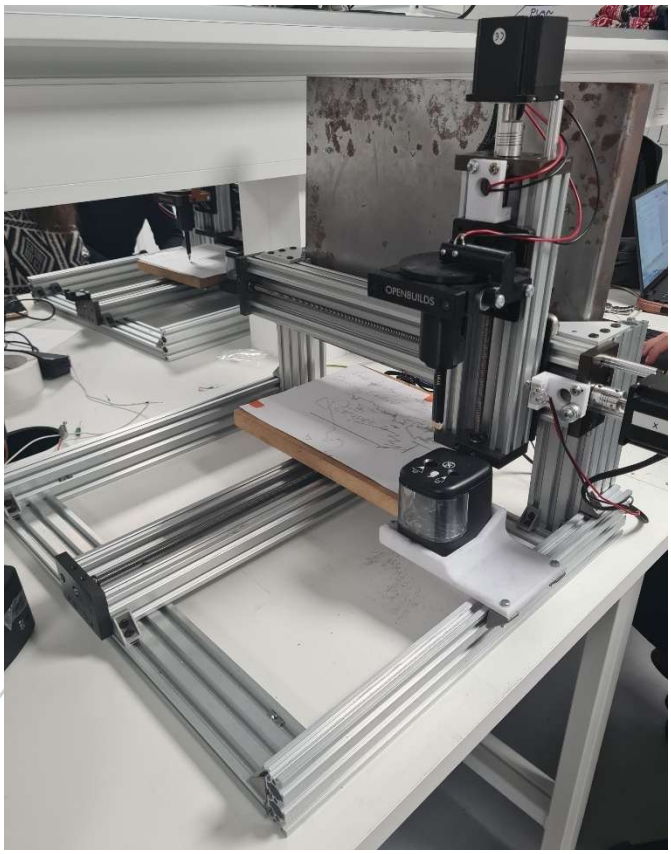
12/18/2024

Semester project

Drawing robot



Teacher: Frederik Hagelskjær

Authors: Noah Valbjørn Vryens, Emily Rodenas Aparicio, Emma Fredensborg Schmidt, Louise Marie Koch, Dominik Felten, Frederik Svend Vester

Group: 6

Organization: SDU

Line: Bachelor of engineering - Robotics

Contents

Distribution of tasks

Table 1: Distribution of Tasks:

Task	**Main responsible people**
Image processing	Frederik Svend Vester
Communication between Java and PLC	Noah Valbjørn Vryens and Emily Rodenas Aparicio
Graphic User Interface	Noah Valbjørn Vryens
Processing of G-code on PLC	Frederik Svend Vester, Noah Valbjørn Vryens and Emily Rodenas Aparicio
How can we move the robot in the z-axis?	Emma Fredensborg Schmidt, Louise Marie Koch and Frederik Svend Vester
How can we move the robot in the x-axis and y-axis?	Dominik Felten and Frederik Svend Vester
How can we null all 3 axes?	Dominik Felten and Frederik Svend Vester
How can the robot move away from the limit switches?	Dominik Felten and Frederik Svend Vester
Pencil sharpener	Emma Fredensborg Schmidt, Louise Marie Koch and Frederik Svend Vester
System integration	Frederik Svend Vester and Noah Valbjørn Vryens
Evaluation and testing	Frederik Svend Vester and Dominik Felten

Introduction

The main objective of this project is to develop a robot that can draw a 2D image from a digital image. This robot is developed using a 3-axis cartesian robot, moving with 3 stepper motors and controlled by a PLC. To draw the image, a holder for a pencil and a large pencil is attached as a toolhead, and paper can be either laid or taped to the drawing board below. An electric pencil sharpener has also been attached to the side of the robot, to facilitate the sharpening of the pencil. To convert the digital image into something our robot can understand and draw, a program in Java was developed. To test the final robot, multiple images were selected to be drawn. These include but are not limited to Figure 24. The results of these drawings can be seen under Figure 26.

Problem analysis

To achieve the project's objectives, it is essential to understand the components of the drawing robot. The given drawing robot components consist of a PLC (X20CP1382), a stepper driver board (ST330), DC stepper motors, limit switches, a power supply, paper, a pencil and a pencil sharpener. This robot operates with 3-axis indicated by the DC stepper motors.

One of the main parts of the problem is how to take an image and make it into code that the robot can read and make into movements that then results in a drawing.

Furthermore, the communication between the Java program and the PLC is crucial for precise control of the robot's movement to draw based on the generated commands or the path data.

As the robot draws, the pencil wears down and therefore must be sharpened periodically to maintain the drawing quality. Therefore, it is essential to figure out how the pencil sharpener can automatically sharpen the pencil and how it will be integrated into the system.

Components of the 3-axis robot drawer

PLC

PLC is an abbreviation for 'Programmable Logic Controller'. The PLC is from BR Automation Studios (BR AS). PLCs are digital computers and primarily used to control industrial automation machinery processes. PLC monitors the inputs and outputs (I/O), process data and control sensors/actuators. PLC X20CP1382 is what controls the 3-axis of the robot with the help of sensors and actuators in communication with Java program.

Sensors and actuators

The robot drawer has limit switches for the 3 DC stepper motors. The limit switches are connected to the input terminal on the PLC. The limit switches help determine movement boundaries and the position of the tool/pencil with the help of the DC stepper motors. The PLC is connected to the stepper driver board, while the driver board connects the DC stepper motors. Stepper driver board ensures that the DC stepper motors receive correct power and controls signals.

In the context of the project, this part shows how the hardware components of the robot drawer are connected.

JAVA program and PLC communications:

Communication and file transfer

To use the results from the PC, i.e. the g-code (generated commands), on the PLC, we decided to use TCP, which is explained below, to communicate and send our data from the PC to the PLC. This is done via an ethernet cable and using the PLC as a "host" or "server", which the PC connects to and sends information back and forth with. Alternatively, we could have used a USB thumb drive to transfer the g-code, but we wanted to make it more automatic, thus only requiring the input of an image to be drawn. Another option was setting up an OPC server, but due to our limited knowledge using OPC, we elected to use TCP.

G-code

Geometric code, in short is g-code, is a programming language widely used to control CNC (Computer Numerical Control) machines. G-code is a series of generated commands that direct the machines' movements with specific coordinates, speeds, tool control and other operation functions to perform precise tasks at hand point to point. In our case, the CNC machine is our robot.

To read the series of g-code commands, we need to know some key ingredients for understanding the structure of the g-code. Here, we have a g-code structure:

G## X## Y## Z## F##

And here is a list of ingredients on how to read g-code:

Movement commands:

- G00: means rapid positioning, meaning moving the tool/pencil at max speed to a new position point.
- G01: linear interpolation (a straight-line movement).
- G02/G03: circular interpolation (clockwise or counterclockwise).

