Homework 1 : Movement

Kshitij Patil
Department of Computer Science
North Carolina State University
kspatil2@ncsu.edu

Movement is a basic component of building game artificial intelligence. There are various approaches to get the movement

While implementing various Kinematic and Steering behaviors, a lot of unexpected observations were made. In this write up, I will try to explain all the observations and unexpected behaviours made while experimenting with different parameters and different approaches in getting that particular behavior.

The paper is divided into different sections for each behavior that was coded and all the inferences made while coding it.

1. Kinematic Motion

The first part was the building block of the entire movement code. In this part, I learnt how to PApplet in running the code using Processing library. Once I ran a demo code provided in the documentation available on the official Processing site, it was ready to be modified to achieve Kinematic Motion.

Firstly, the class containing the main function has settings(), setup() and draw() function. This is the main class for all the subsequent parts of the homework. settings() function is used for setting the size of the output window. setup() runs the first when rendering the first frame. Setup() function basically sets up the objects to be created at the start of the program. These objects are then used in the draw() function while rendering the objects after each frame. In draw() function, we firstly set the background color of the output window. Then we call different object functions like update() to update the object parameters and display() to draw the object on the output window. In this part, even the drawBreadCrumbs() function is used to draw small rectangles from all the places that the object has gone through.

I create another class named "Kinematic_data_structure" which consists of the 4 basic Kinematic variables required for kinematic Motion, namely: position, velocity, orientation and rotation.

In this part, I created another class for the object to be displayed. I named this class as "Player_pointer". This class extends the Kinematic_data_structure class. This class basically consists of Player_pointer constructor function where all the Kinematic behaviour variables are initialized. So, the initial position, velocity, orientation and rotation is set. This also has the display() function which draws the individual circle and rectangle representing the pointer object on the screen. This class also contains the drawBreadCrumbs() mentioned before.

Apart from the structure and various functions in the code, there were various design decisions to be made. The first part of the assignment required the pointer to be moving along the boundary of the output window.After initialising the position as bottom left part of the window, I had to decide how to change the position of the pointer and what happens when he has to take a

turn. In the update() function, velocity of the pointer is applied in the direction of motion depending on the current position of the pointer. So, if the pointer is near the bottom of the output window, the pointer velocity will be applied from left to right direction. The update function is not implemented in the way it actually should be done for updating position and velocity.

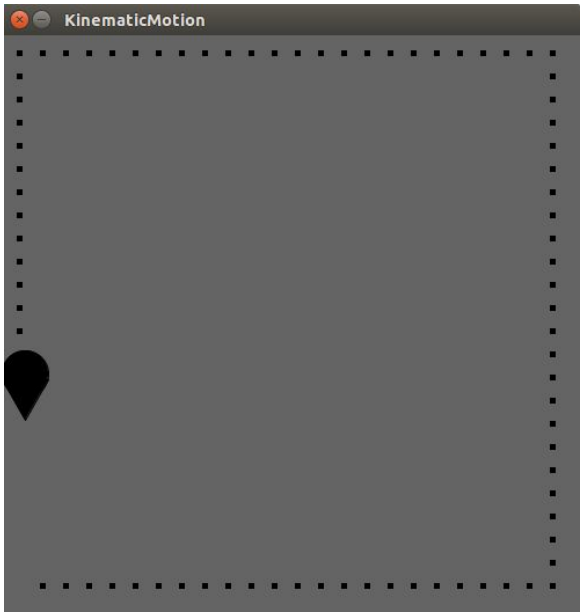

FIG 1. Kinematic Motion (Part 1) output

One design decision I took was at the point when it has to take a turn. Whether to change the orientation of the pointer directly as the pointer reaches any corner or to change it as reaches the corner. I chose to reach any corner first, rotate 90 degrees and then apply velocity again after rotating. That's about it in the Kinematic motion of the assignment.

2. Arriving Steering Behaviors

The second part differs from the first part, mainly, because of the use of Steering_data _structure class which implements the steering data structure. It has 6 attributes namely : position, velocity, acceleration, orientation, rotation and angular orientation.

I modified the Player_pointer class to extend Steering_data_structure instead of Kinematic_data_structure. Secondly, I changed the constructor for Player_pointer to get the initial values from the main function itself. Along with the initial values of the steering behavior parameters , I also included pointer radius functionality that can change the size of the pointer by taking the value from main function. This would eventually help me when I am code for Flocking where I have to display around 50 to 100 pointers on the output window of the same size.

I moved the update() function from Player_pointer to Steering_data_structure. In order to implement the arrive on mousePressed(), I created the mousePressed function which saves the current position of the mouse, save it and send it arrive() function.

In the arrive() function, I set the radius of satisfaction(ros) and radius of distance(rod) and experimenting with these values will give different outputs that I would discuss in the coming paragraphs. I implemented the arrive functionality as it was discussed in class. I calculated the distance magnitude and direction from current character position. Based upon the relation between rod, ros and distance, decided what acceleration to give to the velocity in order for the character to halt at the desired target.

Now comes the interesting part of playing with the parameters. The main parameters which affect the arrive behavior are max_velocity, acceleration, ros and rod. Let me describe some notable observations that help us understand the role of these parameters.

Firstly, after specifying the target, the pointer will move towards the target, go past it and then oscillate to and fro around the target. By studying this behavior, this happens because the maximum velocity is too high to be reduced to zero when the distance reduces from rod to 0.

Secondly, there is a time when the pointer would oscillate to and fro before reaching the target. This happens because the character(or pointer)

accelerates towards target but after it passed the rod, it starts to decelerate to and the accelerate is much greater than the appropriate value. Hence, its velocity changes direction before it reaches the target.

Thirdly, this to and fro behaviour around the target was more prominent in my code because I had not given a constant deacceleration between rod and ros. Instead I had given a deacceleration proportional to the distance from the target. The deacceleration was never enough to reduce the velocity to 0.

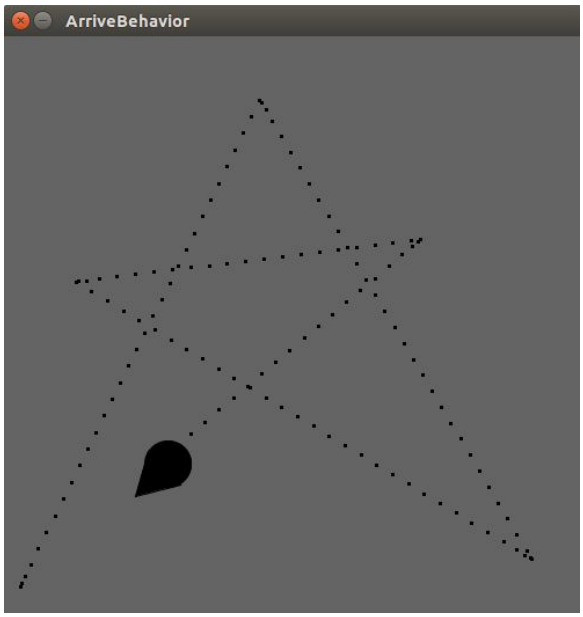

FIG 2. Arrive Behavior (part 2) output

Fourthly, I had not set the velocity to zero once the character enters the ros. So the object kept just touching the target and then go back some distance and try reaching the target again.

These were some of the problems that I faced while writing the code for arrive. In arrive, I used two ways to implement it. First was to reduce the velocity with a constant deacceleration once the character comes closer than rod. Second was a part wise deacceleration where the character was slowed with a greater force between rod and (rod+ros)/2 and a lesser force between between (rod+ros)/2 and ros. The second one gave a more smooth halt at the target.

3. Wander Steering Behaviors

Most of the main functionalities required for wandering were already implemented in the previous subparts of the homework. In this one, I just added a wandering function where a random binomial is selected and multiplies with max_rotation to get a rotation in random direction.

An unexpected problem occured after running this code was the pointer direction used to

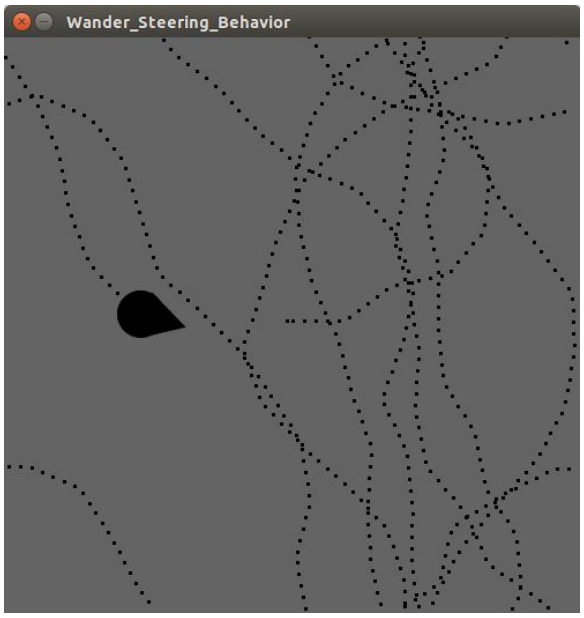

FIG 3. Wander Behavior (part 3) output

flicker a lot between different direction but on an average it would move in the direction of the pointer. I resolved this issue by calling wander() once every 4 frames. The rotation was being applied too many times before the character even moved much on the map.

