A path planning for Rob.

Stefano Leonardi
Enrico Saccon

University of Trento

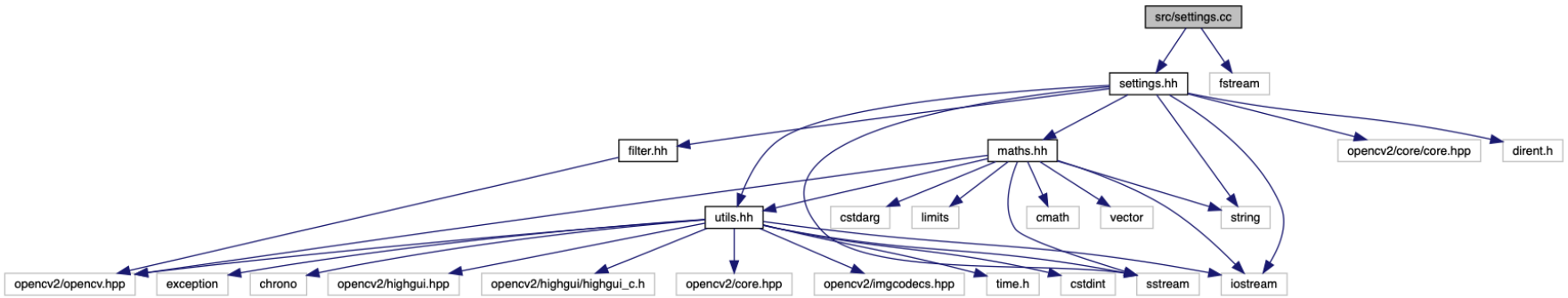
Steps

- Calibration
- Unwarping
- Detection
- Path planning
- Dubins interpolation
- Localization

But first Settings

`Settings.xml`

Settings

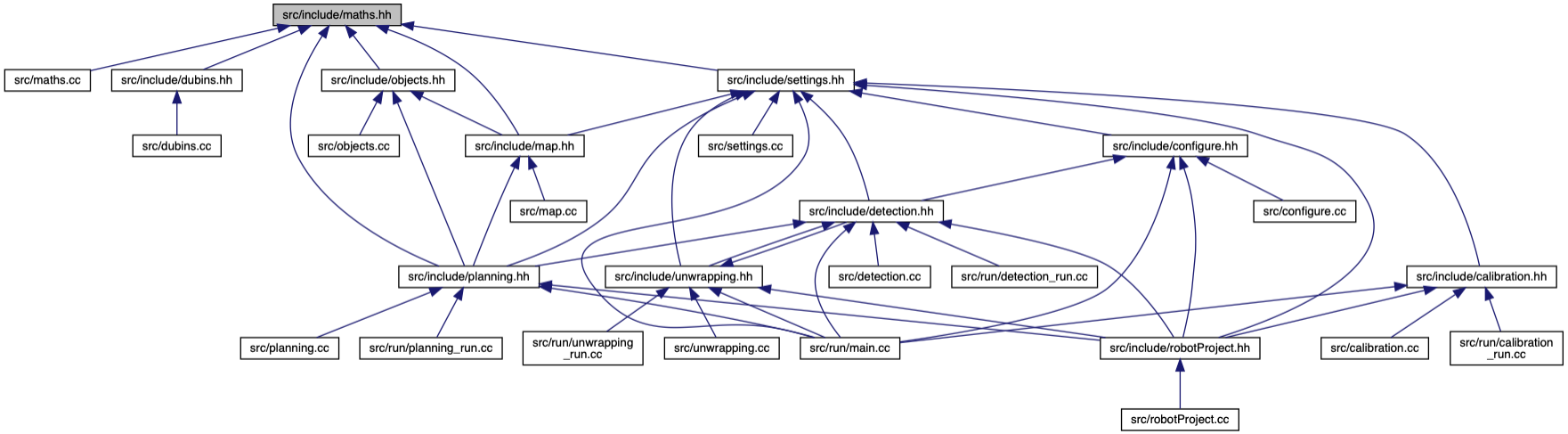


Link to settings.hh

Support objects – maths.hh

- Angle: class to handle both RAD and DEG angles.
- Tuple: an extension of vector.
- Point2: a simple point in the cartesian plane, with useful functionalities.
- Configuration2: a `Point2` with an orientation as `Angle`.

maths.hh


src/include/maths.hh
src/maths.cc
src/include/dubins.hh
src/include/objects.hh
src/include/settings.hh
src/dubins.cc
src/objects.cc
src/include/map.hh
src/settings.cc
src/include/configure.hh
src/map.cc
src/include/detection.hh
src/configure.cc
src/include/planning.hh
src/include/unwrapping.hh
src/detection.cc
src/run/detection_run.cc
src/include/calibration.hh
src/planning.cc
src/run/planning_run.cc
src/run/unwrapping
_run.cc
src/unwrapping.cc
src/run/main.cc
src/include/robotProject.hh
src/calibration.cc
src/run/calibration
_run.cc
src/robotProject.cc

Support objects – filter.hh

Class attributes:

- low_h: Lower value for hue
- low_s: Lower value for saturation
- low_v: Lower value for value
- high_h: Higher value for hue
- high_s: Higher value for saturation
- high_v: Higher value for value

Class functions:

- High();
- Low();

Each returns a `Scalar` with the correspondent values.

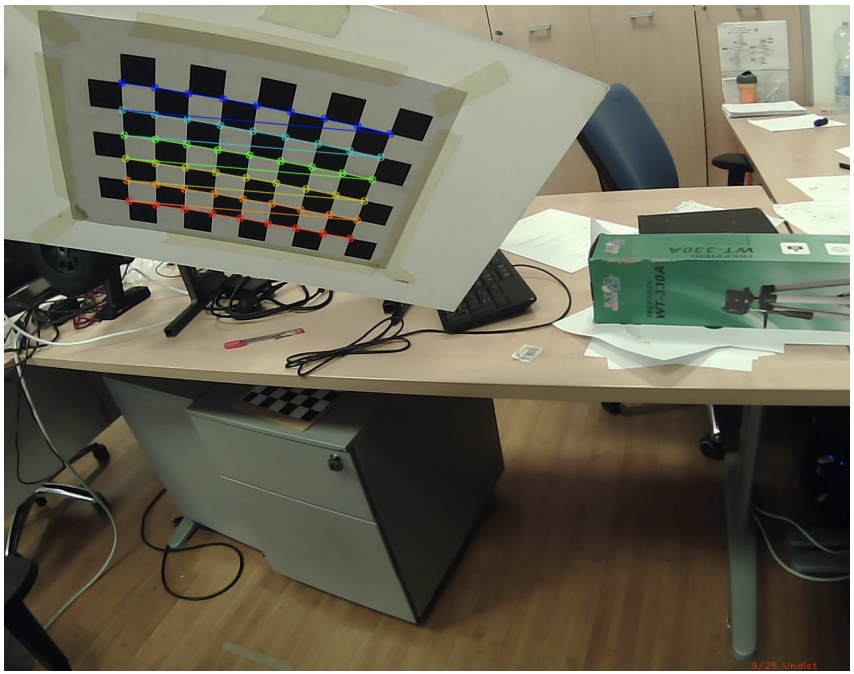
Calibration

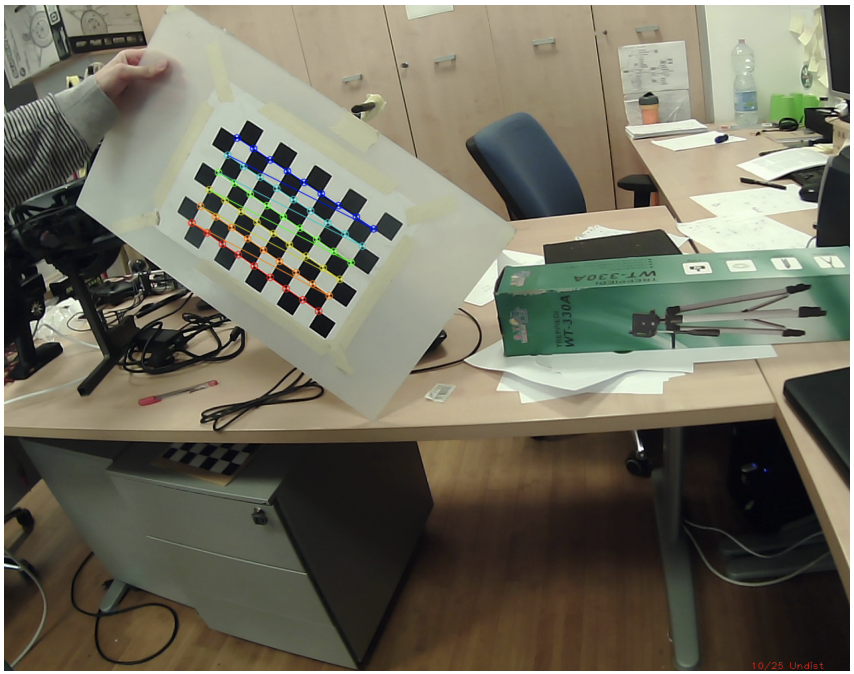
Given a set of pictures taken with a camera, the object of this step is to find the camera matrix A and the distortion coefficients $\dot{d}$.

