SFU Rembrandt Robotics Competition 2017

Documentation

Team Drawesome

Michael Lam
Eric Liu
Thomas MacClements
John Nguyen
Joshua Shin
Caleb Taylor

1. Objective

Create a system to take picture, convert picture to drawing instructions, then output to a physical canvas using a robotic arm

2. Design

Our arm design went through a number of iterations. We initially opted to have steppers on each joint to articulate each length of the arm independently, but realized the torque of the "shoulder" stepper motor was not enough to support the lengths and weights of each joint after it. We iterated through platforms and rail designs for the base to handle drawing in the x-plane, but settled on a fixed, rotating base instead. This design supported the most weight and reduced the workload of our stepper motors. This base acted as a simple gear box, articulating 3-bar linkages allowing for smooth planar and spherical movement. The arm lengths themselves were 3D printed to allow for rapid prototyping. We chose fill percentages based on load bearing needs of each length and to reduce the weight of the overall mechanism.

Even our stepper motors were an afterthought. We initially wanted to go with servos, but realized the weight we needed to move, and the fidelity the drawings required could not be completed with servos without considerable configurations. Steppers were an obvious second choice, due to their angle encoding and high precision, high torque outputs.

The raspberry pi as a design choice was to allow for easy integration between the motors, image processing software and the UI. The pi is powerful enough to run all systems at once, and displays seamless UI elements and visual feedback to the user.

3. Hardware

Link to 3d printed robot is in "Resources"

Microcontroller

For our micro-controller,we are working with the Raspberry Pi 3 due to its strong processing capabilities for our image processing. Additionally, being able to program in python allowed us to create a simple user interface for the project.

Arm

In terms of our arm design, we used an open source arm from Thingiverse by a person with the user name Ftobler. In using their design, we re-modelled almost every piece of Ftobler's arm designs using Autodesk Inventor to match the lengths and dimensions we needed for this project. The whole arm was assembled with 3 NEMA 17 stepper motors, 2 stepper motor hats, and a desktop power supply. We are drawing on blank 8.5’ x 11’ piece of paper using a black sharpie attached to the end of the arm.

Camera

For our camera, we went with the 8MP Raspberry Pi Camera due to its high resolution relative to its cost and its compatibility with the raspberry pi.

4. Software

Overview

Our software is written in Python 2 for the Raspberry Pi. Please see “Resources” for the list of external libraries we used.

On a high level, the software is divided into the following modules: user interface, image processing, and arm control. We designed the software such that we can easily replace one or more modules without affecting the functionalities of the others. This lets us, for example, use a different type of robot while using the same image processing procedure.

