Assignment 2

For the first task of this assignment, we were required to build a virtual class inheritance structure and learn to sample mouse positions. While creating the virtual structure, I learned the difference between passing and a reference and passing a value. We need to take care when we pass the reference as the updates are not reflected back resulting in the sprite not updating it's kinematics. Since updates are made after a time interval of 0.02 seconds, the difference in the mouse position during this interval gives us the velocity directly. Greater the changes, higher is the rate of change of velocity and hence acceleration is not necessary to be calculated by the method I choose to implement. A separate Update class inherits all the properties of the required class so that every instance of this class has all kinematic values and functions. Orientation is also calculated in a similar fashion as the velocity and is smoothed over 60 steps which is a value that works better with our chosen time interval of 0.02 seconds. We utilized the sf::Mouse functions to get mouse positions without the requirement of polling the events. One thing should be noted that it works only when we use the window as a reference. If this is not done the global mouse position seems to cause rapid sprite movements and the velocity calculations do not work.

For the second task of the assignment, the sf::Mouse functions we used previously seem to not work as effectively because it picks up multiple mouse locations for the short duration the mouse left key is pressed. It is now better to poll the events for the mouse position and capture the position when the mouse key is released. We start with the target mouse position of 300, 300 so that the sprite starts moving when the program starts. It is now imperative that the acceleration is calculated. This is because the velocity will have to decrease as we cross the radius of satisfaction and deceleration. Not doing so will result in abrupt stops. The updates still take place in the decided time of 0.02 seconds and hence not required in the calculation of acceleration for the implementation I've chosen. From the screenshots, we can observe that the frequency of the breadcrumbs increase as we reach the target position. This shows that the deceleration is working as intended as the more crumbs are drawn between radius of deceleration and satisfaction (because crumbs are also drawn in the same fixed time interval). We can also confirm this by printing the velocity values. From the experimentation in implementation of the "*arrive"* algorithm, the values of radius of satisfaction and deceleration must be large enough otherwise we won't be able to see the deceleration. In some of the cases the sprite is not able to come to a complete halt and overshoots resulting in large changes in the acceleration values and the sprite oscillates about the target. Any goal speed can be chosen but a larger value does not give enough time for the sprite to be able to orient itself. Hence choosing lower max speed values and somewhat big radius values provides the best result. Usually we would need to clip the acceleration but since acceleration is calculated by velocity changes over a unit time and the velocity is always clipped by the max speed depending on the sprite location.
The *"align"* algorithm is implemented by the directional changes over a period of 30 steps. The amount of rotation will be in 30 steps which has been fixed with the 1/60th of a second time interval in which we make updates. The angular variable will decide the

amount of movement in each step. From the experimentation, the map2range function discussed in the class seems to be working for almost all cases that I could observe. If the step value is too small the rotation is not smooth and it is too large the rotation will take too much time. Since the steps keep the rotation in check, we need not clip the maximum rotation allowed.

