Can You Feel My Sign? Sending shapes to a haptic display

KIONA BIJKER, Universiteit Twente, Netherlands

Gentle stroking touches are considered highly relevant in social interactions. When loved ones are apart they cannot express or receive these touches. In 2021 C.Stork developed a haptic sleeve to mediate such affective touches during a video or phone call [7]. This haptic display was limited to straight strokes. In this article the development of an algorithm will be discussed that simplifies user input shapes to straight lines. This adds more expressiveness to the sleeve by allowing user drawn shapes to be simulated. The sleeve has been adjusted during this project based on suggestions from F. Tang et al. [9], these adjustments will also be discussed.

CCS Concepts: • **Human-centered computing → Interactive systems and tools**; • **Mathematics of computing →** *Numerical analysis.*

Additional Key Words and Phrases: haptic display, mouse-movement, shape approximation, Tactile Brush, affective touch

ACM Reference Format:
Kiona Bijker. 2023. Can You Feel My Sign? Sending shapes to a haptic display. In . ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Gentle stroking touches are considered highly relevant in social interactions. When loved ones are apart they cannot express or receive these touches. In 2021 C.Stork developed a haptic sleeve to mediate such affective touches during a video or phone call [7]. The design was based on the TaSST by G. Huisman et al. [4] together with the Tactile Brush algorithm as defined by A. Israr et al. [5]. Stork's tactile display was limited to straight strokes. This severely limits the expressiveness of the tactile display. Within this paper the aim is to extend the sleeve with shapes for more expressive communication, this is reflected in the main research question:

RQ How to send shapes from user input to the haptic display?

To do this an algorithm needs to be developed to turn the shapes input by the user into straight strokes that can be simulated by the haptic sleeve. This algorithm will have to be real-time enough to allow for uninterrupted drawing by the user. This leads to the first sub-question of this paper:

SQ 1 How to design an efficient algorithm that translates shapes into polygons?

The polygons the algorithm sends to the sleeve need to be compatible with the sleeve. The polygons the haptic sleeve simulates also need to be recognisable to the user as otherwise the mediation of the affective touch is inaccurate. This leads us to the second sub-question of this paper:

SQ 2 What are the requirements from the sleeve to display a polygon that can be recognised by a user?

This second question can be divided into several sub-questions:

SQ2.1 What is the delay between sending a vector to the Haptic Sleeve and the vibration motor output?[1]

SQ2.2 What are the minimum size and duration of a stroke for the sleeve?

SQ2.3 At what resolution can the user no longer accurately determine the figure they feel? [2]

SQ2.4 What is the difference between various shapes in the needed resolution to accurately recognise the shape, if any? [2]

Within this article the working principle of the tactile display of the sleeve will be discussed in section 2. This lays the basis for answering SQ2.1 and SQ2.2 in section 3 where the sleeve is discussed in more detail. From there we move to answering SQ 1 in section 4. SQ2.3 and SQ2.4 remain unanswered as an experiment was designed but could not be conducted within the required time frame. This experiment will be discussed in section 6. Efficiency of the algorithm will be discussed in section 7 with a reasoning for the approximate workload and working memory load of the algorithm, as a full complexity analysis was outside the project scope. Within this article the words vector and line are used interchangeably to mean the visual representation for a straight stroke.

ACKNOWLEDGMENTS

I would like to thank my supervisor, dr. Angelika Mader for her support and faith in me. I would like to thank prof. dr. Marc Uetz and prof.dr.ir. Bernard J. Geurtz for their explanations and help regarding algorithmic complexity and efficiency. I am grateful for the support from dr. Mariet Theune and my research track colleagues in navigating this bachelor project. Last but certainly not least I would like to thank my friends and family. In particular I would like to thank Erik for being a great bug catcher, my parents for their motivational support, and my fiance for taking the time to hear me complain about the latest bug. Without these people I would not have gotten where I am today.

2 WORKING PRINCIPLE OF THE USED HAPTIC DISPLAY

A haptic illusion is a touch related illusion. For this project two haptic illusions were used: the phantom tactile sensation and apparent tactile motion. These illusions were used to create the Tactile Brush algorithm [5].

2.1 Phantom tactile sensation

The phantom tactile sensation is also known as the funneling illusion. If two vibro-tactile actuators are placed in close proximity on the skin and activated the subject will experience a single vibro-tactile actuator between the two physical actuators. This phantom actuator's location depends on the ratio between the intensities

TScIT 38, February 3, 2023, Enschede, The Netherlands

[1] A large delay between strokes may lead the user to only recognise separate lines rather than a coherent picture

[2] These questions remain unanswered

of the two physical actuators, while the intensity depends on the absolute intensity of both physical actuators [5].

2.2 Apparent tactile motion

Apparent tactile motion is an illusion created by activating vibro-tactile actuators in close proximity in succession with overlapping times. The subject then experiences this as a single actuator moving between the two physical actuators [5].

2.3 Extension to 2D grid

Both aforementioned illusions were corroborated using line-based tactile displays. These findings were used to extend to a 2D grid display. By combining the apparent tactile motion and funnelling illusion within a 2D grid of vibro-tactile actuators, an illusion of a stroke can be generated in any direction within the grid [5] [3].

