# Camera Equipped CNC Plotting Machine

As copy of a submission to

NAIT instructors

Submitted by

Michael Manning, Student

April 22, 2019

# Table of Contents

# List of Figures

## 1.0 Introduction

Tools with computer numerical control (CNC) are very versatile devices. While a plotting machine can't cut metal or create 3D objects, it is still different than a standard printer. A CNC plotter was chosen as this capstone project as it allows printing with any drawing or writing utensil on a variety of surfaces which are not limited to paper.

This report will discus the design of the robot and how it functions mechanically. It will explain how movement commands are created and eventually translated into physical motion of the robot. Next, it will dive into detail of how the path generating software works. While many CNC pa` th generating programs and libraries exist, this project sets out to generate all of the movement from the ground up, only relying on third party libraries to accomplish networking, computer vision, and optical character recognition tasks.

Beyond being a simple drawing machine, this project has served as versatile platform to create a variety of artistic and mathematical programs which take advantage of its visual nature. This report will cover the many programs that generate the art which the robot can create as well as the journey of mathematics and computer graphics that made everything possible.

# 2.0 Motion Control Overview

## 2.1 G-Code

CNC devices are usually controlled by stepper motors which not only require precise signals across multiple channels but may be one of many configurations which effect how the motor's angles determine the final coordinates of the tool head. In order to control CNC devices in an organized way, G-code has been used as a standard format for describing movement instructions. The controllers which drive CNC stepper motors have firmware which can be configured to interpret universal G-code and translate those instructions to the corresponding movements the machine is designed for.

There are dozens of universally supported G-codes as well as many firmware specific instructions, usually called M-codes for Miscellaneous function. G-codes might be in the format of "G1 X0 Y0 F100" for moving, while a firmware specific code might look like "M18" for disabling motors. Generally, in order to control the machine to complete jobs, thousands of G1 or G2/3 commands are sent in succession. These commands describe a linear movement and a clockwise or counter clockwise movement respectively. These simple instructions are the foundation of what makes the drawing machine operate.

## 2.2 Core-XY

The most common way of configuring motors to control an XY table is to assign one (or two synchronized motors) on each axis. This is usually referred to as a Cartesian system and usually involves a motor driving a belt which is directly connected to a moving axis or armature.

The drawing machine uses a different system referred to as Core-XY in which two motors are connected together in a single belt loop. Both the X and Y axes are controlled by turning both motors together in different combinations of direction and speed. For instance, to move only the primary axis, both motors move in synchronization. To only move the other axis, both motors must turn in opposite directions as visualized in the following image.

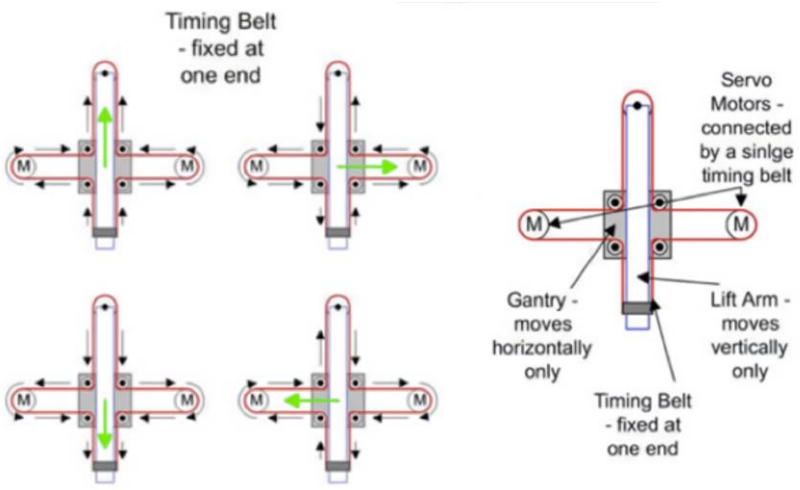



*Figure 1: CoreXY Motion Diagram*

(CAROLINEROSSING, 2014)

Besides the fact that the project is closely modeled after the AxiDraw V3 by Evil Mad Scientist Laboratories, this movement design was chosen because it allows control of both axes without having one of the motors mounted on a moving axis. Without Core-XY, the second motor would have to be attached to the moving gantry card of the primary axis which adds weight and complexity to the motion system. Instead, a four-pulley junction is used to rout tension across

both axes which keeps the moving sections of the machine lightweight allowing rapid changes in direction.

# 3.0 Project Design

## 3.1 Data Flow

For the machine to function, the linear movements generated by software need to be transferred to the motors at a speed of at least 115200 bits per second in order to draw highly detailed drawings at a high speed. To do this, the G-codes must be sent by the software over WIFI where they are received by the Raspberry Pi. The Raspberry Pi then forwards this data to Duet 3D printer controller over a serial where it is added to an internal buffer which is translated to motor movements. The Duet responds over serial with a status, and this status is sent back to the software over WIFI which confirms the command was processed successfully.
In addition to motor movements, the pen actuator as well as the camera must be operated over WIFI. The pen servo driver circuit is driven by a simple high or low 5 Volt signal. Since the Duet board is an older model that is unable to directly drive a servo, its firmware was modified to accept a command which can switch an unused pin to control the servo circuit.
For the camera, the Raspberry Pi program checks every message received over WIFI for special characters. When these characters are detected, instead of forwarding them to the Duet board, it takes an image with the connected camera and saves the file with a name specified by the WIFI command. The images taken are retrieved asynchronously over SFTP at which point they can be processed by the software.

## 3.2 Hardware

### 3.2.1 Mechanical Component

As stated in the motion control overview, the machine is closely modeled after the AxiDraw V3 by Evil Mad Scientist Laboratories and uses a similar core-XY movement system. Unlike the creators of the AxiDraw which manufactured custom aluminum extrusions for both axes, we were limited to off-the-shelf parts. For the primary axis, the aluminum extrusion, wheels, and gantry plate where purchased from OpenBuilds. For the secondary axis, the standard aluminum extrusions couldn't be used as they are both too heavy and so thick that pulleys wouldn't be able to fit underneath.

Many cheaper and more simple versions of this design including the AxiDraw V2 will use linear bearings instead of wheels to get around this problem as the linear rods are relatively small and can also provide structural integrity. Unfortunately, compared to wheels, linear bearings are less precise, noisy, and can introduce vibrations.

The secondary axis design for this project uses a thin aluminum edge piece from OpenBuilds called OpenRails. The primary axis uses V-Slot rails which the wheels fit inside of. OpenRails on the other hand, uses wheels which have an indent inside them which the thin OpenRails are meant to fit into. These rails are instead fastened to a standard V-Slot rail so that a gantry sliding on top won't accumulate saw dust and debris. Out machine uses this in reverse where the rails themselves are what move while the wheels stay stationary.