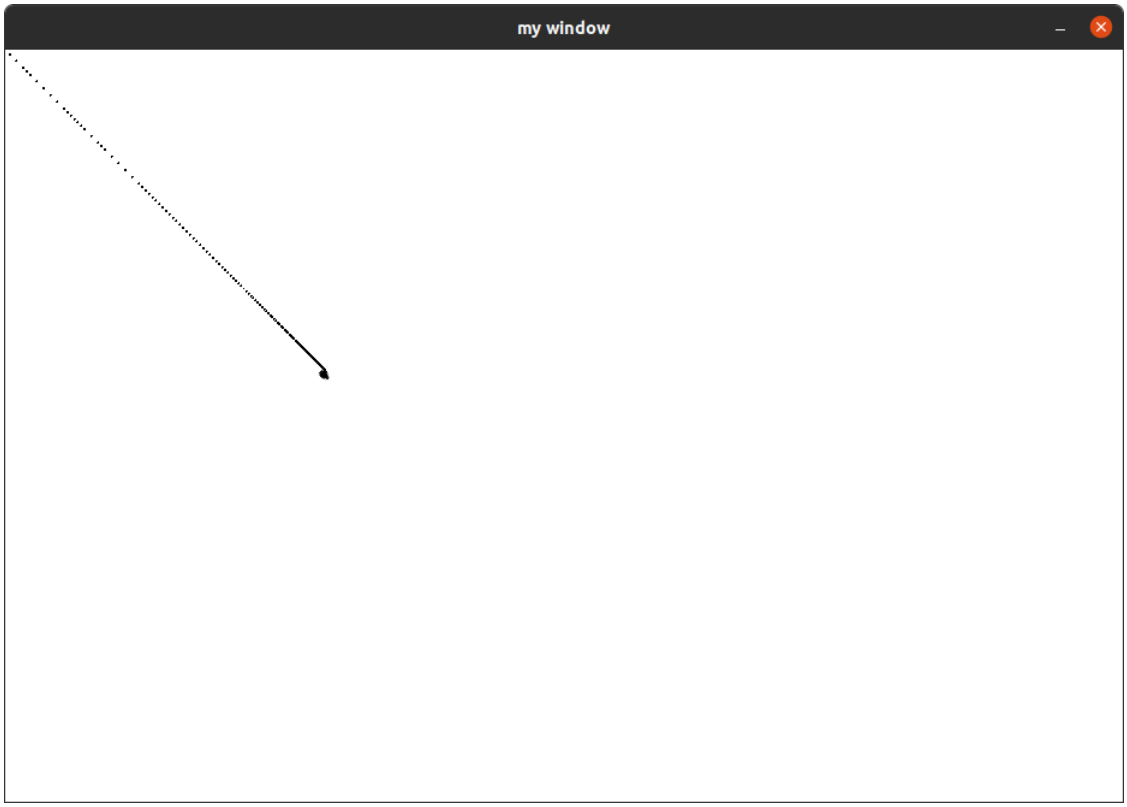
For the third task, we utilize the kinematic seek to chase a moving target. A simple binomial randomization produces the random direction in which we wander. We start with the direction fixed as 100, 200 and then randomize it every 3 seconds. From the random distribution we can get an orientation which we can then convert to a direction. This direction can then be used to calculate a position which is always 10 units ahead of the sprite by moving it with the velocity of the sprite every step. We need not clip rotation since we use our current orientation to get the randomized orientation. Making it so that it is never over 90 degrees in both directions. This makes sure we can move through the available display since the range from which the random number is drawn changes after every new character orientation. We can handle the boundary conditions by simply reversing the direction from which the sprite arrives. This is effective mostly for non corner situations. Since a single ray at a distance of 10 units decides it is about to go out of bounds, maybe a multi ray setup will perform better. A secondary timer is utilized to calculate the random direction and is reset after every potential wall collision. A timer greater than 3 may cause the sprite to oscillate between two corners. Since the amount of rotation is capped, the method implemented looks good enough for our test case.

