

To organize my code for this assignment, I created a header file (`steering.h`) to store the library I built. I based my implementation on the breadcrumb example, but I decided to create a dedicated header file for movement and steering behaviors. This included Arrive and Align, kinematics, steering behavior subclasses for all four primary vectors (position, orientation, velocity, and rotation), and flocking behaviors (Alignment, Cohesion, and Separation).

This was my first time using virtual attributes in C++. While I'm familiar with abstract classes in Java, I wasn't as comfortable with them in C++. Looking back, I probably didn't take full advantage of them. However, I found creating a header file extremely useful and will continue using this approach in future projects.

At the top of the header file, I defined the `Kinematics` class, which started with a few essential variables but became increasingly bloated as the assignment progressed. Ultimately, it contained ten variables—the four main ones (position, orientation, velocity, and rotation) and six additional constraints, such as max speed, acceleration, angular acceleration, and steering radii.

For the steering behaviors, I created a pure virtual `SteeringBehavior` class with the function:

```
virtual sf::Vector2f calculateSteering(const Kinematic& character, const Kinematic& target) = 0;
```

as specified in the assignment. I then implemented subclasses for different kinematics. In all my `calculateSteering` functions, I simply used `target.position - character.position`. I likely could have implemented more refined calculations.

For most of the assignment, I relied on the move and draw functions from the breadcrumb example on Moodle, heavily modifying the `Boid` class while leaving the breadcrumb mechanics mostly intact. The `Boid` class was extensively reworked to incorporate and manage kinematics.

Arrive and Align

For Arrive and Align, I implemented two approaches: one from class and another based on a ChatGPT-generated suggestion. I created separate classes for Arrive and Align, which were similar to those discussed in class.

Arrive Behavior

The `Arrive` class calculates a steering force that adjusts the boid's velocity based on its distance from the target. The process follows these steps:

1. **Calculate Direction** – The boid determines its movement direction by computing the vector difference between the target's position and its current position.
2. **Check Arrival Radius** – If the boid is close enough to the target, it stops moving by setting velocity to zero.
3. **Adjust Speed Based on Distance** – If the boid is far from the target, it moves at max speed. If it enters the slow radius, its speed is proportionally reduced to allow for a smooth stop.
4. **Compute Steering Force** – The target velocity is determined based on the adjusted speed, and the steering force is calculated as the difference between the target velocity and the boid's current velocity.

HW2 Write ups

This ensures the boid approaches the target efficiently, slowing down naturally rather than stopping abruptly.

Align Behavior

The **Align** class is responsible for adjusting the boid's orientation smoothly so it faces the correct direction. The steps are:

1. **Compute Required Rotation** – The difference between the target's orientation and the boid's current orientation determines how much it needs to rotate. This value is normalized to stay within $[-\pi, \pi]$ to prevent excessive rotation.
2. **Check Alignment Radius** – If the difference is small enough, the boid stops adjusting its rotation.
3. **Adjust Rotation Speed** – If the boid is far from the target direction, it rotates at max speed. As it gets closer, it slows down to avoid overshooting.
4. **Apply Steering Force** – The force is applied smoothly, ensuring the boid turns naturally without abrupt movements.

Initially, the ChatGPT-generated Align class didn't work—the boid wasn't rotating at all. It was supposed to face its direction of travel, so I ended up using velocity vectors to determine the rotation angle instead.

I also implemented two mouse-controlled targeting functions:

- One tracks mouse movement and updates the target position in real-time.
- The other updates the target position when the left mouse button is clicked.

Wandering Algorithm:

While implementing Arrive and Align, I also added a feature to handle screen wrapping. If a boid moves beyond 1002 or -2 in either x or y direction, it wraps around to 0 or 1000, respectively. I also incorporated max speed and acceleration variables into the **Boid** class.

Although wandering is typically implemented as a circle with random targets in front of the boid, I took a different approach. I created two wandering functions:

1. One randomly selects a target anywhere on the screen.
2. The other selects an off-screen target ($\{500.0f, -500.0f\}$, $\{500.0f, 1500.0f\}$, $\{1500.0f, 500.0f\}$, $\{-500.0f, 1500.0f\}$). Because of the screen-wrapping logic, the boid never actually reaches these points; instead, it moves indefinitely until conditions trigger a target switch.

In both cases, the target changes after a set number of iterations.

Flocking Algorithm:

When I first implemented flocking, I asked ChatGPT for guidance. It suggested creating a separate file due to how different the flocking logic was from my other behaviors.

The flocking system combines **Separation, Cohesion, and Alignment**, allowing boids to move together while avoiding collisions.

HW2 Write ups

Separation ensures boids don't get too close to each other:

1. **Detect Nearby Boids** – The algorithm scans for nearby boids within a separation radius.
2. **Compute Repulsion Force** – If a neighbor is too close, the boid moves away. The closer a neighbor is, the stronger the repulsion force.
3. **Apply Steering Force** – The combined repulsion forces determine the final steering adjustment.

This prevents clustering and reduces collisions.

Cohesion:

Cohesion ensures boids remain close to the flock:

1. **Find Center of Mass** – The algorithm calculates the average position of all nearby boids within a cohesion radius.
2. **Steer Toward Center** – The boid applies a force pulling it toward this central position.
3. **Balance With Other Forces** – Movement remains subtle, preventing the boid from converging too aggressively.

This keeps the flock together while allowing for natural movement.