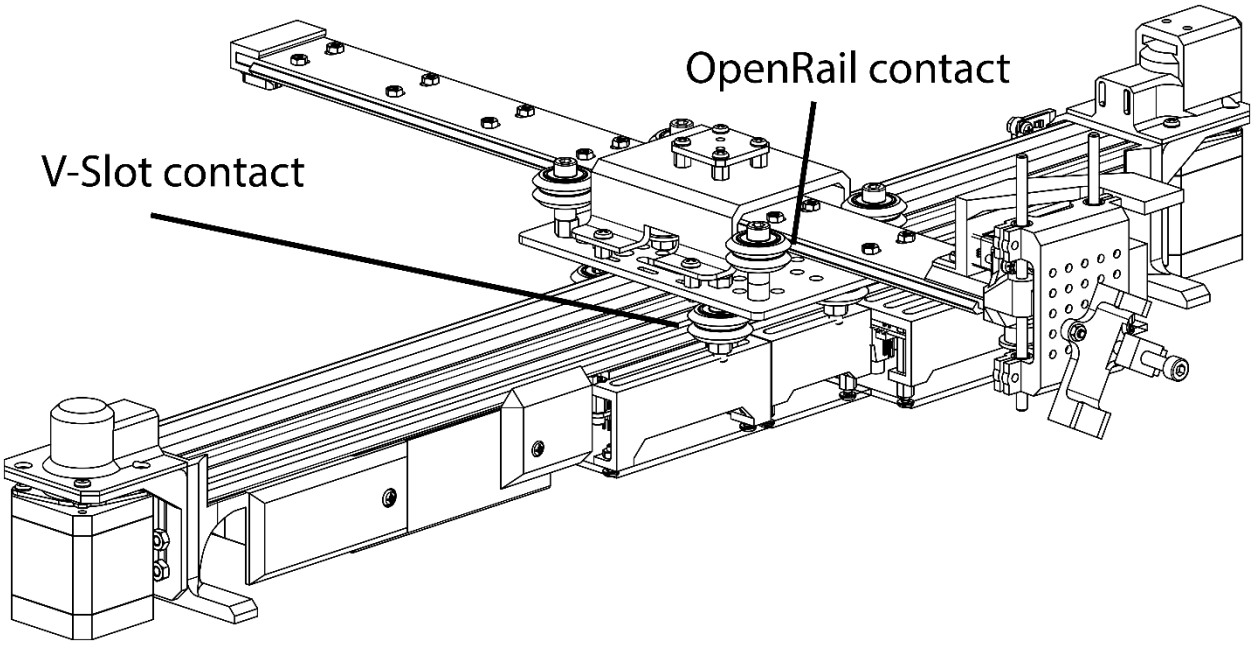



*Figure 2: Full Assembly*

Since the rails provide almost no structural integrity, a custom piece of 4mm, 7075 aluminum was cut to hold the OpenRails together. This solution worked well as it is strong enough to resist bending, but thin enough to fit the pulley junction underneath. Unfortunately, since this piece is an aluminum sheet instead of an extrusion, it tends to vibrate which reduces quality at high speeds when the secondary axis is fully extended.

### 3.2.2 Electrical Component

The NEMA17 stepper motors used to control the machine work most effectively at 24 Volts. However, The Duet board and Raspberry Pi run on 3.3-Volt power, and the servo runs off 5 Volts. Fortunately, by providing the Duet with USB power it internally steps down 5 volts to 3.3, and the Raspberry Pi also has this feature when powered through the GPIO pins. The machine has an internal 24 Volt power supply. This power is routed to the motor drivers on the Duet board. This rail is also connected to a custom circuit which contains an LM2596 step down

converter which provides 5-Volt power to the servo and Raspberry Pi. The Duet board is connected to the Raspberry Pi through USB which provides a connection for both power and serial through the same cable.

The servo is controlled by a PWM signal. This signal is generated by the custom circuit which has an ATtiny85 microcontroller. This simple microcontroller is programed to read from two trimpots which describe the settings for which angles should be targeted. When it reads a high or low signal from the duet board, it creates corresponding PWM signal and sends it to the servo.

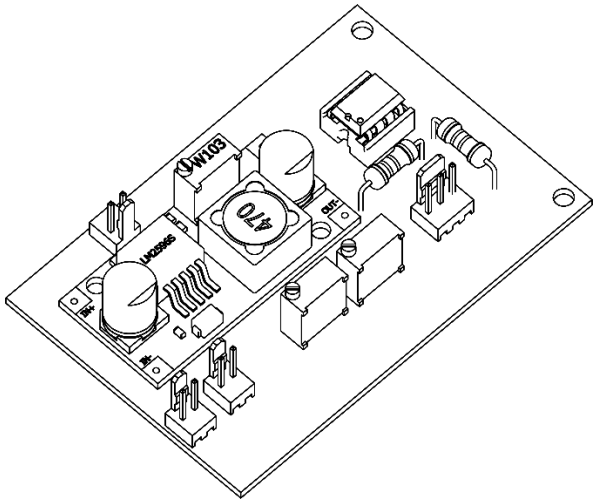


*Figure 3: Physical Circuit Layout*

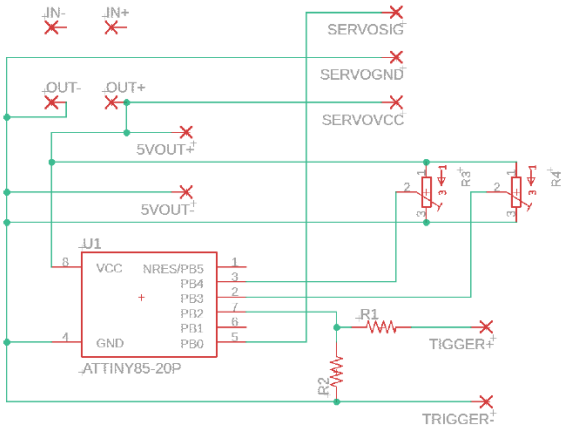



*Figure 4: Circuit Diagram*

## 3.3 Software

### 3.3.1 Desktop App

With the hardware and firmware set up, the machine can be controlled entirely by sending G-Codes over WIFI using the TCP protocol. The main desktop program called G-code Workbench has layers of abstraction to make this easier.

A curve structure was created which allows a universal description of a linear, arc, or Bezier movement which is used across the entire program. In addition to implicit mathematical and type conversions, there are functions which can convert a vector of these curves into a series of G-code commands. A vector of curve vectors describes loops. When these loops are passed into the G-code generator, each loop ensures that G-code is inserted before and after which lifts and lowers the pen between loops with an appropriate delay. This allows the programmer using the curve struct to create any sequence of paths without having to consider when to lift the pen or when to switch between drawing and traveling movement speeds.

A header for TCP communication was created using the boost library. Once connected to the machine, it can receive a series of G-codes which are added to a buffer and transferred to the Raspberry Pi automatically and asynchronously.

### 3.3.2 Mobile Application

The machine can be controlled independently of the desktop app. The phone app can communicate with Raspberry Pi just as the desktop app can. In addition to being able to use certain functionality from the C++ application through wrapping functions into a native library, there are a variety of programs that make use of the touchscreen.

