WPI

Worcester Polytechnic Institute
Robotics Engineering Program

RBE 4815 Final Project:

Robotic Drawing of Pointillism Style Images

Submitted By:

Hannah Baez

Nikolas Gamarra

Ryan O'Brien

Jacob Remz

Date Submitted : 12/15/18

Course Instructor: Professor Putnam

Lab Section: RBE 4815 B18 Team 6

Executive Summary

We set out with the intent of drawing pointillism style images using an ABB IRB 1600 industrial robotic arm controlled with ROS in order to familiarize ourselves with the arm. This was done with a ROS action server that sent poses and cartesian motion plans to reach each point drawn and handled rotations between markers. The markers themselves were spring loaded and held in a microscope-style tooling to allow them to be depressed enough to draw clearly. We were able to consistently and accurately draw images imported in Python provided they consisted of a predetermined set of colors. In order to draw a wider range of images, we created a dithering script that could generate PNG files using only the set of user specified colors. While simple images could be drawn easily, large scale dithered pictures required several hours and so we only produced a partially drawn image.

Table of Contents

Introduction

This project was an exercise in using an industrial robotic arm to perform the complex task of drawing an image in the style of pointillism. Custom tooling was made to hold three dry erase markers to draw pre-processed images. Accomplishing this task required numerous frame transformations between the markers and precise motion to avoid damaging the tooling. The real-time logic needed to perform this task required the use of ROS to control the arm. Using ROS allowed us to write our control software in Python and easily process images.

Goal Statement

The goal of this project was to draw pointillism style drawings using an ABB IRB 1600 1.45/6 industrial robotic arm. These drawings can be drawn using using any three colors mounted in our end of arm tooling (EOAT) plus white (the background color). Additional colors can be added in multiples of three requiring a tool change and more cycles. The size and resolution of the drawing is also scalable.

Task Specifications

In order to achieve our goal, we broke down each task into discrete steps to determine what was required to meet them. First and foremost, the arm has to be able to linearly depress a marker onto a whiteboard without damaging the marker through excessive force. Both mechanical and software precautions were taken to prevent this from happening. The EOAT was designed such that the markers were springloaded, allowing for enough force to draw but not enough force to damage the felt tips within the range of motion of the spring. Furthermore, the tooling was made to quickly and accurately measure forces on the three tool frames and be able to arrest the arms motion if large forces were detected. The markers and their respective coordinate frames need to be known, to ensure that points drawn in different colors are not misaligned.

The second defined task is to change between tool frames while still drawing accurate points. Each marker can be oriented along their major axis differently within the tooling, so

consistent poses need to be established. To ensure each marker draws points in the same relative location, offsets need to be added to two markers to bring each frame in line. Without the calibration process to establish consistent offsets for each marker, points drawn with different colors may be misaligned resulting in the image being unrecognizable.

Workspace Setup

Our workspace consisted of the ABB robot and a whiteboard. The whiteboard we used was 34x48 inches and covered most of the available level workspace. This provided us with a wide range of possible drawing points. The whiteboard rests on the wooden platform that the ABB is mounted to such that in the robots base frame the *z* coordinate is very close to zero. We placed the board on the table with the center approximately over the negative y-axis of the base frame of the robot as this was the largest flattest area. When testing we discovered that points that were further away from the *y*, *z* plane and approximately 2/3rd of the work cell reach away from the robot worked best as they avoided any potential singularities.

EOAT Design

A custom tooling was designed and created to be used for this project. However, a force sensing mounting plate designed by the AVRS MQP (which shares members with our team) was reused here. The plate has three load cells connected to an Arduino Mega with load cell amplifiers. The upper range of sensing is far more than what is needed for this project, but is still sufficient for this application. A pneumatic tool change with 15 pin I/O throughput is mounted on the tooling end of the force sensing package. We designed our end of arm tooling (EOAT) to be mounted to the tooling plate of the tool change used in the force sensing package. Although we did not utilize the tool change as part of this project, further expansion of our projects functionality by using the tool change is a possibility.

Shown below in Figure 1, three markers can be inserted into the slots and will be held at 25 degree angles from the center. The consistent positioning of the markers in the slots allows for a calibration sequence to be used to calibrate the known tool transformations between marker tool frames resulting in the same relative *x* and *y* location, which is absolutely necessary to draw

with any accuracy. Otherwise, attempting to draw a point in the same location would result in a different point for each color. In order to draw a consistently sized point, the markers need to be pressed with enough force. When markers are inserted and the tool is mounted to the plate there is a small 0.35 inch gap between the back of the marker, as shown in the cross section in Figure 2. Springs were placed to keep the markers pushed forward but also allow them to be depressed safely. We varied the strength of these springs and tested the forces required to depress them to find a safe amount of compression.

We tested three spring strengths before finding the most ideal one. The first springs used took upwards of 3 lbs to compress any noticeable distance, at which point some of the markers gave out and the felt tip was pushed into the marker body. The second was significantly lighter, requiring only half a pound to fully compress the spring. This did not draw enough without pushing the markers down further. Third we tried combining the second set with other similar strength springs. With this configuration, the markers can be fully compressed with around 1.5 lbs of force, but can draw clear points while only partially compressed.

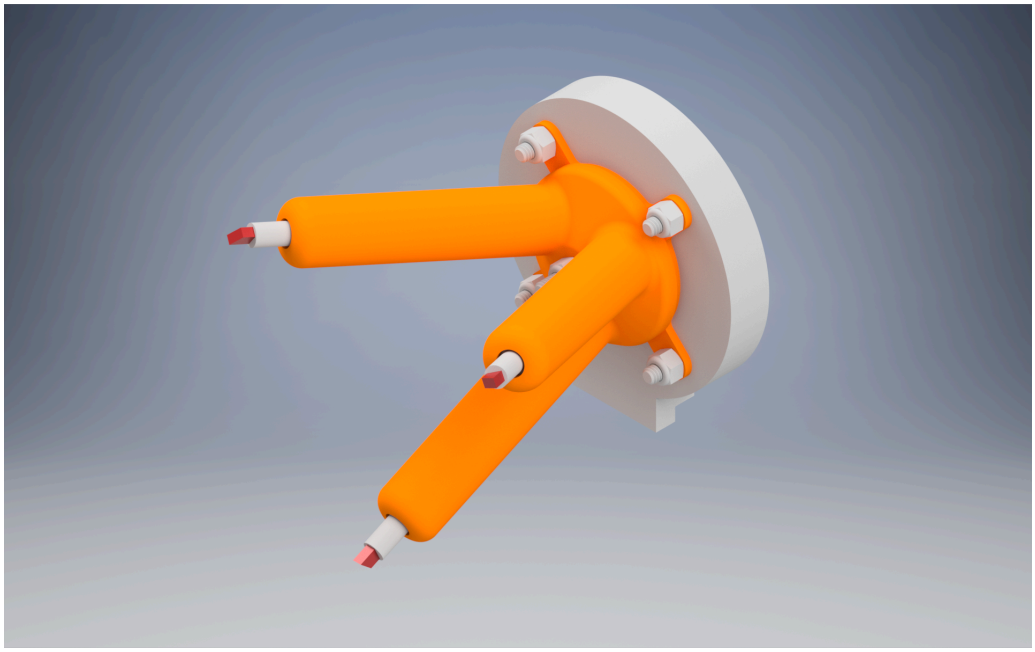

Figure 1: Tool with Markers

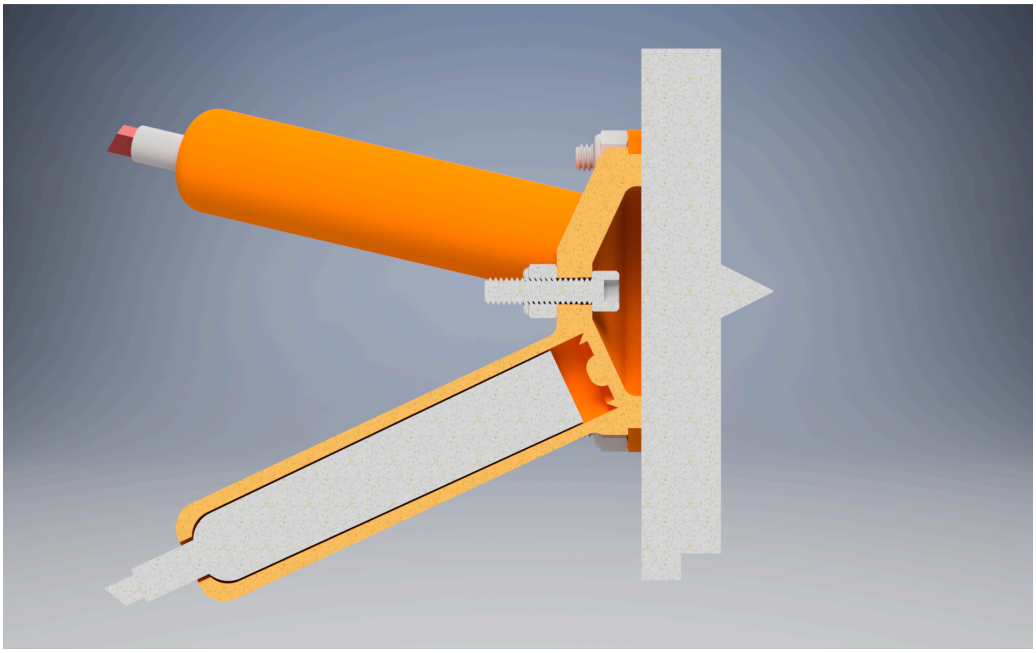

Figure 2: Tool Cross Section

Programming

We created the control algorithms in both C++ and Python inside the ROS framework. We chose to control the robot using ROS instead of Robotstudio and Rapid because it allowed us to do image processing (reading in .png files) in Python, a task that would be impossible in Rapid without using .csv files as an intermediary. A flow chart of our system architecture can be seen in Figure 3. The ABB controller had to be configured to run various modules provided by ROS Industrial in order to communicate with the ROS industrial stack running on a networked Ubuntu machine.

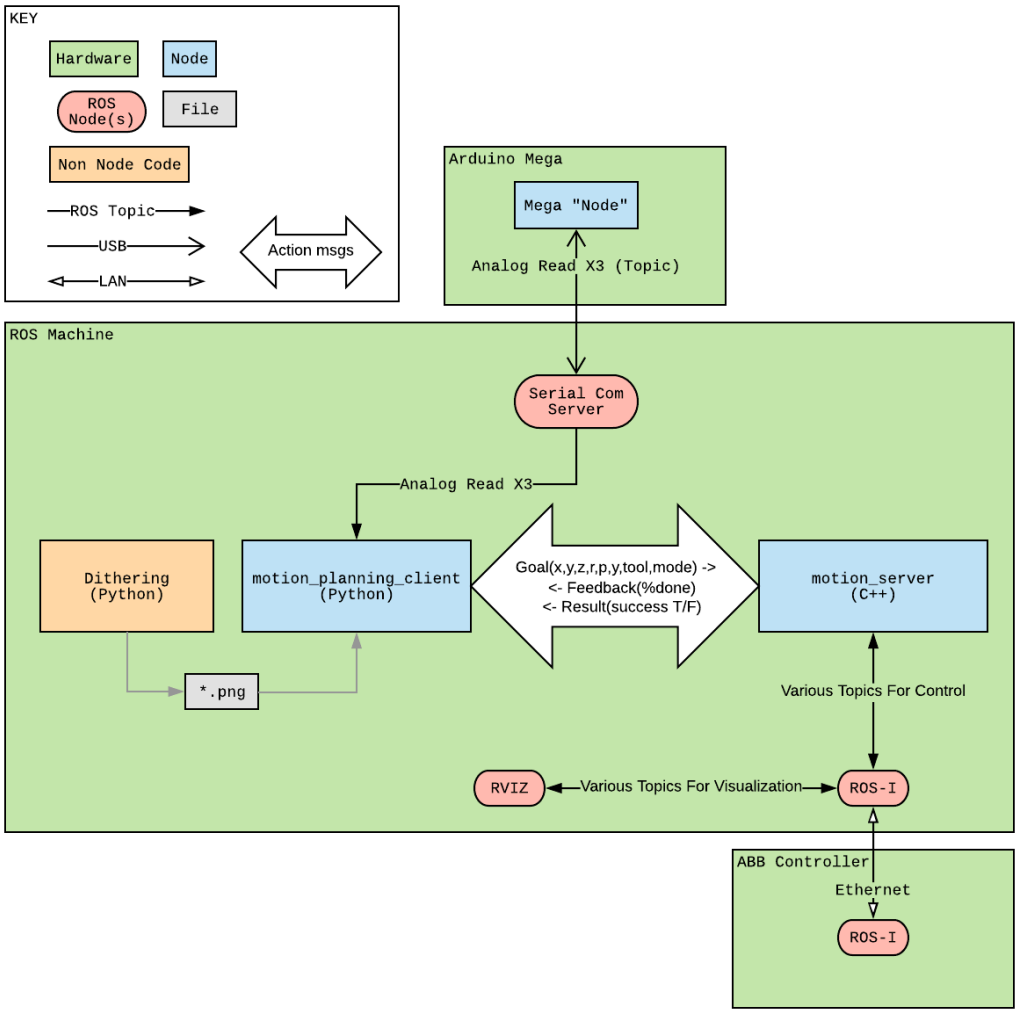

Figure 3: System Architecture Flowchart

When launched the ROS-I stack establishes a connection with the robot controller, solves the kinematics using frame transformations defined in our URDF file and the TRAC-IK Kinematics Solver, and prepares to receive ROS MoveIt commands. Currently only C++ MoveIt interfaces have been implemented for ROS Industrial, so ROS actionlib was used to simplify motion commands and allow us to control the robot using Python. Actionlib allows for a call and response communication between nodes. This was important because we did not want new motions to be sent to the robot until we had verified that the last command was executed successfully. The *motion_server* node waits to receive goal messages from the client in the form

(x, y, z, roll, pitch, yaw, tool, mode) and then uses ROS MoveIt to create cartesian (linear) motion plans that are then sent to the ROS Industrial stack. The tool field specifies which chain of fram transformations should be used to solve the kinematics as defined in the URDF file. The mode field sets motion planning to either “moveJ”, “moveL”, or “peck”. The first two functioning similar to the Rapid equivalent. Peck mode moves to an *x*, *y* location at a clearance height then goes to a touching height and lastly returns to the clearance height. Having these three locations in the server side of the action nodes allows for faster run times as one goal from the client can plan three motions. In other modes each individual motion must successfully execute and confirmation be sent to the client before another goal request is sent. During execution the server sends feedback messages of percent completion with the motion and upon completion it sends back a boolean representing if the motion was successful. ROS actionlib allows for preemption, thus we can cancel motion requests if we desired which will prevent the server from continuing to send requests.

The *motion_planning_client* is the main path planning code. It reads in a .png image that can be created using any image editor or using our dithering script. If in debug mode, the program will go to a single user specified location. This is useful for ensuring your z depth values are accurate. Under normal operation, the program will loop through the colors that are currently in the EOAT defined as an array of strings. For each color to be drawn the program loops through the x and y values of the image. If a pixel in the image matches the color currently being worked on the robot will go to a clearance height over that location, down to a touching height, and back to clearance. The location of each point is calculated using the equations in Table 1 within two nested for loops that represent the coordinate axes. The color specific offsets were added in to account for problems low accuracy but high precision we experienced when changing tools.

Table 1

Value	Equation
xTarget	= xOrigin+colorSpecificOffsetX +(i*scalingFactor)
yTarget	= yOrigin+colorSpecificOffsety +(j*scalingFactor)

The color by color method was selected because it resulted in much faster cycle times than doing each cell in order. Changes in pose take much more time to do than simple translations in the same orientation and going color by color rather than point by point minimizes the number of rotations. Even faster programs could be created by prioritizing neighbors of the same color instead of iterating over the whole image looking for pixels of the current color tooling. Implementing this program was made easier by creating a python class with many important helper functions that are explained in Table 2.

