VILNIUS UNIVERSITY

FACULTY OF MATHEMATICS AND INFORMATICS

DEPARTMENT OF COMPUTER SCIENCE

DRAWING ROBOT "BOT'Vinčis"

Project work

Group members:

Artūras Chramčenko

Liudas Indrašius

Žygimantas Ulevičius

Domantas Varapnickas

Supervisor:

Agnė Brilingaitė

Vilnius, 2014

Contents

Contents

Terminology

leJOS - a firmware replacement for Lego Mindstorms programmable bricks, which includes a Java virtual machine.

I/O operations - input/output operations.

GUI - graphical user interface.

.nxj application - application written in java language for NXT microcontroller, using specific libraries.

Servo motor - is a piece of hardware which consists of a suitable motor coupled to a sensor for position feedback. It allows precisely control angular position, velocity and acceleration.

Summary

Process automation is a prominent research topic spanning across different disciplines and with strong expectations. This project is focused on a task of drawing shapes and contours from images accurately on a large scale canvas. The methods of this project can be separated into two major groups: algorithms to get picture contours out from an image and sort them (this group includes Canny edge detection algorithm) and software to communicate between LEGO NXT robot and computer, as well as to correctly convert picture into commands for robot to execute (this group involves working with leJOS library). The result of the project is ambiguous - although the minimal goal was achieved, robot's lack of precision appeared unfulfilling. However, additional features allowing to control robot with keyboard or to upload image drawn inside the program can be used to achieve satisfactory results.

Santrauka

Piešiantis robotas "BOT'Vinčis"

Procesų automatizavimas - aktuali ir gerai žinoma tema įvairių sričių specialistas, vis dar kelianti didelių lūkesčių. Šio projekto tikslas į tikslų skaitmieninių (kontūrinių) paveiksliukų, schemų perpiešimą and didelio ploto paviršiaus. Metodai ir priemonės naudoti tikslui pasiekti gali būti suskirstyti į dvi pagrindines grupes: metodus, skirtus iš paveiksliuko išskirti ir tiksliai surūšiuoti kontūrus (angl. Canny edge detection algoritmas priklauso šiai grupei) ir metodus, užtikrinančius komunikaciją tarp kompiuterio ir LEGO NXT roboto, o taip pat ir tikslų kontūrų pavertimą komandomis, kurias robotas galėtų įvykdyti (šiai grupei priklauso darbas su leJOS biblioteka). Projekto rezultatas dviprasmiškas - nors minimalus tikslas ir buvo pasiektas, tačiau roboto tikslumo stoka neatitiko lūkesčių. Tačiau į projektą buvo įtraukta papildomų, iš anksto neplanuotų funkcijų, leidžiančių valdyti robotą klaviatūra arba vartotojui piešti programos aplinkoje. Šių papildomų funkcijų dėka buvo pasiektas patenkinamas rezultatas.

Introduction

Today more and more robots are being used for daily human needs. It becomes crucial to learn to use robots to our advantage and expand their capabilities. Our project - drawing LEGO NXT 2.0 robot could be first step to create a device capable of drawing large scale paintings, schemas or other kind of works, which are too large for casual printers. The idea of drawing robot is not new and has already been implemented, however any kind of device capable of printing large scale drawings are still hard to come by. The idea of the project is to check possibilities to create a device capable of printing large scale (possibly without any size limitations) using very simple and easily accessible resources, such as LEGO NXT robot. The goal of the project will be considered achieved if robot and its software will be capable of interpreting contoured picture and drawing it on a paper. The structure is divided into four main parts: analysis (gathering of basic information needed), design (possible looks and functionality of the robot), implementation (whole process of programming with more detailed information) and testing.

1. Analysis

This chapter will cover a significant part of information which was gathered before starting the project, such as task itself, available hardware, related work or why one or another decision about design of the robot, software to be used was made.

1.1 The task

The task our group chose to do sounded like this: "Drawing of the schema (contour, outline) with the help of the robot. Lego Mindstorms Educational robot could be used as basis. (Recognition of the edges of the drawing)". The task also required to draw proportionally correct picture in the center of an A0 format page.

During the meeting with our client the task was discussed in more detail. The client mentioned that the minimal requirement is that the robot could as accurately as possible draw a

square. It was also mentioned, that working with the robot and its software has to be user friendly - all processes have to be centralized and GUI must be simple to operate for casual user.

1.2 Hardware

The robot was supposed to be at our disposal only one month after the start of the project, so the first task was to get information and capabilities of it. Full list of parts [1] was found and the most important information about the NXT 2.0 brick (original NXT Intelligent Brick) [2] - the brains of the robot, which contain microcontrollers, input/output ports for external devices and are responsible for communicating with a computer.

The main NXT brick is quite weak at first look. The main 32-bit microcontroller has 256 KB flash memory and 64 KB RAM. That means that the controller is not able to calculate difficult operations and has to receive only simple commands from a computer. It has 4 input ports for sensors and three output ports for the motors. It has USB port, so the robot can be controlled via USB cable, or, in our case, use Bluetooth Class II V2.0 connection. The brick is powered by six AA rechargeable batteries.

The motors included were said to be precise, easily programmed to function together in order to move forward or backward or to turn in opposite directions to turn robot around. They also have programmable speed options.

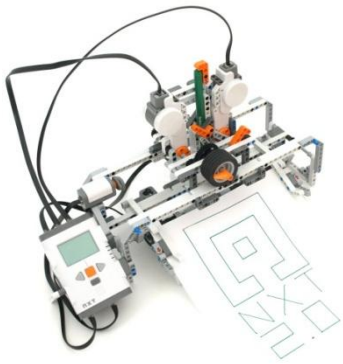
1.3 Design

During first lecture possible designs of the robot were considered:

- a printer like robot with moving head (Picture 1). Unfortunately, page size described in the requirements (A0) was too large, so we would have needed to make it move around or be capable of moving the page. Another problem that came to our minds that such kind of robot could draw horizontal and vertical lines easily, but diagonal lines would be

a problem. Although we thought this type of robot would have too many drawbacks, we decided to wait and see the parts of the robot we will have.

- a toy car like robot (Picture 2). This type of robot would be more practical because of its small size and maneuverability, so any type and size of drawing would become mostly a matter of programming.



Picture 1: A printer like drawing LEGO robot [3]

Picture 2: Possible looks of toy car like drawing robot [4]

From the whole list of accessible parts, such as touch sensor, three servo motors, ultrasound distance sensor, light and color sensor, the main brick with CPU and hundreds of non-electronic parts the printer-like robot idea was discarded immediately because of the lack of parts needed. It was considered as not practical, even though it probably would be easier to program.

The decision to make a toy car like robot was made and more detailed ideas of robot's design were considered:

- use a light/color sensor in addition to motors and make robot follow projected lines on the paper;

  This option appeals due to initial simplicity of programming, however that would require additional resources needed to create a suitable environment for it to work (e.g. a projector). While this option is suitable for drawing simple shapes consisting of continuous connecting lines, it is insufficient when it comes to replicating more complex

pictures which contain elements that are apart from each other — that would require some sort of navigation.

- to make a robot having only motors and orienting it by calculating motor turnings;

This option utilizes only the main brick, three motors, wheels, supporting part for the robot's chassis and marker lifting unit. That meant that all calculation of robot's current position, distances, turn angles have to be done through motor turns. Although it might not appear difficult task at first because of existing libraries, the robot has to make many turns and moves, so margins of error have to be taken into consideration.