The app has a drawing mode which allows the user to scale their screen size to represent scale ratio in millimeters. Then, using a finger or a stylus pen, an image can be drawn onto the screen while the machine replicates the lines drawn in Realtime.

# 4.0 Algorithms

## 4.1 Font Recreation

To create a program which allows machine to draw letters as a user types them, a line-based representation of the letters had to be generated. To accomplish this, a library is used which can extract Bezier information from any TrueType font file called STBTT. It can also find the vertical, horizontal, and kerning offset between letters which enables accurate letter formatting. Unfortunately, the resulting data provided by the library is in the format of a series of straight lines and quadratic Bezier curves. The only two movement commands the G-Code interpreter supports are effectively either a straight line or an arc segment. While the Beziers could be subdivided into a series of lines to create an approximation which could be represented with G-code, the algorithm goes a step further by approximating the Beziers as the closest arc segment with optional subdivision for additional accuracy. The advantages of this method over lines alone is a much higher data compression. A Bezier approximation comprised of lines might require many movement commands while an arc approximation could produce what is effectively the same result with one or two commands.


*Figure 5: Arial Text Path With Visualized Bezier Aproximation*

## 4.2 Vector Tracing / Loop Sorting

One way of recreating images with a series of lines is to draw the outline. This technique only works with black and white images without many greys in between such as silhouettes or clip art. To accomplish this, OpenCV is used to find the contours of an image which come in the form of a series of polygon loops. At this point the contours could be scaled to fit a drawing surface and directly converted to G-code.

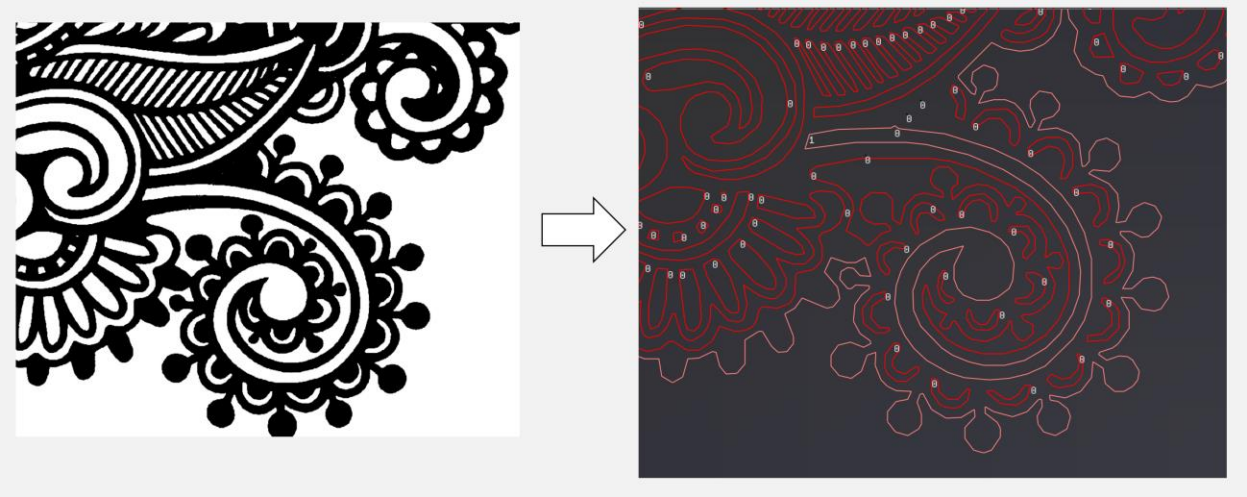


*Figure 6: Contour Example with OpenCV*

This didn't work well however as the final tool path was effectively random. With highly detailed images, this led to the machine jumping around the page, drawing all the details inefficiently.  This was orders of magnitude slower than an optimized draw path.

To sort the paths with 100% efficiently is almost impossible as it is similar to the travelling salesmen problem. While it is never perfect, the draw path can be optimized by sorting the loops from their initial positions. Starting at a random position, the following loops are ordered by choosing the next closest loop which has not been traversed. This method speeds up drawing times significantly, but still takes a long time to compute with detailed images as the number of distances that must be computed is equivalent to the square of the number of loops. This method is also impossible to compute in parallel with multiple CPU cores as sorting each line depends on the last line to be sorted. Most importantly, this leads to a memory bottleneck with frequent memory swapping to keep track of which loops have and have not been traversed.

To increase efficiency, a memory node structure was devised to represent all the loops. The nodes contain the memory location to up to 50 other nodes. In parallel, all CPU cores are used to find the 50 closest nodes for each node. This information is stored as data within the nodes alongside the exact distances, sorted from closest to furthest. Once all the distances have been computed, a single CPU core can navigate through the node structure. A final path is created by simply choosing the closest available node, very rarely having to re compute any distances with no need to swap memory in a list. Once this process is complete, the same path as the previous method is created, but in a small fraction of the processing time.

## 4.3 Rectilinear Infill

While many of the algorithms such as letter generation and vector tracing can provide outlines of shapes, another algorithm is required to create filled in areas. To fill in a square for example, the pen would have to traverse the entire inside area of the square to fill it in black. Given the width of the pen, a series of lines must be created to efficiently cover the entire area of a shape without crossing over itself or wasting too much time. To solve this problem, inspiration was drawn from how 3D printers fill in an area with plastic. The pattern used is called rectilinear. First, the bounding box of a polygon is found. Then, a series of lines, usually at 45 degrees are calculated which are even distance apart and trimmed to the size of the bounding box. The lines must be trimmed to only exist inside the polygon, so for every line segment, the algorithm checks for linear intersections with every single side of the polygon. Once the intersections are found, they are sorted by distance from the start of the line, and new segmented sections of line are created from every two intersection it had with the polygon. In this way, segments that appear outside of the polygon are discarded. At this point, if all these lines were randomly

traversed, it would in fact fill the shape correctly. This is extremely inefficient however, so efficient pathfinding was necessary. Unfortunately, with more than a few dozen line segments, the most brute force solving the most efficient solution is beyond the capability of any computer.

To solve this, a memory node structure was created. The nodes can contain data representing a single line segment as well as the memory location of up to 4 other nodes. Entry points to the memory structure are created at the top left end of every line and are filled with the line segment data that they are representing. The nodes are then linked together lengthwise which allows traversal along the original lines they were subdivided from. Then for each node/line segment, ray cast calculation is done tangent to their angle to find intersections with neighboring segments that overlap widthwise. Once these calculations are finished, the found neighbors are linked together in memory. This allows the algorithm to easily find all the sections of nodes which exist in isolation. That is, sections of lines where none are next to or overlapping more than one line on either side. These groups of nodes are put in a new memory structure of node networks.

