Homework II Report

Variable Matching Steering Behaviours

I used a variable called mouse_bound to get the first position of the mouse on the window and a flag called mouse_out that sets itself to true whenever the mouse enters the viewport and false when it leaves the viewport. The variables mouse_curr and mouse_final track the initial and final positions of the mouse pointer at each frame. We calculate the offset between the two mouse pointers and calculate the euclidean distance the sprite has to move.

Boundary conditions are handled as follows, if the sprite exits from one side of the screen, it reappears on the opposite side. This is applied to both the height and width of the viewport using the checkBorder function. The speed of the sprite is adjusted using the max_speed variable part of the velocity class. It takes the value of 3 for my PC. The drawback of the method I have used is that the velocity of the sprite depends on the framerate of the PC. If I had started my assignment earlier, I would have utilised the sf::Clock to make the motion frame independent. Whenever the sprite exceeds the max_speed value, we normalise the sprite velocity and multiply by max_speed to ensure the sprite's acceleration and velocity is capped.

Arrive and Align

I began designing the algorithm for arrival by modifying the SFML code to take input from mouse click rather than track mouse movement. I set the initial position of the sprite and final position in variables called initial and target as part of the sprite class, calling the move function until the target is reached. I calculated the velocity required when the mouse was clicked and each time the sprite moved. I reduced the goalSpeed[1] to 0 when the position of the sprite was within the radiusOfSatisfaction, set it to MAX_SPEED when the sprite exceeded the radiusOfDeceleration and set goalSpeed to MAX_SPEED * distance / radiusDeceleration when the sprite is within the radiusOfDeceleration.

I tried using vector algebra to curve the sprite when the mouse was clicked mid movement, more specifically the formula (x_vel * sin a + y_vel * sin a, where x_vel and y_vel are components of velocity in the x and y direction and a is the angle in which the sprite was rotated by) , but I was unable to do so as evidenced by the breadcrumbs in the screenshots provided in the appendix. The sprites zig-zagged their way through the screen at times instead of steering gracefully. I calculated rotation using the same logic, using the variables rotationSatisfation and rotationDeceleration in place of radiusOfSatisfaction and radiusOfDeceleration.

I had trouble when calculating the angle to rotate the sprite when trying to align the sprite. I initially used an integer to store the rotation the sprite was to do each frame, but I realised that since my code was frame dependent, the rotation movement looked very snappy. I had to use an integer to store the

[1] Formulae obtained from Dr. Robert's lectures to calculate velocity based on the sprite's current position

rotation angle since I had to use the mod operation (by 360). I changed the rotation variable to a float and converted it to an integer whenever I had to perform a modulus operation.

Doing this and the fact that I set max_speed to 2 resulted in a much smoother motion. When max_speed was set at 1, it resulted in very slow motion and anything above 4 resulted in the sprite moving too quickly. The drawback of programming movement frame by frame and not dependent on time is that different computers will observe different results. So when my code is being graded, the max_speed that works for me might not necessarily work on other PC's. I set maxRotation to 0.3, rotationSatisfaction to a very minute value (0.0001), timeToTargetRotation = 1.5 and rotationDeceleration to 0.2. Similarly for arrival, I set radiusDeceleration to 100, radiusSatisfaction to 2 and timeToTargetVelocity to 10.

I used a double ended queue to implement breadcrumbs. I used the crumb class provided in the bread_crumbs.cpp file. I had a maximum of 5 crumbs show up on screen at any point. A crumb was added to the double ended queue when the sprite travelled a euclidean distance of 50 pixels. If the queue already has maximum size, I popped the front of the queue and pushed the new crumb to the back of the queue thereby maintaining the max_size property.

Wander

Wander is an extension of the arrive and align algorithm and uses more or less the same code as used before. The only difference being I used a random function to generate the target my sprite is going to move to. The parameters that I used are the same from the arrive and align algorithm. The sprite accelerates until it reaches the radiusOf Deceleration where it accelerates to a halt. The result is a very snappy motion, akin to that of the snake in the Snake game.