1.4 Software

Our team had possibility to meet students who previously worked with the robot. They shared some basic information about firmware and programming of the robot (which is done by using Java language). Unfortunately the topics of our projects were not similar and we did not get any specific information apart that we could use several libraries (i.e. leJOS) .

Information about vectorizing images, extracting contour of an image was one of our early priorities. At first it was not clear how the process of image recognition should look like. The preliminary way to do this was to convert image to binary (black and white colors only) and then start algorithm to mark coordinates of every black pixel (and even better to mark beginning and ending points of every line). First thought was to work with vectorial graphics formats (i.e. svg), but it was decided that it would not be user-friendly decision, so working with more common picture formats, such as JPEG was chosen.

After the research was completed, the list of probably useful libraries was made: leJOS_NXJ [5] library was crucial for application inside the NXT brick;

net.coobird.thumbnailator [6] - a library which was supposed to be responsible for image processing;

many standard java libraries for i/o operations, GUI, logging and other tasks.

1.5 Related work

Although it was decided to work on our own and try to solve problems our way, some similar open source projects were quite easily found on the internet.

One of these projects was made and published by Lauro Ojeda in www.robotnav.com [7] and it was a first sign that decisions that had already been made were possible to implement. The code for the project was written in C language, so no certain decisions or conclusions could be made, because used libraries were different, not to mention all the methods, classes and so on. However, this work helped to make the first steps of the project.

The major difference which was noticed that the project mentioned above had an additional piece of hardware - gyroscope, which according to its description made robot's navigation a lot easier and more precise. As this gyroscope was impossible to find in Lithuania, it was decided to stop looking for this device and make the robot without it.

2. Design

In the design part it will be discussed the design of the robot itself as well as about software part with GUI to easily use the robot.

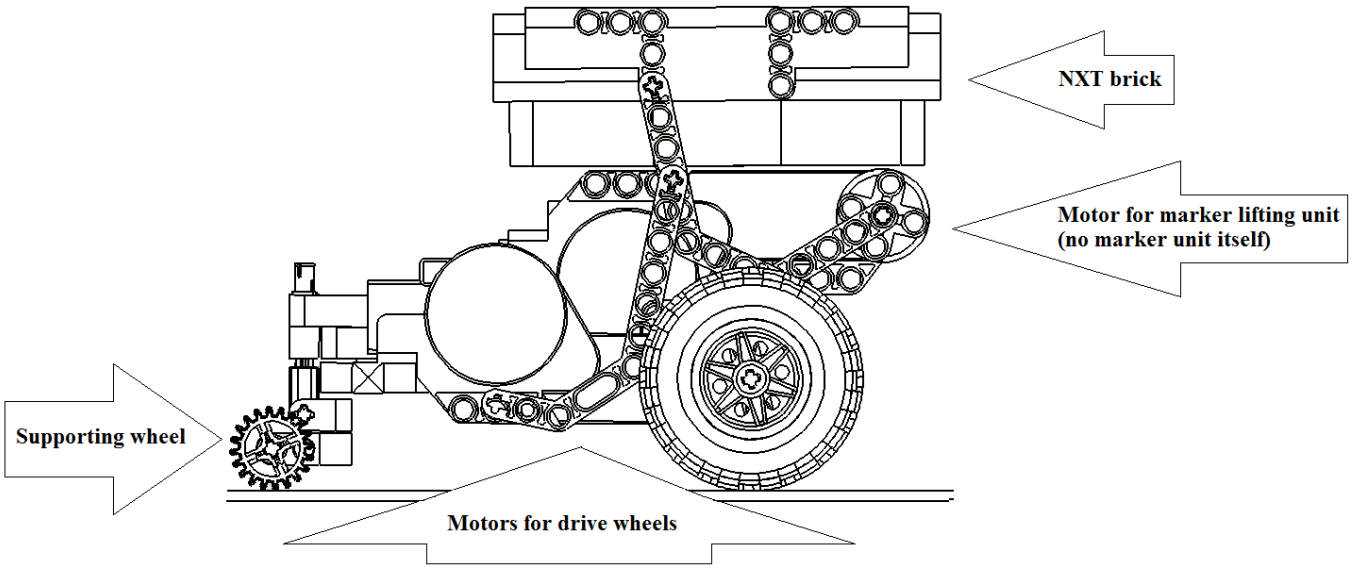
2.1 Hardware design

As it had been already decided to make toy car like robot orientating only by motors, last thing to discuss were several possible designs of the robot: to make it with tracks, four wheels or three wheels. Because high maneuverability was more important than the ability to drive over

rough terrain, it was chosen to make our robot with three wheels: two drive wheels and one small wheel for support. The looks of the robot changed only slightly during whole time of the project.

List of main parts used:

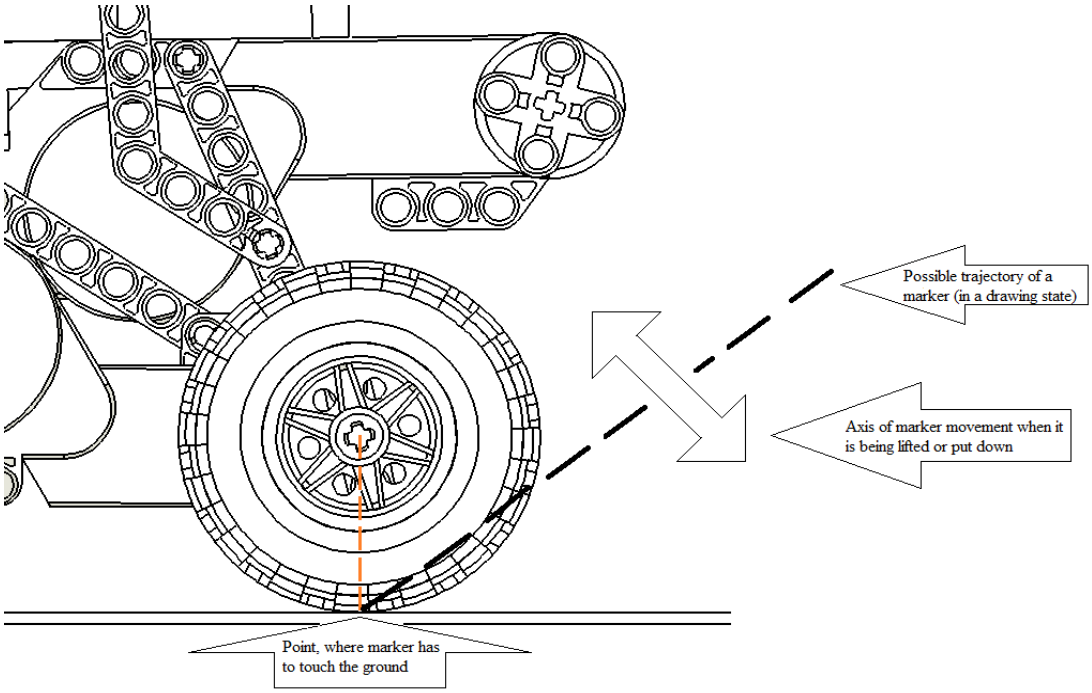
- two motors for each of main wheels so the robot could turn easily;
- one motor for lifting marker up and down;
- a supporting wheel at the back of the robot: the main tasks for it is to keep balance and robot horizontally, also to reduce friction;
- main body of robot with the NXT brick on top;



Picture 1: Side view of the robot

Full list of parts is in the annexes. The side view of the robot is shown in Picture 3.

