Building a Lego Robot that Finds and Collects Boxes with Specific Images

Mihail Dunaev
s1567045

December 10, 2015

Abstract

We built a lego robot that can navigate from one room to another one, in a maze that consists of 6 different rooms. The robot will scan the room for resources from all the corners, using a low resolution camera, trying to recognize the image specific to one type of resource. For object detection (resource recognition) we used the Viola-Jones[1] algorithm, already implemented in OpenCV[2]. Once the robot finds the resource he needs, it will grab it. To travel from one room to the next one we use a simple wall following algorithm. The algorithm needs one infrared sensor in front, one infrared sensors that can be placed on either sides, controlled by a servo-motor, two switches connected to two lego bricks in front (so when we hit something the switches get activated). We also needed two additional light sensors on the lego bricks, to detect collision with light objects. To grab the resource, the robot will just go towards the resource, and stop when the resource is inside the trap. To know when to stop, we added an additional light sensor in the front, inside the trap. The robot is able to identify the room he is in by detecting colored lego bricks with the camera, and matching patterns of them to a specific room.

All the hardware is connected to a fit-PC[3], where our program runs. We read the sensor values and control the motors with a Phidgets[4] board. We grab the images from the camera using OpenCV, and we also use the Phidgets library for an easier control of the hardware.

1. Introduction

The task is to build a robot capable of moving by itself in a map with multiple rooms, collect resources and bring them back to its base. There are 4 different types of resources and the robot is looking for only a specific type. He has to identify the correct one and pick it up, and then take it back to its base. A resource is basically a box with a picture on its sides so the robot has to recognize this picture. Different resources have different pictures. The rooms also contain lego bricks of different color which should help the robot identify the room it is in. To achieve this task, first we have to build the robot from hardware components, such as a power board, fit-PC, Phidgets board, motors, servo-motor, camera, and sensors (infrared, light sensors, switches etc) and lego parts (wheels, lego bricks, lego gears etc) and then we have to write to code that will make the robot achieve this task.

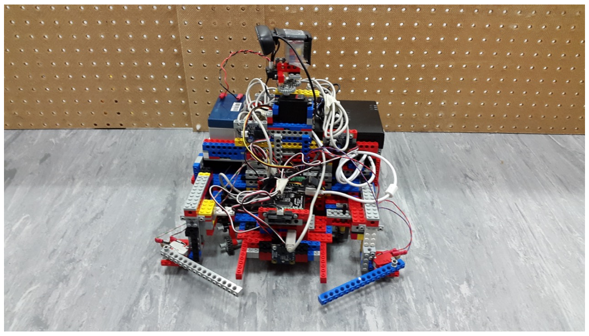
Our approach was to build a simple lego robot, on 4 wheels (2 base ones, and 2 casters). Most of the heavy hardware sits on the 2 big wheels. The camera sits at the top and it can be turned around using the servo-motor. We use it to identify the room we are in and the resource type. We use the infrared sensors, the two switches and the light sensors to follow the wall and navigate from one to room to the other. We also use them to avoid collisions. In our program we used libraries such as the OpenCV and the Phidgets libraries. We used OpenCV for computer vision algorithms and to interact with the camera (e.g. grabbing images from the camera) and the Phidgets library to interact with the hardware (command the motors, blink LEDs etc).

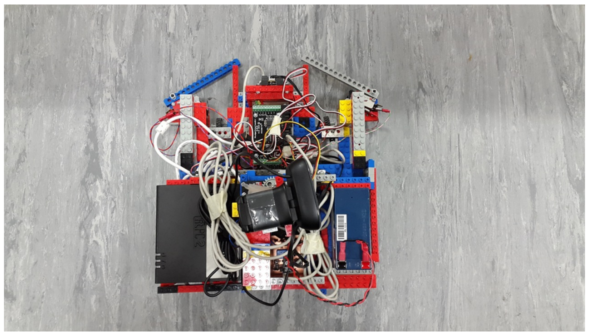
2. Methods

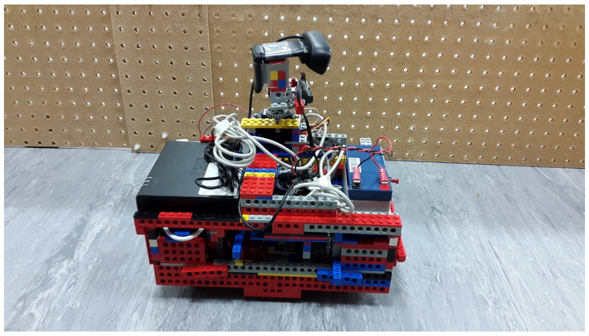
The methods we used to achieve this task can be split up into several parts. First we discuss the mechanical design and how to build the robot structure using legos (and what influenced our decisions in the building process). Then we talk about the program we used and the different modules that it has, which helps in achieving sub-tasks of the final task. In the end we explain how to combine all of the modules together to do the final task.

