Painter Robot Design Report 8-15

Samantha B. Chong[1], Olivia Markham[2], Kathryn Robertson[3], and Melissa Jacobs[4]

[1]20995990
[2]21002887
[3]201029321
[4]21002517

December 6, 2022

Contents

1 Introduction **1**

2 Scope **1**
2.1 Changes in Scope . . . 2

3 Constraints and Criteria **2**
3.1 Constraints . . . 2
3.2 Criteria . . . 2
3.3 Changes in Constrains & Criteria . . . 2
3.4 Guiding Project Design . . . 2

4 Mechanical Design and Implementation **2**
4.1 Overall Description . . . 2
4.2 Components . . . 3
4.2.1 Chassis . . . 3
4.2.2 Motor Drive . . . 4
4.2.3 Sensor Attachments . . . 4
4.2.4 Overall Assembly . . . 5
4.3 Design Decisions and Trade-Offs . . . 6
4.3.1 Design Decisions . . . 6
4.3.2 Trade-offs . . . 7

5 Software Design and Implementation **8**
5.1 Summary . . . 8
5.1.1 Tasks explanation . . . 8
5.1.2 Demo Task List . . . 9
5.1.3 Description of Functions . . . 10
5.1.4 Data storage . . . 13
5.2 C++ . . . 14
5.2.1 Summary . . . 14
5.2.2 Design . . . 15
5.2.3 Utilization of buttons and labels . . . 15
5.2.4 Data Storage . . . 15
5.3 Variables . . . 16
5.4 Testing . . . 19
5.5 Problems Encountered . . . 21
5.5.1 Inability to move x-axis . . . 22
5.5.2 Overheating of the z-axis . . . 22
5.5.3 Inconsistent movement of Paper . . . 22
5.5.4 Lack of Precision . . . 22
5.5.5 Too much water in the brush . . . 22
5.5.6 Non-linear movement of the paint cups . . . 23

6 Verification **23**
6.0.1 Start-up . . . 24
6.0.2 Regular-Tasks . . . 24
6.0.3 Shut Down Procedure . . . 24

7 Project Plan **24**

8 Conclusions **25**

9 Recommendations **26**
9.1 Mechanical Design 26
9.2 Software Design 26

10 References **27**

11 Appendix A - RobotC Code **28**

12 Appendix B - C++ **33**

List of Figures

1 View of Final Design Assembly . . . 3
2 Brush Chassis . . . 3
3 Y-Axis Chassis . . . 3
4 Orthographic View of Paint Cups . . . 4
5 X-Axis Belt Motor Gear Reduction . . . 4
6 Touch Sensor Mount . . . 5
7 Overall Assembly . . . 5
8 Original Paper Feeder Design . . . 6
9 Current Paper Feeder . . . 6
10 Original Paint Cup Layout . . . 6
11 Paint Belt . . . 6
12 Brush Belt Before Adding Tension and Added Supports . . . 7
13 Brush Chassis with Added Support Bars . . . 7
14 Brush Chassis made from Lego . . . 7
15 Brush Squeeze Drying Method . . . 8
16 Flow Chart of Main Robot C Program . . . 9
17 Flow Chart of Set Encoder Functions . . . 11
18 Flow Chart of Go To Point Function . . . 11
19 Flow Chart of Dip Brush Function . . . 11
20 Flow Chart of Wash Brush Function . . . 12
21 Flow Chart of Check If Colour Exists function . . . 12
22 Flow Chart of Get New Colour Function . . . 13
23 Flow Chart of Paint Function . . . 13
24 The Colours(B#_#) Function Flow Chart . . . 14
25 The Button Clicked Events Flow Chart . . . 14
26 Main Code Flow Chart . . . 15
27 GUI with a Purple Pixel . . . 16
28 Array with a Purple Pixel . . . 16
29 SunSet C++ Program View . . . 20
30 Sunset Image 1 . . . 21
31 Sunset Image 2 . . . 21
32 Sunset Image 3 . . . 21
33 Sunset Image 4 . . . 21
34 Flower Painting . . . 25
35 Flower C++ Program . . . 25
36 Whale Painting . . . 26
37 Whale C++ Program . . . 26

List of Tables

1 Robot C Constants 17
2 setEncoderX, setEncoderY and setEncoderC Variables 17
3 goToPoint Variables 18
4 DipBrush Variables 18
5 WashBrush Variables 18
6 CheckIfColourExists Variables 18
7 getNewColour Variables 18
8 Paint Variables 18
9 RobotC Main Variables 18
10 C++ Variables 19

Executive Summary

The aim of this project was to create a robot capable of painting a user-generated pixel art image onto a standard 8.5" by 11" piece of paper. These criteria were completed using an EV3 LEGO Mindstorm and Tetrix kits. The resulting robot was able to paint a variety of images using an input from a C++ program, with these variable-sized images each containing multiple colours. The goal of the robot was to aid individuals less capable in the arts in completing their class homework, and as shown through this report, the team succeeded in their goal.

Acknowledgements

The team wants to thank Mathew for his help in the C++ programming, Arni for her help with the mechanical side of things, specifically in recommending the Tetrix kit, Ryan for providing our team with all the belt and the University of Waterloo for providing us the opportunity to create this system.

1 Introduction

The aim of this project was to create a robot capable of painting an image from user-generated pixel art. The robot is able to paint and change colours.

This robot will use four axes to complete these functions, two y-axes, an x-axis and a z-axis. These axes will allow the robot to plot and change the paint colour.

A C++ interface allows users to select the colours they want in each location for the painting. To create the painting, the robot will iterate through each pixel on the painting and paint them colour by colour. The x and y axes will be the two main axes used to reach the correct location to paint. The z-axis will then dip the brush against the paper in the correct location. The second y-axis is the colour axis or the c-axis. This axis moves in the y-direction parallel to the y-axis, moving the paints to the correct location so the brush can reach the correct colour.

The robot is made using an EV3 LEGO Mindstorm kit, a Tetrix kit, and an assortment of LEGO bricks. Also used were four metal rods from a shoe rack, duct tape and Vaseline.

An important consideration is the order of the colours. As the robot will likely be unable to get the brush completely clean between each colour, it is essential to order the colours so the robot has the highest chance of success. Based on this, yellow is the first colour and black is the last colour. The colours between yellow and black were determined based on colour mixing. The colour order is yellow, orange, red, purple, blue, green and black. As a result, this is also the order the colours are referred to within the code, with yellow being one, orange being two and continuing similarly with the rest of the colours.

2 Scope

The robot's main task is to paint an image using a variety of sub-tasks. Using the C++ interface, the user can create a pixel art image which will then be sent to the robot for it to paint.

It is important to understand some of the terminologies used in this report before the sub-tasks the robot completes can be discussed. The axis are named based on their movement, similar to 3D printing where x is the horizontal axis going left-right, y is the forward back axis, and z is the vertical axis.

To complete the above-mentioned tasks, the robot uses belts to move both the paint-brush on the x-axis, and the paints on the y-axis. The y-axis of the paper uses a pair of wheels which moves the paper back and forth. To dispense the paint, the robot uses a sponge brush and a motor that moves the brush in the vertical direction.

The robot uses four inputs to complete the painting. The first input, and one of the most important, is the file it receives from the C++ program. This file creates an array of pixel colours for the robot to read.

The second input is the motor encoders which allow for precise movement in the x and y-axis of the brush and paper so that the paint is accurately plotted.

The third is a touch sensor, which aligns the colours for the paintbrush to collect. The touch sensor ensures the colour is centred, so the brush does not miss when it dips into the colour. Originally a distance sensor was used in parallel to the touch sensor to determine the placement of the colours. This was replaced with a motor encoder due to inaccuracies in the system.

The fourth input is the buttons used within the initialization process for the user to alert the program that the initialization steps are complete.

The robot will recognize that the painting is complete when it exits the for loop that iterates through the 2D array. Additionally, once the for loop has been exited, the robot

will go into a shutdown procedure in which it washes the brush, dispenses the paper to the user and returns the brush to its initial start position.

2.1 Changes in Scope

Initially, the robot would receive an image, turn it into pixel art and then paint the resulting pixel art. Due to issues with downloading the OpenCV library, it was decided that it would be better to have the user create the image and then output the resulting data to the robot using a different method.

3 Constraints and Criteria

3.1 Constraints

A constraint of this robot is the accuracy of the robot with the available materials. For the painting to look good and be an accurate reflection of the users drawing, the placement of each pixel needs to be accurate within a millimetre or two. Another constraint of the robot is the size of the paper. The robot was designed for standard 8.5" by 11" paper, meaning that images with more pixels will be unable to be painted due to the standard mechanical system size. Any image requiring more than 16x16 pixels will be unable to be painted on this system due to that size requirement.

3.2 Criteria

One requirement of this robot is the ability to change the colour of the paint. Colour changing will allow for better-looking images that accurately portray the creator's desires. Additionally, the robot requires four axes of movement, two on the y-axis and one on both the x and z-axis. These axes provide the robot with the needed mobility to complete each painting. The robot must be able to receive a pixel art image from the C++ program, allowing the art to be painted.

3.3 Changes in Constrains & Criteria

The one significant change was that, initially, the user would be able to turn an image found elsewhere into pixel art. However, due to software issues, the user is no longer able to simply upload a picture themselves, but now need to create pixel art in the C++ program.

3.4 Guiding Project Design

Keeping these constraints and criteria in mind while designing was essential in keeping the project on track and from diverging into a completely different idea. Specifically, it was helpful to keep the constraints in mind while designing, as issues had been thought of before they came up, and solutions had been designed around them.

