Documentation

Bernhard Fritz, Mike Koch, Mario Zelger

Abstract

During the summer semester 2015 we implemented a Robot that was able to navigate its way through a workspace and collect colored balls. This documentation describes which approaches we used to solve the appointed tasks.

1 Introduction

Moving a robot around a workspace and letting it collect different colored balls is quite a complex task. It includes self-localization, obstacle avoidance, color detection and general strategies how to deal with occurring problems. In this documentation, we want to explains our version of odometry, i.e. how we made our robot move around the workspace including obstacle avoidance and self-localization. Furthermore we describe how our color detection works, which we needed to collect balls and self- localization using colored beacons. Additionally we portray our algorithm for caging a ball and our strategy for the tournament. We conclude our documentation with a brief work contribution breakdown and a final conclusion.

2 Technical Content

The robot provided had already a few commands at our disposal. We used an android phone to use this commands effectively. This had the advantage, that we already had a built in camera and didn't have to implement our own camera functions. Because our phone was running on android we were able to write the logic for our robot in java. The following sections explain in detail solutions for the parts mentioned in the introduction.

2.1 Odometry

For odometry we implemented a simplified approach as an alternative to the technique discussed in lecture. In general, robot movement can be distinguished between rotational and transnational movement. Our idea was that if you are aware of the robot's turn and movement speed, you have enough information to make it move wherever you want to. While turning/moving, the robot continuously updates its own estimated position and orientation.

Of course due to the fixed time intervals there will be some small error over time but even the best odometry has troubles dealing with these kinds of problems.

2.2 Generating motion commands to attain goal location

After the first two exercises we concluded that we can no longer use the default moving and turning robot commands ('k' and 'l') since they were very inaccurate. So we decided to use a combination of the following commands instead:

- 'w' for moving forward
- 's' for stopping
- 'i' to set a specific velocity (we used this to turn in place)
- 'q' for sensor measurements

These commands as well as Java's capability of letting a thread sleep for a specific amount of time, enabled us to implement more precise motion commands. We decided to use a design pattern called command pattern as seen in figure 1 to structure our code. Following commands have been implemented:

- Translation
- GoTo
- RelativeRotation
- AbsoluteRotation
- Measurement

2.2.1 Translation

The *Translation* command can be considered as a command for relative robot movement. As soon as the *Translation* command is called we send a 'w' character to the robot. This results in sudden forward movement of the robot. Given a distance in cm as parameter and the robot's velocity we measured when we got the robot, we are able to calculate the time we need to wait to send a 's' character to the robot to make it stop. While the robot is moving we constantly keep track of its x and y coordinates in world coordinate system.

2.2.2 RelativeRotation

As soon as the *RelativeRotation* command is called we send an 'i' character with specific parameters to make the robot rotate in place counter-clockwise. Given an angle in radiant as parameter and the robot's turn velocity we measured when we got the robot, we are able to calculate the time we need to wait to send a 's' character to the robot to make it stop. While the robot is turning we constantly keep track of its angle in world coordinate system.

2.2.3 Absolute Rotation

AbsoluteRotation is an extension of *RelativeRotation* and only uses some math to enable us to make the robot turn to a specific angle in world coordinate system.

2.2.4 GoTo

GoTo consists of two commands: *AbsoluteRotation* and *Translation*. Given two parameters x and y (world coordinates) and the robot coordinates, we are able to calculate the angle we need to turn the robot so that it is facing the goal as well as the distance to the goal using trigonometry.

2.2.5 Measurement

The *Measurement* command is used whenever we want to read sensor values. We discovered that only three of five sensors are actually working:

- front left sensor
- front middle sensor
- front right sensor

The *Measurement* command is exclusively invoked by the *SensorManager* which is responsible for parsing received sensor data, calculating an average estimate of sensor values as well as notifying observers about imminent obstacles.

2.3 Obstacle avoidance

To avoid obstacles we needed a way to keep track of sensors while moving. At first we used a Java thread to move and poll the sensors at the same time. This didn't work too well since there should always be some delay between two robot commands. That is why we decided to try a different approach. Since it is not necessary to poll the sensors all the time (e.g. turning in place), we only focused on polling the sensors while moving. We used a