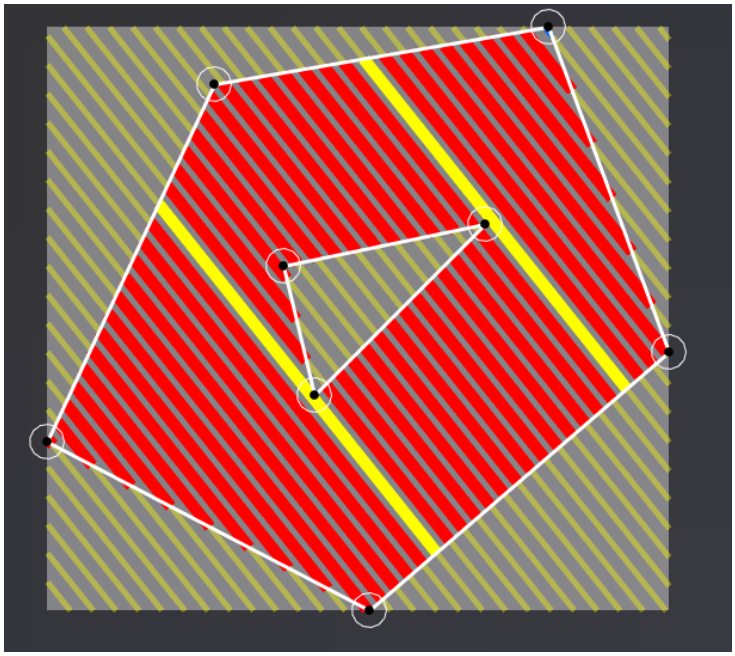


*Figure 7: Intermediate Infill Processing With Highlighted Fork Lines*

The infill sorting problem can now be simplified as a series of networks which have a start location and an ending location once traversed by a pen. In addition to the networks, there are also “forks” which are lines that are connected to multiple networks. These paths are then simply sorted with the same method used for sorting the tracing loops.

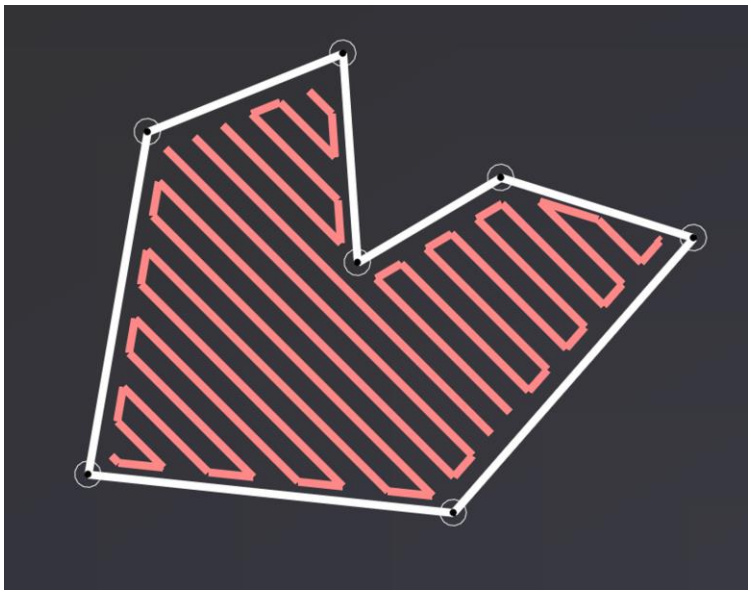


*Figure 8: Completed Infill Pattern With Linked Networks*

## 4.4 Wave Art and Spiral Art

To create drawn replications of photographs, a method of creating many horizontal lines or one spiraling line is employed. The darkness of the image is represented by modulating the height or radius of the lines with a sin wave. To accomplish the horizontal lines, a shader program was created which runs on the graphics processing unite. It can run millions of instances of a function simultaneously with varying data. The shader averages the darkness of all pixels in each horizontal strip that the final line will traverse and stores it in a separate block of data. With this data, a series of horizontal lines can be created using many small lines. The height of this line is calculated as the sin of x, where the amplitude of the line is a function of the average pixel darkness at that x value. To modulate the frequency, the derivative of varying darkness slope is simulated by accumulating all previous darkness values in memory and multiplying that into the f(x) function. To increase the speed of the preview, this process was later calculated entirely through a shader program. The average darkness and simulated derivative values are pre-calculated and stored into what is referred to as a texture in graphics programming. Then, a program runs on every single pixel of the resulting preview simultaneously. The shader function then determines whether each individual pixel resides on one the final sin wave and writes a black or white pixel to a separate block of memory.

*Figure 9: Line Art Source Image*

*Figure 10: Line Art Preview Generated by Shader Program*

To compute the spiraling version of this effect, using shader proved too complicated a method for remapping and averaging pixels values in a radial fashion. To compensate for this, the images are first blurred slightly to average pixel values in all directions which still allows for the result to look up to standards. A function was created to represent an even angle increment because of a given radius. This allows the algorithm to generate a spiraling series of lines which are approximately the same distance apart regardless of where they exist within the spiral. At this point, similar calculations to the horizontal method are used, but instead of modulating the height of the points, their distance from the center of the circle are modulated to create dark areas. To speed up rendering of the preview, all these lines are then transmitted as a single

block of memory to the GPU and rendered independently of the CPU through a technique called instancing.

*Figure 11: Spiral Art Source Image*

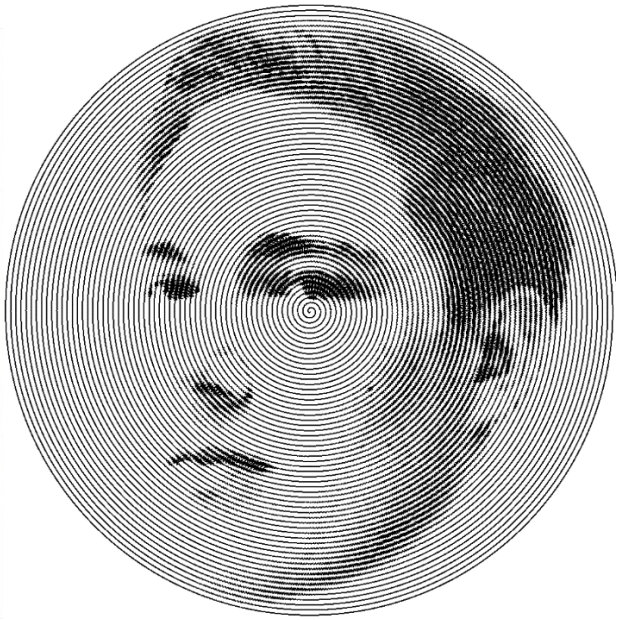


*Figure 12: Spiral Art Preview*

# 5.0 Camera Functionality

## 5.1 Purpose

The purpose of including a camera in the robot design was to allow the software to use data from what is on the drawing surface rather than just creating its own markings.