Cartesian coordinates in three-dimensional system or x-, y- & z-coordinate system:

- This part specifies where the position of the pencil must move to. For example: G01 X10 Y10, the pencil must now move coordinates (10.0, 10.0).

Speeds:

- F: Feed rate, the speed of the tools movement.
- S: Spindle speed, the speed of the spindle.

Other:

- M codes: Miscellaneous functions like for ex. M00 = Program Stop.
- G28: Return to home or its reference point.

Now, let's look at one g-code command line, which we copied from another site (Dejan):

G01 X247.951560 Y11.817060 Z-1.000000 F400.000000

So, in this g-code example, we are reading the instructions of the pencil/tool movement from our reference point, where we are now, to move to this new point (247.95…, 11.81…, -1.0) in a straight-line movement with the speed of 400. The unit selections for the g-code above can be in inches or in millimeter, which are calibrated behind the scenes.

In the context of this project, g-code is our generated commands, and is used to control the precise movements of the drawing too, converting image data into drawing/movement commands that the robot can execute.

Transmission Control Protocol (TCP)

TCP is a fundamental framework for computer networking. TCP has a built in "handshake" protocol that ensures, that the delivery of data between applications over the Internet Protocol (IP) is reliable, ordered, and error-checked, so it is guaranteed that the packets or small segments of a larger message is delivered accurately. In this project, TCP will be used to configure a communication bridge for Java program and the PLC, so that the robot drawer can move with reliable information. (geeksforgeeks, 2024)

With this information of g-code structure, TCP communication and how the robot hardware is set up, our robot can now be effectively controlled and be consistent with the drawing quality. However, we expect that there will be challenges in setting all of this up, and perhaps new information will be found later. Any new information will be documented and written in the "Subproblems" or in later sections of this report.

Problem statement

Based on the problem analysis, a problem statement has been created:

How can the 3-axis robot be used to draw an image that is controlled by PLC?

As part of answering the problem statement above, the following questions and tasks needed to be answered:

- How to process the image needed to be drawn by the 3-axis robot?

- How can the Java code send the file through TCP?
- How can the robot load the g-code?
- How can the robot process the g-code?
- How can we move the robot?
- What is g-code and how is it used to solve this project?
- How and when do we use the pencil sharpener in the drawing process?
- Which/what other drawing methods could have been used?
- What implementations can be improved in the system?
- What components does the robot drawer consist of?
- How can we improve the user experience?

Project limitations

To limit the scope of the project, the following choices have been made:

- BR Automation Studios will be the coding program of choice with the Structured Text programming language for the PLC of the 3-axis robot.
- The Java programming language will used for the drawing instructions of the image and to generate the g-code (commands).
- The image drawn on physical paper will be only black and white.

Subproblems

How can we process the image needed to be drawn by the 3-axis robot?

PHP version

One of our groupmates, Frederik, has been working in the industry and had a CNC mill he could access for many years. So, he already has a working solution to this, made in PHP. Therefore, he is very knowledgeable on this part of the project already, when we were given the assignment. However, we have made many new versions since then, also due to the edge detector, a java program made and given to us by SDU.

The PHP version, which he made many years ago, was made with g-code version of an image so it could be milled in a steel plate. It worked by making the image greyscale and pixelating the image $2^8 = 256$ to save some time, as there is no reason to run a new command every pixel. The depth was dependent on the grey color of the image.

There was a configuration file that made it possible to grade the depth. Dark black was 0 and white was 255. An unsigned byte is normally 8 bits, which represents a number: $2^8 = 256$.
0 is also represented here so we have only up to 255. Nothing says that you can only use a byte to represent the greyscale. Therefore, it was just an arbitrary choice to only divide it into a byte or 256. You could set max and min as you wanted, and it would define how the z-axis would be placed at any point. It was running from the top of the y-axis, running from start x0 to the end of x-axis. Then starting on the next y-line.

However, for this assignment, it was required to use Java and not PHP, so we needed a new version in Java.

Java version 1

Version 1 was a simple replica of the PHP version with a configuration file. Reading from an image and converting it to different z-depth. The size of the g-code output was directly related to the size of the image. A pixel in the image filled a space in an array with a byte. So, a 1000pixel × 1000pixel image would be 1mio array bytes with a number between 0 and 255.

Due to the large processing arrays, it took a long time to convert an image. So, we were only using pictures of very low resolution.

It was running like the PHP version, like a printer and was printing an image in different z-depth. This was before we saw the printers, so we thought we could use the same principle and apply different pressures to each point, making the point more or less dark.
Due to the setup of the printer, this version was abandoned, and a new version was created.

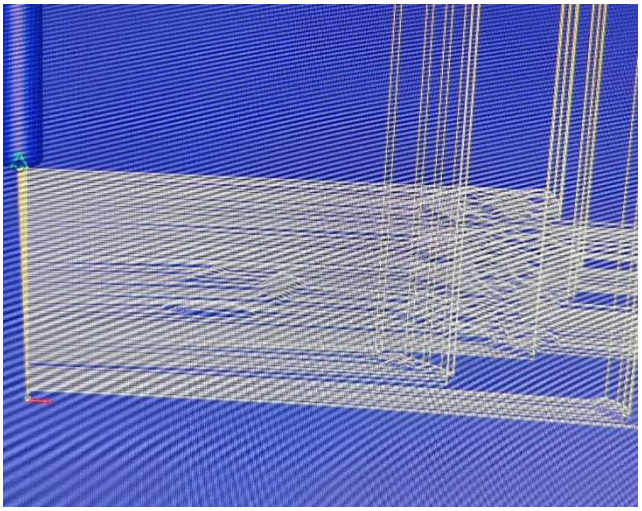


Figure 1 LLP Vemmelev ApS logo printed out as g-code

Why did we choose g-code?

As an industrial technician, Frederik is familiar with g-code in work with robots. The main reasons we choose g-code:

- Frederik has a lot of experience reading it. So, it's easy for us to debug.
- We have programs to help with this.

- It's used everywhere. From your home 3D printing to industrial scale user programming
- We can command it to do more things than just to draw.

Downsides:

- Everything is a string, so it fills up a lot of space on the machines, including empty space that could be removed if it was all numbers.
  We could, for example, send an array of 3 doubles instead. That is 4 bytes pr. float which equals 16 bytes of data on each line, compared to each line being 12 chars of 2 bytes which is 24. Note: We would use floats instead of doubles due to the nature of the programs which are outputting low numbers. This is because we would run out of surface on the A4 paper we are drawing on. So, we would never draw more than $210mm \times 297mm$. And we would never have to draw more detailed than the pencil made possible. Which is about 1mm in diameter on the tip.
- We must get the numbers out of the strings at the other end.