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

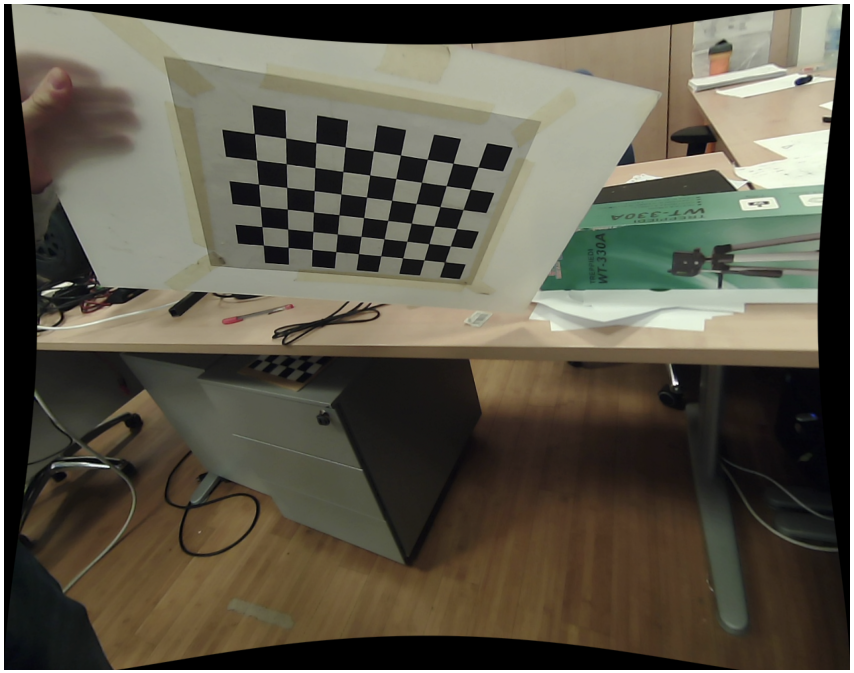
$$\begin{aligned} x_{distorted} &= x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ y_{distorted} &= y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ x_{distorted} &= x + \left[2p_1 xy + p_2\left(r^2 + 2x^2\right)\right] \\ y_{distorted} &= y + \left[p_1\left(r^2 + 2y^2\right) + 2p_2 xy\right] \\ \dot{d} &= (k_1, k_2, p_1, p_2, k_3) \end{aligned}$$

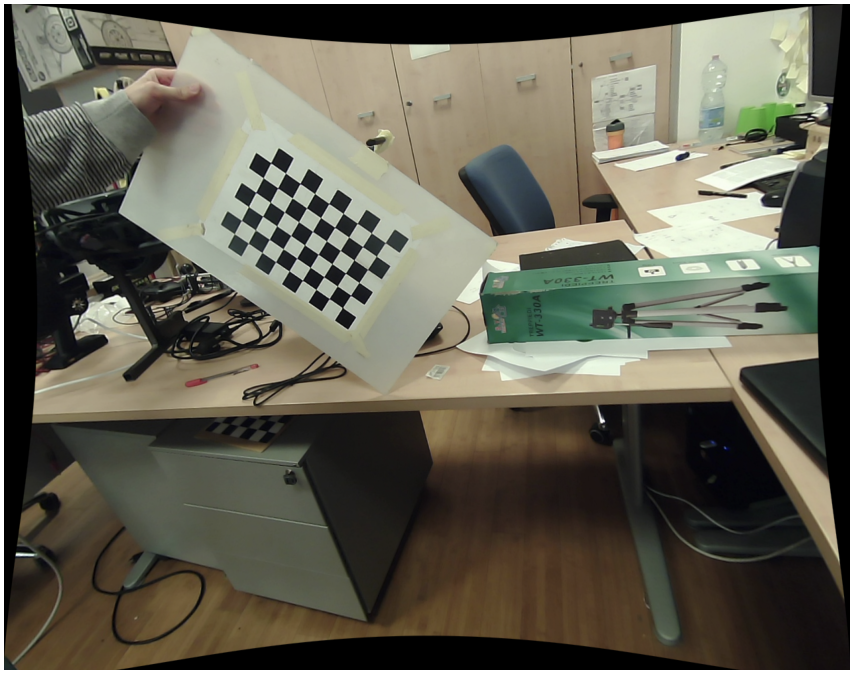
Calibration





Calibration





$$A = \begin{bmatrix} 8.4247565095622963 \times 10^2 & 0 & 6.3709750745251142 \times 10^2 \\ 0 & 8.4247565095622963 \times 10^2 & 4.9404840840221556 \times 10^2 \\ 0 & 0 & 1 \end{bmatrix}$$

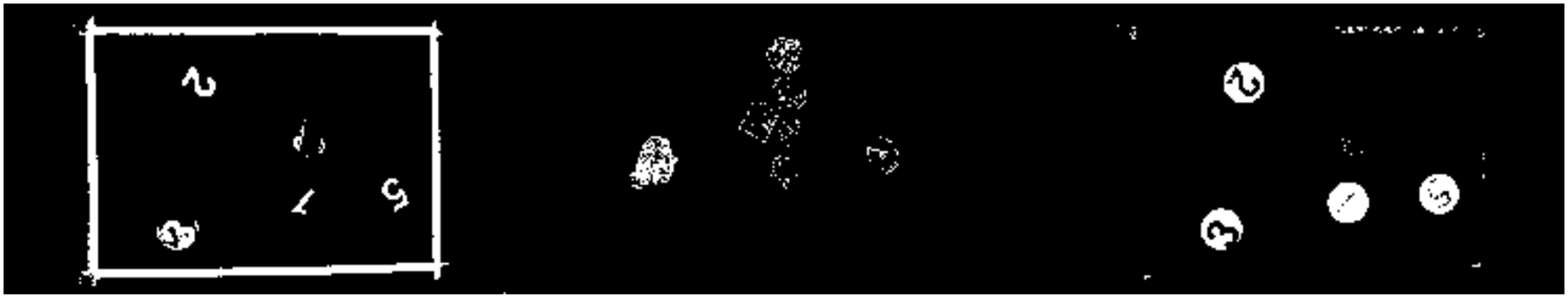
$$\dot{d} = (-2.5214446354851400 \times 10^{-1}, 7.2467634259951161 \times 10^{-2},$$

$$-3.7212601356153754 \times 10^{-3}, 4.3313659139950872 \times 10^{-4}, 0)$$

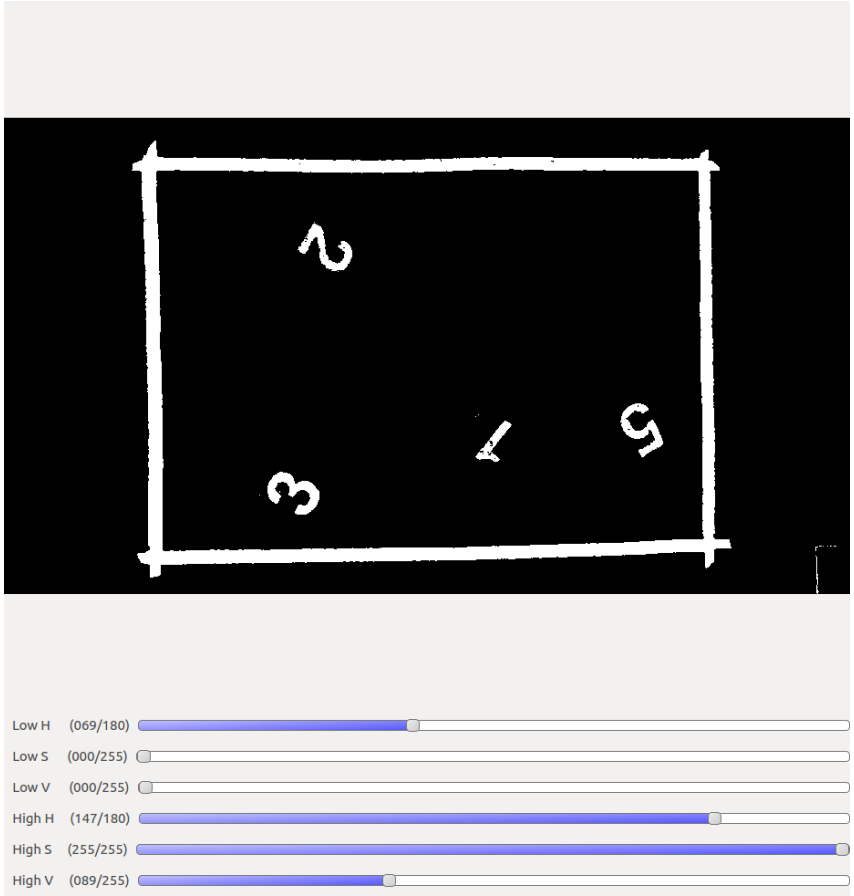
Configure

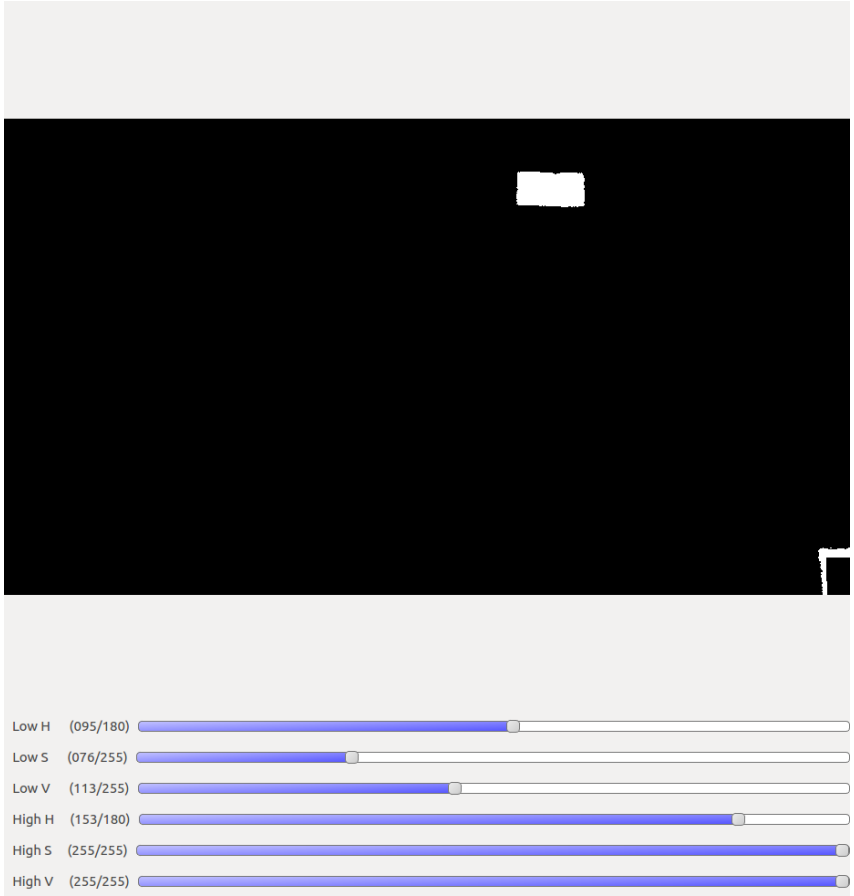
Function to change the filters if the result of their use doesn't look good.