4.1 Image Processing

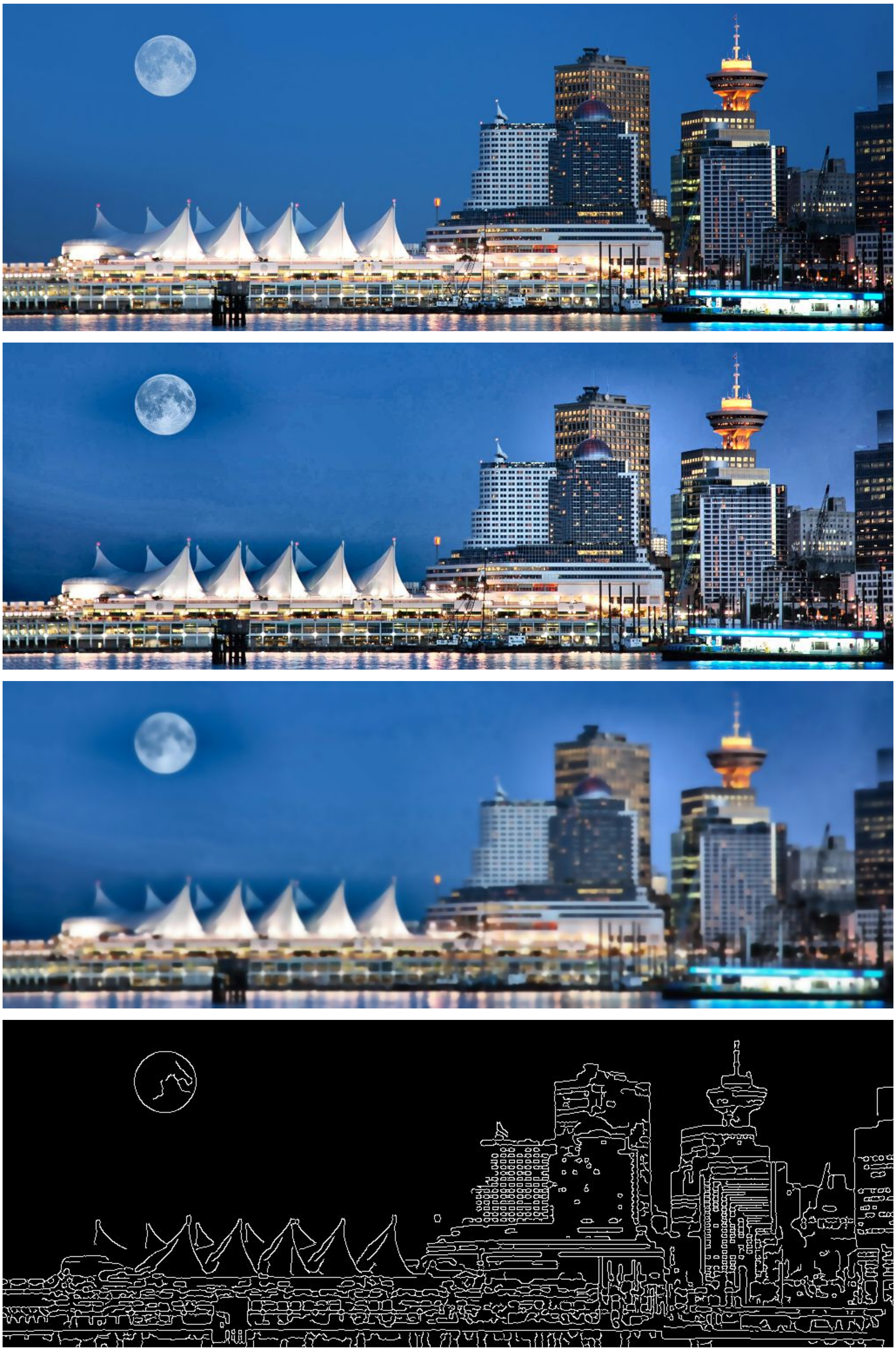
The job of the image processing module is to take an image as input, then output a list of **drawing instructions** that the robot arm can use. Each drawing instruction is defined as a list of x-y coordinates and a color. The robot arm would place the pen on the canvas at the first coordinate, move to the rest of the coordinates in order (like “connect-the-dots”), then lift the pen and be ready for the next instruction.

For this competition, we are not allowed to create a “printer-like” robot which outputs the image row-by-row. Instead, we draw the image in two stages. First, we trace the outlines of objects in the image, then fill in the colors by shading.

Edge Detection and Outline

When we first receive the image, we run the following preprocessing operations:

1. **Scale** the image to the desired dimensions while maintaining its aspect ratio. This helps us tune the parameters for other functions, since we would always be working on images with similar size.
2. Perform histogram equalization using the **CLAHE** method to make edges more apparent. We convert the image into the L*a*b* color space, then perform the equalization on only the L (lightness) channel. Then, convert the image back to RGB space (b).
    a. The procedure is described here: https://docs.opencv.org/3.1.0/d5/daf/tutorial_py_histogram_equalization.html
3. Blur the image using a **Gaussian filter** to remove most of the texture in the image. This smooths out the image, so the edge detection wouldn’t find too many small edges that result from rough textures. Then, use the **bilateral filter** to sharpen any edges that are still left over after the Gaussian blur (c).



Top to bottom: (a) source image; (b) after histogram equalization; (c) after blurring; (d) edge detection output.

To find the outlines of objects in an image, we need to run an **edge detection** algorithm. We use the Canny edge detection algorithm on the preprocessed image to get a binary image containing all the edges found (d).