3 THE HAPTIC SLEEVE

Within this project a haptic sleeve was used with vibro-tactile actuators in a 2D grid formation. This sleeve was developed by C. Stork in a previous project [7] based on the TaSST by G. Huisman et al.[4].

3.1 Original sleeve

The original sleeve consists of a flexible 3D printed grid with vibro-tactile actuators. These actuators are controlled by a Raspberry Pi using two i2c multiplexers. The activation times, intensities and duration are determined using the implemented Tactile Brush algorithm. This was implemented using Python. The sleeve could simulate a straight stroke starting and ending on the lines of the vibro-tactile grid. This stroke to simulate can be sent to the sleeve using an MQTT protocol [7]. A more in-depth guide on the usage of the sleeve can be found in Appendix A.

3.1.1 Basic workings. The sleeve connects to the MQTT server to receive a message with a stroke. It then disconnects and uses the Tactile Brush to create an actuator schedule. Once this schedule is executed the sleeve again connects to the server to receive the next stroke. The duration between receiving the message and the start of tactile output depends on the length of the stroke as a longer stroke requires more scheduled actuators.

3.1.2 Tactile Brush. The Tactile Brush algorithm uses the phantom tactile sensation and apparent tactile motion illusions to simulate a stroke using vibro-tactile actuators. Given a stroke from point A to point B with a certain distance and duration, the Tactile Brush can generate a schedule for vibro-tactile actuator activation. This includes activation time, activation duration and intensity. To determine these variables a number of formulas are used which can be found in the article by A. Israr et al. [5]. For this project the formulas of interest are the intensity formulas used to create a funneling illusion v with intensity A_v between actuators 1 and 2:

$$A_1 = \sqrt{1-\beta} * A_v, A_2 = \sqrt{\beta} * A_v$$

Here A_1 is the intensity for actuator 1, A_2 the intensity for actuator 2. β is the ratio of the distances from v to 1 and from v to 2 [5].

3.1.3 MQTT. MQTT is a communication protocol that consists of a publisher, a broker, and a subscriber. At the broker there is a list of topics messages can be published to. When the publisher sends a message to a for a specific topic every subscriber to that topic is forwarded that message [1].

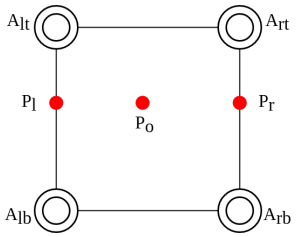

Fig. 1. A visual representation of the adjusted sleeve algorithm

3.2 Code adjustments

The sleeve as developed by C. Stork was limited in stroke start- and endpoints. To allow for start- and endpoints of a stroke to be anywhere in the grid instead of only on grid lines the algorithm for the sleeve needed to be adapted. The adaptation was based on a paper by F. Tang et al. [9]. The Tactile Brush algorithm creates virtual actuators representing phantom tactile sensations on the grid lines using a number of formulas for the intensity, activation time and activation duration. This is what allows the Tactile Brush to simulate strokes starting and ending between physical actuators [5]. Tang et al. suggest adding an abstraction to allow start- and endpoints to be anywhere within the grid. This abstraction applies the formulas used by the Tactile Brush team a second time. The starting point between grid lines is simulated using two virtual actuators on the grid lines. These two virtual actuators are simulated by the physical actuators on the same lines [9]. As an example take P_o which is a virtual actuator between grid lines. This example is illustrated in Figure 1. P_l and P_r are virtual actuators created on the grid lines to allow for the illusion of P_o. P_l and P_r are not physical actuators, they are simulated using the physical actuators. In this case A_{lt} and A_{lb} for P_l and A_{rt} and A_{rb} for P_r. This way by activating A_{lt} , A_{lb} , A_{rt} , and A_{rb} the funneling illusion allows for the illusion of P_o.

4 ALGORITHM DEVELOPMENT

The goal of the algorithm is to extend the expressiveness of the touch communication as mentioned in the Discussion and Recommendation section of C. Stork's paper on the development of the sleeve [7]. To achieve this the algorithm will simplify the drawing into straight strokes and send these to the sleeve.

4.1 Existing mouse-movement to vector algorithms

Literature research was conducted to find algorithms that approximated computer-mouse movement using vectors. While this search was by no means exhaustive, no articles were found on real-time computer-mouse movement approximation using vectors. Most articles that turned up in the computer-mouse movement search referenced predicted computer-mouse movement towards specific targets either on or off the screen. For examples see [6] [8].

