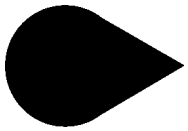
SHARATH SREENIVASAN
STUDENT ID - 200109355

CSC 584 BUILDING GAME AI HOMEWORK 1 REPORT

The main task of the assignment was to explore various movement algorithms. This was done with the help of Processing, an open source flexible software sketchbook and a language to code within the context of visual arts. Processing 3 was used to implement the different types of motion each of which have been described in detail in the parts below.

Each implementation involves a particular shape which is shown below:



Kinematic Motion

The initial step was to implement basic kinematic motion. The shape starts at the bottom left corner of the screen and traverses across the edges until it returns to its starting location. The logic used to implement it is as follows:
Initially the size of the canvas is set to 400 x 400 with the help of the size() method. Additionally the background and the color of the strokes are set to blue with the help of the background() and stroke() method. Initializations take place in the setup() method

Processing's draw() method is called directly after setup(). The draw method is called until the program stops execution. Each time it is called the background is set to white again. The shape is drawn with the help of an ellipse and a triangle by giving appropriate co-ordinates for x and y. Two of the three vertices of the triangle are at two diametrically opposite points on the circle and the third is on the line across the center of the circle parallel to the x axis.

The condition for horizontal movement ($x < 350$ and $y = 350$) is checked and at each instant x is incremented by 5, the ellipse and triangle are redrawn and the bread crumbs are plotted. The breadcrumbs are plotted with the help of a for loop. Every time the loop variable runs up until the the center of the shape and at each instance, a small ellipse is drawn to show the breadcrumbs.

The values of x and y are checked repeatedly and they always fall within a fixed range. To determine the vertical movement on the right side of the canvas, i.e. bottom right to top right, the value of the x co-ordinate is constant and the value of y is lesser than or equal to a value marginally lesser than the width of the canvas (In my case 350). Similarly, for the top right to the top left and the top left to the bottom left (initial position).

I have used co-ordinates to implement the basic kinematic motion. Other methods to implement this basic motion is by using the translate method which involves the transfer of the entire axes to the new point. I used the co-ordinate method since it makes the code readable and there was

no necessity to use the translate method to achieve the basic motion. The bread crumbs are deleted each time the shape reaches a corner.

Seek Steering

In the seek steering section, the shape seeks the location of the mouse click which is implemented using the mousePressed() method.

The position of the mouse clicks is recorded in the target_pos vector and passed as a parameter to the steer method of the SteerShape class which is used to determine the actions and attributes of the shape. The constructor of the SteerShape class initializes the vectors. The Euclidean distance between the target_pos and the the position of the character is calculated and compared with the radius of satisfaction (ros) and the radius of deceleration (rod), If the distance is lesser than ros, the shape is decelerates and proceeds to a halt. On the other hand if the distance to target_pos is greater than rod, the speed of the character is set to a maximum speed(max_vel). The direction of motion is calculated by finding the difference in position of the target and the character's position. Once the magnitude and direction of the velocity is calculated, the update method is called which is responsible for updating the velocity, position and acceleration of the the shape. The acceleration is set to 0 in each case. The display function is called each time to draw the shape on the canvas. The translate method is used to move the axes of reference to the shapes position. The ellipse and triangle are then drawn by using the new origin.

Another method to implement the seek steering is to make the shape seek the point but without the radius of satisfaction and the radius of deceleration. In that case, the shape overshoots the target position and starts oscillating around the target. As a result, I decided to implement seek with the arrive as it was smoother and pleasing to the eye.

One possible enhancement I think that is possible is making the tip rotate gracefully to the target position before seeking towards the target position. As I could see in my code, the movements are sudden.

Wander Steering

The wander is an extension of the seek steering. In simple terms, the mouse click is replaced by a random position on the screen. The position of the target is selected as a pair of random integers in the range of the height and width of the canvas respectively. This selection happens in once in every 25 calls so as to show the motion of the shape towards the random point. The value of count is reset to 0 each time for the next iteration.

If the distance between the current position of the shape and the random position of the target is lesser than the radius of deceleration, then the shape will slowly decelerate to the target (arrive to the target) before it moves to the next random point. The rest is the same as described in the seek algorithm.