2.1 Mechanical Design

We started with the wheels and the base of the robot. We had to connect them to the motors using a geartrain. The one we used has a reduction of 1-to-16.197. We used a total of 4 wheels, 2 of them connected to the motors and 2 used as casters. The casters are small wheels in the front and the ones connected to the motors are big wheels, sitting at the back of our robot. We wanted to shift the whole weight towards the big wheels in the back. With this in mind, we built the base for our robot and placed the battery (heaviest hardware equipment) at the bottom of the base on one side, and the fit-pc (second heaviest equipment) next to it, on the other side. In between them we have the Phidgets board and the power board. At the top we placed the servo-motor with the camera attached to it. We did this so we can do a 360 degree scan with our camera, without any other component getting in the way. After multiple tests (which we will discuss later, at the wall following part) we also placed one infrared sensor under the camera, attached to the servo-motor as well (so we have a spinning infrared sensor). At the front we needed something for collision detection. We used the other infrared sensor, since they both are very reliable. Since our robot turned out to be very large, we needed some additional sensors at the front, especially on the sides (front-left, front-right). What we did was to use the switches connected to long lego parts, and every time the lego part hits something we would get a reading of a true value. Sometimes, depending on the speed we have, the switches won't get activated if we hit something light (e.g. a resource box). As an improvement, we added light sensors to the lego parts, so when we hit a resource box, there won't be too much light picked up by the sensor, so we know we hit something. Finally, we braced our lego structure as much as possible to make sure it won't fall off when the robot moves. The spot we used to grab resource boxes is the one at the front, between the two lego parts connected to the switches, right under the front infrared sensor. Once a resource is in this spot, the only way to lose it is if we start moving backwards (which we never do). Full pictures of our robot can be seen in Figure 2.1 a)-d).


(a) Front view of the robot.


(b) Top view of the robot.


(c) Back view of the robot.

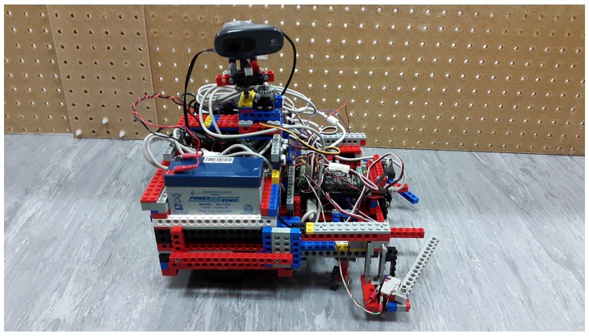

(d) Side view of the robot.

Figure 2.1: The robot from different angles. It has two lego bricks in front attached to switches, a trap with a light sensor at the back and an infrared sensor above it. On the left side we placed the fit-PC and on the right side the battery to balance the robot. They are both at the back, above the 2 big wheels. We braced it several times to make sure it won't fall apart. The power board is between the fit-PC and battery. The right caster wheel can be seen slightly behind the switch mechanism. The top infrared (used for side wall scanning) and the camera are attached to the servo-motor.

2.2 Code

Before starting to work on any sub-task, we wanted to check all the hardware equipment and do some initializations. To check the hardware equipment, we just checked all the sensors (by displaying their values to a computer screen while we test them), the servo-motor and the camera. For the motors, we wanted to make sure the robot can go straight. We first tried both motors at 100% power (in fact, our motors are symmetric with respect to the center of the robot, so one had to go 100% and the other one -100%). This apparently was not good, since our robot had a small deviation to the right. There are several possible reasons for this. First, the motors are not exactly the same (so one can spin faster than the other, even if both are at 100%). Another reason is that even if we tried to balance everything, we couldn't do it in a perfect way. If the weight is not equally distributed on both wheels, then they have different frictions and thus they spin at different speeds. One last possible reason is that they are not perfectly straight. With all this in mind (it's close to impossible to fix all these problems and have perfection), we tried to reduce the power on the left motor (since the deviation was on the right) until our robot moved (almost) straight. We obtained a value of 85% power for the left motor, for which the robot moves straight.

The next thing we did was to look into the turning mechanism. Our idea for turning was to make one wheel go straight and the other one backwards. We also wanted some flexibility, so we decided to make a function *turn(direction, angle)* that will allow us to turn left or right by any angle that we want. If we start the turning motion, do nothing for some time t, and then stop, we will turn by some angle θ. We should be able to map all of the θ values to time values. Our intuition was that if we wait t seconds to turn 90 degrees, we have to wait twice as much for 180 degrees (so time depends linearly on the angle). We measured the time it takes to do 90 degrees (we got a value of $t_{90} = 1.4s$) and then we just made the robot wait ($t_{90} \times \theta/90$) for any other angle θ. It turned out that our intuition was correct and we got really small deviations for angles between 0 - 180 degrees (less than 5 degrees errors)[1].

One last thing we did before we actually started working on the task was to make a simple, *debugging* version for toddler (toddler is our main program for the robot). In this version, one can control the robot using the keyboard and read the values from the sensors and camera in real-time (our basic control was w, a, s, d to move straight/backwards for 500ms, turn left/right for 5 degrees; we also included servo control with u, o for turning and t for taking a picture with the camera). We used this version to get a feel of the sensor values when the robot is inside the maze (we will refer to all the rooms in the map as the *maze* from now on) and take pictures that should help us with the task at hand. One of the things we noticed with our debugging toddler was that the only reliable distance sensors were the two infrared ones. The whiskers were returning true too late (they almost snapped), and the sonar and light sensors were not too useful for big distances.