One of the major problems which was faced while designing the robot was adjusting the marker so the tip of it would be located precisely in the middle of the imaginary axis between drive wheels so it remained in place when the robot is turning (schematics shown in Picture 4). The main drawback is that every time robot is used, the position of the marker needs to be recalibrated manually and that takes some time.



Picture 2: Schema of marker placement

Unfortunately, it was impossible to create absolutely solid model of robot — main wheels were bending because of soft axis and the marker unit was not precise because the motor's head (and whole construction attached to it) had a certain amplitude of free movement. Only the brick was connected to the chassis with additional pieces, that increased overall constructional strength and reduced robot's height.

2.2 Software design

As it was known that working with the robot should have been as easy and user friendly as possible, it was understood that the project has to have all software parts merged together and it is crucial to make a user friendly GUI.

The first idea, which was kept for the most time of the project was to make GUI which only requires user to select image, its brightness and size. All other processes proceed automatically without user even knowing about them. But implementation progress revealed need of additional functions (or features), which apparently resulted in more complex GUI.

3. Implementation

In this chapter it will be explained technical details of the project and sequence of tasks to reach the goal.

3.1 Drawing Digital Image

3.1.1 Process overview

The first choice of programming software was NetBeansIDE [8]. Unfortunately, a problem arose that NetBeans had to use NXTConnector library in order to connect to the robot (via Bluetooth or USB), so decision to switch to coding with Eclipse [9] was made, since it did not have this drawback and was also better suited to work with leJOS library. What is more, after installing leJOS NXJ plug-in Eclipse was able to create NXT application instead of java application and it could import nxt libraries easily.

At first libraries to communicate with the robot via Bluetooth were found. One of the first breakthroughs was to be able to give simple commands to the robot (Picture 5).

```
travel(4);

Motor.A.rotate(17);

arc(4, 360);
```

Picture 3: Example of commands to draw preprogrammed pictures

First line was used to draw a simple line. The number in brackets shows distance to travel measured in arbitrary wheel diameter units. Second line of code puts down/raises the marker. Motor.A indicates the motor used for marker lifting unit (as Motor.B and Motor.C were used for movement), and the parameter shows how many degrees the motor has to turn. The third line of code makes the robot to draw a circle. The first number in brackets indicates the radius of an arc, and the second number indicates how many degrees the robot has to turn. When the number is 360, a circle is drawn.

That made a good start to begin working with the main problem - drawing chosen pictures of various difficulty.

3.1.2 Image Processing:

For easy usage of the robot a quite simple but natively understood and effective GUI was written. All what user sees when program starts - a main GUI window (shown in Picture 6) with four different modes available (tooltips help to choose option for new users).

3.1.2.1 Selecting Image:

Process of selecting an image and its initial processing is explained in steps 1 through 6:

1) User selects option "Choose image" from main window and selects image he wants to be drawn (implemented file filter so only image types can be selected). It is done by pressing "Import image" button (see Picture 8) and navigating to the desired picture or dragging the image into an image field (implemented “file drop” functionality);

2) Selected image is resized to fit proportion of the page (594 x 841) but it's size is set to be 297 pixels in height and the width is set to 420 pixels because of the size of a marker (dimensions of

a dot made by the marker is 2 x 2 mm) by using thumbnailator-0.4.7-all.jar library. This is graphically explained in Picture 7;

For steps from 3 to 6 graphics2d [10] was used:

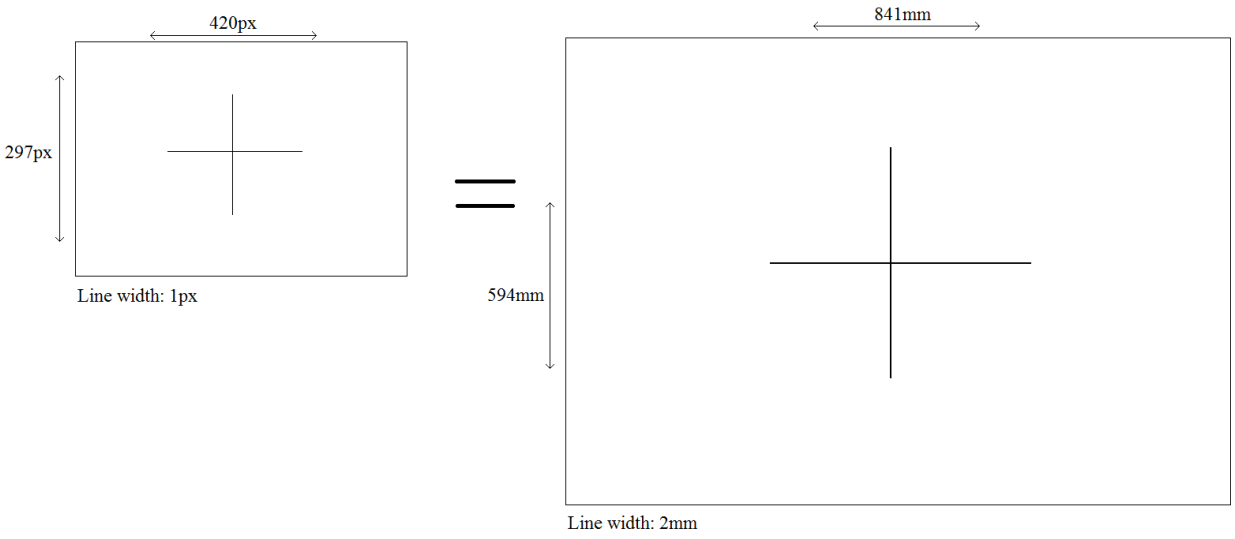
3) Image is two times brightened in order to get more precise contours;

4) Selected image is showed on the left side of the window;

5) Image is converted to binary (black/white) image and showed on the right side of the window;

6) User can change the intensity of the image (amount of black lines (spots) to be drawn) by moving slider on top of the window (see Picture 8);

Picture 4: The main window of GUI



Picture 5: Explanation why image is resized. On the left side - resized digital image; on the right side - image on A1 format page.

Picture 7 illustrates how given image is resized to scale well with given page format and drawing tool — in this case marker with a tip diameter of 2mm.

Canny Edge Detection:

As shown in Picture 8, the "Canny Edge Detection" checkbox is present to allow user to process the image using Canny Edge Detection algorithm [11].