Another method, which was explained in class, is to use directions instead of positions. As explained previously, the parameters of the function call to steer are x and y co-ordinates. Instead of this, a random new direction can be chosen and velocity can be applied in that direction. The direction of the velocity is calculated as a difference between the character's current direction and the new direction. The magnitude of the velocity, which is fixed, is then added to the shape in that direction so the shape moves in that direction. If the shape goes across any of the edges, it will re appear on the opposite side. This is taken care of by the border function which compares the position of the character with the width and height of the canvas and according sets the new position of the character.

I feel the second method is simpler as in wander there is no need to arrive smoothly. The shape should just move around the screen. The second method results in more concise code and seems more logical than the first.

Flocking Behavior

In the flocking behavior, the leader is designated a different color (in my case, blue) and all the other shapes seek the leader.

An array list is used as the basic structure which stores all the shapes or boids. Each boid has a certain property to designate it as a leader or not. The first boid that is created is assigned to be the leader of the flock and a flag value is altered to make sure the others are not assigned as the leader. All the boids are created with the starting point to be from the middle of the screen. As it can be seen in the render function, the color of the leader is different.

The borders method takes care of cases when the boids move out of the screen and makes sure it appears on the opposite side of the canvas. The separate method is used to determine the size of the flock. By increasing/decreasing the desiredseparation variable the size of the neighborhood of the boid is altered as it determines the separation distance between the boids of the flock. As it can be seen, the variable 'd' contains the magnitude of the distance between the current boid and all the other boids of the flock. Depending on the magnitude of the separation distance, the boid is added to the flock.

The vel_match is responsible for making sure the velocities of the boids in the neighborhood of a boid are all of the same velocity. As it can be seen, the velocities are added and the average is found out by dividing it with the number of boids in the neighborhood.

The cohesion function is responsible to determine whether to seek the leader or to make the leader wander. The function checks if the lead variable is assigned to 1. If yes, then the leader wanders. If not, then the boid is not a leader boid and it makes the flock seek the leader which is always the first element of the ArrayList. The basic concept is to make the other boids of the flock 'seek' the leader and allow the leader to wander.

However, on implementation I noticed that it takes a short time duration for the leader to reach the front of the flock. Moreover, when the leader goes out of the canvas and appears from the opposite side, the boids towards the end of the flock tend to go straight up to the leader and not follow the path. This caused the output to be a little chaotic. This problem is amplified when the leader exits the canvas at either the bottom right or the top left of the canvas. Parts of the flock are mirrored on the left side and the rest is mirrored on the top. This again creates disorder in the flock. The neighbor distance can be varied to solve this problem but this will result in flocks of smaller size. With an increase in the number of followers, there is a possibility of mini flocks forming with individual velocities seeking the leader.

Another method of implementing the flocking concept I thought was to have the leader as a separate object and make it wander. The other objects are part of an array list and each of them is made to seek the leader. In the event that there are two wanderers, the size of each flock will keep varying as the other boids will seek the wander closest to its position. As a result the output will be distorted.