Configure

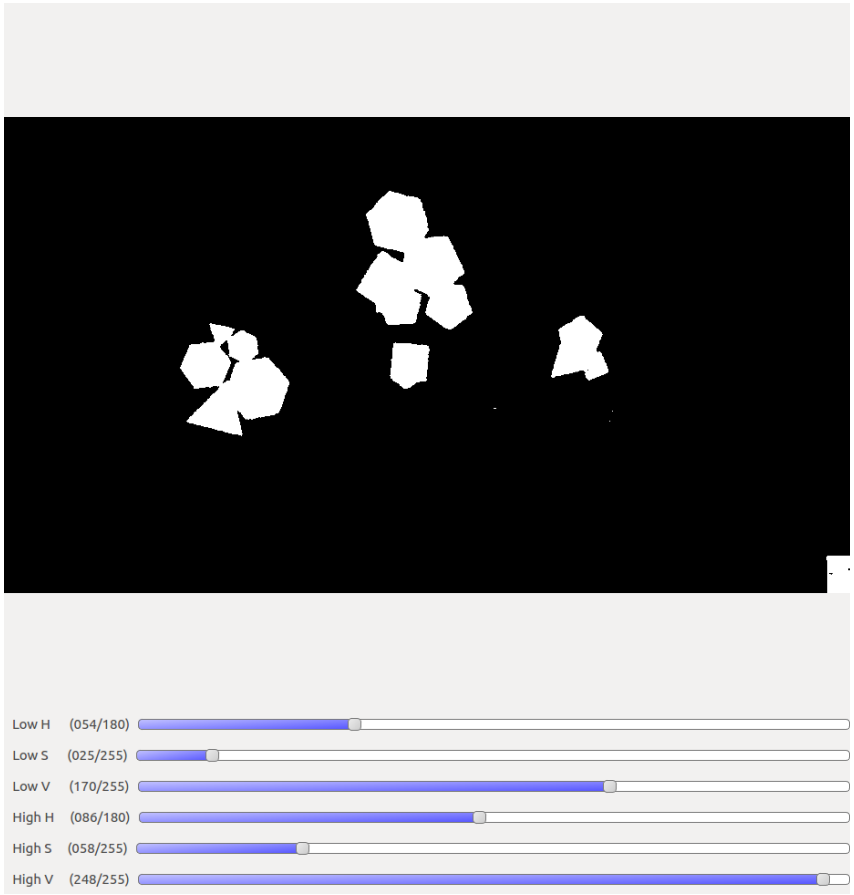


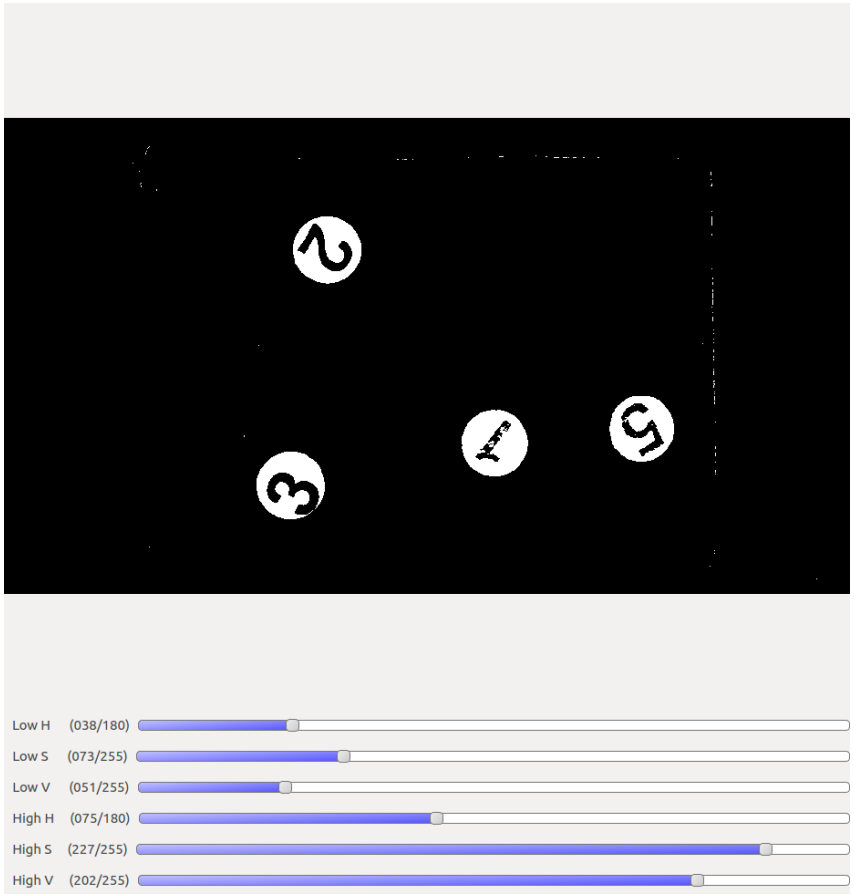
Configure


Low H (069/180)
Low S (000/255)
Low V (000/255)
High H (147/180)
High S (255/255)
High V (089/255)

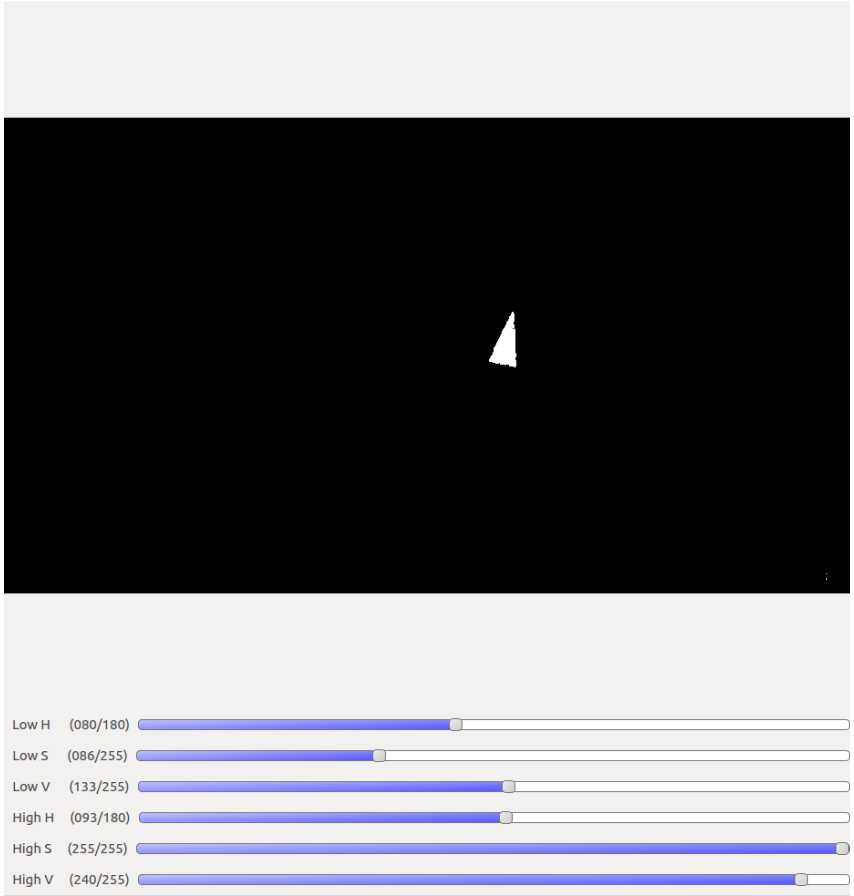

Low H (095/180)
Low S (076/255)
Low V (113/255)
High H (153/180)
High S (255/255)
High V (255/255)

Configure


Low H (054/180)
Low S (025/255)
Low V (170/255)
High H (086/180)
High S (058/255)
High V (248/255)


Low H (038/180)
Low S (073/255)
Low V (051/255)
High H (075/180)
High S (227/255)
High V (202/255)

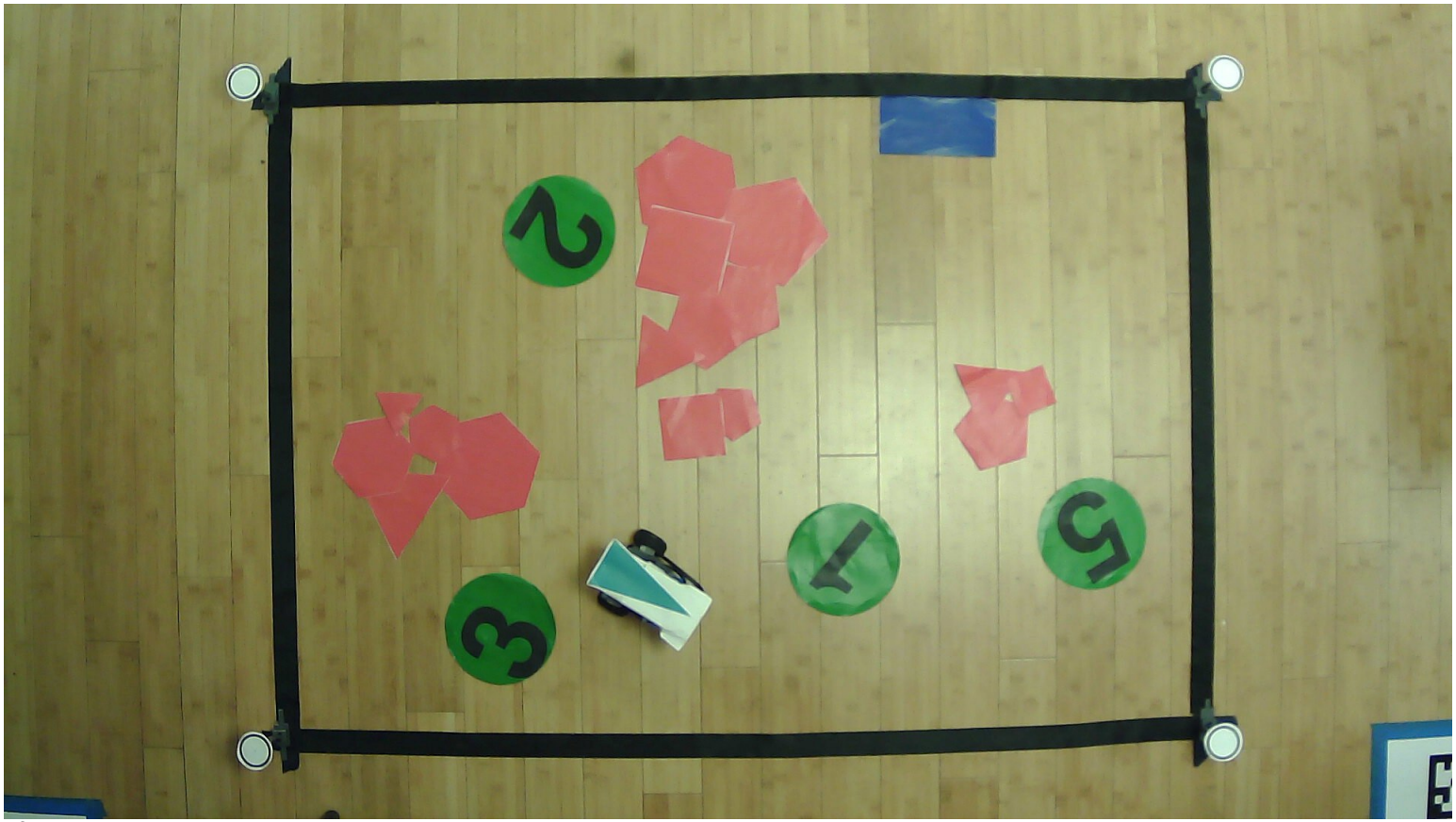
Configure



Once finished setting them up, the filters are saved in settings.xml and available during all the execution.