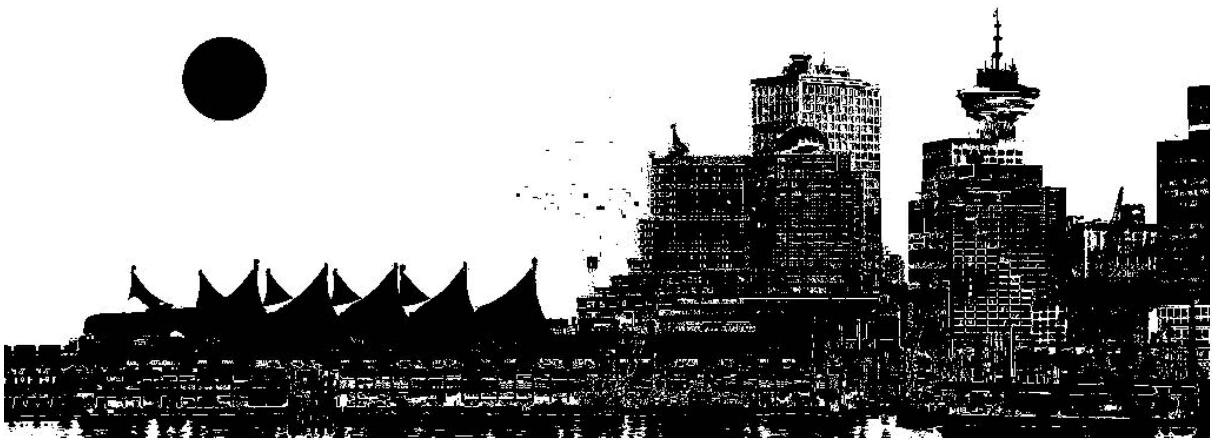
Next, we wrote a function called `detect_outline()`, which takes the image of edges as input, and returns a list of drawing instructions for those edges. It scans the edge image until it finds an edge pixel. The coordinates of this pixel is the beginning of a new drawing instruction, or line. Then, it continuously checks for adjacent edge pixels and adds them to the current line. Whenever an edge pixel is visited, erase it from the image so it does not get visited twice. When the next edge pixel no longer has any other adjacent edge pixels, the line is ended and is added to a list. Then, continue scanning for another edge pixel from the start of the line and stop once the entire image is scanned.