APPENDIX

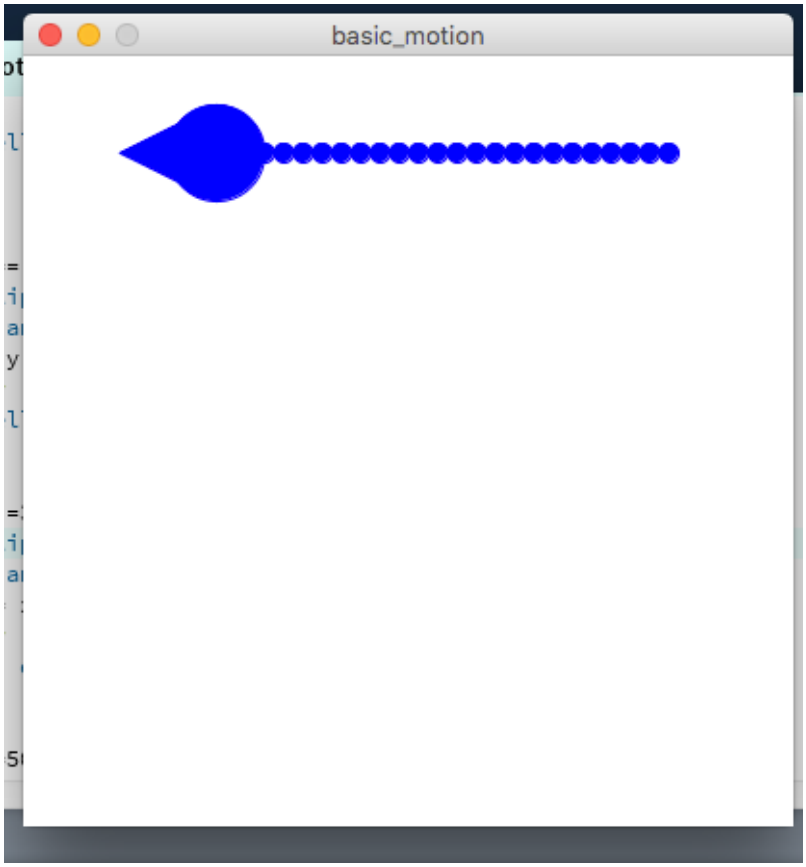


FIG 1 -BASIC MOTION – 1

SHARATH SREENIVASAN
STUDENT ID - 200109355

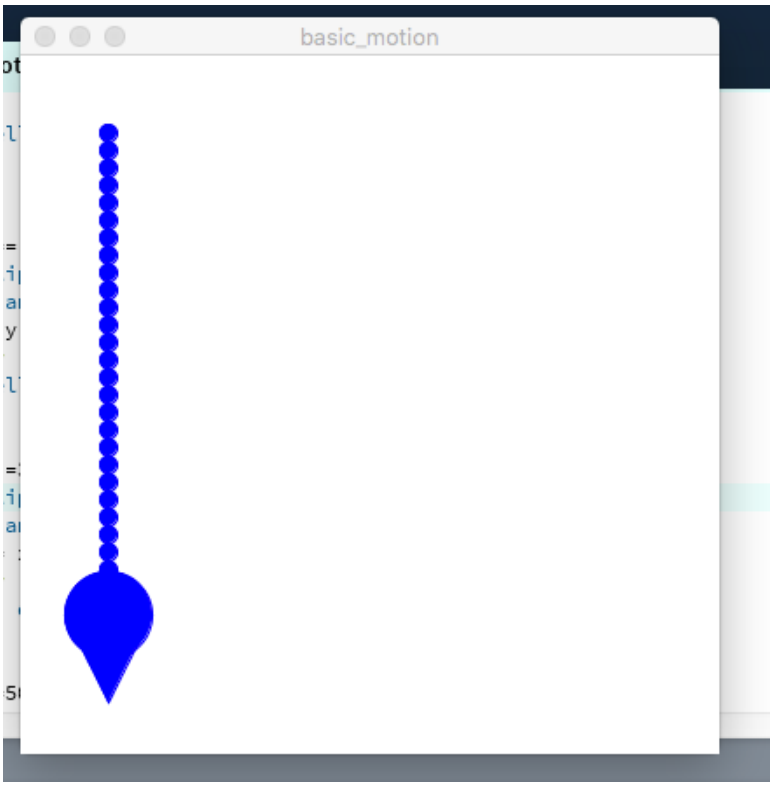


FIG 2 - BASIC MOTION – 2

