Convex Hull Tracer Using Sphero BOLT

A project for the computational geometry unit of MATH 3052

Group members: Saad Saeed, Niousha Daghighi

Background Information

Given a set of points P ∈ R^n , the convex hull of P can be defined as the set R ⊂ P that forms the smallest convex polygon and contains all of the points in P [1]. An intuitive analogy is using a rubber band to trace the border of a set of pegs on a board. Such a problem is easy to solve visually. However, solving it computationally can become an involved process as there are a few things that the implementer has to take into consideration. Therefore, the purpose of this project is to experiment with one such implementation of the convex hull problem and extend it to incorporate a robot that can be controlled to trace the convex hull of a set of points in R^2.

The algorithm of choice for this project is the Jarvis March algorithm. This algorithm is one of the most intuitive implementations for finding the convex hull of a set of points; P. A high-level description of this algorithm is that the starting point is always the coordinate with the minimum x-value. To find the next point, loop through the rest of the points and find the one that is the most “counterclockwise” (essentially the left-most point with reference to the point already in the hull) [1]. This process is repeated until we have arrived back at the starting point. All these points are stored in a dynamic data structure and are outputted to the user in the order that they were added. The asymptotic runtime for this algorithm is O(nh), where n is the total number of points and h is the number of points on the convex hull. This can be attributed to the fact that there is a nested loop structure: the inner loop checks every point p’ for each point already in the hull [1]. The worst case occurs when all of the points are on the hull (O(n^2)). As a result, Jarvis March is not the most efficient algorithm and is outperformed by algorithms such as Graham scan which has a run time of O(nlogn).

Implementation Details

This section is broken down into two sections: Technological Requirements and Code Analysis. The former provides a brief overview of the technologies used to implement this project, while the latter provides an in-depth analysis of the written code that was used to accomplish the task.

Technological Requirements

Hardware

1) **Sphero BOLT [2]:** On the hardware side of things, a Sphero BOLT robot was used to actually trace out the points in the convex hull. This was provided to us by the course director.

Software

1) **Visual Studio Code (VS Code) IDE [3]:** The project was developed in two stages: first the base code was written and its output was thoroughly tested, then the code

was ported over to Sphero BOLT and modified slightly to incorporate the commands for movements. To write the base code, VS Code was used. This program supports multiple programming languages and offers a clean and intuitive user experience.

2) **NodeJS [4], Sphero API [5], Code Runner extension [6]:** Sphero requires the code to be written in JavaScript. Plain JavaScript, however, is mostly used for client-side verification when someone visits a website. Hence, it was infeasible to use it for this project since we were not trying to make a web application. To overcome this issue, a specialized framework of JavaScript called NodeJS was used. It allowed us to execute the code without embedding it in HTML tags and monitor the output of the program via the command line interface (CLI) for debugging purposes.

   To access the Sphero-specific commands (regarding movement and sounds), an online API, which is maintained by the manufacturer, was utilized. It contains detailed description of each method and provides sample code which developers can experiment with for their own projects.

   To run the base code on VS Code, a separate add-on called Code Runner was installed. This extension gives VS Code the ability to run the code in languages that it supports. This extension can be installed by searching for it in the Extensions tab of VS Code.

3) **Sphero Edu [7]**: This program was used to communicate with the Sphero BOLT. It offers a text editor where the user can write code and control the robot. The base code was copied over from VS code into Sphero Edu and modified slightly so that it could be executed on the BOLT.

Code Analysis

As mentioned above, one codebase was developed and thoroughly tested for bugs. Afterwards, it was ported over to the Sphero Edu program and refactored so that the BOLT could be controlled. For the sake of completeness, the original codebase will be analyzed in detail. The differences between the original code and the one written for the BOLT will also be analyzed; albeit to a lesser extent since the only difference is that the BOLT code uses specific method calls to control the movement.