4 Mechanical Design and Implementation

4.1 Overall Description

The robot moves a paintbrush over a piece of paper and paint cups to paint pixel art. It does this through movement on the x and y axes. Additionally, it moves in the z-axis to dab the brush on the paper, paint, water or paper towel. These functionalities are achieved using three main components: a paint conveyor, a paper feeder and a brush conveyor. The

paint conveyor moves along the y-axis, placing the desired paint colour directly under the brush's path. This is accomplished using motor encoders and a touch sensor to locate if the correct colour is placed properly. Alongside the paint conveyor, the paper feeder shifts the paper on the y-axis to adjust which section of the paper was being painted. The paper feeder calculates the position of the paper using a motor encoder. The motor encoder is attached to the motor that drives two wheels that move the paper by friction. The paintbrush predominantly moves along the x-axis with z-axis movement to dip the brush into paint or onto the paper. When this is all assembled (Figure 1) the robot is able to create 16x16 pixel art.

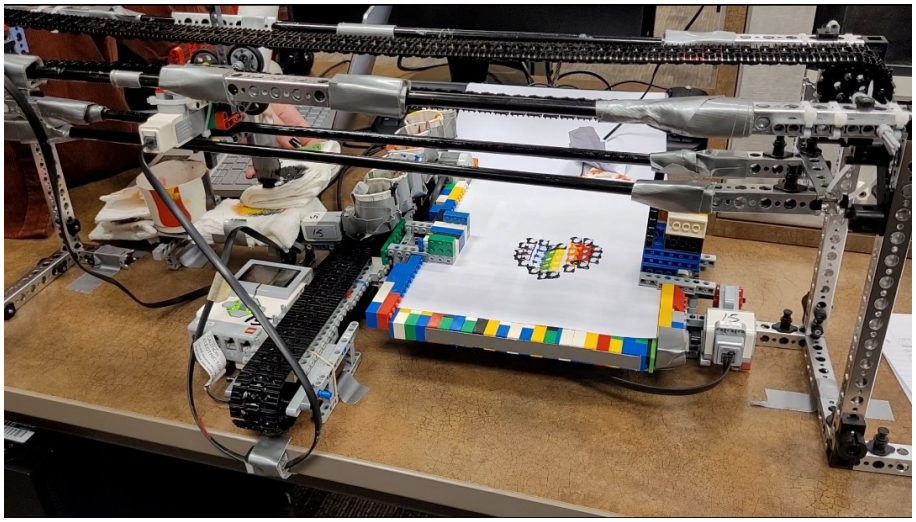

Figure 1: View of Final Design Assembly

4.2 Components

4.2.1 Chassis

The chassis has two main sections: the brush (Figure 2) and the y-axis chassis (Figure 3).

Figure 2: Brush Chassis

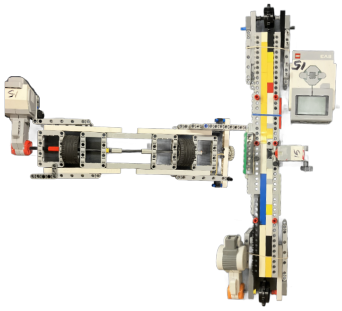

Figure 3: Y-Axis Chassis

Brush Chassis: The brush chassis is used to support the belt and sprockets that move the brush along the x-axis. The end supports that hold the sprocket axles and

reinforce the brush's support rods are made using a Tetrix kit. The chassis also includes four metal rods: two connect and support the end axle supports, and two act as tracks for the brush mount to ride along.

Y-Axis Chassis: The base of the y-axis consists of Lego Techno pieces as a frame for two wheels on the same axle spaced about 16 cm apart. To make a smooth surface for the paper to move normal Lego blocks are placed on top of the frame with two holes, one for each wheel to go through. The tops of the wheels are raised slightly above the Lego surface for them to be in direct contact with the paper. To ensure the paper moves with the wheels, guides are placed on top, adding friction. Connected to the main frame is a smaller frame that holds the paint belt and sprockets.

4.2.2 Motor Drive

The design uses a total of four motors: two motors on the brush chassis and two motors on the y-axis chassis. On the brush chassis, a large motor drives the belt holding the brush along the x-axis. The motor is parallel to the belt and has a 2:1 gear reduction (Figure 5) so there is enough torque to drive the belt. A second medium-sized motor is attached to the underside of the belt; by converting the motor's rotary motion into linear motion, the paintbrush moves on the z-axis. The other two motors are used on the y-axis chassis to move the paper and paint. The large motor is connected to one long axle connecting two wheels on opposite ends of the y-axis chassis frame, allowing the paper to move. The final large motor drives a belt, holding the seven paint cups (Figure 4), along the y-axis directly beside the paper feeder.

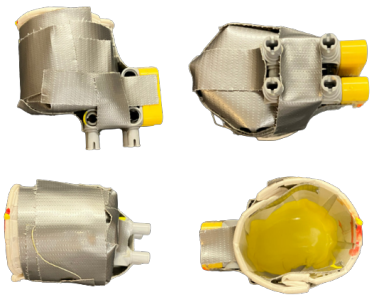

Figure 4: Orthographic View of Paint Cups

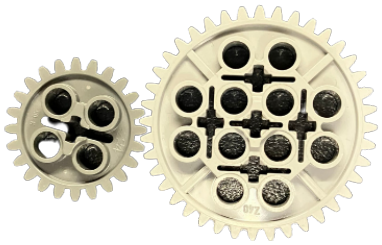

Figure 5: X-Axis Belt Motor Gear Reduction

4.2.3 Sensor Attachments

Two types of physical sensors are implemented on the robot: motor encoders and a touch sensor. Three motor encoders are mounted on the robot alongside their corresponding large motors. The motor encoders mounted on the paper feeder motor and the x-axis belt motor provide the location information needed to place the paint on the paper accurately. The third motor encoder is attached to the paint belt's motor on the y-axis; this encoder works alongside the touch sensor. The touch sensor is triggered when a paint cup is located where the brush can access it. The touch sensor is mounted to the side of the paint belt to achieve this, as shown in figure 6.

Figure 6: Touch Sensor Mount

4.2.4 Overall Assembly

The overall assembly consists of two separate assemblies. One of the assemblies combines the paper feeder, paint belt, and wash station. Placed over this assembly is the other assembly, which consists of the brush's belt and the brush's dabbing mechanism.

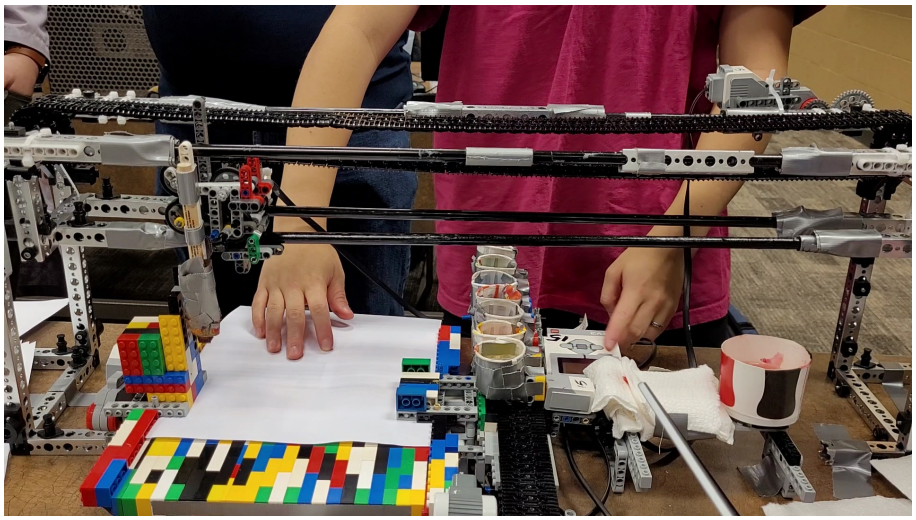

Figure 7: Overall Assembly

As shown in figure 7, the paper feeder is on the far left; it consists of a base that holds two wheels approximately 16 inches apart. Placed around these wheels is a platform made of Lego for the paper to slide along. The paint belt connects to the right side of the paper feeder, comprised of a belt running along the y-axis with paint cups, figure 4, connected to the top. The brush washing station is directly to the right of the touch sensor on the paint belt's frame. Paper towels and a cup of water are taped to the surface of a Lego table washing station.

The other assembly, shown in figure 1, includes the paint brush's dabbing mechanism

and the belt intended to position the brush along the x-axis. The dabbing mechanism uses wheels and friction to convert rotary motion into linear motion. The motor of the mechanism is connected to the underside of the belt running along the x-axis. The motor is supported by two metal rods placed underneath it to take tension off the belt.

4.3 Design Decisions and Trade-Offs

4.3.1 Design Decisions

Figure 8: Original Paper Feeder Design

Figure 9: Current Paper Feeder

The first significant design decision was with the paper feeder. Initially, the design had two layers of wheels stacked on top of each other, as shown above in figure 8. The decision to change to the current design (right side of figure 9) was due to the free-spinning top wheels allowing the paper to slip more and move a shorter than intended. Instead, Lego blocks were placed almost flush with the top of the wheels to apply friction to the paper and reduce slippage.

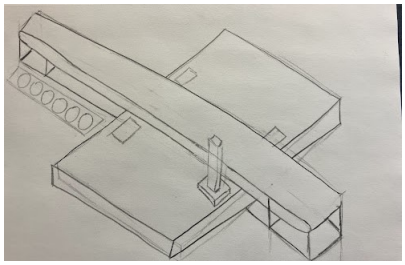

Figure 10: Original Paint Cup Layout

Figure 11: Paint Belt

The next decision was to change the paints from being stationary to being on a belt. Initially, the plan was to have the paints lined up in a row along the x-axis (Figure 10), and the brush would move to them. However, to improve efficiency, the paints were switched to a belt running along the y-axis (Figure 11), allowing them to move to the brush. This meant the brush could move a shorter distance to get paint. Resulting in a more compact and efficient design.