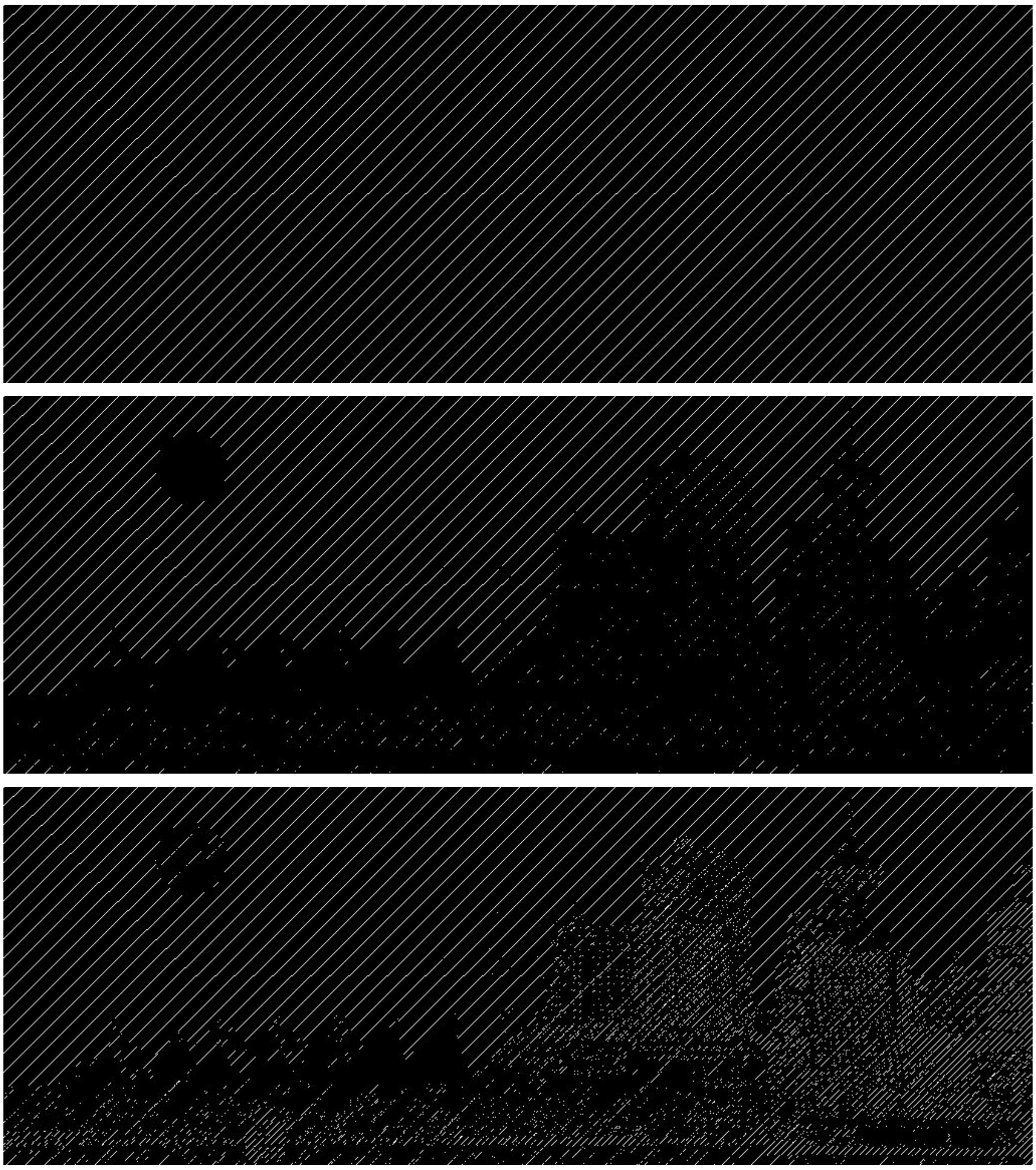
Black and White Shading

After detecting the edges in the image, we try to use shading to match the brightness. Our robot is limited in the number of colors it has access to, so we decided to use hatching of different sizes emulate different brightness levels.

First, we convert the image into grayscale. Then, we reduce it down to only 8 brightness levels by reducing the range of the color values from 0-255 to 0-7. We can then easily isolate each brightness level and use it as a mask for the hatching (a).

We are simply replacing regions of each brightness levels with hatchings of various sizes. We vary the spacing between the hatches to emulate the brightness of the image. Then, we perform bitwise-AND between each brightness level and corresponding hatching image. All the results are then combined together (d).





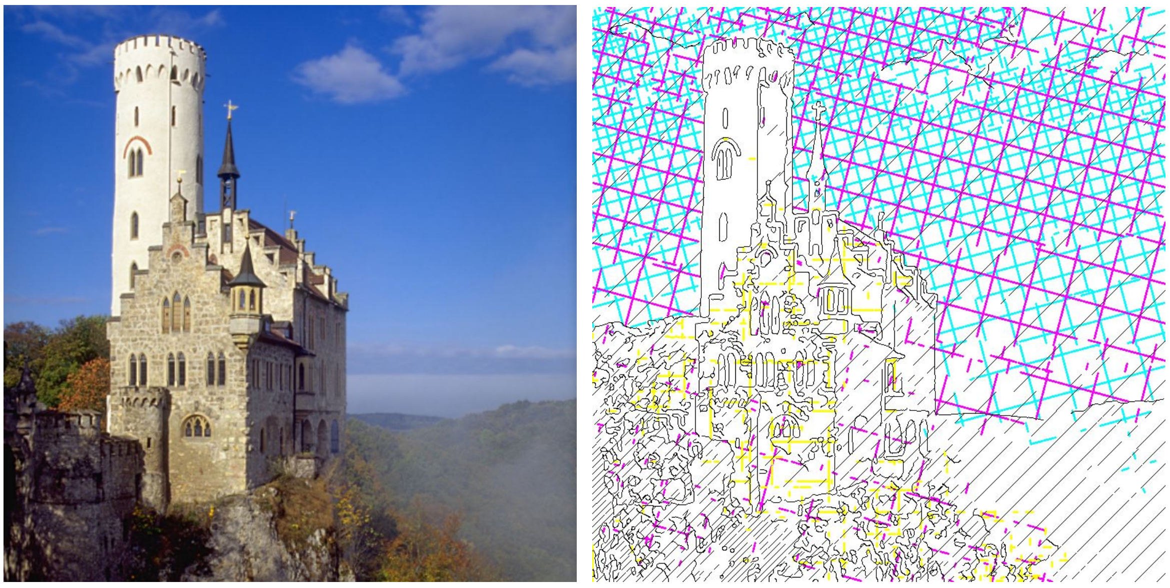
Top to bottom: (a) Isolated brightness level; (b) Hatching;
(c) Bitwise AND result of image (a) and (b).
(d) Combination of hatching at all brightness levels.

Then, we use the same `detect_outline()` described in the previous section to find the (x, y) coordinates of each hatching line and add them as drawing instructions.

Colored shading
Shading in color is very similar to shading in black in white. The only difference is, we first convert the image into **CMYK** (cyan, magenta, yellow, black) channels, and perform the shading/hatching operation individually for each channel. We chose these colors because they are the same colors used by almost all modern printers. However, our robot lacks the

accuracy and precision to output an image in such a way that seamlessly blends the colors together.

