AI - Movement Algorithms

In this writeup, I will explain the implementation of different movement algorithms and my findings when I executed them with different values for the parameters that affect the algorithm. I've used the OpenFrameworks for C++ with Visual Studio 2019 to implement all my algorithms. OpenFrameworks is a very simple yet powerful library that provides support for multiple systems such as Graphics, GUI, Sound etc.

Movement algorithms can be classified into two – kinematic algorithms and dynamic algorithms. Before we jump into the details, I want to discuss the data structure that holds the information about the kinematic state of any 2D object. Below you can see a simple struct named sKinematic which holds the data for position, orientation, velocity and rotation. Position specifies the current location of the object, Orientation gives us the angle it is rotated from the horizontal axis following either the left hand system or the right hand system, velocity is the rate of change of position and finally rotation is rate of change of orientation. There are some other functions to calculate and/or update the kinematics data.

```
struct sKinematicData {
	Math::sVector2D position;
	float orientation= 0.0f;

	Math::sVector2D velocity;
	float rotation = 0.0f;
	float wanderOrientation = 0.0f;

	void UpdateKinematic(Movement::sSteeringKinematic i_steering, const
float i_dt);

	void UpdateDynamic(Movement::sSteeringDynamic i_steering, const float i_dt,
const float i_maxSpeed, const float i_maxRotation);
```

```
        void OrientToDirection();

        float GetOrientationFromDirection(Math::sVector2D i_direction);

        Math::sVector2D GetUnitVectorAlongOrientation(const float i_orientation);

        bool operator ==(const sKinematicData i_other) const;
};
```

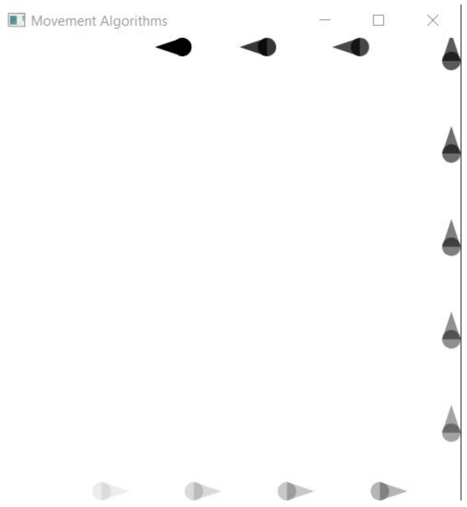
The main difference between kinematic and dynamic algorithms are, in kinematic algorithms, we override the current velocity and rotation with the new velocity and rotation whereas in dynamic algorithms, we add them to the new velocity and rotation.

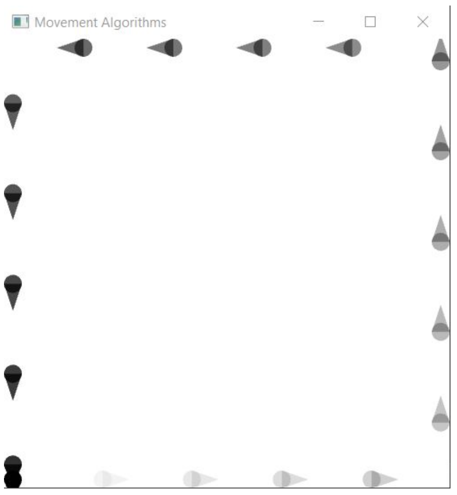
The way all the movement algorithms work is first they get the required steering needed for the behavior we're seeking and then update the object with the steering returned. Where steering is a struct that holds the data for linear and angular acceleration or velocity based on the type of algorithms we are using. The below depicts the pseudo code for this.

```
Steering = GetSteeringForBehavior(Algorithm type);
UpdateObject(steering,i_timeSinceLastUpdate);
```

Algorithms:

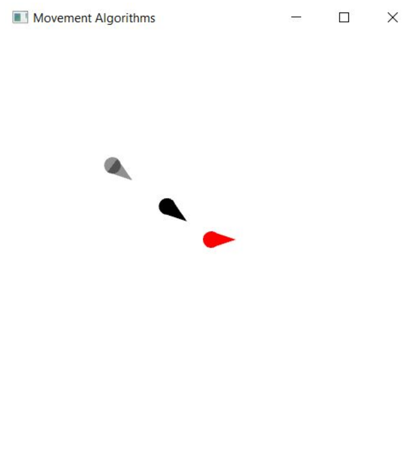
1. Basic Motion: I've started out with a simple movement of an object (boid) that moves around the edges of the screen. I've used the kinematic motion for this. On a higher level, it tries to reach the four edges of the screen one after the other. I've made some position checks to know the current edge it is trying to reach and provided with a velocity. The below screenshots provide an overview of one cycle movement.