2.2.1 Localisation

The localisation problem is simple: we don't know the room we start in and we have to figure that out just using the hardware. Every room has lego bricks with colors specific to that room, so we had to detect the colors of the lego brick inside a room, and map the colors we found to a specific room. Our first idea was that instead of trying to detect the color of the object (lego bricks), we should try to detect the object itself (so look at the object shape) with some object detection algorithm (for example Viola-Jones which is already implemented in OpenCV). However, we found out afterwards that the shape of a lego brick can change, so we switched back to lego brick color detection. What we did was to take pictures of the lego bricks with our toddler in debugging mode, and analyze the HSV (hue, saturation, value) values for the lego bricks inside each picture (and room). Examples of such pictures can be seen in Figure 2.3. We took a total of 105 pictures out of which 71 are background images (with no lego bricks) and around 5 pictures for each lego brick that has a unique color. We used HSV because it does a better job at separating colors, as can be seen in Figure 5. Our idea was that if an image had a number of pixels (above a threshold) with similar HSV values to one of the colored lego bricks, then that picture also has the lego brick (we count the number of pixels with colors similar to those of lego bricks; by similar we mean they

[1]As a side note, instead of measuring for 90 degrees, we could have done it for 180, since a small error for 180 implies a small error for 90 degrees, but this would have taken too much time, so we just did it for 90.

are between certain limits of each other). As we can see from Figure 5, for most colors the separation is based on the hue values. The analysis we did was to look at histograms of hue values for each lego brick (Figure 2.4), and find out this way the hue interval for each color. The limits that we used can be seen in Table 2.1 (the saturation values is always between 150 - 255 and the value is between 50-255). We decided on the number of pixels (thresholds) empirically (by looking at pictures). The values we found are in Table 2.1. So, if our robot takes an image, converts it from BGR (OpenCV default) to HSV, and finds out there are more than 2500 pixels in that image, with a hue value between 67 - 76, saturation value 150 - 255, and value 50-255 (green), then that image must contain the green lego brick (we can assume this since our maze is not that diverse, the walls are brown and the floor is a light blue).

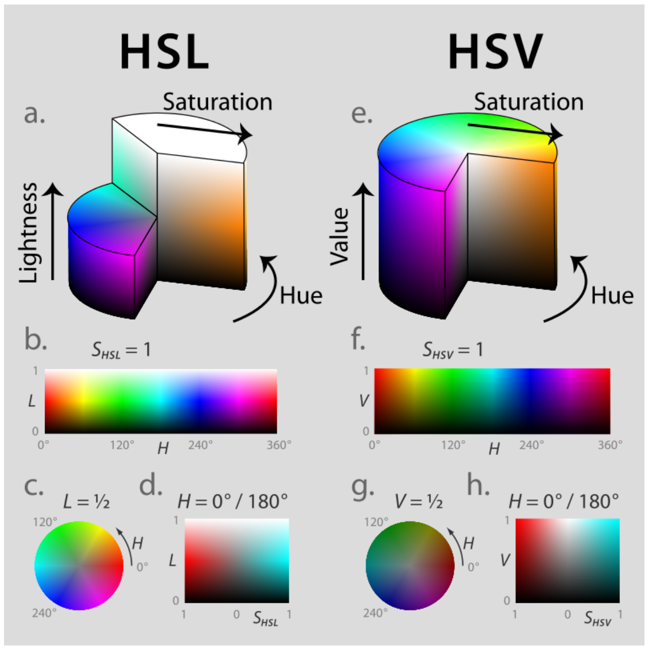


Figure 2.2: Representations of the HSL and HSV color spaces. Picture taken from wikipedia.

For white and black we used HSL (hue, saturation, lightness) values of (0,100,100) and (0,0,0) (the threshold is 2000 for both of them).

When our robot tries to find out what room it is in, it will do a 360 camera scan (in increments of 5 degrees). At each step it will look for all the colored lego bricks (just like in our previous example with the green lego piece). Based on what it finds, it can tell its current room. The final algorithm that we used is in Figure 2.5 (pseudo-code). The idea behind it is that there are colors you can find only in some rooms (like white - C, red - E, orange - B) and combinations of colors you can only find in other rooms etc. If we finished the 360 scan and we only found one green or blue lego brick, then we are in room F or A.

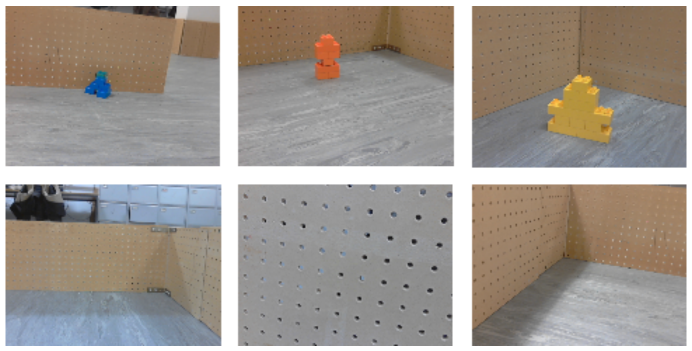


Figure 2.3: Pictures with lego bricks on the first row, and without any lego bricks (background) on the second one.