Table 2

Function	Use
forceCB	Gets called by subscriber when a force topic is published to by the Arduino. Used to cancel motions if force is too high.
clear	Keeps the same x,y,r,p,y but executes a motion in z to the clearance height
touch	Keeps the same x,y,r,p,y but executes a motion in z to the touching height
changeTool	Keeps the same x,y,x,r,p,y but changes to controlling in a different tool frame
moveXYZ	Keeps the same r,p,y but executes motion in x,y,z
moveXY	Keeps the same z,r,p,y but executes motion in x,y
move	Executes a motion in x,y,z,r,p,y for a specified tool frame
draw	Keeps the same r, p, y but executes an x,y motion in peck mode

In order to interpret images and draw them with the colors we had available, a python script was used to dither the images. The script was written with help from Jacob Henry, a friend of the team members. Dithering is a process that reduces the number of colors in an image by grouping colors to create the illusion of a wider range of color. We used Stucki dithering, which is a particular algorithm for distributing error in each point to subsequent pixels. For example, if the first pixel of the drawing was a shade of purple closer to blue, the algorithm would likely make it become blue and mark the error between the original shade and the new one. That error would include the ignored red value, a negative blue value, and minimal green that would be randomly distributed among a small random number of subsequent pixels. If another similar shade of purple receives enough red compensation, it might then be chosen as red. Applied to an entire image, this algorithm produces accurate representations of images using only six colors.

The colors to be dithered down to can be specified by changing the RGB values held in an array. The imported image is then run through the algorithm that results in an image with the selected colors in a selected resolution. This PNG image could then be directly imported into the *motion_planning_client* to be drawn.

Results and Discussion

The ultimate goal of the project was to draw a six color image of high resolution, and naturally numerous issues arose as we began drawing. Before we even began drawing, we noticed that our motion commands were not linear. During our initial tests it became clear the markers were drawing in slightly different frames that were uncoordinated with each other. Additionally, as we began to draw test images it became apparent how long it takes to draw large pictures.

Linear motion is necessary to draw in a 2D plane since joint motion could include rotations that go through said plane. We found that the arm moved in a slight arc when attempting to move the tooling linearly and initially suspected motion planning. However, the cartesian motion planning used was not the source of this error. It occurred to us that the URDF model could be inaccurate and upon checking the values found that link 2 was set to the length of the 1.2 meter reach model of the arm. Given that we are using a 1.45 meter reach, this understandably caused the error we were observing. Correcting the value smoothed out our motions and prevented us from smashing any more markers than we already had. Once this was corrected, the arm was able to draw with such consistency that the force sensing became trivial. The markers were always depressed the same amount, so the only case when the sensing would be needed is with faults. In a full industrial solution this would still be implemented as a precaution, but we did not include it in our final product.

As mentioned in Task Specifications the markers have the ability to rotate within the tooling that houses them, resulting in the need to calibrate two of the markers to the third in the x and y dimension to ensure the drawn points are properly aligned. The process of calibration was achieved by first orienting all of the chisel tips of the dry erase markers such that they were turned away from the center of the tooling. After the orientations of all of the markers is known,

the arm proceeds to draw a dot with marker one, followed by two dots drawn with marker two above and to the left of the marker one dot, and finally two dots drawn with marker three below and to the right of the marker one dot. This five dot pattern allows for offset values to be measured for markers two and three in both the *x* and *y* directions, and is shown in Figure 4. This calibration sequence can be repeated as necessary to achieve the level of alignment prefered by the user.