This is however an easy problem, due to the fast CPU of today, and we would choose this way because it is so easy to read, which also became handy later.

We had our first drawing printed, but found that after some lines, the printed picture was not recognizable. This was because the PLC was not getting its buffer removed. The readability of g-code really made it easy to see where the problem was coming from.

Java version 2

To save time converting images, we changed the configuration file to contain resolution, maximum x-axis space and maximum y-axis space. The idea was to resize the image down to fit the resolution in the file and thereby save time converting the file.

At this point in time, we were shown the drawing robots. So, another change we made was to make the z-axis binary. If the image contains a color that is darker than 200 in binary, it will print. Else we would stay above the paper.

Figure 2 Low resolution image, that we used to generete g-code

Java version 3

We were given the edge detector, that finds edges and applies a filter to make a line around the edge. Therefore, we could abandon the way to greycolor the image and run it binary.

Figure 3 A picture we took from the internet. (dyrenesbutik.dk, 2022)

Figure 4 Output from edge detector

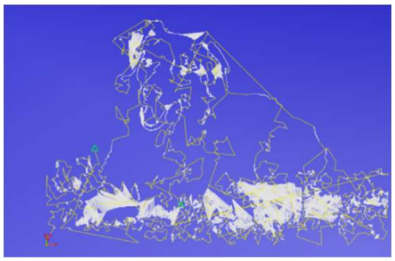


Figure 5 Output from gcode converter

Another change was the way we were drawing the image. Since we were running binary and either should draw or stop drawing, we had no reason to draw from left to right like a printer anymore. We would like to print where the lines were, but how would we do that? Instead of running a loop in a loop and printing, choosing if we should print, we changed it so that we would loop through the whole image until, we found a point that was true and was not already printed. Then we would check if a point on the sides was also true and not already printed.

We did that by having two arrays of the image. The first array is a binary/boolean representation of the image. The second array is the "hasTaken" array. It tells if we were there before.

If we were to meet more than one point, we would load the others up on a linked list and then when we were done, we would go to the first one on the linked list. Run the same way through that point and so on.

As it runs through the points it adds to a string that is the g-code. The g-code then gets printed out in a file to debug and send through TCP.

Due to the program, however, we encountered some failures. If we were to draw a hand, it would not only fill out the fingers but also the space in between the fingers, because it was expanding from the center out to each finger on the linked list at the same time. This was a problem. To fix this problem, we put up some simple code. If the resolution was higher than 12 points, we would reposition the pencil while above the paper. Go up in z and down where the new point was. This worked, but it created some new problems. Instead of printing lines, the printer was instead making points all over the place. We were running with this solution until we did the first tests. Where it was apparent that this solution was too slow due to the z-axis. One of the images (Figure 3) 100mm x 56,25 mm took an hour to draw.

Java version 4

Since the x and y axis speeds are much faster than compared to the z-axis. We decided to find the shortest path around instead of lifting the z-axis, because lifting the z-axis or the

pencil takes a longer time. However, if it is not possible to find the shortest path, we need to lift it, find the nearest spot and go there instead.

A* Pathfinding

To find the shortest real path, we chose to use A* pathfinder.

A* pathfinder is a highly used way to find the shortest way. We learned some of it on a field trip to Universal Robots. So, it was natural to use. It works by finding the shortest direct path to the goal and trying to go that route. At the same time, its counting up, how many steps it took to go here and where it came from.

If we do this in a loop and always try to go the shortest way we can - at some point, we will end up at the goal. Here we can backtrack the route we took to get there, and we will thereby get the shortest route.

This can be described using an analogy. For example, take 1000 people and tell them to run a little in different directions to each other multiple times and see who gets there first.

However, this generated a lot of lines, so we had to find a way to remove some of it. The robot was drawing a lot on the same place on multiple times, because it was naturally expanding in both directions, so it would cross all the time. We did some sorting of the linked list to fix this. This would mean, in the analogy previously made, that we would stay on the same finger if we were printing a hand. We did some tests on this. These are described in “Java update to use A* pathfinding instead of Pythagoras to reduce errors in pictures being drawn” in evaluation.

How can the Java code send the file through TCP?

To send the g-code via TCP, we need both a server and a client. We set up the PLC as the server and the Java program as the client. Instead of sending the entire g-code program at once, we send it line by line. The Java program reads each line of the g-code file using a scanner and sends it to the PLC. After sending each line, the program waits for a response before sending the next line. This handshake mechanism ensures data integrity and prevents packet loss, which is crucial given the PLC's speed limitations.

PLC TCP server state description:

The PLC's Structured Text (ST) program is configured to receive g-code commands. It uses a state machine to manage different stages of TCP communication, shown in these following steps:

1. **Initialization:** The server socket is created, allowing the PLC to accept connections.
2. **Waiting for Connection:** The PLC waits for the Java client to connect.
3. **Receiving Data:** The PLC receives g-code lines from the Java client, storing a line in an array.
4. **Sending Acknowledgment:** After a line is received, the PLC sends an acknowledgment back to the Java client.
5. **Closing the Connection:** Once all data is received, the connection is closed to ensure resources are properly managed.

Java TCP client state description:

The Java TCP client is responsible for sending g-code lines to the PLC and waiting for acknowledgments. To allow for easier control, and allow the implementation of a graphic user interface, also known as GUI, the TCP client is separated into 3 methods.

The first method, connectToPLC, creates a connection between the PLC server and the client. An input and output stream are opened at the same time, to allow two-way communication.

The second method is used to send the previously made g-code, line for line to the PLC. Firstly the connectToPLC method is used to establish a connection, then a scanner reads through the g-code file, sending one line, then waiting for an OK to send the next. This was done to ensure that no lines were lost in the transfer. The TCP protocol already does this partially, with the 3-way handshake process. We added this in addition, because the PLC runs in cycles, and could potentially be in between 2 cycles, and therefore add up to multiple lines of g-code to its memory, which would lead to loss in the file transfer. Lastly the connection between the PLC and the Java code is severed.