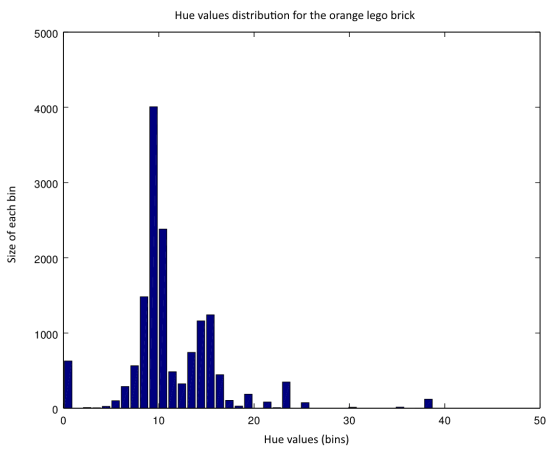
Color	Blue	Green	Orange	Red	Yellow
Hue values	103 - 107	65 - 76	8 - 10	2 - 5	19 - 24
Threshold	2000	2500	2000	2000	2000

Table 2.1: Hue limits and pixel thresholds for lego bricks.

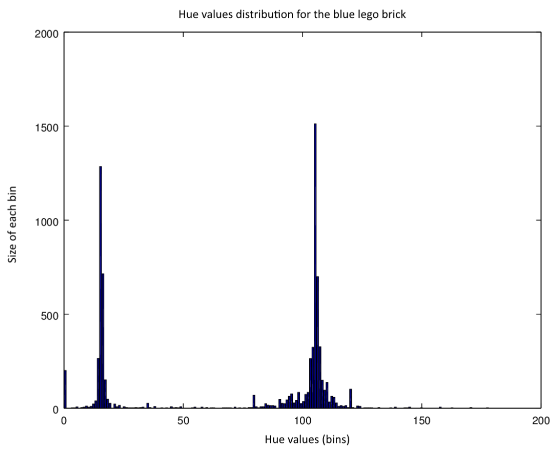
2.2.2 Navigation

For navigation we had several ideas. Our first idea was to use a matrix. To implement this, we had to find the position of our robot, and then divide the whole map into a matrix with squares of the same size as the robot. The way we build this matrix is that we start from our robot and put it in a square. After this, we expand the matrix in all directions until all the map is covered by squares (the edge squares might go outside the map, but this is not a problem). Since we know the position of each wall, we mark all the squares that contain a wall segment as impassable (all the edge squares should be impassable by now). After this, when we want to go from point A to point B on the map, we just compute the matrix position for point B, and then use BFS (breadth-first search) to compute the quickest route from A to B. Every time we detect an obstacle in a square that is passable, we mark it as impassable and we go outside the square (in an adjacent one). Our second idea was to try to discretize the whole map. By this, we mean that we should have some roads and nodes between which our robot can travel. If our robot is outside these roads/nodes, we just have to compute how to get back on the road. If we do the same for the destination point, in the end we can come up with a route from start to finish (going through our nodes). We later found out that this is actually a roadmap approach to the problem. The last idea was to just try and follow the wall from one room to the next one, until we reach our destination.

Since we didn't have much time to implement and test ideas, we decided to go with the simplest one, and that is a wall following approach. While not optimal, this is a simple to implement method. In this approach we first try to get near the wall. For this we look at the infrared sensor values in front, left and right. We pick the biggest value since we don't want to spend too much time reaching the wall (also, lowest value might be from a different



(a) Histogram for the hue values of the orange lego brick. Orange is between 8 - 10 and the brown wall is around 15.



(b) Histogram for the hue values of the blue lego brick. We notice that we have two main colors in this image, one is blue (103 - 107) and the other one is the brown wall (around 15).

Figure 2.4: Histograms with hue values for blue and orange lego bricks.

room, while highest one is guaranteed to be in the same room, since we must have a wall from the current room either in front, left or right), and then we go in that direction. When we reach the wall (we use the front sensor for that), we know that if we go to the left and follow the wall in that direction, at some point we will reach our destination room. The same is true if we follow the wall in the right direction. So we just have to compute which way is the shortest towards our destination room and go that way. To see why this works, we have to think that our maze is like a circle (Figure 2.6). If our current location is A we can see that the shortest way to D is if we go left, but to get to E the shortest way is to go right. Once we compute the direction that will give us the shortest way, we go in that direction and every time we detect a new room, we update the current one (we use two variables for this algorithm, one for the current room and one for the destination room; the initial value for the current room is the one we detected with our color detection algorithm). If the new room is the destination room, we stop. The way we detect a new room is explained in the next section.

After we detected the room we are in, and reached the closest wall, we also have to align to it (so we have to get from a position where we are facing the wall to a position where the wall is sideways). To do this, we set the sensor on the side where our wall should be, using the servo-motor. After this, we turn the robot until we find a maximum value for the sensor. We also check that the sensor value is above some threshold, so we don't align to some distant wall instead.

