Java Programming – Written Report – Y3857545

1. Abstract

The program simulates a two-dimensional world in which the user can create individuals, called entities. These entities follow the same set of simple rules when they move, resulting in more complex behaviours. Eventually, they move as a coherent group, known as a flock. The user can vary the rules the entities obey, known as the "flocking parameters." A predator, from which the entities will flee, can also be added to the simulation.

The task was broken down into steps, which, if carried out chronologically, will allow the programmer to test their new code using the old code, making this an iterative design process. The steps identified were the creation of the GUI, the entity class, the obstacle and finally the creation of any extended features.

An ideal solution using the mouse to place entities rather than placing them randomly would be more intuitive. More extended features could also be implemented, such as predators "catching" entities and a welcome screen when the program is first started. However, the current version is fully functional and would provide a good base on which to build a more consumer friendly program.

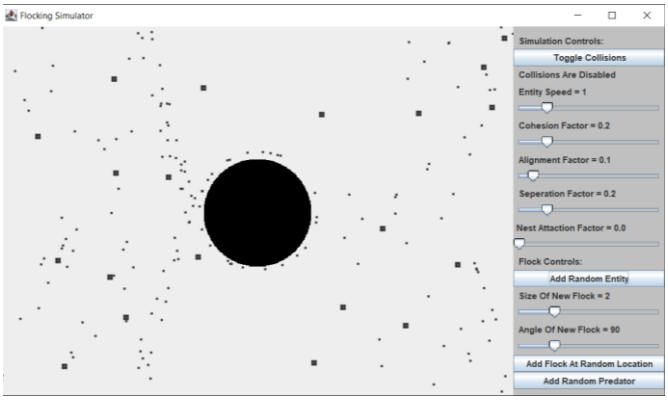

Fig. 1. A screenshot of the completed flocking simulator.

2. Problem Description and Analysis

Requirements

The company requested the creation of a two-dimensional world, in which individuals, or entities, will reside. They will travel around the world interacting with other entities using a set of rules which result in behaviour that constitutes flocking. The user must be able to control the number of entities on screen and their behaviour. An obstacle in the centre of the screen (350x250 pixels) with a radius of 75 pixels must also be created and be impervious to the entities. The simulation must also be extended to include other complexities.

Limitations

The immediate restriction is that the object must be central at all times at a fixed location. This prevents the program from being resizable as the location of the centre of the screen has been defined in the request.

Initial Analysis & Iterative Approach

The simulation is split into several key sections where the previous section must be completed before the next section can be tested. Thus, it is evident that an iterative approach should be taken during development.

Iteration is the defining of several steps, that upon completion can be tested and used as a building block in order to test the next piece of the program. Iteration will enable the simulation to be tested during and after each of the development stages, allowing adjustments to be made throughout the development process. Any compatibility issues would be dealt with during the building of the simulation, rather than at the end.

The steps identified were the creation of:

1- GUI – This includes the "world" in which the entities will exist, as well as the controls for their behaviour and creation.
2- Entities – This includes their behaviour as well as their aesthetics.

3- Obstacle – The visual creation of the obstacle as well as supplementing the entity class to ensure they avoid the obstacle.
4- Extended Features – A sub-class of its own. Iterative design allows for flexibility when programming. This allows for the outlining of several steps, with each one containing one extended feature. Each step is completed after another until either they are all functional or the time runs out. The advantage of this is that if the time runs out the program can be reverted to the last, fully functional step with no issues and some extended features rather.

There is one issue with the plan outlined above. Step one requires knowledge of how the entities will behave in order to program their behavioural controls, but the behaviour of the entities would be developed in step two. So, the actual mathematical behaviour of the simulation must be defined, but not programmed, before attempting stage one.

Extended Features

The extended features that will be implemented in chronological order are:

- Simulation speed controls – Control over the speed of every entity.
- Collision detection – Drastically change an entities direction if there is a collision.
- Predators – A type of entity that will move across the screen seeking out flocks, which will flee from it.
- Nest attraction – Draw all the entities towards a certain point onscreen.
- A high and low "resolution" mode – In high resolution mode the black dots that will represent entities would be replaced by actual pictures of the user's choice such as birds or fish, complete with a suitable background.

As many of the extended features as possible will be implemented, but as explained previously, each one will be produced iteratively until the time runs out.

3. The Flocking Algorithm

Before the actual development of the program can begin, the flocking behaviour must first be defined mathematically, as explained in the initial analysis.

