CSC 584 Building Game AI
Homework 2 Report
Unity ID - jgajbha

Part 1:
For this part of the assignment, the first order of business was to implement a Kinematic variables class. I have instead created a class which consists of the 4 kinematic variables, as well as the steering variables and named it SteeringData. This class consists of two constructors, one of which acts as the default constructor and sets all the variables to 0. The second constructor initializes the values to whatever you want to set it to. I understand that this implementation of creating a steering variables class may take up more memory than required, and I haven't made use all the variables in the class for this assignment, but I have done it this way with future interests in mind.
After that, I created the pure virtual class, SteeringBehavior as instructed. This class has a function, other than the virtual function and it's destructor, called distanceBetween, which takes in two vector positions as input and returns a float, which is the distance Euclidean between those two vectors. Throughout this assignment, you will see it used sparsely but this has been created with future tasks in mind as well. Other than these functions, there are three variables that have been declared but may have been overridden throughout the assignment.

Velocity Matching:
For this part of the assignment, I had two implementations, the first one of which behaved like an arrive and align steering behavior, as I had implemented the align function and the goal velocity was set such that the boid wanted to always come to the goal position, i.e., target position was set as the mouse's position and velocity was calculated as target.position – character.position. I will include this implementation in my submission folder as "Part1-omitted.cpp".
In the second implementation however, I fixed up the code and first calculated the goal velocity using the mouse's current position and it's previous position and implementing current (position – previous position)/time.
The time variable, named as "dt" is being calculated in seconds throughout this assignment, and I have used the Clock class from SFML.
After calculating the goal velocity, I pass this goal velocity to the calculateSteering function in the VelocityMatchingBehavior class which outputs acceleration, and using this acceleration, I update the character's velocity and by extension, position, and then draw it on screen. The acceleration produced by this function is clamped down to a certain "maxAcceleration" value, for which I found, after much trial and error, the value *100.0* looked the best and was most pleasing to the eye. If you were to change this value to a higher constant, you would observe that velocity

matching works better but in some cases, the character may go faster than your mouse velocity. If you were to set it to a lower value, you'll see that the character may struggle to keep up with the mouse.
For this part of the assignment, I have not implemented an align behavior and not handled boundary violations and therefore, the boid may go off screen. Although, you can easily get it back on screen by moving your mouse in the opposite direction of the motion which made it go offscreen.
Apart from the kinematic aspect of things, I have implemented a CustomSprite class whose constructor sets the boids position, texture, scale and orientation to a certain default value, for the first part of this assignment, it only sets the texture and scale of the sprite.

Part 2:
Initially, you may see the boid move up to the top left corner to the screen without any change in orientation as the default constructor of the SteeringData class sets the goal variable values to (0,0) and 0 degrees respectively.
For this part of the assignment, I have used the SFML Mouse class and the isButtonPressed function to determine goal position. Once the goal position has been determined, it is assigned to the position variable in the goal object of the SteeringData class. This object, along with the character SteeringData object is then passed to the calculateSteering function of the PositionMatchingBehavior class. This function returns a linear acceleration which is then used to determine position of the boid.

<ins>Arrive:</ins>
For this algorithm, I experimented with many different values for maxAcceleration, radiusOfDeceleration and radiusOfSatisfaction. The best combination of these variables was 100, 200 and 5 respectively (all float values). These values have been determined after a lot of trial and error. Initially, I went with a higher value for maxAcceleration and a lower value for radiusOfDeceleration, which made the boid oscillate about the goal position. I then realized that my best bet would be to set a maxSpeed variable(100.0 in this case), and decrease the desired speed of the character according to distance between the character's position and the goal position. In essence, desired speed = maxSpeed * (distance_to_goal / radiusOfDeceleration), this simple formula, also known as linear interpolation, helps to decrease the character's velocity according to the distance to goal and as the distance to goal becomes lower than the radiusOfDeceleration, desired speed eventually tends to 0, which is how the character slows down within the radiusOfDeceleration. When the character enters the radiusOfSatisfaction, desired speed is set to 0, which helps the character stop when it reaches the goal position. In theory, this implementation of using linear interpolation to set desired speed should accommodate any maxSpeed value without any problems, but the best value for maxSpeed in this case is 100. I believe that this is the case because of the structure of the function itself. For the sake of

maintaining uniformity, this function returns a variable acceleration which is calculated as follows:

```
desired_velocity = desired_velocity / std::sqrt(desired_velocity.x *
desired_velocity.x + desired_velocity.y * desired_velocity.y) * desired_speed;
sf::Vector2f acceleration = desired_velocity - character.velocity;
```

Acceleration is clamped down to certain value maxAcceleration here, which is set as 100.0

If the character velocity were to be updated directly without using function's return value, acceleration(which is possible as the functions are being called by reference), I believe that the character will be able to move with any velocity and come to a perfect stop on the goal position without any oscillatory motion. I have not implemented the arrive algorithm in this way as it looks unnatural and coming to a standstill suddenly, exactly on the goal position is not how natural motion would occur with regards to a moving object.