Another thing to note, is that the cross hatching for each channel is angled slightly different from each other. This allows the colors to cover as much of the canvas as possible and avoid being drawn on top of each other. Printers use the same technique to avoid Moire patterns. (see https://en.wikipedia.org/wiki/Screen_angle)



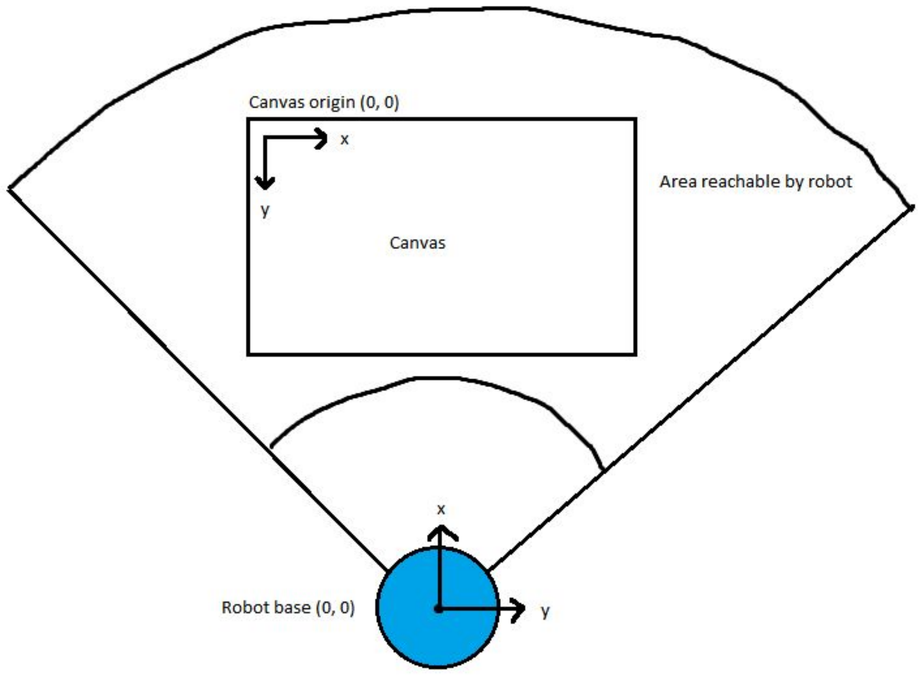
(a) Source; (b) Color hatching output

4.2 Arm Control

The image processing module calculates a list of lines in the form of x-y coordinates in the image. The arm controller translates the output from image processing into a format usable by the robot, then starts the physical drawing process.

Calculating Motor Angles
The first stage is to calculate a sequence of angles that the robot arm must move to in order to draw the image.This includes placing the pen on the canvas, moving the pen to the required coordinates, then lifting the pen. The robot uses 3 stepper motors. One at the base to control rotation (*base_angle*), and two to control the angles for the two joints (*j1_angle* and *j2_angle*).

First, we must transform the x-y coordinate system relative to the image into a coordinate system relative to the base of the robot arm. This coordinate transform is based on the physical dimensions of the robot, and transforms the coordinates from pixels into millimeters (mm). The transformed coordinate system is also in 3D, where *z* represents the elevation and *z*=0 at the base joint of the arm.



Top down view of robot base and canvas placement.

From the transformed canvas x-y coordinate, we can determine the *base_angle* required for the arm to reach it using simple trigonometry from a top-down view.

We can also calculate the horizontal distance *r* between the origin and (x,y) with Pythagoras Theorem. We have hardcoded z-values for how high the pen tip should be to draw on the canvas. With r and z values, we can use inverse kinematics to determine *j1_angle* and *j2_angle*. All the calculations for the angles are done in the function `point_to_angles()`.

Controlling Steppers
With the required angles for each movement, we can calculate how many steps and in what direction each stepper has to move.

4.3 User Interface

The focal point of the user interface were simplicity and aesthetics, with the added emphasis on user feedback and operator intervention.
To achieve this, we used Python, with the libraries Tkinter for basic layout and functionality and Tkinter-TKK for added aesthetics.

Simplicity
All UI elements were fitted into a single widow to promote simplicity, with the canvas (the main display for showing all currently relevant image) being the largest, in the center, taking up most of the window space.
Only the four most essential functions - LOAD (load image), CAPTURE (take photo), PROCESS(process image), DRAW (start drawing) - were added to the main window, right below the canvas.
Below the buttons, a status bar was added.