In wandering, the random orientation can be selected using either one random variable where the orientation value was dependent on the uniform distribution of the random variable. The second way of implementation was to use two such uniformly distributed random variables and subtract them to get a normal distribution which gave an orientation more closer to the current

value with a greater probability. Hence, the character will not flicker much as compared to how it flickered with one random variable.

4. Flocking Behavior and Blending/Arbitration

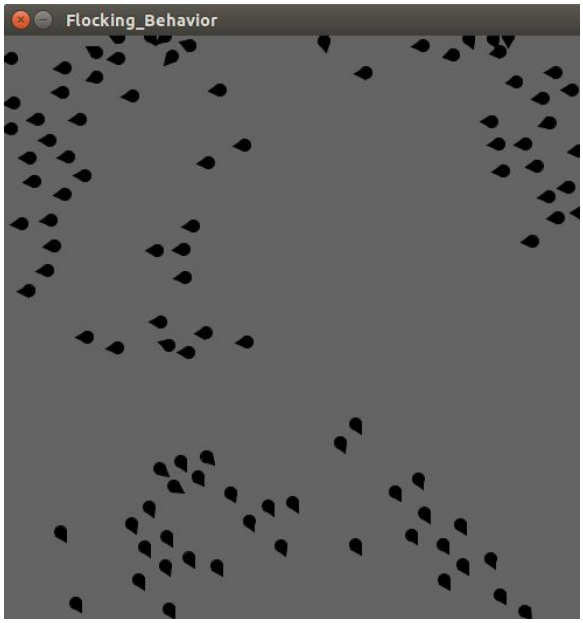

FIG 4. Flocking behavior (Part 4) output

This part was the most interesting one. Firstly, I would like to discuss about the general flocking consisting of the three main forces generating the flocking behavior: Separation, Cohesion and Velocity Match. The weights given to each of these forces affect the flocking behavior giving different results.

Giving separation a very high weight resulted in all the characters to flock around the edges of the output window. At the start of the runtime, every character would go in all the different direction forming a flower like structure. This was because it is at this positions that they would maximize the distance between the adjacent characters. Although the flock_distance used for considering another character to affect the present character was kept very high while testing this. Hence, almost all the characters were affecting each other in the flock.

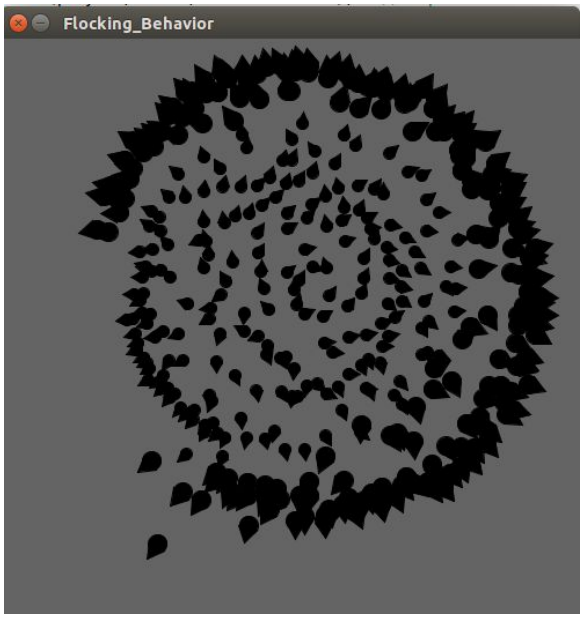

FIG 5. Separation weight too high

Giving cohesion a very high weight resulted in all the characters to stay close to each other. This force would overpower the separation force keeping all the characters at nearby position. Although if we kept cohesion stronger than separation but very close to each other, there will be multiple distinct flocks visible in the output.