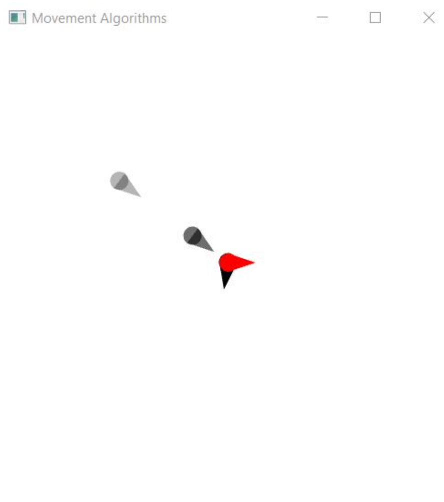


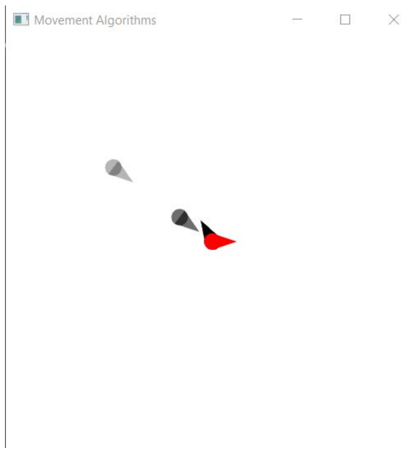


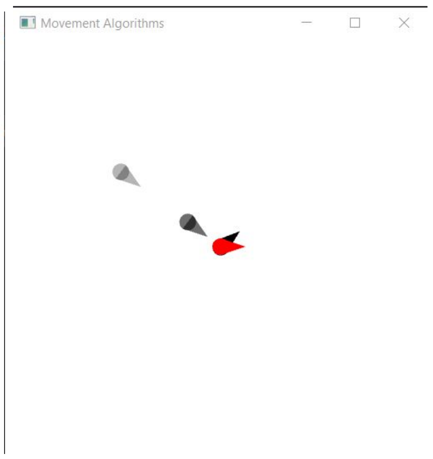
I’ve added the breadcrumbs to help understand the path it traversed. This is a very simple motion, you can adjust the speed of the boid to traverse the cycle faster.

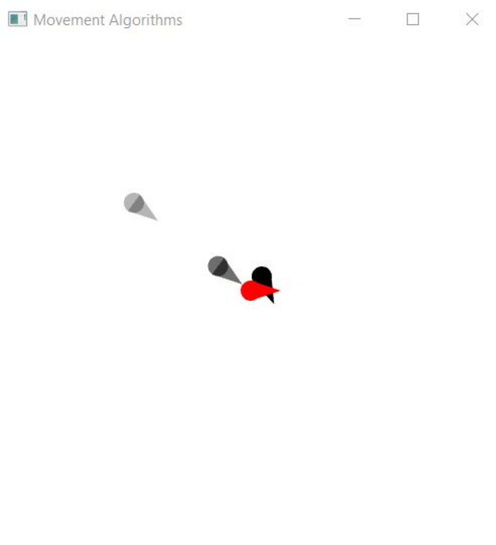
2. Seek - steering: The behavior of seek is to get to a position. In practical terms, the seek steering never arrives at the target it is seeking, it will generally overshoot and oscillate around the target it is seeking.
   For the seeking behavior I’ve implemented four algorithms, to see which one looks better and is actually arriving at the target.
   - Kinematic Seek: In kinematic seek, I calculate the direction to move based on the target’s position and then I multiply it with the max speed (magnitude), so that it always moves with the max speed in the direction of target.
     Since, we are not reducing the velocity at any point, the boid will overshoot its target and then try to reach back to it, which will result in an oscillating motion. The series of screenshots below show the kinematic seek behavior