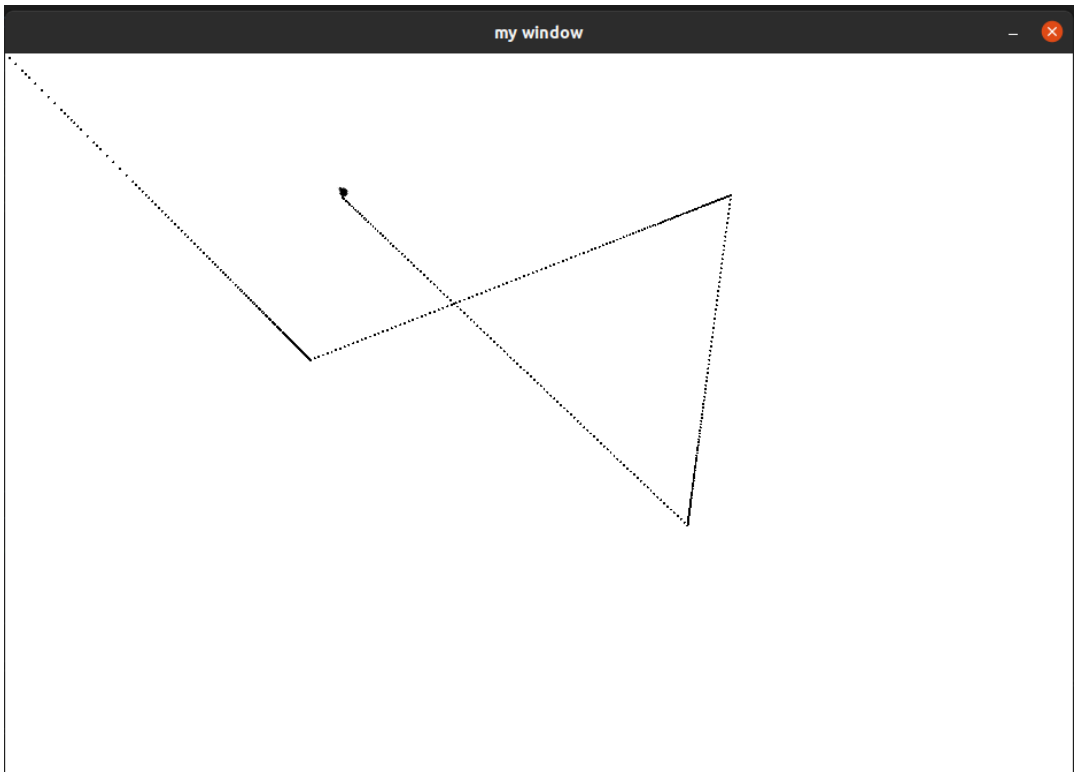
For the fourth task, we start with 4 different classes, each of which handles vector operations, the main game, the character properties and the application of separation, alignment and cohesion. We start from the (almost) center of the screen and travel outwards with the 100 characters forming different groups. Once they reach the end of display bounds they warp from the other side. The characters in a group only avoid collisions from other characters in the same group. We are aiming for a separation value of 20 and count the other characters at a distance of 50. Any lower value for distance, than 50 for the 100 characters seems to be causing the characters to ignore the remaining ones and the flocking seems more like random movements. These values have been decided on the basis of the standard VGA screen size we have been using. If more screen is available for the characters to move we can comfortably increase their numbers and change the separation values as well.
For each of the characters we calculate it’s distances with the others, we decide on the separation based on the fact whether the characters are followers or not, i.e. they are following or leading the group. If both the characters are followers then the separation between them should not be much creating a flock but if one of them is a follower and the other is not, the distance has to be large based on the standard boids algorithm. Similarly for each of the characters during alignment we look at a distance of 50 units and count the other characters in that range. We calculate the average velocity for these for velocity matching.

We utilize a similar distance of 50 units for calculating the center of mass for cohesion, by taking the average of the locations of the character in that radius. All these behaviors affect acceleration and hence are applied to it.