Unwarping

Once the camera matrix and the distortion coefficients are known and the filters are nice, the next step is to focus on the target, that is the circuit, and then to "un-distort" it from this:

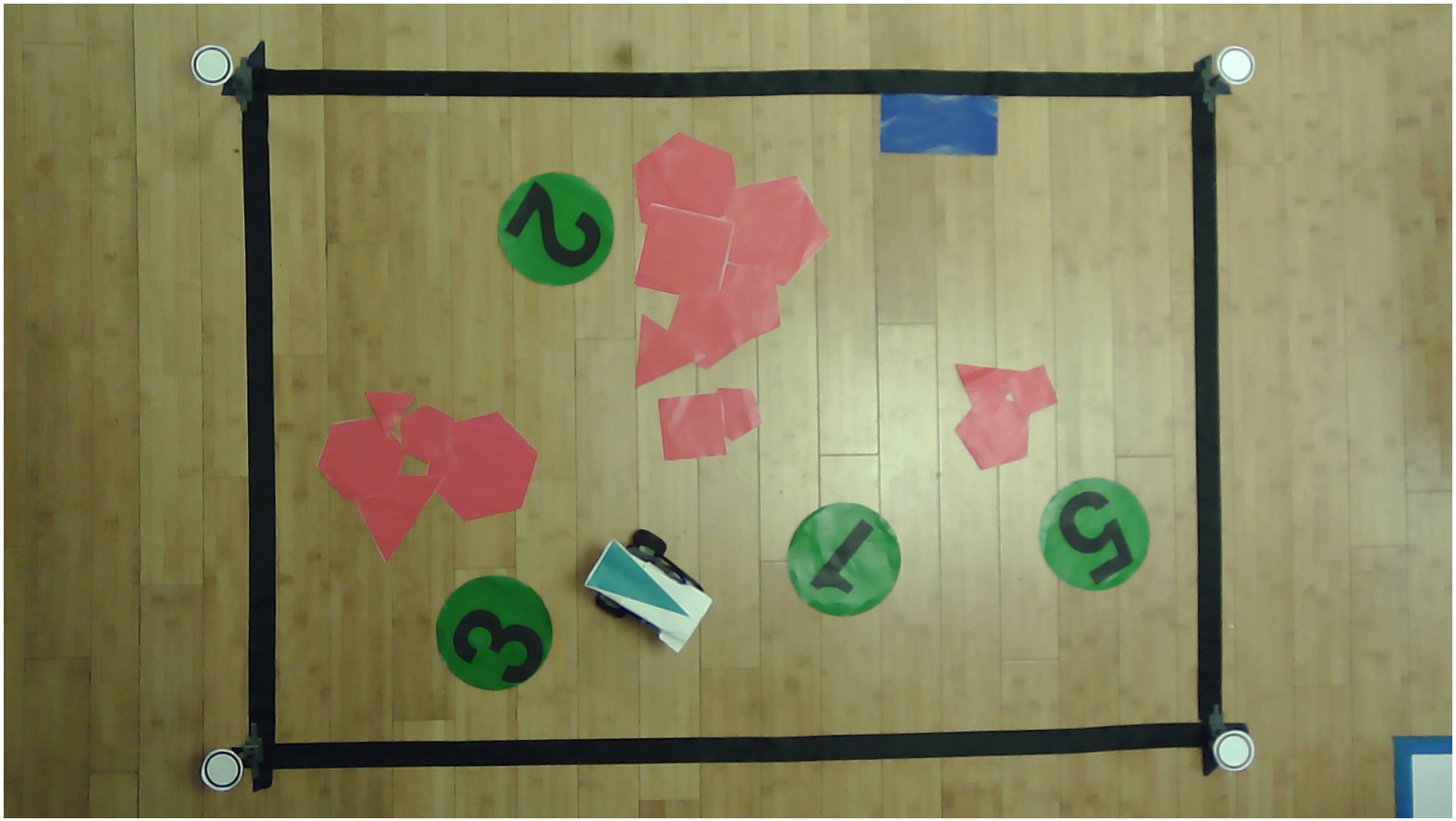


Unwarping

To this:

Unwarping

Straighten the image using the distortion coefficients.




```
loadCoefficients(calib_file, camera_matrix, dist_coeffs);
undistort(or_img, fix_img, camera_matrix, dist_coeffs);
```

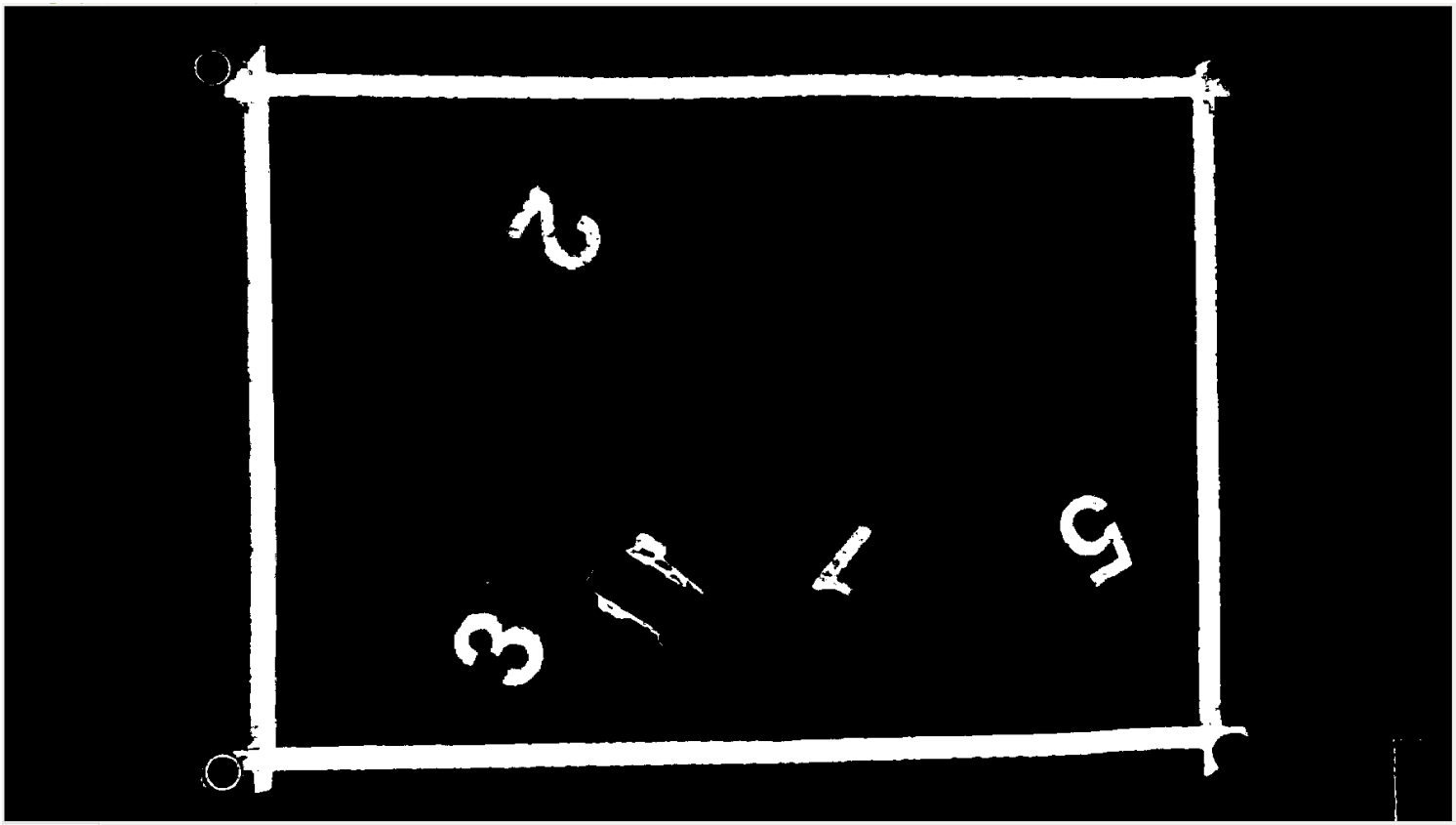
Unwarping

Then the image is converted from RGB to HSV.


```
cvtColor(fix_img, hsv_img, COLOR_BGR2HSV);
```

Unwarping

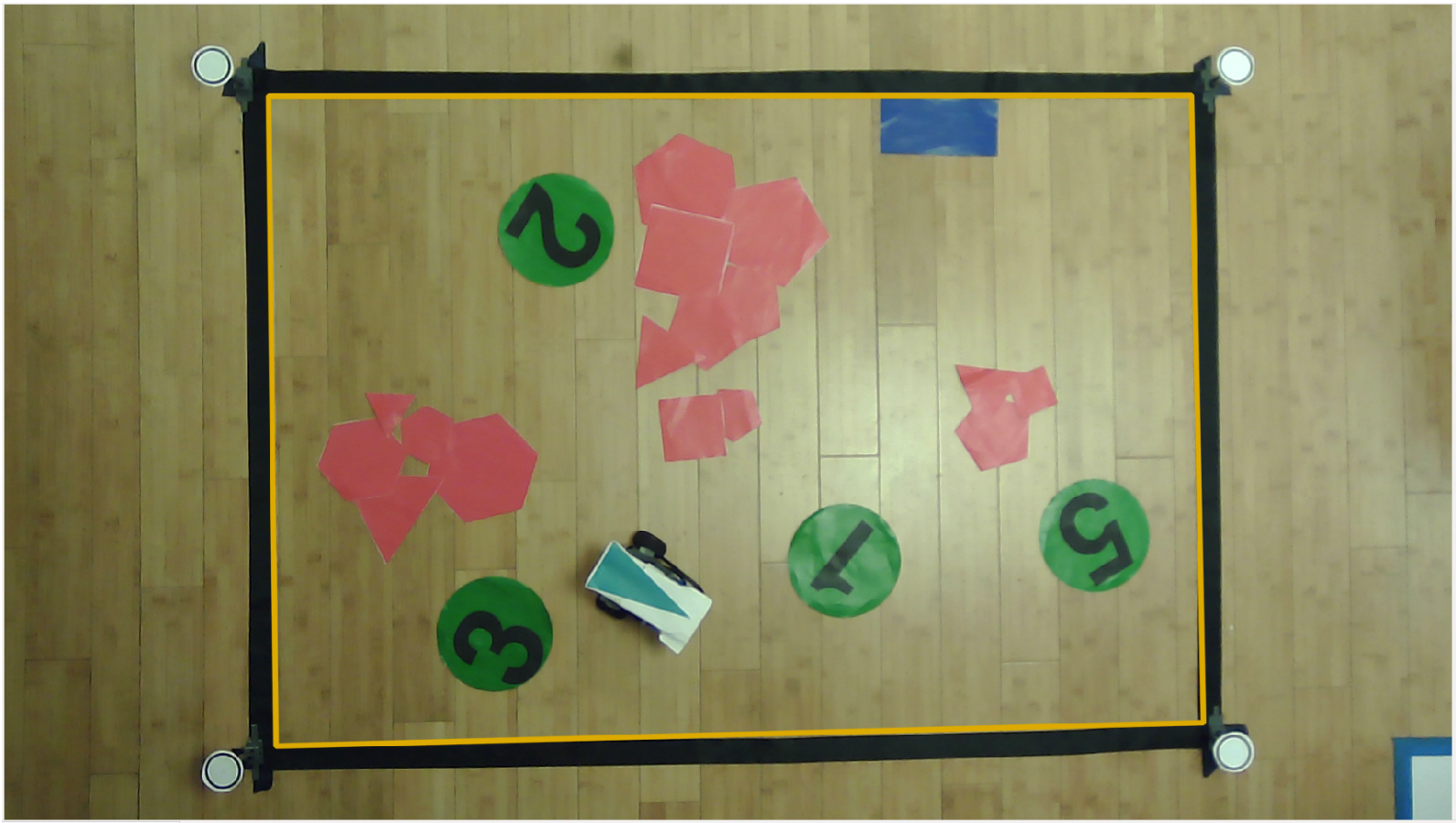
A black filter is added to highlight the black lines and areas.



```
Filter blackFilter = sett->blackFilter;
inRange(hsv_img, blackFilter.low(), blackFilter.high(), black_mask);
```

