Rebecca Lee
013919802
Kayte Chien
014217970

Lab 1 Report

STEERING BEHAVIOR

Our steering behavior accounts for both singular targets and multiple waypoints. In regards to slowing down the agent, we chose to implement an arrival radius, an area where the agent gradually starts decreasing its speed upon entering. We check for three different conditions before the agent is allowed to slow down. The first check confirms if the agent is within the slowdown range, which we set to be 10 times the arrival radius. The second check confirms if the agent is in the actual arrival radius. The last check is what accounts for the waypoints, if there are any at all; if there are, then the agent is not allowed to come to a complete halt until the last waypoint becomes the agent's current target. Only when all of these conditions are met (within slowdown range, arrival radius, and heading to the last target) will the agent decrease its speed and rotational velocity until it is virtually unmoving.

We had some issues when the waypoints were too close to each other; the boid couldn't speed up fast enough to slow down in time to reach the target, and would stop before it reached the target. We alleviated the issue somewhat with a reduced slowdown radius (see below) and applied a "fix" by adding a lower bound, so the boid would go at a fixed speed when visiting close waypoints. This issue could have (theoretically) been resolved if the boid slowed and sped up at different rates, but we found little success in the implementation: slowing down faster than it sped up led to the boid zooming away backwards, then coming forwards, then going backwards again, and speeding up faster than it slowed down led to strange behavior where the boid jerkily moved forward, then slowed, then moved forward, then slowed (and took a considerable amount of time to get to the target). Using a lower cap was the most practical solution we could implement.

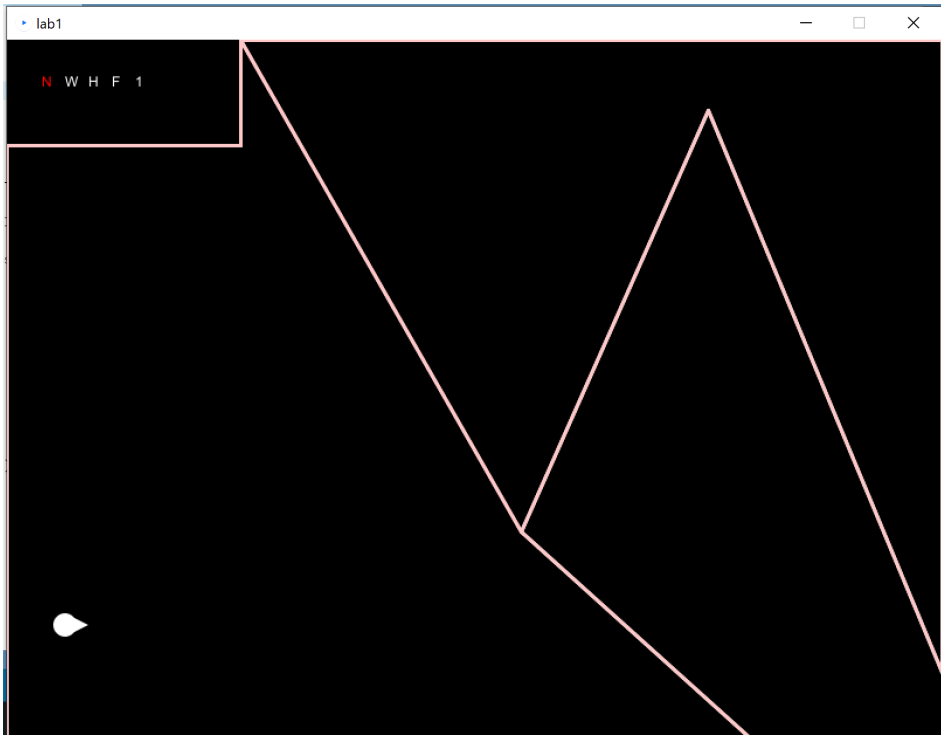
Another notable problem we encountered was that our default arrival radius was too large for the various map tests; because the boid was constantly within the slowdown zones, it would never speed up and proceed through the other waypoints at a slow crawl (the aforementioned lower limit). We alleviated this by manually decreasing the size of the slowdown zones so that, between closely-placed waypoints, the boid would have less time to lose velocity and more time to regain it.