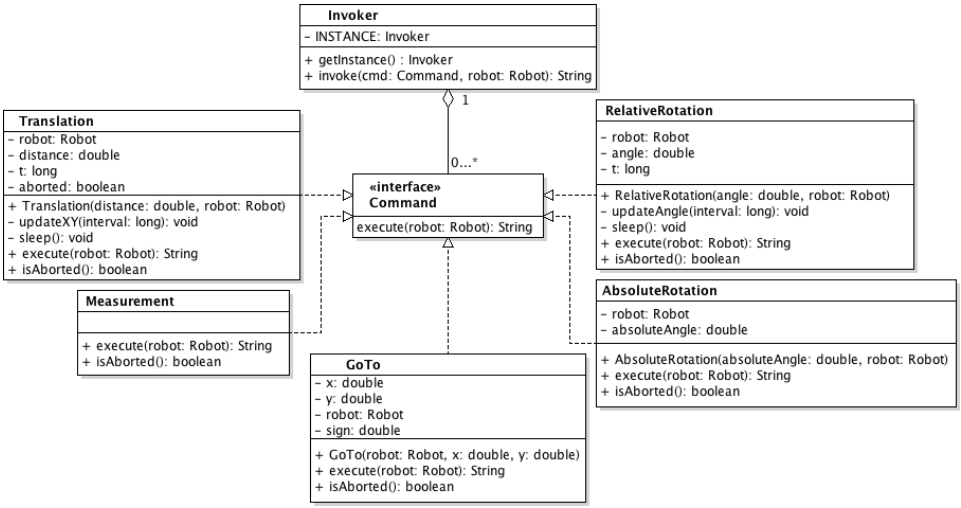

Figure 1: Command pattern

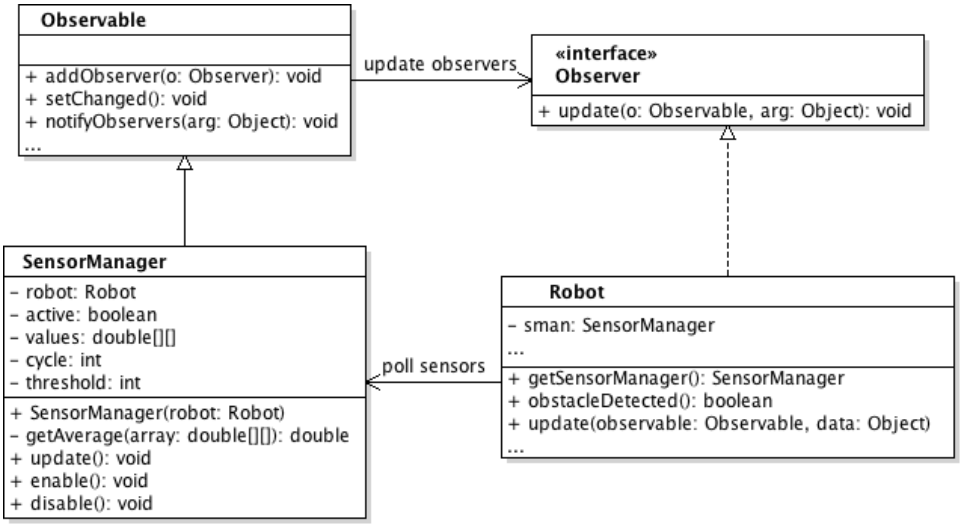

Figure 2: Observer pattern

design pattern called observer pattern as seen in figure 2 to conveniently let all observers know if an obstacle was detected or not.
If an obstacle was detected the current *Translation* command will be aborted and our obstacle avoidance algorithm will be started:

```
sign = 1;
while(Goal is not reached) {
  Turn towards goal;
  Move towards goal;
  if(Moving is aborted) {
    Turn sign*90°;
    Move 30 cm;
    if(Movement is aborted) {
      sign *= -1;
    }
  }
}
```

Essentially this algorithm is also known as "Bug 0" algorithm as seen in (Howie Choset, 2007, 7)

2.4 Color correction and beacon detection

For the color correction and in addition the beacon detection we used the *Color Blob Detection* example included in the OpenCV library (`http://opencv.org/downloads.html`). In the example code there is a class *ColorBlobDetector* which includes the contour calculation for a specific HSV color. We used that class for finding the contours of our colored balls and beacons.
The beacon contours we defined as two rectangles, one above the other, having a greater height than width.
After we filtered that contours out of our image the remaining contours must be the balls the robot can see from it's position.
The beacon detection had cost us a lot of time because we had many problems with the different colors in different lighting positions. This was our main problem implementing a correctly working beacon detection.

2.5 Self-localization

The self-localization is done like we heard in the lecture. We calculate a homography matrix from a given chessboard picture. With that matrix and the respective OpenCV functions we can easily calculate the distance from the robot to a beacon for example.
The self-localization (only position without the angle) is done with the following algorithm:

```
r1 = distance to beacon1;
r2 = distance to beacon2:
c1 = circle with radiance r1;
c2 = circle with radience r2;
p[2] = intersection points of c1 and c2;
pos = point out of p[2] which is inside the workspace;
```

This algorithm works since we know the boundaries of the robots workspace. The robots angle calculation was quite more difficult. In the end we did it similar to the algorithm we have learned in the lecture. We calculated the linear equation for the line between the two beacons. With that equation we get the intersection point of that line with the robots line of sight. Including one of the two beacons and the distance between the beacon and the robot we have a triangle. From that triangle we know all three sides so we can calculate the wanted angle with the law of cosine.
This calculation in combination with mostly inaccurate beacon detection caused also a lot of problems. In the end we fixed that problems and most of the time the self-localization worked quite fine.