To meet the core criteria for flocking, each entity must have the following features:

- Cohesion – An attraction towards the position of nearby entities.
- Separation – The opposite of attraction, a will to move away from nearby entities.
- Alignment – A need to move in the same direction as the other nearby entities
- Obstacle avoidance – A force moving the entities away from the obstacle in the centre of the screen.
- A line of sight – Entities must only be affected by other nearby entities, so a line of sight must be decided. It would be best to take an educated guess at the required line of sight and then refine it during testing.

The Flocking Algorithm – Cohesion

All entities present must be drawn towards other nearby entities to form a flock in the first place. This can be done by creating a list of all the entities in the subject's line of sight and calculating their overall average position. This will be the centre of the flock. This is done by:

$$FlockCentreX = \frac{SumOfNearbyXCoordinates}{NumberOfBirdsInFlock} \& FlockCentreY = \frac{SumOfNearbyYCoordinates}{NumberOfBirdsInFlock}$$

From here, the subject's angle of travel can be altered to draw it towards the centre of the flock. This is done by calculating the angle required to turn the subject towards the centre of the flock. To achieve this, a triangle is created, with the opposite and adjacent sides of the triangle representing the x and the y distances between the centre of the flock and the subject.

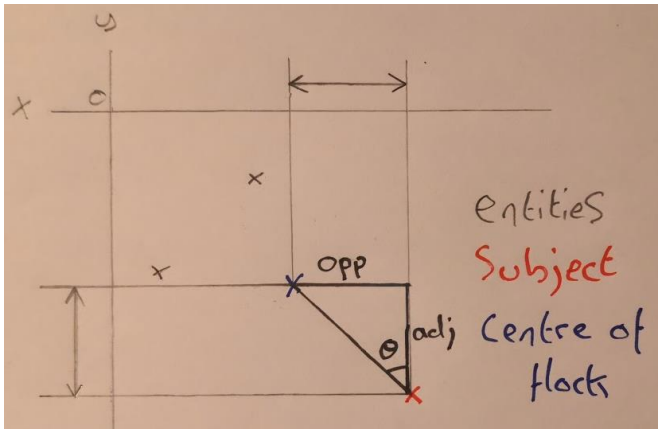

Fig. 2 – A diagram of how trigonometry is applied during cohesion.

The opposite side of the triangle is found by:

$$Opposite = CentreOfFlockX - SubjectLocationX$$

And the adjacent:

$$Adjacent = CentreOfFlockY - SubjectLocationY$$

Now theta, θ, can be found using trigonometry:

$$\theta = tan^{-1}\left(\frac{opp}{adj}\right)$$

The angle required to turn the subject in such a way that it will head towards the centre of any nearby entities, is now known. However, this angle cannot be applied as it stands, otherwise entities would all just head directly towards one another rather than move as a flock. This is where the cohesion factor comes in.

$$SubjectsCurrentAngle = SubjectsCurrentAngle + CohesionFactor * AngleRequiredForCohesion$$

Where the angle required for cohesion is θ. The cohesion factor is a number by which the cohesion angle will be reduced, creating a far more gradual, natural turn towards the centre of the flock. It is worth noting:

$$CohesionFactor \leq 1$$

This is so that the cohesion angle is only reduced or maintained and not increased. In addition, the cohesion factor is what the user will be granted control over. In summary:

$$MovementAngle = MovementAngle + CohesionFactor \times tan^{-1}\left(\frac{CentreOfFlockX - SubjectLocationX}{CentreOfFlockY - SubjectLocationY}\right)$$

The Flocking Algorithm – Separation

In order to prevent cohesion allowing entities to crash into each other and remain like that, it must be counteracted. This is where the separation factor comes in. Just as was done for the cohesion factor, an angle will be calculated that would place the subject on a direct course for the centre of the surrounding entities. However, this time the angle will be subtracted from the subject's movement angle.

$$SubjectsCurrentAngle = SubjectsCurrentAngle - SeperationFactor * AngleRequiredForSeperation$$

This will turn the subject away from the centre of the flock, counteracting the cohesion factor. Just as before, the user will have control of the separation factor. The separation factor must obey the rules set out for the cohesion factor. In summary:

$$MovementAngle = MovementAngle - SeperationFactor \times tan^{-1}\left(\frac{CentreOfFlockX - SubjectLocationX}{CentreOfFlockY - SubjectLocationY}\right)$$

The Flocking Algorithm – Alignment

In order to move as a flock, the entities in the flock must move in the same direction. This behavioural aspect will be controlled by the alignment factor. Alignment can be achieved by making a list of all the entities in the subject's line

of sight and adding up all of their current movement angles. By taking the average of all these angles the flock's average movement angle is obtained.

$$FlockMovementAngle = \frac{SumOfEntityMovementAnglesInFlock}{EntitiesInFlock}$$

The angle required for alignment can now be found.

$$FlockMovementAngle = FlockMovementAngle - EntityMovementAngle$$

This is done to obtain the angle required to turn the subject to face in the direction of the flock as shown in Figure 3.