Giving velocity match the highest weight gave a

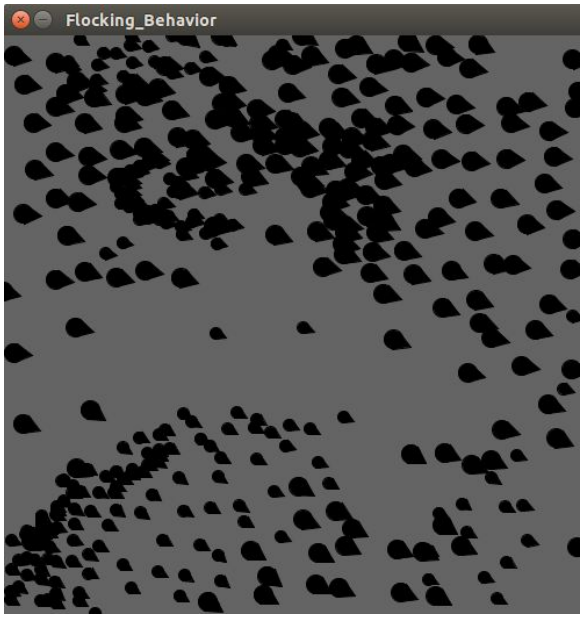

FIG 6. Velocity Match weight too high

very clear motion of all the flock in the same direction from the start of the run time. The separation of all the characters was very slow in this case as compared to when velocity was not this strong. Also, all the characters(or pointers) align themselves to a move in a single direction. This is the general case when velocity match does not have the highest weight too.

Another experiment I tried was that I gave half the characters different maximum velocity and acceleration and see if they form a flock together or not. I ran the code for a very long time but there did not seem to be a pattern in this as I was expecting. I guess “Birds with the same feathers do not flock together”. I am still not sure if I did this right because there might be some other factors other than max_velocity and acceleration that should be same to make these characters birds with the same feathers.

So after experimenting with a lot of parameters, I found the parameters just good enough to get the flocking behavior. As given in class, the separation weight was higher than the others but I would always keep cohesion just near separation to get the close flocks in the output. Velocity match was somewhat not that vital as the characters would eventually align to the same velocity. Although a very low velocity match weight resulted in chaotic behavior.

It is difficult to comment on whether the result was good or bad in terms of more followers. I think flocking effect was more visible when the followers were greater in number but not too high which would make it difficult to decipher.

For the flocking with two wanderers, a lot of tuning of parameters were required to see a distinct flocks following their wandering leader. In this part, only the separation remains like the normal flocking. Cohesion and velocity match will now depend on the attributes of the leader.

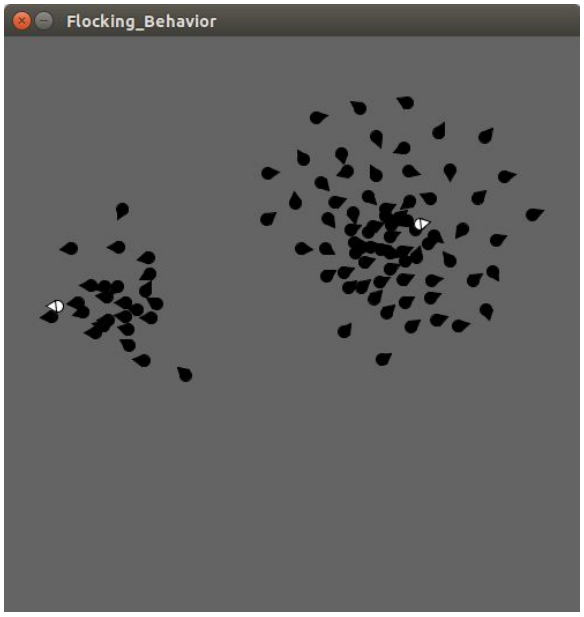

FIG 7. Flocking with wandering leaders

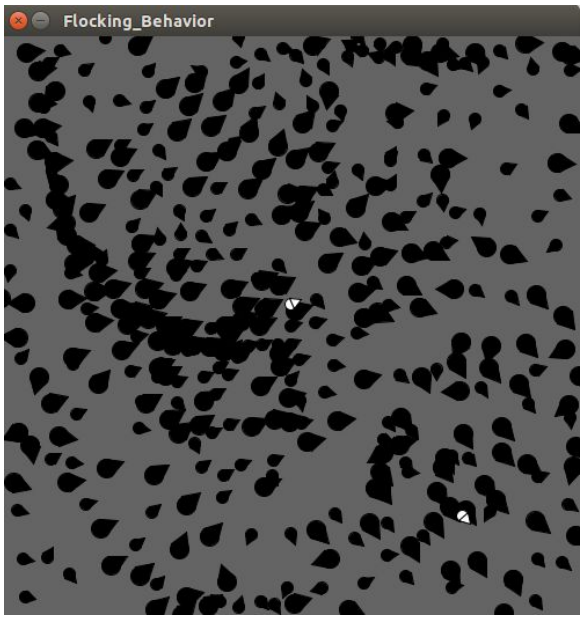

FIG 8. Flocking with wandering leaders and many followers.

REFERENCE

https://processing.org/examples/flocking.html
https://processing.org/tutorials/