Unwarping

Then the largest area enclosed by black lines is selected.



Unwarping

At the end the image is cropped and turned.

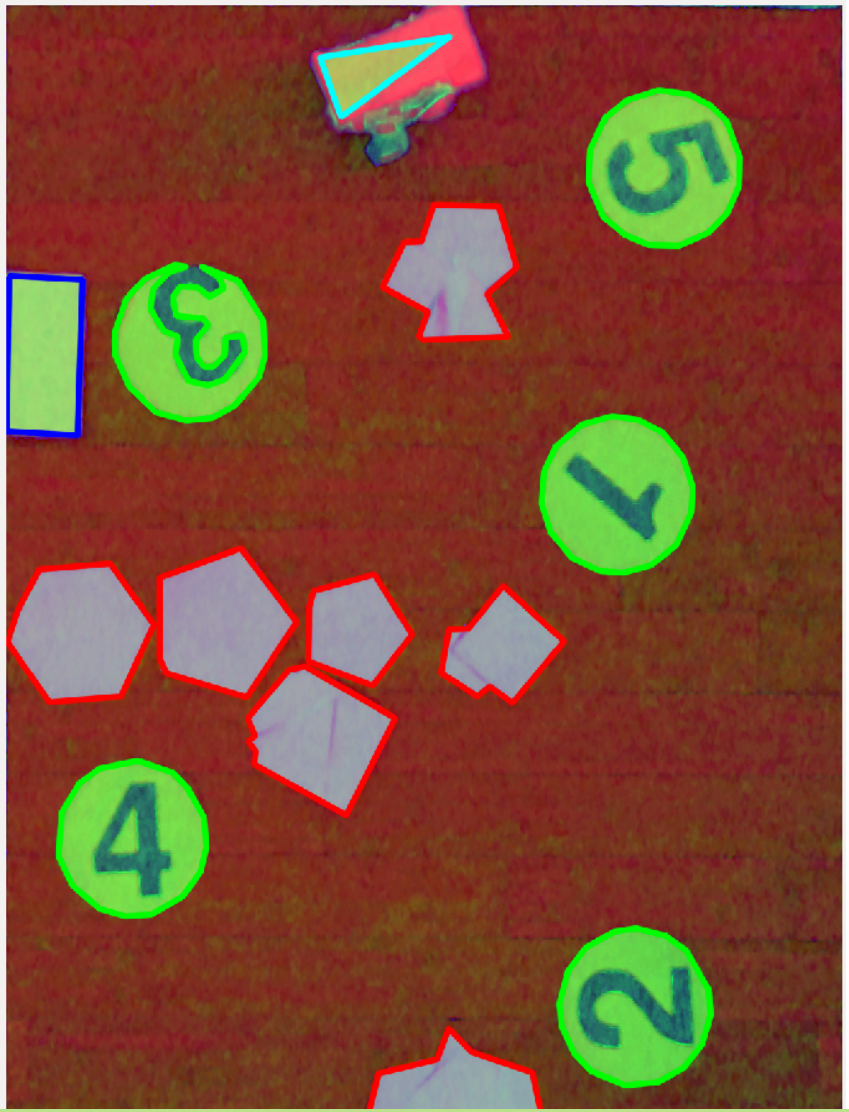
Detection

The detection phase consists of three more steps:

- Shape detection;
- Digit detection;
- Robot detection (aka localize).

Shape detection

The shape detection consists of scanning the image to locate the various obstacles, targets and the arriving point.
The various items are going to be stored in a file with the coordinates of their vertexes as convex hull.



Digit detection

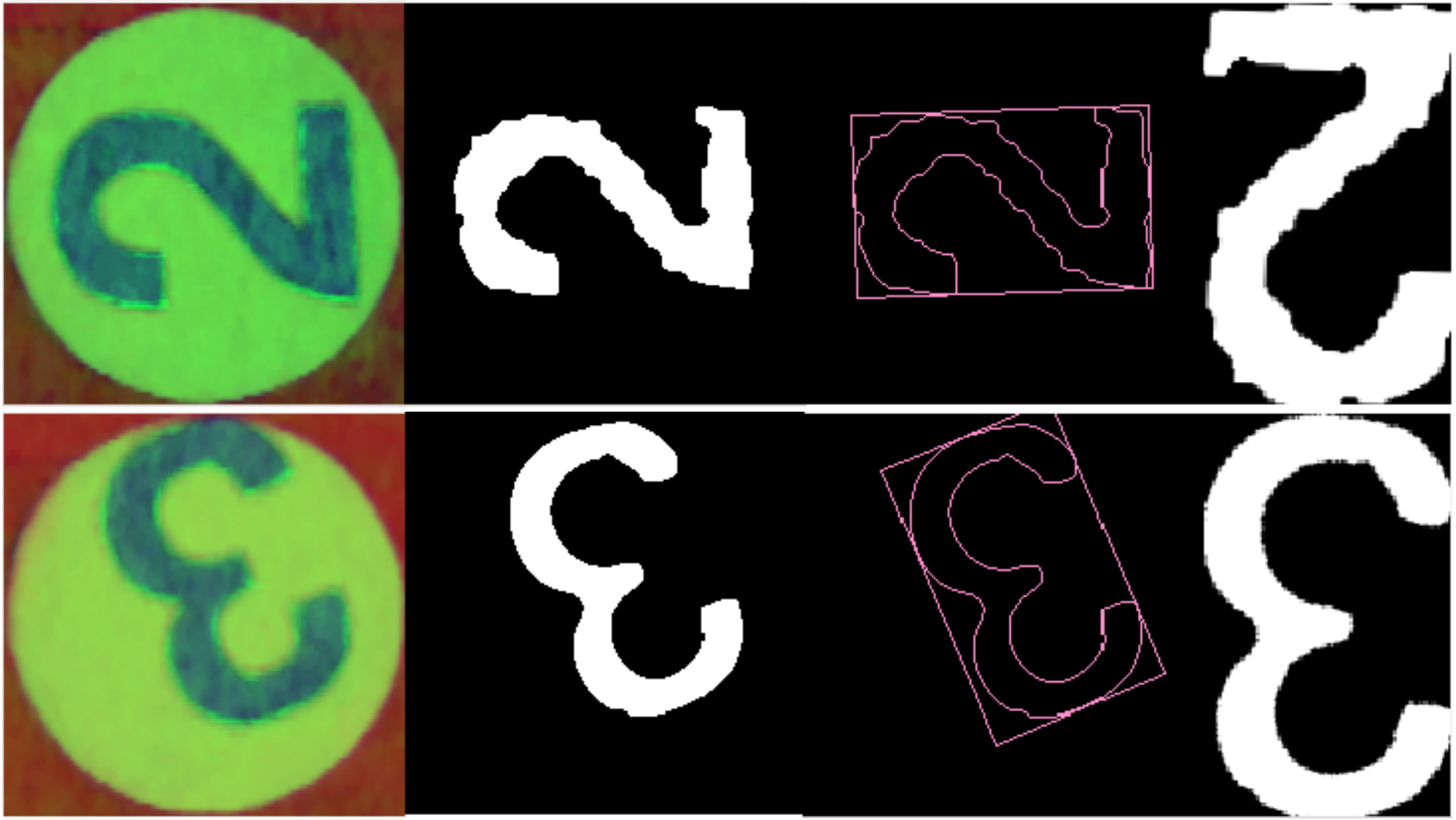
The goal of this phase is to consider the area around the targets, which is our ROI, and try to identify the number inside it.
In order to have better results the area around each target is:

- Cropped;
- Resized to 200 $px \times 200$ px;
- Given a black filter thought for the victim;
- Passed through a function (`erode_dilation()`) which aims at removing noise and isolating the number;