<u>Align:</u>

The first and arguably, the most important thing to find for the align algorithm to work would be the goal orientation. I have achieved this using the mouse position. More specifically, I calculated the orientation by finding out the tan inverse of the difference between y and x components of the goal and character positions. I used the following approach:

```
            float dx = goal.position.x - character.position.x;
            float dy = goal.position.y - character.position.y;
            goal.orientation = atan2(dy, dx);
```

The character's current orientation was calculated using the getRotation function of the SFML sprite class:

```
character.orientation = sprite.sprite.getRotation() * (M_PI / 180.f);
//converting degrees to radians
```

After calculating the two variables, the character and goal objects were passed to the calculateSteering function of the OrientationMatchingBehavior class.

To ensure that the character's movement looks natural, I implemented the following code, so that the character doesn't turn more than π radians at a time:

```
        // Calculate the difference in orientation between character and goal
```

```
float rotation = (goal.orientation - character.orientation);

// Ensure rotation is within [-π, π] range
while (rotation > M_PI)
{
   rotation -= 2 * M_PI;
}
while (rotation < -M_PI)
{
   rotation += 2 * M_PI;
}
```

After calculating the rotation, the angular acceleration is calculated as follows:
float angularAcceleration = rotation / timeToTarget;

timeToTarget is the main variable here that decides the value of angularAcceleration, i.e., angularAcceleration is inversely proportional to timeToTarget.
timeToTarget has been set to 1.0 in this case, as if it were to be any higher, the motion would look unnatural, i.e., the sprite might keep rotating even after reaching the goal position, and if it were to be any lower, the sprite would rotate too quick and possibly overshoot from the desired goal orientation. Angular acceleration has been clamped down to a certain maxAngularAcceleration, 100.0, in this case, and if it were to be any higher or lower it would look as though the sprite is overshooting it's goal orientation or rotating too slow respectively.

Since the calculateSteering function returns a 2D vector and in this case, a vector isn't needed, I have returned a 2D vector object with the x component set as 0 and the angular acceleration being set to the y component. This output is used to calculate the character orientation.

Screenshots have been attached in the appendix.

Part 3:
As opposed to the conventional wander algorithm, the wander algorithm I have implemented, does not use a wander jitter, wander radius, or wander angle. Instead, I have two different approaches. Both of which include setting a random goal position and relegating the movement to an arrive and align steering behavior. The two approaches have been listed as follows:

i. <u>Distance based random goal position selection:</u>

For this approach, I have taken a certain POSITION_MATCH_THRESHOLD , 150.0 in this case. The purpose of this variable is to change the goal position whenever the character distance from the goal is less than 150.0
150 has been chosen because the radiusOfDeceleration(discussed in the previous section) is 100.0 in the PositionMatchingBehavior class. This means, that the character never decelerates, essentially making it a seek behavior. The following if statement will make things more clear:

```
        if (goal.position == sf::Vector2f() ||
getDistance(character.position, goal.position) <
POSITION_MATCH_THRESHOLD)

        {
            goal.position = getRandomPosition(0.f, window.getSize().x,
0.f, window.getSize().y);
            float dx = goal.position.x - character.position.x;
            float dy = goal.position.y - character.position.y;
            goal.orientation = atan2(dy, dx);
        }
```

The getRandomPosition takes in 4 parameters as input, which are as follows:
float *minX*, float *maxX*, float *minY*, float *maxY,* and outputs a random position vector.

As it is evident from the above code, 0.f, window.getSize().x, 0.f, window.getSize().y have been passed in as inputs to this function which ensures that the random goal position is never set outside the window. This takes care of the boundary violation case, because, if the goal is never set outside the window, the character can never seek any position outside the window, and therefore, never goes outside the window.

This approach yields a smooth, non-jittery, seek behavior with a random path.
Screenshots have been attached in the appendix.

ii. Distance and Time based random goal selection:

This approach is, in essence, similar to the first approach, the only things that have changed are the POSITION_MATCH_THRESHOLD, which has been set to 20.0 in this case and the if statement, which is as follows:

```
if (goal.position == sf::Vector2f() || getDistance(character.position,
goal.position) < POSITION_MATCH_THRESHOLD || dt.asSeconds() >
0.0040)
```

As it is evident from the above code snippet, dt is a variable of type Clock from the SFML class, and a new goal position is selected if dt is greater than 0.0040 seconds.

This approach yields a jittery wander behavior, which may look like the character is confused about where to go, in some cases. POSITION_MATCH_THRESHOLD has been lowered in this case as dt is very small and in most cases, if POSITION_MATCH_THRESHOLD were still to be 150.0, this approach would yield very little to no movement, along with a lot of jitters.
Screenshots have been attached in the appendix.

Part 4:
For this part of the assignment, I have two implementations. One of them, with the breadcrumbs and 30 boids, and one without the breadcrumbs and 60 boids (why the number of boids and breadcrumbs matters will be stated in the text below). Instructions on how to run both will be provided in the readme file attached with the submission.

For these implementations, I have created three classes, Alignment, Cohesion and Separation, each with their own functions alignForce, cohesionForce and separationForce respectively. I could've used the calculateSteering function extending from the SteeringBehavior class to calculate these but for ease of nomenclature, and also because I needed to pass an array to these functions in order to make flocking work, I have gone with the above classes. Please note that I have overridden the calculateSteering function from the pure virtual class SteeringBehavior so that function calls are possible.

These three functions essentially do the same task of taking in a boid's steeringData along with the steeringData of the rest of the boids and outputs steering force.

Align : Outputs steering force for a boid based on its neighbors' average velocity, such that the boid can align it's own velocity with it's neighbor's average velocity, given a certain perception radius.

Cohesion : Outputs steering force for a boid based on it's neighbors' position, such that, if two boids are within a certain distance of one another, they will be attracted to each other.

Separation : Outputs steering force for a boid based on it's neighbors' position, such that, if two boids are within a certain distance of one another, they will be repelled from each other.

These three forces in harmony create a flocking behavior.

Now, to discuss about the two implementations, they are as follows:

i. <u>Flocking with 30 boids and breadcrumbs:</u>

   For this implementation, I have multiplied the Flocking forces as follows:
   *0.8 * Cohesion, 1.0*Alignment and 10.f*Separation*

   I have done it in such a way because I observed that when a small constant factor is multiplied with the separation force, the boids tend to clump up together. The cohesion force in this case overpowers the separation force, this is because the perception radius that I have used for the separation function is much lower than the perception radius used for cohesion and alignment forces. I have done this so that the boids can move in "tighter" flocks.

   I believe that there is interference from the breadcrumbs in this case as I will explain in the next part of the writeup.

ii. <u>Flocking with 60 boids and no breadcrumbs (original implementation):</u>

   For this implementation, I have used the following factors to multiply the Flocking forces:
   *1.0*Cohesion, 8.0*Alignment, 150.0*Separation*

   In this case, I believe that since there are no breadcrumbs, a separation force of 150.0 times is makes more sense, as the perception radius is very low (I haven't been able to figure out why breadcrumbs is interfering with the flocking algorithm as of now). However, when I add my implementation of breadcrumbs to this code, the boids do not move and oscillate in place due to the high separation force.

   The main agent in this implementation of flocking is the alignment force. In this case, I have chosen a very high factor for the separation force so that the boids don't clump up together.

In both the above mentioned implementations, I have noticed that clumping does take place no matter how high the separation place and this is because of the boundary violation conditions which make the boids wrap around to the other side of the screen whenever the boids cross the edge of the screen. Due to this, the boids may sometimes end up on top of each other, or just slightly touching each other. Other likely causes of this clumping up behavior may be due to the fact that some boids may spawn quite close to each other, and that I am giving the boids an “artificial boost”(mentioned ahead in the write-up), in some cases. I have created a function so that boids spawn at random positions at the beginning of the simulation and as a safety measure, I have also implemented another function which makes sure that boids don’t spawn on the same location.

All the boids get a random velocity at the beginning of the simulation, and a function has been created for that as well.

To make sure that the simulation never stays still, I have given it an “artificial boost” such that, if the magnitude of any boid’s velocity ever goes below 100.0, the magnitude gets set back to 100.0. To clamp down the velocity, I’ve chosen the value of maxSpeed to be 200.0. Both of these measures together ensure that the speed of the boids stays between 100.0 and 200.0.

Finally, to make sure that the boids are always pointed to where they are headed, I have implemented the following function, which gets the orientation from the velocity of a boid:

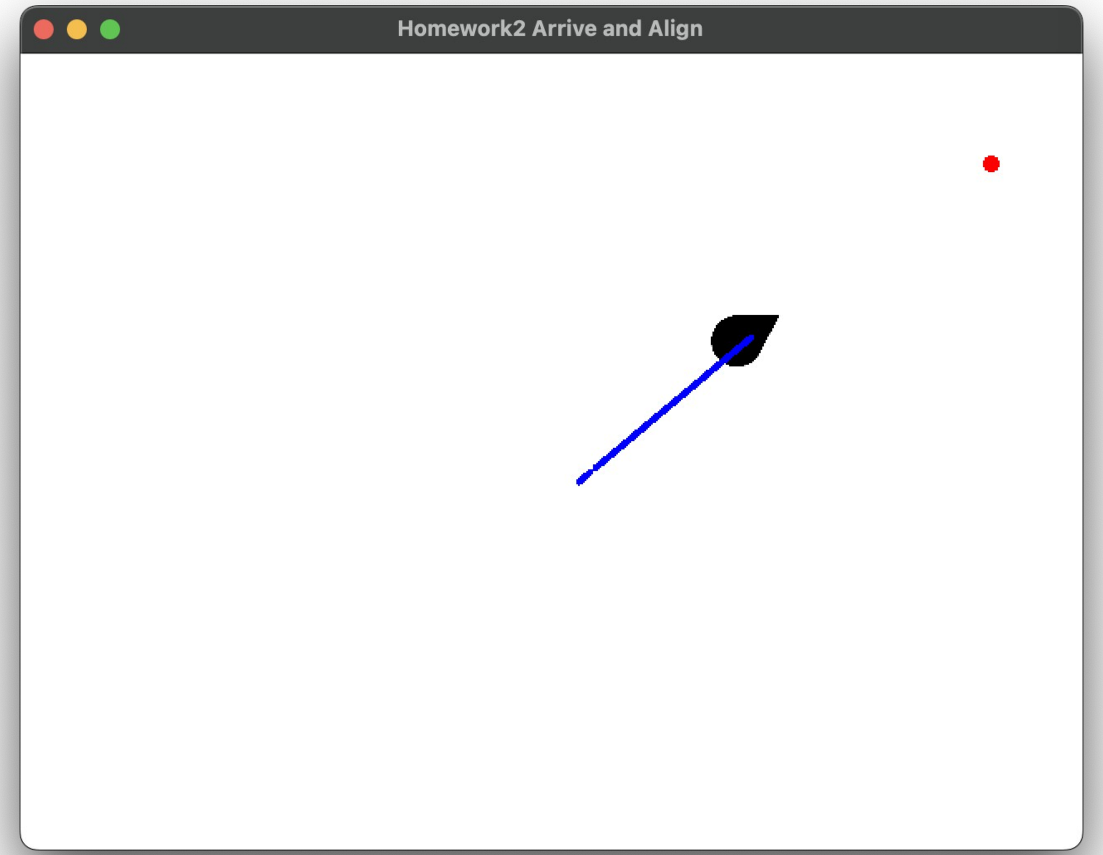
```
float getOrientationFromVelocity(const sf::Vector2f &velocity)
{
    return std::atan2(velocity.y, velocity.x) * 180.f / M_PI;
}
```

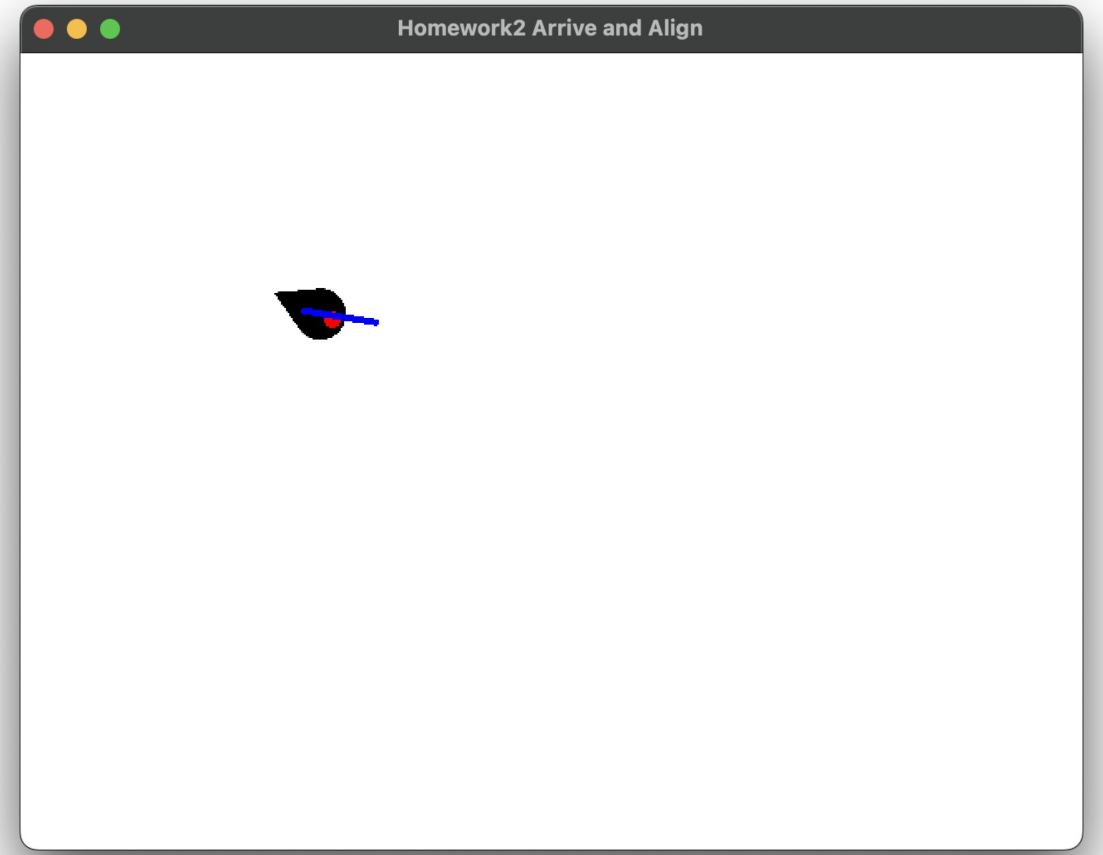
Screenshots of both the implementations have been attached in the appendix.

Appendix

Screenshots of all the implementations have been provided in this section along with required labels and context.

Part 2:

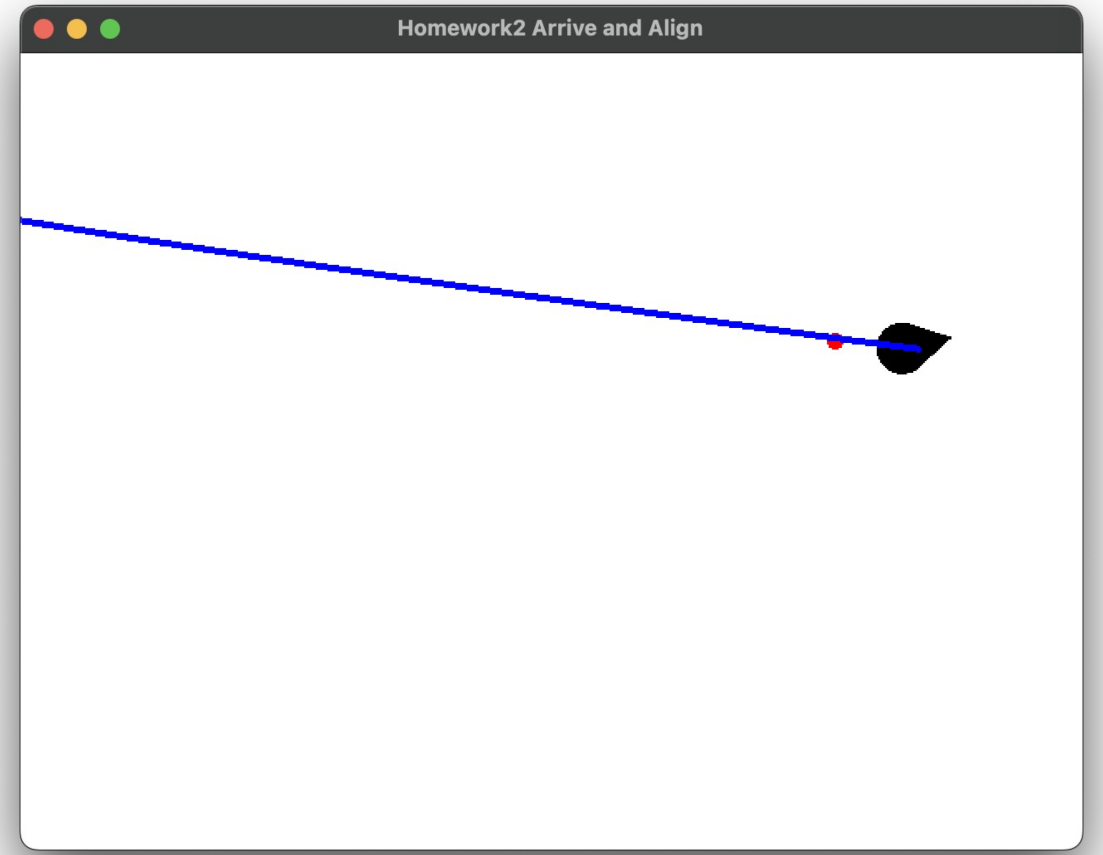




The above screenshots demonstrate the arrive and align algorithm with the following parameters taken into consideration:

```
    float radiusOfDeceleration = 200.f;
    float radiusOfSatisfaction = 10.f;
    float maxSpeed = 100.f;
    float maxAcceleration = 100.f;
    float maxAngularAcceleration = 100.f;
    float timeToTarget = 1.f;
```

The red dot denotes the location of the mouse pointer left click.

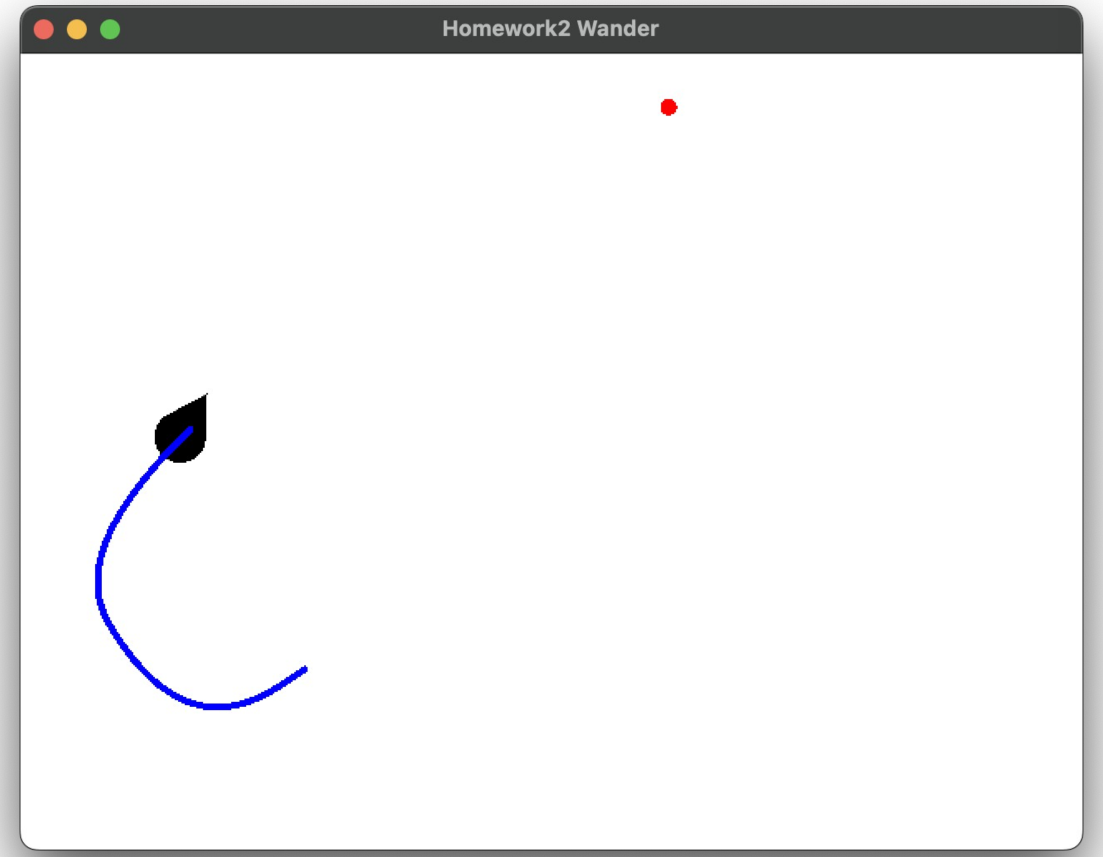


The above screenshot illustrates arrive and align behavior with the following parameters:

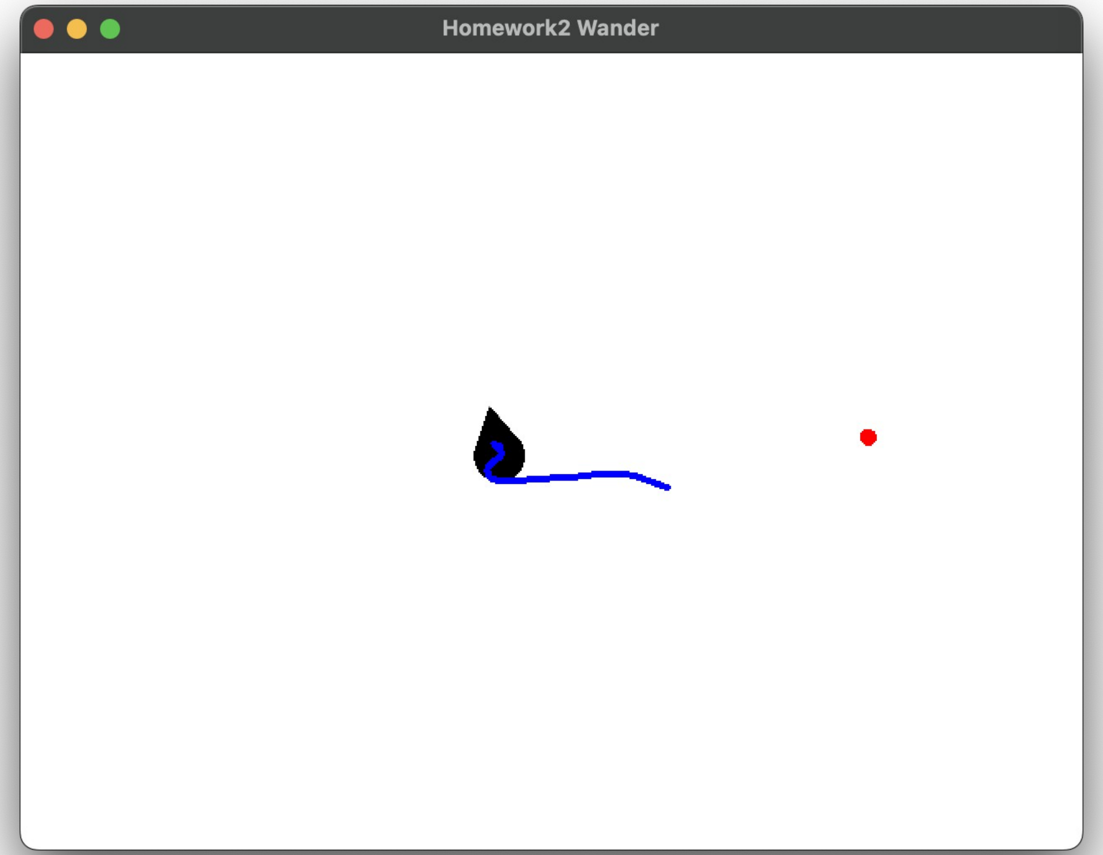
```
float radiusOfDeceleration = 200.f;
    float radiusOfSatisfaction = 10.f;
    float maxSpeed = 200.f;
    float maxAcceleration = 100.f;
    float maxAngularAcceleration = 100.f;
    float timeToTarget = 1.f;
```

As you can see, the boid overshoots it’s goal position in this case.

Part 3:

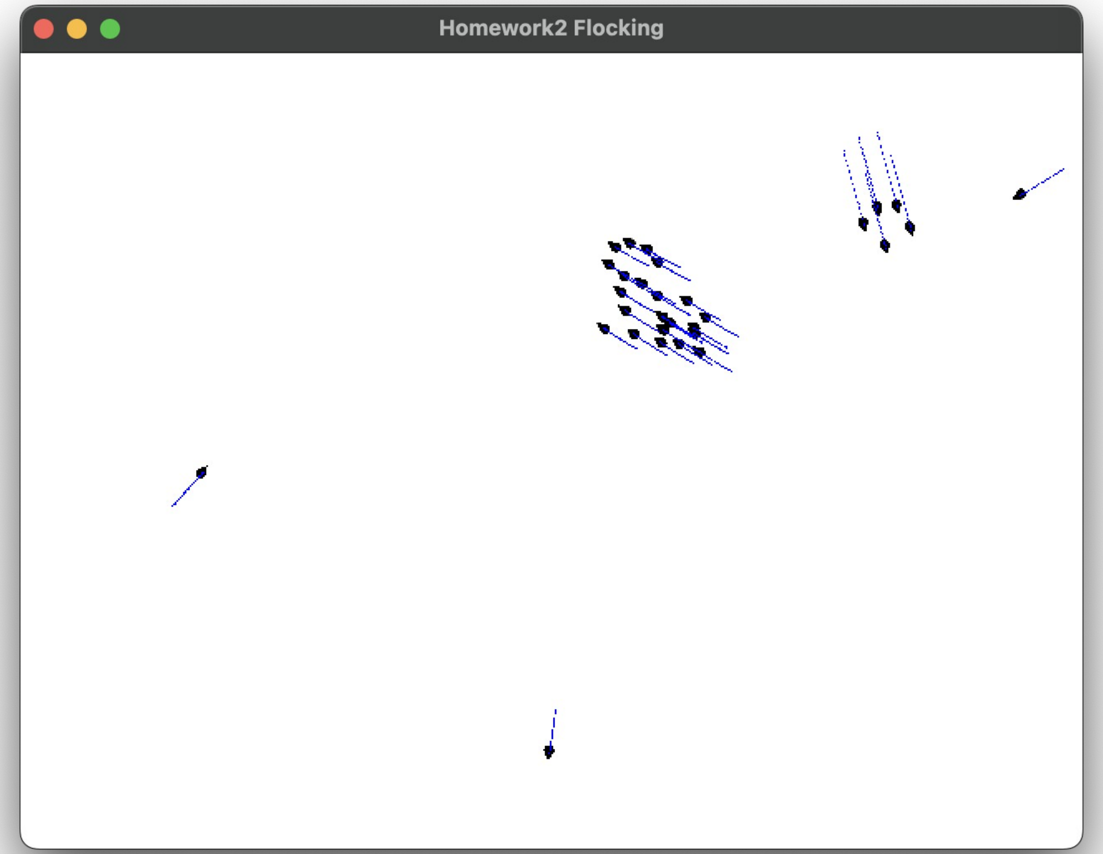


The above screenshot illustrates wander behavior using POSITION_MATCH_THRESHOLD = 150 and no time based goal position selection.

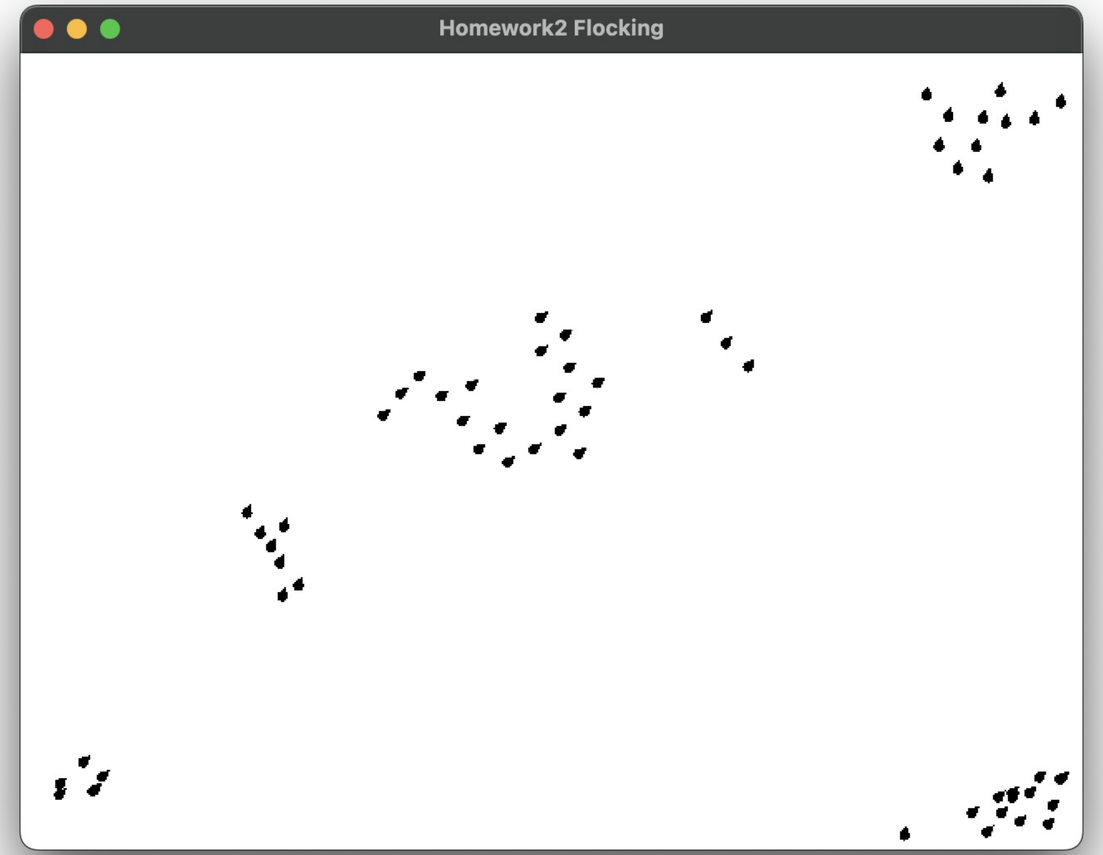


This screenshot illustrates wander steering behavior with POSITION_MATCH_THRESHOLD = 20 and random goal position selection based on certain time constraints. Evidently, the path is much more jittery as compared to the previous screenshot.

Part 4:



The above screenshot demonstrates flocking behavior with breadcrumbs on and the following parameters:
0.8 * cohesionForce + 1.0 * alignmentForce + 10 * separationForce
Number of boids = 30



The above screenshot demonstrates flocking behavior with breadcrumbs on and the following parameters:
1.0 * cohesionForce + 8.0 * alignmentForce + 150.0 * separationForce
Number of boids = 60