategy described above, it cannot hit the parking walls due to traffic objects not allowed near them.

Also If at corner no traffic light is seen, it steers a bit to the correct side to smooth transition between sides of the field.



The algorithm isn't ideal, sometimes that path can go through the wall.

If the percentage of the wall pixels on a side of the wall is too big, it overrides the direction to steer away as fast as possible (max allowed servo angle).

If both sides have too big of a percentage, it prioritizes one by direction of travel.

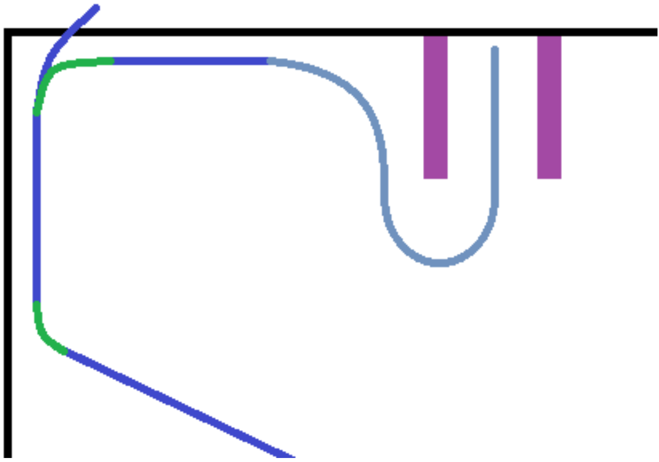
This algorithm can drive robot around the field, but it doesn't know when and how to stop.

To know its position and direction of travel robot uses map lines.

If direction is not known, it counts as if it is clockwise.

Every time the robot sees a line, it increases corner count. A full cycle of the square field is 4 corners.

If corner count is 8 (2 laps) and the last traffic light was red, it performs a timed sharp U turn, that overrides everything else. The direction of the turn is determined by the direction of travel, and in theory shouldn't hit walls.



And if the corner count is 12 (3 laps) it begins the parking alignment procedure. During which every direction calculation done before are ignored. The speed is lowered to make parking more consistent.

First it drives straight forward into the wall

Then when the sum of the percentages is high enough it begins to drive around the outer wall, using a P algorithm that is similar to the one found in Open challenge.

Using direction of travel, it chooses a wall to drive around.

The difference from the Open challenge is that to not bump into wall on corners, it checks the percentage of the other side.

If it is too big it overrides the P algorithm to turn at max angle.

While it drives around the outer wall, it checks the area of parking walls. if big enough, it ends the program.

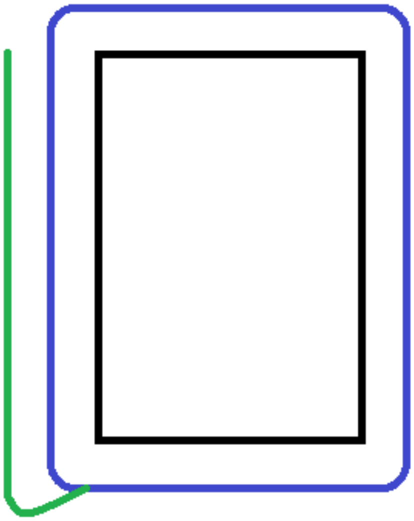
After the main cycle part of the program ends, finish part checks what was the cause.

If it is the button it doesn't do any moves, and starts saving / printing data.

However, if the program ended by itself, meaning it robot is successfully aligned near parking walls it parks.

The maneuver is hard-coded, it consists of two turns and a ride forward.

Open challenge main cycle



Open challenge program is the cut version of the Obstacle challenge program.

It has the same functions, but not all of them included.

The data it gets is:

- Walls, as percentages of their pixels on left and right sides of the frame.
- Map lines as the amount of their pixels.

The lap count is calculated using the same approach as in Obstacle challenge.

It calculates steering using a P algorithm to drive around the central wall.

At start it doesn’t know its direction so it drives in the center between both walls.

To finish in the correct place, it has a little delay before stopping the program after the final corner is detected.