### 5.2 Hardware Component

**I**n addition to the pen actuator, the tool head also contains a camera. The camera is an OV5647 which has a 5MP sensor and a lens with adjustable focusing distance. If it were not for the camera, a Raspberry Pi would not have been necessary for the project to work, but a simpler micro controller would not have the processing power to transfer images at high speeds with

little cost savings. The camera kit that was used comes with two infrared LEDS. The nature of this setup reduces colour quality of images to almost nothing. The wavelength of the light also renders ink from certain pens invisible. The advantages are still great however as since the camera only picks up light from the IR LEDs, the brightness and general quality of the images stays consistent in any lighting conditions. The camera even functions normally in complete darkness to the human eye. The image consistency is important for doing computer vision tasks on the images as it allows more assumptions to made about the image when processing it.

When testing the camera with the one-meter long cable that was ordered, the camera functioned without any noticeable issues while waiting for a longer cable. A longer cable was needed as the it had to span the max distance of both axes as well as reach the Raspberry Pi which is slightly over one metre. Unfortunately, the next longest camera cable available was two metres. This would not be an issue due to the extra length alone as the excess was able to be contained underneath the machine, but the cable had issues with providing power to the camera and on the rare occasion when images were able to be taken they came back dark or distorted. It was postulated that the cable was having issues due to its length, and when measured, the wires had an average of 1.8 ohms. Seeing as the original cable that came with the camera had 0.2 ohms, it was determined that the extra resistance might be creating issues with providing enough power to the camera was well as lowering signal strength. To solve this, 40cm was cut from the ribbon cable. Then, the plastic insulation was scraped away on both cut ends with a knife. After practicing multiple times on some scrap cable, the cut ends were

spliced together with some fine soldering work. After this process was completed, the cable wires had an average of 1.4 ohms and the camera functionality returned to normal.

## 5.3 Software Component

### 5.3.1 Distortion Correction

To make use of the images that are taken by the robot, they need to cover a large area. Since the camera is positioned just a few centimeters away from the drawing surface, individual images are of little use without being stitched together. In addition, the short distance combined with the cheap sensor and lens both create distortions, most notably “fish eye” distortion which makes straight lines appear curved. This distortion makes stitching images together impossible as an overlapped section on one image could be wildly different in position and be warped compared to another. To solve this, OpenCV was employed to create a distortion matrix for the particular camera and lens being used. This involved taking dozens of images of a flat checkerboard pattern at a variety of different positions and angles. Then, the OpenCV functions were able to use these images to generate and save a simplified distortion matrix which is a profile of the errors that the camera produces in its images. This matrix can be loaded and then applied in reverse to any other images taken which significantly reduces the distortion once processed.


*Figure 13: Raw Image with Lens Distortion*


*Figure 14: Distortion Correction Applied to fg.13*

The process is not perfect however as, while the images now produce straight lines, they are slightly skewed. The images also must be slightly cropped as the stretching and compressing of the image leaves areas around the edges with empty data.

### 5.3.2 Image Stitching

To produce an image which resembles a large portion of the scannable surface, many images must be taken all over the area and then recombined in software. Images are first taken in an evenly spaced grid pattern around an area and distortion correction is applied. Originally, an attempt was made to use OpenCV to combine these images in the larger picture, but processing was slow, and consistency was poor. This is due to how the algorithm OpenCV uses relies on finding overlaps between images to determine where they exist relative to each other spatially. This works great for panoramic photographs, but not with this case. The images from the camera not only still have some distortion, but they may also contain multiple images of featureless white paper which makes it difficult for the algorithm to find overlapping patterns. On top of this, there are two bright areas on every image created by the IR LEDs which made a success very rare.

Fortunately, the algorithms in OpenCV were not needed to stitch the images. Unlike the previous algorithm which relied on overlapping patterns, the new algorithm could use a completely different set of data. Since the robot can position itself very precisely, the physical location where each image is taken can be recorded. This allows the software to closely approximate where to position each image. Simply placing all these images together based on the raw data however was not even close to accurately creating a combined image.

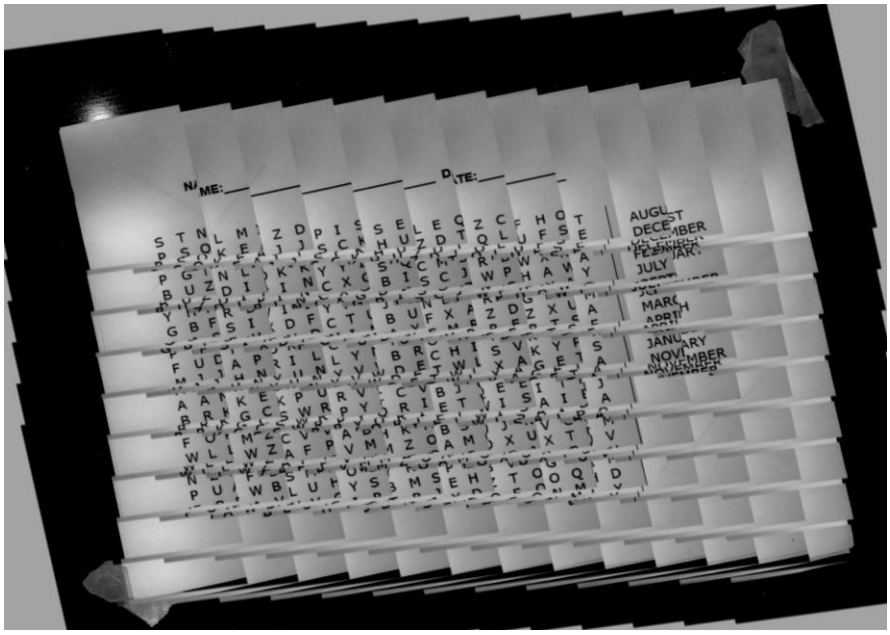


*Figure 15: Combined Scan Data from Raw Transformation Input*

This is due to the scale of pixels being different than the distance in millimeters each pixel spans. It is also impossible to perfectly angle the camera straight down or rotate exactly in line with the robot's axes. To solve this, a more in-depth program was made which has 13 individual parameters for calibration. These includes the spread of the images of both X and Y, positional offsets based on the image distance from the global center, and local transformations such as angle and skew. Once a set of images is taken, these parameters can be adjusted, the result of

which is displayed in real-time. The correction profile can be saved and tends to produce satisfactory results until the camera is physically touched or adjusted. Despite the images covering about 10 centimeters of the surface below the camera, the images are taken only 2 centimeters apart from each other. This slows down the process as to cover a standard piece of paper, 140 images must be taken. The reason this is done is that it dramatically increases the quality of the finally image.

