School of Physics, Engineering and Technology

[ELE00055I-S2-A] Software Design
Flocking Program

Individual Report

31/05/2025

Exam Number
Y3936305

Flocking Algorithm

The core of my flocking simulation is based upon the original Boids algorithm developed by Craig Reynolds. This algorithm seeks to implement three steering behaviours - Cohesion, Separation, and Alignment - which, when combined, allow for flocking characteristics to emerge. In addition to the three core flocking behaviours, I also implemented obstacle avoidance and collision handling. These features enable boids to actively steer away from obstacles in the environment, and if a collision does occur, the boid will respond by bouncing off the obstacle in a smooth and visually coherent manner. Each of these behaviours besides collision handling are controlled by adjustable strength values and a visibility radius which allows the boids to see and react with their environment accordingly.

To enable flocking behaviours, each boid first performs its visibility check to identify the nearby boids/obstacles that will influence its steering calculations. This check starts by utilising a spatial grid, which significantly reduces time complexity by limiting the search for potential neighbour boids to only nearby grid cells, rather than checking every boid in the simulation. I did not implement a spatial grid for obstacles, as there are only three obstacles in the simulation – making a spatial grid overkill. For each neighbour boid, the Euclidean distance is then calculated, if this distance falls within a specified range, then the neighbouring boid is added to a 'neighbours' list. If a boid in the neighbours list is not the current boid, and falls within the current boid's 270° field of view, it is considered visible. A similar process is applied to obstacles: if the Euclidean distance from the boid to an obstacle is within range and the obstacle is within the 270° field of view, it is also considered visible

For each visible boid/obstacle, a behavioural vector is calculated based on its proximity to the current boid. In my algorithm, the vectors are weighted by distance, so that closer neighbours have a stronger influence than those farther away. This is done by dividing the Euclidean distance to the neighbouring boid/obstacle by the desired range, then subtracting that value from 1. For example, a boid 10 pixels away when the range is 50 pixels would yield a weight of 1 – (10/50) = 0.8.
Cohesion, Separation, Alignment, and Avoidance vectors are all calculated slightly differently, and are done as follows:

- In cohesion, the vector is scaled by the positive weight, pulling the boid towards the neighbour.
- In separation and avoidance, the vector is scaled by the negative of the weight, encouraging the boid to repel.
- In alignment, the neighbouring boid's direction is converted into a unit vector, which is then scaled by the weight.
- If none of these modes apply, a zero vector is used.

All contributions and weights are accumulated for the specified mode, and the resulting vector is then scaled by the desired strength factor divided by the total weight. This produces the final output vector representing the influence of the selected behaviour.

Finally, collision handling is implemented as a last case resort if the boid was not able to successfully avoid the obstacle in time. This is calculated separately as it isn't a flocking behaviour per se, but rather a one-time response to a detected collision. The process begins by generating a hitbox for each obstacle and for the boid's next position. If the boid's hitbox is projected to overlap the obstacle's hitbox, a collision is detected. The bounce direction is determined based on which side of the obstacle the boid collided with, resulting in a perpendicular bounce that simulates a natural deflection.

My design

Overview

The design of my flocking simulation project aims to include appropriate Object-Oriented Design Practices as much as possible. My project is organised into distinct and sensible packages, each containing fully commented classes which have a clear, singular purpose and contain modular methods throughout - promoting both maintainability and scalability. At its core, the architecture of my project is built around encapsulation, inheritance, polymorphism, and abstraction via interfaces.

Encapsulation

In my flocking simulation, I use encapsulation to maintain data integrity and manage controlled access to object states. A prime example of this is my SpatialGrid class. It keeps its cellSize and a Map of grid cells private, completely hiding its internal GridCell class and grid structure from other parts of the program. SpatialGrid instead offers controlled access through public methods like insert(), getNeighbours(), and clear(), which allow other classes to interact with the grid without needing access to any of its code.

This approach prevents direct, potentially erroneous, manipulation of the grid structure and ensures accurate spatial queries. For instance, when a DynamicTurtle needs to find nearby turtles, it simply calls getNeighbours(), it doesn't need to know how the spatial grid functions internally.