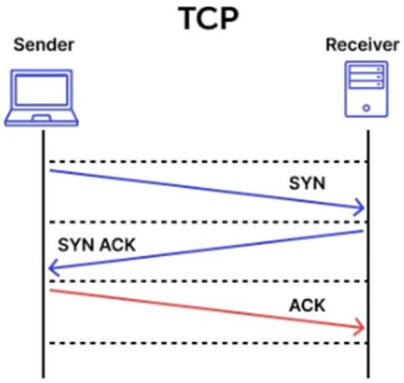


Figure 6 TCP 3-way handshake (Beschokov, 2024)

The third method, sendSingleCommand, is used to send single commands to the PLC. These commands are explained more in detail in the subproblem “How can we improve the user experience?”. In short, it establishes a command with connectToPLC, then sends a single worded command, and lastly close the connection again.

How can the robot load the g-code?

To load the g-code on the PLC we first receive the data via TCP, then wrote it to an array stored in the memory, to then loop through line for line. This way it was possible to disconnect the connection between PLC and PC after transfer. A different option we explored was utilizing a file on the PLC side as well as on the Java side. This would have allowed us to continue drawing from the same file, even after a power outage or similar disruption, as well as disconnect the PC after transfer. After a bit of trial, and talks with a representative from B&R, we decided against using the file for storing the data, both due to the difficulty of the libraries needed and upon recommendation of the aforementioned representative.

How can the robot process the g-code?

When the g-code is loaded up into the array, the address of the array gets passed around to the executer program. The program retrieves the data line by line when we need it. A counter keeps track of the moves and every time we have completed a move the state machine returns to IDLE and loads up a new line.

To read the instructions on the line we must split up the commands into its individual parts. We do this by first loading the whole command into a temporary variable. A loop is started until the temporary command string is empty. A backup method was set as a fail-safe that exits the loop, if the loop runs for more than 80 times (the size of the string maximum can be) it should exit.

Here, we check the first letter as our command. While the other different commands do different things and the rest move either to the next space, or if no space is found, the end of the line must be a number.

After we are done reading a command, it is removed from the temporary variable, and the process continues until the line is empty. When the line is empty, it then commands program to then exit the loop and the command get executed.

If there's a problem with the line, the Boolean used to activate it, gets disabled, making the drawing robot wait on the next line. This allows for troubleshooting and understand what went wrong. It does execute that line but stops the process on the next until the we manually reset the Boolean.

Another failsafe has also been added. Checking that some variable must have been changed and if not, flipping the same Boolean to false.

How can we move the robot?

Table 2: Overview of the wiring setup for the PLC, driver board and stepper motors.

Component	**PLC connection**	**Notes**
Driver Board for X-axis	Digital output - X3 - 1 (ENABLE)	Enable signal for X-axis
	Digital output - X3 - 9 (DIRECTION)	Direction control for X-axis
	Digital output - X3 - 10 (STEP)	Step signal for X-axis
Driver Board for Y-axis	Digital output - X3 - 2 (Enable - EN)	Enable signal for Y-axis
	Digital output - X3 - 11 (DIRECTION)	Direction control for Y-axis
	Digital output - X3 - 12 (STEP)	Step signal for Y-axis
Driver Board for Z-axis	Digital output - X3 - 3 (Enable - EN)	Enable signal for Z-axis
	Digital output - X3 - 4 (DIRECTION)	Direction control for Z-axis
	Digital output - X3 - 5 (STEP)	Step signal for Z-axis
Limit switch X+	Digital input - X1 - 3	Detects positive end of X-axis
Limit switch Y+	Digital input - X1 - 2	Detects positive end of Y-axis
Limit switch Z+	Digital input - X1 - 1	Detects positive end of Z-axis
Limit switch Z-, Pencil limit switch	Digital input - X1 - 2	Pencil position Detects negative end of Z-axis
Pencil Sharpener	Digital output - X3 - 8	

Z-axis

The movement of the robot in the z-axis happens in the mode “GCODE_RUN”. The PLC doesn’t have enough PWM signals to move the z-axis, so the z-axis isn’t able to move as fast

as the x- and y-axis. Therefore, the z-axis is running as fast as possible. To move the z-axis, the z-axis is first set to zero by moving the z-axis up in the positive direction until the upper limit switch is hit. Then, once the upper limit switch is hit, the z-axis position is set to 0, ensuring the z-axis doesn't miss any steps. When the z-axis moves down, the z-axis moves in the negative direction until the lower limit switch is hit, which sets the z-axis to 0.

While the robot is drawing, the z-axis doesn't move more than some millimeters, so that the z-axis, which is the slowest axis, slows down the drawing process the least amount possible.

X-axis and Y-axis

The movement of the robot in the x-axis and y-axis also happens in the mode "GCODE_RUN". To move the robot in the x-axis and y-axis we check if the limit switches of these two axes are activated and if the current x- and y-coordinates match the coordinates where the robot should move to. The structured text program gets the current coordinates come from the PLC and the new coordinates from the g-code.

However, the current coordinates of the robot are very big numbers, as they have 16 bits before and after the comma. So, we need to convert them into more useful numbers. We do this right at the start of "PROGRAM_CYCLIC" before entering the "GCODE_RUN" mode by using the shift right function. We shift the coordinates 16 bits to the right to get rid of the comma. We then subtract the new coordinates from the starting positions at the limit switches, which are also right shifted by 16 bits, and then multiply the result by -1. This ensures that we don't get coordinates outside of the limit switches and gives us coordinates that are similar to those from the g-code.

If the coordinates do not match and the limit switches are not activated, then it should move to the new coordinates. However, because the coordinate variables are REALs and not INTs, we cannot directly compare if they are identical. Instead, we compare if the difference between them is small enough. We made two new variables that describe the difference between the old and new coordinates, called "deltaX" and "deltaY" respectively. To move the robot to the new coordinates, the motors in the X-axis and the Y-axis need to be enabled and a velocity that is different from 0 needs to be set for each of the motors. The speeds can also be negative depending in which direction the robot is moving. Both can be done by setting up dedicated output variables at the PLC. However, it is important that the motors in the X-axis and the Y-axis reach the goal at the same time, so that the pencil takes the shortest route to its destination. This means, for example, if the distance in the X-axis is twice as long as in the Y-axis, then the motor at the X-axis needs to move at twice the speed of the motor at the Y-axis. To do this we defined an integer variable "speed" under program initiation that is set to the maximum speed. We could then make two new variables for the

velocities of the two axes, “speedX” and “speedY” by multiplying “speed” with “deltaX” and “deltaY” respectively. However, that would lead to velocities higher than the maximum velocity, therefore we need to divide these by the length of the direct path from the current to the new position. This, in fact, is the same as multiplying “speed” with the cosine of the angle between the X-axis and direct path in the case of the x-axis and with the sine of the same angle in case of the y-axis as shown in Figure 7.