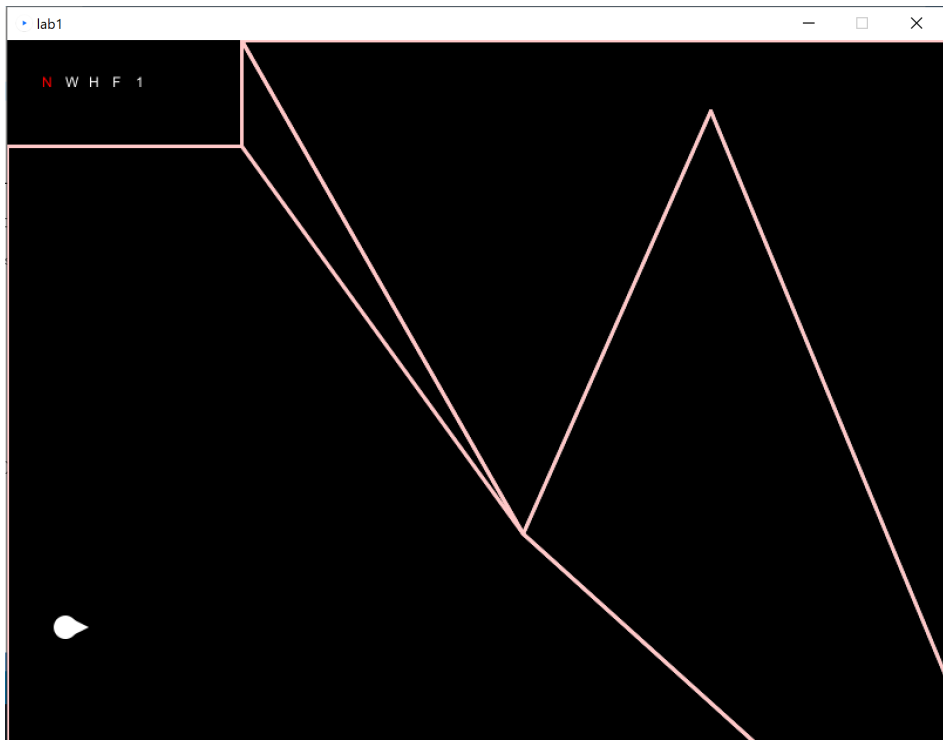
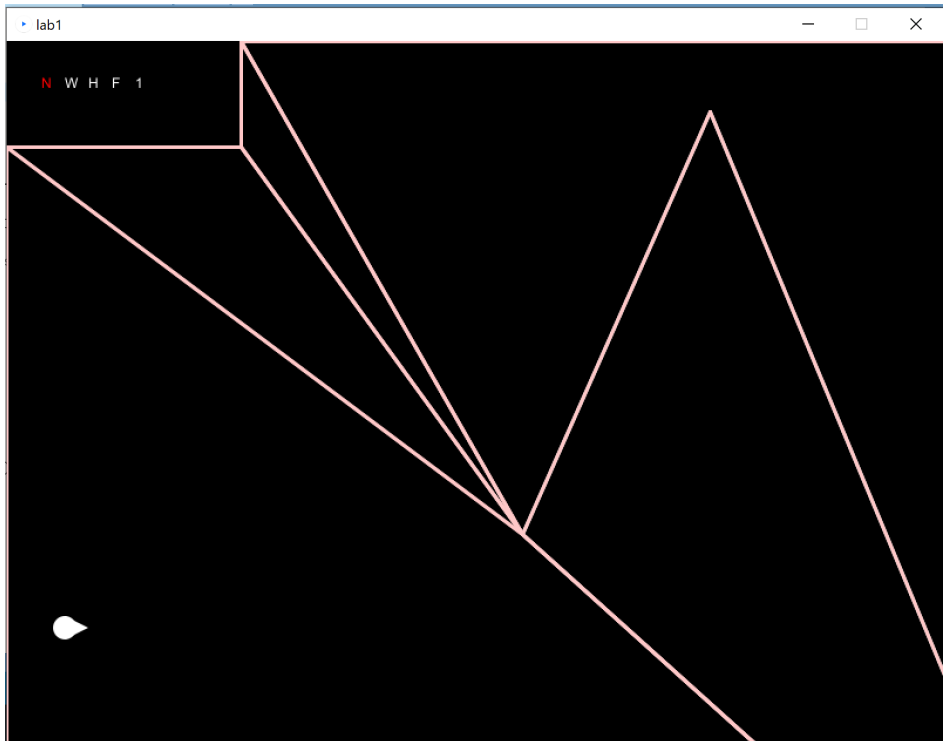
NAVMESH