Inheritance

The project features a well-structured inheritance hierarchy, particularly evident in the design of the turtle classes. At the base is the Turtle class, which is responsible for fundamental operations such as rendering and basic movement. This class is extended by DynamicTurtle, which introduces advanced behaviour like dynamic movement, flocking interaction, and spatial awareness. Further behaviour specialisation is achieved through the use of TurtleBehaviour, which encapsulates individual flocking rules such as cohesion, separation, and alignment. This layered approach facilitates strong code reuse and maintains a clear separation of concerns: Turtle handles graphical and positional essentials, DynamicTurtle governs motion and neighbour interactions, and TurtleBehaviour dictates how boids respond to their surroundings.

Similarly, Shape serves as an abstract base for geometric shapes. The Shape interface defines a common contract for geometric objects, while the Rectangle class provides a concrete implementation for rectangular obstacles.

Polymorphism

My project also utilises polymorphism within its turtle hierarchy. The FlockingProgram class interacts with all boids uniformly as instances of the DynamicTurtle class, despite the fact that individual boids may exhibit distinct behaviours. DynamicTurtle serves as a base class that is extended by more specialised implementations such as TurtleBehaviour, each overriding key methods, such as update(), to define unique behaviour patterns.

This polymorphic design allows FlockingProgram to iterate through a collection of DynamicTurtle objects and invoke shared methods without needing to differentiate between specific subclasses. For instance, a single call to update() on each turtle may result in diverse actions: one turtle might apply standard flocking rules, while another incorporates additional logic such as obstacle avoidance.

Abstraction via Interfaces

My project effectively utilises abstraction via the FlockingGUI.SliderCallback interface. This interface establishes a clear contract for responding to GUI slider value changes, while abstracting away the internal specifics of user interaction handling. The SliderCallback interface defines methods like turtleCountChange(), speedChange(), and various behaviour adjustment methods for flocking factors such as cohesion, separation, and alignment. These methods provide a standardised way for the core program to react to user input, regardless of its origin. This abstraction allows for a clean separation of concerns: the GUI manages user interactions, while the core logic independently processes behavioural changes.

By decoupling the GUI from the program logic, the design enhances modularity and maintainability. For example, while the current implementation uses sliders to update values, the underlying logic would remain functional if inputs were instead provided through an external file.

Program Implementation

Core

FlockingProgram

The Core classes of the program are FlockingProgram which contains the central logic of the simulation, and FlockingGUI which handles the complete setup of the user interface.

The FlockingProgram class serves as the simulation's core, orchestrating the simulation's top-level control. It manages boid agents, obstacles, and a spatial partitioning grid for optimisation, while also launching the GUI. In each game loop iteration, it clears the screen, updates boid positions based on flocking rules and environmental obstacles, draws each turtle, and pauses briefly to maintain a steady frame rate. FlockingProgram is also responsible for applying any changes made to the flocking behaviours when they are adjusted via user input from the GUI.

Despite its multiple responsibilities, FlockingProgram stays lightweight, by delegating rendering and UI updates. This design keeps it focused on high-level simulation flow, ensuring manageability and extensibility.

FlockingGUI

The FlockingGUI class builds the user interface for the simulation. The layout uses a combination of BorderLayout and FlowLayout, with the main simulation canvas placed at the centre of the main frame. A collapsible menu containing sliders and buttons is positioned on the right side of the window, and can be toggled via a button in the top-right corner. This design keeps the menu unobtrusive—as it does not overlap the canvas but instead expands or contracts the entire window when shown or hidden.

The menu is organised into six key sections: General, Cohesion, Separation, Alignment, Obstacle Avoidance, and Buttons. Across these sections, users can interact with ten labelled sliders—including controls for turtle count, speed, and the range and strength of each flocking behaviour—as well as two buttons: 'Disable All Behaviours' and 'Reset to Defaults'.

All UI event listeners—including those for sliders, buttons, and the menu toggle are defined within FlockingGUI. These listeners immediately propagate changes to the main FlockingProgram, ensuring real-time simulation feedback.