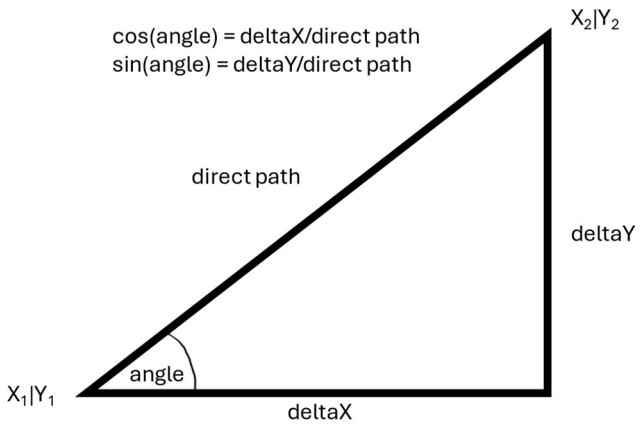


Figure 7 Sine and cosine calculations when moving from the current coordinates (X1 | Y1) to the new coordinates (X2 | Y2). Made in Microsoft Powerpoint.

This angle can be calculated by taking the arccosine of the division of “deltaX” with the length of the direct way. However, if “deltaY” is negative we need to subtract this from 2pi. This is because the arccosine has the same results for the positive and negative “deltaY”.

If the conditions of the initial if statement are not fulfilled (current and new coordinates are not matching and limit switches are not activated), the motors in the x-axis and y-axis get disabled. The program then checks if the z-axis has also reached its goal and if that is the case the robot switches into the “IDLE” mode.

How can we null all 3 axes?

If we need to null the axes we enter the mode “AXIS_ZERO”. We start first with the z-axis, then the x-axis and at last the y-axis. It is important to start with the z-axis first, so the robot does not draw on the paper while nulling the axes. However, it does not matter if the x-axis or y-axis gets nulled first, so it was chosen arbitrarily to null the x-axis before the y-axis. To start with z-axis, we first check if the limit switch z+ is not activated and then move the motor up in the z-axis to the limit switch, if it is not activated. This will then be repeated for the other two axes. Once all 3 limit switches are activated, we have successfully nulled all 3 axes, and we switch into the mode “FREE_AXIS”.

How can the robot move away from the limit switches?

After we have nulled all 3 axes the robot has switched to the mode “FREE_AXIS”. This state is used to move the pencil away from limit switches and to continue the program. First, we check if the limit switches of the x-axis or y-axis are activated. If one of them is activated the robot moves away from the switch.

If neither of them is activated both the x-axis and y-axis get disabled and the new coordinates get loaded. Then we check if the sharpening cycle is activated. If that is the case, the robot switches to the mode “CYCLE_SHARPENING”. If that is not the case the robot switches to the mode “IDLE”.

How can we sharpen the pencil?

As the drawing robot keeps drawing, the pencil starts to dull. To combat this problem, a pencil sharpener has been incorporated. In incorporating the sharpener, three problems must be solved: Turning on the sharpener, moving the pencil to the sharpener, and deciding when to sharpen the pencil.

The first problem to be solved is how to turn on the pencil sharpener. The pencil sharpener that is incorporated is a regular DC motor, which only needs access to electricity to run. That means there only needs to be a part of the code, which sets the pencil sharpener Boolean to TRUE.

Moving on to the problem about how to move the pencil: It starts with nulling all the axes, followed by moving the x-axis 150 steps out, so the pencil is centered over the pencil sharpener. The z-axis then moves down until the upper limit switch on the z-axis is hit, which then activates the pencil sharpener. The sharpener then runs for 20 seconds. After sharpening, the pencil moves up until the top limit switch has been hit, bringing the pencil back out of the sharpener. After this, the pencil is now free to continue drawing from where it left off before it sharpened the pencil.

The last remaining problem is when to sharpen the pencil. After a few test runs, the decision of when to sharpen the pencil was made to be in the start and end of each drawing, along with every 1000 millimeters drawn, to make sure the pencil doesn’t get too dull in the middle of drawing.

How can we improve the user experience?

To help the user better understand and use our program, a GUI was used. This both makes testing and actual use easier by removing the need-to-know commands. Even if the user

would get confused, a help menu can be toggled with a button, to show a description of how to use the program and what the buttons do. To implement the GUI, most of the java program is controlled by the GUI and not the main method. This made the program less autonomous but gives the user more control.

The GUI functions in 5 different steps. First the GUI is opened running the mainRuntime class without arguments. To use the GUI itself the user must first select an image to print. This is done graphically so the need to remember a name is not necessary. When the user has selected an image, the option to generate the g-code becomes available, and at the press of a button, it calls the needed methods to convert the image and store it in a file to later be read by the TCP client. Third the option to send the g-code to the PLC is unlocked, which establishes a connection, transfers the data, and closes the TCP connection again. Lastly the user has a few commands to choose from. To start the print, stop the print if it is running, start single line printing and clearing the array of data. Starting and stopping the printing is rather simple and does as advertised. Single line printing will toggle on and off a setting to print one line at a time. To print the next, the start button must be pressed. Clearing the array of g-code resets the data sent to the PLC and locks the use of commands, until g-code has been sent again.

System integration

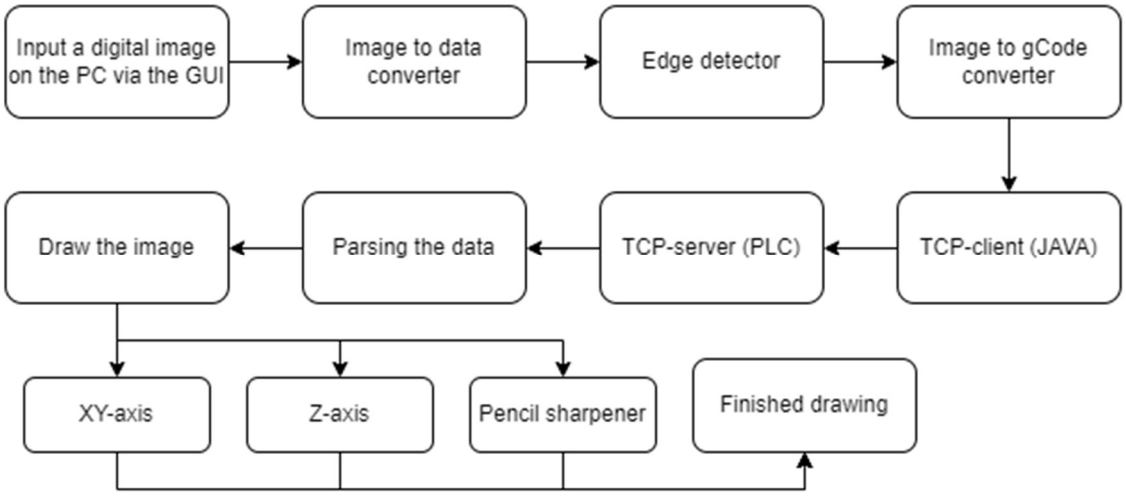


Figure 8 A diagram of the entire program and workflow

Most of our Java code is written utilizing methods and was thus very easy to fuse together. First our GUI is opened, where the user can choose their desired image, and control the

system through. When an image is selected in the GUI, the button to generate g-code calls most methods in our program.

