Homework 1 Write up

Mohamed Albashir - maalbash

Introduction:

This is a write up about the first assignment for CSC584 Building Game AI. Everything was written by me, using processing and were written using IntelliJ IDEA IDE. As a reference I used the book (Artificial Intelligence for Games: Second Edition), the lecture notes and the documentation on the processing site. At first I created the shape using an ellipse representing the body and a triangle representing the head of the character.

Kinematic Seek:

For kinematic movement I've implemented it using the pseudo code in the book as a reference. There are two speeds that the character can move with, either zero (which means the character is stationary) or max speed. In this part I've implemented two ways to orient the shape towards the goal. One way is to get the orientation of the movement using (velocity.heading()). This approach instantly changes the orientation when the target changes, which goes well with the nature of kinematic movement. The other way is to rotate with maximum rotation until the desired orientation is reached. Both behaviors can be observed by pressing any mouse key, which will toggle the behavior between the two kinds of rotations.

Another thing about the Seek behavior is oscillation which happens when you reach the target and then try to stop. Because switching from maximum speed to zero cannot be done instantly. So the character overshoots the target and comes back and overshoots in the other side. This can be fixed by calculating the distance between the character and the target regularly and update the target before we actually reach the target (5 pixels away for example).

I should also note that I've removed the system time from my calculations at first and changed it to a counter that is incremented with each frame, which made the calculations much more accurate, because with each frame the time change will be constant. And eventually I removed time totally out of my calculations which made everything look much better.

Another requirement that I've implemented which will continue during the entire assignment is the trail of bread crumbs that the character leaves behind as it moves. I will not mention this in further parts of the write up, but the reader should implicitly know that it was implemented. Appendix A has screenshots about the kinematic movement.

Steering Arrive:

For this movement I've also used the pseudo code I found on the book as a reference, plus some of the pseudo code in the class notes. Unlike kinematic, with steering behaviors we have the elements of linear and angular accelerations, which make the movement more lifelike. The

character starts with zero speed and then increases its speed until it reaches a certain distance from the target. This distance from target is called the radius of deceleration and it's what differentiates seek and arrive. Once the distance between the character and the target is less than that radius, the movement speed of the character is reduced by a parameter called the time to target velocity. Until the character's distance from target reaches another radius called the radius of satisfaction, after that the character's speed is reduced to zero and the character stops almost instantly. These radii and time to target velocity are chosen intelligently to cause the movement to stop when the character reaches the target.

At first I've chosen a rather big radius of deceleration and a small radius of satisfaction which made sense because thinking about it, that's more likely to make the character stop at the target and not overshoot, or stop before it. However, I didn't know how relatively big the radius of deceleration should be, and I didn't keep in mind how the high speed of the character is going to affect the behavior. At first The radius of deceleration was 2 times the radius of satisfaction. This caused the character to oscillate a bit. So instead I chose a radius of satisfaction equal to the max speed the character can with, and a radius of deceleration that's 4 times the radius of satisfaction. I also reduced the time to target velocity a bit because I didn't want the movement to be too slow. This led to a behavior that was perfect and made the character completely stop when it reaches the point I pressed the mouse at.

Till this point I only concerned myself with changing the position of the character, I then shifted my attention to changing its orientation. I used the Align algorithm discussed in class, which is almost the same as arrive. This one however, was trickier than arrive. I kept having the character overshooting its intended orientation and then keep rotating to get back on track. This was a problem with the selection of parameters. Because we're changing the orientation every frame or every couple of frames, we sometimes never reach the desired orientation, or never reach a point within the angular radius of satisfaction. That's the reason for the multiple 360 degrees' rotations that happens sometimes. This is fixed with proper parameterization. And a radius of deceleration that's 5 times bigger than the radius of satisfaction, and a relatively big time to target rotation value. Appendix B contains screenshots showing this behavior.

Steering Wander:

This is the first movement which contains another kind of movement algorithm as a part of it. The basic idea is that we want the character to randomly roam around the canvas. The challenges that arise contain the setting of the character's target and computing the orientation. The way this is done conceptually is by imagining a circle that's positioned in-front of the character. And then randomly choosing a point on the circumference of the circle and have the character "arrive" at it. "Randomly" here refers to binomial random, which is not uniform, but has a more probability to be close to 0 rather than 1 or -1. Later in the write up I'll mention another implementation of random in Wander. The radius of this imaginary circle is called the Wander radius, and the distance between the character and the circle is called the

Wander offset. Another parameter that we need to set is the wander rate which will be mentioned again below.

First thing to do is to get the orientation of the target so we can use that to “align” the character’s orientation with it. Next is to get the position of the target, which is the orientation we got previously, multiplied as a vector by the wander offset and the wander radius. Now that we have the target orientation and position we simply “arrive” at it and “align” to it.

One problem with wander that must be addressed is the issue with the character moving out of the canvas. This can be fixed by creating pads on the edges of the screen and have the character just bounce back, or when the character gets close to the edges, it can change its orientation inwards and stay inside the canvas. I however, opted to make the movement continue by having the character appear at the other end. This in my opinion makes it look as if the character didn’t change its movement or orientation, it just kept going on its way and we simply moved the screen to see what’s happening to it, as if we moved to a new canvas because the character got out of the old one.

Another implementation of wander that I’ve done is the use of uniformly distributed random instead of the binomial random. With this kind of random, the character is less likely to move close to the center and is more likely to roam around more. The previous method sometimes looks better, especially with the method of handling edges that I ended up going with. However, the second one sometimes provides a wider coverage of the canvas. But, since that’s not an objective of this movement, both are okay to use. Appendix C shows both kinds of the Wander algorithm.

Flocking:

In Flocking we’re basically using the previously implemented movement algorithms to have multiple character move in a manner close to the flock of bird movement. To implement this movement, I used Boid’s algorithm, described by Reynolds. As a reference I used a pseudo code I found online that details the three behaviors that are needed to implement flocking. The three behaviors are referred to as: Cohesion, Separation and Velocity Matching. Details for each are listed below:

Cohesion: Here I simply calculate the position of the center of mass for each Boid (character) which is the average of all other positions, then I get a vector that is this center of mass minus the character’s location.

Separation: Here I have a radius for each character and against that radius we check the distance between the character and all other boids. If the distance is less than the radius, then calculate a vector that is in the opposite direction of that boid.

Velocity Matching: Here we basically get the average velocity of all other boids, and return the difference between that average and the character’s current velocity.

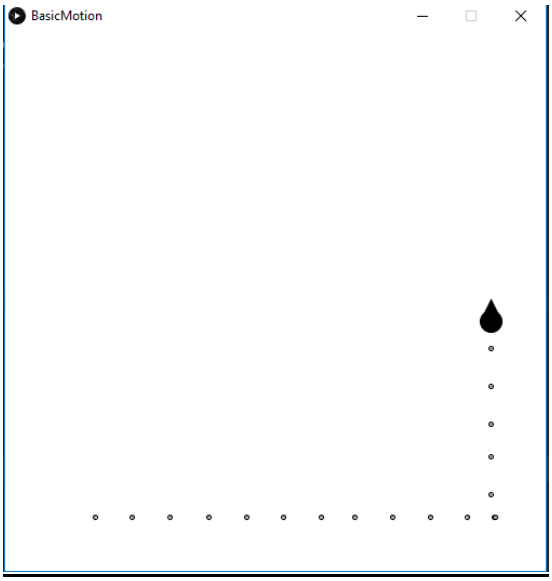
Once all that is done, we normalize those three vectors, multiply them by the goal speed (calculated by the arrive algorithm), and give each a weight. The weights have been chosen in a way that leverages separation more than cohesion and velocity matching to prevent collisions from happening.

Here there is a complication that should be mentioned. Due to previous decision of having the wandering character appear at the other end of the screen when it's out of canvas, this causes the flock to change its direction and pursue the character at the far end of the screen instead of following it out of the canvas and appearing on the side behind it. One solution to that is to either have a tolerance area for the leader, where the leader can go out of the canvas, but still not change its position immediately, until the followers are also out of the canvas, and then they'd all appear at the other end, or change the behavior and have the leader change its orientation if it reaches the edges of the screen. The second solution seems more appealing aesthetically.

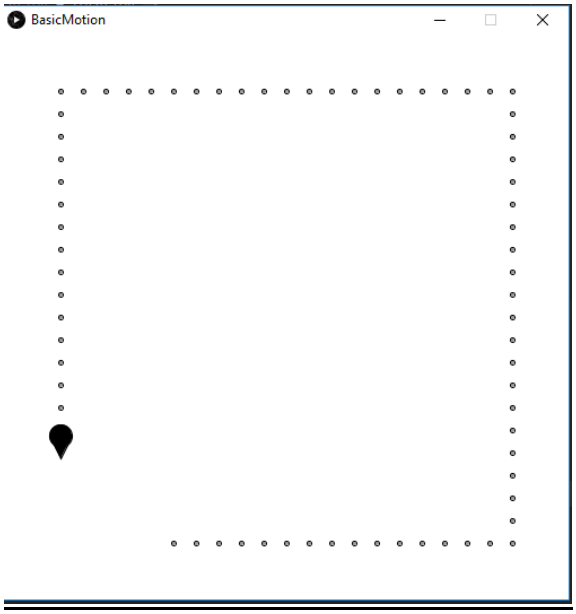
Another behavior that was implemented was having the followers split from their flock and follow another one when the leader of another flock is closer to them. This behavior can clearly be seen especially when a leader goes out of the canvas and becomes really far from its followers, they immediately abandon the wanderer as a leader and go for the closer one. I've also implemented the functionality of spawing new boids using mousePressed() method, and having them follow the leader closer to the spawning location (which is the mouse location).

Finally, there was the problem of having too many followers. That presented a difficulty in the separation behavior. The more the number of boids, the more difficult it is to avoid colliding with one another. Perhaps further tweaking of parameters might do the trick.

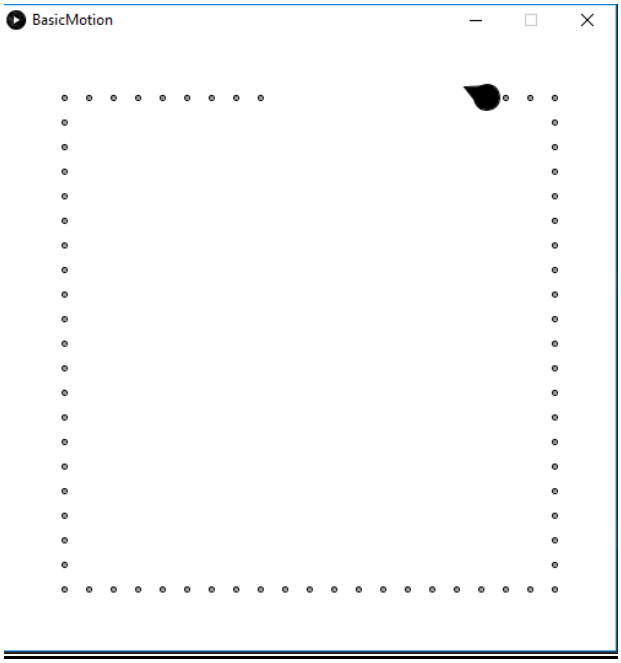
Appendix A:



Kinematic motion, with some oscillations that can be seen on the right corner from the bread crumbs trail.

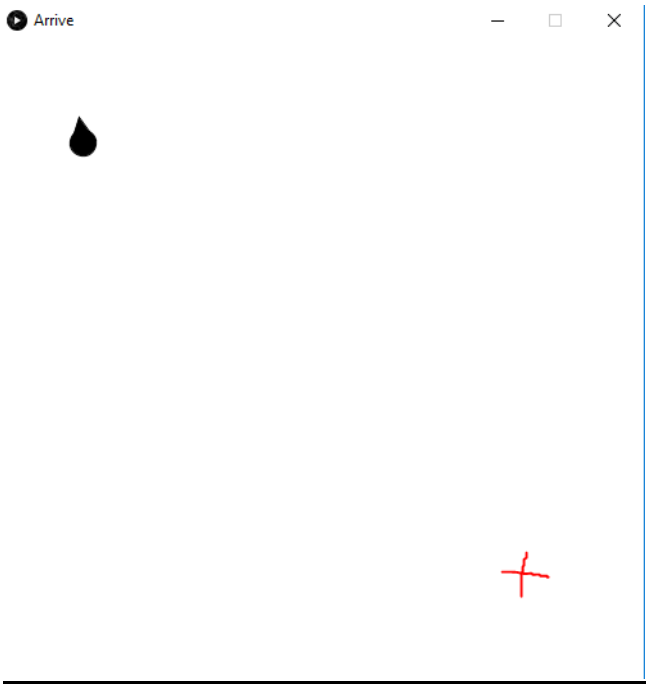


Kinematic movement without oscillations using instant change in orientation.



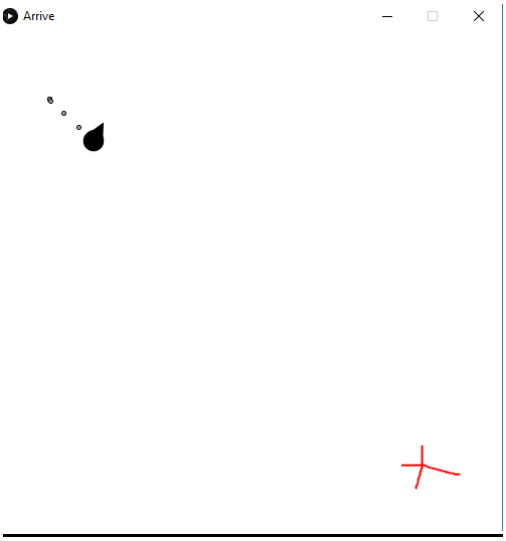
Using rotations with kinematic. (this screen shot was taken during orientation change)

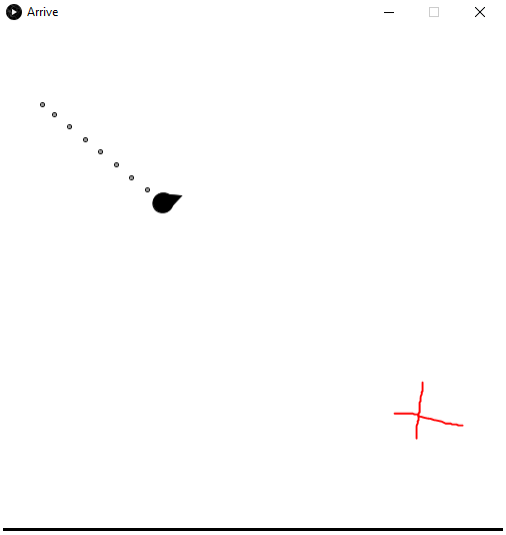
Appendix B:

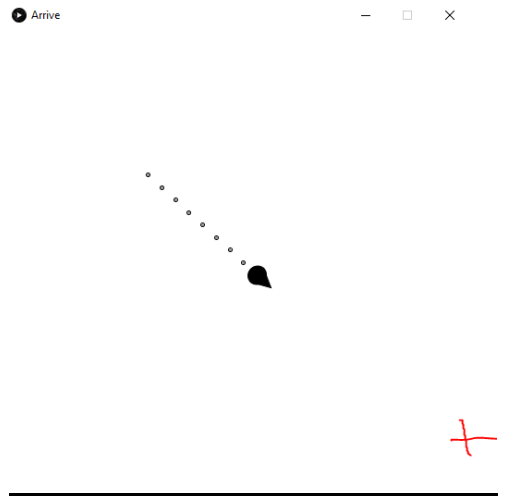


This is a screenshot immediately after pressing the mouse on the red cross location.

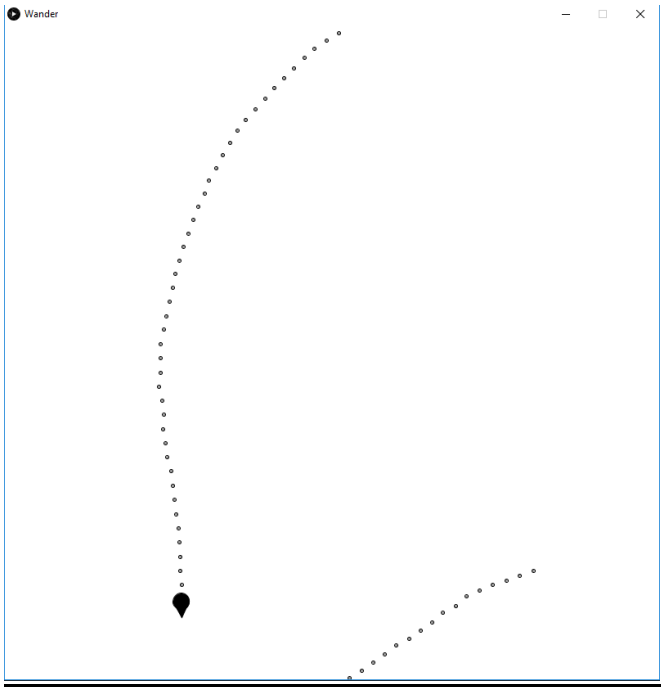
The following 3 screenshots show the movement and change in orientation


Arrive

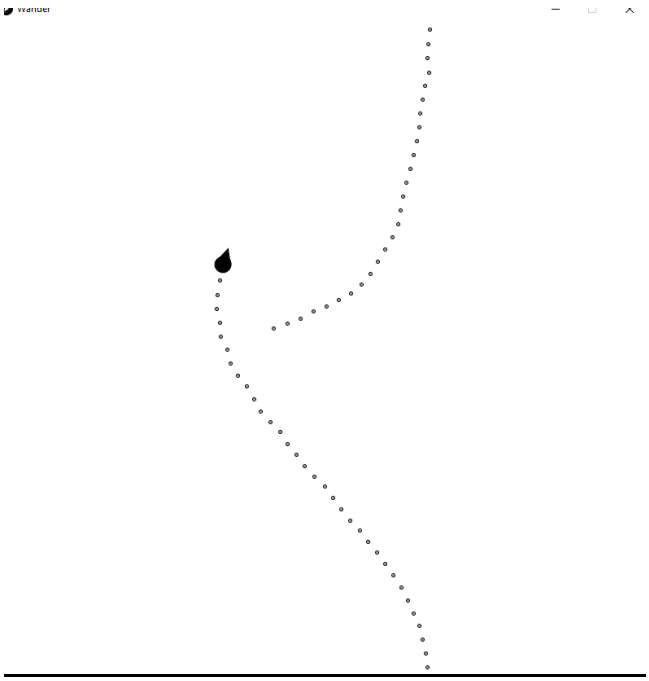

Arrive


Arrive

Appendix C:

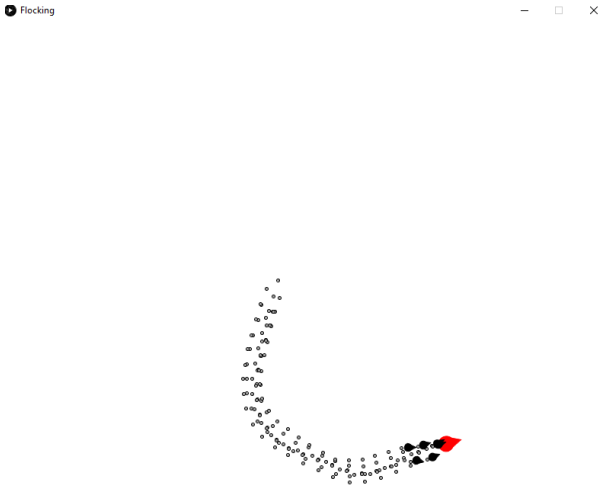


Wander with random binomial

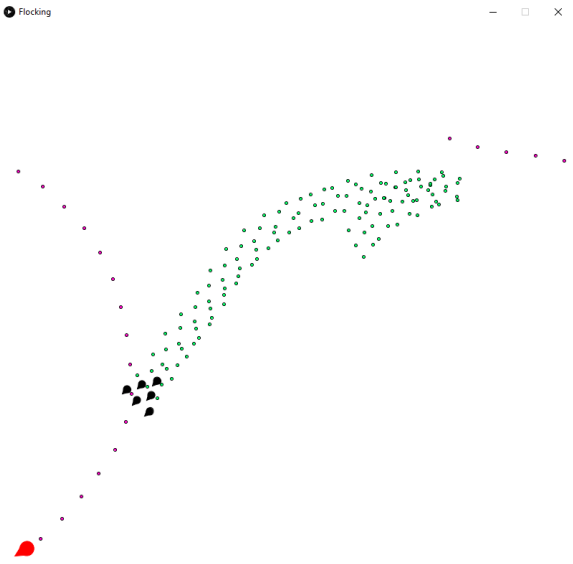


Wander with uniformly distributed random

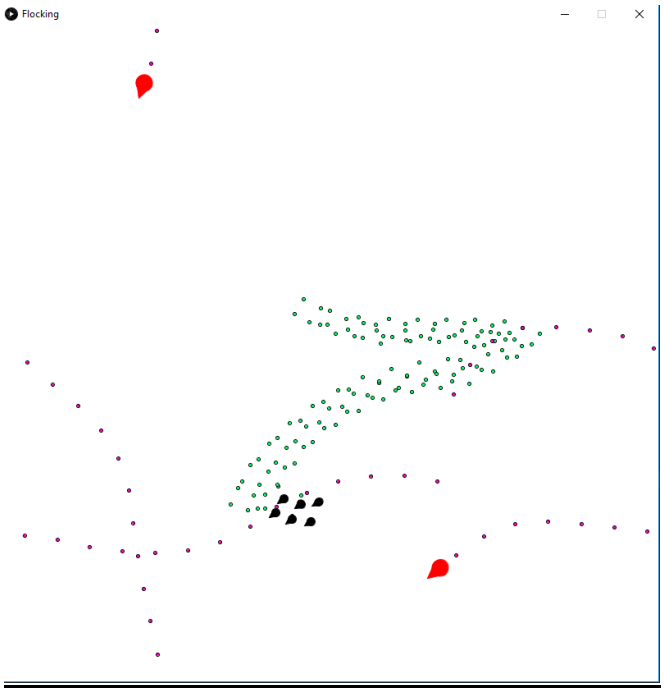
Appendix D:



Screenshot shows a wandering leader with followers trying to arrive at its location



When leader is out of the canvas and position changes.



Flock changes the leader they follow from the one at the top to the one at the bottom, because its closer to them.