Figure 12: Brush Belt Before Adding Tension and Added Supports

Figure 13: Brush Chassis with Added Support Bars

During testing, the belt was found to be sagging too much (Figure 12), causing the brush mechanism to be on an angle and have inaccurate placements. To overcome this two metal rods were added for the motor to sit and ride on (Figure 13). This took the weight off the belt, fixing the issue of it sagging but adding an issue of friction, causing the belt to seize a lot. The friction was fixed by adding lubricant to the bars.

4.3.2 Trade-offs

The main design trade-offs that had to be made were: using metal over Lego for the x-axis, the brush cleaning procedure, and only using one brush instead of multiple.

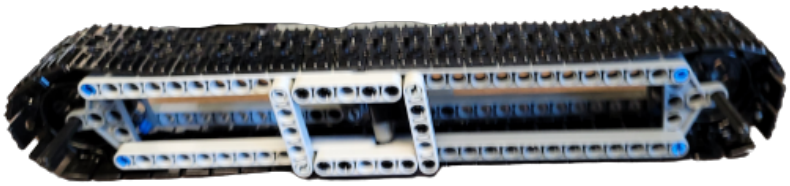

Figure 14: Brush Chassis made from Lego

For the x-axis chassis, the original plan was to use Lego for the supports and beams (Figure 14). However, the Lego was too flexible to tension the belt properly, so those parts were switched to Tetrix parts. The trade-off from switching to metal made the design much heavier and bulkier than the original one made from Lego. Once the x-axis was built using only a Tetrix kit, another issue arose with the amount of play in the Tetrix fasteners. The design underwent another change. The vertical supports did not change from being Tetrix. However, the horizontal supports became metal rods duct taped to the supports. The final design is not as light or compact as the original but is more functional in its current iteration.

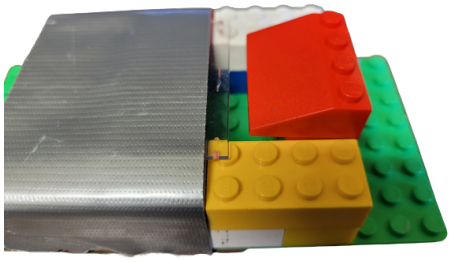

Figure 15: Brush Squeeze Drying Method

Another trade-off that had to be made was the brush cleaning procedure. Initially, the brush would be dipped into the water cup through an open slit slightly smaller than the brush (Figure 15), squeezing the moisture out when the brush is lifted out of the water. This design would have allowed the brush to be dabbed onto the paper towel fewer times. Due to the quality of the brush, this design was impractical due to it destroying the brush. The brush life was extended by only dabbing the brush on a paper towel, but in return, time efficiency was lost.

The original plan for changing paint colours was to have a dedicated brush for each colour and a gripper system. This design would eliminate the need for a brush-washing procedure. However, the weight of a gripper system and a z-axis movement system was more than the Lego Techno belt and axles could handle. The design change to a washing procedure results in a more complex design and a longer cycle time.

5 Software Design and Implementation

5.1 Summary

The project utilizes both C++ and RobotC programs. C++ creates a graphical user interface (GUI) where the user can create a 16 x 16-pixel image with a maximum of 8 colours. The RobotC then uses the resulting 2D array as an input. The program iterates through all the colours and pixels to correctly paint each pixel on the paper. Functions such as goToPoint, getNewColour and dipBrush are used to accomplish this.

5.1.1 Tasks explanation

The robot's tasks are split up into three major components: initialization, major procedure, and shutdown. These are all shown in figure 16. The overall program was broken into smaller blocks, with those blocks being broken into even smaller tasks in order to make both the writing of the code, as well as testing easier to focus on and debug.

Initialization

Once the program has been selected, the y-axis will continuously turn at half the regular speed so the paper can be loaded into the machine. Once the user presses the up button on the EV3, the y-motor will stop, and the x-axis will move over the paint containers. This will allow for verification that the paint containers are in the correct place and that everything else is initialized correctly. The program will continue to the painting stage once the enter button is pressed. During this process, displayString() will display the instructions to the user. This procedure is as shown in the initialization section of figure 16

Major Procedure

The major procedure contains a large set of tasks the program must complete. It first uses the CheckIfColourExists() function to determine what colours the painting contains. Next, it iterates through a for loop for all of the colours in the painting. Then the Paint() function is called. This function calls the simpler task functions. It first iterates through each colour of the painting. In each iteration, it accomplishes the following. It calls the getNewColour() function to get the correct paint, then uses two nested for loops to access the individual pixel. Consequently, it can iterate through the whole painting in that colour and paint only the appropriate pixels. This function includes getting more paint when the brush runs out. After Paint() is called, WashBrush() and getNewColour are called to set up for the next iteration. The major procedure concludes once all of the colours have been iterated through. This procedure is as shown in the painting process section of the figure 16. The robotC code can be found in Appendix A.

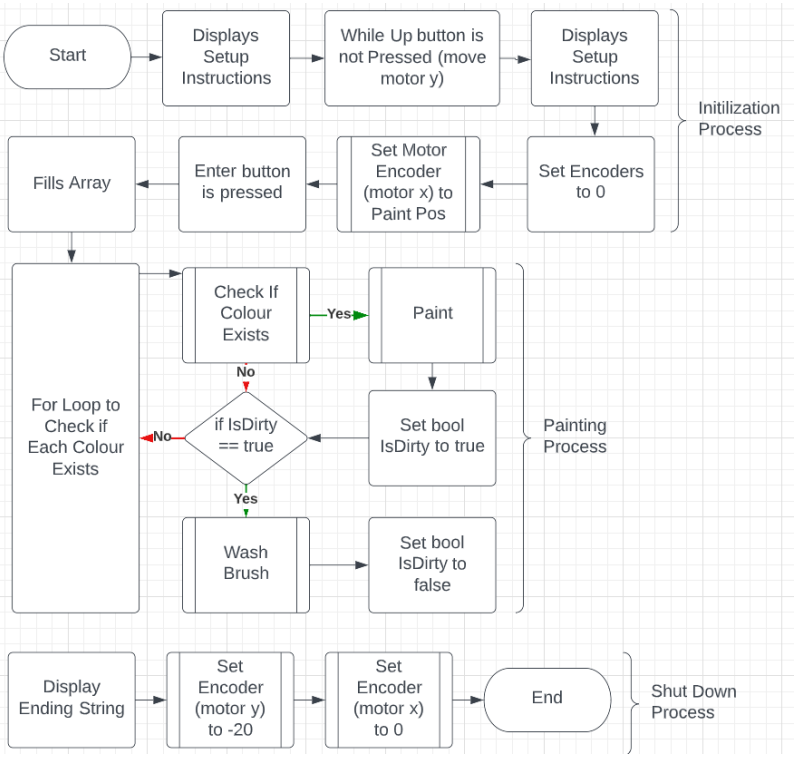

Figure 16: Flow Chart of Main Robot C Program

Shut Down

The shutdown procedure begins by calling the WashBrush() function to clean the brush. The y-axis motor is then run to push out and display the paper fully. Finally, the displayString() will output the end of the shutdown procedure. This procedure is as shown in the shut down process section of figure 16.

5.1.2 Demo Task List

Below is the task list. When testing it was found to be much easier to have an initialization program to make sure that all key components of the robot were in the correct position before conducting the main program. Therefore this new task list contains a description of the start-up procedure.

Start Up

- Robot moves in the y-axis until the up button is pressed.
- Moves the paintbrush to the start position, the top left corner of the page.
- Displays the starting instructions.
- Fills Array
- Begins when the enter button is pressed and released.

Major Procedure

- Goes through the painting colour by colour to ensure it exists.
- For each colour, gets the correct colour on the paintbrush by moving the paint y-axis and dipping the brush.
- Goes to each point by iterating column by column and going to 0 in the x-axis
- Refills the paintbrush every 4 pixels
- Once finished with the colour, cleans the brush.
- Continues to next colour

Shut Down

- Wash Brush
- Push out Paper
- Display ending String

5.1.3 Description of Functions

The robotC code can be found in Appendix A, and described in the flowcharts below.

void setEncoder (float distance)

The functions setEncoderX and setEncoderY are the same. However, one is for the x-axis and the other for the y-axis. The function receives a distance which is used to determine where the paintbrush will move. setEncoderC is very similar, with the only significant difference being that it uses a quadratic equation to convert from pixels to cm. The distance refers to the distance from the origin rather than the distance from the current point. This method reduces the math needed in other areas of the code. The function uses a motor encoder to move their axis until the correct distance is reached using a while loop. The program also allows for bidirectional movement using if-else statements and uses a cm to clicks conversion to accurately calculate the distance. These functions were both written by Samantha and are shown in figure 17. The variables are also referenced in table 2.

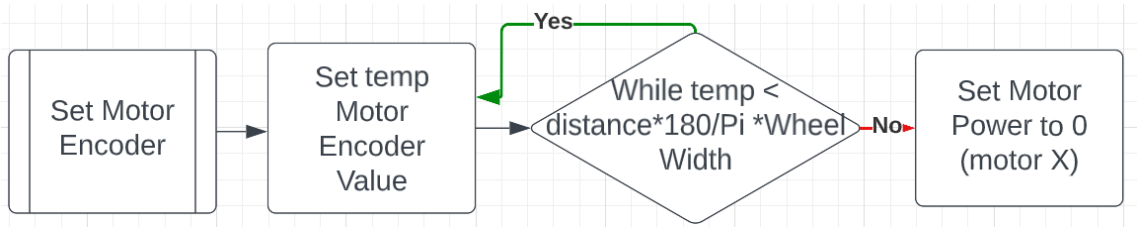

Figure 17: Flow Chart of Set Encoder Functions

void goToPoint (int x, int y, motorPower)

The void goToPoint(int x, int y, motorPower) moves the paintbrush to the corresponding point on the paper. It uses the setEncoderX and setEncoderY functions with parameters of the x-coordinate and y-coordinate of the point. Samantha wrote this function and it can be visualized through figure 18. The variables are also referenced in table 3.