First, we process the given image into data that the program can work with. This data is then grey scaled, and the edge detector finds outlines to be drawn. Afterwards the program will place dots on these lines, to then connect utilizing A*. This allows the program to determine the easiest/shortest path for the pencil to travel, while filling an area out. When the given area is filled completely, the program will look for the nearest unfilled area, to then repeat this process. All these commands are translated into g-code and stored in a file, both to allow checks for errors, and to store the data before being sent to the PLC.

When the program has finished translating the entire image into g-code, and no more dots are left, it calls the TCP client. First the connection is established, then it sends the whole file, line by line.

The PLC then receives the code, storing it in memory. To draw the image, the PLC first parses the data to figure out what command to run, then changes the given coordinates to the new. This in turn moves the pencil and allows drawing. The g-code can also call the pencil sharpener as a command, and it will loop through the array of commands until none are left. This results in the final drawing.

Evaluation

Java update to use A* pathfinding instead of Pythagoras to reduce errors in pictures being drawn

Because we had an issue with the robot drawing between points that were placed on two different peninsulas it would draw between the points directly. To stop this, we made a fix which was, if the points were placed more than 12 points between them, we would lift the pencil and go there. But this only created a lot of dots. Due to the robot drawer or printer's slow z-axis, this was a problem.

Another solution had to be made. We would use A* pathfinding to find the best way to the point, instead of using Pythagoras. This created some extra points because we had to move all around from one peninsula to another. Apart from that, we would sort the points on the linked list and always go to the nearest one.

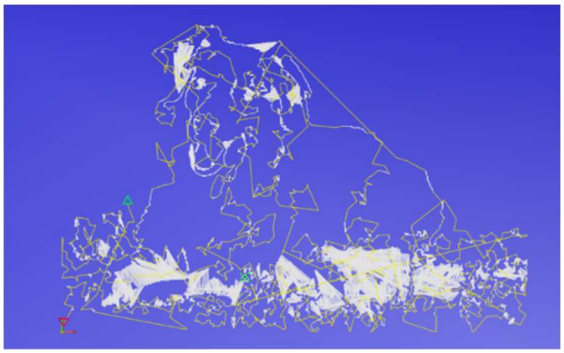


Figure 9 Simulationdrawing with pythagoras

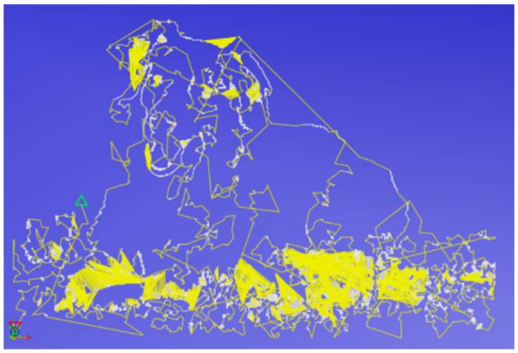


Figure 10 Simulationdrawing with lift of pencil if the points are placed with more than 12 pixels apart

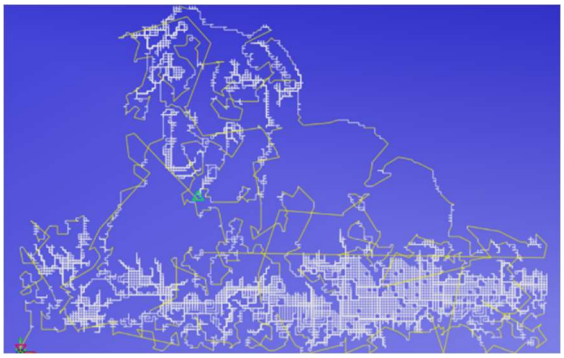


Figure 11 Simulationdrawing with pathfinding around peninsulas.

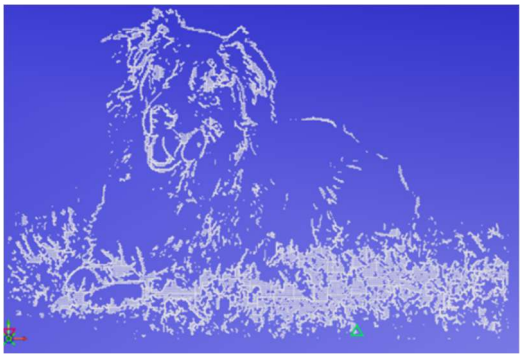


Figure 12 Simulationdrawing with pathfinding around peninsulas with a sorted linked list. The yellow has been disabled here

Testing the g-code and drawing a physical picture

We started testing the g-code and how it would look when we printed it. We tested all images drawing max 100mm in X and max 100 in Y. Scanned versions are on our attachment zip-folder.

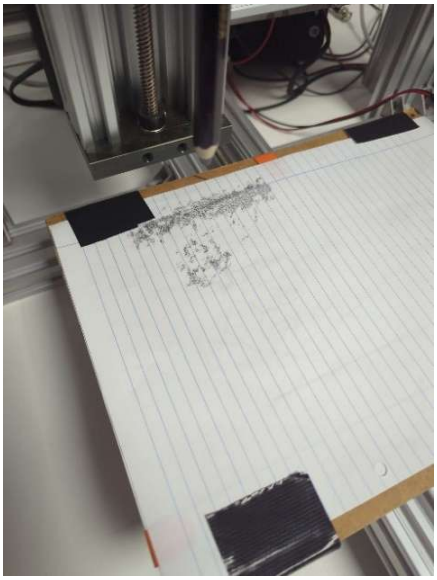


Figure 13 First physical drawing. The image is printed on its head which we fixed. - 1 hour print time

Figure 14 Test of precision of the printer and calculations we made. It was precise

Figure 15 We printed it again and the edge detectors lines have been made bolder - 40 min print time

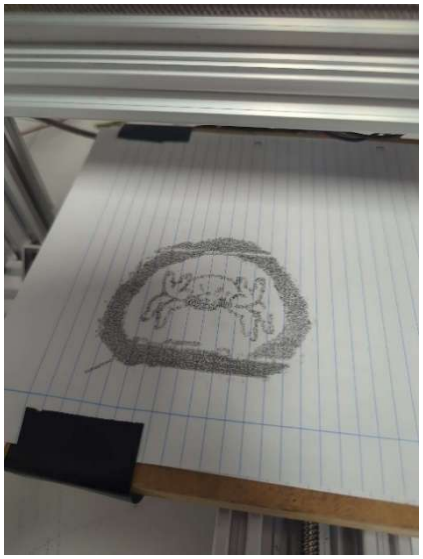


Figure 16 New drawing. We can see there is some drift of the axis when it runs. - 35 min print time

We can conclude that the printer is precise and fast, however it will lose some steps. Therefore, we have made it zero out its axis every time we have to sharpen the pencil.