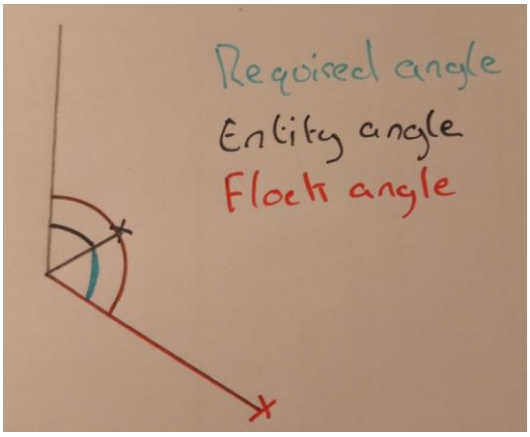

Fig. 3 – A summary of the alignment angle.

This required angle can then be applied to the subject's movement angle.

$$SubjectsCurrentAngle = SubjectsCurrentAngle + AlignmentFactor \times AngleRequiredForAlignment$$

As before, the user will have control of the alignment factor in order to control the magnitude of alignment observed. The alignment factor must also obey the rules set out for the cohesion factor.

The Flocking Algorithm – Obstacle Avoidance

As stated in the project brief, an obstacle must be implemented at the centre of the screen. The subject should make some effort to avoid this obstacle. However, if it collides with and is reflected off the object, it will create a more realistic scenario and could produce more interesting behavioural patterns in the rest of the subject's flock. In order to implement this scenario, the subject will have an obstacle avoidance factor.

Obstacle avoidance will be implemented in a similar manner to the subject's separation factor. Except, rather than using the centre of the flock as a point of reference, the centre of the object will be used as a point of reference. A more in-depth explanation of the following equation can be found in the separation and cohesion sections. In summary:

$$MovementAngle = MovementAngle - AvoidanceFactor \times tan^{-1}\left(\frac{CentreOf\text{Obstacle } X - SubjectX}{CentreOf\text{Obstacle } Y - SubjectY}\right)$$

The avoidance factor should be set to 0.5 initially, unless testing proves otherwise.

As mentioned previously, the subject could hit the object and rebound. This can be done simply by comparing the two objects locations. However, this is not related to the flocking algorithm so it will be dealt with later in the report.

The Flocking Algorithm – Extended Features

Two of the proposed extended features would use some form of the flocking algorithm. Firstly, the predators would need to affect the behaviour of the entities. This could be done in a similar way to the separation factor, except it would only be implemented if a predator was in the subject's line of sight. A list of any nearby predators would have to be compiled, then a separation factor would be computed and applied. The factor should be set at 0.5 unless testing proves otherwise.

Secondly, the nest attraction method would use cohesion and a nest attraction factor to move all the entities onscreen towards their nest. The cohesion calculation would use the nest location and the subject's current location. The nest attraction factor should be accessible, so it can be set to zero, disabling nest attraction, if desired.

4. Design

GUI – Graphical User Interface

The first step identified in the iterative development process was the design and development of the GUI. This will provide a base on which to build the rest of the program. A criterion to test the GUI was developed.

The GUI must:

- Provide an intuitive control scheme. Where this is not possible reasonable measures should be put in place to inform the user as to how they should proceed.
- Provide all necessary controls.
- Display all the entities and associated objects in real time.
- Conform to the given dimensions.
- After the GUI is instantiated, provide a while?? loop in which the rest of the program can run.

From here the problem was compartmentalised into a class structure.

- Package – Default
    - Class – FlockingProgram. The entry point into the program. This will be the centre of the program, where the main while loop and calls to the set-up methods will reside. It will boot the program up, import the necessary packages for the GUI and hand over to the GUI. When the GUI has returned control to the default program, run the main while loop.
- Package – Window
    - Class – Window. The first class to run, it will create the actual window and the two panels inside of it. One for the entity's world and another for the controls.
    - Class – controls. The second class to run, it will populate the control panel with all of the necessary controls.
    - Class – obstacle. The third and final class. It will simply create the obstacle object, which will need a location and image.