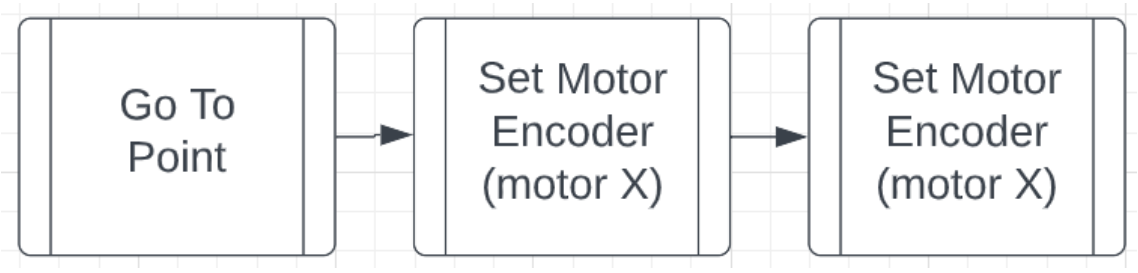

Figure 18: Flow Chart of Go To Point Function

void DipBrush()

void DipBrush() does not take any parameters. It uses the timer to move down for a specified amount of time and back up. This function gets paint onto the paintbrush, dips the brush into water, and then onto a paper towel to remove the paint. Kathryn and Olivia wrote this function, and is shown in figure 19. The variables are also referenced in table 4.

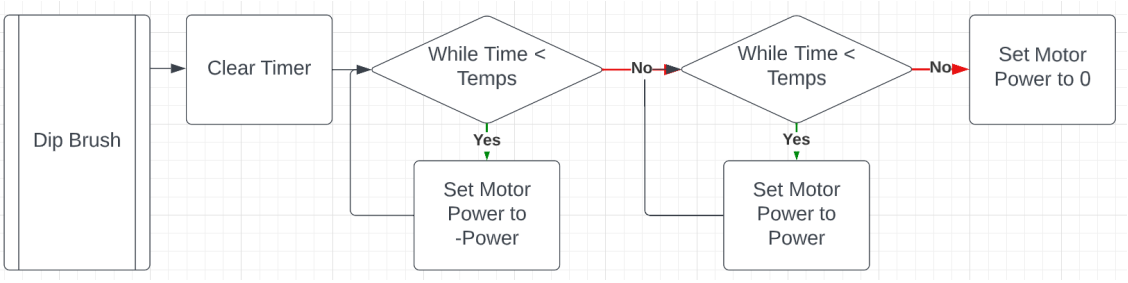

Figure 19: Flow Chart of Dip Brush Function

void WashBrush()

void WashBrush() does not take any parameters. It uses the setEncoderX function to go to the cup of water and then the DipBrush() to get rid of the paint. Using the setEncoderX function again, it moves the brush to the towel and dips it to get rid of the water. The

setEncoder function uses the constant distance variables WATER_POS and TOWEL_POS as parameters to accurately go to the respective position. Olivia wrote this function, and the visualization of this process is explained in figure 20. The variables are also referenced in table 5.

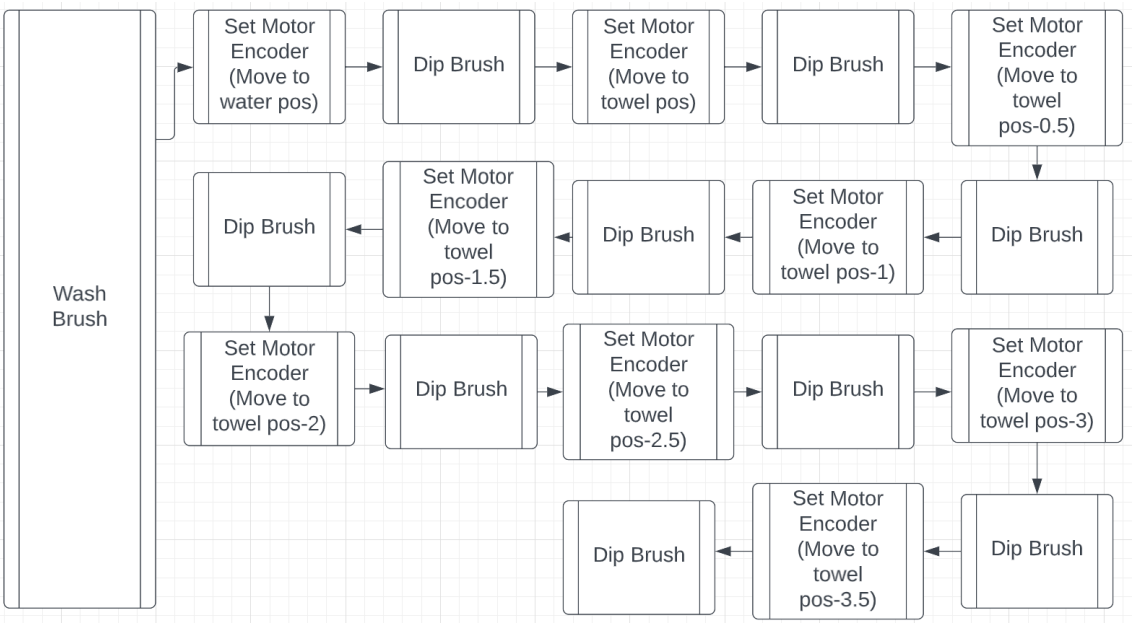

Figure 20: Flow Chart of Wash Brush Function

bool CheckIfColourExists (int ColourNumber)

bool CheckIfColourExists(int ColourNumber) goes through each pixel and checks if the colour exists. The function uses two for loops in order to access the point and returns true if the pixel in the array matches the colour. After going through all of the pixels, it returns false. Kathryn wrote this function, with figure 21 describing the process. The variables are also referenced in table 6.

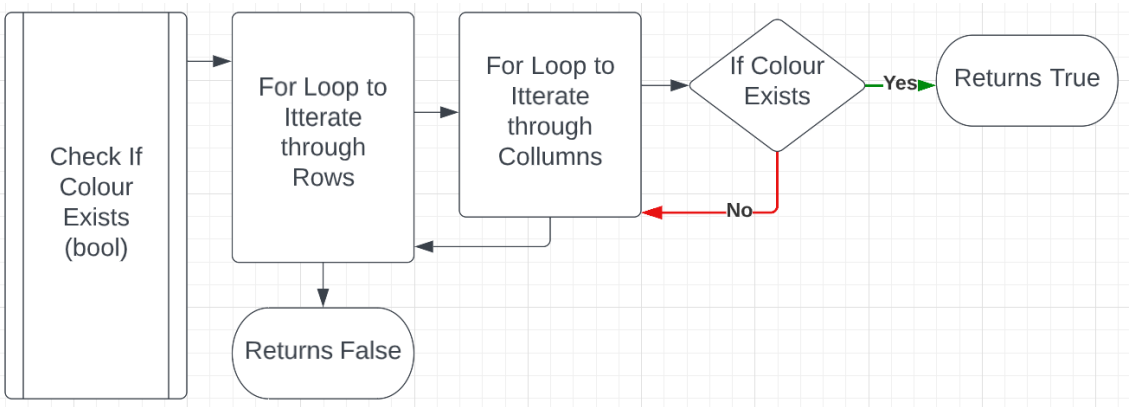

Figure 21: Flow Chart of Check If Colour Exists function

void getNewColour (int colour)

void getNewColour(int colour) takes in the parameter colour and utilizes the x, z, and colour motors to complete its tasks. It first washes the paintbrush using the washBrush() function, then runs the paint belt until the desired colour is reached using the touch sensor to ensure the paint pot is centred. Next, it uses setEncoderX again to move to the paint and DipBrush() to get the paint onto the brush. Melissa wrote this function. Figure 22 is the flowchart for this process. The variables are also referenced in table 7.

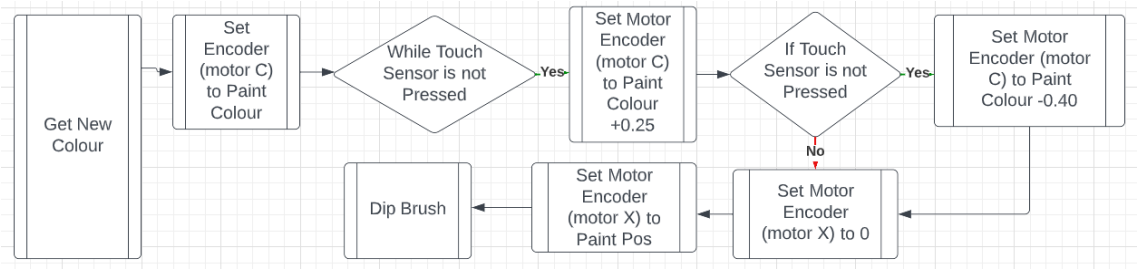

Figure 22: Flow Chart of Get New Colour Function

void Paint (int colourNumber)

void Paint(int colourNumber) iterates through each colour and paints the whole painting for that colour. First, it calls the getNewColour() function to get the correct colour on the paintbrush. Next, it uses two for loops to iterate through the 2-D array and access the individual pixel. If the pixel colour matches the colour currently on the brush. Then the function calls goToPoint() with the column and row indexes to go to the point. Next, it calls DipBrush() to paint the pixel. Additionally, a counter 'painted' increments after DipBrush() is called in the nested for loop. The program can use an if statement to check if the paintbrush requires more paint. If it does, it uses setEncoderX to get to the paint position and then DipBrush() to apply the paint. Samantha wrote this function, with the process being demonstrated in figure 23. The variables are also referenced in table 8.

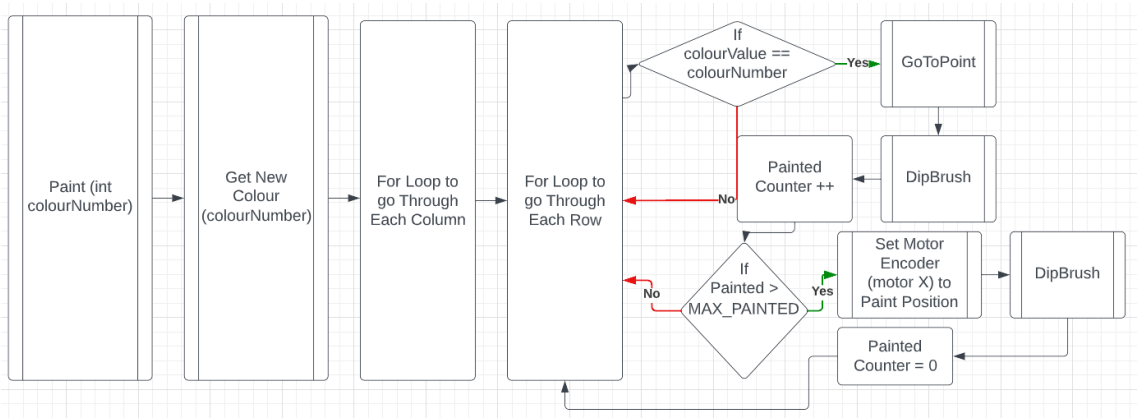

Figure 23: Flow Chart of Paint Function

5.1.4 Data storage

The RobotC program uses a sixteen by sixteen 2D array to store the position and colour of every pixel in the drawing. The program receives the data from the text file "output.txt".

The main variables for this program are also referenced in table 9, with the constants in table 1.

5.2 C++

5.2.1 Summary

As introduced in the subsection above, the RobotC program was written to manipulate the body of The Pixel Painter. Although the C++ program does not control or power the components of the body directly, the program is a crucial aspect of The Pixel Painter. The C++ program allowed a user to create a pixel art image on a sixteen by sixteen grid and with eight colours. This whole process can be visualized with the aid of figures 24, 25 and 26.

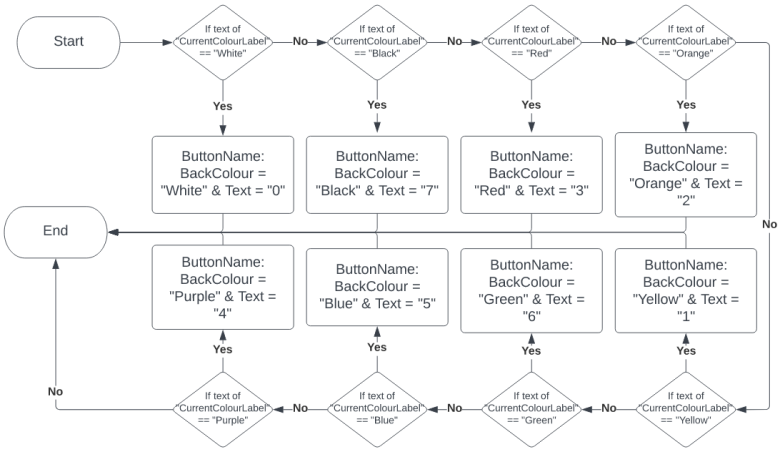

Figure 24: The Colours(B#_#) Function Flow Chart

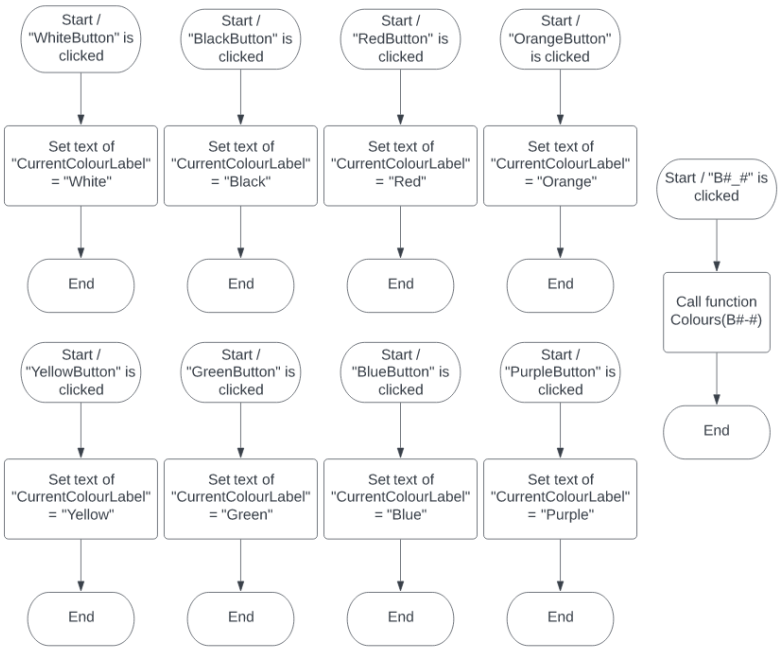

Figure 25: The Button Clicked Events Flow Chart

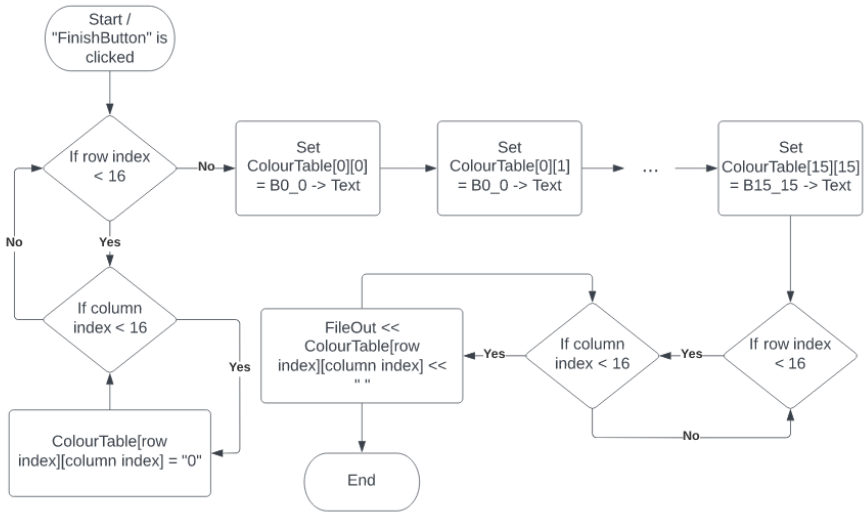

Figure 26: Main Code Flow Chart

5.2.2 Design

The C++ program was written with the IDE Visual Studio Code 2019 and used a GUI application. The GUI contains a button for each colour and pixel. There is also a button to finish the drawing. Every button used for the pixel art image had a default setting of a white back colour and the text that read "0".

5.2.3 Utilization of buttons and labels

The mentioned variables in the C++ subsection can be found in table 10 and the code can be found in Appendix B. When the user clicks a colour button, it changes the label's text to display the colour's name which can be seen in the flow chart 25. This label is then used to determine a pixel button's back colour and text which is shown in the flow chart 24.

e.g. The purple was clicked, and the label's text would display "Purple". The button in the top left corner was clicked, the program will then check what is displayed on the label and read "Purple". The pixel button will now be the back colour of purple and display the text "4".

5.2.4 Data Storage

The C++ program uses a 2D array to store the colour displayed on each button when the user presses the finish button. The array comprises of sixteen columns and sixteen rows that correlate to the canvas the user draws on. Each array index is input with the text displayed on the correlating pixel.

e.g. The button in the top left corner will correlate to the top left index of the 2D array (which is [0][0]). From the example above, the button has the text "4" and in which the array at index [0][0] will be "4". This process can be visualized with the figures below (28 and 27).

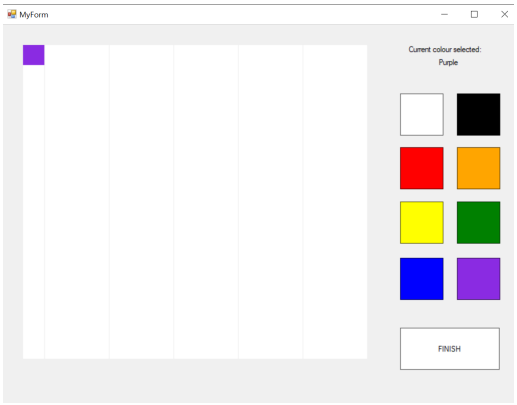

Figure 27: GUI with a Purple Pixel

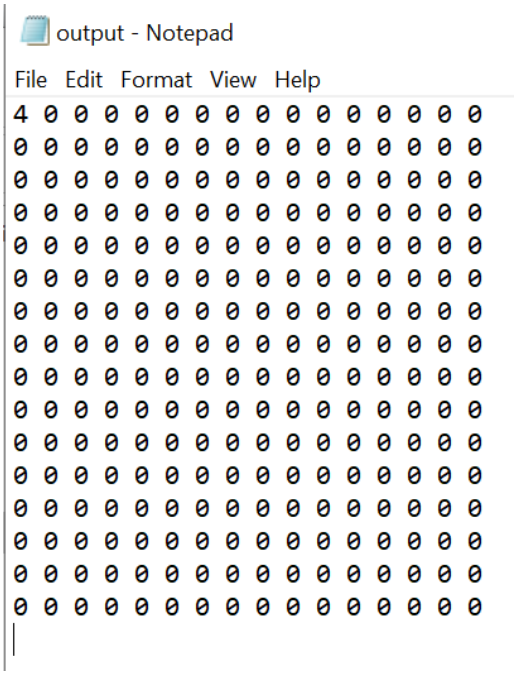

Figure 28: Array with a Purple Pixel

This array is output to a file named "output.txt". The RobotC program then reads the file and uses the data to paint the painting. The flow chart of this process is shown in flow chart 26.