4.2 Requirements

To develop an algorithm the requirements need to be identified. The algorithm needs to be responsive enough to allow for unobstructed drawing on the screen. This is needed for the near real-time requirement as mentioned in the introduction. Since the sleeve works over MQTT the algorithm will need to send MQTT messages that the sleeve can process without errors. A given requirement is to use the Processing language, which is geared towards user interaction. Processing is designed as a programming sketchbook [2] with basic methods that make graphic-based user interaction easily accessible. This makes the language a good fit for the development of this user-made drawing to polygon algorithm. To summarise, the algorithm should:

- Take computer-mouse-like input
- Be responsive enough to allow for unobstructed drawing
- Communicate with the haptic sleeve over MQTT
- Interface with the haptic sleeve without errors
- Send correct messages to the haptic sleeve for functional haptic sleeve output
- Be implemented in Processing

4.3 Early iterations

There were various iterations before the algorithm described in this paper was developed. In this section the two main versions before the final algorithm will be discussed.

4.3.1 Averaging angles. The first version of the algorithm was based on an intuition from sketching. The program would create a line for the mouse movement between each frame. These separate lines would then be averaged in angle, whilst adding the lengths, to create a line that approximates the total mouse movement. To determine when a new line should start the end point of the line would be compared to the current mouse position. If the end of the averaged line and the actual mouse position were more than a certain distance apart a new line would be started. This algorithm slowed down significantly as the run time progressed. Next to that the distance based cut-off made it difficult to predict how the program would react to various drawing speeds. The lines produced were of varying accuracy as the drawing speed had more influence over the accuracy than the degree to which the line approximated the drawing.

4.3.2 Adding angles. A second version of the algorithm is closely related to the final version. In this algorithm the mouse movement between the first two frames is taken as the initial line. Then for each frame the difference in angle between the new movement and the existing line is added to the angle of the existing line. For the length of the line the distance of the mouse-movement is added. This algorithm results in a line that follows the mouse on screen and does not slow down as run time progresses. To determine whether a new line should be started the difference in angle between the existing line and latest movement were compared to a cut-off angle difference. This made the algorithm vulnerable to unnecessarily short vectors especially at lower drawing speeds. Next to this the achieved line was very similar to a line drawn from the starting point, usually where the first click happened, to the current mouse position. This suggested there may be a more efficient way of achieving the same line.

4.4 The developed algorithm

The final version of the algorithm is significantly simpler than the previous iterations. Upon the start of a drawing or new line the start x and y positions of the mouse are saved. For each frame a new line is constructed from the start position to the current mouse position and the current position is saved. Every n frames a check is performed. This checks if the line for the last n frames and the line from the start position to the current position differ more in angle than a maximum angle a. If the angle difference is bigger than a, a new line is started and the line from the last check is sent to the sleeve. The new line will have the mouse position of n frames ago as the start position. If the angle difference is smaller than a, the program continues with the current start position for the line. The full pseudo-code for the algorithm can be found in the appendix in subsection B.4.

4.5 Implementation

The algorithm was implemented using Processing extended by Java elements. Processing was chosen for its focus on user interaction as also described in subsection 4.2. Java was chosen both for being based on the Java PApplet class and developer familiarity. For the main program Processing was used. MQTT was implemented using Java and the org.eclipse.paho.client.mqttv3 library. Next to this three new exceptions were made in Java:

- ImproperPointSaveException: Thrown when the X and Y lists for saved mouse positions are not the same length. Recovery is attempted by dropping the last item of the longer list, if unsuccessful the program stops using a RunTimeException.
- NotEnoughDistanceException: Thrown when the vector to be sent is too short for the sleeve. No recovery, vector is not sent. The program continues.
- NotEnoughDurationException: Thrown when the vector's duration is too short for the sleeve. No recovery, vector is not sent. The program continues.

For the duration and distance exceptions the minimum variables are set at program initialisation. These minimum values will depend on the used haptic display and the algorithm used to create haptic illusions. The minimum duration is greater than or equal to the minimum actuator activation duration. A stroke with a shorter duration than this will not allow the actuator to reach the needed intensity to simulate it. The minimum distance is the distance at which the user will perceive a stroke rather than pressure at a single point. The final program works on PC with any computer-mouse-like input. It was tested both on Ubuntu 22 and Windows 11. As both Processing and Java are available for Mac-OS and no operating system commands were used, the program is assumed to work on Mac-OS as well.

5 VERIFICATION

To verify that the algorithm works as intended a number of trial runs have been conducted. Within these trial runs a computer-mouse as well as a drawing tablet connected to a computer were used to draw

various shapes. See Figure 2 for some of the trial run outputs on the graphic side. This shows that the algorithm successfully breaks up the lines drawn by the user into straight lines, which can be simulated as strokes. When sending these strokes to the sleeve the test run showed no errors and the actuators activated as intended. The delay between the drawing and the activation of the motors is approximated to be a second long. It is unknown whether this delay is caused by the implementation of the algorithm or the sleeve or caused by external factors such as network speed.

Fig. 2. Graphical outputs from the algorithm

6 EXPERIMENT DESIGN

The experiment is designed to answer the following questions:

SQ2.3 At what resolution can the user no longer accurately determine the figure they feel?

SQ2.4 What is the difference between various shapes in the needed resolution to accurately recognise the shape, if any?

These questions can be answered through a short user test. Within this test various shapes would be given to participant with varying accuracy. The participant is then asked to determine the shape they received. After the experiment the data should be anonimised to protect participant privacy. This experiment could not be conducted due to project time constraints. The designed procedure is described in subsection 6.1.