Original Codebase

The original codebase consists of helper functions that are used to produce the correct output of the program. This code is available at [8], and is summarized as follows:

Point(x,y) - This constructor function creates an object that represents a 2D point. This function takes 2 numbers as parameters and returns the memory address of the point object.

product(a,b,c) - This function takes an ordered triple of points a, b, c and determines their placements with respect to each other. To find the placement of these points, we compute the pairwise slopes of the line segments ab and ac. If the slope of ab is less than ac, then we know that point c lies to the left of both a and b. If it's the other way around, we know that point c lies to the right of both a and b. If the slopes are the same, they are collinear. Using this idea, we can conclude that the placement depends on the sign that we obtain when we take the differences of the two pairwise slopes [9]. This function returns a 1 if the sign is positive, 2 if the sign is negative, and 0 if the points are collinear.

jarvisMarch(pointsArray) - This is the function that determines which point gets added to the convex hull. It starts off by finding the leftmost point and adds it to an array. Afterwards, it loops through the array and checks to see which set of points are the most counter-clockwise with respect to the two end-points, p and q. To accomplish this, we call the product function. If it returns a 2, we know that the point in question is to the left of p and q. The index value of this point is stored and added to the the array at the next iteration of the outer-loop. This process is repeated until all the points have been inspected. After the execution of this function, an array is returned to the user that contains all the points in the convex hull.

generateAngle(hullArray) - This function computes the absolute value of the angle of the line defined by two consecutive points in the hull. Since the points are in clockwise order, this operation makes sense. Furthermore, this return an array that the robot uses to turn a set number of degrees at a given point.

distanceCalculate(hullArray) - This function takes the array containing the points that make up the convex hull and computes the pairwise Euclidean distances. These distances are stored in an array that tells the BOLT how far to move at a given point.

orientation(hullArray) - This function takes the points of the convex hull and computes the orientation of point p2 with respect to point p1. There are 8 orientations to consider:

1) **Top-bottom-right:** p1 is NE of p2. This means that to get to p2, the BOLT will have to travel in a SW direction.
2) **Top bottom-left:** p1 is NW of p2. This means that to get to p2, the BOLT will have to travel in a SE direction.
3) **Right-Right:** p1 is E of p2. This means that to get to p2, the BOLT will have to travel W.
4) **Right-Left:** p1 is W of p2. This means that to get to p2, the BOLT will have to travel E.
5) **Top-Bottom:** p1 is N of p2. This means that to get to p2, the BOLT will have to travel S.
6) **Bottom-top-right:** p1 is SW of p2. This means that to get to p2, the BOLT will have to travel in a NE direction.
7) **Bottom-top-left:** p1 is SE of p2. This means that to get to p2, the BOLT will have to travel in a NW direction.
8) **Bottom-top:** p1 is S of p2. This means that to get to p2, the BOLT will have to travel N.

These waypoints are crucial for ensuring the BOLT executes the correct turns. These directions, along with the output from the generateAngle function determine where the BOLT should orient itself and how many degrees it should turn to get the right placement. For example if p1 is 56° bottom-top-right of p2, this means that the BOLT must be pointing E and then turn 56° counterclockwise to reach p2.

Code for Sphero BOLT

The code is available at [10]. The program uses all the function mentioned in the last section, along with a set of Sphero-specific functions to control the movements of the BOLT and trace out the convex hull. Their uses are defined as follows:

1) **roll(heading, speed, time)** - combines heading(0-360°), speed(-255-255), and duration (s) to make the robot roll with one line of code [5].
2) **setHeading(angle)** - sets the direction the robot rolls. Assuming you aim the robot with the blue tail light facing you, then 0° is forward, 90° is right, 270° is left, and 180° is backward [5].
3) **resetAim(angle)** - sets the specified heading to be 0°.
4) **calibrateCompass() and compassDirection(angle)** - compassDirection sets the real-world direction based on the last compass calibration. 0° is due north, 90° is due east, 180° is due south, and 270° is due west. It requires the Calibrate Compass command to be run before you can set this value [5].
5) **speak(String)** - speaks a string from the programming device using the text-to-speech engine [5].

The way the code works is that it is assumed that the BOLT is placed on the left-most point in the convex hull. A loop is initialized and run as many times as there are points in the convex hull array. Within the loop precomputed values for angle, orientation, and distance are fetched and the BOLT is moved based on these values. The program ends when we have arrived back to the point that we started from.

Discussion

This section explains some of the challenges we encountered during development, and how we overcame them, and any unavoidable issues that limited our success.