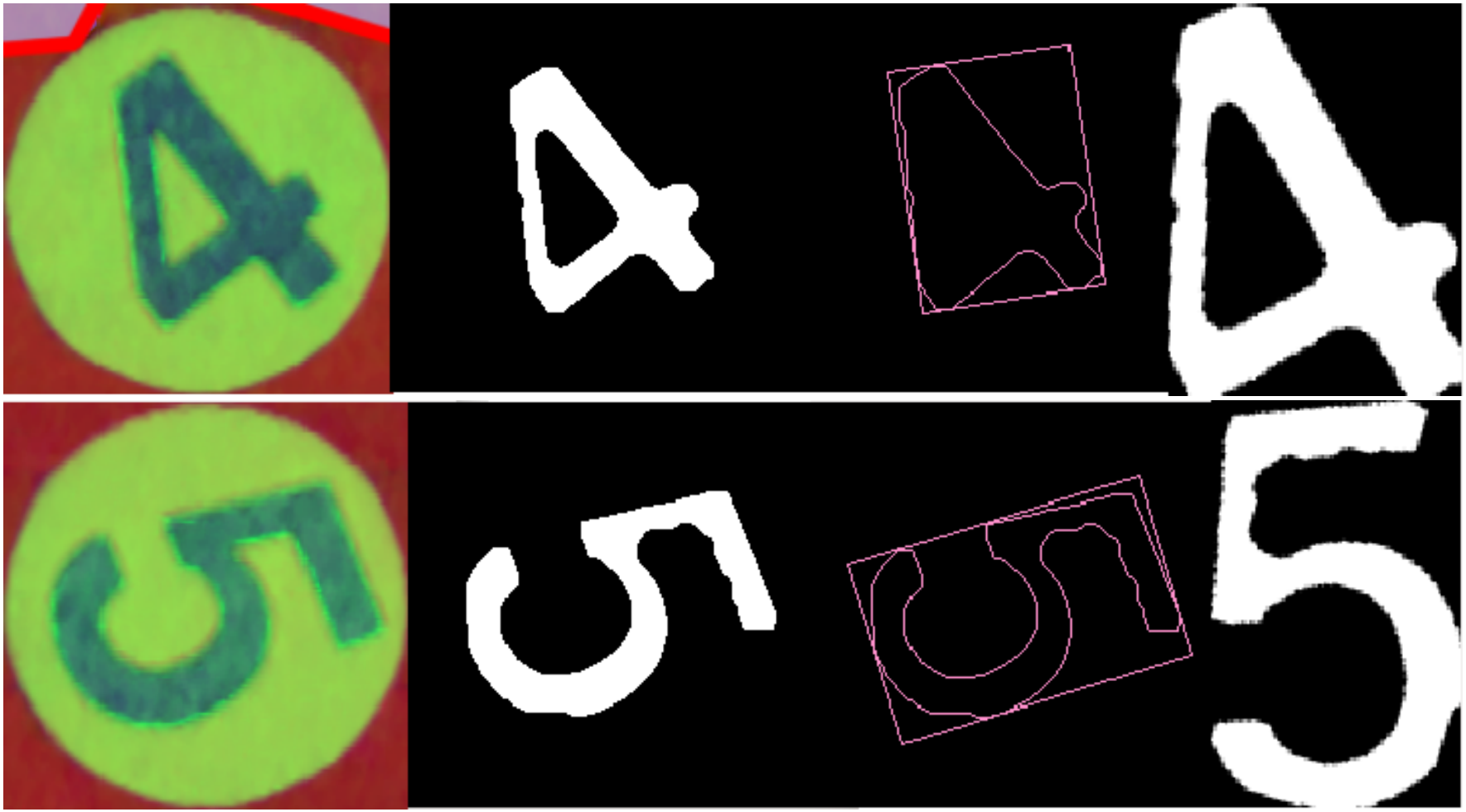
At last the digit detection is done with templates since the success rate has been greater than using Tesseract.

Number templates

Digit recognition

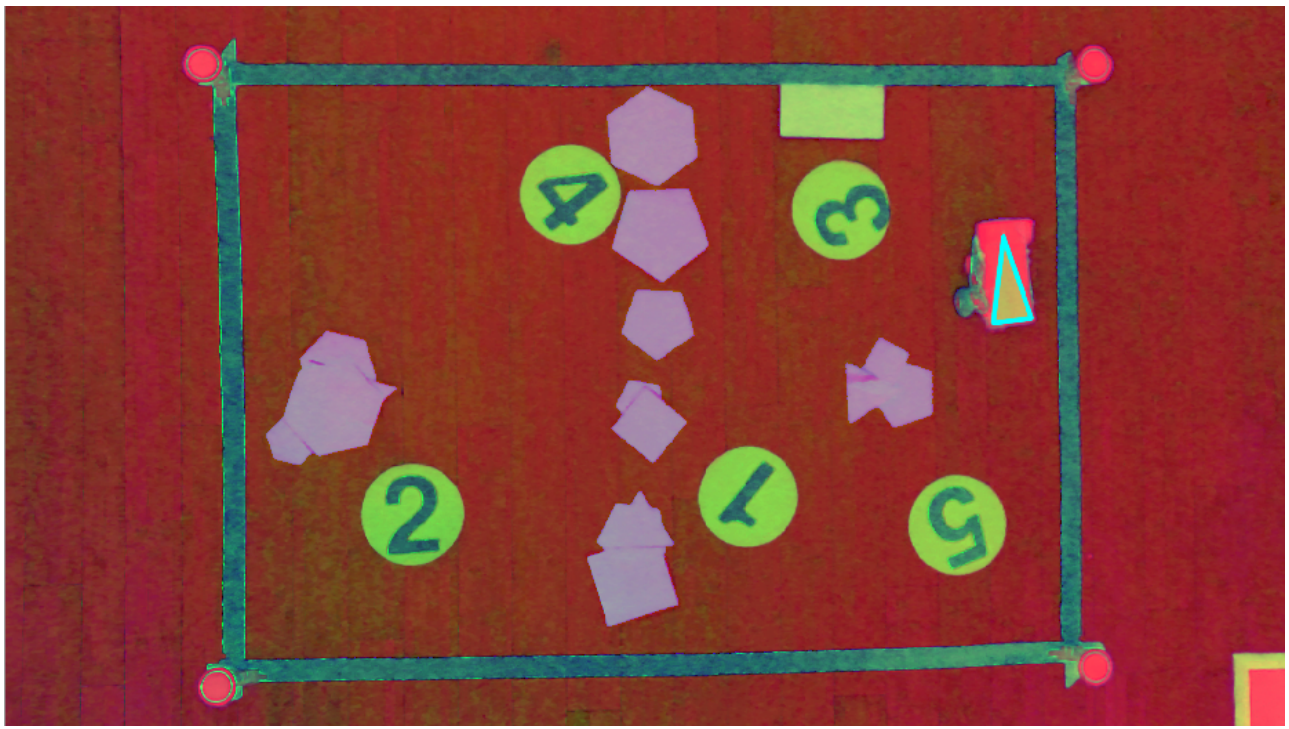


Digit recognition



Robot detection aka localize

The barycentre of the robot is calculated as the mean of the *hopefully* three vertexes of the triangle.
The function is pretty fast since it works constantly under $20ms$.



Robot detection aka localize

Since the robot is not on the same plane as the arena, a first matrix transformation is needed to be able to use the same coordinates.

$$\begin{bmatrix} 0.004009178727557017 & 1.058423184075202 & -85.87660834428323 \\ -1.129379950738086 & 0.016010838996287 47 & 1828.394707270977 \\ -2.212284394884848e-05 & 6.807241218637246e-06 & \end{bmatrix}$$

```
getPerspectiveTransform(...);
```

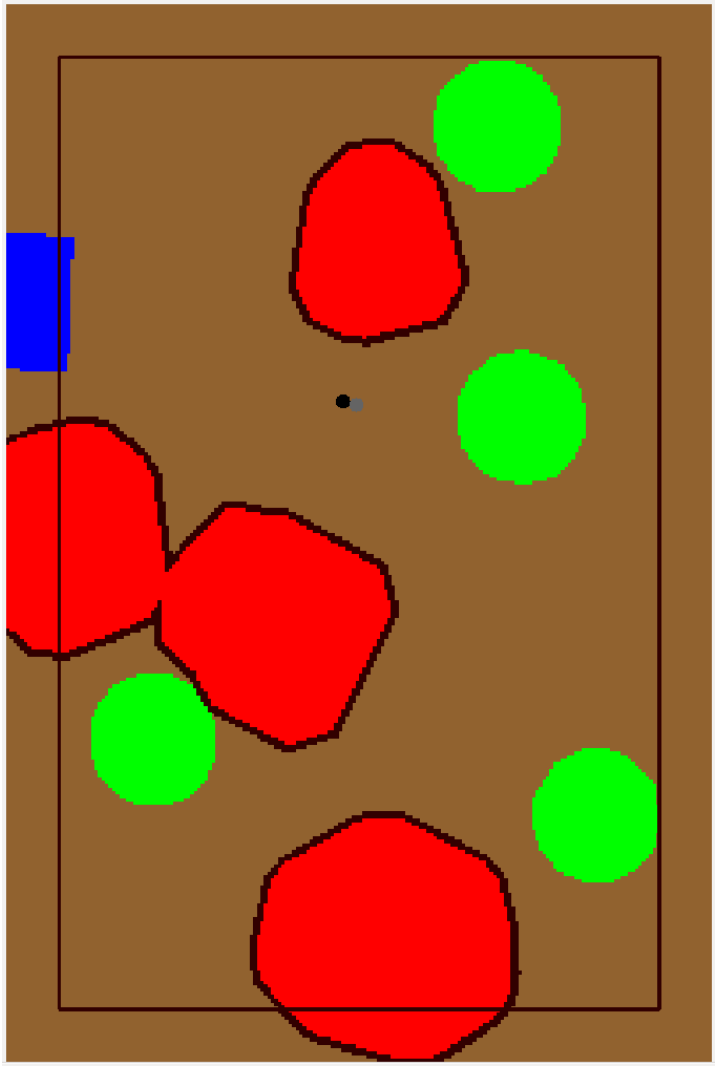
Map creation

Map is represented as a matrix of cells. Each cell can be of 5 different types:

- FREE
- VICT
- OBST
- GATE
- BODA

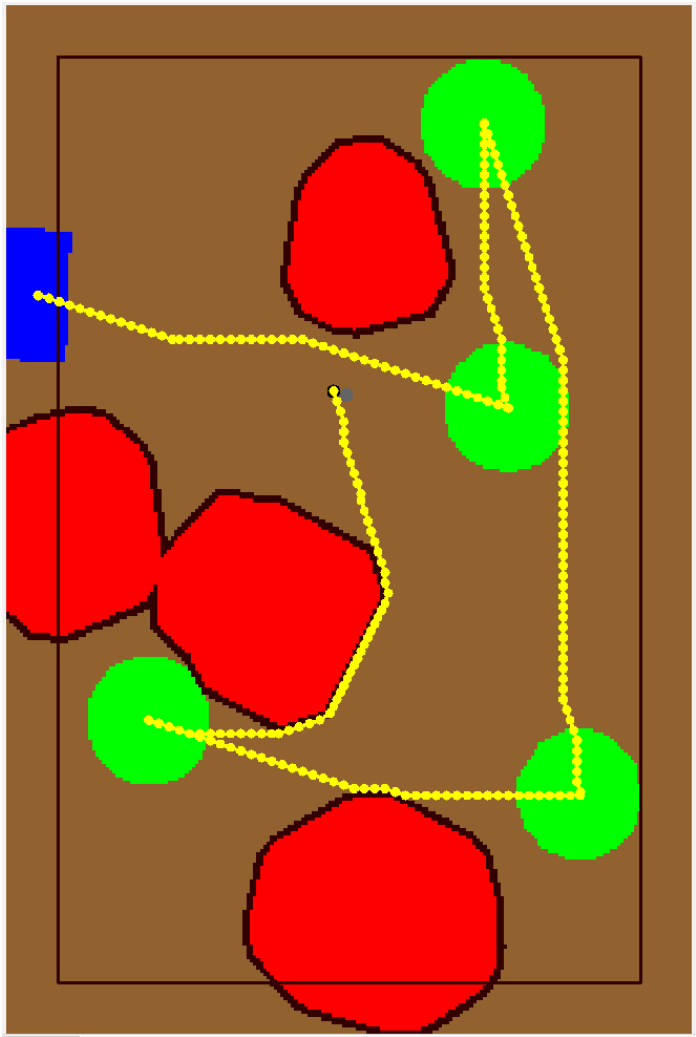
Each cell gets their type at the creation of the map by loading objects from file and then eventually applying an offset to the obstacles in order to consider the robot as a point.
Link to map.hh