Wall Following

Once we are close to the wall and decide to follow it in one direction, there are multiple things that can still happen. We identified 4 such things. Because our robot is not going

```
360 degree loop
        if we found a white piece
                room C
        if we found a red piece
                room E
        if we found an orange piece
                room B
        if we found a black piece
                if we also found yellow
                        room D
        if we found a yellow piece
                if we also found black
                        room D
                if we also found green
                        room B
                if we also found blue
                        room D
        if we found a green piece
                if we also found yellow
                        room B
end
if we found a yellow piece
        room B
if we found a black piece
        room E
if we found a green piece
        room F
if we found a blue piece
        room A
```

Figure 2.5: Pseudocode to determine our current position (room).

straight and the maze walls are also not straight lines, at times we can get too close to the wall(1) or too far away(2). Also, at some point the wall might *"disappear"* from our side(3) (in case of an outside turn; we call this a *void* on the side). So we need a way to see what's happening with the wall next to us. For this we need a sensor that is reliable and can also work with decent distances (so it can make a difference between going too far away from the wall and a void). The only sensor we had for such a task is the infrared sensor. The problem here is that we can have walls both on our left and right side (depending on the direction we are following the wall), so we need an infrared sensor on left and right (and in front as well, for the collision detection). To overcome this we connected one of the infrared sensors to the servo-motor, along with the camera. This allows us to turn the infrared to either left or right, depending on the side the wall is on. The last(4) thing that can happen is to have something in front of us. In this case we just turn 90 degrees (the direction depends

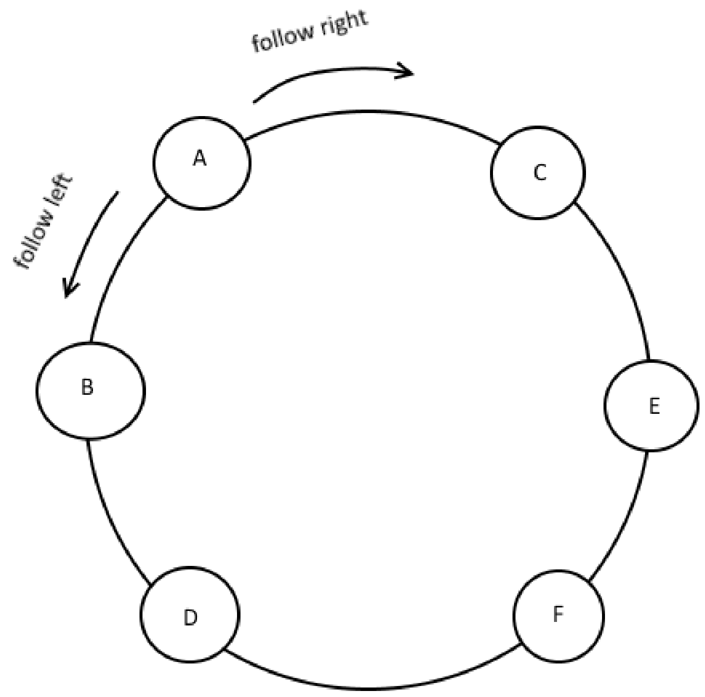
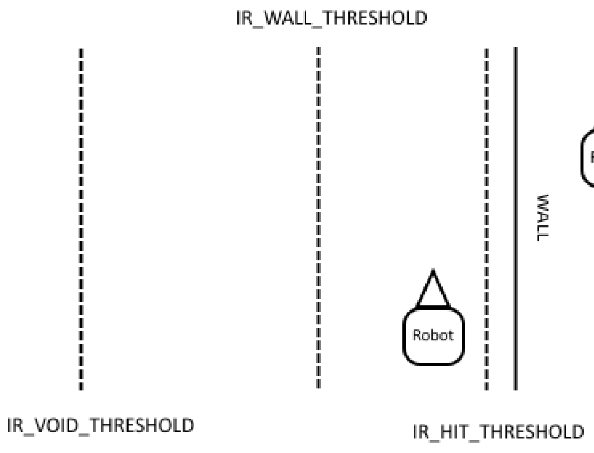


Figure 2.6: Using a wall following approach will allow us to get from any room to any other (either following the wall on the left or right).

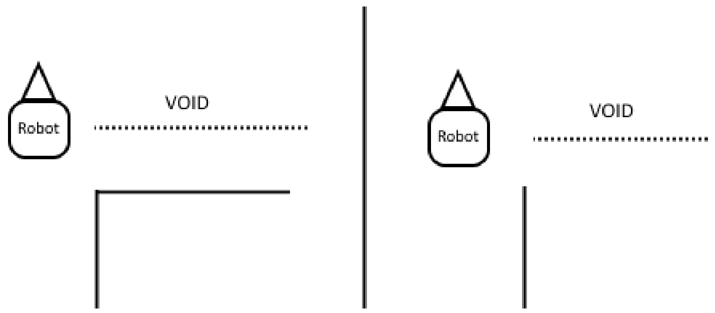
on which side we follow the wall) and we keep following the wall.