Trade-off

One of the most challenging things to execute were the turns. The issue arises from the fact that by default, the BOLT uses absolute headings (cardinal directions) to orient itself when turning. For example, turning the BOLT 90° changes the heading in a clockwise direction (where 0° is considered North). The idea of relative direction does not exist, and as a result, creative approaches had to be devised.

One such approach was to store the direction of an arbitrary point p2 relative to the point that came before it, say p1. This is done in the orientation function. From there, we know that p2 can have 8 different orientations relative to p1 (see the section named Original Codebase for details). We can then calibrate the BOLT compass to point in one of the four cardinal directions (based on the direction the BOLT needs to turn), set that equal to 0°, and then turn in a clockwise or counterclockwise direction based on the information returned by the generateAngle function. For example, going from points (-2,6) to (0,0) returns “top-bottom-left” from the orientation function. Furthermore, suppose that the generateAngle function returns 72°. What this means is that p1 (-2,6) lies N18°W of p2 (0,0). To get to p2 from p1, the BOLT would need to face E and turn 72° clockwise to get the correct heading. Continuing on with this example, coding this is straightforward. First you calibrate the compass to 90. This turns the BOLT in the eastern direction. From there, you use the resetAim function so that it treats the current heading as N (0°). Finally, you use the setHeading function and pass in 72 so that the BOLT turns 72° in a clockwise direction and faces the correct heading. This process is repeated for all turns.

One of the issues with this approach is that the calibration is unreliable the higher up you are in terms of altitude. When testing the program on the ground floor of the Ross building at York University, the BOLT had no problem tracing out the convex hull of a set of predetermined points. However, when the same experiment was run on the 21st floor of a condo building, the BOLT was having issues establishing the four cardinal directions, which resulted in inaccurate heading and ultimately, incorrect turns. This anomaly can be attributed to the rather inexpensive sensors (specifically the gyroscope) that are built into the BOLT. It has a hard time establishing the cardinal directions because the building slightly sways. However, given the hardware and time constraints, we were happy to see it perform well in one case.

References

[1] A. Mandal, “Convex Hull | Set 1 (Jarvis's Algorithm or Wrapping),” GeeksforGeeks, 29-May-2018. [Online]. Available: https://www.geeksforgeeks.org/convex-hull-set-1-jarviss-algorithm-or-wrapping/. [Accessed: 13-Feb-2019].

[2] “Sphero BOLT,” Sphero. [Online]. Available: https://www.sphero.com/en_ca/sphero-bolt. [Accessed: 20-Feb-2019].

[3] “JavaScript Programming with Visual Studio Code,” 14-Apr-2016. [Online]. Available: https://code.visualstudio.com/docs/languages/javascript. [Accessed: 12-Feb-2019].

[4] N. Foundation, “Download,” Node.js. [Online]. Available: https://nodejs.org/en/download/. [Accessed: 12-Feb-2019].

[5] “Programming wiki for Sphero Edu platform,” Sphero Edu JavaScript. [Online]. Available: https://sphero.docsapp.io/docs/get-started. [Accessed: 12-Feb-2019].

[6] J. Han, “Code Runner - Visual Studio Marketplace,” Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=formulahendry.code-runner. [Accessed: 12-Feb-2019].

[7] “Get The App,” Sphero Edu. [Online]. Available: https://edu.sphero.com/d. [Accessed: 15-Feb-2019].

[8] SaeedS28, “SaeedS28/MATH3052,” GitHub, 05-Feb-2019. [Online]. Available: https://github.com/SaeedS28/MATH3052/blob/master/Project/JarvisMarchToJs/jarvisMarch.js. [Accessed: 12-Feb-2019].

[9] R. Agarwal, “Orientation of 3 ordered points,” GeeksforGeeks, 11-Feb-2018. [Online]. Available: https://www.geeksforgeeks.org/orientation-3-ordered-points/. [Accessed: 15-Feb-2019].

[10] SaeedS28, “SaeedS28/MATH3052,” GitHub. [Online]. Available: https://github.com/SaeedS28/MATH3052/blob/master/Project/JarvisMarchToJs/spheroHull.js. [Accessed: 17-Feb-2019].