Map creation



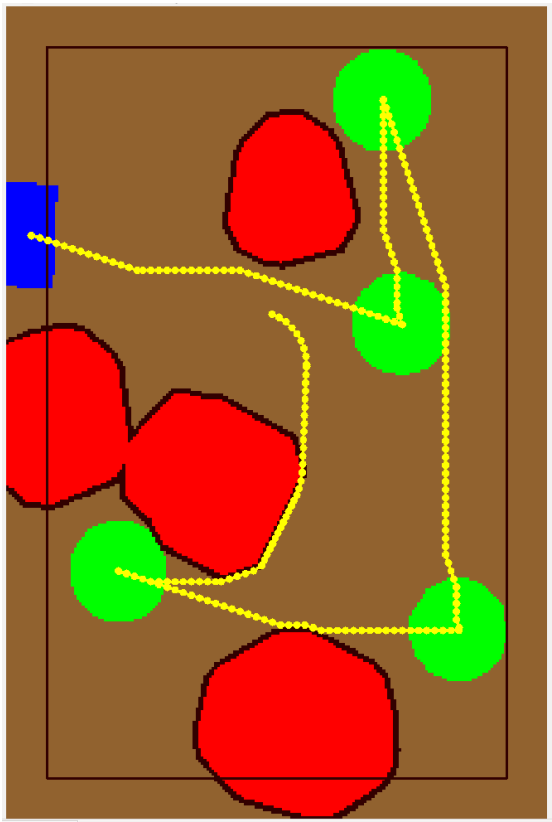
Path planning

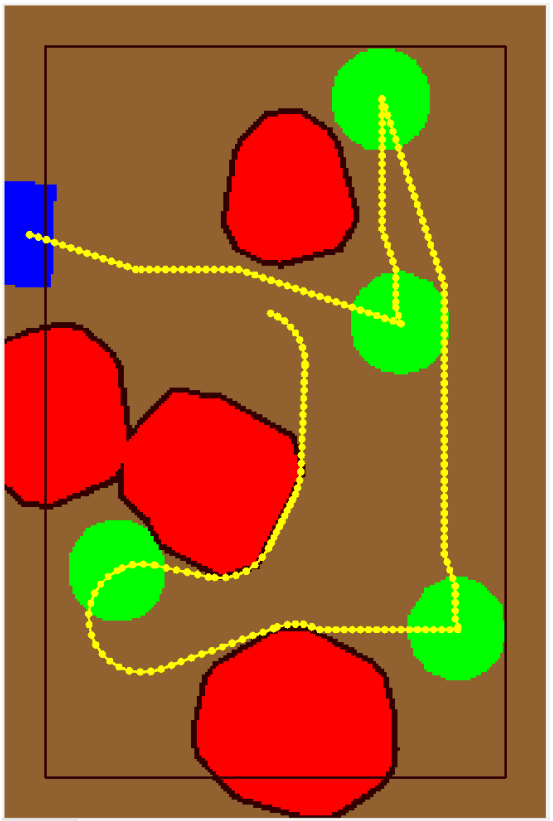
Once the map is created, we look at it as a graph and then compute a BFS to connect the robot to the gate through the victims in the right order.
Here the direction of the robot means nothing: the total minimum path is always the sum of distance between each couple of cells.
The complexity of this algorithm is $O(N + N(2R+1))$ where N=nodes and R=range=3.



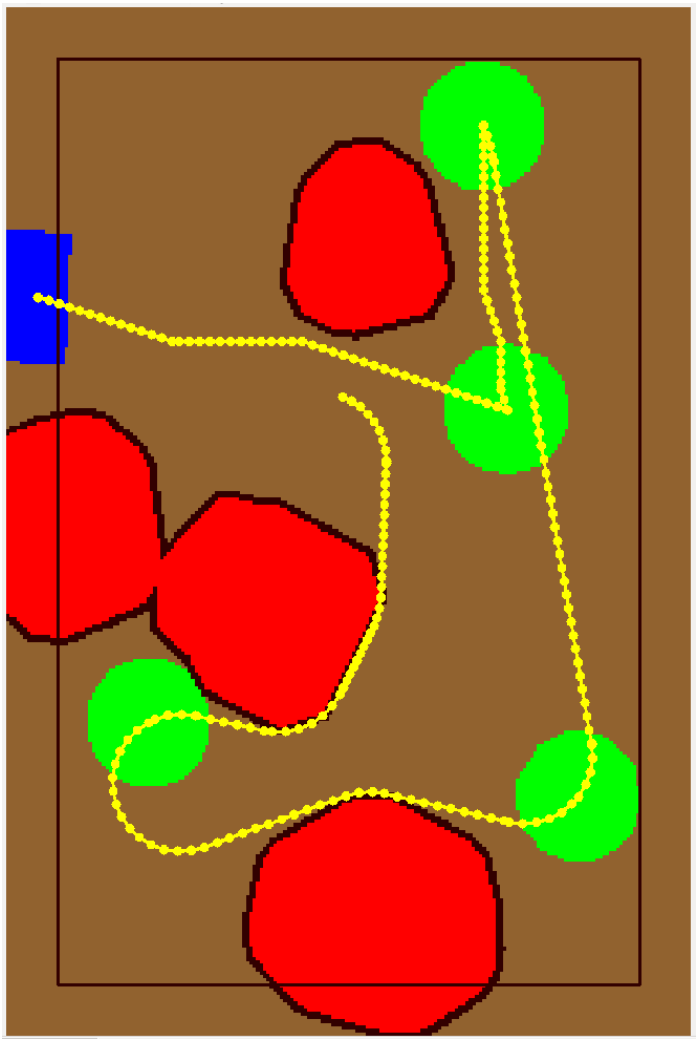
Dubins

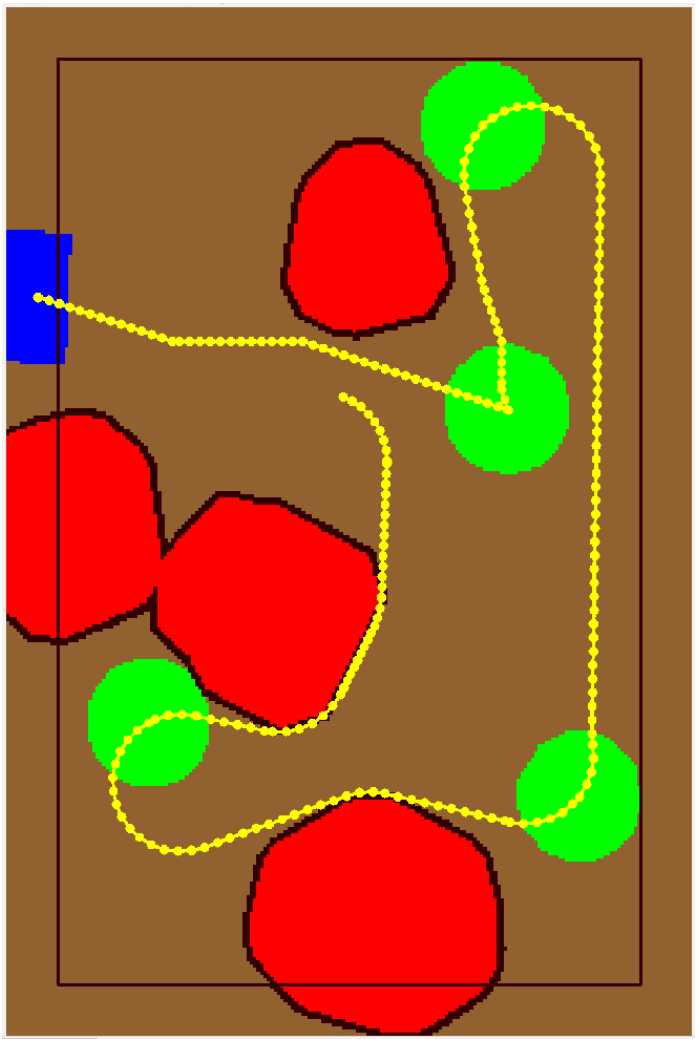
Once a min path is been computed, then we try to smooth it by applying Dubins curves.