6.1 Procedure of the experiment

(1) The participant receives the haptic sleeve and is asked to put it on. Any required help is provided.
(2) The participant is asked whether the sleeve is comfortable and any needed adjustments will be made to make the sleeve fit as comfortably as possible.
(3) The researcher sends a set of predetermined shapes to the sleeve with various numbers of vectors for each shape. After each shape the participant is asked what shape they think they felt (without options given), if any.
(4) Once the set of shapes is completed the participant will be given a questionnaire to asses how comfortable the use of the sleeve was.

7 COMPLEXITY AND EFFICIENCY

A full analysis of the complexity and efficiency of the algorithm was not possible within the time frame of this project. However, one can reason about a general complexity. The goal was to reach real-time complexity. In this section the reasoned complexity for the implemented functions will be discussed, a full argumentation can be found in Appendix C. These functions are ranked based on reasoned complexity and the number of calls to the function. This ranked list will indicate for which functions an improvement in complexity will impact the overall time for the algorithm the most.

7.1 Reasoned complexity

The algorithm takes a mouse position per frame as input, this makes input n unlikely to grow. Next to this not all tasks are performed for every frame. To allow for comparing of task costs the approximate workload per second will be considered. This allows for a comparison between tasks that takes both per-call workload and call frequency into account. For tasks with an unknown frequency, such as the repairing of lists, the workload per call will be multiplied with "numberOfCalls". Only code written within this project will be considered. This excludes functions from Processing and the used libraries. Environmental variables outside code control, such as network delay, are also excluded.

In the ranked list as seen in Table 1 it is assumed that a high accuracy and high number of frames per second are desired. The functions are ranked from most to least expensive regarding these variables. For readability frames per second is shortened to fps.

7.2 Profiler

Due to the way the program was designed it is difficult to test for the same input multiple times with limited changes. To still allow for some analyses of the trial runs a profiler was used. The used profiler was the built-in 'Intellij Profiler' from Intellij by JetBrains. Within the profiler flame-graph and method call tree the program 'shapeToVector', which is the project program, takes up approximately half of the total run time. This section's run time is mostly the draw() function, which according to the reasoning on the complexity is the most costly. A surprising cost is found in the setup() function which initialises all variables and the MQTT for the Processing sketch. Upon closer inspection it was found that the MqttClient constructor from the eclipse.paho.client.mqttv3 library is the main contributor to that cost. This is illustrated in one of the profiler outputs as seen in Figure 3.

8 DISCUSSION

8.1 Limitations of the project

Many of the research questions within this project were unknown at the start. This resulted in an expanding project. It gives many opportunities for further research, as not all questions could be answered within this project's allotted time frame.

8.2 Limitations of the algorithm

This algorithm has limited accuracy to the drawings made by the user as it works on a frame-by-frame basis. Next to that the accuracy is limited by the real-time nature of the algorithm. An algorithm that simplifies the finished drawing would likely be more accurate than the algorithm developed in this project. However, only sending finished drawings would make the characteristics of the sleeve closer

Table 1. Functions ranked from most to least expensive regarding approximate workload

Rank	Task	Approximate relative workload per second	Approximate working memory load
1	Drawing a frame	$x * (numberOfSavedVectors + 1) * fps$	$6 * (numberOfSavedVectors) + 4$
2	Updating *currentVector*	$(y + 2) * fps$	$6 + 2 * PVector$
3	Resets	$(10 + w) * (fps \div checkFrame)$	$8 + 2 * PVector$
4	Accuracy checks	$(z + 1) * (fps \div checkFrame)$	$6 + 2 * PVector$
5	Recovering after a failed check	$(6 + newPVector) * (fps \div checkFrame)$	$6 + PVector + 2 * checkFrame$
6	A passed check	$(v + 2) * (fps \div checkFrame)$	$2 + Pvector$
7	Sending a vector	$(5 + u) * (fps \div checkFrame)$	$13 + (numberOfSavedVectors * 4) + String$
8	Repair of broken position list	$(8 + FloatList.remove) * numberOfCalls$	$2 + 2 * (checkFrame + 1)$

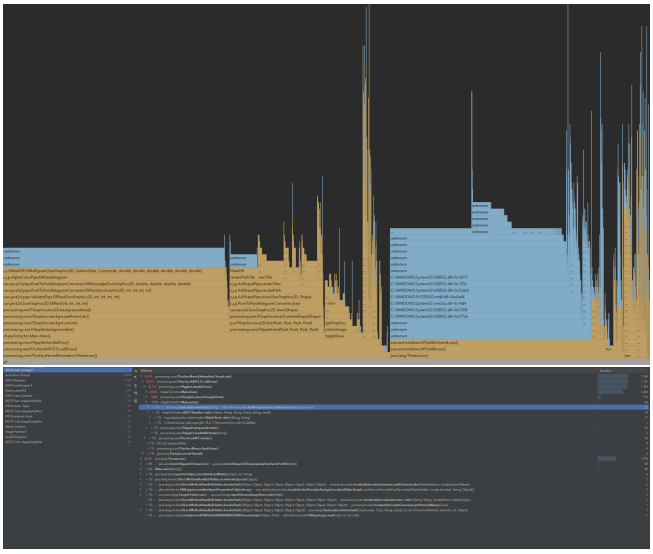

Fig. 3. The flame-graph and method call tree of one of the profiler runs

to an instant messenger rather than a real-time display to be used next to phone or video calls.