2.6 Algorithm for caging a ball with and without obstacles

Our algorithm for caging a ball without obstacles looked like the following:

```
while (no more balls at the workspace) {
  Search for 2 or more beacons;
  while (not 2 or more beacons found)  {
    Explore the workspace;
    Search for 2 or more beacons;
  }
  Do self localization;
  Search for a ball;
  while (no ball found) {
    Explore the workspace;
    Search for a ball;
  }
  Drive to the ball with distance - 5cm;
  Catch the ball;
  Go to the pre-defined target position.
  Release the ball;
}
```

For the algorithm with obstacles we simply added the obstacle avoidance described in section 2.3.

2.7 Strategy for the tournament

Our strategy for the tournament was to catch all the balls in a short time without hitting the other robot. Since the different lighting conditions we had problems with the beacon detection and the self-localization. We did some nice tries with our robot but in the end it mostly detected the wall as an obstacle and did erratic movements.
The tournament still made a lot of fun and the big challenge with all robots driving at the same time in that small workspace was the highlight.

2.8 Miscellaneous

The following two divisions of our documentation show our user interface and list our work contribution breakdown.

2.8.1 User interface

We had two different user interfaces in use. A regular one for the robot and a second one to check our hsv-color values. Both interfaces were in landscape orientation since this gave us a broader field of sight.

The regular user interface had the camera full screen. A menu could be reached through a button on the top right of the screen. The menu contained a button to connect and one to disconnect furthermore it contained a button to calculate the homography and to start some action. The press on the action button started our routine, which started with self localization followed by catching a ball.

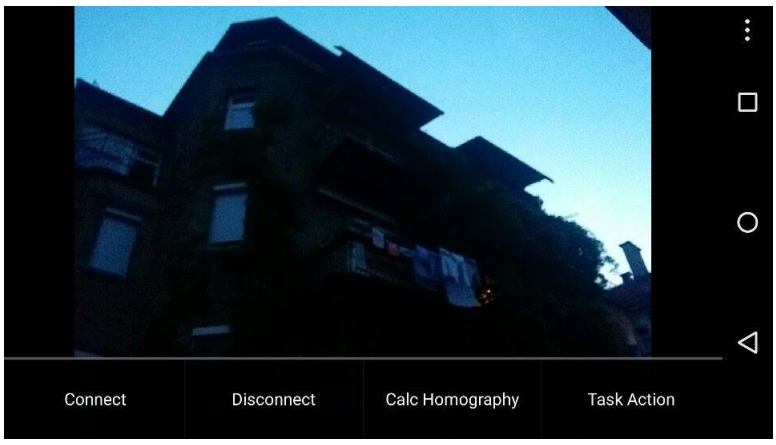

Figure 3: Robot user interface

The second user interface was made to check and read hsv-color values.

On the left of the screen was a button to start color detection and three sliders, each for a hsv dimension. On the right of the screen was our camera window. It displayed the current camera frame with additional information on top of it. The additional information consisted of the current hsv-color value and a contour of the found objects in that color. A tap in the camera area led to an update of the current hsv-color value.

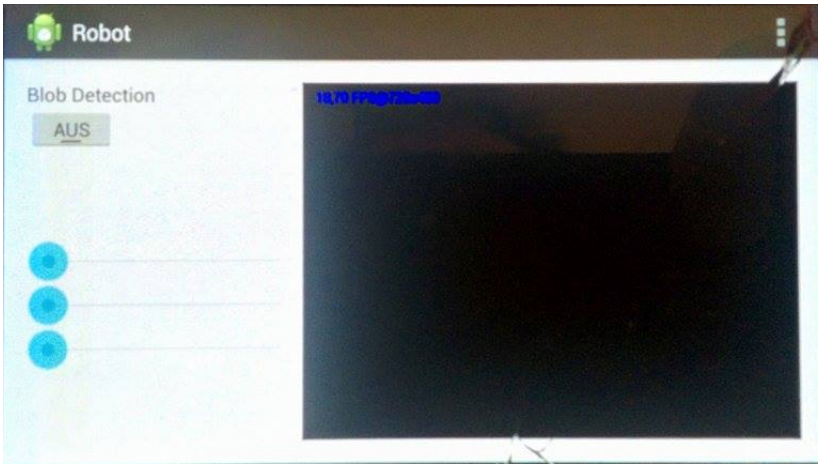

Figure 4: HSV detector

2.8.2 Work contribution breakdown

We divided our work like the following:

- Bernhard:
  Bernhard was in charge of odometry and obstacle avoidance.
- Mike:
  Mike was in charge of color detection.
- Mario:
  Mario was in charge of homography and self localization.

3 Conclusions

To conclude we have to say that we learned a lot. As the semester continued our project grew and grew. The amount of work we put into this project was immense, but in the end it led to quite successful solutions for the tasks given. Our biggest problem were the different lighting situations which led to unsuccessful detection of colors. In the future we would need to implement a solution that adapts or works around the different lighting situations.

References

Howie Choset. Robotic Motion Planning - Bug Algorithms. `http://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf`, 2007.