Lucas Liu Project 3 Phase 2 Implementation Details:

Step 7:

The robot’s movement is computed using two steps. The first step involves identifying the direction the robot is currently headed. This is done using a destination corner, an origin corner and a point representing the current position of the robot. The destination corner is a corner that is part of the triangle that forms the circumcenter the robot is currently headed to. Using this corner and the triCircumcenter function, I computed a vector from the previous corner’s circumcenter to the destination corner’s circumcenter and normalized it for the direction vector. The robot’s point is then updated with the direction vector scaled by the speed vector computed in the next step. The speed of the robot has three different settings computed with updateSpeed. In the first setting, the robot heads at a constant speed. In the second, the robot takes the square root of the distance between the current robot location and the location of the mouse projected to the floor. If this distance, scaled by some constant, is greater than three, then three is the speed of the robot. Else, the speed is 0.5 +sqrt(d(robot, mouse)/2)). In the third step, the robot makes a gentle start and stop by computing two constants, the distance from the current location to the destination circumcenter and the distance from the current location to the origin circumcenter. The minimum of these two constants is taken then scaled by 1/20 to produce the speed.

The robot detects that it has arrived by the junction, junction being a circumcenter, using the atPoint method. This method takes the distance between the current location point and a destination point. If this distance is less than five, then the current location is set to the destination and the method returns true, else, the method returns false.

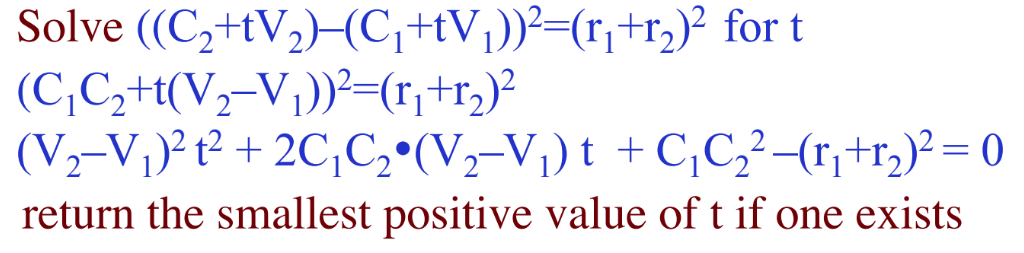
My roomba does navigation using an adjacent corner table, which is an array of size nc that contains at each index c a list of corners that form delauney triangles adjacent to the triangle formed by c. Seeing as how there are three corners per triangle, the opposite of each such corner being part of an adjacent delauney triangle, I iterate over the three corners in each triangle and compute a list of opposite corners. This is done in setAdjacencyTable. If the opposite of a corner is itself, then the triangle the corner belongs to is on the border and has only two adjacent delauney triangles / corners. Once my robot arrives at a circumcenter belonging to the destination corner c, I retrieve a list of corners adjacent accessed with adjacencyCorner[c]. I iterate over these adjacent corners and find the corner whose circumcenter is closest to the mouse. The robot sets this corner as its new destination corner. This is done in computeNearestCircumcenterCorner. In short, my robot selects the corner whose circumcenter is closest to the mouse in a greedy way.

Note: because of the difficulties with ensuring a stable framerate and distance updating (increment distance traveled by speed float happens at draw function call), my robot updates the distance traveled only when at a circumcenter, at which point it increments the total distance traveled by the distance from the previous circumcenter to the current circumcenter.

Step 8

My robot navigates a minimal path to cover all triangles using a modified version of depth first search. The typical construction of a main dfs method and a recursive helper is modified with a path list and a parent parameter. After expanding a vertex, a path from the parent to that vertex is added to the path list, and after expanding the children of a vertex, a path from the vertex to its parent is added to the path list. This removes the “teleporting” behavior seen with dfs. The path list is returned and processed by simulating a movement through corners / circumcenters, and once all triangles have been visited, the remaining paths in the path are removed. This prevents the robot from returning to its start position and minimizes total number of steps. In my implementation, the robot does not search the world as it moves and instead follows the pre-computed path. This allows the user to disable “following path mode” and attempt to beat the robot’s AI in a game, as the robot’s path length has already been computed.

Step 9:

My robot detects pillars by using several different helper methods. The first helper method I implemented was a quadratic equation solver that given the constants a,b and c of a quadratic equation, will produce the roots. I use this to solve the equation on the professor’s slides:

The quadratic equation solver method is incorporated in my check collision method that checks whether a ray shot from the robot’s current position to a specified pillar will collide with that pillar, and if so returns the time it takes for the particle to reach the pillar assuming unit vectors are used for direction. This is then incorporated into the vector collision method that solves for the smallest time T that the ray shot from the robot will collide with any of the pillars on the map by iterating check collision over all pillars. Finally, my shoot rays method takes in an integer parameter for the number of rays to be shoot and produces that many different rays uniformly distributed over a unit circle. Pillars seen are tagged and a ray is rendered.

When coming up with an algorithm that would allow the robot to detect all the pillars, I realized that the only way this could be guaranteed is that if the robot were to be hit every circumcenter on the map and by doing so come into contact with the pillars, therefore guaranteeing pillar detection. As such, I used the same path computed in step 8 with the modification that if the robot were to detect all the pillars, the robot would terminate its path following behavior and return the length of the path it has traveled.

I did not attempt step 10.

Attempted objectives:

Step 7

Step 7 with user specified variable speed options (Extra Credit)

Step 8 with near minimal path

Step 9.

Hotkeys:

P = display pillars

. = hold to adjust pillar radius

h = hold to adjust pillar height

j = increase roomba volume

k = decrease roomba volume

a = pause animation / stop robot

\* = cycle movement types

To edit the mesh, do not use live mode and instead pause the robot, then reset the robot

r = reset robot

J = toggle path following behavior mode

q = switch between step 8 and step 9 configuration (roomba and robot guard)

l = increase ray count

m = decrease ray count

Notes:

moving the mouse too far off to the horizon breaks everything