8.3 Limitations of the hardware

The grid used in the haptic sleeve was 12 vibro-tactile actuators (3 in the y direction, 4 in the x direction) in size with a length of 12cm and a height of 9cm. This limits the possibilities for detailed touch drawings. Each of the used vibro-tactile actuators has a minimum activation time. This is the minimum time needed between the actuator being activated and being deactivated to notice a vibration. This limits the speed of touch drawings.

8.4 Contribution

Within this project the ability to send unique touch drawings to the sleeve was added. With this new feature more research opportunities open up.

9 CONCLUSIONS

In this paper an algorithm was developed to extend the sleeve made by C. Stork [7] with shapes. This algorithm is expected to be linear in complexity and trial runs suggest a real-time performance for drawing with a short delay between drawing and haptic display activity. This answers SQ1. Regarding SQ2 the tentative conclusion can be reached that the requirements include: a minimum duration greater than the minimum activation time of the used vibro-tactile actuators and an unknown minimum distance. For the main research question (RQ) this means that an algorithm was developed to send shapes from user input to the haptic display. Further research is required to determine the best values for the parameters used in the program to allow for accurate shape representation.

10 OUTLOOK

10.1 Answering SQ2

Since SQ2.3 and SQ2.4 could not be answered, SQ2 could not be fully answered. The parameters of the developed algorithm allow for scaling in the resolution of the output polygons. This enables future researchers to answer SQ 2.3, SQ2.4 and with that SQ2.

10.2 Complexity

While a reasoning for the expected complexity of the algorithm was given in section 7, a full analysis and optimisation is still needed. A more optimised algorithm can run on devices with fewer computational resources, opening up more applications for the sleeve.

10.3 User Interface

The present algorithm implementation has a very limited user interface, only allowing the user to draw. A user interface should be added that allows the user to determine the accuracy and whether the screen is cleared on a new touch or click. This makes the product easier to use since the user no longer needs to adjust the source code or request adjustments.

REFERENCES

[1] Inductive Automation. [n. d.] What is mqtt? Retrieved Jan 19 2023 from https://inductiveautomation.com/resources/article/what-is-mqtt.

[2] The Processing Foundation. [n. d.] Overview. Retrieved Jan 27 2023 from https://processing.org/overview.

[3] Gijs Huisman, Aduén Darriba Frederiks, Johannes Bernardus Fransiscus van Erp, and Dirk K.J. Heylen. 2016. Simulating affective touch: using a vibrotactile array to generate pleasant stroking sensations. English. In *Haptics: Perception, Devices, Control, and Applications* (Lecture Notes in Computer Science). 10th EuroHaptics Meeting 2016, EuroHaptics ; Conference date: 04-07-2016 Through 07-07-2016. Springer, Netherlands, (July 2016), 240–250. ISBN: 978-3-319-42323-4. DOI: 10.1007/978-3-319-42324-1_24.

[4] Gijs Huisman, Aduén Darriba Frederiks, and Dirk K. J. Heylen. 2013. Affective touch at a distance. *2013 Humaine Association Conference on Affective Computing and Intelligent Interaction*, 701–702.
[5] Ali Israr and Ivan Poupyrev. 2011. Tactile brush: drawing on skin with a tactile grid display. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*.
[6] Atsuo Murata. 1998. Improvement of pointing time by predicting targets in pointing with a pc mouse. English. *Plastics and Rubber Processing and Applications*, 10, 1, (Dec. 1998), 23–32.
[7] Connor A. Stork. 2021. *Haptic Wearable for Affective Mediated Touch*. Bachelor's Thesis. University of Twente, Enschede, Netherlands.
[8] Kazuki Takashima, Sriram Subramanian, Takayuki Tsukitani, Yoshifumi Kitamura, and Fumio Kishino. 2008. Acquisition of off-screen object by predictive jumping. English. In *Computer-Human Interaction - 8th Asia-Pacific Conference, APCHI 2008, Proceedings* (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)). 8th Asia-Pacific Conference on Computer-Human Interaction, APCHI 2008 ; Conference date: 06-07-2008 Through 09-07-2008, 301–310. ISBN: 3540705848. DOI: 10.1007/978-3-540-70585-7_34.
[9] Fei Tang and Ryan P. McMahan. 2019. The syncopated energy algorithm for rendering real-time tactile interactions. *Frontiers in ICT*, 6. DOI: 10.3389/fict.2019.00019.

