ELE00041I-A Java Programming Flocking Simulation

Abstract

This report is written to include explanations of the design used to reproduce a flocking simulation, modelling coordinated animal motion, originally developed by Craig Reynolds.

Within this report there will be an overview of how a flocking system operates and the three parameters which control it. Following this, an explanation to the program design used in this assignment will be written, including the choices made when structuring the classes and graphical user interface (GUI) which the user sees.
The program implementation will be explained, going further into the details of how and why implementations were made, specifically looking at the flocking algorithms used and how the GUI works.

At the end you will find an evaluation which includes a summary of the test procedures used and the results achieved, analysing the successful implementations as well as the limitations and unsolved problems (bugs).
Furthermore, this will look at the overall quality and performance and finish with a conclusion stating some improvements for further adaptions.

Explanation of the Flocking Algorithm

Overview

The original flocking algorithm was designed and developed by Craig Reynolds, creating it to model coordinated animal motion such as birds, called “boids” in this program. The algorithm works based on three independent parameters, which each give the flock unique steering behaviours. These parameters control each of the boid’s individual properties, and controls how the move based on the angles and velocities of the nearby boids in the “flock”.

Explanation of the flocking properties

The three parameters mention above are alignment, cohesion, and separation. These can all be controlled individually for each boid, to cause different flocking to be seen.

Alignment

Alignment is the property which controls how much the individual boids should steer themselves into the average direction of the boids withing a specified radius around them. As can be seen in Figure 1, the current boid being updated is the green one, this boid is directed slightly off from the average direction of the surrounding blue boids. In this case, the alignment works by directing the green boid towards the average direction of all the other blue boids within the grey radius.
The amount that the green boid turns by towards the average is determined by the alignment constant, which when at its maximum will mean the green boid turns very quickly towards the average direction. However, when this value is very small, close to zero (which would mean no alignment occurs), the green boid will turn by much smaller amounts each time it is updated. Hence creating less alignment visually across all the flocks visible on the GUI window.

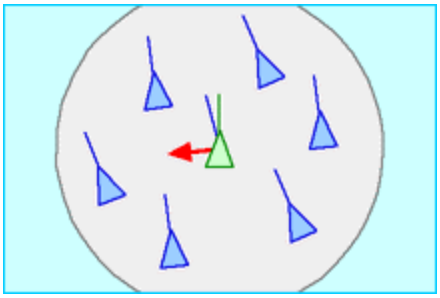

Figure 1 Visual diagram representing Alignment

Cohesion

Cohesion is the property which is based on the centre of mass of local boids. It controls how much the individual boids should steer themselves in order to move towards the centre of mass of the local boids.
Figure 2 again shows the current boid being updated is the green one. All the boids within the local grey radius have a centre of mass shown by the dot connected to all the boids.
The cohesion algorithm finds the centre of mass of these local boids, and then calculates how much the green boid should turn in order to move towards the centre of mass. When this is iterated over the whole group it makes them all steer towards the centre, grouping into a much smaller group.

When cohesion is at its maximum, the green boid will turn very sharply towards the centre, meaning all of them cluster together much faster. However, when this value is very small, close to zero (again, at zero no cohesion occurs), the amount by which the boid turns is much smaller meaning the grouping together is much more gradual across all flocks.

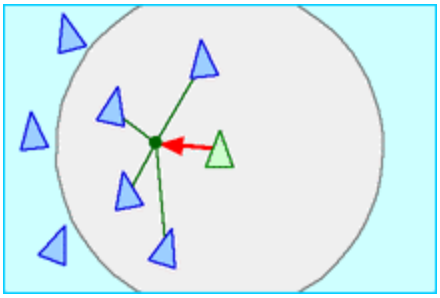

Figure 2 Visual diagram representing Cohesion

Separation

Separation is very similar to cohesion and uses the same algorithm to calculate the local centre of mass. However, rather than directing a boid towards the centre, it steers it in the opposite direction, i.e., away from the centre. Hence separating the flock and making them less clustered. Figure 3 displays how this works visually. When the separation algorithm is applied, the centre of mass is calculated, which in this case is at the green boid. The algorithm then calculates the difference between the current direction and the direction that points 180 degrees away from the centre of mass. The green boid is then steered by a fraction of this angle difference, hence separating the boids.

When separation is at its maximum, the green boid will turn very sharply away from the centre of mass, in the opposite direction, meaning all of the boids separate out much faster. However, when the separation is very small, close to zero (no separation occurs at zero exactly), the amount by which the boids turn is much smaller, making the flock separate more gradually and smoothly.