If I had the chance to better improve the motion of the sprite, I would choose a new target mid movement and possibly curve my sprite. This would result in a much smoother movement, without many abrupt changes in velocity. I would again base my logic on using time dependent motion over frame dependent motion. Breadcrumbs are implemented using the same logic.

Flocking

My algorithm[2] is based on an internet article that explains flocking behaviour based on 3 coefficients namely Cohesion, Alignment and Separation, whose values can be adjusted in my code in the beginning of the definition of the Flock class (alignMul, cohMul, sepMul). I had to refactor my code into two classes namely Flock and Boid. The Flock class does most of the work using the update function that calculates all the average forces, moves and displays the Boids. I initialised 30 Boids in a vector, whose positions are randomised using a normal distribution depending on the dimensions of the screen and the velocity is initialised using an uniform distribution that ranges from -1 to 1.

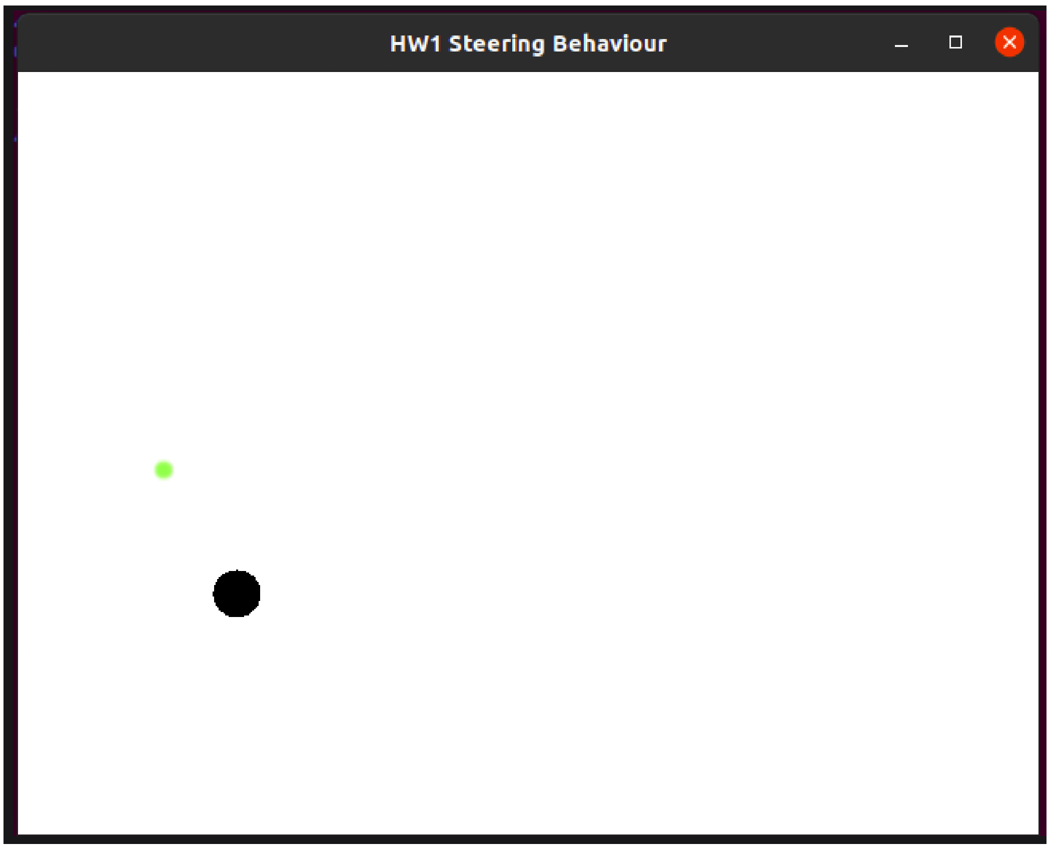
The update function is called on the Flock object that performs calculation of the average forces acting on the Flock. The averageForce is a result of the AlignmentForce, CohesionForce and

[2] https://www.oreilly.com/library/view/ai-for-game/0596005555/ch04.html

SeperationForce and each sprite is moved based on the sum of the three vectors. In order to perform the calculation, we set a perception radius for each of the three flocking parameters. We iterate through each Boid and calculate the resulting force that exists between the Boid in consideration and every other Boid. If the two Boids are equal, we skip computation (duplicates). The alignmentForce depends on the velocities of the other Boids, cohesionForce based on the position of the other Boids and separationForce based on the distance between the Boid in consideration and all the other Boids within their respective alignment, cohesion and separation perception radii. The total force is averaged out with the number of other neighbours within the perception radii and normalised if the force were to exceed the maxForce. Each of the individual forces are multiplied by their respective coefficients (alignMul, cohMul, sepMul).

If the Boids were to exit the boundary, the checkBorder function used before will reinitialise the position of the Boid exiting from the screen using the logic as mentioned before. Manipulating the perception radii and their respective coefficients results in differing behaviour, with a few resembling a flock of birds, sticking together and in cases repelling each other.

APPENDIX



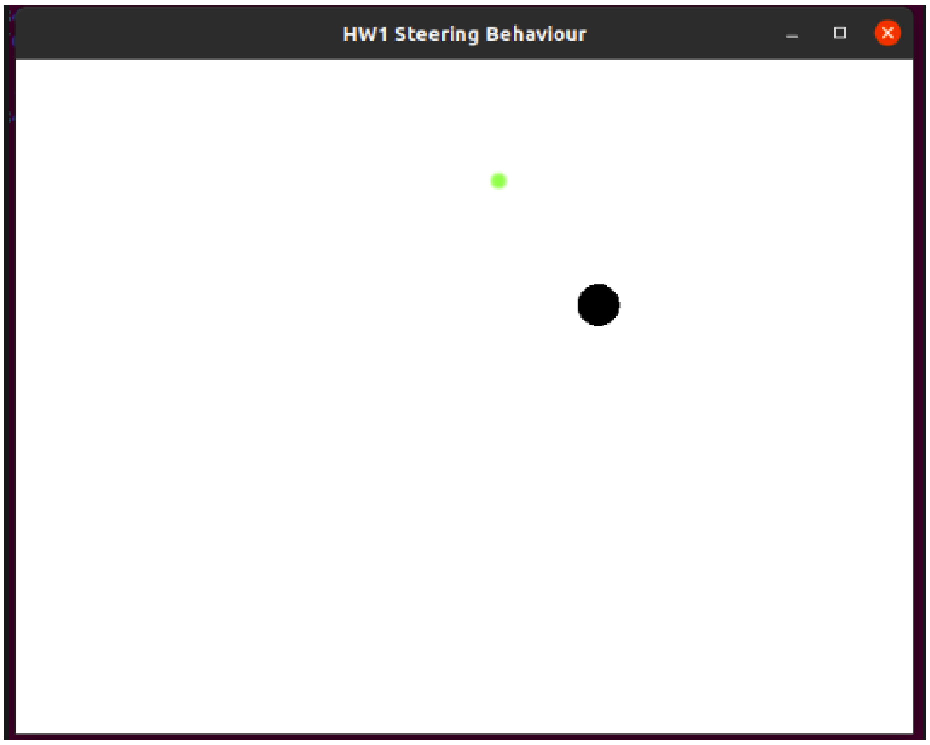
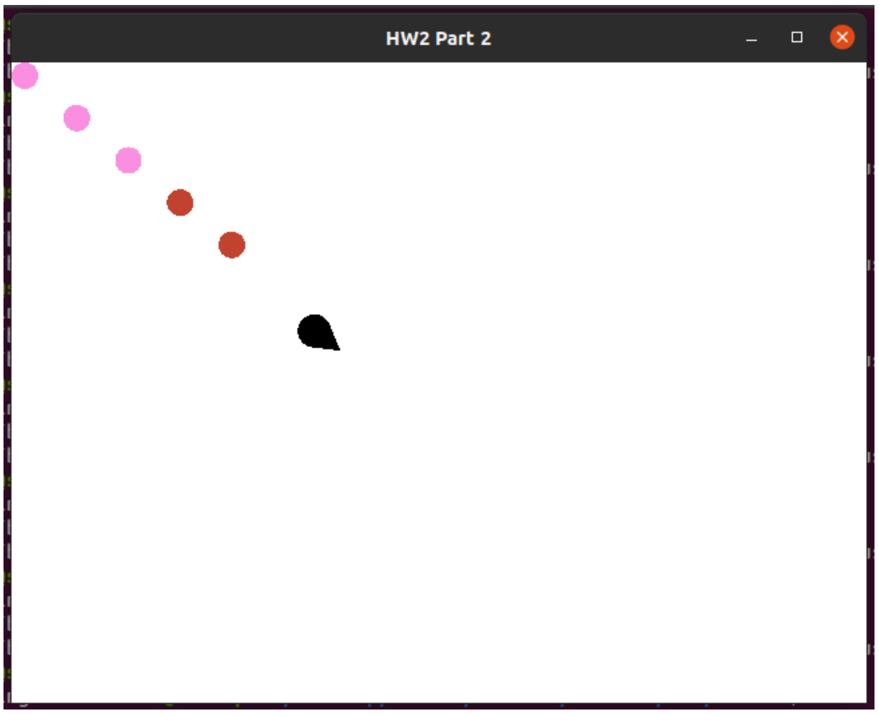


Fig 1 & 2: Steering behaviour performed, with the green cursor representing the mouse position.



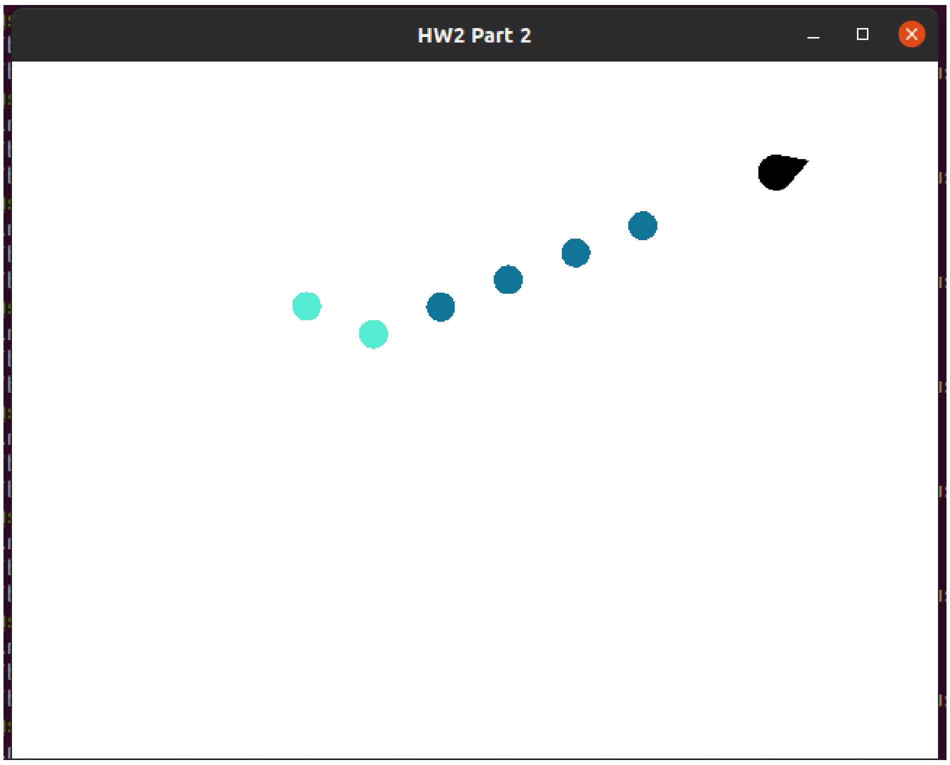


Fig 3 & 4: Arrive and align with breadcrumbs(changing colour for effect), sprite aligns itself in direction of motion

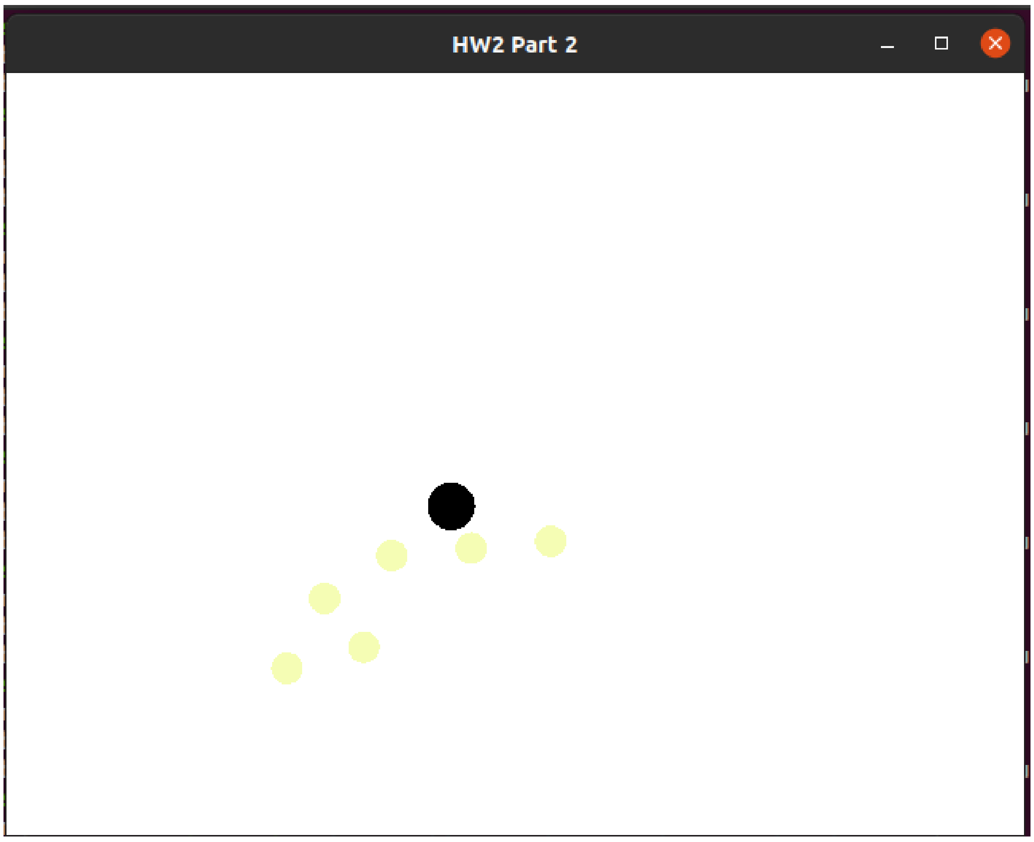


Fig 5: Wander behaviour with sprite exhibiting zig-zag motion

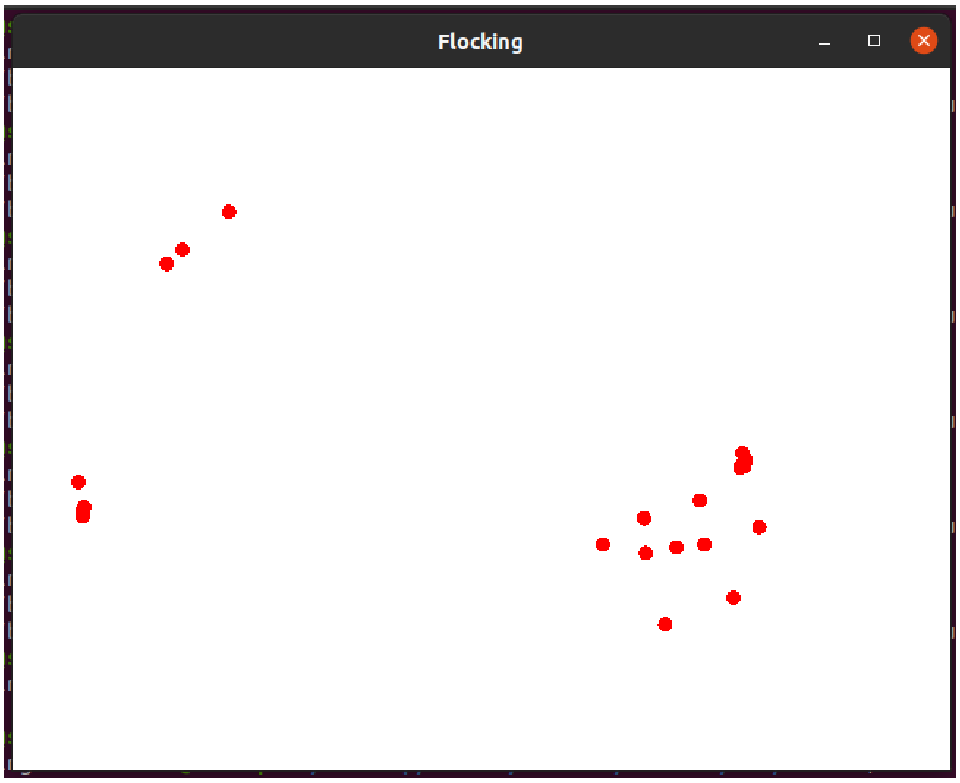


FIg 6: Flocking with alignMul - 1.4, cohMul = 1.1, sepMul = 1 and perception radii = 50

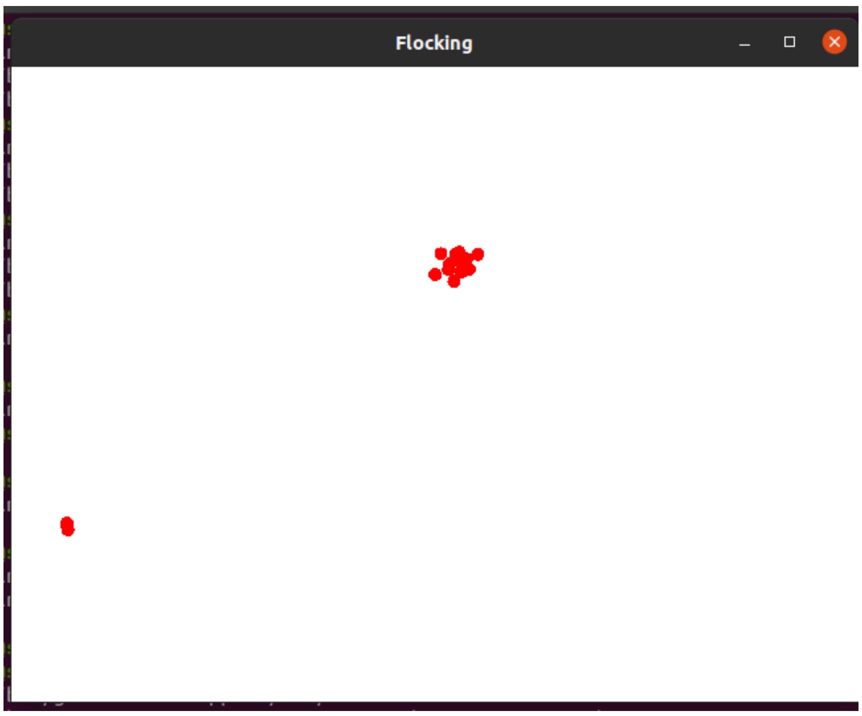


Fig 7: Flocking with alignMul - 2, cohMul = 1.1, sepMul = 1. Alignmentperceptionradius = 20, cohesionperceptionradius = 50 and separationradius = 10. Notice that the Boids stick closed when the separationperception radius is decreased.