A before and after side by side comparison of a basic test image can be seen in Figure 5. The improvement

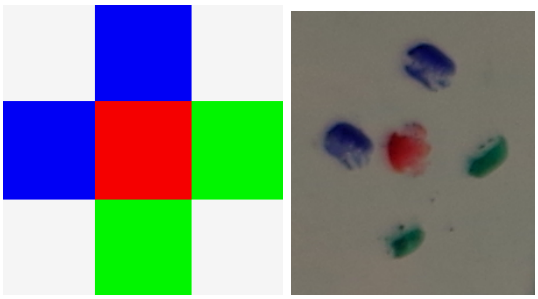

Figure 4: Calibration Square

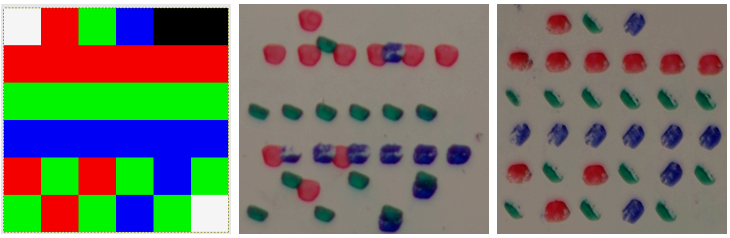

Figure 5: before and after comparison calibration sequence

Once we were able to consistently and accurately draw dots across multiple colors we began attempting to draw dithered images. There is a significant amount of overhead involved in the action server and initially it took around 4.5 seconds to draw a single dot, including moving between points. This made drawing images of larger resolutions (above 50x50) take an extremely long time. Figures 6 through 9 below show a larger image we attempted to draw in various stages. The original picture was dithered at a 300x300 scale with six colors and while it

was a very accurate recreation, it was too large to attempt. Figure 8 displays the image dithered with the same colors but at a 100x100 resolution. Even though the quality dropped significantly, it is still recognizable. One initial pass, shown in Figure 9, was drawn but took two hours to only partially complete the first color. Every time we moved the arm a position be sent from the client as a message, which then had to be calculated as a cartesian motion and executed, and finally the server had to send a 'completed' flag back to the client. In order to speed up the drawing process, we took the motion command for lowering to draw and moving back up out of the client and put it in the server. This meant that only one message had to be sent from client to server, the x-y position, in order to draw each point. Drawing time sped up significantly, taking only about two seconds for each point.

As a proof of concept, we decided to draw an undithered image with six colors. This demonstrated our ability to accurately draw with more than three colors, which required changing out the markers mid way through. Figure 10 shows the original image compared to the drawn one.