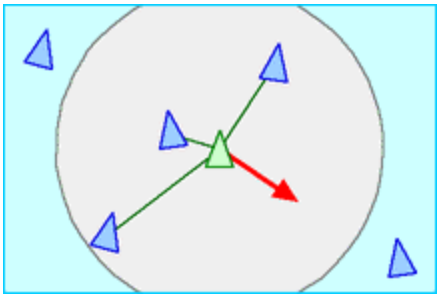

Figure 3 Visual diagram representing Separation

Design Explanation

This flocking simulator has been designed to incorporate multiple good programming techniques and has been structured in a way that allows for an efficient class structure so that the program is as modular as possible.

Class Structure

The class structure starts with an entry class which contains the main methods to run the flocking simulator. This class then handles all other method calls in order for the simulation to setup and run continuously. From within the Flocking Simulator class, all the GUI and birds will be setup from the constructor, hence these are the first things completed when the program is run.
Each class has been put into a package which describes what type of class it is. Figure 4 shows the packages and the class structures within them. Each one of these packages is named to represent a top-level view of the components to the program, which allows for the modular design and enables new classes to easily be put into its respective package.

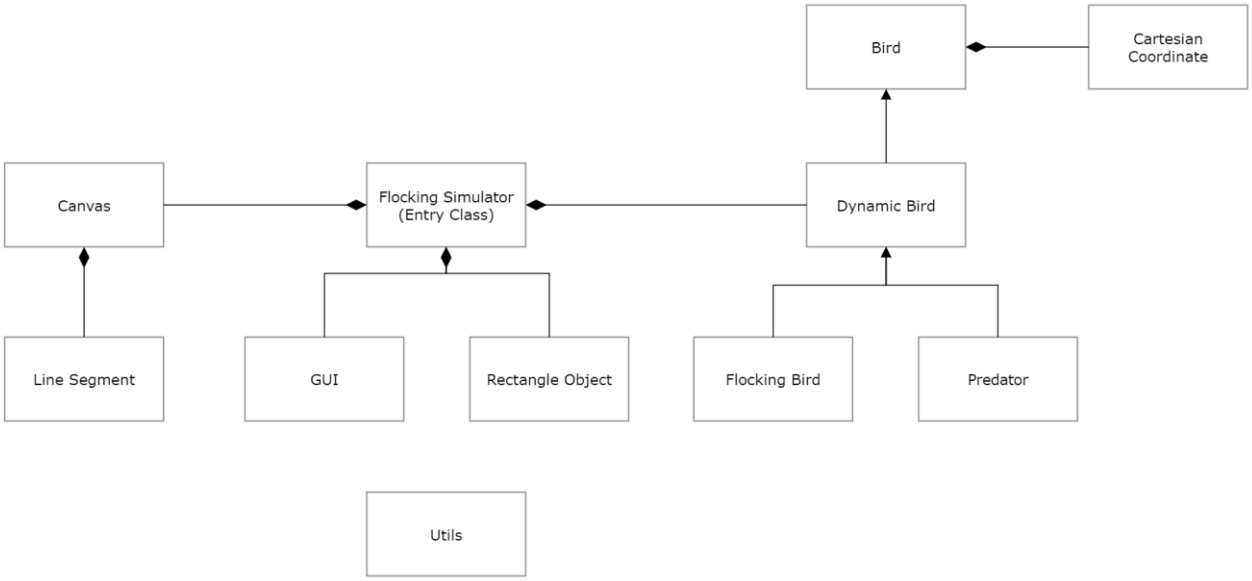

Figure 4 UML Class Diagram of the Flocking Simulator

The bird package is the main package which contains all the algorithms for the flocking boids and the methods providing functionality to these “birds”. These classes have been designed using inheritance to break down the functionality of the birds into various types. Bird is the parent class which contains the ability to draw and undraw the bird, as well as move and turn it. Dynamic Bird is-a Bird as it inherits from Bird but adds the functionality to dynamically update the position of the Bird as well as avoid objects. Flocking Bird is then a further sub class which is-a Dynamic Bird. This Flocking Bird

class implements the flocking algorithms, handling the key flocking movements whilst still having access to the fundamental methods in its parent classes.

As well as inheritance, composition has been used to provide better code reusability. Within the main class “Flocking Simulator”, the class ‘has-a’ list of birds, and a list of objects. This is essential for the program to run as these two lists are utilised throughout the main loop. Composition can also be seen in the Bird class by instantiating a Cartesian Coordinate object to give each bird a position on the canvas.