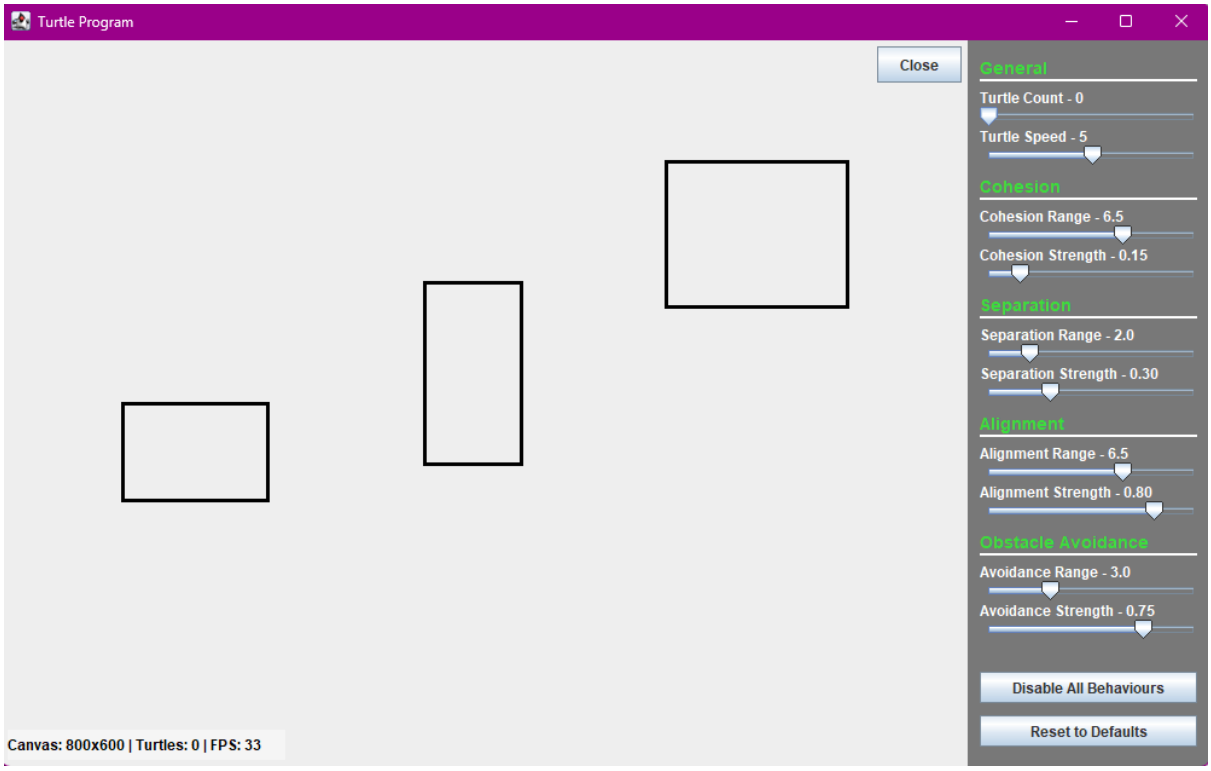


Figure 1 - My GUI layout

Boids / Turtles

Turtle

The core foundation of the simulation's movement and rendering is within the Turtle class. This class was written first as a way to provide basic drawing and movement logic for a turtle that can be observed on the screen. The class is responsible for tracking the turtle's position, orientation, and pen state, and supports basic methods which allow the turtle to turn and move. This class serves as a minimal and generic base class which allows for a clean separation between graphical rendering logic and flocking behaviour, allowing it to be extended into other classes to serve a more specific purpose.

DynamicTurtle

The DynamicTurtle class extends the base Turtle class, providing foundational capabilities for turtles designed to participate in behavioural simulations, such as flocking boids. While the Turtle class handles low-level drawing and basic movement, DynamicTurtle defines the 'how' a boid should behave in a simulation through the use of new higher-level methods. These methods allow the DynamicTurtle to implement randomised angular velocity, enabling autonomous and natural-looking movement, as well as being able to update this angular velocity according to behavioural factors caused by flockmates/obstacles. Additionally, DynamicTurtle handles edge-wrapping logic, so that when a boid moves beyond the canvas boundaries, it reappears on the opposite side, preserving the simulation.