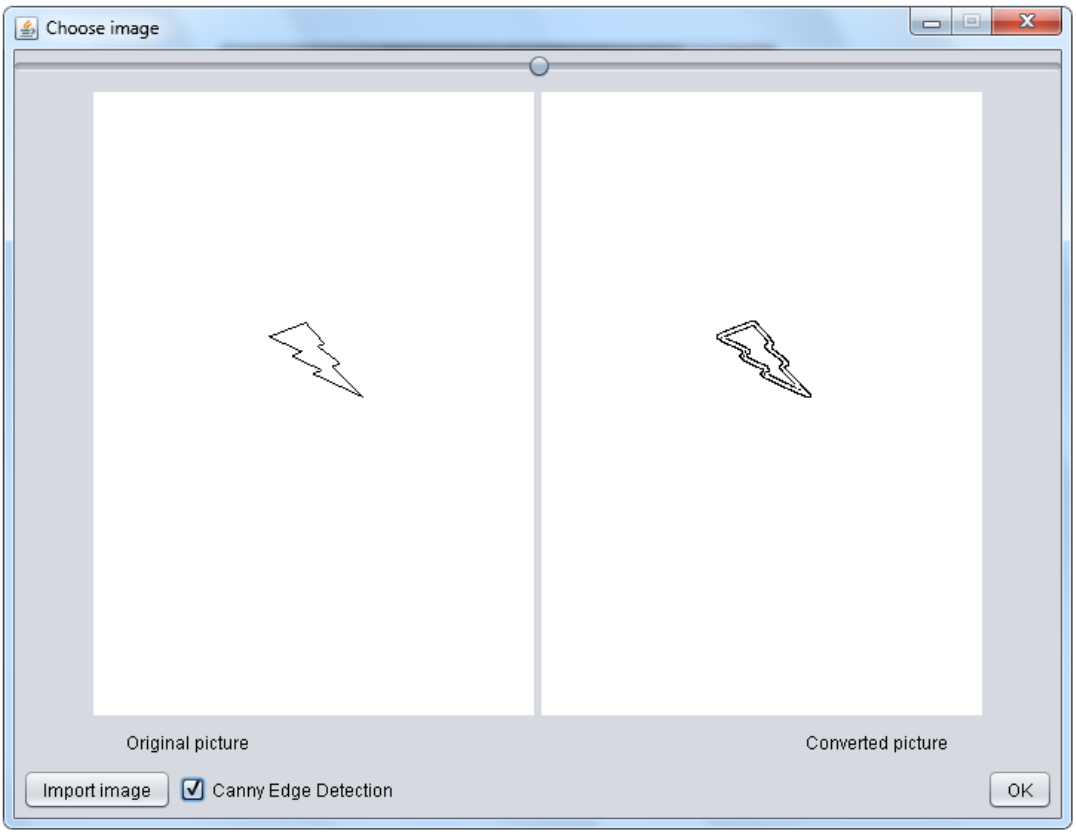
The algorithm works in multiple steps to find many edges in a given picture [12]:

1. Blurring – removing noise, to avoid treating minor contrasting points as edges.
2. Finding Gradients – finding where the pixel intensity varies most quickly
3. Non-Maximum Suppression – uses gradient vectors, each direction vector is rounded to nearest 45˚ angle, which makes pixels point to one another. Gradient vector magnitudes are then compared – for each pixel the corresponding magnitude is compared to the magnitudes of the pixels of the north and south directions according to its unrounded direction. If value of the current pixel is greater than both – the

positive and negative directions – the pixel is kept, otherwise it is removed from the image.

4. Double Thresholding – remaining pixels are compared to two set thresholds – high threshold and low threshold. Pixels with the values above high threshold are marked as strong, pixels with values that fall between the thresholds are marked as weak, and pixels with values below the low threshold are removed from the image.
5. Edge Tracking by Hysterisis – pixels are classified by the classification of their neighboring pixels. Pixels that are marked as strong in step 4 (Double Thresholding) are classified as edges. Weak edges are kept and classified as edges only if they are touching strong pixels/edges, those that do not are removed.

Image processed with Canny edge detection algorithm is shown to the user in the "Converted Picture" field, as shown in the Picture 8.



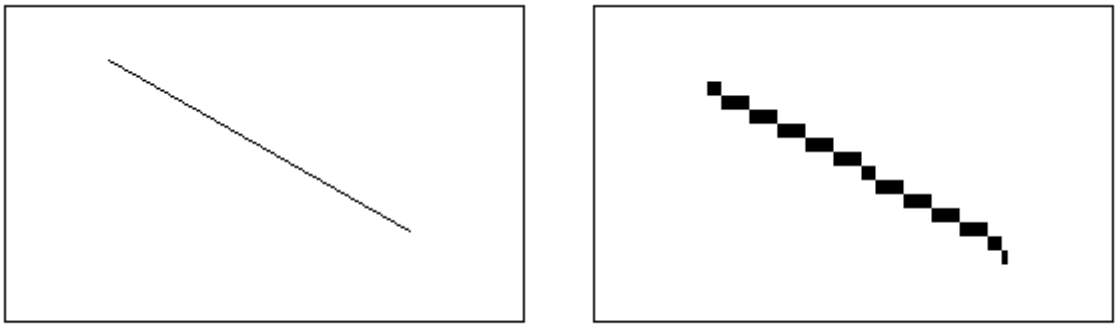
Picture 8: Example of image converted utilizing Canny edge detection

3.1.2.2 Image Vectorization

After user clicks OK , selected image is sent to processer class to be analyzed as a matrix of black and white pixels.

It is important to note that all lines in a picture are vertical, horizontal, diagonal (45˚) or a combination of those.

Diagonal lines that are not 45˚ are a prime example of combination of different lines to make, what appears to be, a straight line (Picture 9).



Picture 9: On the left- diagonal line which appears straight to human eye; On the right - same line zoomed in

Lines are represented as vectors consisting of two points — point of origin and ending point. They are stored in one dimentionalPoint [13] arrays where first element of the array (indexed [0]) is the beginning of the vector and the second element (indexed [1]) is the end of the vector.

Picture 10. An illustration of line with coordinates to be vectorized

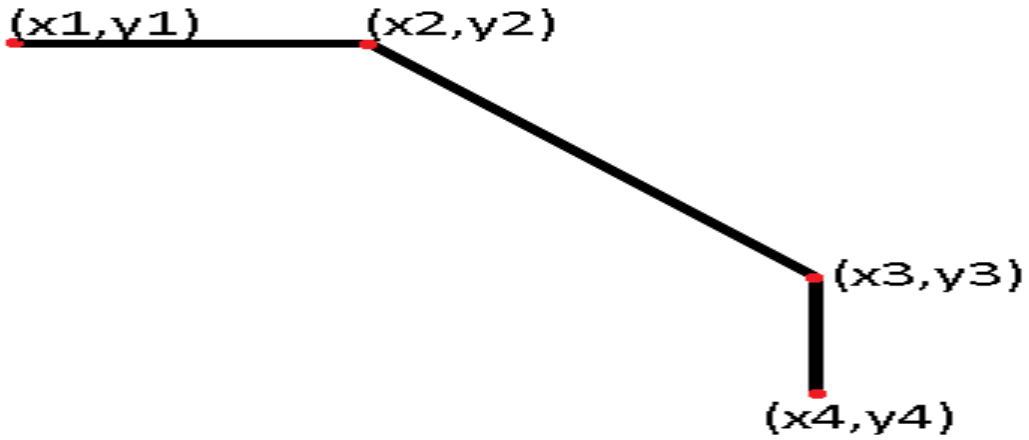
E.g. 1: A line shown in Picture 10 would be represented as

Point[0] is 160, 160

Point[1] is 660, 160

Therefore the robot would draw given line left to right.

All of the lines are stored in an ArrayList that holds Point arrays (defined as ArrayList<Point[]> *var*)



Picture 11. An illustration of multiple lines to be vectorized

E.g. 2: A curve such as one shown in Picture 11 would be stored as:

*Vertical[0]=x3,y3; *Vertical[1]=x4,y4;

*Horizontal[0] = x1,y1; *Horizontal[1]=x2,y2;

*Diagonal[0]=x2,y2; *Diagonal[1]=x3,y3;

And stored in an array list in that order (vertical, horizontal, diagonal), due to being found in that succession, their position in the picture bears no meaning to their position in the array list.

* - arbitrary name used only to make representation clearer, not actual name used in a program

All vectors are found using the same principles:

- Find the beginning of the line
- Look in the same direction until line ends
- Add found line to the list of all lines
- Remove found line from the picture to avoid repeatedly finding it again

The beginning and ending points of the connecting lines are duplicated so it would be easier to identify them as such. (E.g. 2: diagonal line connects to both horizontal and vertical line, therefore its beginning and ending coordinates are same as horizontal endings and vertical beginnings respectively).