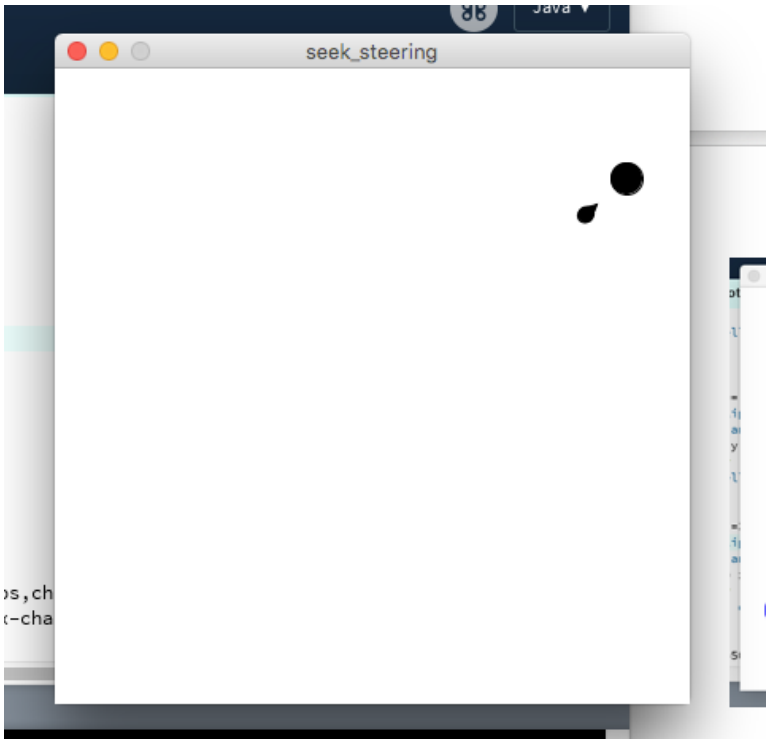


FIG 3 – SEEK STEERING

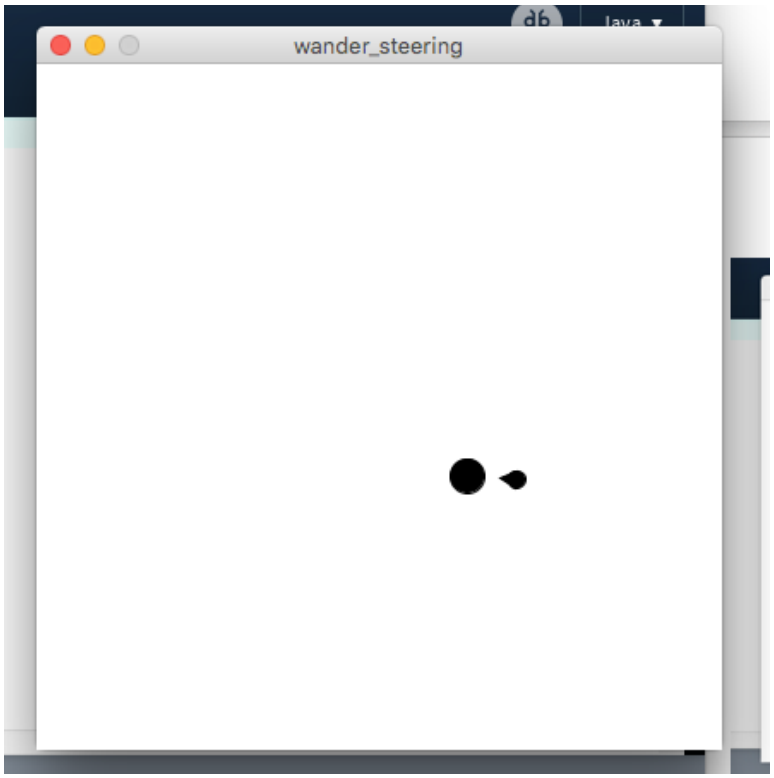


FIG 4 – WANDER STEERING 1

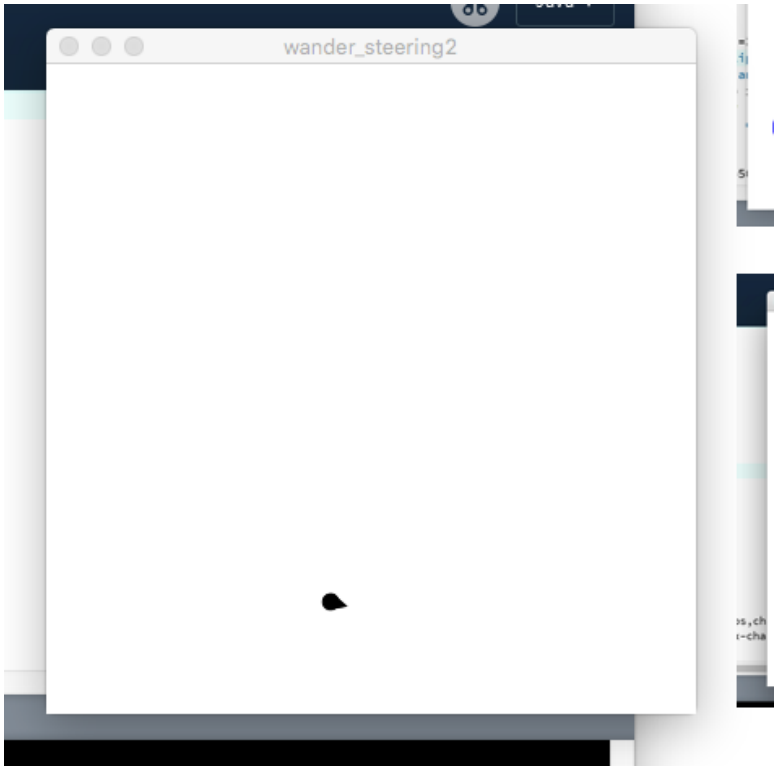


FIG 4 – WANDER STEERING 2