DynamicTurtle is the 'how' focusing on the underlying simulation mechanics designed to be extended in TurtleBehaviour where the 'what' will be defined. This separation promotes modularity and flexibility.

TurtleBehaviour

The TurtleBehaviour class extends DynamicTurtle, which extends Turtle, making TurtleBehaviour a subclass of Turtle through inheritance. TurtleBehaviour is a class that operates on a DynamicTurtle to define and apply behaviours, serving as the final layer in the behavioural hierarchy of the simulation. While DynamicTurtle defines 'how' a turtle can behave through movement mechanics and environmental interaction, TurtleBehaviour defines 'what' the turtle chooses to do at any given moment, based on its local perception and simulation parameters.

TurtleBehaviour encapsulates the core flocking logic, combining cohesion, separation, alignment, and obstacle avoidance into a unified system that defines each boid's real-time behaviour. This is achieved by holding references to all turtles and obstacles in the simulation, and using them to determine overall steering forces relative to one another. It allows for external influence over behaviour strength/radii from changes in slider values.

Essentially, the TurtleBehaviour class takes movement potential and converts it into purposeful simulation logic, executing the final step in a layered design that separates rendering (Turtle), movement dynamics (DynamicTurtle), and behavioural flocking (TurtleBehaviour).

Shapes

Shape

The Shape class is a very basic and lightweight class which defines a foundational geometric abstraction for geometric shapes that can be drawn on a canvas using a turtle. It provides a consistent contract for all shape types to implement, ensuring that any shape can be queried for its size, angle, and position.

Rectangle

The Rectangle class extends the Shape interface and represents a rectangular region. It provides concrete implementations of the containment and get closest point methods, allowing boids to detect and respond to rectangular obstacles uniformly.

Tools

SpatialGrid

The SpatialGrid class introduces a spatial partitioning system that divides the canvas into a uniform grid of cells. This enables efficient querying of nearby boids by restricting the nearby boid searches to only relevant grid cells, this significantly improved the performance of my simulation, especially when multiple calculations were occurring in dense situations, such as when a large flock collided with an obstacle.

Each cell in the grid maintains a dynamic list of TurtleBehaviour instances currently occupying that spatial region. The grid is rebuilt each frame to reflect the latest boid positions, allowing for accurate calculations. This strategy minimises calculations by avoiding a brute-force approach to checking if another boid is within range across the entire flock. This allows for my flocking algorithm to handle large numbers of boids in real time.

Vector2D

The Vector2D class provides a lightweight 2D vector representation, supporting basic mathematical operations such as addition, subtraction, scalar multiplication, and magnitude calculation. It serves as the numerical backbone of the simulation, enabling consistent and readable manipulation of direction, velocity, and position data throughout the system.

Whilst Double could have been used for calculations, I felt vectors were much more readable and simplified mathematical functions within my code, and so felt it was worth adding.

SystemProperties/Utils

The SystemProperties and Utils classes were two classes which I did not design, and were downloaded from the VLE.

The SystemProperties class prints out system properties of the computer it is running on, including current Java version. The Utils class pauses the simulation for a set period of time in milliseconds.