A SLEEVE REFERENCE

A.1 Setup

To set up the sleeve connect the pi to a PC using an ethernet cable. Use SSH to connect to the pi and enter **sudo raspi-config**. Then connect the pi to WiFi using SSID and a password if required. Due to the limitations of the pi non-enterprise networks are recommended. The MQTT server parameters are in Main.py starting on line 161. Main.py is located in the Documents/Haptic_Code folder. The size of the sleeve and distance between the actuators can be adjusted in line 13: **tactileBrush = TactileBrush(3, 4, 3.0)**. Within this TactileBrush constructor the variables, in order, are: number of actuators on y axis, number of actuators on x axis, and distance in cm between actuators. If the number of actuators is changed then also change line 26-40 by adding or removing actuators. tca and tca1 are the multiplexers used in this project. To run the code enter in the command line:

- cd Documents/Haptic_Code
- python Main.py

The console should output connected successfully. You can now start sending messages to the pi.

A.2 Standard MQTT message

The topic used by default by the sleeve is "deviceOne/line" . The standard message expected by the sleeve consists of:

startX	float; x coordinate within the sleeve grid for the starting point of the stroke
startY	float; y coordinate within the sleeve grid for the starting point of the stroke
endX	float; x coordinate within the sleeve grid for the end point of the stroke
endY	float; y coordinate within the sleeve grid for the end point of the stroke
duration	foat; duration of stroke in milliseconds, used to determine the stroke speed

The message is expected as a string with the following format: startX,startY,endX,endY,duration

A.3 Common errors and problems

A.3.1 Point not in grid. This error happens when the given coordinates are not within the coordinate bounds of the sleeve. Often the cause is either a scaling error or an off by one error. Remember that these coordinates start at 0 on the first actuator.

A.3.2 MQTT messages not received. If the pi is connecting to the MQTT server but not receiving messages, check that messages are sent to the correct topic.

B ALGORITHM REFERENCE

B.1 Setup

There is a number of different constructors with various combinations of the following variables:

- *frameMax*: int : The maximum line duration, make sure to set this to be higher than or equal to *fps*. Set to 0 to disable maximum line duration
- *checkFrame*: int : The number of frames between checks for vector accuracy, minimum value of 2
- *maxAngleDifDeg*: int : The maximum angle difference in degrees between the current vector and the new mouse movement
- *fps*: int : The number of frames per second
- *clickClear*: boolean : Whether a new mouse click should clear the screen
- *recover*: boolean : Whether the algorithm should attempt to make a line out of the mouse position for the frames between a passed and a failed check. Set to true if you start notice gaps in the drawing.
- *sleeveX*: int : The number of actuators the sleeve is wide, starting count at 0
- *sleeveY*: int : The number of actuators the sleeve is high, starting count at 0
- *screenX*: int : The number of pixels the UI screen is wide
- *screenY*: int : The number of pixels the UI screen is high
- *minimumDuration* : float : The minimum needed vector duration in milliseconds
- *minimumSize* : float : The minimum needed vector size on screen

With the most minimal constructor the assumed default values are:

- *frameMax*: 0
- *checkFrame*: 5
- *maxAngleDifDeg*: 20
- *fps*: 60
- *clickClear*: true
- *recover*: true
- *sleeveX*: 3
- *sleeveY*: 2
- *screenX*: 1920
- *screenY*: 1080
- *minimumDuration*: 6.1F
- *minimumSize*: 0.1F

Once initialised simply run the main function.

B.2 MQTT

Within the sketch on line 35 to 43 are used for the MQTT variables. They are:

- *serverURI*: String : The link to the MQTT server
- *user*: String : The username used for the MQTT server

- *pass*: String : The password used for the MQTT server
- *topic*: String : The topic used for the MQTT messages, this should be the same as the topic for the sleeve

If you wish to use an MQTT server without a username and password simply remove the username and password from the constructor for MQTTHandler in line 232 in setup(). This will call the alternative constructor for this purpose

B.3 Common errors and problems

B.3.1 NotEnoughDistanceException. When this error shows up it will not crash the program but may cause a gap in the drawing. This error tends to show up for drawings that are more detailed than the sleeve can represent or when the accuracy requirements are to high. Tweak the *checkFrame* and *maxAngleDif* to adjust the accuracy. Make sure *minimumSize* is set to the correct value for your haptic display.

B.3.2 NotEnoughDurationException. When this error shows up it will not crash the program but may cause a gap in the drawing. This error tends to show up for drawings that are drawn quickly or more detailed than the sleeve can represent or when the accuracy requirements are to high. Tweak the *checkFrame* and *maxAngleDif* to adjust the accuracy. If you set the *frameMax* to a number that is lower than your *fps* set it to be at least equal. *frameMax* determines the maximum duration of your strokes so setting it at less than a second can cause issues. Make sure *minimumDuration* is set to the correct value for your haptic display.

B.3.3 Screen too full to draw or drawing keeps disappearing. The boolean *clickClear* determines whether the screen will be cleared on a mouse-click or start of a touch. When true, the screen is cleared, when false the drawings will persist. Check that *clickClear* is set to the correct value for your use.