For cases (1) and (2), we used 2 different threshold values for the infrared sensor: one if we are too close (we called it IR_HIT_THRESHOLD) and one when we are too far away (IR_WALL_THRESHOLD). We determined both values experimentally: 300 for the hit threshold and 170 for the wall one. So if we get too close, we just turn away by 5 degrees (this can happen multiple times in a loop) and if we get too far way, we turn towards the wall by 5 degrees.

Case (3) is a special one. Here we have an outside turn and we can either stay in the same room, or reach a new one. First, we detect case (3) when the sensor value is too small (smaller than IR_VOID_THRESHOLD, 100). Next, we turn towards the void and then we move forward until we feel something on the side (note, there are also special cases here when we won't feel anything on the side: if we have a small lego piece there, we won't be able to detect it so we have to take that into account as well; also, in some cases we won't be able to detect something on the side because of our alignment towards the wall, so we introduced a maximum waiting time after which we always turn). This can be the start of a new wall we have to follow (so we go back at following the wall) or it can be a thin fragment of wall - this is when we reach a new room. So if our sensor detects a wall and then drops back to some low value, we are in a new room. We used this to update our current room.



(a) The robot must be between two threshold values of the infrared sensor when following a wall (between HIT and WALL).



(b) Two possible cases when we detect a void on the wall side: in the first one we are still in the same room, so we have to turn and keep following the wall, and in the second one we know we will change room.

Figure 2.7: Threshold limits for the infrared sensor and possible cases when we detect a *void*.

Collision Avoidance

The collision avoidance is already implemented in our wall following algorithm. Every time we move straight we check for collision in front and towards the wall. There is nothing else we need to do to check for collisions.

2.2.3 Resource Identification

For resource identification we tried to use Viola-Jones, which we also tried previously for the color detection. OpenCV stores the paramaters for the Viola-Jones in an XML file. So first we had to create such a file (this is the training part, which uses boosting). To do this, we need a dataset of images that contain the object we are interested in detecting. We also need to mark the position of the object in each image. This is usually stored in a VEC file. OpenCV helps us create VEC files as well (using the *opencv_createsamples* tool). Examples of how we can do this in OpenCV is in Figure 2.8. What we did was to take the pictures with Mario, Wario, Fry and Zoidberg and use them as object images. For background images we used our previous pictures with nothing in them, as well as the ones with colored lego bricks. We then just used the *opencv_createsamples* tool that applies a transformation to our object image and then adds it to a background image. It also saves the position of the object image in the new generated image. We generated 1500 such images and stored them in a VEC file. Next, we just had to use this VEC file as our training data for the Viola-Jones algorithm. For the training part we used another OpenCV tool, *opencv_traincascade*. An example of how to use *opencv_traincascade* is in Figure 2.8.

The training step usually takes a long amount of time, so we couldn't experiment too much with the data. The arguments we picked for the *opencv_createsamples* and *opencv_traincascade* are purely experimental and influenced by Naotoshi Seo's tutorial on object detection[5]. Once we have the XML file, we can load it in a python script and use it to detect objects

Resource	Mario	Wario	Fry	Zoidberg
Precision	0.72	0.86	1	0.9
Min Neighbors	43	40	11	12

Table 2.2: Precision for resource detection on a small test set.

in any image. This is done by calling the function *detectMultiScale* of a cascade classifier object. One parameter of the function that we experimented with is the minimum number of neighbors. A higher value for this parameter means a stricter object detection (the image patch must have more neighbors from the training set, of class c to be considered part of c). To test how good our detection is for an XML file, we took 10 pictures for each resource type (mario, wario etc) with our robot in debugging mode, and then tried to use the generated XML file to detect the resource type in any of them (locally, on a PC). We also changed the values of the minimum number of neighbors so we can get better results. We included our previous images (with nothing and colored lego bricks) in the test scenario to look for false positives as well. The final results are in Table 2.2. The metric we used is the precision (number of true positives / (true positives + false negatives)). A good result is that we didn't get any false positives. An example of object detection can be seen in Figure 2.9.

```
# creating a vec file from the mario image
opencv_createsamples -img mario.png -bg ng.txt -vec mario.vec
-num 1500 maxxangle 0.6 -maxyangle 0.6 -maxzangle 0.3 -maxidev
40 -bgcolor 0 -bgthresh 0 -w 90 -h 90

# training the calssifier (create the xml file) with the
# generated vec file
opencv_traincascade -data mario_cascade_dir -vec mario.vec -bg
ng.txt -numStages 5 -minHitRate 0.999 -maxFalseAlarmRate 0.5
-numPos 100 -numNeg 70 -w 90 -h 90 -mode ALL -precalcValBufSize
512 -precalcIdxBufSize 512
```

Figure 2.8: Generating train samples and training a cascade calssifier (Viola-Jones) with opencv tools.