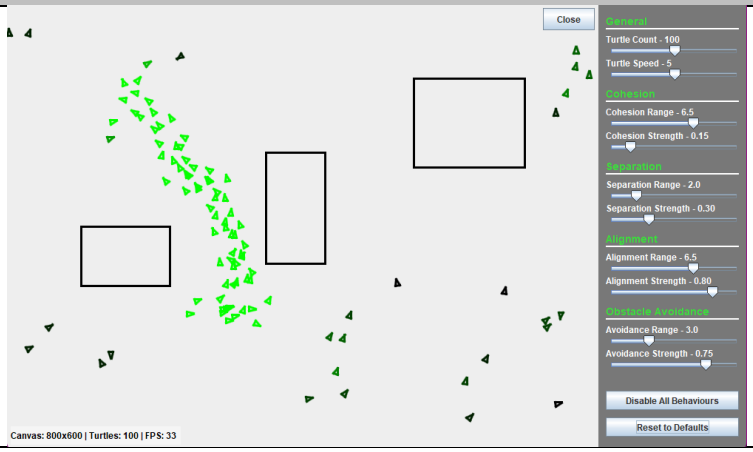
Geometry/Drawing

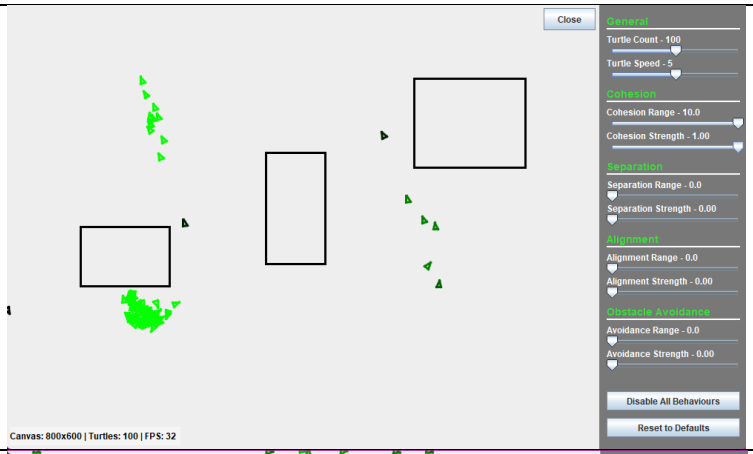
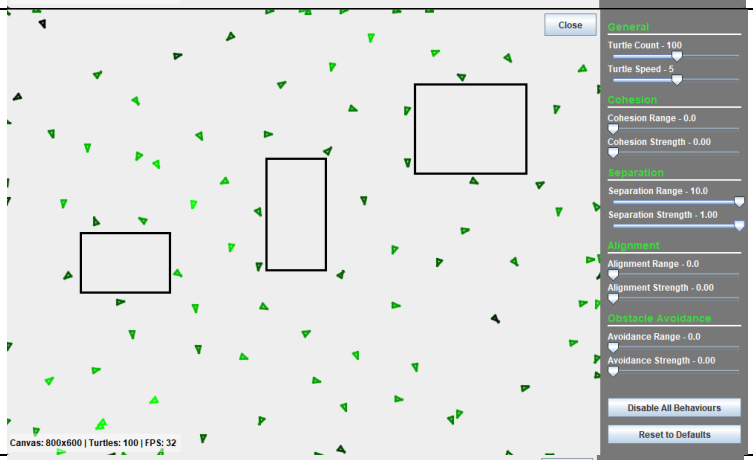
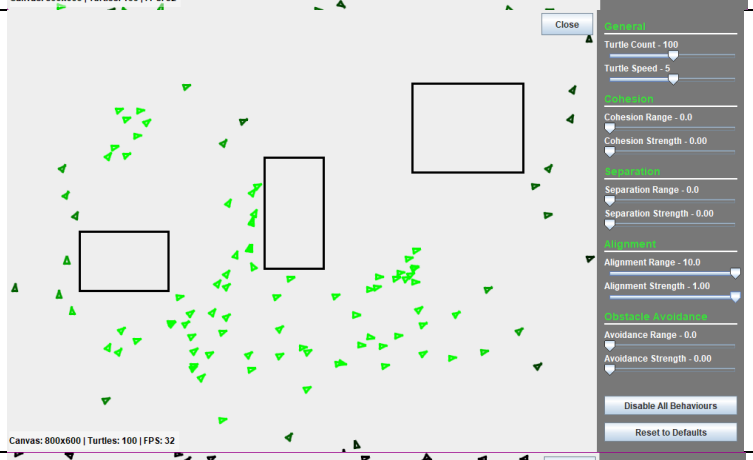
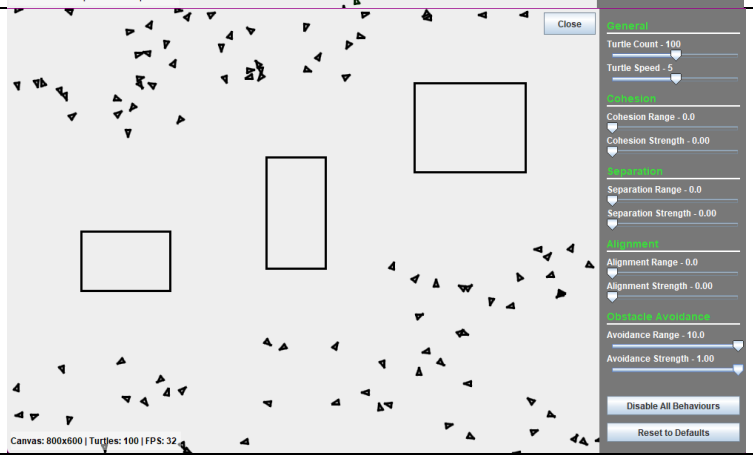
LineSegment/CartesianCoordinate/Canvas