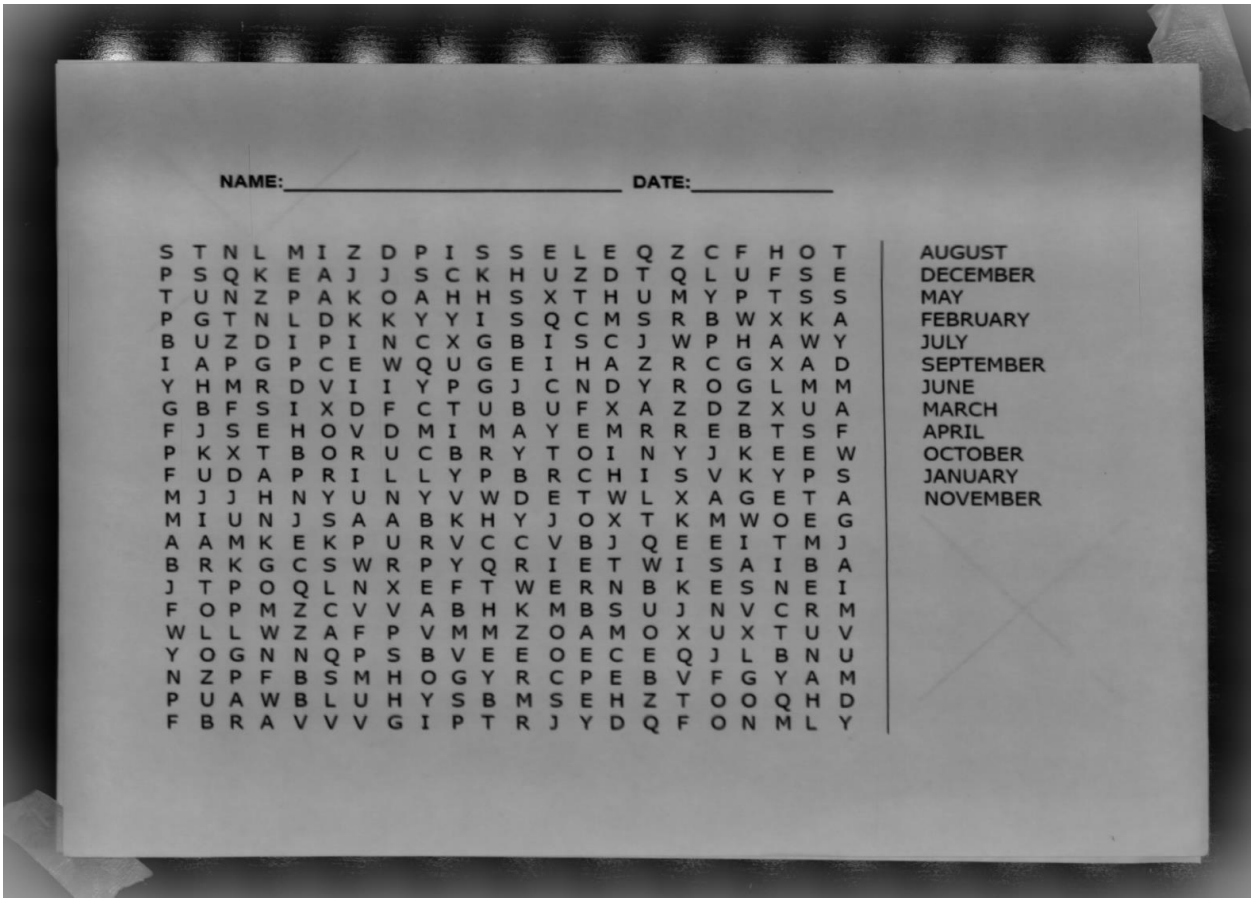



*Figure 16: Combined Scan Data with Calibrated Transformation and Blending*

The center of each image tends to be the truest to life while there is higher distortion closer to the edges (although distortion correction helps significantly). A center section of each image is preserved entirely in the combined result, while the pixel values of the in between areas are averaged together using many images that surround it. In this way, areas that were not part of

the center of any of the images are determined by many samples rather than one which helps filter out erroneous data. The final images produced using this technique produce a soft look on paper while often creating repeating artifacts on the reflective surrounding surface due to the LEDs. While the final "scanned paper" is not even close to what you might observe from a single photograph taken with a normal camera, it can be cropped and processed using binary thresholding techniques. The result of all of this is a grid of pixels which are either perfectly white where there was paper, or completely black where there was ink on the page. This process would be useless when scanning a photograph but represents an ideal situation for performing the required tasks for the project such as character recognition.

### 5.3.2 Physical Pixel Mapping

At this point, images can be taken to extract useable data about the surface under the robot. With this data alone however, no interactive functionality can be done. This is because stitching images together to create an image of the surface is a completely different problem than calculating the physical location of a pixel. Once an image of the surface is taken at a certain X and Y position, if you picked a particular pixel from that image such as the center of an observed letter, you do not have enough information to know how to move the tip of the pen to where that letter exists in reality. Consider an image is taken at the position 100, 300 in millimeters relative to the robot's minimum. The image would be 648 by 486 pixels. Given this information, if you looked at the image and wanted to start drawing on the physical location the pixel at 100, 400 was seeing, you would have to guess. Even choosing pixel at 0,0 on this image would be a positive number in millimeters. The exact position of the pen is offset from

software position of the machine, and the position of the camera is offset from the position of the pen. It's not even correct to say the position of the image is simply an offset from the pen, because it could only apply to the center pixel of an image as the image might cover an area of 10cm wide while being made up of 648 pixels wide.

This was first solved on the scale of a single image. First, a large "X" is drawn using two 50mm long lines. The center position of the x is known from the positional commands sent to the robot. At this point, the software machine position that the Duet board uses is disregarded and wherever the pen tip happens to contact the paper at a given location is considered the true current position at any given time. This is done because the machine position is already arbitrarily relative to the geometry of the machine, so it is simpler to consider the pen tip the actual position. The pen tip is usually very roughly 30mm away from the camera on both X and Y, so the tool head moves to this offset and takes a picture. OpenCV is then used for its implementation of the Hough lines algorithm which can find line-like geometry from pixels. This data is filtered into two lines in the geometric sense with beginning and end X and Y locations. The intersection of these lines is calculated which approximates the center location of the X in pixels as it appears in the image.