GUI Design

By keeping all the GUI within a separate class, it makes it clear to read through and allows a set of methods to be called when initialising the program. The GUI has been designed to be user friendly and simple to use. It utilises four “JPanels” to hold all the buttons and sliders on, along the top and bottom of the window. These buttons and sliders can be interacted with through simple mouse clicks and movements allowing for the program to be very easily controlled.

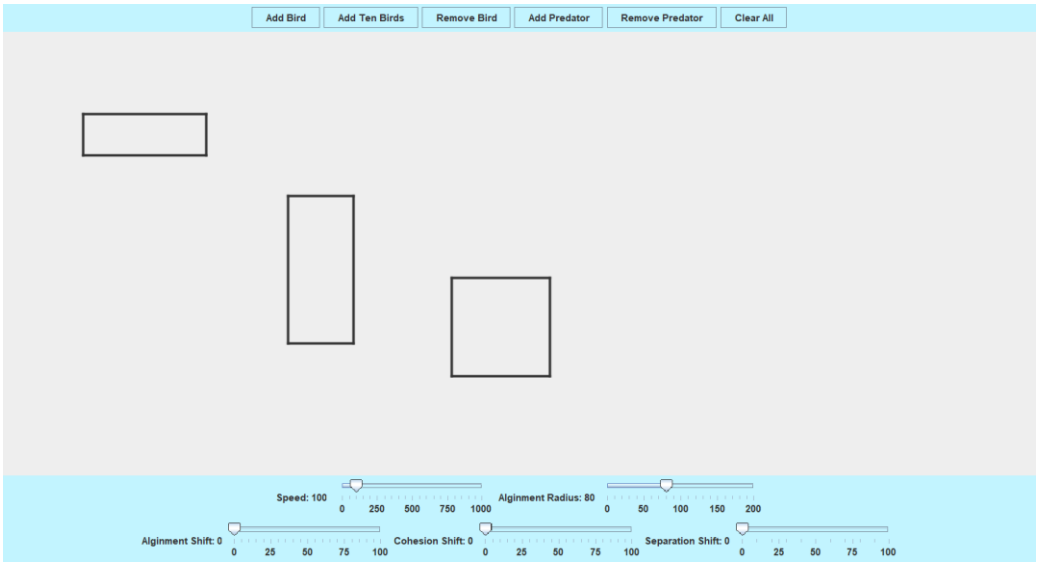

Figure 5 Screenshot of the finished program and its GUI design

Figure 5 shows this design, and it can be seen that it fits nicely together. All the sliders have been placed along the bottom, and all the buttons along the top. This distinction allows all flocking algorithms to be controlled from the bottom, and all birds and predators to be controlled from the top panel.

Main Program Loop – High level overview

The main program loop has been simplified into a flow chart as shown in Figure 6. By utilising a high-level flow chart, the user is able to see what the program is doing rather than how it’s doing it. From looking at this representation it shows that the program first sets up and initialises all variables then proceeds to loop and update the flocking behaviour input by the user from the GUI until told to stop otherwise.

The program loop was designed into three main parts. Firstly, any birds on screen should be undrawn, the birds then update their positions and directions. This is done by calling a series of methods to flock, avoid objects and for predators to chase prey, and finally update, which updates the position of the birds given the framerate set by a constant.
The birds are then all redrawn onto thee screen and a delay occurs so the user can see the birds in their new positions with their behaviour. This is the simplest design that is most efficient and effective to work to the design specifications.