User Feedback
The status bar is used to give user feedbacks on the current status of the program. Whenever an action is made by the user, a message is displayed, letting the user know of what is currently in progress. Whenever a time intensive action is being made, (image processing or image drawing) a continuous feedback is given, to make sure the user understands the program is in progres.

Operator Intervention
When the drawing process is started, the user will be stuck in a lengthy drawing duration as the robot arm works. To give user flexibility, when the DRAW button is pressed, the four existing buttons (load, capture, process, draw, which are now irrelevant) are hidden, and two new buttons - PAUSE, ABORT - are displayed in place. PAUSE button can be used to temporarily pause the drawing process, and ABORT button can be used to kill the drawing process and revert back into the previous four buttons for a new set of actions. When the drawing is complete, the buttons also revert back to the original set.

5. Limitations

Arm movements

With the base determining x-axis translation, we ran into issues with the resolution of lines drawn further away from the base. The further away from the base, the more backlash and loss of resolution we experienced. A line drawn near the base of the arm is clean and detailed, but any movement while the arm was fully extended caused the arm to swing and jitter wildly. We found issues with the hyperextension of joints, causing calibration issues while drawing. Anytime the weight of the arm overpowered the torque of the stepper motors, we experienced stepper motor slippage and again a calibration issue. Some distances or coordinates near the edge of our drawing space caused mathematical errors in our software like impossible angles and undefined numbers, halting the drawing process. This, coupled with the errors in profile, pitch, and tooth thickness, created immense backlash with our gears when in motion.

Speed of painting

In order to have a high resolution drawing, and to support the weight of the arm, the steppers have to run at a slower speed/higher torque. This added a large portion of time to our drawing speed.

Color limitations, CMYK

Additive colour mixing is difficult to achieve with the materials we had access to. CMYK requires precise ratios and blends to achieve believable colours, something we simply could not physically achieve. Our software does write coordinates to allow for said colouring, but also adds significant amounts of time to the painting process.

Processing speed on the Pi

We noticed that, when running all software, the Pi’s CPU load reaches a high percentage. The program will occasionally hang or freeze, or simply take too long to process images.

6.Finance

Part	**Cost**	**Units**	**Total**
Stepper Motors	25	3	75
Raspberry Pi 3	40	1	40
Raspberry Pi Camera	30	1	30
Stepper Motor Driver Hat	25	2	50

Bearings, Screws, Nuts, Ball Bearings	40	1	40
Wires and Solder	10	1	10
Poster boards & Art Supplies	10	1	10
Total			255
Final Total			255

7. Conclusion

Our team has met almost every week for the duration of this competition. We've been kicked out of rooms, soldered without a table, stayed overnight in a cold classroom and other adventures along the way. We've yelled at inanimate objects for not working, been burnt, cut, sleep deprived, starved and stranded without transit. And as a team of non-engineers, we've spent countless hours learning electrical basics, burnt motors, corrupted hard drives and cut wires that we weren't supposed to. While making a robot is none of our strong suits, we've put a ton of time and effort into this project that we're all extremely proud of.

We learned a lot

Over the last months we've learned a lot. Everyone in the group had a chance to solder, code, and assemble the arm. We've all stepped out of our comfort zones numerous times to get things done, and surprised ourselves with our abilities to adapt and fulfil responsibilities.

Things to do differently

If we were to do this again, we would definitely spend more time on the initial research. Weeks of work were wasted more than once after we've tardily discovered a limitation with the direction we were going in.

How we would improve this design

To improve this design, we would work on its current limitations and try to find a workaround. We would use stronger stepper motors, design more correct gears, make the joints have less friction.

8. Resources

Open source 3D printable robot arm design, from which we modified for this project:
https://www.thingiverse.com/thing:1718984

Github link to our image processing and robot control software:
https://github.com/CobaltBlack/project-drawesome/

External software libraries that we used in our software implementation:

Library	**Purpose**	**Link**
OpenCV 3.2.0	Open source computer vision library, with many useful image processing functions	https://opencv-python-tutroals.readthedocs.io/en/latest/
TkInter	Simple library for building a graphical user interface	https://wiki.python.org/moin/TkInter
Adafruit Python Library	Library to interface with the Adafruit Motor HAT, which controls the stepper motors	https://github.com/adafruit/Adafruit-Motor-HAT-Python-Library