5.3 Variables

The below tables list our variables within the entire robotC program. Table 1 shows every constant used within the program while the rest of the tables show the variables used within each function.

Table 1: Robot C Constants

Function	Type	Name	Value	Description
Global	int	ART_SIZE	16	This defines the size of the array.
Global	int	Array[ART_SIZE] [ART_SIZE]	N/A	This defines the array for the file to write into.
Global	int	zMotorPower	40	This is the speed of the z motor.
Global	int	xMotorPower	60	This is the speed of the x motor.
Global	int	yMotorPower	35	This is the speed of the y motor.
Global	int	cMotorPower	25	This is the speed of the colour motor.
Global	float	CONVERSION	180/PI	This is the degree to radian conversion factor.
Global	float	WATER_POS	46	This is the distance from 0 to the water.
Global	float	TOWEL_POS	39	This is the distance from 0 to the paper towel.
Global	float	PAINT_POS	24.57	This is the distance from to the paint.
Global	int	DIP_PAINT	750	This sets the timer for DipBrush() in paint.
Global	int	DIP_PAPER	850	This sets the timer for DipBrush() on paper.
Global	int	NUM_COLOURS	7	This is the amount of colours the robot can paint with.
setEncoderX	float	XCONVERSION	1.18	This is the conversion factor for the x-axis.
setEncoderY	float	YCONVERSION	2	This is the conversion factor for the y-axis.
WashBrush	float	MOVE_FACTOR	0.5	Sets the increment it moves while dipping the brush on the paper towel.
setEncoderC	float	EQ1	91.9	This is the a value of the quadratic equation to convert from px to cm.
setEncoderC	float	EQ2	105	This is the b value of the quadratic equation to convert from px to cm.
setEncoderC	float	EQ3	0.0536	This is the c value of the quadratic equation to convert from px to cm.
Paint	int	MAX_PAINTED	4	This is the number of times it paints before getting more paint.

Please note that the variables in tables 2, 3, 4, 6, 5, 7, 8 and 9 are the non constant variables with all the constant variables found in 1.

Table 2: setEncoderX, setEncoderY and setEncoderC Variables

Type	Name	Description
float	distance	Is the value in px that the determines the distance the motor needs to go.

Table 3: goToPoint Variables

Type	Name	Description
int	x	Determines the point in the x-axis the brush will move to.
int	y	Determines the point in the y-axis the paper will move to.

Table 4: DipBrush Variables

Type	Name	Description
int	amount	The amount of time the brush will go up or down for.
float	temp	The name of amount within the DipBrush function.

Table 5: WashBrush Variables

Type	Name	Description
int	distance	This determines the distance the brush needs to move when dipping on the paper towel.

Table 6: CheckIfColourExists Variables

Type	Name	Description
int	ColourNumber	The integer value of the colour.
int	RowIndex	The row value which is iterated through in a for loop.
int	ColIndex	The column value which is iterated through in a for loop.

Table 7: getNewColour Variables

Type	Name	Description
int	colour	Tells the code what colour needs to be accessed.

Table 8: Paint Variables

Type	Name	Description
int	painted	The number of pixels painted.
int	c	The column value used in a for loop.
int	r	The row value used in a for loop.

Table 9: RobotC Main Variables

Type	Name	Description
int	RowIndex	The row value which is iterated through in a for loop.
int	ColIndex	The column value which is iterated through in a for loop.
bool	isDirty	Determines if the brush needs to be washed or now.
int	colour	Tells the code what colour is being used.

Table 10: C++ Variables

Type	Name	Value	Description
int const	MAX_CANVAS_SIZE	16	This constant represents a side dimension of the pixel art canvas.
string	ColourTable [MAX_CANVAS_SIZE] [MAX_CANVAS_SIZE]	N/A	This array is used to show the coordinate on a pixel and it's colour.
Label	CUrrentColourLable	N/A	Displays the the colour that the user is currently drawing with.
Button	B#-#	N/A	The #s can be any number between and including 0 and 15. These objects are used to create the canvas on the GUI. When one this is objects is clicked, the function "Colours(B#-#)" will be called which will change the back colour and text displayed on the object.
Button	BlackButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Black".
Button	WhiteBUtton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "White".
Button	RedButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Red".
Button	OrangeButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Orange".
Button	YellowButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Yellow".
Button	GreenButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Green".
Button	BlueButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Blue".
Button	PurpleButton	N/A	When this button is pressed, the label "CurrentColourLabel" changes to "Purple".
Button	Finish Button	N/A	When this object is pressed, the text displayed on every button that creates the canvas (B_) will fill the 2D array "ColourTable".

5.4 Testing

The testing procedure for this robot began with individual tests of each motor to ensure that none of our hardware was faulty. Although this may have been slightly unnecessary, it eliminated several future checks for programming issues.

Once it was certain the hardware worked, testing of the actual code began. Each function was tested individually and modified to ensure it correctly completed its tasks

without any issues.

When all of the functions were verified, dry runs of paintings began. Dry runs entailed the main code being tested without the initialization or shutdown sequences. These tests were run without paint or water to ensure the robot would not make a mess of paint. These tests were run under various conditions to check for consistency and ensured the robot was resilient to any possible issues, including missing the water and paint cups.

In the third testing stage, a pen was attached to where the brush would be, and the code was run using the pen. The pen allowed the team to check the precision and consistency of the robot. These tests found the robot to be relatively accurate and consistent unless the pen got caught on anything.

Once the above tests were completed, and it was unlikely that the robot would immediately fail, testing with paint began. At first, only one colour was used as the plotting was tested with paint. Once that worked tests began with multiple colours. Most tests used a sunset image of four colours, yellow, orange, red and purple. What the sunset was intended to look like can be seen in figure 29.

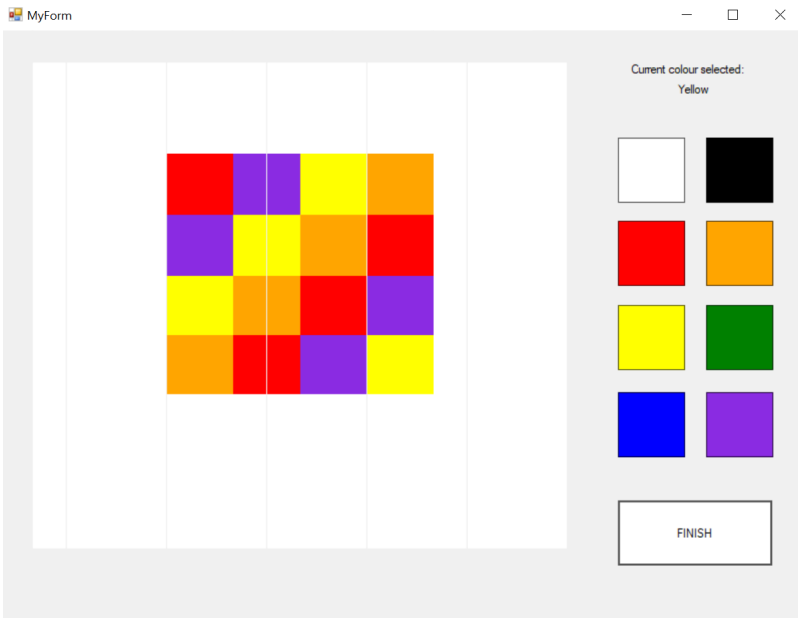

Figure 29: SunSet C++ Program View

Figure 30: Sunset Image 1

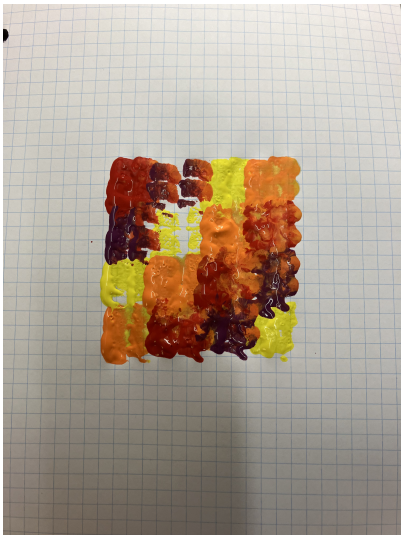

Figure 31: Sunset Image 2

The paintings displayed in figures 30 and 31 are the first two good paintings from the robot of the sunset image. This image was the main testing image used due to its relatively simple nature and its variety of colours. Later versions can be seen in figures 32 and 33.

Figure 32: Sunset Image 3

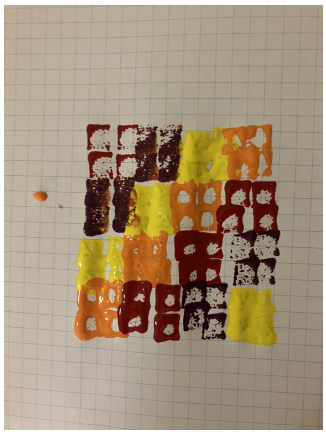

Figure 33: Sunset Image 4

Once the team began running multi-colour tests, other issues began to present themselves. As the brush was easily compressible, it collected too much water in the sponge. The brush was dabbed more times on the paper towel to alleviate this. This did not completely solve the problem but it helped improve it.

Additionally, once multiple colours were introduced, the robot was found to be less accurate than expected in the x-axis when doing small movements (1cm). As a result, the code was changed so that it iterated through the painting column by column rather than row by row. In addition, the team had the brush return to the 0 position in the x-axis between every column, increasing the distance it has to travel to the point in the x-axis. It is important to note that this also had the effect of decreasing the speed of the robot.