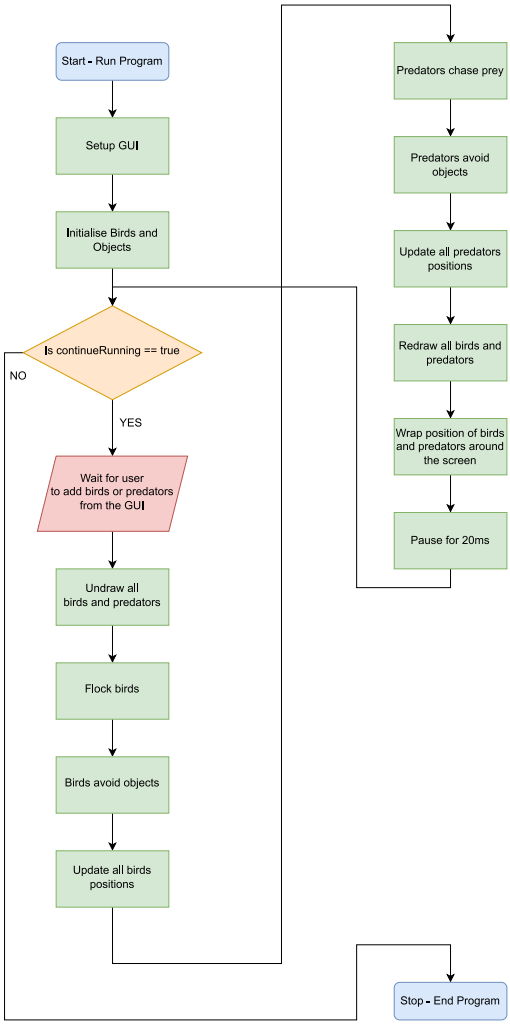

Figure 6 Flow chart displaying a high-level representation of the main program loop

Program Implementation

Initial implementation of the program was simplified by having classes which were built up within lab scripts. These classes could be used as a basis for the flocking simulator program. However, some modifications had to be made, including renaming any packages and classes to fit the program specifically, as well as tidying up any code or removing any that would not be required.
From this point the addition of new functionality within existing classes and the creation of new classes could commence and by utilising the existing classes, a structure to the program was put together incorporating the design choices described above.

Use of Composition & Polymorphism

Expanding upon the use of composition within the Flocking Simulator class, it can be seen that since the flocking simulator requires birds and objects to operate, the "Flocking Simulator" class therefore has-a "listOfBirds" and "listOfObjects" relationship and uses these throughout the class. Within the main loop, the list of objects "listOfBirds" methods are consistently called which handles all of the flocking and object avoidance.
The list is instantiated as a Dynamic Bird as follows: $private\ List < DynamicBird > listOfBirds;$. By creating the list of type Dynamic Bird, it allows for polymorphism to be used, which is achieved by adding multiple types of sub class objects into the parent class defined object. This is particularly useful since it allows one list to be iterated over and depending on the type of

object in the list, it will call its respective method to carry out the function. The added benefit of utilising polymorphism here is the type of birds are hidden from the user within the main class. Abstraction like this keeps the program cleaner and simpler for programmers and users reading the code in the future. Within the GUI class, these anonymous classes are constructed by creating a "$new\ ActionListener()\ \{\}$".

Inheritance

Inheritance is utilised within the Bird package to enable clearer and easier code to be written. Inheritance has been used here by extending the Bird class within the Dynamic Bird and extending the Dynamic Bird within the Flocking Bird and Predator, as can be seen in the class diagram in Figure 4. This has allowed the use of private and protected field variables and methods so that the correct data abstraction is used. Setting fields as protected allows the sub classes access to these fields and methods without having to expose them to the other classes.
$calculateTurnAngle()$ is an example of how protected is suitable here, since the method is called from within both Flocking Bird and Predator, so rather than duplicating it in both those separate classes, it has been implemented in Dynamic Bird and set to protected, which allows both classes to use the one method.

GUI and Anonymous Classes

Within the GUI class, all of the objects are instantiated sequentially which keeps the code clear and consistent, maintaining a high level of readability. The most efficient way to add all the GUI components to the window frame was to setup all the buttons and sliders, setting their properties such as enabling them, setting their text, and initial values. All these interfaces could then be added to their respective panels in an order that was most user friendly. The most computationally expensive operation was then left to last where all the GUI panels and interfaces were added to the “JFrame” (main window).

When one of the buttons or sliders is interacted with, event listeners have been utilised which allows the action created by the user to cause the change to happen as required for each button. Event listeners are an object which listen for fired events to occur then execute them. To implement the event listeners anonymous classes have been used. These are similar to inner classes but without a name, and they are useful since they have access to all the fields and methods of their outer classes (the class in which they are defined). These classes are only used in one place, so they are suitable for this requirement. Additionally, anonymous classes can only be used once, so when they are instantiated in one place, it makes the code more concise.

Flocking Algorithms

The flocking algorithms have been implemented within the Flocking Bird class, since these are properties of a flocking bird. There is a main method that is called from the main class to handle each of the three flocking algorithms. This takes the list of birds, and the current bird that is being updated. The impact of each algorithm on the birds, is determined by four sliders in the GUI. One of these adjusts the radius in which nearby birds will align with one another. The other three adjust the parameters (alignment, cohesion and separation) which determine how much each individual bird will flock with the surrounding birds within that radius.

To simply the process for calculating each of the algorithms, utility methods were created which created better code readability. These methods were a displacement and bearing calculation methods, returning the displacement between two birds and the bearing from one bird to another respectively. By utilising these methods and applying the logic required for the algorithms, each factor for the flocking was implemented.

Object Detection

The first part to object detection, was adding them onto the canvas. A simple method was created which was called from the main constructor, and created 3 objects of type Rectangle, which drew the objects in at the given corner and x and y side lengths.
The rectangle class also had properties to make it easier to detect if a bird was going to collide with it. These properties were two Cartesian Coordinates, named "noFlyZoneMin" and "noFlyZoneMax", and by using these two positions the whole area of the object can be known about so it's possible to apply any object avoidance based on these two positions.

Initially the object detection just involved when the bird came into contact with the edge of the object, it would rotate 180 degrees and continue with its motion. This was very basic and didn't work so well with flocking due to birds which collided travelling off in a random direction away from the flock.
A solution to improve this meant checking the specific edge that the bird was flying at. Then an invisible boundary was applied 20 pixels away from the edge so that the birds head didn't fly through the wall of the object.
Once the bird came within this boundary it would rotate 45 degrees away, either plus 45 degrees or minus 45 degrees, depending on the initial direction the bird was heading towards the object in the first instance. This method was placed within Dynamic Bird, which allowed both the flocking birds, and the predators to call this method, since both need to detect and avoid objects, and since Dynamic Bird is the parent class of both sub-classes, they each have access to all the fields and methods.

Test Procedures and Results

Implementing this program required a lot of testing to get results which are as user friendly and accurate as possible. This section aims to provide an in-depth analysis of the testing methods used as well as results achieved throughout the testing process.

The main sections which required testing involved the flocking birds and predators, since these involved the most complex of the algorithms. Once the initial algorithms for each of these features had been written, they could be tested and improved over time.
Within the flocking birds, each algorithm for the separation, alignment and cohesion needed to be tested. This was achieved by running the program and watching how the birds flocked around the screen. To make this motion as accurate as possible, the values that were received from the sliders were divided by a custom constant that was adjusted until the best scaling was applied which produced the most accurate flocking appearance. These constants were responsible for controlling how much each bird turned in its respective direction depending on the algorithm being applied.

When testing the predators, a basic predator was implemented initially, which when a bird, counted as prey (any flocking bird), came within a specified radius the predator would eat the bird straight away and it would be undrawn from the canvas and removed from the list.
This allowed all the simple functionality such as getting the prey to actually be removed from the list and successfully interact with the predator sorted. Upon having this basic implementation, further work could be added to make it so the predator chased any prey within its sight radius, and by giving the predator a greater speed when chasing prey, it could catch up and eat the prey once within a much smaller radius.
It was worked out that giving the predator a speed 1.5 times faster than the prey when chasing the prey was suitably fast to catch up but slow enough that it was possible to see the predator chasing the prey.

Throughout development, whilst testing multiple features involving birds, one of the main testing strategies involved using two singular birds and spawning them at the same position, by temporarily removing the random spawning feature.
This made debugging and stepping through code much simpler, as only two birds were required to

iterate over to test the birds' values. This was particularly useful when testing cohesion and separation, since it was much easier to apply manual calculations to work out where the centre of mass should be for one bird (right at the centre of the bird itself), used to compare to the codes calculated values, rather than it being much more complex with 10 birds. For the purpose of testing having any more than two birds is unnecessary considering that the outcome will be the same as with multiple. Since the flocking applies the algorithm to a singular bird based on the birds in the surrounding area.

Evaluation

From looking and experimenting with the final program, the flocking tendencies can be clearly seen to be working well, and all the features work succinctly together. Birds will vary the amount they flock by when the sliders are varied, and when all of them are set to zero, no flocking occurs which is as expected.
The object detection has minimal bugs, and birds do not seem to get stuck within the objects, meeting the requirement for objects to be impervious to individuals. Additionally, the extra feature of predators which chase prey has been implemented successfully and enables the program to show the use of polymorphism efficiently.

Each subsystem to the program was developed thoroughly and tested to a point which would meet a success for the program task. Within the program, where a small task or calculation was required, a method has been added to provide the feature. This has enabled the code to be well structured, and easy to understand how each section works.

A further modification for future improvements, would be to improve the intelligence of both the flocking birds and the predators, so that the flocking birds will actively try to avoid predators by overriding their flocking behaviours. And predators would be given a hungry and not hungry state, so when they are not hungry, they glide around peacefully until a certain time in which they become hungry and start chasing birds to eat them.

References

Flocking Diagrams, Figure 1, 2 and 3: - https://www.red3d.com/cwr/boids/ accessed 14/05/2023