Alignment (Move in the Same Direction)

Alignment ensures boids move uniformly:

1. **Find Average Velocity** – The algorithm calculates the average velocity of nearby boids within an alignment radius.
2. **Adjust Boid's Velocity** – The boid steers toward this average direction.
3. **Smooth Transition** – The transition is gradual, avoiding sudden changes in movement.

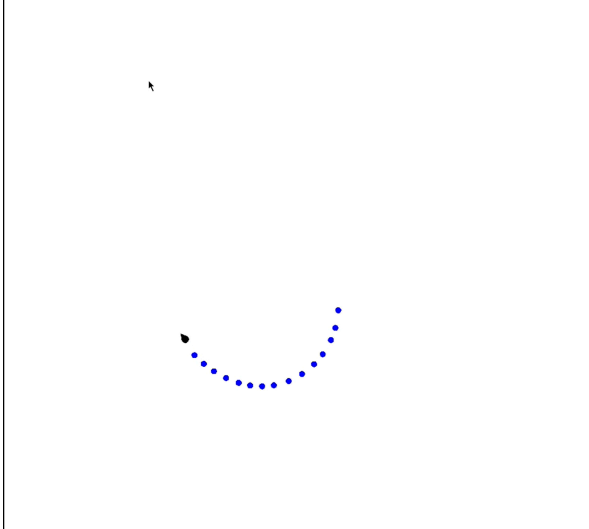
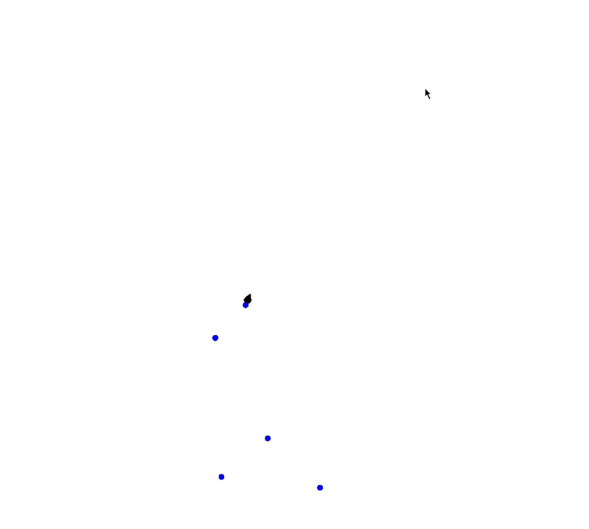
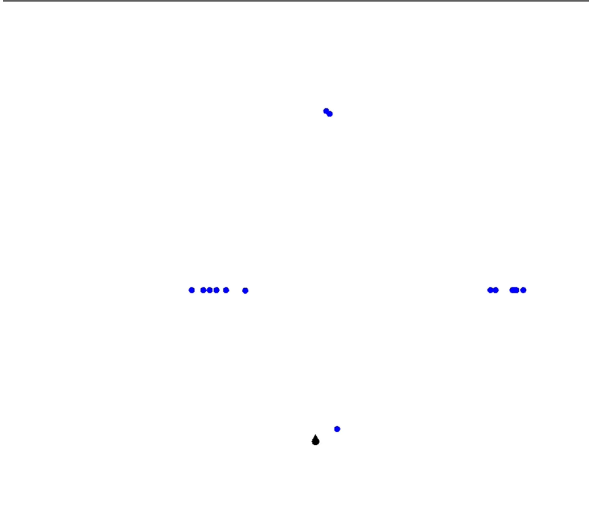
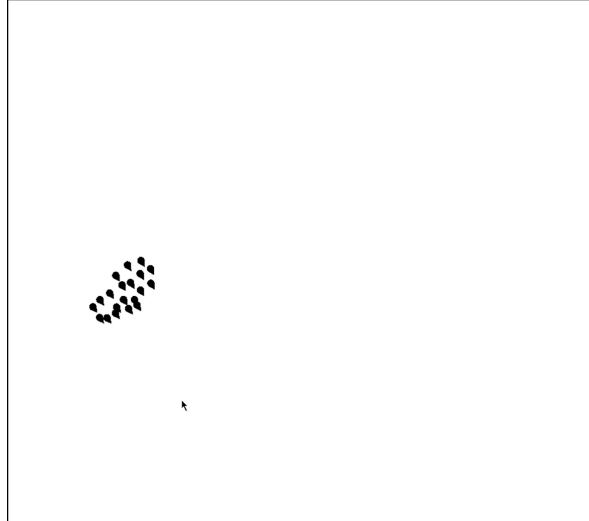
This ensures the flock moves cohesively, preventing erratic behavior.

Final Thoughts

This project gave me a better understanding of steering behaviors, kinematics, and flocking dynamics. Using a header file for organization made the development process more manageable, and I gained experience working with virtual functions in C++. While I could have optimized some behaviors further, I learned a lot and will carry these techniques into future projects.

(This write up was rewritten by chat GPT)

HW2 Write ups

Mouse Tracking	Wander random
 A visualization showing a mouse cursor (black arrow) at the top left. A series of blue dots forms a curved path starting from the cursor and moving towards the bottom center.	 A visualization showing a mouse cursor (black arrow) at the top right. A series of blue dots forms a scattered, irregular path starting from the cursor and moving towards the bottom center.
Wander edge	Flocking
 A visualization showing a mouse cursor (black arrow) at the bottom center. A series of blue dots forms a path starting from the cursor and moving towards the top center, with some dots scattered along the way.	 A visualization showing a mouse cursor (black arrow) at the bottom center. A dense cluster of black dots is located in the upper left quadrant, representing a flock.

Chat Gpt prompts are in a folder in called CGPT in the zip