Java update to reduce lines so we can send a better picture.

We tested if we could reduce the number of g-code lines we were sending through TCP to the drawing robot, so we tested what had changed in the simulated drawing. The change was that the points on a line were removed, so we only set a point when the angle was changing.

It reduced the lines from 49.245 lines on a 200mm picture resized in X, to 23.495 lines, without changing the picture.

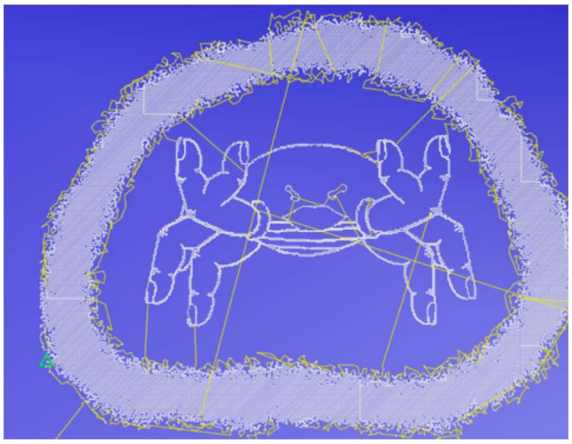


Figure 17 Simulation of an example drawing before the update

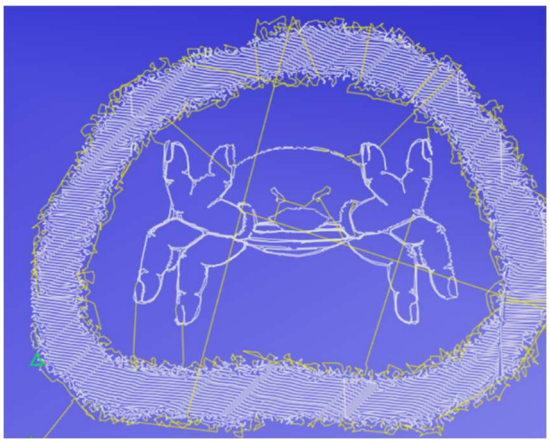


Figure 18 Simulation of an example drawing after the update

Drawing in Taxicab distance versus Chebyshev distance

To find the shortest route around a peninsula, you must decide on a method. We have tested the different methods to find the next point on our list:

Taxicab or the Manhattan method is when you go either up, down, left or right, like a rook in chess.

Another method is the Chebyshev method, where you can go like the taxicab method but also diagonally, like the queen and king in chess.

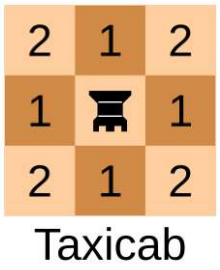


Figure 19 Demonstration of how many moves a given direction would take running like a rook. (.wikipedia, 2024)

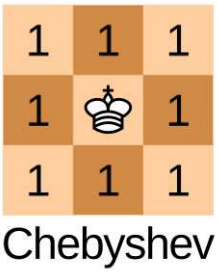


Figure 20 Demonstration of how many moves a given direction would take running like a queen. (.wikipedia, 2024)

Figure 21 Original image (Matusiak)

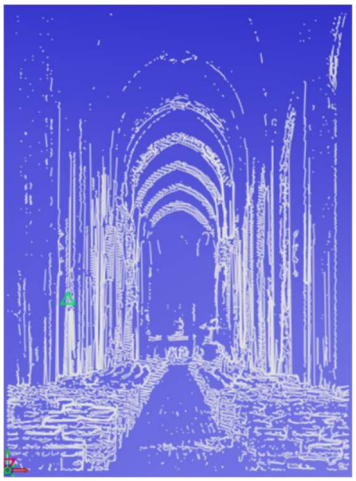


Figure 22 Simulation of Taxicab method

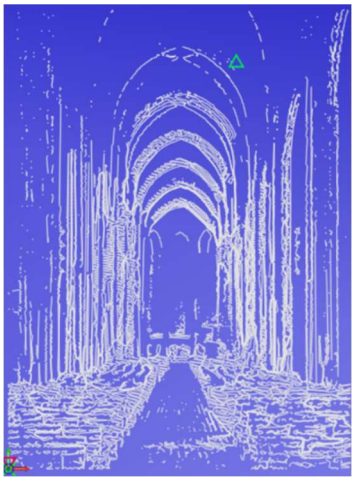


Figure 23 Simulation of Chebyshev method

The result is clear. Doing the Chebyshev method is a little more expensive in lines, because we cannot remove as many lines as before. However, it is even more detailed than before, and therefore we accepted the costs. The number of lines increased from 22462 lines to 25470.

Drawing a more detailed picture

Next, we tested if the robot was able to draw a more detailed picture. For this we chose a picture of the marble church in Copenhagen (Figure 24). Overall, this worked as intended as the robot was able to draw the picture (Figure 26). However, the lines were very thick with 1.90 mm thickness (Figure 27). Therefore, we concluded that we should sharpen the pencil more often which we addressed in the next test.

Figure 24 Original picture of the marble church in Copenhagen (FlyGRN)

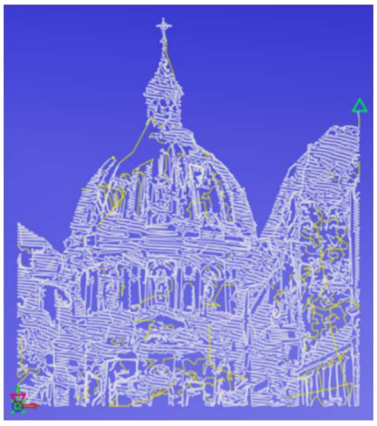


Figure 25 Output of the g-code

Figure 26 The physical drawing

When should the robot sharpen the pencil?