2.2.4 Resource Grabbing

Once we detect the desired resource in the image, we have to grab it. To do this we want to position the resource in the center of the image (on the x-axis). When the object is in the center of the image, since we know the servo-motor angle (relative to the robot's angle), we just need to turn the robot such that it has the same angle as the servo-motor. At this point the object should be in front of us, so we just have to go straight and grab it (this is because of how we built our trap). Since we have some deviation when going straight, we tried to implement two different methods to achieve this task. In the first method, after we have the

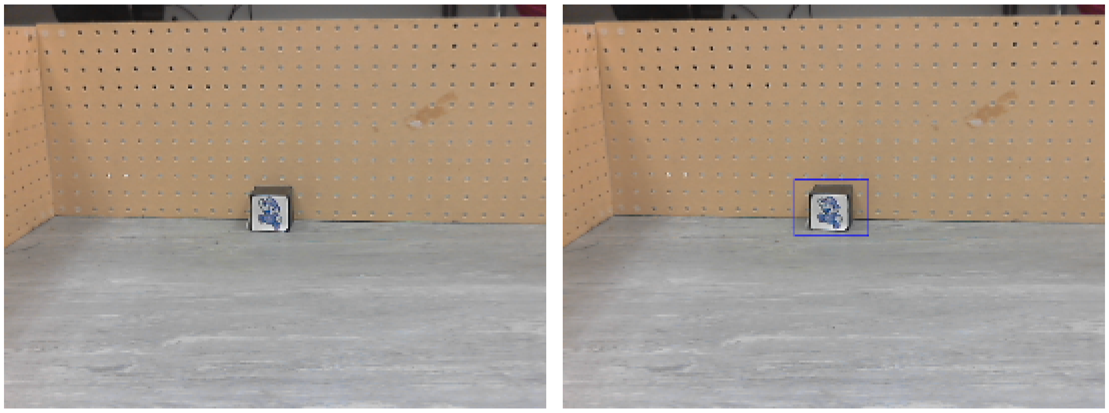


Figure 2.9: A resource box with Mario image detected by the robot, using the Viola-Jones algorithm from OpenCV.

object in the center of the camera, we just go straight and at some point we will hit it. In some cases the resource will end up in the trap. However, in other cases the resource will end up hitting the lego brick connected to the switch. When we tested, since the resource box is too light, it didn't trigger the switch (this is actually the first time when we noticed that we can hit something without getting any sensor value). To overcome this, we added two light sensors in front of the lego bricks. After doing this, we were able to detect the resource box every time. If the resource box ends up hitting one of the two front lego bricks, we just have to turn by some degree (we found out that 30 degrees is enough for all cases) and then move straight until the resource is in the trap. To check that the resource is trapped, we have a light sensor inside the trap, so when the resource touches the sensor and blocks the light, we will get a reading of a low value from that sensor, so we know we trapped it.

In the second method, we did everything we did in the first method, but instead of just going straight until the robot reaches the resource, we go straight for some time, stop and perform the scan again, just in case we deviated too much from our initial trajectory (we align again such that the object is in front of us). The problem with this approach was that first, once we got too close to the object, we were not able to detect it again, so we have to use our previous method again, and second, when we noticed that it takes too much time to reach the resource (performing the scan again at each step) and we want to be fast as well (because of battery concerns).

Once we detected to object, we had to bring it in the camera's center. To define this camera center region, we just look at the x coordinates, and say that the object is in the center if its x value is between $[x_{midcam} - \epsilon, x_{midcam} + \epsilon]$. Since we used medium resolution for the camera, x_{midcam} is 320. For ϵ we used 10 pixels. To get the object in this region we turn the camera left or right by some delta angle (we started with 20), in a loop. If we turn one direction and then notice that we skipped the object (now it's on the other side of the center interval), we just lower the delta angle (first by half, and then by minus 1, until we get to minimum 3 degrees), and try again. After some iterations the object will end up in the middle.

One more thing to add to all this is the final algorithm that we use when we search a room for resource objects. The first thing we do is a 360 degree scan with our camera from our initial position. If we can't find the object, then we know that it must be in the same room, hidden behind some other object (that, or our algorithm didn't detect it). Our robot will then try to get to the closest wall, and follow it. Every time it reaches a corner it will perform a new camera scan. This approach should be enough to detect the resource, if it is in the current room. If the robot detects a new room, it will stop and turn around (by 180 degrees). We also save how many corners we scanned from, before we reached the new room. Next, the robot goes back into the room, skips the first n corners that we already scanned and continue scanning the next ones until it reaches the room on the other side. If it didn't find anything, then it will print that the resource is not in the room. If the robot detects the resource while he is in some corner, it will try to place it in the camera center region and then grab it.

2.2.5 Homing with the Resource (Theory)

We didn't get to implement the homing with the resource part. However, we will describe what our idea was on how to do this. Once we have the resource trapped, everything we need to do is set the destination room as our base room (where our black patch is), get close to a wall, and follow it using the existing algorithms. The only way we can lose the resource is if we go backwards, but since in our wall following algorithm we just move forward and turn around, we will never lose the resource.