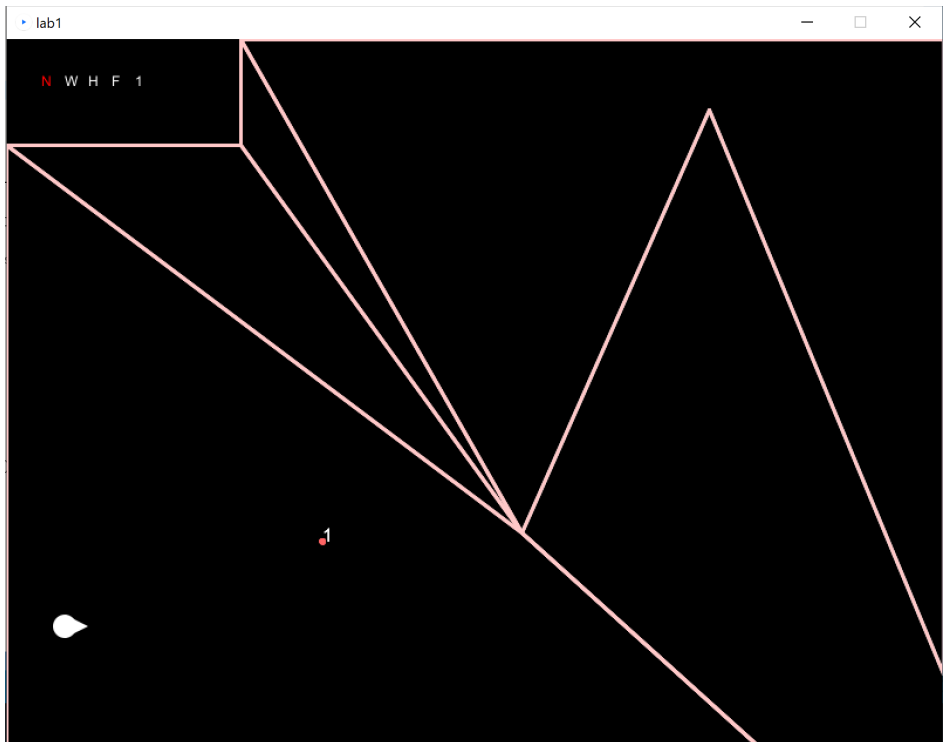
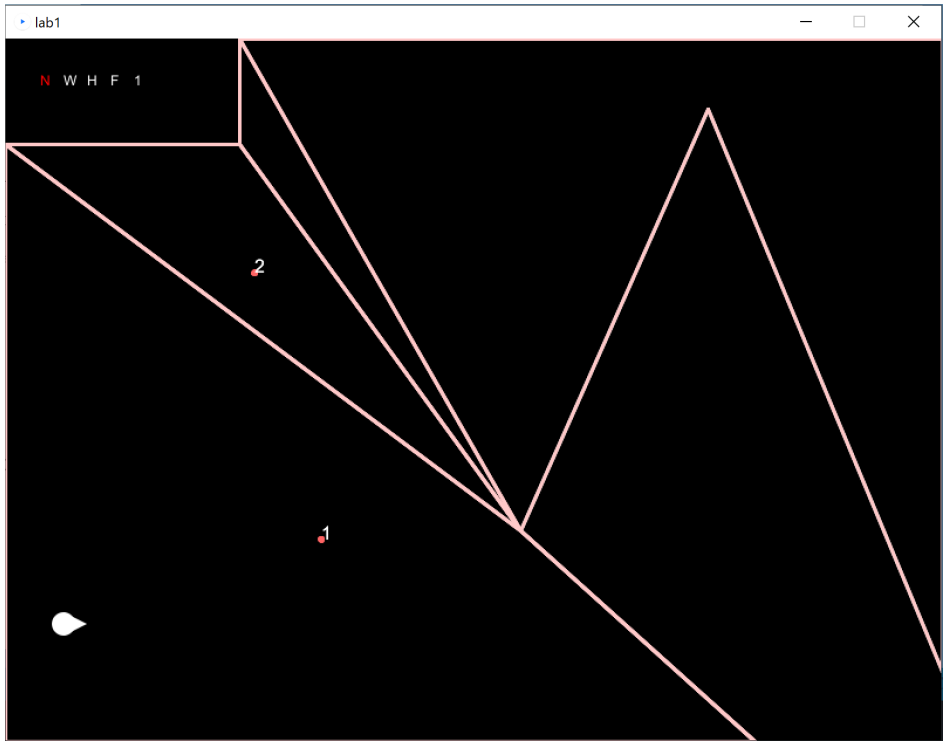
To calculate the NavMesh, the program calls the `breakdown()` method, which takes in a `Node`, representing a polygon, and breaks it into smaller, convex shapes.