B.4 Pseudo-code

Below is the pseudo-code for the algorithm.

```
init sleeveX, sleeveY, screenX, screenY, frameMax, checkFrame,
     maxAngleDif, framesSinceCheck, framesVector,
     currentVector, lastVector, startX, startY, lastX, lastY,
     recover
while (frames being rendered){
    if(new mouse button press){
        startX, startY = lastX,lastY = mouseX, mouseY
    }
    if (mouse button is pressed){
        if(framesSinceCheck >= checkFrame){
            if(angle between(currentVector, vector(lastX,lastY
                 to mouseX,mouseY)) > maxAngleDif){
                send lastVector
                if(recover){
                    currentVector = vector(lastX,lastY to
                         mouseX,mouseY)
                    startX,startY = mouseX,mouseY
                }
                else{
                    startX,startY = lastX,lastY
                }
                framesVector = framesSinceCheck
            }
            else{
                lastVector = currentVector.copy
                lastX,lastY = mouseX,mouseY
                framesSinceCheck = 0
            }
            currentVector = vector(startX,startY to mouseX,mouseY)
            framesVector ++
            framesSinceCheck ++
            if(framesVector > frameMax){
                send currentVector
                startX,startY = mouseX,mouseY
                framesVector = 0
            }
        }
        draw previously sent vectors
        draw currentVector
}
```

C COMPLEXITY REASONING FOR SEPARATE TASKS

C.1 Drawing a frame

For each frame, all vectors in the sent vector list are drawn, as well as the *currentVector*. This makes the work load per frame linear with the list of sent vectors. The size of this list depends on how long the program has been running as well as the accuracy requirements (*checkFrame, frameMax, maxAngleDif*). As the accuracy increases, so does the number of vectors. Let us take x as the work load to draw a single vector. Then the work load for the drawing of a single frame is $Workload_{draw} = x * (numberOfSavedVectors + 1)$ For the workload relative to other functions this is simply multiplied by *fps*.

$$Workload_{draw} = x * (numberOfSavedVectors + 1) * fps$$

The work memory load consists of 4 floats per vector drawn from the list. There are 2 added floats from simple arithmatic operations needed to determine the line end coordinates. The *currentVector* requires just 4 floats. This memory can be reused for the next frame. This totals to a work memory load of approximately

$$MemoryLoad_{draw} = 6 * (numberOfSavedVectors) + 4$$

C.2 Updating the current vector

Updating the current vector consists of making a new vector from the *startX*, *startY* to the latest mouse position. This happens for every frame rendered, thus the workload is linear with the number of frames per second. A new vector is constructed and copied using 4 variables for each frame and two simple arithmetic operations. Simple arithmetic is seen as O(1). Naming the constructing and copying of a vector y, the workload per frame is: $Workload_{update} = y + 2$. For the workload relative to other functions this is multiplied by *fps*.

$$Workload_{update} = (y + 2) * fps$$

The work memory load consists of 4 float reads, a PVector copy, a PVector construction, and 2 integers per frame. For simplicity int and float are assumed to be the same size of 1 in memory. Since a PVector construction and copy are both the size of a PVector in memory this will be taken as $2 * PVector$. The variables used in the construction of the PVector are already taken into account in the 4 floats. This memory can be reused making the total load approximately:

$$MemoryLoad_{update} = 6 + 2 * PVector$$

C.3 Accuracy checks

An accuracy check consists of making a new vector, calculating the angle between two vectors and comparing it to a given angle. Comparison has a workload of O(1). For readability the construction of a new vector and calculation of the angle are named *z*. This makes the workload per check $Workload_{check} = z + 1$. The number of checks within a certain run time for the program are determined by $fps \div checkFrame$. This makes the relative workload:

$$Workload_{check} = (z + 1) * (fps \div checkFrame)$$

The work memory load consists of a float from an angle calculation, a constructed PVector from 4 floats, the PVector *currentVector*, and a given angle, which is also a float. This memory can be reused at the next check. Again taking the memory load of a float to be 1, the work memory load approximates:

$$MemoryLoad_{check} = 6 + 2 * PVector$$

C.4 Resets

This task contains all the operations needed to start a new vector, such as assigning a new starting point, clearing position history and resetting counters to 0. This consists of 8 assignments, 2 PVector copies, 2 arithmetic operations, and 2 lists that are cleared. Simple arithmetic operations and assignments are assumed to be O(1). The workload of PVector copies and list.clear is unknown but may be represented as *w*. This makes the workload per reset $Workload_{reset} = 10 + w$ The workload compared to other tasks depends on the number of times this reset is performed. Since this task is performed on a failed check the accuracy and the number of frames per second determine the workload. This is at worst linear with $fps \div checkFrame$, which would mean every check fails. The reset task is also performed when a mouse button is pressed or when the maximum vector duration, *frameMax* is reached. If *frameMax* is set lower than *checkFrame*, but not 0, the work load becomes linear with $(fps \div frameMax)$. If a user clicks more frequently than $fps \div checkFrame$ and $fps \div frameMax$ the work load is linear with the number of clicks. This gives three options for the work load compared to other functions: $Workload_{reset} = (10 + w) * (fps \div checkFrame)$ or $Workload_{reset} = (10 + w) * (fps \div frameMax)$ or $Workload_{reset} = (10 + w) * mouseClicks$. The typical use case will likely have an *fps* of 30 or 60 with a *checkFrame* of half the *fps* and a *frameMax* of a multiple of the *fps*. This makes the most likely workload:

$$Workload_{reset} = (10 + w) * (fps \div checkFrame)$$

The working memory load consists of 2 integer writes, 6 float writes, 6 float reads and 2 PVector copies. For this section we assume list.clear has no memory load as no list elements need to be saved. The memory can be reused for each call. Taking again floats and integers to have a memory load of O(1) the memory load approximates:

$$MemoryLoad_{reset} = 8 + 2 * PVector$$

C.5 Recovery after a failed check

This task consists of checking that the position saving lists are the same size and then using those lists to construct a new vector. The workload consists of 2 list.size calls, comparison of 2 integers, and creating a new PVector with 2 simple arithmetic operations. FloatList.size() is O(1) since the size of a list within Java is stored. Again assuming int comparison and simple arithmetic are O(1), this makes the workload approximately $6 + newPVector$ per call.Assuming *recover* is set to true this happens on every failed check. This makes the number of calls equal to $(fps \div checkFrame)$ for the worst case where every check fails. Thus the total workload compared to other tasks approximates:

$$Workload_{recover} = (6 + newPVector) * (fps \div checkFrame)$$

The working memory load consists of 2 int reads, 2 float reads from 2 FloatLists (total 4 reads), and a PVector write. The lists used are lists of mouse positions between checks, each list has a size of *checkFrame* or less. The memory can be reused every time the task is performed. Again, assuming an int and a float are memory load O(1), the memory load approximates

$$MemoryLoad_{recover} = 6 + PVector + 2 * checkFrame$$

C.6 Repairing broken position lists

If the lists of past mouse positions for X and Y are not the same size the last element of the longer list is dropped so the program can continue. Then the list sizes are checked again. If the sizes still differ a RuntimeException is thrown. Here we will reason up to the throwing of an exception. The workload consists of 5 FloatList.size calls, 2 int comparisons, a simple arithmetic operation, and a FloatList.remove call. As seen before this can be simplified to $5 + 2 + 1 + FloatList.remove = 8 + FloatList.remove$. The number of times this task will be done is unknown.

The working memory load consists of 2 integers from FloatList.size and a FloatList.remove call. It is assumed that FloatList.remove has a memory load of the length of the list +1 for the index of the element to remove. The length of these lists is at most equal to *checkFrame*. The memory used in this operation can be reused. This results in a total working memory load approximating:

$$MemoryLoad_{repairPositionList} = 2 + 2 * (checkFrame + 1)$$

C.7 Passed check

When a check is passed the *currentVector* is copied to *lastVector*, the *lastX* and *lastY* are updated, and the position lists cleared. This, assuming assignments are O(1), gives an approximate workload per call of $Workload_{passedCheck} = Pvector.copy + 2 + 2 * list.clear$. $PVector.copy + 2 * list.clear$ is renamed to v giving $Workload_{passedCheck} = v + 2$ per call. This task can happen up to $fps \div checkFrame$ times. This makes the workload compared to other tasks approximately

$$Workload_{passedCheck} = (v + 2) * (fps \div checkFrame)$$

The working memory load consists of 2 floats and a PVector copy. List.clear is again assumed to have no or negligible load for the working memory. The memory can be reused. This means the working memory load approximates

$$MemoryLoad_{passedCheck} = 2 + PVector$$

C.8 Sending a vector

This task consists of multiple checks followed by the formatting of the MQTT message. The duration is calculated using a float division. This is followed by a float comparison. The distance is based on the PVector components x and y, which are floats. These are compared to floats to check that the line is long enough. This brings us to a total of 3 float comparisons of O(1). The end points of the line are calculated using the PVector components and *startX* and *startY*. A new float[] is made consisting of *startX*, *startY*, endX, and endY. These 4 floats are then each multiplied with *scaleX* or *scaleY*. The float[] is added to the list of sent vectors and the message is formatted as a String. The workload of this function approximates $Workload_{send} = floatDivision + 3 + 4 * PVector.component + 2 + 4 * floatMultiplication + Stringformatting = 5 + floatdivision + Stringformatting + 4 * (PVector.component + floatMultiplication)$ per send. For readability the $floatdivision + Stringformatting + 4 * (PVector.component + floatMultiplication)$ is renamed to u, resulting in $Workload_{send} = 5 + u$. The number of times a vector is sent can be *frameMax* or $fps \div checkFrame$ or the number of clicks. This depends on each case as also seen in subsection C.4. Again the typical use case will result in a workload of approximately

$$Workload_{send} = (5 + u) * (fps \div checkFrame)$$

The working memory load consists of 9 floats, a float[] of size 4, the *vectorsSent* list to which an element is added, and a String (the MQTT message). We assume the memory load of a float is O(1) and the memory load of a float[] or list is equal to its length. The memory load of the *vectorsSent* list is equal to 4 times its length as each element in the list is a float[] of size 4. We may approximate the working memory load as:

$$MemoryLoad_{send} = 13 + (numberOfSavedVectors * 4) + String$$