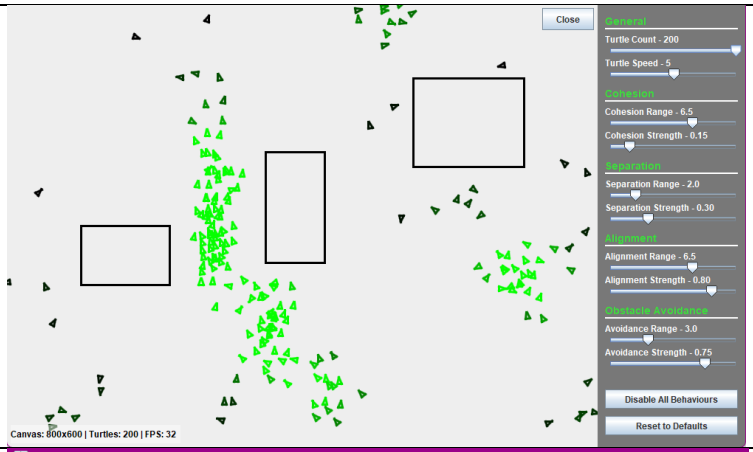
Similarly to SystemProperties and Utils, LineSegment, CartesianCoordinate, and Canvas were classes which I did not design and were instead downloaded from the VLE.

The Canvas class represents the surface used for rendering the simulation. CartesianCoordinate defines a point in 2D space on the canvas, which LineSegment uses to draw a line between two such points.

Tests and results

Test	Procedure	Expected result	Actual result	Photo evidence
Realistic Flocking	Set all behavioural sliders to a reasonable value (as seen in photo)	Boids should form smooth natural flocks	Boids form smooth natural flocks (Evidenced by the green coloured boids, which indicate flocking behaviours)	

Cohesion Test	Maximise cohesion, set other behaviours to 0	Boids should clump together	Boids clump together, much more tightly than they did in the realistic flocking test	
Separation Test	Maximise separation, , set other behaviours to 0	Boids should all seek to keep their distance	Boids stay separated from eachother	
Alignment Test	Maximise alignment, , set other behaviours to 0	Boids should all move in a general uniform direction	Boids all move in a rough general direction together	
Obstacle Avoidance Test	Maximise Avoidance, , set other behaviours to 0	Boids should not display flocking, but should avoid anywhere near obstacles	Boids avoid obstacles, and also do not display any flocking (no green coloured boids means no flocking behaviour present)	

Stress Test	Add the maximum number of boids (200)	There should be no or minimal lag	No lag observed, note the FPS count remains the same (32)	
Menu Test	Close the menu	The menu should slide shut and retract the entire window, leaving the main canvas unchanged	The menu slides away correctly, canvas is unchanged (note the window's minimise, maximise and close buttons are in the correct position)