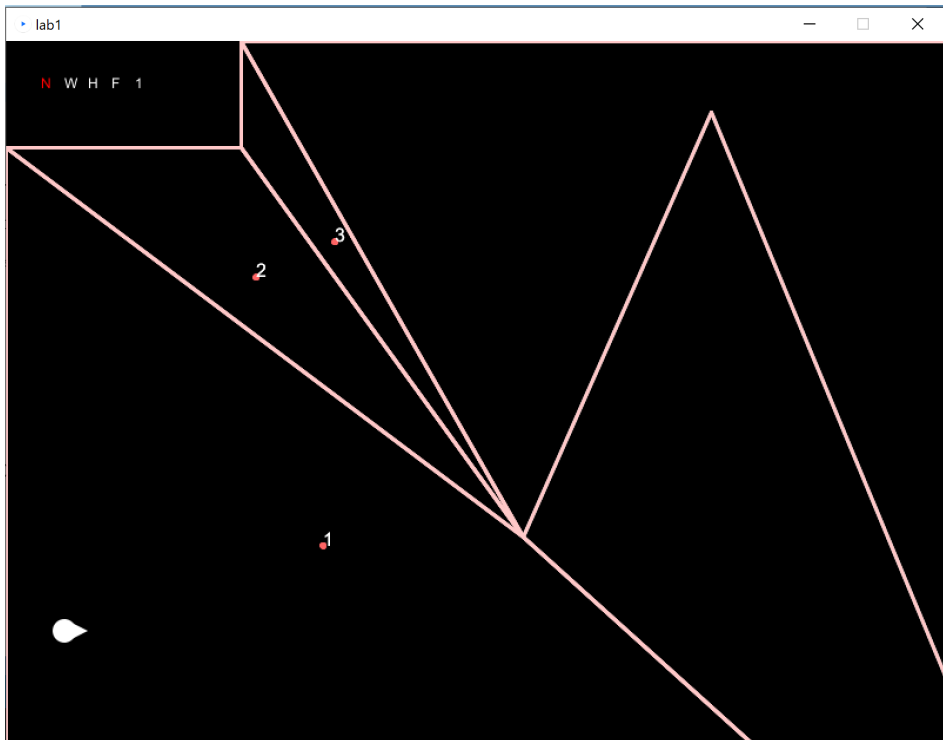
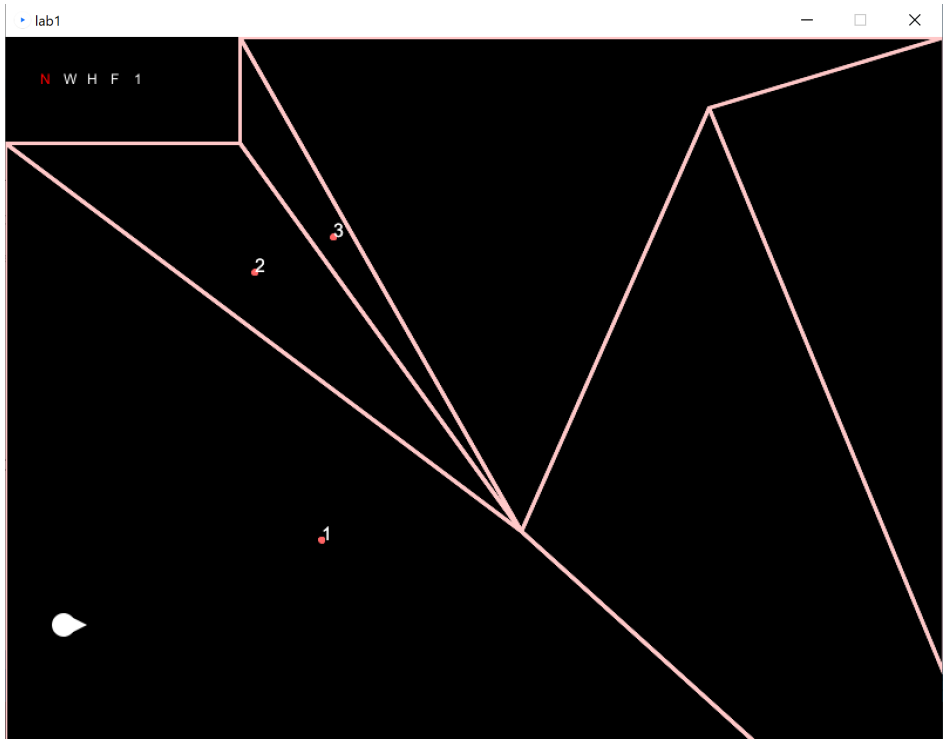
The method checks all corners (points where the walls join) in the polygon. If a corner is reflex, the method then looks for another corner to connect a line to, checking counterclockwise from the reflex corner and making sure the line is in-bounds and does not intersect the map's walls. Once the line is found, it is used to split the polygon into left and right "sub-polygons", which are then entered into `breakdown()`.