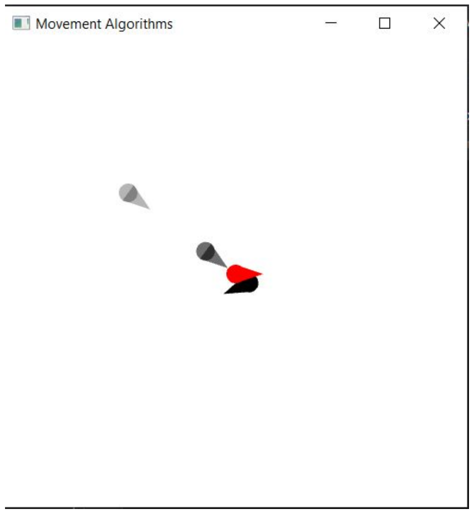

Movement Algorithms

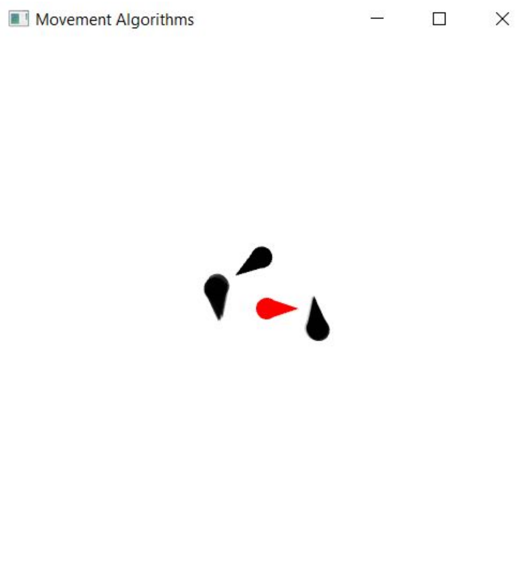

Movement Algorithms

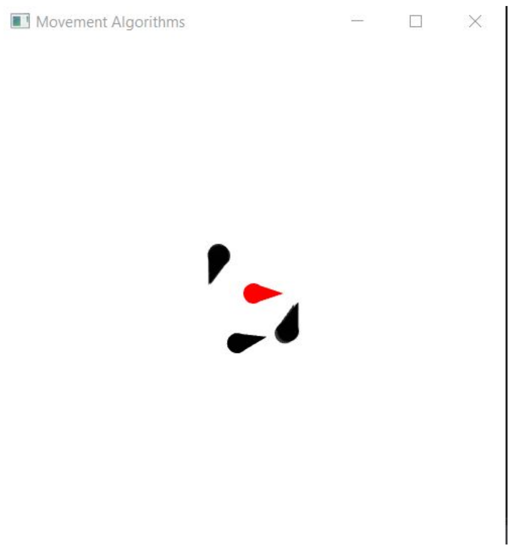

Movement Algorithms


Movement Algorithms





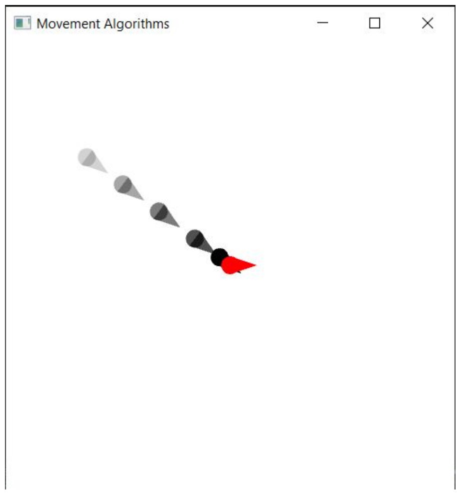


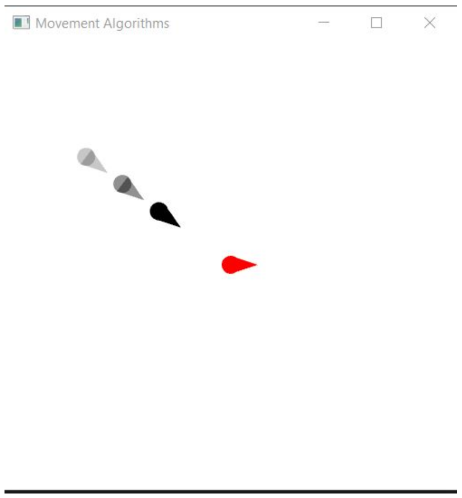


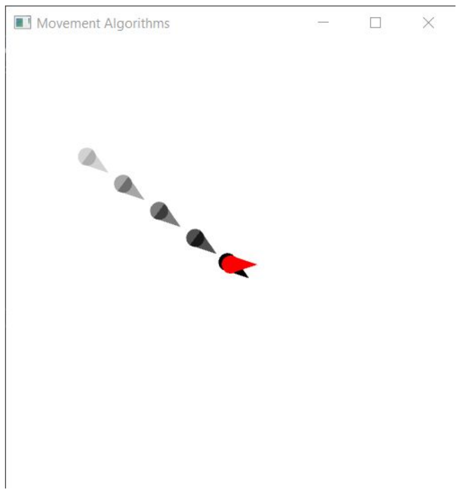
The last two screenshots show the behavior when the max speed is set very high. With very high max speeds, the radius of the oscillation behavior increased.

- Kinematic Arrive: The goal of the arrive is to stop at the target. For this we take additional input of targetRadius, which is basically the threshold from where we should start reducing the velocity so that it reaches to zero upon arriving at the target. In the implementation of the algorithm, we check if it is outside the targetRadius then we move with max speed, if it is inside the target radius, we scale down the velocity based on another input variable timeToTarget. This algorithm worked really well and stopped at the target.
  I've tried changing the max speed, target radius and the time to target, to see any interesting behavior but the boid always arrived at the target. The

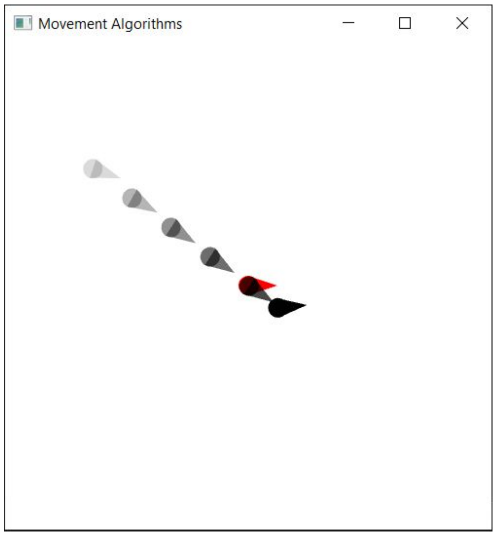
series of screenshots below show the behavior of kinematic arrive.In both seek and arrive, the orientation is matched to that of velocity.

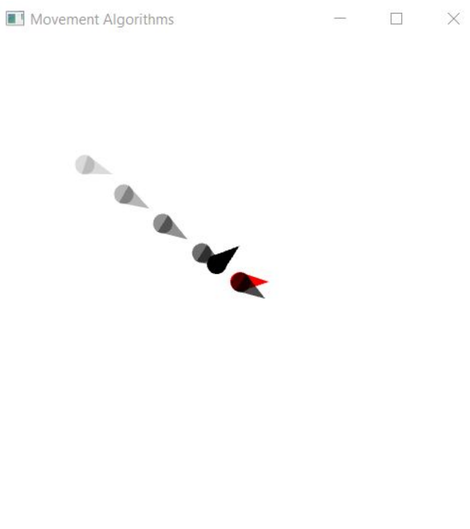


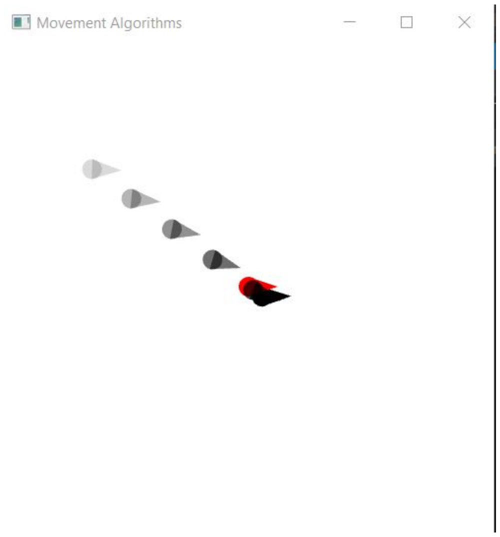




- Dynamic seek: I’ve also implemented the dynamic version of seek. In dynamic seek, we change the acceleration rather than the velocity. The same thing happened with dynamic as with kinematic seek. The boid overshoot and kept oscillating around the target. With increasing the max linear acceleration by keeping the max speed constant, I found that the overshoot distance went down. Below are the screenshots of Dynamic Seek.



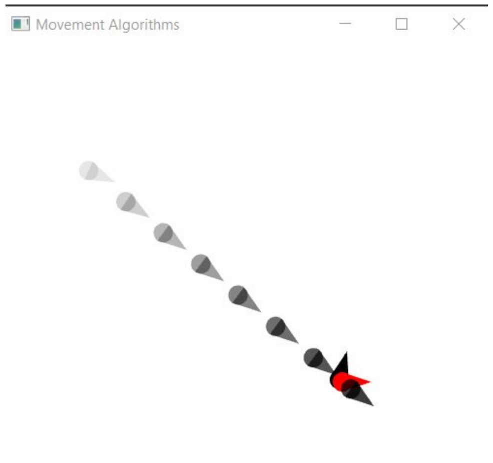


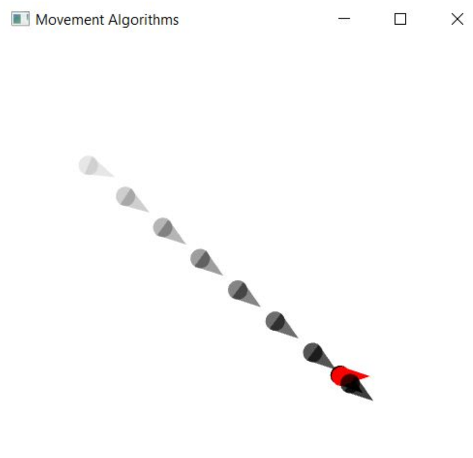


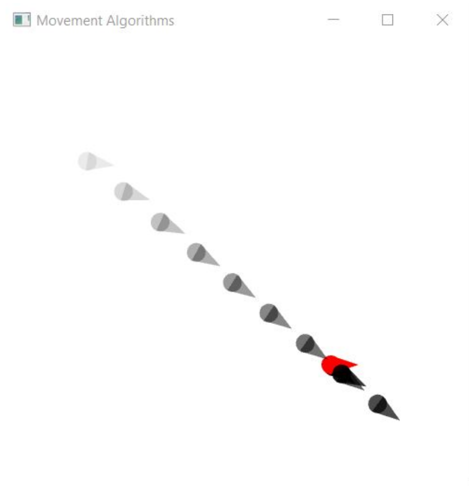
- Dynamic Arrive: In dynamic arrive, we specify two additional parameters: the targetRadius and the slowRadius. Target radius is the stopping radius allowed and slow radius helps to calculate the desired acceleration to slow down the boid to reach the target at zero velocity. Once inside the slow radius we calculate the targetVelocity by scaling it down and then we calculate the acceleration required to match the targetVelocity. If we're moving faster than the targetVelocity then the acceleration will be negative and the boid will start losing its velocity and finally comes to zero when it reaches the target radius.

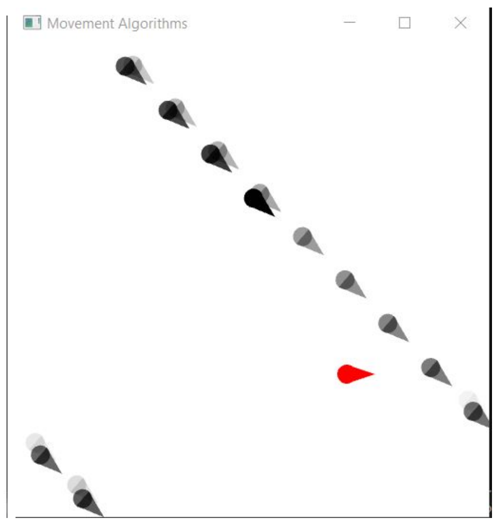
  There should be enough difference between the targetRadius and slowRadius for the algorithm to actually work because for minimal differences, the change in acceleration doesn't fully compensate for the high velocities.

With Dynamic arrive, if we increase the max linear acceleration and max speed, the boid overshoot but eventually arrived at the target. The overshoot distance increased proportionally with the max linear acceleration and max speed. Screenshots of the behavior are below





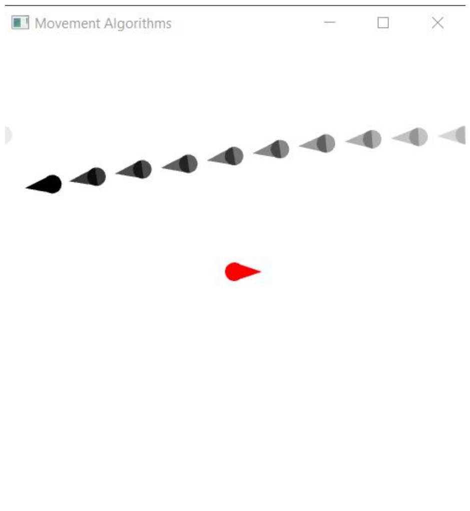


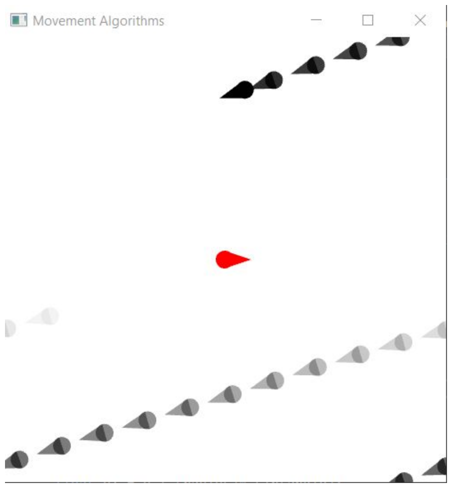


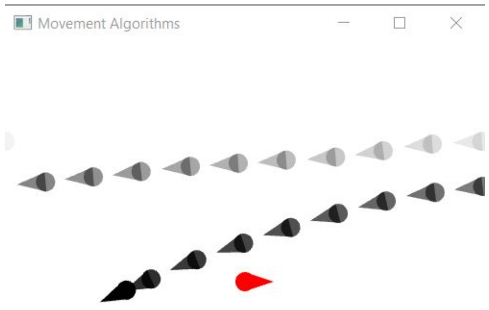
With velocities very very large, the boid overshoot the boundary of the window, (since the world I implemented is toroidal, it was never able to reach the target). The last screenshot shows this behavior.

3. Wander- steering: I've implemented both kinematic and dynamic wander behaviors.
    - Kinematic wander: In kinematic wander we specify a max rotation allowed, which changes the orientation of the boid and will move in that direction. With low maxRotation, the change in orientation was very small, and the

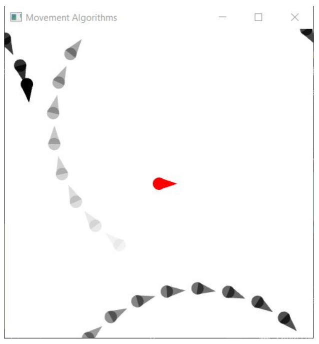
boid looked like always moving in a straight line. When the maxRotation is very large, it constantly keeps altering the orientation and didn't look good. Here are the screenshots showing this behavior

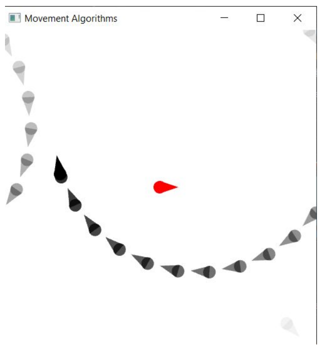


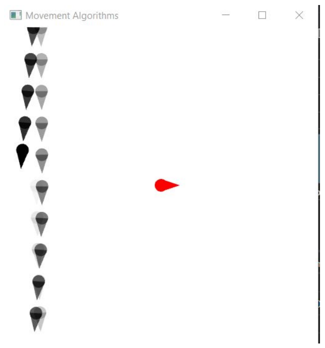




- Dynamic Wander: This algorithm takes some additional input, wanderOffset, wanderRadius and wanderRate. The way it works is, it looks ahead (wanderOffset) for a circle of radius (wanderRadius) and tries to reach a point on the circumference of the circle. With the wanderOffset, wanderRadius high, we can achieve a curved wander path. But if those are really small, it will end up moving in circles. Here are the screenshots of this behavior







4. Flocking- behavior: Flocking is a blended algorithm which consists of many boids trying to follow the leader,which is also a boid. Flocking has 4 algorithms in it.
    - Seek
    - Separation
    - Velocity match
    - Wander (only for leader)

    Each boid is associated with a mass. The flock leader will wander in the world, which shifts the center of mass of the flock constantly, the other boids will then seek the center of mass, by separating themselves from other boids, matching the velocity of the center of mass.

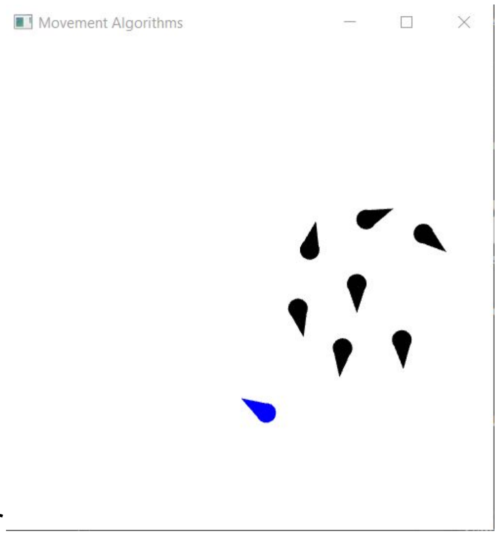
    Each of the algorithms is given a weight to it, so that the flocking - behavior looks intelligent rather than erratic behaviors.