A sketch of the GUI and a list of the necessary controls can now be produced, and the program written. The diagram conforms to the desired size specifications. The canvas is where all the entities will be drawn.

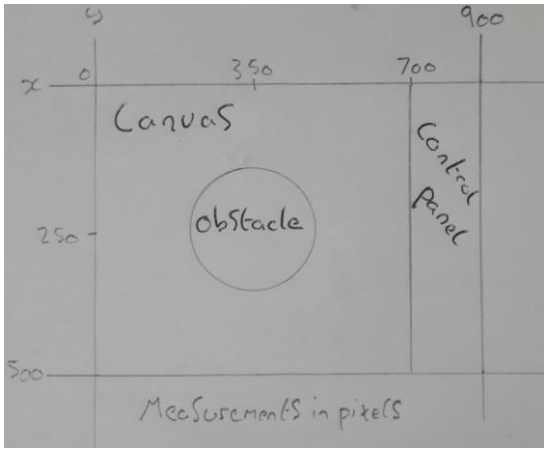

Fig. 4. The proposed layout of the program.

The controls proposed so far are:

- A simulation toggle button.
- An entity speed slider.
- Cohesion, alignment, separation and nest attraction factor sliders.

Further controls required to meet the specification are:

- A button to add a random entity.

- A button to add a random predator.
- A button to add a new flock, with two sliders to control the angle and size of this new flock.

The GUI can now be programmed, and the next part of the design planned.

Entities

The second step identified was the development of the entities. These will be representative of birds or fish travelling in a flock. In order to conform to the given specifications, the program must meet the following requirements:

- Entities must travel as a flock; this point is not actually possible, as every element in a real-life flock would have free will, amongst other things. However, in this case it will be defined mathematically in a two-dimensional space. To summarise, each entity must experience cohesion, separation and alignment as defined in section three, the flocking algorithm.
- The user must have control over how any entities are present, including having a population of zero if required.
- Entities must make an effort to avoid the obstacle in the centre of the screen and not be able to exist inside of it.
- Entities must not leave the screen, unless the user specifies their removal.

In order to meet the specification, entities will need the following features:

- A current location
- A current angle of travel
- A line of sight – defining what will and won't be included in its own "flock".

Entities will also need the following methods:

- Draw and undraw
- Move
- Apply cohesion
- Apply separation
- Apply alignment
- Apply obstacle avoidance
- Wrap entity position – when an entity leaves a side of the screen, it should reappear on the other side.
- Bounce off obstacle
- Check if inside obstacle on creation - When creating an entity, it is crucial that it is not inside the obstacle. As entities cannot exist inside of it, due to the bouncing off the obstacle they may not even be able to leave it.

Judging by the requirements for basic functionality, the entity class is quite complex. Therefore, it will be broken down into two classes: entity and random entity. Where random entity inherits entity. The classes will be stored in the entity package.

Entity will handle the basics of an existing entity which involves drawing, moving, flocking and more. However, entity will only handle the creation of an entity at predetermined coordinates. Due to limitations with the libraries used, the location of the mouse cannot be obtained. Due to this, the location of entities will have to randomised and this will be handled by the random entity class, this includes ensuring the entity is not inside of the obstacle.

The entity package can now be programmed.

Obstacle

Producing the obstacle will involve creating a PNG, the image type specified by the software used. The image will be a white background conforming to the pixel dimensions specified in Figure 4, with an obstacle diameter of 75 pixels, as specified in section two – requirements.

Extended Features

A breakdown of the proposed ideas behind the development of the extended features is as follows (as many of them as possible will be developed in the given time frame):

- Simulation speed controls.
  Simulation speed can be implemented by adding a slider to the control panel that controls the distance each entity moves per iteration.
- Collision detection.
  This can be implemented fairly simply. Firstly, a toggle button must be added to the control panel. Secondly, two for loops must be implemented, one inside of the other and both iterating over the list of every entity in existence. The first entities location will be checked against the location of every entity produced by the second for loop. Then the first for loop will move to the second entity and the process will repeat itself. If two entities collide, their movement angles should be adjusted accordingly.
- Predators.
  The current entity's line of sight should be checked for predators, if one is detected it should react accordingly. A predator should be an entity that only experiences cohesion and should be added by a toggle button in the control panel.
- Nest attraction.
  A cohesive factor should be applied to every entity called the nest attraction factor, with a control slider added to the control panel.
- A high and low "resolution" mode.
  In low-resolution mode, objects should be represented as coloured pixels, with high-resolution mode changing the coloured pixels to small PNG files and possibly adding an improved background.