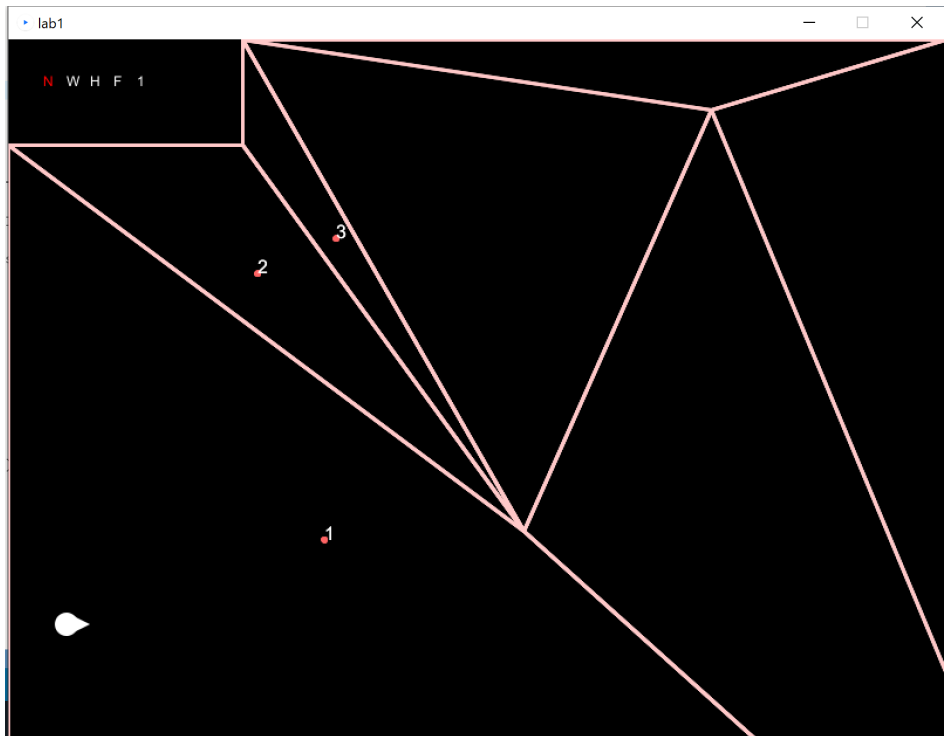
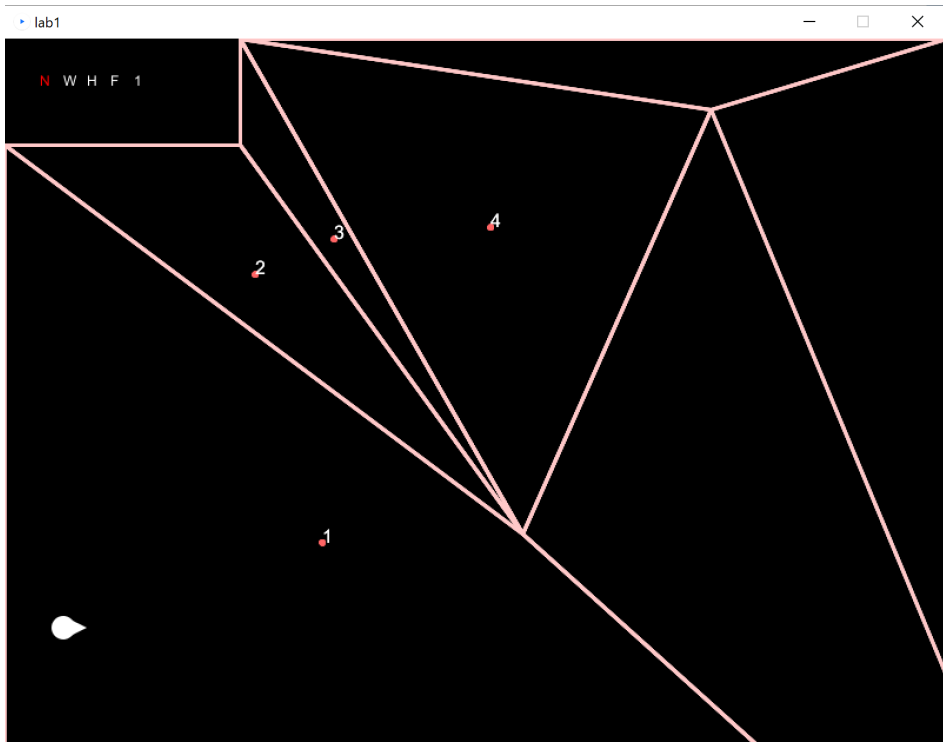
If a polygon entered in `breakdown()` is convex (i.e. has only 3 walls or no reflex angles), it forgoes the split and is added to the ArrayList of `Nodes`, which are then used in further calculations (such as NavMesh).