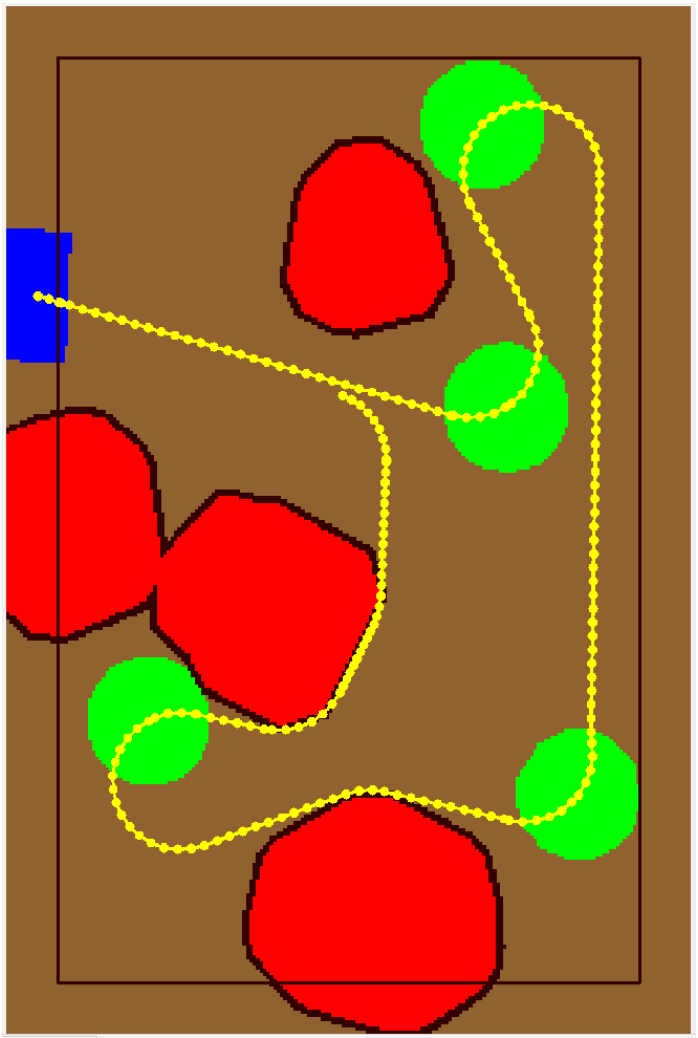


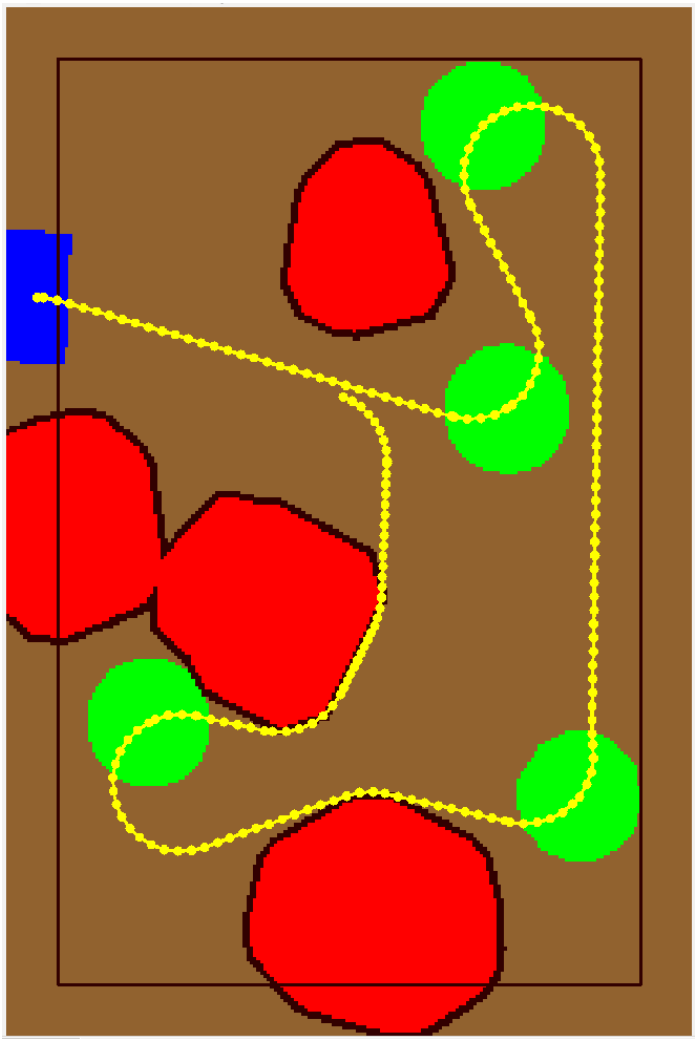
Dubins



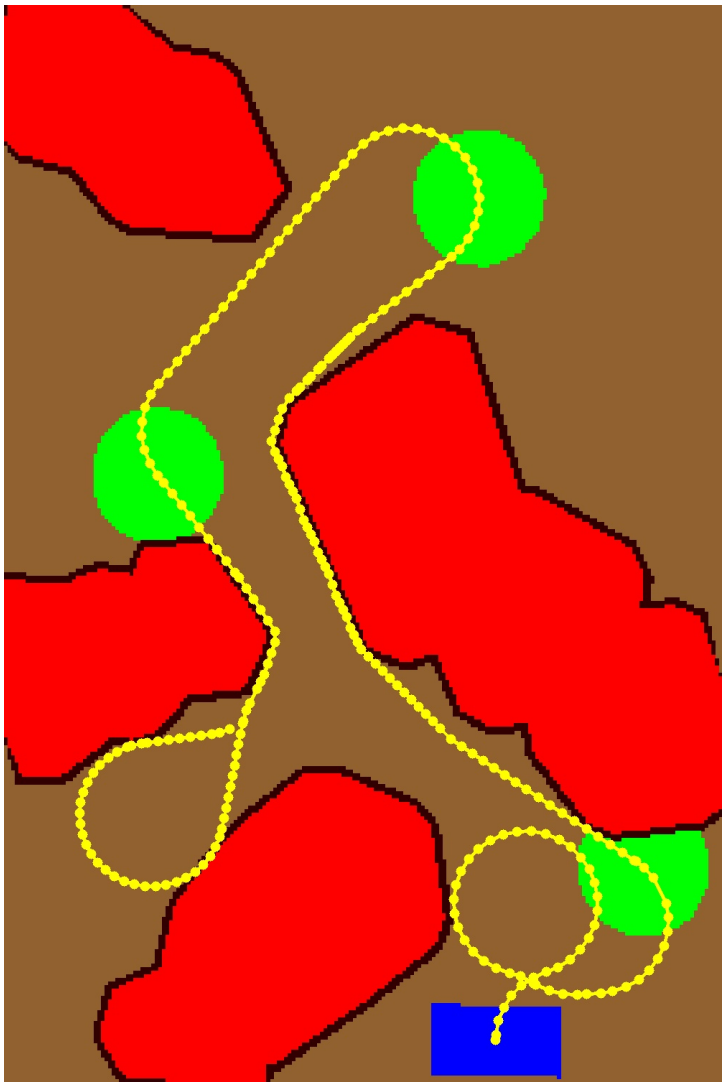


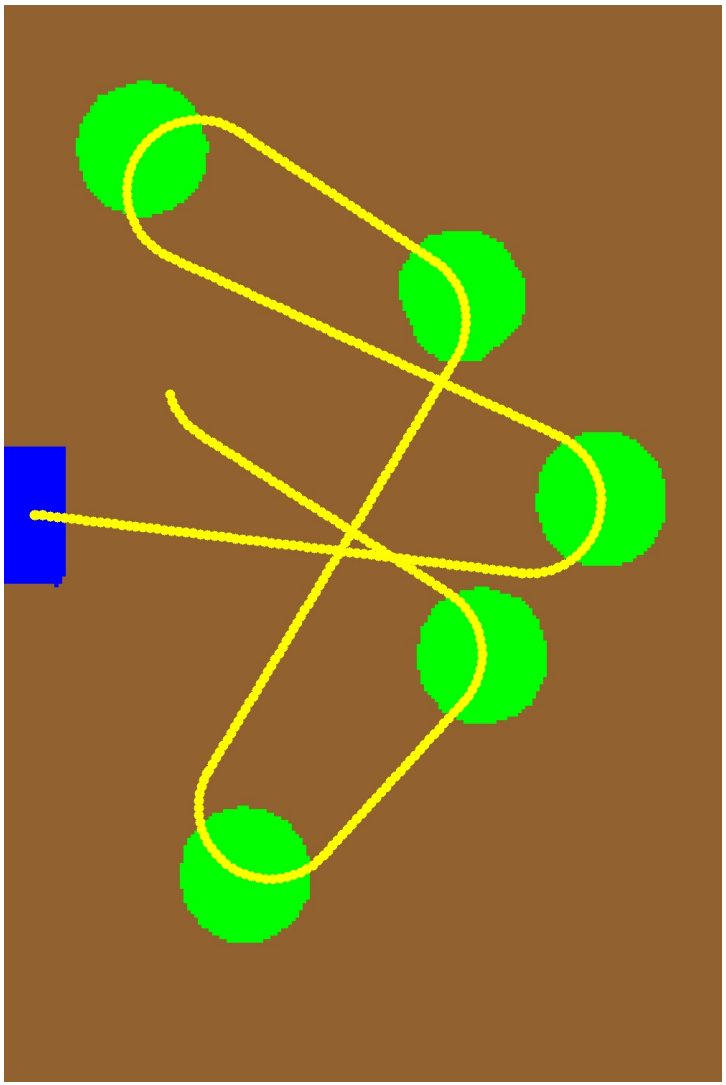
Dubins





Honorable mentions





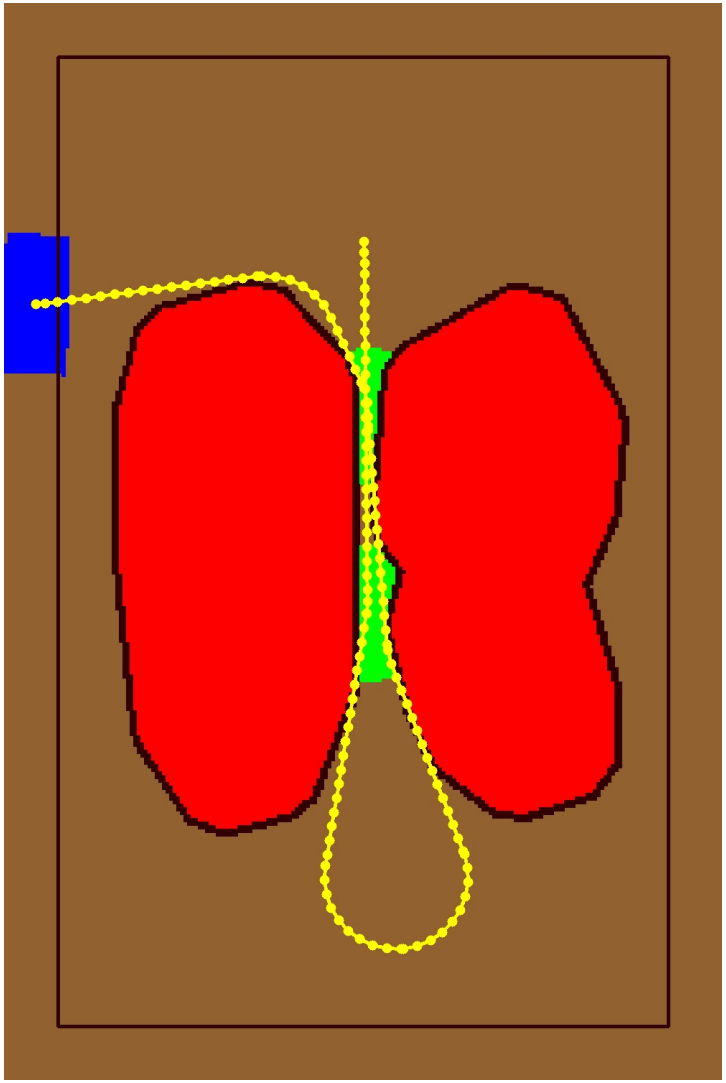
The bonus problem

The robot can save (or not) the victims in any order, getting a prize for each rescue.
To do this, the min path algorithm has been changed:

- It computes the distances between each couple of elements as a complete graph.
- It try all the possible combinations of routes considering the bonus.
- Finally choose the cheapest combination and apply Dubins.

Known problem

A known problem is what we called "The tunnel problem".
This problem is due to the fact that the min path algorithm does not consider directions and the sample algorithm of the Dubins cannot always find a solution.



Acknowledgments

Thank you for your attention.