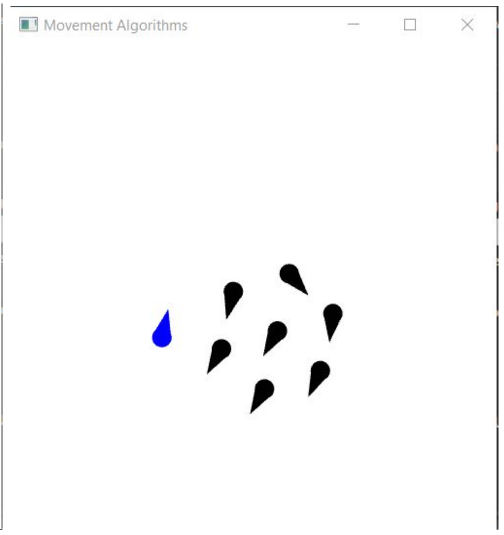
    I've tried out different weights, but no matter what the weights, there is always a point in time, where all the boids are parading with the leader(all move in constant velocity and direction). This behavior sometimes lasted only a few seconds and sometimes it lasted more than a few. Because the world is toroidal, when the leader of the flock went out of bounds and reached the other end, the flock would sometimes change its direction altogether to the back.

    The parade behavior is seen mostly when the max speed of the leader who is wandering is set almost similar to that of other boids. I tried changing the max speed for each of the boids, this resulted in some good looking behavior, there is not much parading,except for a few seconds.

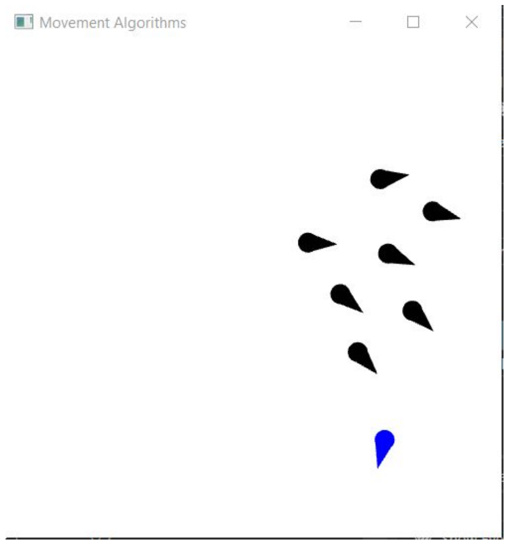
In the beginning, I didn't add separation for the leader, this resulted in the boids clashing with the leader. But when I later added it, this resulted in a much better behavior, because now the leader seemed to wander more, the spread of the leader is more, and the rest of the boids matched it.