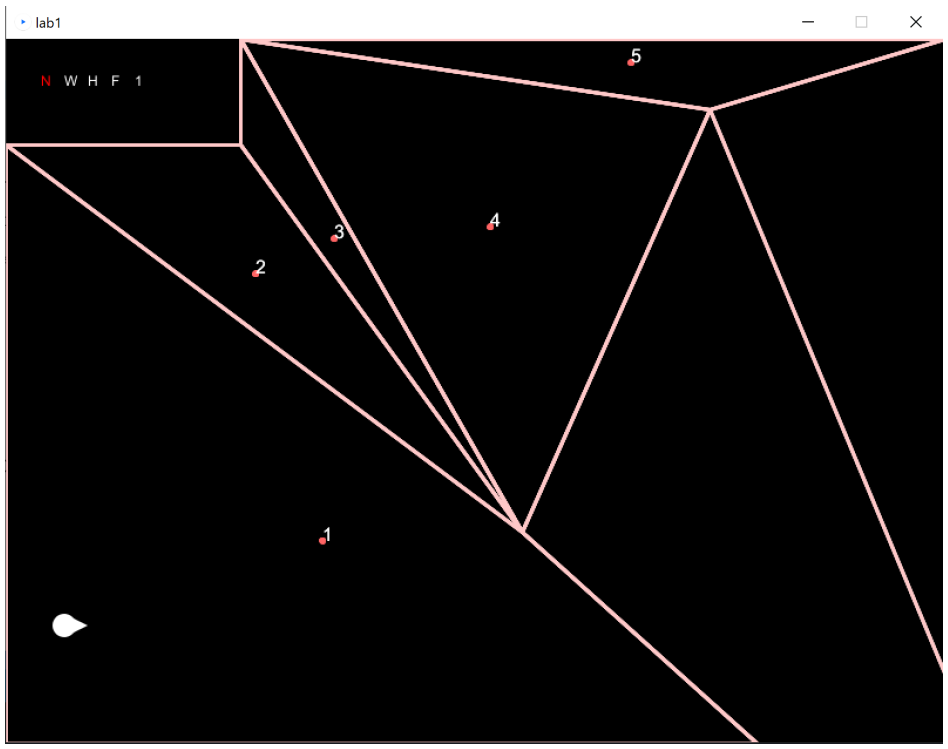
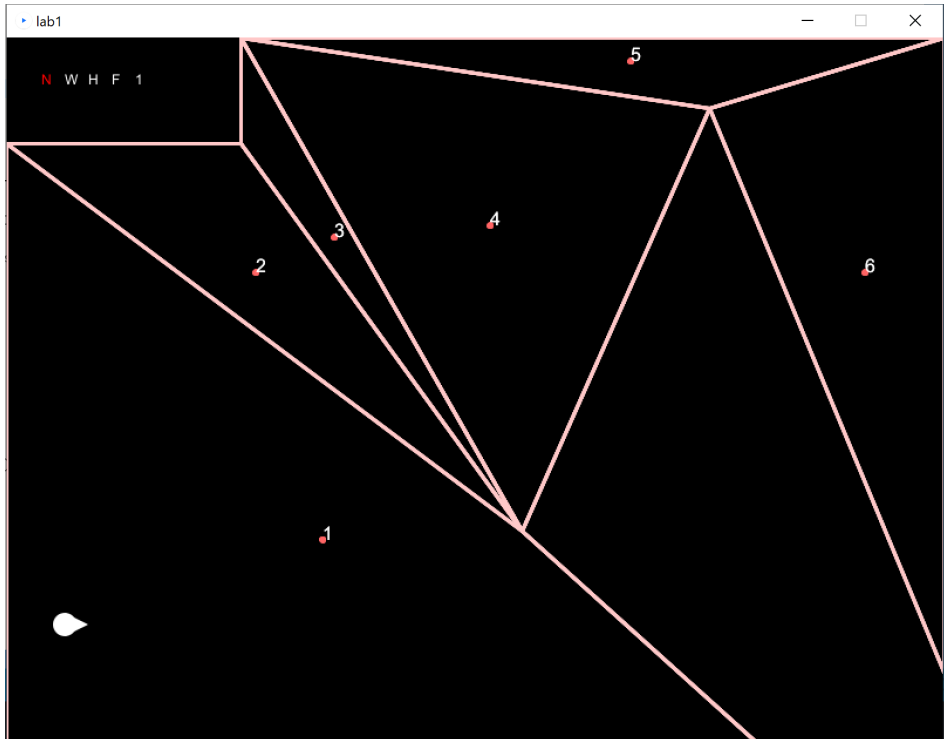
Iteration 1: The entered polygon (in this case, the whole map) is split in two	