5.5 Problems Encountered

Within the testing phase, several problems were found and solved with varying degrees of success.

- Inability to move x-axis
- Overheating of the z-axis
- Inconsistent movement of the paper
- Lack of precision in small x-axis movements
- Too much water in the brush
- Non-linear movement of the paint cups

5.5.1 Inability to move x-axis

In the initial testing phase, the large motor could not move the x-axis in certain circumstances, at which point the motor would skip, or the axle would twist. To solve this, a gear reduction was added to increase the torque of the axis.

5.5.2 Overheating of the z-axis

While testing more complicated paintings, the z-axis motor would overheat. At the time, no solution could be identified, and it was believed that the problem was hardware related. As a result, during the demo day, the robot took constant breaks of 10-15 minutes between each painting to cool the motor down. After the demo day, it was found that the code never stopped the motor from going up once the motor had completed the dipping motion. This change likely solved the issue.

5.5.3 Inconsistent movement of Paper

For some paintings, the paper began to twist partway through the painting, causing twists in the painting itself. Paper twisting most often happened during testing after the robot moved between locations. Via testing, it was determined that the blocks holding down the edges of the paper would move during the move, changing the amount of friction on the page. This would cause one side to have slightly more friction than the other, leading to the page twisting during the painting process. This problem could not be solved as it was found right before the demo.

5.5.4 Lack of Precision

In locations where the x-axis made small movements between 1-2 cm, the robot would often only move part of the distance. Upon further inspection, the team found this was due to give in the gears that helped offset the weight of the brush. As a result, modifications were made to the paint function, so it iterated via columns and then rows. Additionally, the x-axis returned to the zero location between each column to increase the distance the x-axis had to move, increasing the axis's accuracy.

5.5.5 Too much water in the brush

During the washing process, the brush would retain too much additional water, causing subsequent colours to appear watered down. In order to fix this, the number of times the brush dabbed the paper towel was increased. Although this fixed part of the issue, the robot still had this issue as it reached the later colours, more specifically blue, green and black. As a result, during paintings with more paint colours, the paper towel was required to be changed partway through the painting.

5.5.6 Non-linear movement of the paint cups

During initial testing of the paint cups, it was found that although the paint cups were placed a consistent distance from each other on the belt, the belt moved in a non-linear fashion. To solve this, the team measured the belt at several spots, with both distances in cm and encoder value measured. Using a google sheet, these values were plotted where the following equation was found. The equation replaced the cm to px conversion previously used to get the paint cups to the correct location, along with the touch sensor.

6 Verification

System Operation:

Start up demonstrated

- Y-axis moves until the up button is pressed
- Displays initialization instructions
- Moves the paintbrush to the start position(top left)
- Displays the second set of initialization instructions
- Begins main tasks by pressing the enter button

Regular tasks demonstrated

- Go through painting in one colour, looping through each colour
- For colour, get the correct colour on the paintbrush
- Go to each point and paint it
- If the paintbrush is running out of paint, get more paint
- Once finished going through all pixels for that colour, wash and dry brush.

Shut down procedure demonstrated

- Wash brush
- Push out paper
- Stop all motors
- Display painting complete string

The following motors must also work as expected in the following tasks:

- xMotor moves the sponge left and right
- yMotor moves the paper forward and back
- colourMotor moves the colours back and forth
- zMotor moves sponge up and down

Our team was able to meet the above constraints and create a reasonably accurate robot.

6.0.1 Start-up

Once the program begins, the y-axis motor runs in the positive direction until the up button is pressed. This allows the paper to be easily inserted into the system. Then a secondary initialization string is displayed instructing the user to move the x-axis to the starting zero position, the z-axis to the up position and to align the yellow paint cup with the touch sensor. Once the enter button is pressed, the main tasks begin.

6.0.2 Regular-Tasks

Using nested for loops, the program iterates through each column and row of the painting. Based on this, the program uses the array to determine if the pixel contains the colour currently on the brush. If it does, the robot dips the brush and continues to the next pixel.

Because the startup initialization procedure gets all the critical components in the correct location before beginning, the regular tasks can run accurately and require very little verification and assistance. A for loop also runs to ensure that after a set value of pixels are painted, the paintbrush returns to get more paint. After iterating through the pixels for that colour, the program uses predefined distance locations to wash and dry the brush using the xMotor and the washBrush function. Then, the program repeats for the following colours.

6.0.3 Shut Down Procedure

The yMotor runs to push out the completed painting. Finally, a shutdown message is output to the display.

Additionally, the constraint of lack of precision was explained in the 'Problems Encountered' section above. Due to non-linear movement and inconsistency in the movement of all motors, the robot has to iterate through the painting via columns before rows and bring the paintbrush back to the zero location for re-calibration before going to the following pixel location.

Since the preliminary report, the method of input was changed. The team initially planned on using OpenCV and computer vision to detect the location and colour of each pixel in a user-inputted image. Due to issues downloading the library, the team switched to a different method of creating the array. The new version of input was programmed using C++ and a GUI-like interface.

In conclusion, as explained above this robot was able to meet each of the constraints used to demo the project.

7 Project Plan

The first iteration of the project plan was to create a robot to paint the Mona Lisa. Unfortunately, this proved to be too broad of a project idea. In order to have a usable plan, the project depth had to be decreased to be more specific, measurable, and attainable. The plan was to create a mechanical chassis similar to a single-colour plotter mechanism and to use OpenCV to get an image from the internet, analyze it to get the colour of each pixel, and then output that to an array. The RobotC program would then use that as an input to know the location and colour of each pixel it should paint.

The project began with the build of the chassis which was done by Samantha and Melissa. The majority of the system were completed in around three weeks. Parallel to this, Kathryn and Olivia began working on an Open CV program but had difficulties installing the library. Unfortunately, even after requesting aid from the TAs and Emily,

the header file was still running errors. Due to scheduling issues with Nassar and Olivia getting COVID, the programming team was unable to run Open CV. Kathryn had previous experience with GUI's from using Java in high school. She created a program where the user can create a pixel art image. The C++ program then outputted the location and colour of the pixel to an array.

The plan was to use the array as input for the robot to know where to move. The mechanical and software systems were completed around the same time, so thus testing began. The focus shifted to testing each component before testing and troubleshooting the system. The plan was originally to be able to paint on a 16 by 16 cm grid, but due to complications with the motor overheating and time constraints, the team decided to limit the painting size to 8 by 8 or 64 pixels. Larger paintings could be completed; however, the program had to be paused partway through to allow for the cooldown of the z-motor. Additional problems, such as lack of precision on the paintbrush and non-linear movement, delayed testing.

In conclusion, although the project plan was incomplete at the beginning of the build, and there were multiple mechanical and software issues, the team followed the project plan and timeline to finish the project accurately. This allowed for success in all tasks and the project as a whole.

8 Conclusions

This project aimed to create a robot capable of taking an input pixel art image and recreating that image onto a standard piece of paper. The design met the criteria stated in the report, but changes could have been made for optimal efficiency, accuracy, and precision. The mechanical system was based on a piece of paper, with motors used to move the paper, brush, and colours directly around it. The software takes a user-generated pixel art image from a C++ program and outputs the array of pixel locations and colours. This allowed the RobotC program to iterate through the painting to paint the correct spot on the paper. Predefined locations were used for the location of the water cup, paint cups, and paper towel, as well as a zero location for verification of location.

The two final images printed on the robot where a flower and a whale.

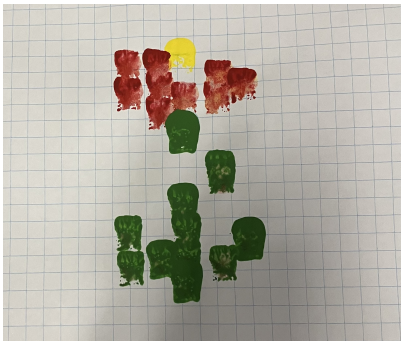

Figure 34: Flower Painting

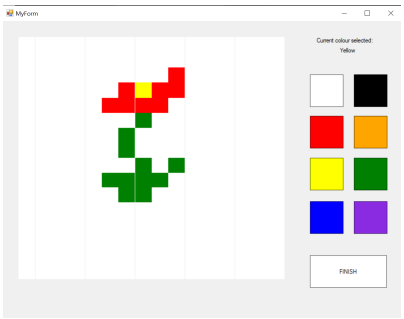

Figure 35: Flower C++ Program

The results of the flower can be seen in figure 34 beside figure 35, the C++ programs image. The results of the whale can be seen in figure 36 beside figure 37, the C++ programs image.

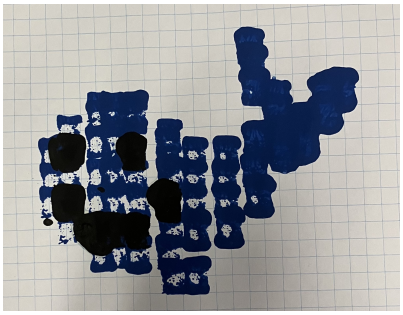

Figure 36: Whale Painting

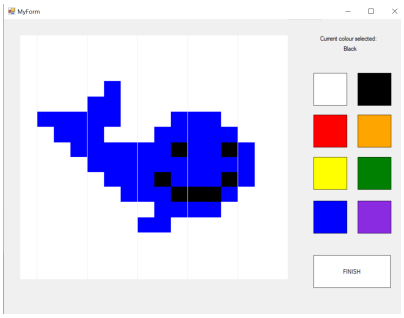

Figure 37: Whale C++ Program

In conclusion, both the software and mechanical systems were very complicated. However, as highlighted in the report, the team was able to accomplish each of the tasks and constraints described and create a working pixel painter robot.