Figure 6: Original Image

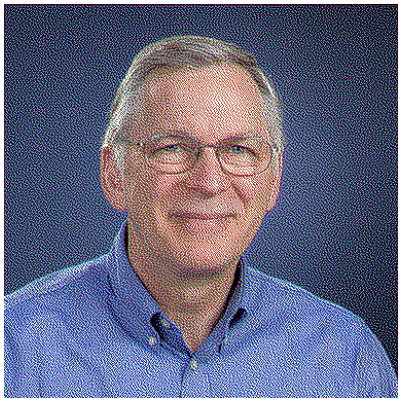

Figure 7: 300x300 Dithered Image

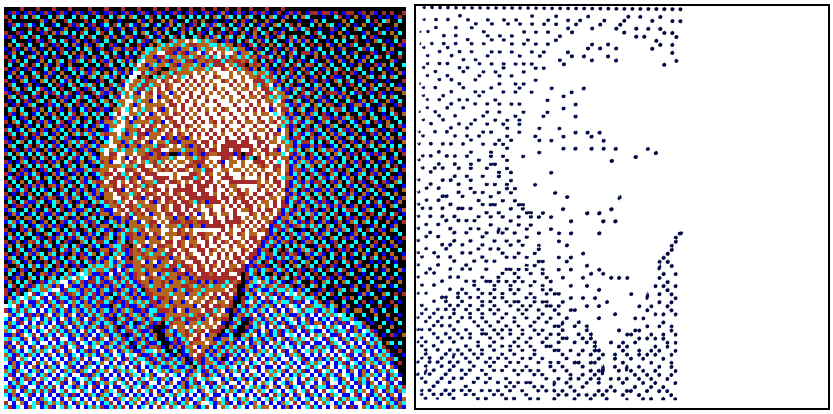

Figure 8: 100x100 Dithered Image *Figure 9: Drawn Image*

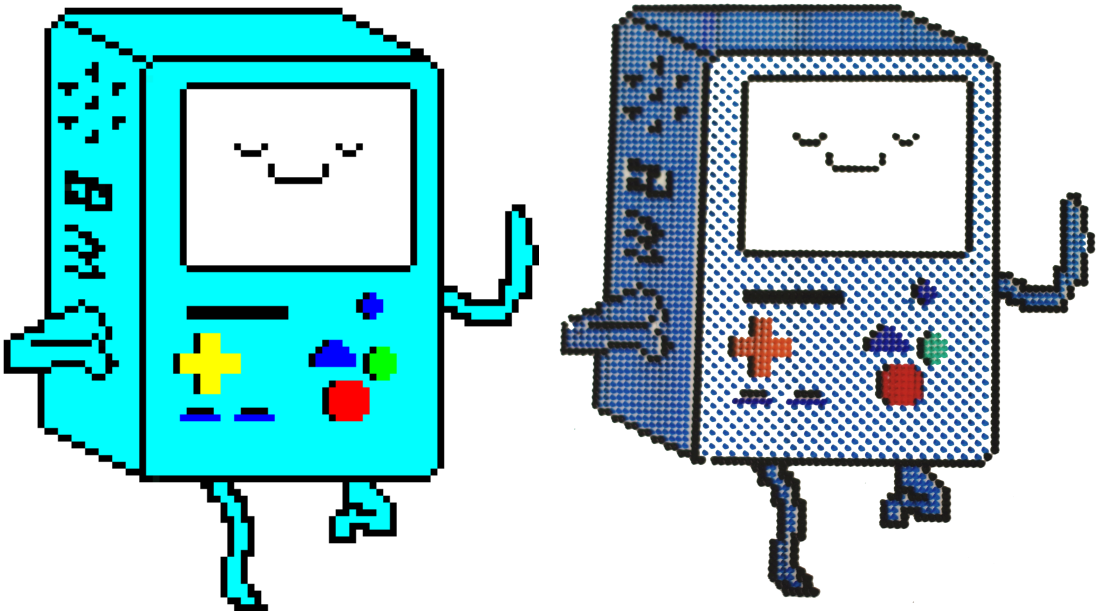

Figure 10: Computer Image Compared to Seurobot Drawn Image

Conclusions

Overall, we very clearly met our goal for this project. The ABB arm was able to draw precice pointillism drawings that very closely matched the dithered images used. The most significant difficulty we encountered during this project was calibrating the markers to ensure they drew consistently. In the process, a handful of markers were damaged, but once we had accurate values the drawing itself was relatively simple. This ties to our major takeaway, that once a system is setup *correctly*, it can be used for a variety of applications with relative ease.

We realized that we can now draw effectively anything, given enough time, and could add a number of features to improve upon and expand the project. Some of these potential improvements for future work include:

- Finding ways to reduce or circumvent ROS latency
- Adding computer vision functionality
    - Auto calibration by inspecting calibration squares
    - Image drawing detection for continuing images not finished previously
    - Detect colors available based on test marks enabling more advanced dithering
- A clearance height outer perimeter trace of the image to verify size a location of picture
- Work through move groups to allow for tool changes with only movement of the sixth joint of the arm

This project provided us with valuable experience in using the ABB IRB 1600 and programming inside the ROS framework. The knowledge gained by members of the team during this project will undoubtedly aid in future work with industrial robotics and ROS.

Bibliography

Jones, David. “Dither and Gamma.” *Code Monk*, 24 Apr. 2009, drj11.wordpress.com/2009/04/03/dither-and-gamma/.

“Pointillism - The Dotted Art of Early Modernity.” Edited by Widewalls Editorial, *Widewalls*, 6 July 2016, www.widewalls.ch/pointillism-dotted-art/.

Appendices

Git Repo:
https://github.com/RBE4815-Team6/Seurobot

Presentation:
https://docs.google.com/presentation/d/1OggcqU1L_PCX-qB2dhqx_fGIgzLUQOF2_rYP6aNHVLE/edit?usp=sharing