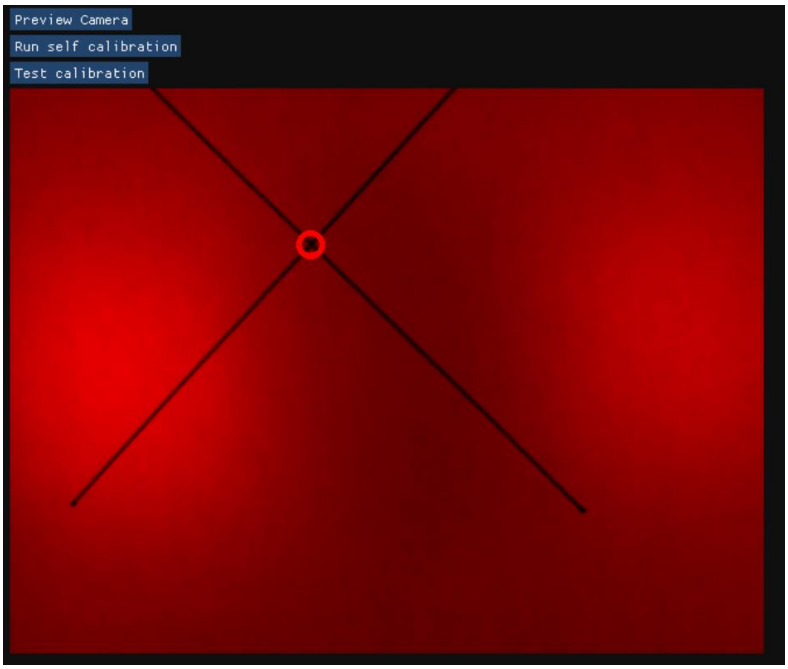



*Figure 17: Calibration Mark With Intersection Position Indicated*

The image is then cropped much closer to this location and the process is performed a second time which increases the accuracy further as the lines contain less error produced by the distortion around the image edges. At this point, the pixel to mm ratio has been manually calculated using a ruler and pixel distances. This value stays fairly constant unless the camera's height is physically moved.  With the drawn X position determined and the pixel to mm scale known, the camera's offset can be calculated. This is done by taking the initial offset (30mm both X and Y) and adding the offset of the detected X center from the center pixel of the image divided by the pixels per millimeter ratio. The calculation is also complicated by the face that the X and Y axes are inverted due to the camera's angle as well as how image pixels are treated as having a 0 Y value at the top and a maximum Y at the bottom, why the robot's Y axis is the

inverse of this. This process informs us of the camera to pen position offset as well as allows mapping of pixel locations to the required pen position on a single image.

Mapping pixels to pen positions is more complicated on a combined, stitched image which is unfortunate as it is where it is most useful. Theoretically, performing the previous calibration process once, gives enough information to map to a stitched image. This proved too difficult as the 13 different transformations performed on each image would have to be calculated in reverse. The difficulty is amplified when considering that in addition to transformation to the image's positions, those 13 steps include per-pixel alterations such as skew which is a non-uniform translation of pixels across the image. The skew process would also be combined with global and local offsets, rotations, etc. Though much slower, the mapping process can be solved in an easier way to execute. The following is a fixed process that only works on a standard sized piece of paper positioned in a specific location, although the settings used in the stitcher can be variable. This was done to vastly reduce the complexity and quantity of calculations due to time restraints.

First, two crosses like the one in the previous example are drawn near the corners of the sheet of paper. Then, the entire surface is scanned and stitched together.

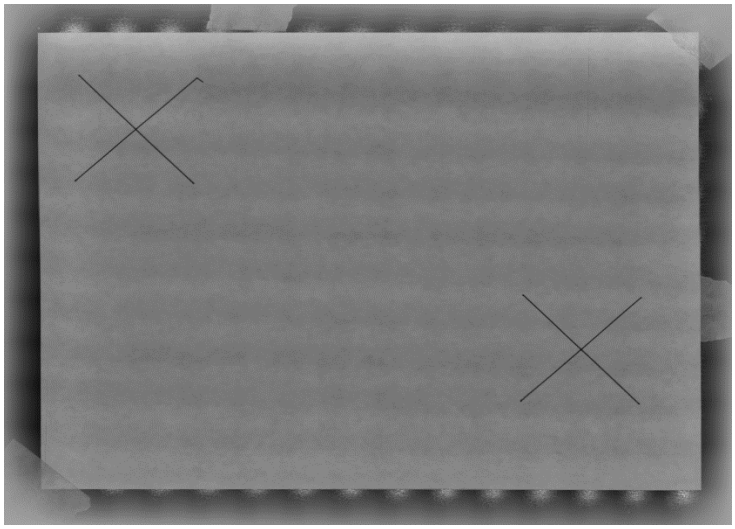


*Figure 18: Fully Scanned and Stitched Calibration Sheet*

The resulting image is cropped to remove the edges of the paper, and then cropped into two halves, each containing one of the X marks. The center of the marks is detected like before, but their pixel positions are relative to the uncropped image which contains both of them. These positions are first used to calculate pixel to mm ratios for the X and Y axes independently as the aspect ratio of the stitched images are never perfect. The offset of the combined image is found by treating the top right as the minimum location of the pen in mm. This means that all pixel locations should be treated as negative and must be dived by the pixel to mm ratio which is a vector of both an X and Y. This previous calculation is performed on the center of one of the X marks. This offset is flipped to a positive X Y value and can be saved. To map a pixel to where it should exist relative to the pen, it is simply flipped to be a negative location, scaled to millimeters, then translated by the pre-calculated offset.

### 5.3.3 Computer Vision Features

With paper sized images fully constructed and with the ability to map individual pixel location the equivalent physical pen location, new functionality is possible. Two features employed in this project are a recreation of a pre-made drawing and automatic word search solving

#### 5.3.3.1 Word Search Solver

To solve a word search automatically, the library Tesseract was used which can perform optical character recognition. Once Tesseract was configured to look for individual letters instead of words, it was simple to read the word list as well as the letters in the puzzle itself. First however, these sections of text had to be isolated from one another as well as rotated to ensure the text was level. The word search puzzles the program is designed to solve are from atozteacherstuff.com which has a long vertical line separating the search list and the puzzle. The program first uses OpenCV to detect the separator line which contains enough information

to crop the two areas of text as well as calculate a new rotation for the image.