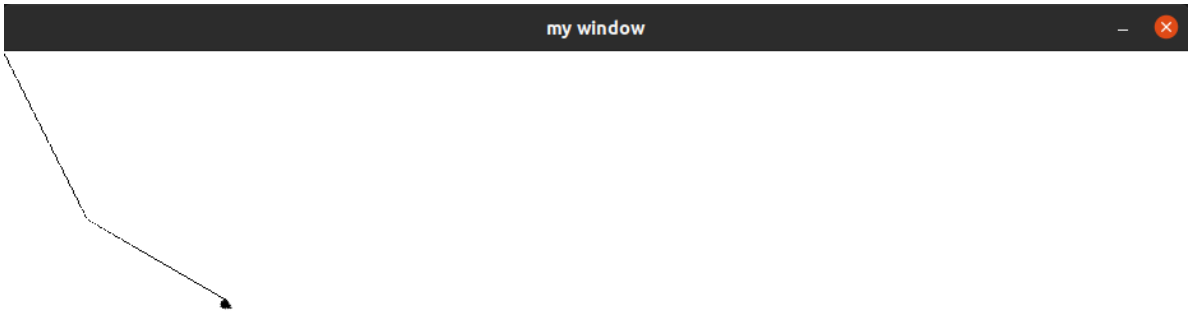
Appendix

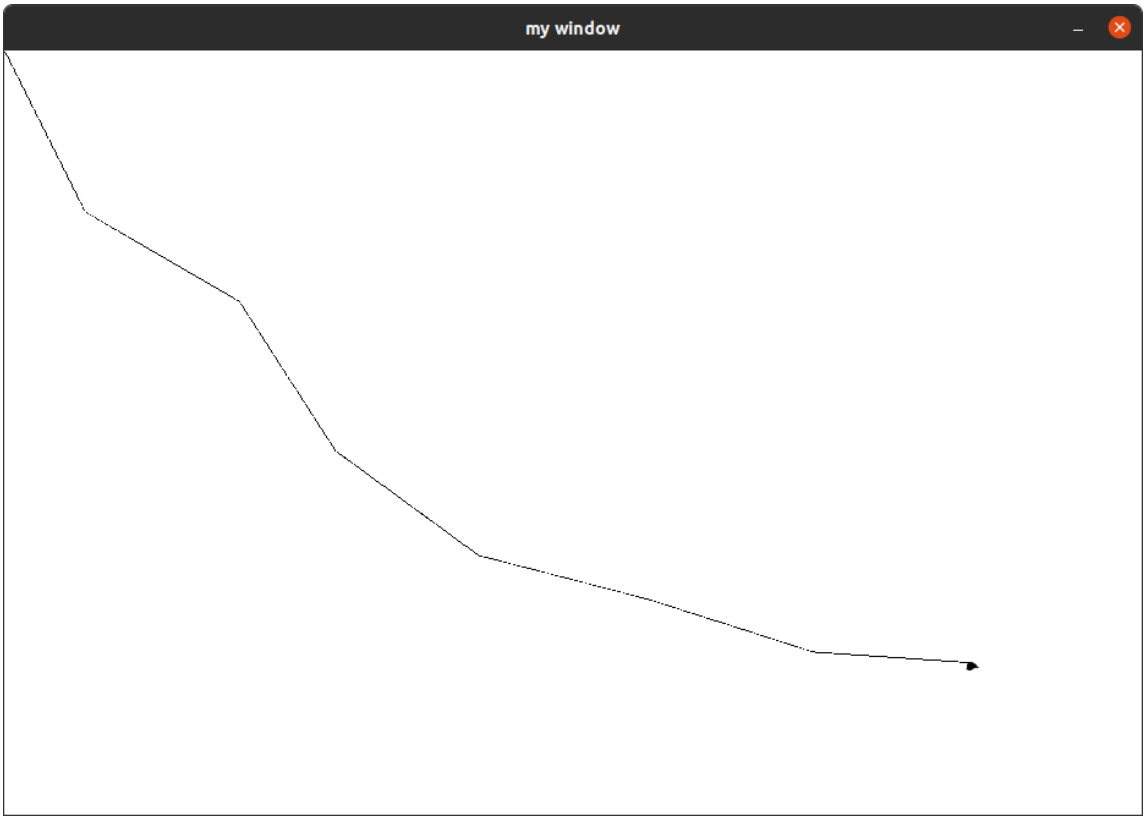
Part2:

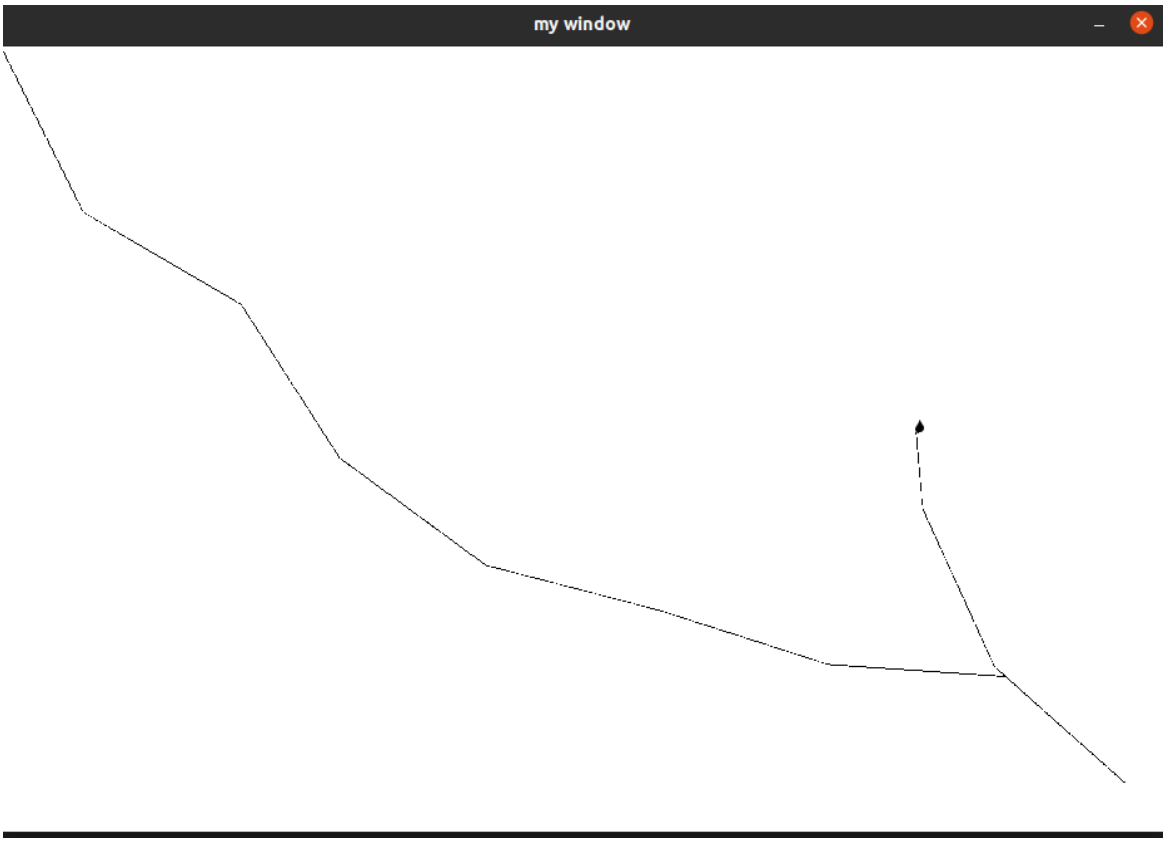


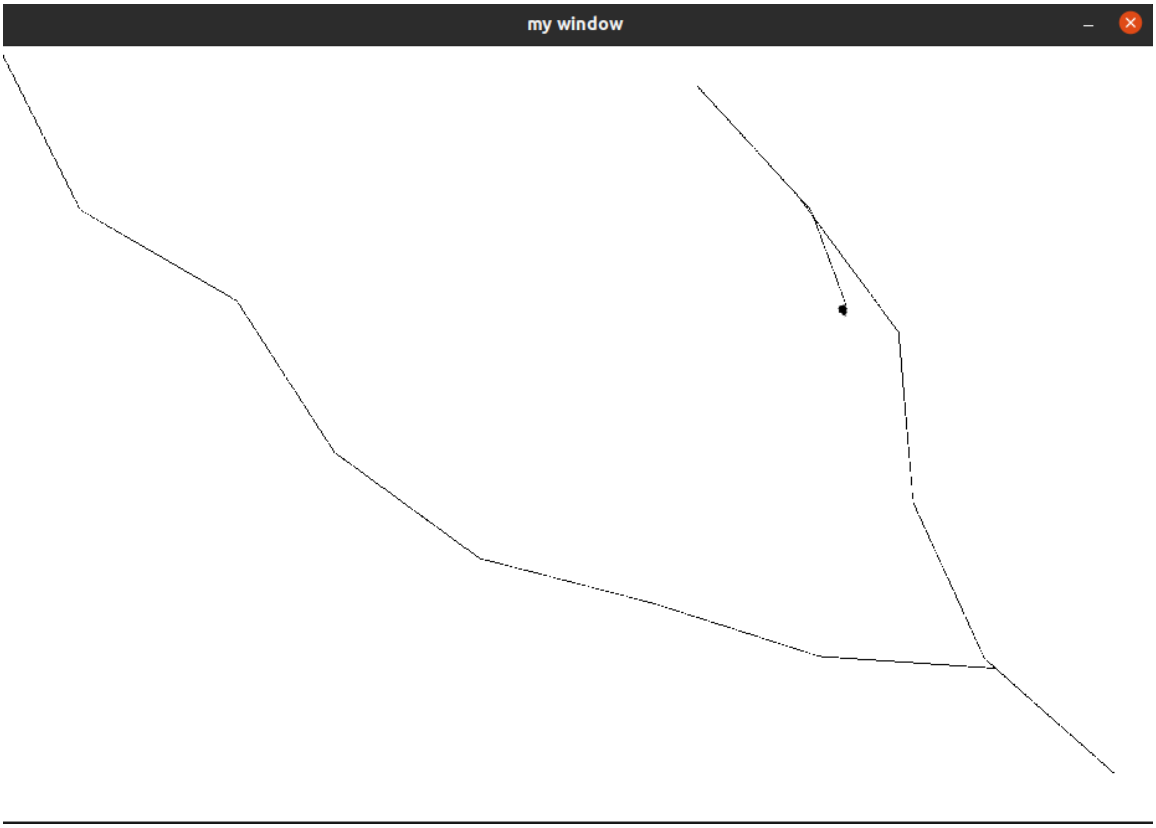


Part3:




my window


my window



Part4: