U1212719: Mit Doshi

Movement Algorithms:

In this assignment, our task was to implement and evaluate how different movement algorithms work. The major point in doing so was to make us understand how the movement is actually the lowest level in the AI hierarchy. Being the lowest level the best movement algorithm should always be O(1) which makes sense as the lowest level should have the least complexity possible to handle more complex behavior at the top efficiently. Movement Algorithms are divided into 2 parts:

- Kinematic Algorithms: Algorithms that assumes that any object travels at its max possible speed always
- Dynamic Algorithms: Algorithms that considers current distance and velocity for calculations.

These Algorithms affect 4 different components of any object which are, position, orientation, velocity and rotation. These components combined together to form the kinematic structure

System Architecture:

For this course, I am using Open Frameworks with c++ using visual Studio 2017. For setting and my movement algorithms, I have tried creating them using multiple structures before finalizing the current structure which works well with movement algorithms. First I had tried creating different class for each algorithm, but that made me realize that I am defining character and target way too much and that is wastage of memory then I shifted that to a single class that hold functions for each algorithm and I just pass the Kinematic (Details of objects in world space) values, the function returns me the Steering output (changes to object in world space). The image below shows how I am declaring my functions for movement algorithms.

```
static KinematicStructure UpdateKinematic(KinematicStructure iObject, float iDeltaTime, SteeringOutputStructure
  iSteering);

static float GetNewOrientation(float iOrientation, ofVec2f iVelocity);
static SteeringOutputStructure KinematicSeek(KinematicStructure iCharacter, KinematicStructure iTarget, float
  iMaxVel);
static SteeringOutputStructure KinematicArrive(KinematicStructure iCharacter, KinematicStructure iTarget, float
  iMaxVel, float iTargetRadius, float iTimeToTarget);
static SteeringOutputStructure KinematicWander(KinematicStructure iCharacter, float iMaxSpeed, float
  iMaxRotation);
static KinematicStructure UpdateDynamic(KinematicStructure iObject, float iDeltaTime, SteeringOutputStructure
  iSteering, float iMaxSpeed);
static SteeringOutputStructure DynamicArrive(KinematicStructure iCharacter, KinematicStructure iTarget, float
  iMaxVel, float maxAccel, float iTargetRadius, float slowRadius, float iTimeToTarget);
static SteeringOutputStructure DynamicFlee(KinematicStructure iCharacter, KinematicStructure iTarget, float
  iMaxAccel);
static SteeringOutputStructure DynamicEvade(KinematicStructure iCharacter, KinematicStructure iTarget, float
  iMaxAccel, float iPersonalRadius);
```

Fig 1. Algorithm Structures

As you can see I am using static functions to return steering output which helps me to call any movement algorithm anywhere, this had helped while creating flocking algorithms which I will be discussing ahead.

Seek:

Seek is one of the basic position changing algorithm. This algorithm makes the character move to a target point in world space. Seek has both kinematic and Dynamic version, Kinematic Seek affects the velocity of character which in turn changes the position, whereas Dynamic Seek affects the acceleration which then affects the velocity. There is also one problem with the seek algorithm is that seeking can result in repetitive overshooting, which leads to the creation of infinite seeking of the object and will never reach the target.

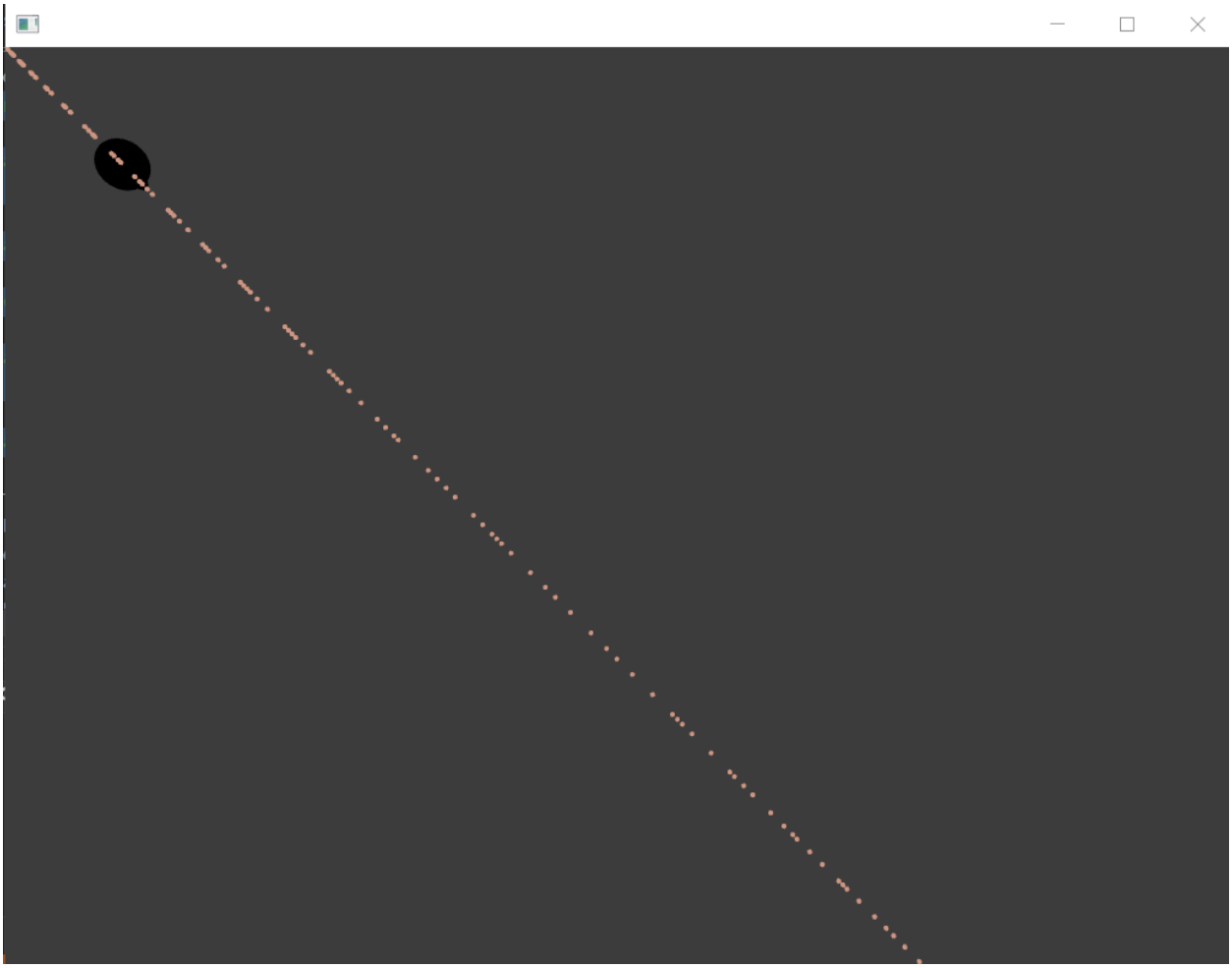

Fig 2. Seek Creating Infinite Seeking to target

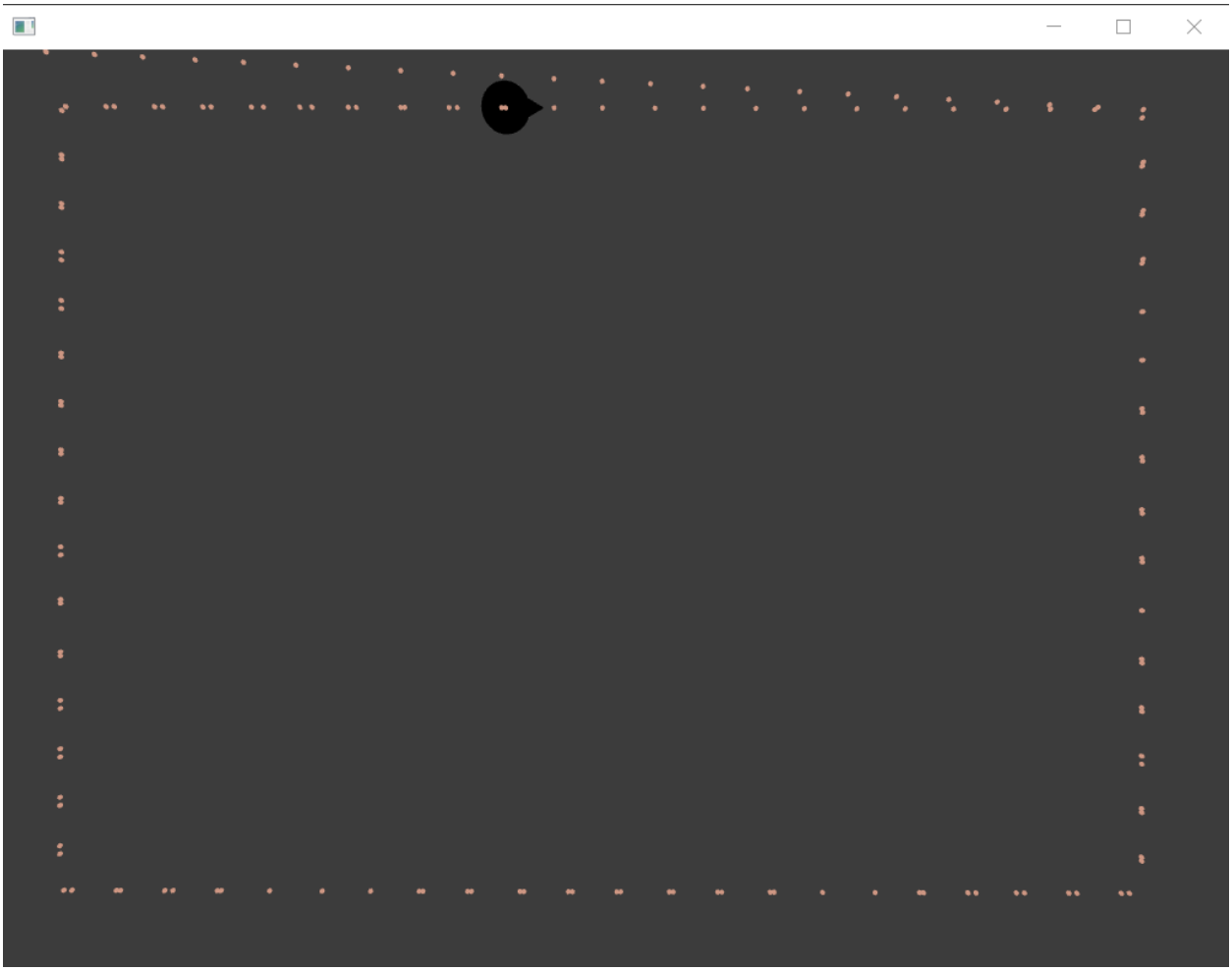

Fig 3. Seeking the four corners

There is a similar but opposite algorithm called Flee, which moves away from target, which also has an infinite condition on when to stop the algorithm.

Arrive:

Arrive like seek is a basic movement algorithm that changes the position. The working of the algorithm is what makes it overcome the problem with seeking algorithm. It uses the radius around the target to know if the player has reached within the scope of the target if so then slow down. Kinematic Arrive using a single radius when the character is inside the radius stop movement. Whereas Dynamic Arrive uses two radii and the character gradually slows down after passing the outer radius and would stop when it reaches the inner radius. Because of this, it gives us a smooth transition to target. I also figure out that Arrive in both cases needs a proper target radius if you underestimate the target radius than the boid will overshoot as it does in seek.

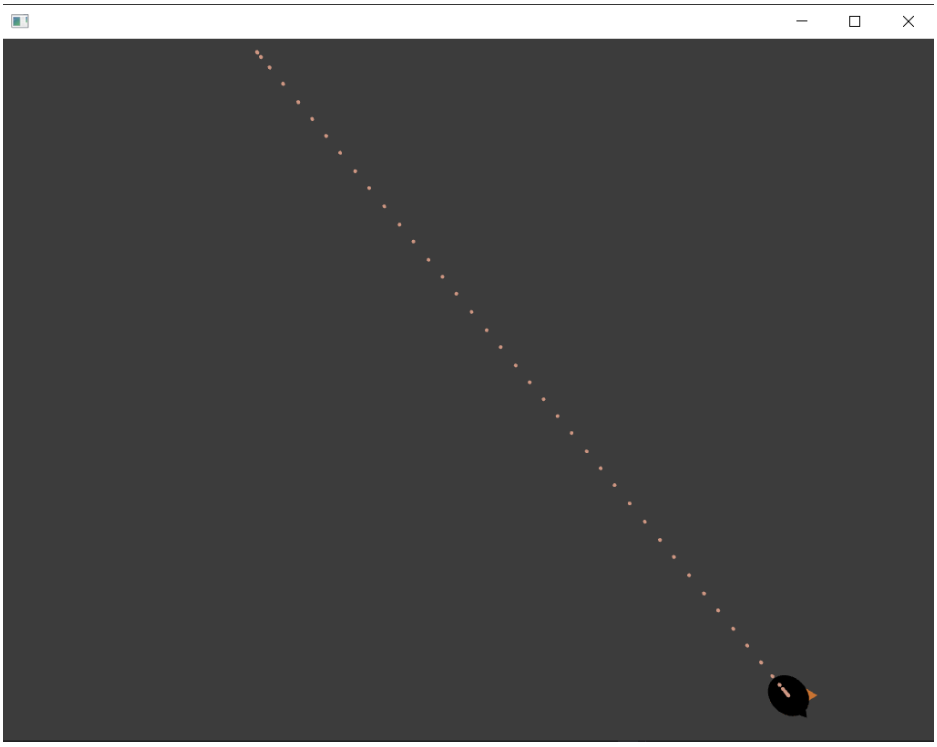

Fig 4. Kinematic Arrive

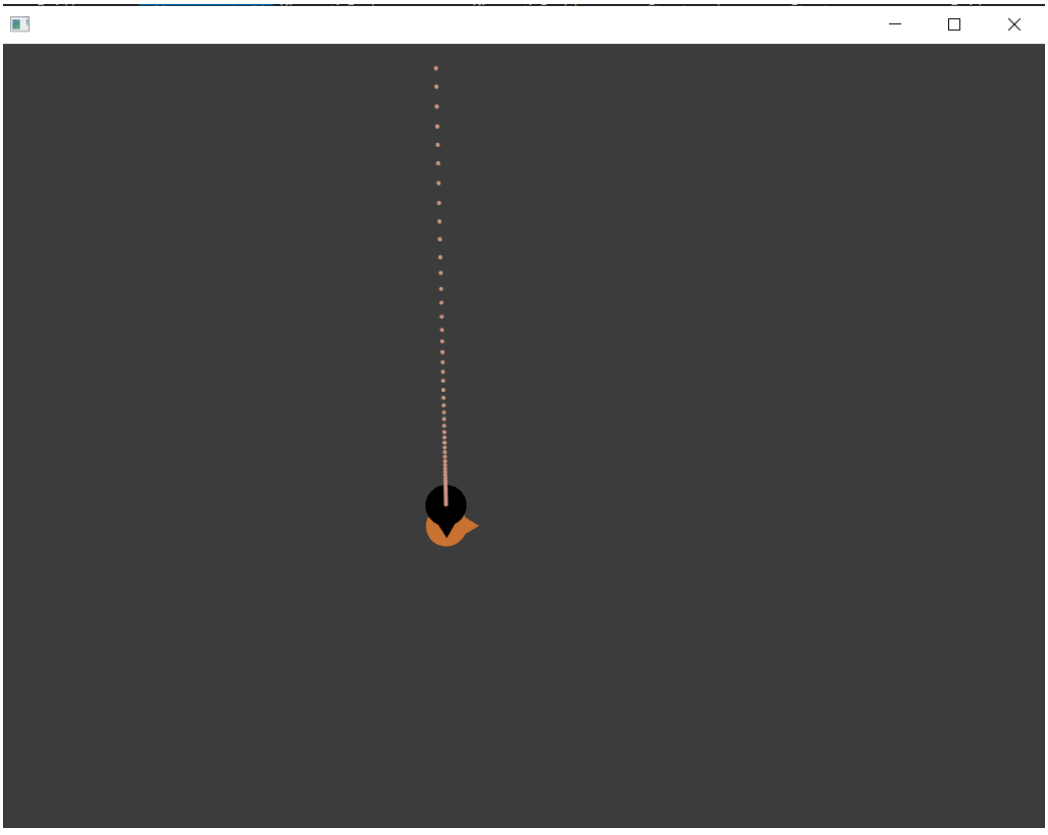

Fig 5. Dynamic Arrive

Align:

Align is an orientation match algorithm and it does not affect the velocity and position of the character. This algorithm aligns the character to the orientation of the target. There is one thing that the person needs to take care while creating this algorithm is the direction from which the character is turning to. For this, the difference in orientation must be in the range -Pi to Pi so that we can take the shortest movement to target orientation.

Velocity Match:

Velocity match is a super simple and basic algorithm that matches the velocity component in the kinematic structure. This algorithm just matches the velocity of character with the target velocity.

Face:

The face is an advance movement algorithm that delegates to Align. This algorithm uses align to face towards the target that the character is moving towards

Wander:

Wander is an advance movement algorithm that delegates Face. This algorithm moves in a random direction which it is facing. For me Dynamic Wander was not giving me the proper result, it was changing direction like taking U-turns, after changing different values I got a result where it is making a snake-like pattern which is also fine. But my kinematic Wander gave me the output results that I want in my algorithm.

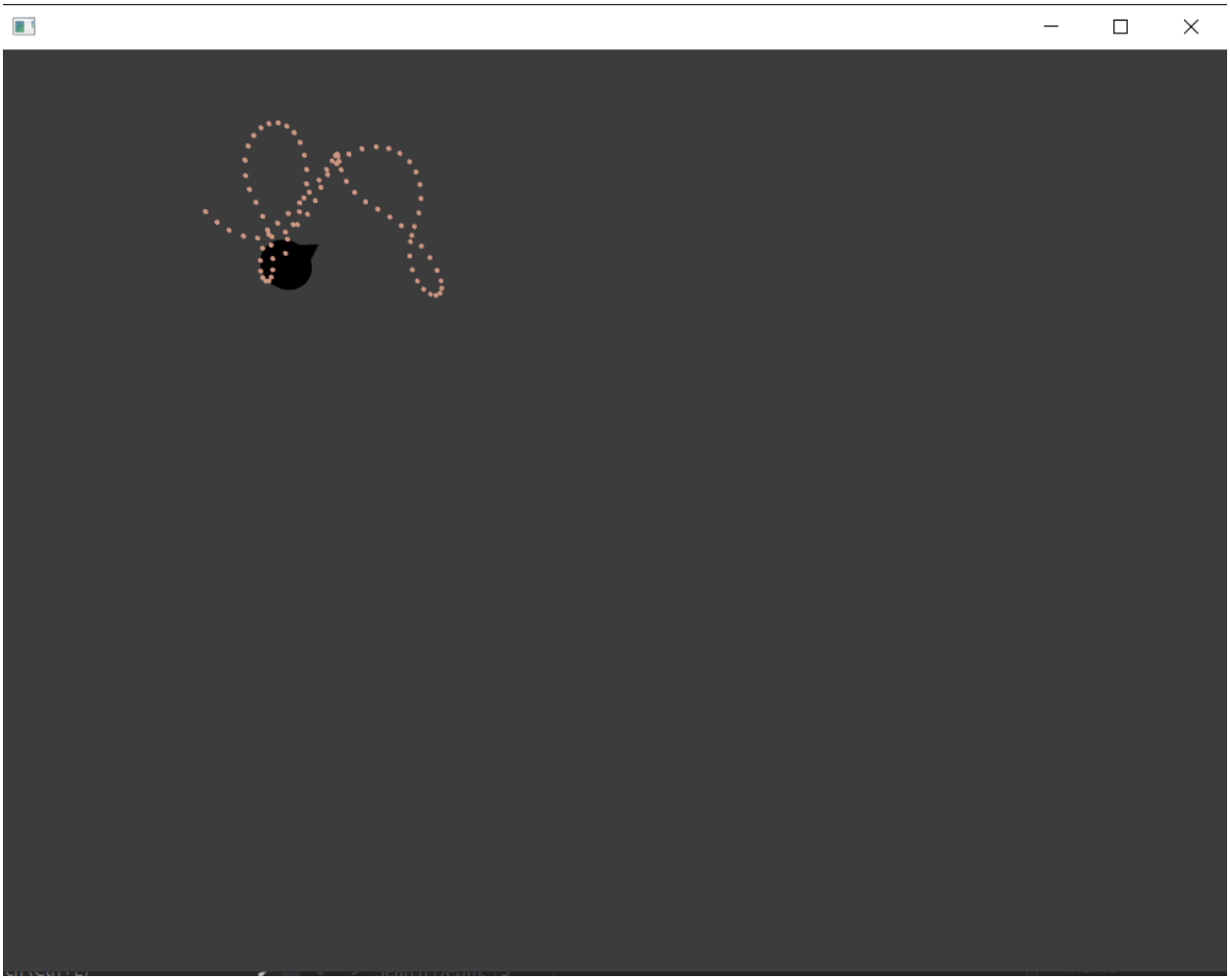

Fig 6. My Dynamic Wander just keeps on creating &

Fig 7. Kinematic Wander

Flocking:

Flocking is a complex movement algorithm, [an algorithm that is achieved by blending or arbitration of two or move movement algorithms], Flocking is created by Blending 4 different types of algorithms, namely Arrive, Look Where you are going, Velocity Match and Separation. Separation delegates to evade which in turn delegates to flee. So, Separation is basically character flee from all other targets. The blending of flocking determines the behavior of the flock, I tried multiple values for blending the flock and I got interesting results, which are shown in the images below. I also tried doing it without separation algorithm at all and all it did was gather at one point; thus, I realize that separation is actually not only making it look organized but also keeps the center of the mass of the flock changing thereby making it not static.

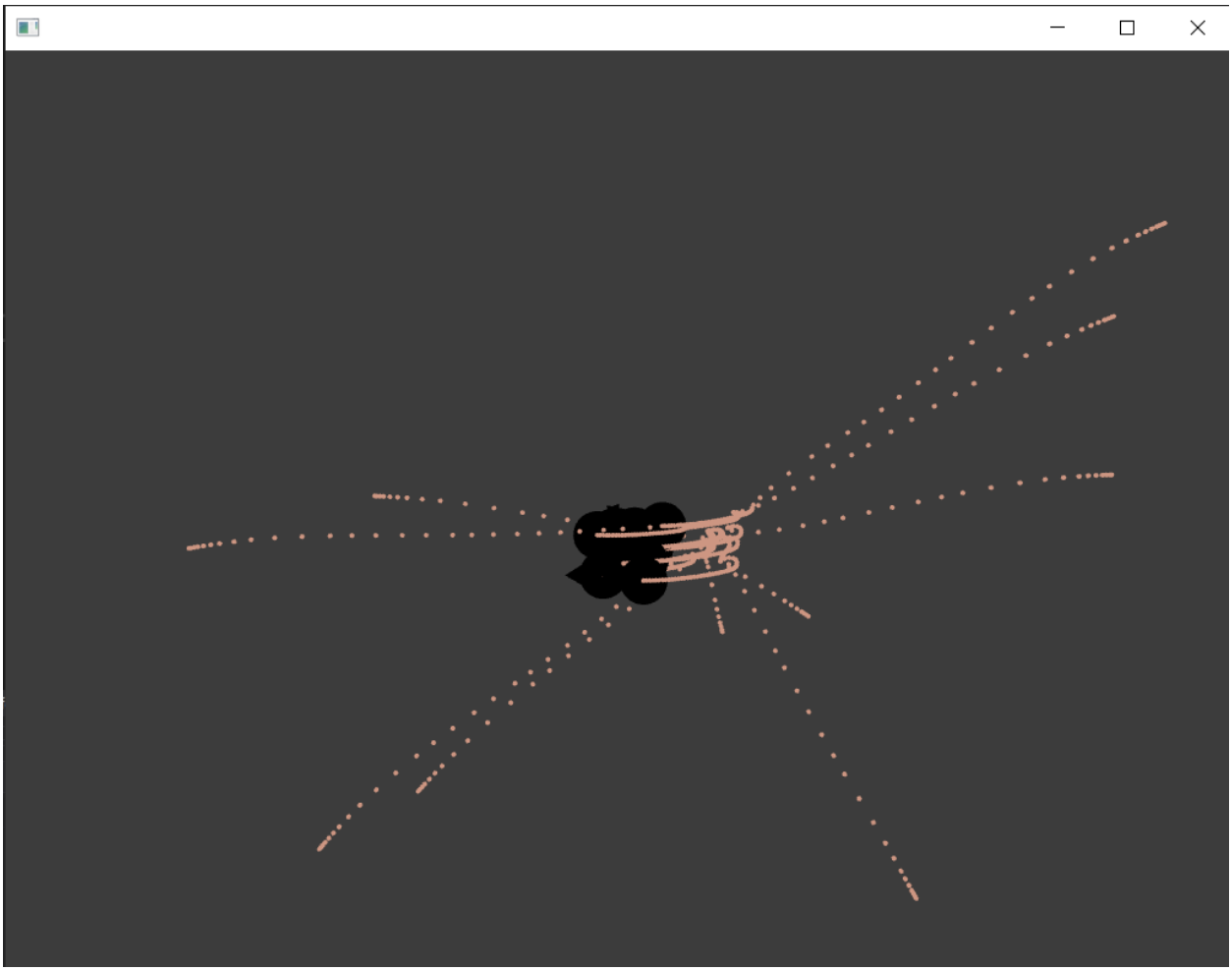

Fig 7. Flock when the Arrive algorithm has the maximum weight in blending

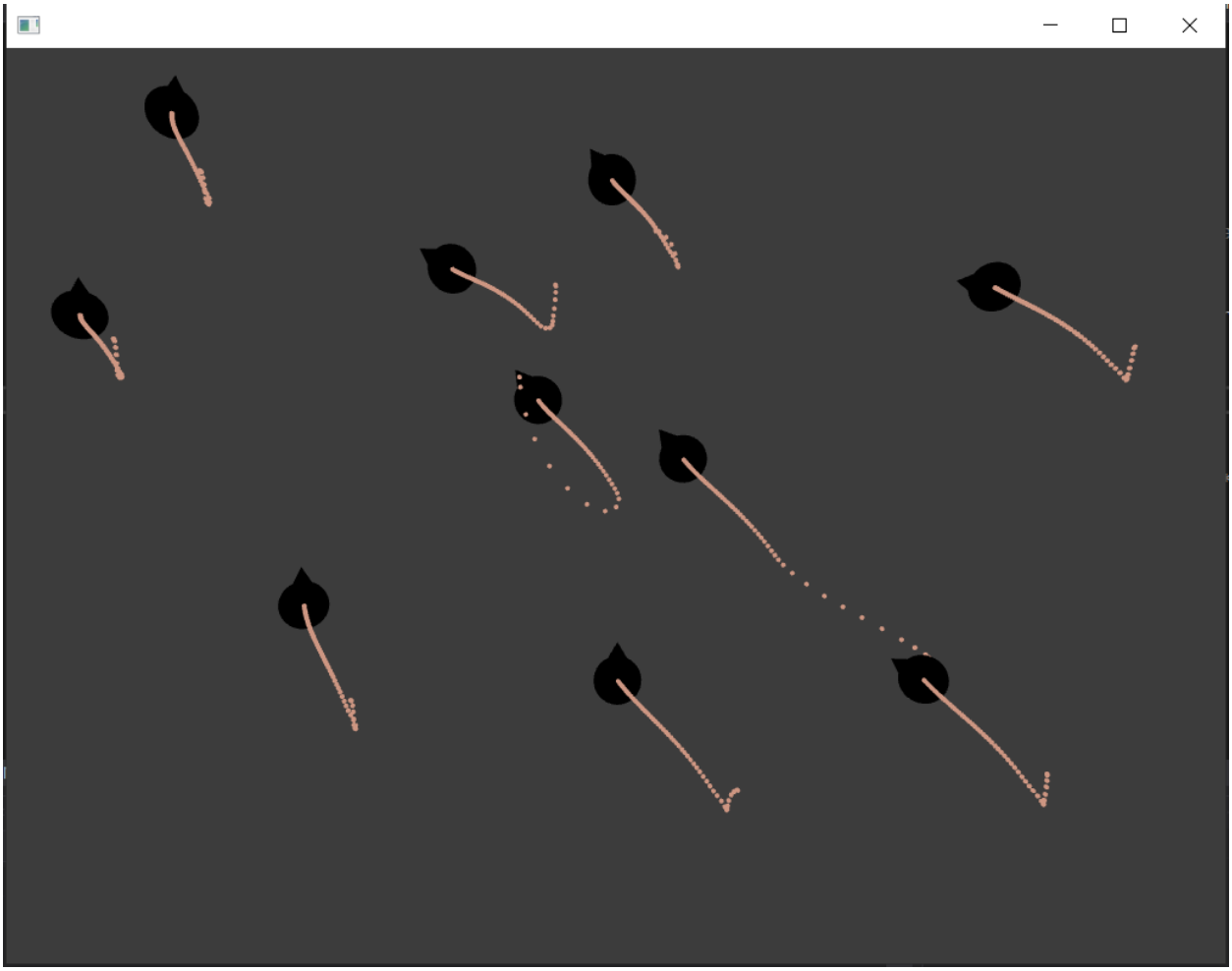

Fig 8. Flock when the weight of separation overweighs the seek by too much (1:10)

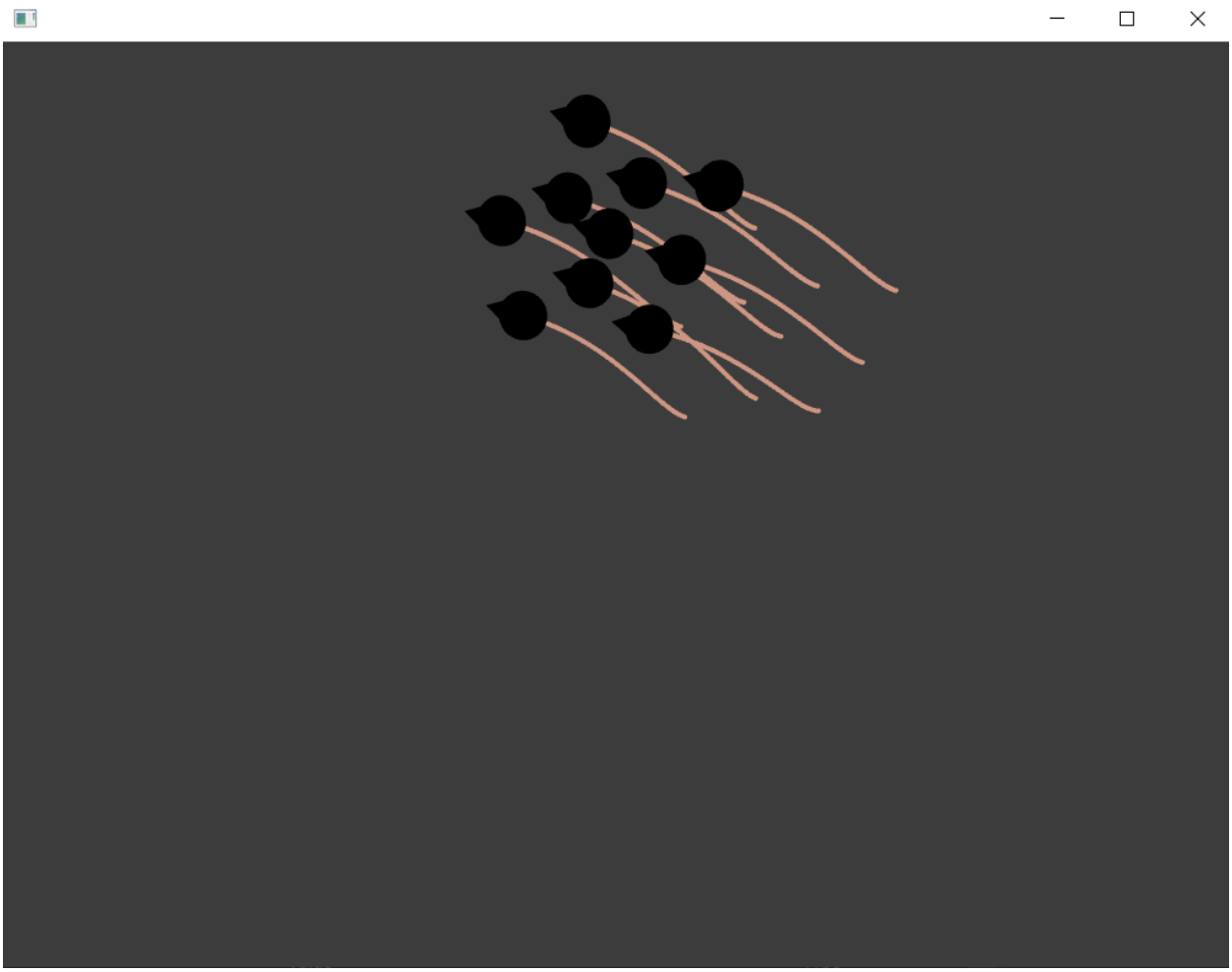

Fig 9. Flock in normal Behavior

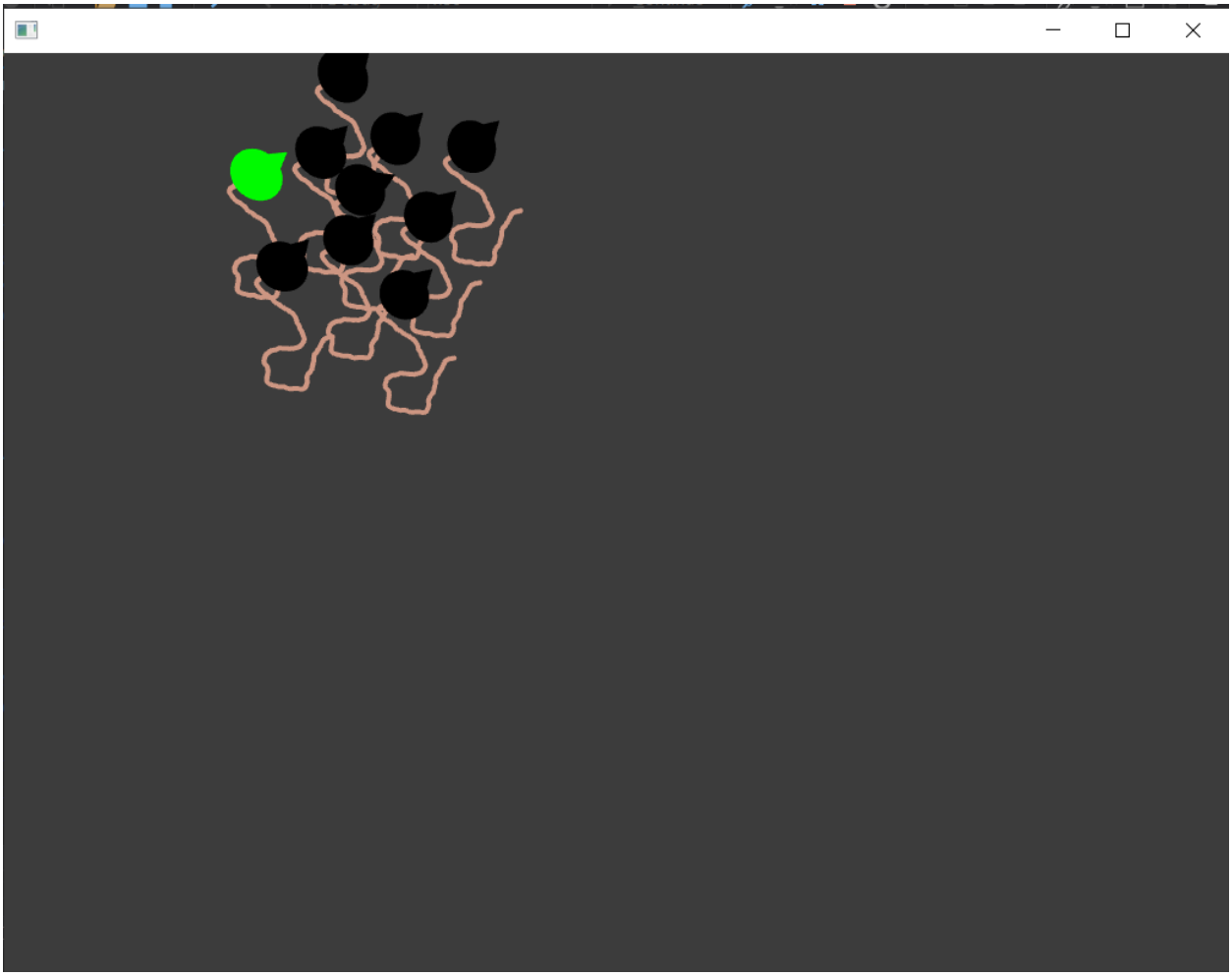

Fig 10. Flock Following the leader's behavior