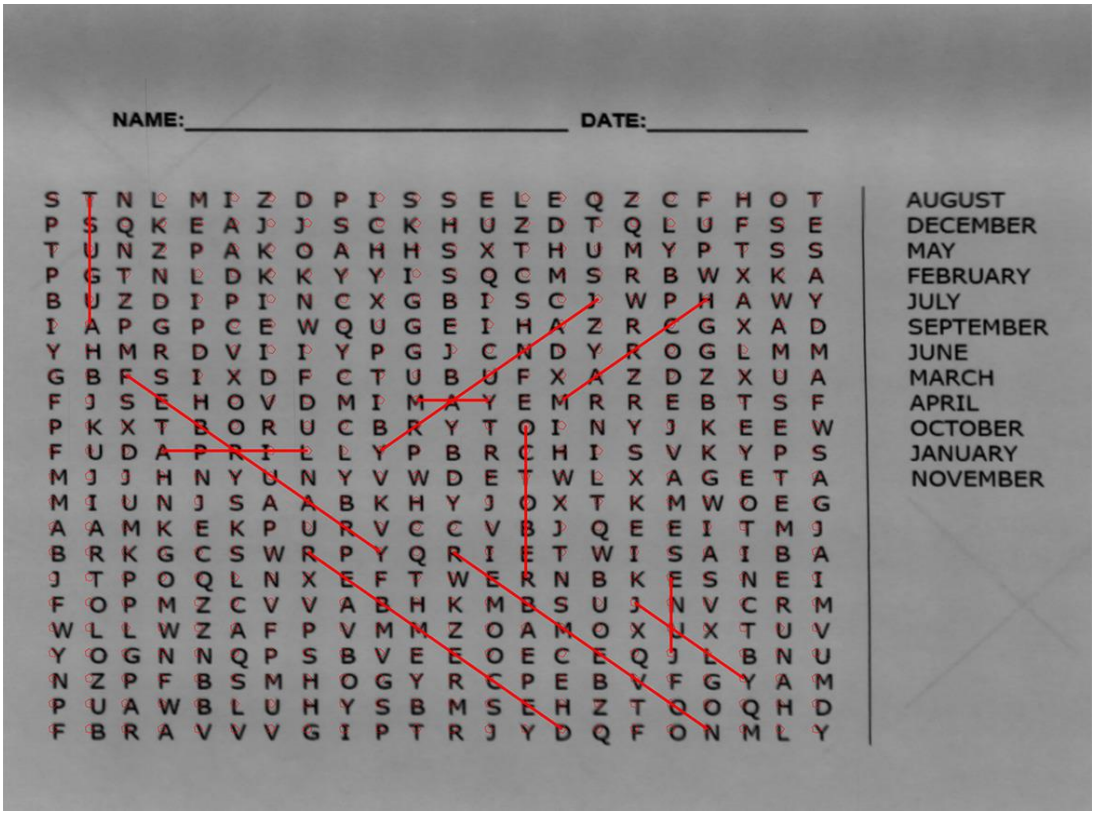



*Figure 19: A Scanned Word Search with 11/12 of the Solutions Detected*

## 5.3.3.2 Drawing Replicator

With all the algorithms discussed so far, it is a simple combination of them to get the robot to try and recreate a hand drawn image. In the example below, I have created a simple drawing with a sharpie. The robot then scanned the image and attempted to recreate what I drew as closely as possible using the

same Sharpie.

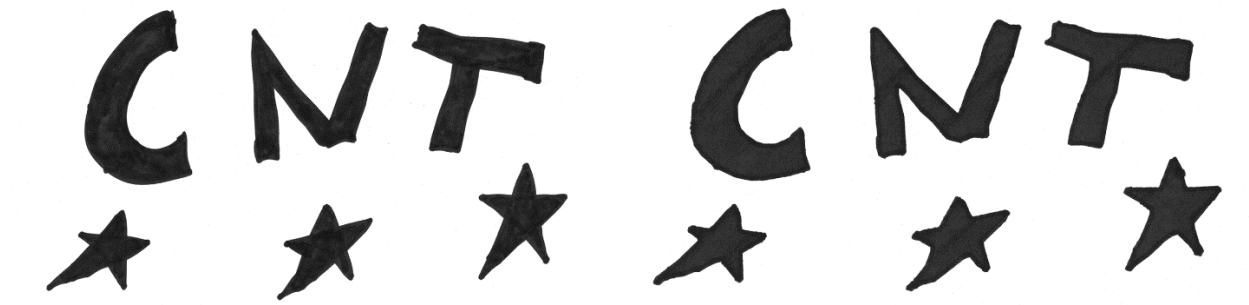
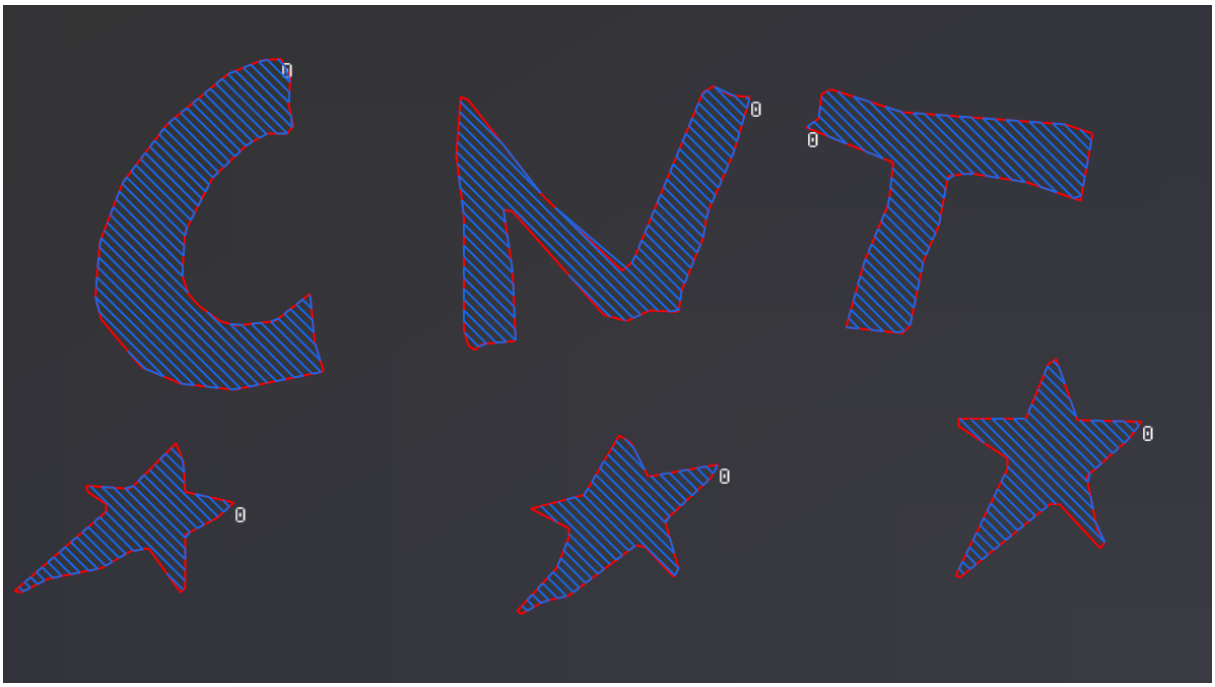


*Figure 20: Hand Drawn Image (left) Next to the Robot's Recreation*

This process is achieved by first scanning the image, then performing a binary threshold and a slight gaussian blur. The result can be directly fed into the image tracing algorithm which produces the following movement path.



The robot was able to convincingly recreate my drawing in a fraction of the amount of time it took me to fill in the letters.

## Conclusion

This project produced working CNC plotter with unique functionality related to the on-board camera. Once the robot was built and receiving commands correctly, a whole world of creative possibilities opened up. This report explored a variety of ways a simple series of lines and arcs can be created to represent something meaningful. The software in this project generates outlines and infill from images to create accurate renditions as well as artistic representations of photographs. The camera is able to combine images which accurately portray the surface below and can perform computer vision tasks which make use of this data.

Over the four months that this project went on, we went from a robot which could only draw simple shapes and letters, to solving a word search without any human assistance.

# References

Carolinerossing (2014). *Group 6 and Mechatronics: XY coordinates, 2 axis motion?.* Retrieved from https://mekatronikgruppe6.wordpress.com/2014/04/04/xy-coordinates-2-axis-motion/

Adrian Kaehler & Gary Bradski (2016) Learning OpenCV3 Computer Vision in C++ With the OpenCV Library. Retrieved from https://books.google.ca/books?hl=en&lr=&id=LPm3DQAAQBAJ&oi=fnd&pg=PP1&dq=opencv&ots=2vMoQidaz9&sig=naAFhHnLoGcWjuXK67xTJYz9HOE#v=onepage&q=opencv&f=false