2.2.6 Putting it All Together (Theory)

To achieve the final task, which is a robot capable of going in all the rooms and look for resources to bring back to its base, we have to put everything we did so far together. We will start by identifying the room we are in (and save it in a *base_room* variable). Next, we mark all the rooms as not visited. We will take each of these rooms, make it as our destination room, and follow the wall until we get there. Once we are there, we scan the room for the resource we need. If we find a resource, we grab it, set the destination room as our base room and get back home. Once we are in our base room, we have to drop the resource (go backwards) and go back to the room we were previously scanning. We can save the steps we did or just scan the whole room from the beginning. When we can't find any resource of the type we need in the room, we mark it as visited and we go to the next non visited room. We stop when we visited all the rooms.

3. Results

First, the mechanical design we chose proved to be efficient. We didn't have falling lego pieces. The robot was able to move at a decent speed. The only few problems were not being able to go perfectly straight, small errors in turning. However, we overcame these problems in our code. For the localisation part, we placed the robot in all the rooms, at different positions. It was able to identify the room it is in every time.

We had some problems with the wall following algorithm. The robot was able to follow the wall from room to room most of the time. We placed it in all the rooms and first try to get to adjacent ones. We then did the same but tried to go across multiple rooms. In both cases, one problem that we had was with outside turns (as seen in Figure 3.1). This was due to small errors in turning and going straight (and angle adjusting as well). Furthermore, this caused the robot to be trapped in some code where it didn't check for collisions, so it eventually crashed into some wall. We came up with some solutions for this problem but we never had the time to test any of them.

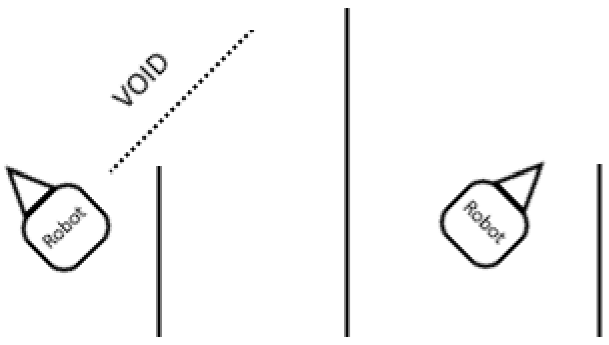


Figure 3.1: An unfavorable case for outside turn. The robot will crash into the wall.

For the resource identification part, even if we got a really good precision on the test pictures that we took with our robot from the maze, when we tested the final program on the robot, we noticed that we have worse results. We were getting false positives for Wario in Zoidberg images, and also false positives for Zoidberg in images with no resource boxes. We increased the threshold for Wario (from 40 to 41) and for Zoidberg (from 12 to 14). We were still getting false positives but the results were better. To test for resource detection we placed the robot with multiple resource in the same room. We then let the robot do a 360 scan and see what resource it can detect. The best results were for Fry (which was detected all the time). Our second best result was Zoidberg, followed by Wario and Mario, which sometimes were not detected. We placed the resource boxes at various distances. The maximum distance from which it can detect a resource box is roughly 1.2m. Given that our robot could identify the resource, it had no problems grabbing it. We tested this with Fry, in room E, from various distances. Our robot was always grabbing the resource.

4. Discussion

Our robot is capable of doing most of the sub-tasks from the big final task. It can correctly identify the room it is in, it can go from one room to another and only occasionally crashes. It can also identify resources with a decent precision. The best detection we got is for Fry and Zoidberg resources and the worst one for Mario and Wario. After it recognizes the resource, it can also grab it. We didn't implement the sub-task in which the robot takes the resource back to its base and the final combination of all sub-tasks. One strength of our robot is that it can detect colored objects and resource objects at a fast speed. Another feature that helped us a lot when coding is that we can take images or use the infrared sensor in any direction (since we attached it to the servo-motor).

One of the most demanding part of the whole process was testing the code. While building the physical robot and the coding part didn't take that long, the testing was tedious, also due to the fact that the battery dies so fast so we had to wait for it to recharge. As a consequence, we have chunks of code that we didn't get to test and fix properly. Another problem with testing was that some errors were not reproductible (because of small variations every time).

There are some obvious improvements we can make to our robot. First, instead of using a wall following algorithm, we can try something more complicated, like the matrix approach. This will allow our robot to reach the destination point faster. Another thing that we can improve is the object detection algorithm. From the testing we noticed we had false positives and also no detection in some cases. Instead of using the *opencv_createsamples* tool, we could try to make the train dataset ourselves.

One of the things we learned from all of this is that even if our algorithm works on paper (or a simulated environment), because of errors from sensors and deviations when moving straight or turning, it might not work in practice. A good way to overcome this is to have a base case scenario, that our robot should reach when something happens that was not supposed to. In that base case scenario we should start scanning the environment again, and try to understand where we are and what we have to do to achieve the task.

Bibliography

[1] P. Viola, M. Jones. *Rapid Object Detection Using a Boosted Cascade of Simple Features.* Conference on Computer Vision and Pattern Recognition, 2001.

[2] OpenCV Website,
`http://opencv.org/`

[3] Fit-PC Website,
`http://www.fit-pc.com/web/`

[4] Phidgets Website,
`http://www.phidgets.com/`

[5] Naotoshi Seo's tutorial on object detection,
`http://note.sonots.com/SciSoftware/haartraining.html`