Figure 27 Measurement of the line thickness in the drawing of the marble

Originally, we sharpened the pencil after every 2500 mm. However, the pictures that the robot drew had very thick lines of 1.90 mm, making the pictures very blurry, see Figure 26 and Figure 27. To improve the quality of the drawings we need to sharpen the pencil more often. Ideally, we would like to have a maximum thickness of around 1 mm. Therefore, we tested how long the robot can draw before the line reaches a thickness of 1 mm. To do that, we set the robot up to draw 10 lines of 200 mm. In this way, we can see how thick the line has become after the robot drew 2000 mm. In this test, the thickness of the line increased from 0.66 mm at the start of the first line (Figure 28) to 1.36 mm at the end of the last line (Figure 29). This is an increase of 0.70 mm. We can therefore conclude that the thickness of the line increases by 0.35 mm every 1000 mm of drawing. To have a maximum thickness of around 1 mm we will therefore need to sharpen the pencil after 1000 mm of drawing. This gives us a maximum thickness of 0.66 mm + 0.35 mm = 1.01 mm, which is close enough to 1 mm. We then proceeded to draw the picture of the marble church again, but with sharpening the pencil already after drawing 1000 mm. This improved the quality of the drawing as it now has sharper lines (Figure 30) and therefore confirmed our measurements.

Figure 28 Measurement of the thickness at the start of the first line

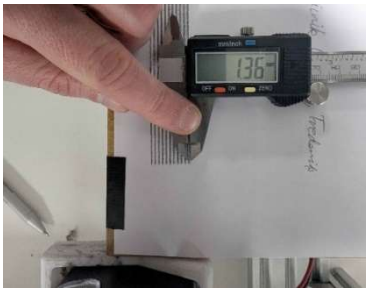


Figure 29 And at the end of the last line

Figure 30 Drawing of the marble church when sharpening the pencil every 1000 mm.

Discussion

Our main problem was “How can the 3-axis robot, which is controlled by a PLC, be used to draw an image?”. The answer we produced includes g-code, TCP and other choices we took, because it was the easiest path for us at the time. It is not necessarily the best path forward if we were to mass produce the product.

As part of answering the problem statement above, the following questions and tasks needed to be answered:

- How to process the image needed to be drawn by the 3-axis robot?
  We have not made the program very object oriented, as this was started before, we learned much about this. So, that means all the methods are static. This could possibly be made smarter in the future. We chose to use g-code as this is a common and easy way to read commands between the user and the machine. We would not change this as we have programs that can make this visible, even though we are losing some dataspace.

- How can the Java code send the file through TCP and how can the robot load the g-code?
  We chose to send it through as an array; however, we would have liked to send it as a file directly or as a string. But as a string on 2MB, it would be hard to part up the string. The function blocks to edit the text in Automation Studio were limited to 255 characters which would not be enough. We could make our own though. We tried sending the text as a file, and the PLC should be able to receive it. We were not so successful and chose to use an array instead.

- How can the robot process the g-code?
  We chose to part up the text by the spaces. However, the g-code does not always contain spaces. This means that the code is very specific to the machine, but it would always be so, because different machines handle g-code differently. You should not be able to enter your own g-code. This means that it is only our own specific g-code that the machine accepts.

- How can we move the robot?
  We use trigonometry to find the way to the next points. This is not necessarily the fastest nor the best way. We could find it from δX and δY directly. We did, however, use this method all around our program so it was the easiest way for us.

- When and how do we use the pencil sharpener in the drawing process?
  We found the number from tests. This was only a single test and with only one size pencil, which could mean that there is a high probability that others might get different results.

- Which/what other drawing methods could have been used?
  We started printing like a printer and continuously line by line. We chose a different path from this, when we got the edge detector. We were suddenly not printing on the whole image, but only to the edges, making it useless to run like a printer.

- What implementations can be improved in the system?
  We would like to free up a PWM channel on the PLC, making it possible to control the z-axis through a PWM channel. This could make it so that the robot can draw faster. However, as the PLC demanded two PWM channels to make position possible through the built-in system, we chose to go with the standard setup. This has made the z-axis very inefficient. It loses steps because it is not consistently in pace when running manual PWM.
  Another thing we could implement is a way to change the color of the pencil. This could possibly be done with a gripper, that can switch pencil from its current pencil to a new pencil. However, this does need upgrades on the hardware.

- How can we improve the user interface?
  We wanted to make the user experience more user-friendly, and with the interface the user does not need to remember commands, but simply press the buttons with a help menu if there are doubts. We feel like this solved the goal of improving the user experience. If we had more time, maybe we could have added some visual aid like pictures to see the progress. This also makes it easier to troubleshoot if there are any problems.

Conclusion

Our task was to program a PLC-controlled 3-axis cartesian robot that can draw a physical image by taking a digital image as an input. While there could still be improvements made to our program, we have successfully been able to make a program that can draw a variety of different images, including detailed images. Potential future improvements could for

example be making a robot that can draw using multiple colors or one that is taking less time to draw the image. However, these improvements require more complicated programs and hardware modifications. Given the fact that we have been given a limited timeframe to complete this project and we lack some of the knowledge necessary to implement these improvements, this lies outside the scope of this project. Nevertheless, this could be something to continue working on in the future and our project provides a good basis for this future work.

Bibliography

.wikipedia. (2024, November 26). *taxicab geometry*. Retrieved from Wikipedia.org: https://en.wikipedia.org/wiki/Taxicab_geometry

Beschokov, M. (2024, February 26). *transmission-control-protocol-tcp*. Retrieved from wallarm.com: https://www.wallarm.com/what/transmission-control-protocol-tcp

Dejan. (n.d.). *G-code Explained | List of Most Important G-code Commands*. Retrieved from howtomechatronic.com: https://howtomechatronics.com/tutorials/g-code-explained-list-of-most-important-g-code-commands/

dyrenesbutik.dk. (2022, August). Retrieved from https://blog.dyrenesbutik.dk/: https://blog.dyrenesbutik.dk/wp-content/uploads/2022/08/gigt-hos-hunde.jpg

FlyGRN. (n.d.). *Citytrip Copenhagen*. Retrieved from pinterest.com: https://www.pinterest.com/pin/646125877781273534/

geeksforgeeks. (2024, Juli 30). *What is TCP (Transmission Control Protocol)?* Retrieved from geeksforgeeks.org: https://www.geeksforgeeks.org/what-is-transmission-control-protocol-tcp/

Matusiak, R. (n.d.). *Grundtvigs Kirke: Nordisk kirkerum i særklasse*. Retrieved from pinterest.com: https://in.pinterest.com/pin/426927239679646380/