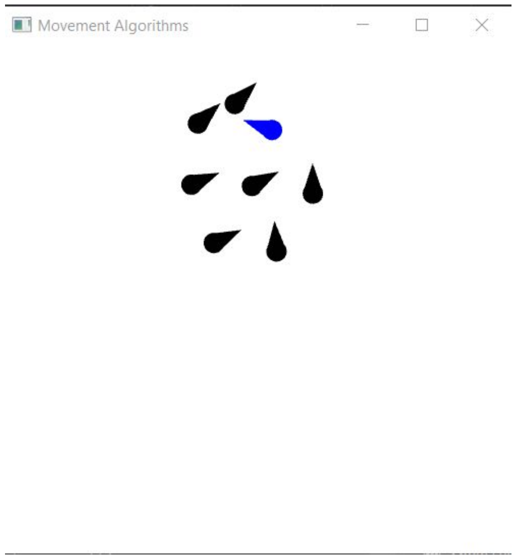
Screenshots of the flocking behavior

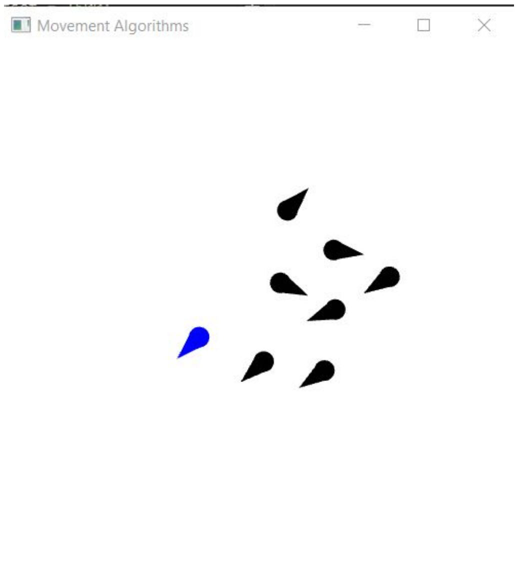


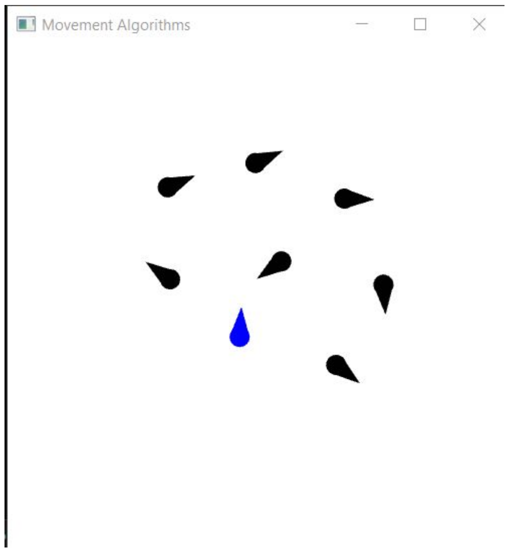


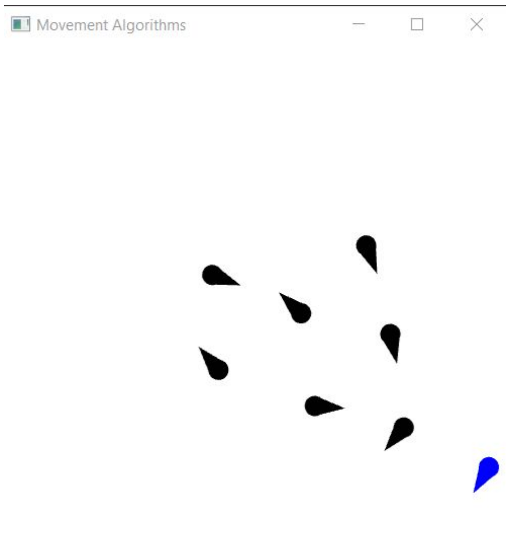
r

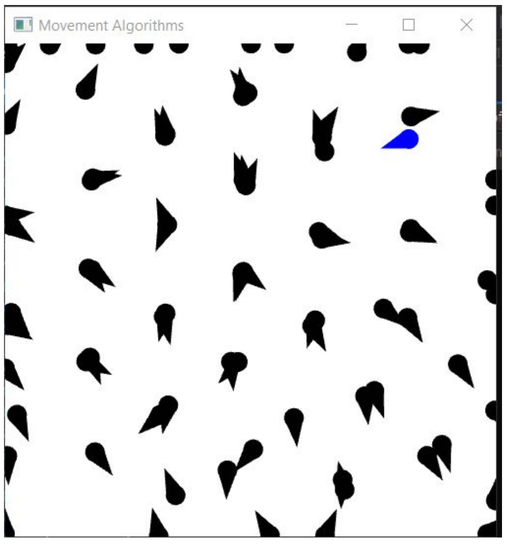


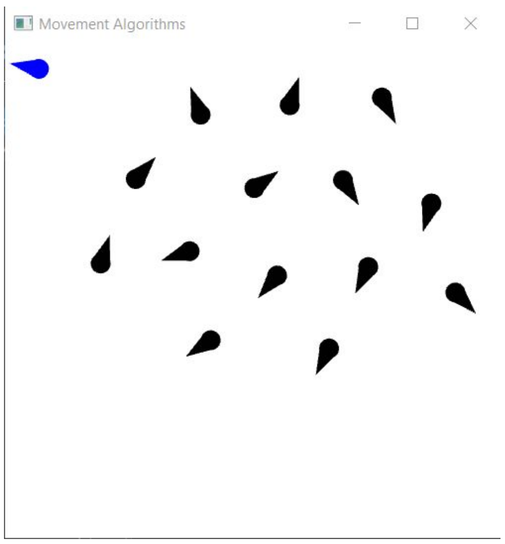


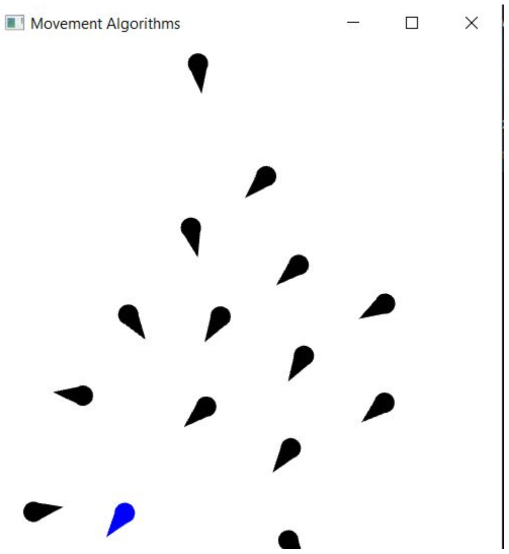

Movement Algorithms


Movement Algorithms


Movement Algorithms


Movement Algorithms


Movement Algorithms


Movement Algorithms

I’ve also tried increasing the number of boids in the flock to 100, they took the entire space and moved away from each other. The optimal number of boids was between 8-12.