Iteration 2: Left polygon (from Iteration 1) splits itself into 2 more polygons. This continues repeatedly until all polygons are convex.	
Iteration 3: The polygon from Iteration 2 is split further.	

Iteration 4: The leftmost shape is identified as convex and is added to `nodes` (shown by its center point and number).	
Iteration 5: The next shape (right side of the Iteration 3 split) is identified as a convex shape and also added to `nodes`.	

Iteration 6: The 3rd shape (right polygon from Iteration 2); is identified as convex and is added to nodes. All pieces from the left polygon created in Iteration 1 have been divided.	
Iteration 7: The next polygon (the right side from Iteration 1) is split into 2 more polygons.	

Iteration 8: The left polygon from Iteration 7 is split once again.	
Iteration 9: The newly created polygon (left of the Iteration 8 split) is identified as convex and is added to `nodes`.	

Iteration 10: The next polygon (right of the Iteration 8 split) is identified as convex and added to `nodes`.	
Iteration 11: The final polygon (right side of Iteration 7) is identified as convex and added to `nodes`. The navmesh is complete!	

The neighbors of a node are determined after the NavMesh is drawn, but details about the neighbors are added to each node during implementation. In `breakdown()`, when a polygon is divided, the line dividing them makes 2 neighboring polygons, and

the line itself is added to `divisionLines`. Each node has an `int[] index`, used in conjunction with its walls, to show which walls correspond to the index of a `PVector` in `divisionLines` (i.e. which walls are division lines/has a neighbor), and -1 if the wall is a map wall. For example, to determine if the nth wall of a node is a division line, we would check the value in the node's `index[n]`.

After the map is broken down (i.e. the `breakdown()` in `bake()` ended), the `nodes` list is iterated to find its neighbors by matching indices in each `node`'s `index` array.

PATHFINDING

Our pathfinding is incomplete. Our main issues lie with implementing the `findPath()` method and successfully implementing the A* algorithm. Our `findPath()` implementation issues go in line with our other boid navigation issues; that is, with the left and right clicks used to mark the boid's next target or add waypoints. This is due to the fact that we implemented the method inside `mousePressed()`. In our current version, the boid will only pathfind (incorrectly) with the first left click, which will set its next destination. Any other clicks will result in its original behavior, where it attempts to go in a straight line to the target destination.

Regarding the A* algorithm, occasionally, the frontier will be empty despite not having reached the target destination. Additionally, there are occasions in which the result array, which contains the midpoints of the walls that the boid wants to travel to, either contains the wrong walls entirely or contains only the first half of what would be the correct path. We have theorized that these issues are due to our A* algorithm being incorrect, but were unable to fix the issue before the allotted due date.