9 Recommendations

9.1 Mechanical Design

This project could be redesigned in the future to be made out of materials such as wood, acrylic, and 3D printed parts rather than Lego. These materials would improve the accuracy of the design as they would tighten the tolerances of the project, reducing room for play within the system and increasing the accuracy of the brush. Smaller tolerances also allow the robot to be moved without the entire system needing to be re-calibrated after.

Additionally, a new design for the colour axis would need to be created in the future. The current design is spatially inefficient, with only half the space used at any given time. This could be reduced by putting the paint on a cart, allowing for more paint expansion, as the current design only allows for 8.

Although the brush was able to be washed in future versions of this project, it would be helpful to add a mechanism which would be able to squeeze excess water from the brush. Melissa created a prototype for this; however, the team determined that it was unnecessary in this version.

9.2 Software Design

In a future version, the C++ program would be created using classes rather than individually making each button. This would increase the program's efficiency and simplify the code making it easier to expand for a larger drawing.

Additionally, it would be helpful to rewrite the robot C code for efficiency to remove unnecessary lines of code or functions. During this process, it would also be helpful to add slightly more complicated code in areas such as the movement of the axes to compensate for their slightly non-linear movement.

10 References

[1] 9to5Answer, "[solved] C++ expression must have pointer-to-object type," 9to5Answer, 04-Jan-2020. [Online]. Available: https://9to5answer.com/c-expression-must-have-pointer-to-object-type. [Accessed: 06-Dec-2022].
[2] Amish ProgrammerAmish Programmer 2, ET al., "C++ .net convert system::string TO std::string," Stack Overflow, 01-Oct-1956. [Online]. Available: https://stackoverflow.com/questions/1300718/c-net-convert-systemstring-to-stdstring. [Accessed: 06-Dec-2022].
[3] Dotnet-Bot, "Button class (system.windows.forms)," (System.Windows.Forms) — Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/api/system.windows.forms.button?view=windowsdesktop-7.0. [Accessed: 06-Dec-2022].
[4] "Installing mingw tools for C/C++ and changing environment variable," GeeksforGeeks, 08-Dec-2021. [Online]. Available: https://www.geeksforgeeks.org/installing-mingw-tools-for-c-c-and-changing-environment-variable/. [Accessed: 06-Dec-2022].
[5] Mattwojo, "Microsoft Powertoys," Microsoft PowerToys — Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/windows/powertoys/. [Accessed: 06-Dec-2022].
[6] R. A. S, "How to create a C++ GUI application using Visual Studio?: Simplilearn," Simplilearn.com, 21-Oct-2022. [Online]. Available: https://www.simplilearn.com/tutorials/cpp-tutorial/cpp-gui. [Accessed: 06-Dec-2022].

11 Appendix A - RobotC Code

```
#include "PC_FileIO.c"

int const ART_SIZE = 16;
int Array[ART_SIZE][ART_SIZE];

int const zMotorPower = 40;
int const xMotorPower = 60;
int const yMotorPower = 35;
int const cMotorPower = 25;

const float CONVERSION = 180/PI;

float const WATER_POS = 46;
float const TOWEL_POS = 39;
float const PAINT_POS = 24.75;

int const DIP_PAINT = 750;
int const DIP_PAPER = 850;

int const NUM_COLOURS = 7;

void setEncoderX(float distance) //distance is in cm
{
    float const XCONVERSION = 1.18;

    distance = (distance*(CONVERSION)/XCONVERSION);
    if(nMotorEncoder[motorA] < distance)
    {
        motor[motorA] = xMotorPower;
        while (nMotorEncoder[motorA] < distance)
        {}
        motor[motorA] = 0;
    }
    else
    {
        motor[motorA] = -xMotorPower;
        while (nMotorEncoder[motorA] > distance)
        {}
        motor[motorA] = 0;
    }
} //Moves x axis to distance

void setEncoderY(float distance) //distance is in cm
{
    const float YCONVERSION = 2;

    distance = (distance*(CONVERSION)/YCONVERSION);
    if(nMotorEncoder[motorB] < distance)
    {
        motor[motorB] = yMotorPower;
   while (nMotorEncoder[motorB] < distance)
   {}
   motor[motorB] = 0;
    }
```

```
    else
    {
   motor[motorB] = -yMotorPower;
   while (nMotorEncoder[motorB] > distance)
   {}
   motor[motorB] = 0;
    }
} //Moves y axis by distance


void goToPoint(int x, int y)
{
    setEncoderX(x);
    setEncoderY(y);
} //moves to corresponding point on the paper

void DipBrush(int amount=1000)
{
    float temps = amount;
    clearTimer(T1);

    while(time1[T1]<temps)
    {
        motor[motorC] = -(zMotorPower);
    }

    clearTimer(T1);
    while(time1[T1]<temps)
    {
        motor[motorC] = zMotorPower;
    }
}

bool CheckIfColourExists(int ColourNumber)
{
    //goes through each pixel and checks if the colour exists
    for (int RowIndex = 0; RowIndex < ART_SIZE; RowIndex++)
    {
        for(int ColIndex = 0; ColIndex < ART_SIZE; ColIndex++)
        {
            if(Array[RowIndex][ColIndex] == ColourNumber)
                return true;
        }
    }

    return false;
}

void WashBrush ()
{
    float const MOVE_FACTOR = 0.5;

    setEncoderX(WATER_POS);
    DipBrush();
    DipBrush();

    for (int i = 1; i < 7; i++)
    {
```

```
        int distance = TOWEL_POS-(MOVE_FACTOR*i);
        setEncoderX(distance);
        DipBrush(DIP_PAPER);
    }
}

void setEncoderC(float paintNum)
{
    float const EQ1 = 91.9;
    float const EQ2 = 105;
    float const EQ3 = 0.0536;

    float distance = (-EQ1 + EQ2*paintNum + EQ3*(paintNum*paintNum));
    //used a google sheet to get the line equation as it wasn't perfectly linear
        which was causing issues
    if(paintNum == 1)
    {
        return; //the above equation doesn't properly account for the first paint
            container so this accounts for that
    }
    if(nMotorEncoder[motorD] < distance)
    {
        motor[motorD] = cMotorPower;
        while (nMotorEncoder[motorD] < distance)
        {}
   motor[motorD] = 0;
    }
    else
    {
        motor[motorD] = -cMotorPower;
   while (nMotorEncoder[motorD] > distance)
   {}
   motor[motorD] = 0;
    }
} //Moves y axis by distance

void getNewColour(int colour)
{
    //Runs paint belt until desired colour is reached
    setEncoderC(colour);
    while (SensorValue[S1] != 1)
    {
        setEncoderC(colour+0.25);
        if(SensorValue[S1] != 1)
        {
            setEncoderC(colour-0.4);
        }
    }

    //gets paint on the brush
    setEncoderX(0);
    setEncoderX(PAINT_POS);
    DipBrush(DIP_PAINT);
}

void Paint(int colourNumber)
{
    int painted = 0;
```

```
    int const MAX_PAINTED = 4; //need to test this value

    int colourValue = 0;

    getNewColour(colourNumber);

    for(int c = 0; c < ART_SIZE; c++)
    {
        for(int r = 0; r < ART_SIZE; r++)
        {
            colourValue = Array[r][c];
            if(colourValue == colourNumber)
            {
                goToPoint(c, r);
                DipBrush(DIP_PAPER);
                painted++;

                if(painted == MAX_PAINTED)
                {
                    setEncoderX(PAINT_POS);
                    DipBrush(DIP_PAINT);
                    painted = 0;
                }
            }
            setEncoderX(0);
        }
    }
} //itterates through colour and paints

task main()
{
    displayString (2, "Beginning initialization process");
    displayString (4, "Insert paper and press up button");
    displayString (5, "when done paper is in place");
    motor[motorB]=(yMotorPower/2);
    while (!getButtonPress(buttonUp))
    {}
    motor[motorB]=0;
    displayString (6, "Ensure yellow paint cup is");
    displayString (7, "touching the touch sensor");
    displayString (8, "Push paint bush against right");
    displayString (9, "endstop");
    displayString (10, "Ensure bush is in the up position");
    displayString (12, "Press center button when ready");
    displayString (13, "to continue");

    nMotorEncoder[motorA]=nMotorEncoder[motorB]=
    nMotorEncoder[motorC]=0;

    setEncoderX(PAINT_POS);
    while (!getButtonPress(buttonEnter))
    {}

    //initialization of motors zero position
    TFileHandle fin;
    bool fileOkay = openReadPC(fin,"output_txt.txt");

    for (int RowIndex = 0; RowIndex < ART_SIZE; RowIndex++)
```

```
    {
        for(int ColIndex = 0; ColIndex < ART_SIZE; ColIndex++)
        {
            readIntPC(fin, Array[RowIndex][ColIndex]);
        }
    }
    bool isDirty = false;
    for(int colour = 1; colour <= NUM_COLOURS; colour++)
    {
        if (CheckIfColourExists(colour) == true)
        {
            Paint(colour);
            isDirty=true;
        }

        if (isDirty==true)
        {
            WashBrush();
            isDirty=false;
        }
    } //paints each colour

    displayString (2, "Painting Complete");
    displayString (4, "Painting Dispensing");

    setEncoderY(-20); //dispences paper out of system
    setEncoderX(0);
    displayString (6, "Painting Dispensed and Tasks Complete");
}
```

12 Appendix B - C++

```
#include "MyForm.h"

using namespace System;
using namespace System::Windows::Forms;
[STAThread]

void main(array<String^>^ args)
{
        Application        ::EnableVisualStyles();
        Application        ::SetCompatibleTextRenderingDefault(false);
        Project2        ::MyForm form;
        Application        ::Run(% form);
}
```

The MyForm.h file was too large to include in this document so it can be found separately in the document labelled MyForm.h.