SHARATH SREENIVASAN
STUDENT ID - 200109355

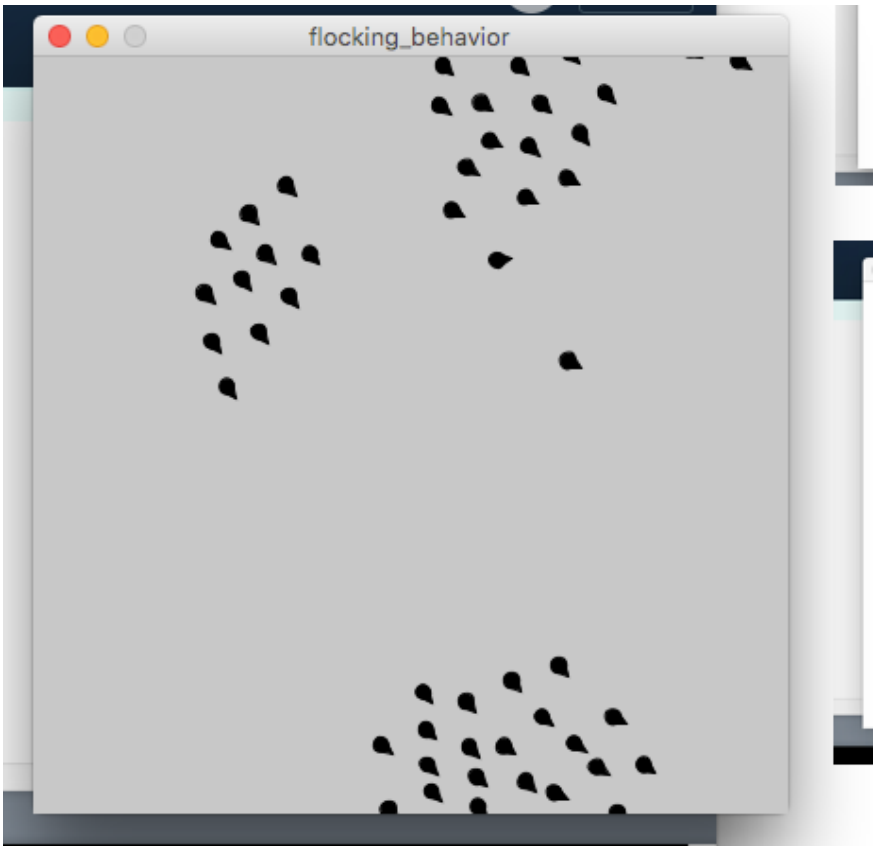


FIG 5 – FLOCKING WITHOUT LEADER

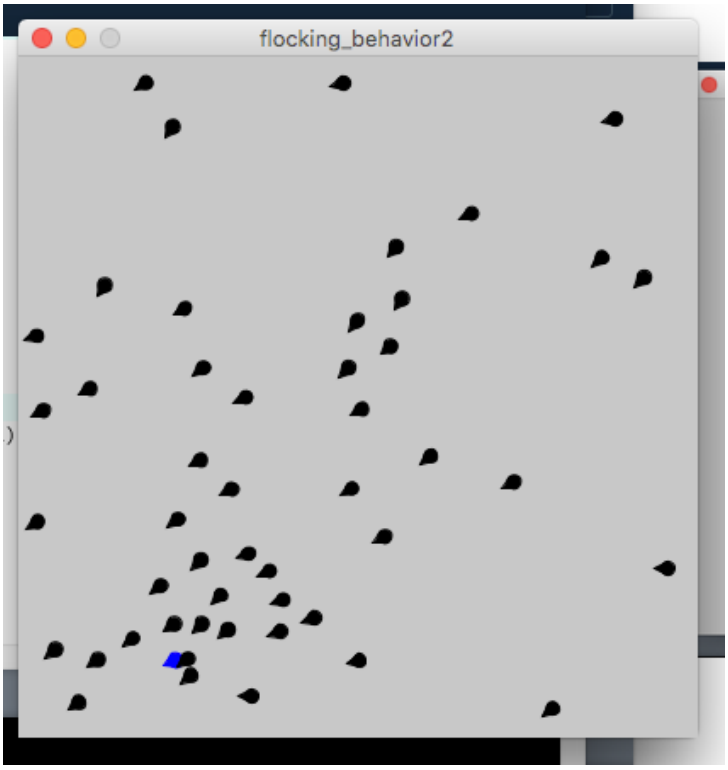


FIG 6 – FLOCKING WITH LEADER