A sample of algorithm to find vertical and diagonal lines written in pseudo-code is presented in Figure 1 and Figure 2:

1. Scan VerticalImage for first black pixel (scanning top to bottom, left to right)
2. Check if there is a black pixel underneath it
3. If there is a black pixel underneath it - set current pixel position as a starting coordinate of a vector. Else - go to step 1.
4. Continue checking for pixels underneath starting pixel coloring all checked pixels white until reaching the bottom of the picture or there are no more black pixels below.
5. Set last found pixel position as vectors ending coordinate and leave it black
6. Check if vectors length is at least 2
7. If length is 2 or more add starting and ending coordinates to ArrayList "taskai", go to step 1. – Else - go to step 1.

Figure 1: Finding vertical vectors

1. Scan DesinenlImage for first black pixel (scanning top to bottom, left to right)
2. Check if there is a black pixel diagonally to the bottom left of it (x-1, y+1).
3. If there is a black pixel to the bottom left of it - set current pixel position as a starting coordinate of a vector. Else - go to step 1.
4. Continue checking for pixels in the same direction from the starting pixel coloring all checked pixels white until reaching the bottom or the edge of the picture or there are no more black pixels next to it.
5. Set last found pixel position as vectors ending coordinate and leave it black
6. Check if vectors length is at least 2
7. If length is 2 or more add starting and ending coordinates to ArrayList "taskai", go to step 1. Else - go to step 1.

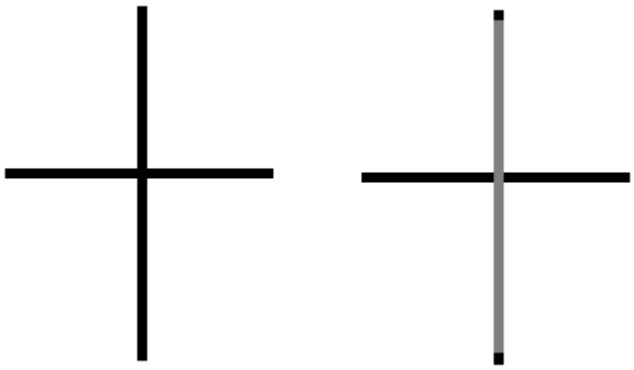
Figure 2: Finding diagonal vectors leaning to the right "/"

In order to ensure that no unnecessary short lines (Picture 12) are added, initially only long lines are found. Only after all long lines are found, remaining short lines are added to the list as connective elements (Picture 13).

Picture 12. Red pixels represent the unwanted line that would be found if no length constraint was placed during initial vector detection (Figures 1 and 2)

Picture 13. Black square represent the remaining pixels after all long lines are found (grayed out). They serve as connective lines between long ones

In order to reduce the number of lines and subsequently the amount of waypoints the robot has to traverse, each part of line finding/vectorizing algorithm works with its own copy of the original picture (noted in Figure 1 and 2), otherwise in case of intersecting lines one of them would be split in two (Picture 14), making it unnecessarily harder to sort them properly.



Picture 14.Case of using single picture for all line finding procedures. On the left is an example of intersecting lines. On the right grayed out line represents already found long vertical line splitting horizontal line in two.

3.1.2.3 Vector Partitioning

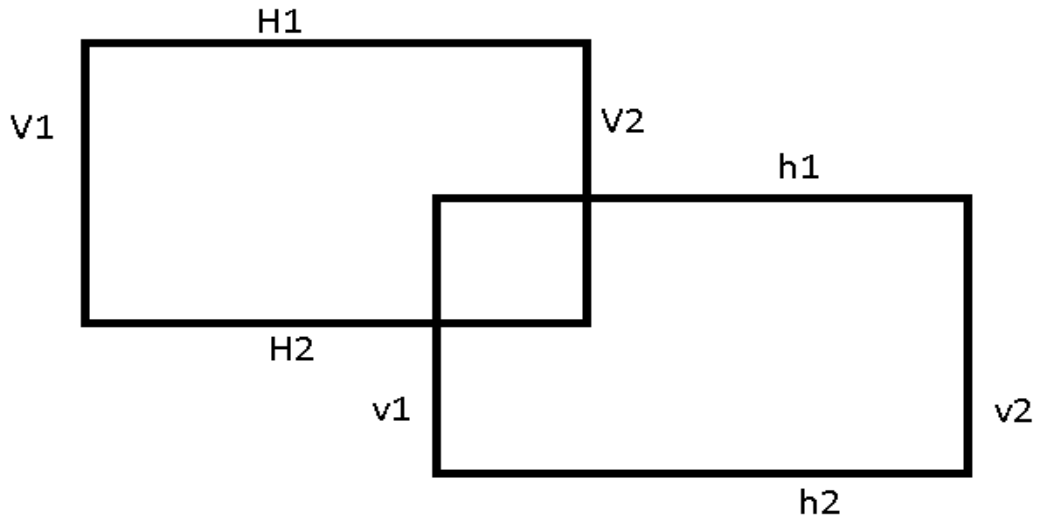
As vectors are found and formed, they are stored in a single list together, that presents an unfavorable environment to properly arrange them for robot to traverse. Considering given pictures are more likely to contain several separate contours/figures it is important to place them in a correct order. To do so it was decided to partition the lines that form those separate contours into different lists and subsequently order vectors in those lists.

General idea of vector partitioning algorithm is presented in pseudo-code in Figure 3.

1. Take a line from the original list of lines(a) and add it to the newly created list of lines(b).
2. Go through the original list of lines(a) and find all lines connected to the line taken in step 1. Add them to the new list of lines(b), remove them from original list of lines(a).
3. Take a line from a new list of lines(b), find all lines connected to it in the original list of lines(a), add them to new list of lines(b), remove them from original list(a).
4. Repeat step 3 until there are no more lines connected to any line in new list(b)/reached the end of the new list of lines(b).
5. Go to step 1.

Figure 3: Partitioning algorithm in pseudo-code

Following example delineates the partitioning algorithm.



Picture 15: Visual aid to clarify the description of partitioning algorithm. Picture consists of 2 distinct though intersecting shapes – 2 rectangles

Picture 15 serves as a simple exemplary scenario where there is a need to separate two distinct figures in a given picture. Letter/number combinations near figures symbolically

represent horizontal and vertical lines of given shapes (“H” and “V” respectively), uppercase letters denoted to upper-left rectangle and lowercase letters to lower-right rectangle.

After finding all lines as described in chapter *3.1.2.2 Image Vectorization* the resulting initial list of lines is as follows:

Original list (a) V1, V2, v1, v2, H1, h1, H2, h2.

Partitioning begins with the line V1, it is placed in a new list (b1) and removed from original list (a)

List (b1) V1

Original list (a) V2, v1, v2, H1, h1, H2, h2.

Next step is to find all lines connected to it and add them to new list (b1) and remove them from original list (a):

List (b1) V1, H1, H2

Original list (a) V2, v1, v2, h1, h2.

Afterwards it is checked if any lines in list (b1) are connected to any lines in original list(a):

- V1 from list (b1) is not connected to any line in original list (a)
- H1 from list (b1) is connected to line V2 from original list (a). Add V2 to list (b1) and remove it from original list (a).

List (b1) V1, H1, H2, V2

Original list (a) v1, v2, h1, h2.

- H2 from list (b1) is not connected to any line in original list (a)
- V2 from list (b1) is not connected to any line in original list (a)

Last bulletin illustrates the important part of partitioning algorithms functionality --- step 3 in Figure 3 defines iteration through newly formed list until the end of the list is reached,

which means that it will keep looking for connecting lines even if new list is modified (2nd bulletin) until there are no more valid matches.

After list (b1) is completed — last element of the list yields no more matches — the process is repeated again on the original list (a), this time starting with v1, which is added to the new list (b2).

After the partitioning is complete there are 2 lists left:

List (b1) V1, H1, H2, V2

List (b2) v1, h1, h2, v2

As intended, lines that make up two separate shapes are divided into two separate lists.

Slip-ups and their correction:

Due to implementation of this partitioning algorithm utilizing looping through lists, deletion of elements (which forces shifting of position of elements in all cases but one, where said element is last in the list) and change of counter as loop progresses makes the algorithm skip certain elements, which in some cases are a part of the given contour. In situations like that those elements are left on their own in their separate lists.

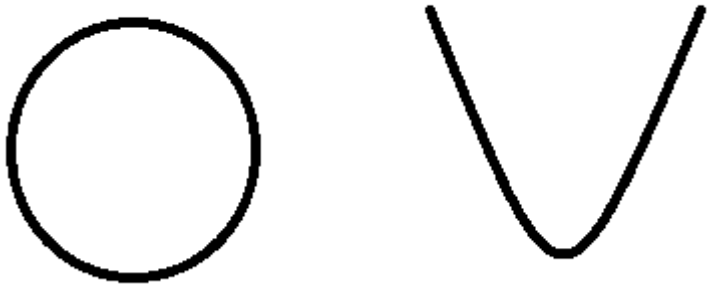
To rectify that, additional crosschecking of elements in all lists is added, which checks if any elements in different lists are connected, if there is a single match — those lists are merged into one, defining them as one contour.

3.1.2.4 Vector Ordering

At this point vectors are already found and divided into separate lists that define different shapes in a given picture. For the robot to correctly draw the picture vectors have to be properly ordered.

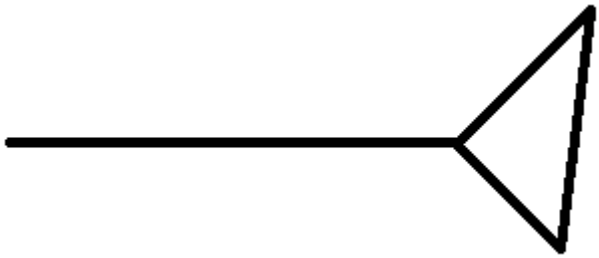
In order to do so, few things need to be done:

1. Every contour has to be checked, if it is enclosed or an open contour (Picture 16).



Picture 16: Circle on the left – closed contour, V-shaped contour on the right – open contour

If contour is enclosed it does not matter which point or vector is the beginning of the contour. If contour is open it is important to choose the beginning of the contour as one of its "edges". Edge is defined as a line that connects to another line on only one of its ends. If a line has lines connecting to it on both ends, it cannot be the edge.

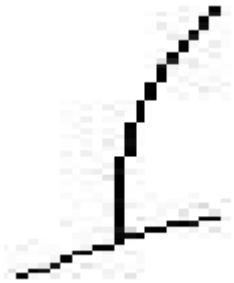


Picture 17: Example of a shape with an edge line

As illustrated in Picture 17, the horizontal line is considered the edge, since it has lines connecting to it only on the right end, therefore it should be taken as a correct starting position of the contour. While example in Picture 17 does not show why is it important to take the edge as the starting point, V-shaped contour in Picture 16 shows that unless one of its edges (top points) is not taken as a beginning point of the contour, it will be impossible to draw it in a single swipe (without lifting the pen and changing the position).

2. Algorithm has to ensure that lines are connected one to another instead of juggled.

   This effectively eliminates the possibility that lines will be in an improper order, which could result in a messed up drawing. That is especially true in cases where there are any kinds of forking of lines



Picture 18: Example of a line fork in a picture

Picture 18 demonstrates a possible fork in a picture, which results in a fork in a drawing path of a robot. As they are these lines are stored in one list, since they match the criteria of the partitioning algorithm --- they connect to one another somehow. However it is clear that it is impossible to draw them in a single swipe, therefore, if ordering of lines is implemented as juggling (changing positions of the lines in the same list) it is bound to make mistakes.

The algorithm to connect the lines properly work as shown in Figure 4:

1. Find the edge of the contour (if exists) in original list, add it to the new list. If contour has no edges, add any line of this to the new list.
2. Move on the next line in the new list (if it is a first iteration --- first line)
3. Find a line in the original list that connects to the added line, add it to the new list if it is not yet present (to avoid duplicates/going back and forth on the same path).
4. Go to step 2
5. If there are no more lines in the original list that connect to the line added in step 3 – remove all lines in the original list that have been added to the new list
6. Go to step 1
7. If original list is empty – remove it

Figure 4: Line connection algorithm in pseudo-code

After these steps have been completed on all lists (all contours), there should be same or higher amount of properly ordered lists. Reason for that being, that due to steps 2 and 3 if there is a fork in a picture there will be formed corresponding amount of lists. In case of Picture 18 – 2 lists, one going from the top, through the forking point to the bottom left corner, another going from the forking point to the right. It ensures the proper way of traversing the path for the robot without unneeded lines.

3.1.2.5 Ordering of Contours

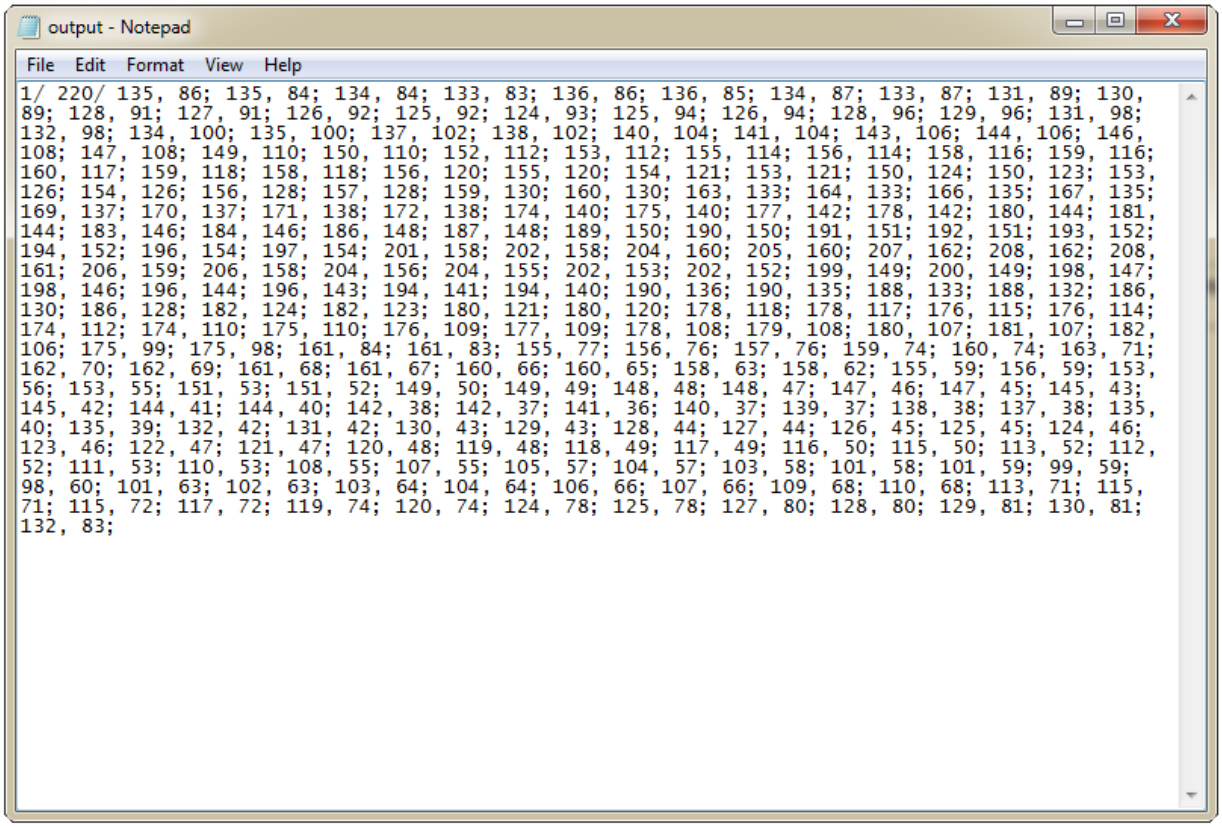
After all properly ordered contours are formed, the remaining part is to create a distance efficient order of contours for robot to draw. It involves measuring the distances to contours from robots position. By default robot has to start from the top left corner of the surface it has to draw on, that point is set as starting position in Cartesian coordinates (0,0). From that point distances to all contours are measured, and the closest one is set to be drawn first. Then same process is repeated, only from that contours ending point, to find the contour that is closest to it.

3.1.2.6 Drawing

Processed contours, consisting of sets of vectors are outputted to a text file for the robot to interpret and draw.

The general format of an output.txt file is: M/ N/ (x_1 , y_1) ; (x_2 , y_2) ; ... ; (x_N , y_N);

where M is the number of separate contours in a picture, N is number of dots that need to be traversed to draw a contour (not every dot in the picture is in the output.txt), x and y are coordinates of a dot.



Picture 19: Example of output.txt

Effectively converting diagonal lines into waypoints appeared to be more difficult than expected. That can be illustrated by looking again at the example of lighting bolt (Picture 8) and the output.txt of that image (Picture 19). There are obviously a lot more coordinates of points than there are turning (corner) points in the image. This happens because the line, which appears straight to human eye is not straight to a computer (Picture 9) and is treated as many horizontal or vertical lines. For this reason the robot has to make a lot more calculations when moving from point to point and the quality of a drawing is significantly lower.

As the process of picture recognition and drawing started to become automated, previously used approach of preprogrammed commands was not suitable anymore. So with a help of a launcher.nxj application it was made that the robot could go through given coordinates in a output.txt file and interpret them as waypoints. These waypoints are easily used by leJOS.robotics library, which has many methods to use for travelling from one waypoint to another.

Unfortunately, this way brought another major problem: DifferentialPilot object, which is most important object in drawing process, was responsible only for two drive motors, so the motor lifting marker was inactive. So robot's movement capabilities were not synchronized with drawing process - marker was being lifted or put down in the middle of the way from one point to another. That happened because software was giving commands, but did not wait or check if hardware managed to complete what it was asked to. This was the point when commands to put the thread to sleep were introduced. A simple schema for that would look like this:

Robot's movement → putting thread to sleep → moving the marker → putting thread to sleep → robot's movement → ... → moving the marker

Explanation of the schema: robot starts with marker lifted up and moves to first contour's first point. Then the marker is put down and robot goes through all points in that contour (that is one robot's movement phase in the schema above). After that, the marker is elevated and robot moves to the first point of second contour (if it exists). This process is repeated until picture is finished. The time of thread sleeping depends on complexity of following command - for more complicated commands more time is dedicated to complete.

3.2 Additional features

3.2.1. Real time drawing

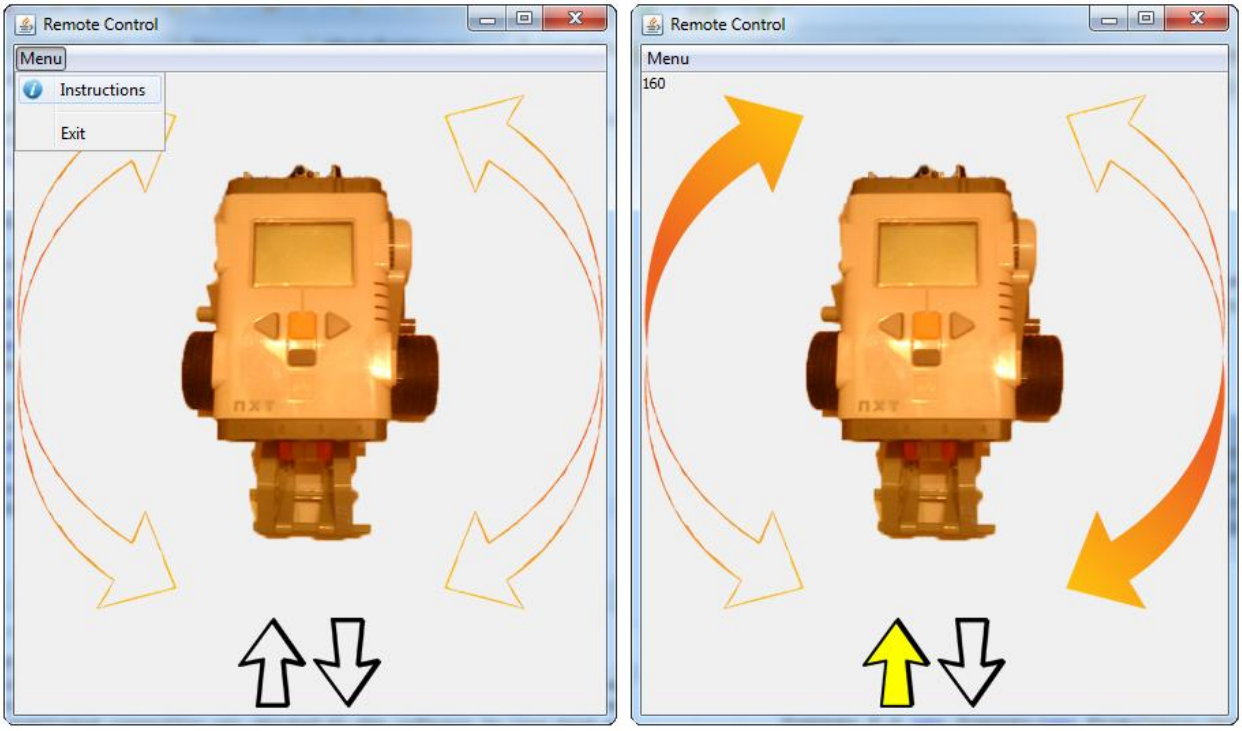
As time was running out and quality of drawn pictures were still far from expected, the possibility to draw in real time was developed. It was considered not only as "nice to have" feature, but possibility to combine this feature with drawing a digital image was kept in mind. This meant that if these two different approaches of drawing were combined, user would have option to correct trajectory of the robot if it drew incorrectly.

The first version of this feature had its own application, separate GUI and could have been launched only as stand-alone program and only if the robot was turned on and in range. The application consisted of main window (Picture 20) and window with control instructions (Picture 21). Later, the feature was improved: it could be used as a standalone feature ("Free mode") or it could be used to adjust trajectory of robot while it drew uploaded picture ("Editing mode").

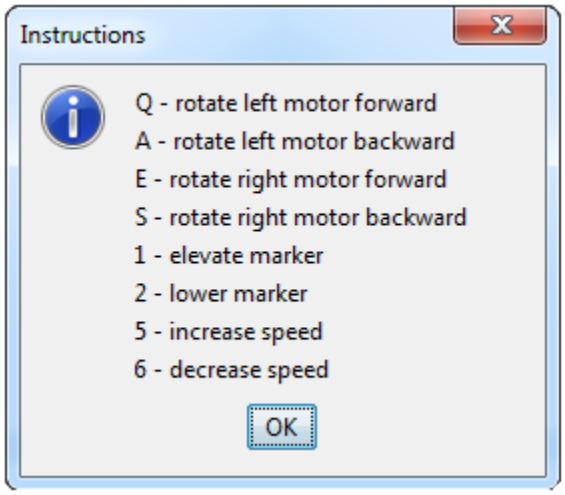
“Editing mode” uses multithreading technology. One thread is responsible for reading drawing path data from output.txt file and moving robot’s servo motors on basis of contours and coordinates which are read from file. Another thread is window with GUI (Picture10). It also contains key adapter which responds to presses and releases of some keyboard buttons and moves respective servo motors and sets speed. When robot is gone through every coordinates application stops working. Manual program terminating is also possible.

“Free mode” is like simpler version of “Editing mode” it only uses later mentioned thread — window with GUI and key adapter. Terminating of this application is only manual.

The control options allow user to rotate drive motors separately, elevate or lower the marker and control speed options (the lower the speed is, the more precise turns are made).



Picture 20: On the left - feature's main window with menu options expanded; on the right - window when several control buttons are pressed



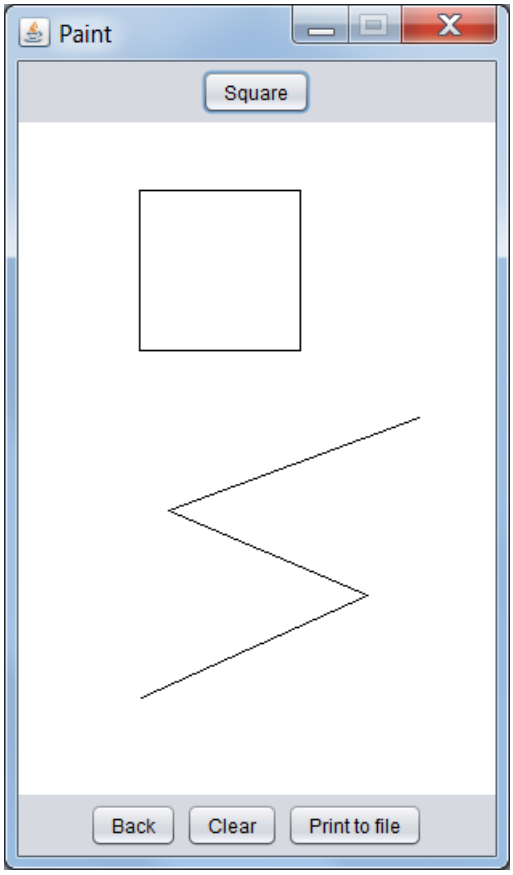
Picture 21: Window with control instructions

3.2.2. Paint

It was mentioned several times in this report, that many unexpected problems were faced during the project - insufficient stability of construction, diagonal lines were recognized as huge sets of shorter lines. Because of these problems quality of pictures were not such as desired, so a

new feature was introduced. It was called "Paint" as it allowed user to draw his own pictures by using straight lines.

The process of this feature is simple: user starts drawing contour picture with left mouse button and adds as many straight lines by pressing left mouse button repeatedly. After pressing right mouse button, current contour is finished and coordinates of points are put into a list. User might start another shape by repeating mentioned actions or start drawing by pressing "Print to file" button on bottom-right corner of feature's window (Picture 22).



Picture 22: GUI Window for Paint feature

The main advantage of this feature is that lines, painted by user, are defined only by two coordinates. That means that one line, no matter of its angle will have only coordinates of its starting and ending point and will not be fractured into small pieces as it happens when drawing uploaded image (Picture 23). What is more, lines are saved in the exact order they are painted. This is a huge perk for the software, as it no longer needs to recognize lines, intersections, do any

kind of sorting. All it has to do - generate output.txt file (format is the same as described previously in the section *3.1.2.6 Drawing*).

Picture 23: On the left - how line is painted when image is uploaded; On the right - how same line is drawn through Paint feature

4. Testing and future work

A certain amount of testing has been done through the whole project, after completing each of the subtask: drawing by using preprogrammed commands, after every update of algorithm, which converts image into coordinates and after additional functions (drawing in real time and Paint) were introduced.

All testing was meant to check robot's ability to draw on paper without skidding or rumpling the page, its precision and time used to draw one or another picture. Unfortunately, precision results were not satisfactional when drawing complex pictures by the time third draft version was written. Even though it was expected, that quality of pictures would become a lot better after finalizing algorithms to find and sort edges of a picture, unfortunately, secretly desired goal of redrawing Mona Lisa was not achieved.

A lot of testing scenarios with pictures of different complexity showed, the decent painting quality is kept while picture contains no more than two separate shapes. Of course, these shapes should not be too complex either.

To reach more realistic goals, for example — drawing schemas on large scale canvas, much more improvement is required. At current point we are feeling confident about algorithms which processes pictures, however, the main problem remains robot's lack of precision. As the this problem is related to constructional weaknesses, lack of more advanced pieces of hardware, limitations of hardware at our disposal and probably imperfection of written software, all these things have to be taken into serious consideration in the future.

References

[1] List of parts in the LEGO NXT 2.0 set.
http://brickset.com/sets/8547-1/Mindstorms-NXT-2-0

[2] Example of similar project.
http://www.legoengineering.com/wp-content/uploads/2013/06/download-tutorial-pdf-2.4MB.pdf

[3] Image of a printer like LEGO robot.
http://robotsquare.com/wp-content/uploads/2012/02/nxt-printer.jpg

[4] Image of a toy car like LEGO robot.
http://www.olvnorthville.org/images/robotics.jpg

[5] Official site of leJOS libraries for LEGO Mindstorm robots.
http://www.lejos.org/

[6] Main page of thumbnailator library.
https://code.google.com/p/thumbnailator/

[7] Example of similar project.
http://www.robotnav.com/drawing_bot/

[8] Home page of NetBeans IDE.
https://netbeans.org/

[9] Home page of Eclipse IDE.
https://eclipse.org/home/index.php

[10] ORACLE documentation of Class Graphics2D.

http://docs.oracle.com/javase/7/docs/api/java/awt/Graphics2D.html

[11] Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence.

http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4767851&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4767851

[12] The Canny Edge Detection Algorithm M.Skinner.

https://www.youtube.com/watch?v=-Z3kr26Eci4

[13] ORACLE documentation of Class Point.

http://docs.oracle.com/javase/7/docs/api/java/awt/Point.html

[14] Online list of LEGO NXT parts with detailed information.

http://brickset.com/inventories/8547-1

Annexes

Parts needed [14]

Amount	Image	ID
2 x		4184286
2 x		4297210
3 x		4297008
2 x		4177430
3 x		4119589
6 x		4211573
3 x		4211639 (5M)

Amount	Image	ID
1 x		4535768 (9M)
3 x		4211622
1 x		4210751
1 x		6034375
1 x		4522939
1 x		4107081
1 x		370826 (12M)

4 x  4211668

6 x  4495932

4 x  4514553

2 x  4666579

1 x  4211889

2 x  4210668

1 x  4211651

12 x  4211807

1 x  4297200

2 x  4142865

1 x  4143466

2 x  4537417