After development was completed, four of the five proposed extended features were implemented due to time constraints.

5. Final Product Technical Specification

Prescribed Features

- The ability to create entities at will, randomly or with the desired movement angle.
- A control board, called the side panel in the program itself, which offers full control of the flocking parameters and the creation of entities.
- An obstacle at the centre of the screen, with a radius of 75 pixels, that all entities will try to avoid but find impossible to pass if they collide with it.
- Complete termination of processing upon clicking the cross.
- Several extended features, including:
  - Control over the simulation speed.
  - Predators that affect the behaviour of entities.
  - A cohesive factor that draws entities towards a common "nest."

Libraries and Flowchart

Several libraries from outside the source folder are required, they are all contained in Javax.swing, Java.util and Java.awt

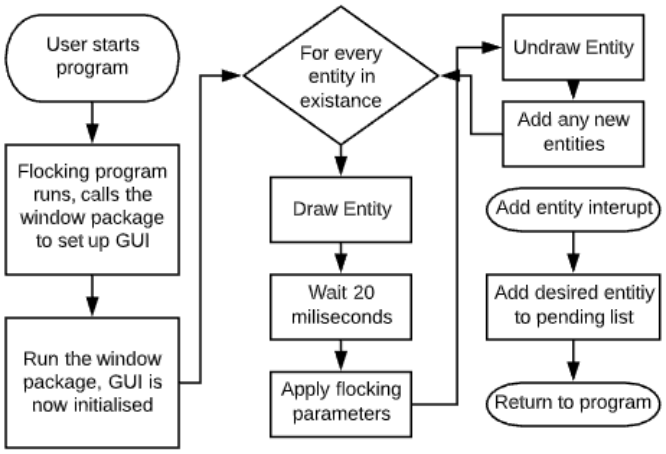

Fig. 4. A flowchart outlining the basic function of the flocking simulator.

- Package – default Package
  Contains the entrance point to the program and the base upon which the rest of the program is build. All high-level control of the program is here.
    - Class – FlockingProgram:
- Package – drawing
    - Class – Canvas.java
      EXAMINER PLEASE NOTE - Canvas.java was provided by Dr Stuart Porter as a part of Laboratory 3. As requested in the assignment, this has been made clear in both this report and the program. The file has not been altered in any way. Canvas.java must be used as is stated in the assignment itself.
- Package – entity
  The entity package contains anything related to the creation and manipulation of the entities, including the predators.
    - Class – Entity
      All the methods relating to the control of an existing entity are here.
    - Class – PredatorialEntity
      A combination of RandomEntity and Entity, PredatorialEntity overwrites some of the methods that do not apply to it, altering some others, including draw, so that a predator can be recognised easily.
    - Class- RandomEntity
      Random entity handles the placement of an entity of any type at a random, but in a valid location.
- Package – geometry
  The geometry package contains anything related to the positioning system implemented in this program.
    - Class – CartesianCoordinate
      A Cartesian Coordinate is an object that contains two values, an x and a y coordinate.
    - Class – LineSegment
      A Line Segment is made up of two Cartesian coordinates, representing the start and end of a line.
- Package – tools
  Tools contains generic tools for the manipulation of the program as a whole.
    - Class – RandomNumberGenerator
      Defines an object with an upper and lower limit that can be called repeatedly to give another random number within those limits.
    - Class – Utils
      EXAMINER PLEASE NOTE - Utils.java was provided by Dr Stuart Porter as a part of Laboratory 4. As stated in the assignment, this has been made clear in both this report and the program. I have added comments and understand the running of the class fully. However, no actual code has been altered. Any implementation of a try/catch statement to make the author's main simulation loop wait would have been incredibly similar to this.
- Package – window
  The window package contains everything related to the controls and the GUI.
    - Class – Controls
      The controls class contains the blueprints for the creation of the control panel, known in the program itself as the side panel. The user's inputs are also handled here, and the necessary actions taken.
    - Class – Obstacle
      Obstacle draws the obstacle image onto the screen.
    - Class – Window
      Window deals with the creation of the GUI, including their sizes and they layouts used.

6. Evaluation

Currently, there is an issue which makes the program lag when the collision detection is enabled. A more efficient method would be required if the program was developed further. Also, a "high-resolution" mode could be implemented, which uses pictures rather than dots would make the program more appealing. Using the mouse to place entities rather than placing them randomly would also make the game easier to use. However, the simulation itself functions well and several extended features have been implemented. So, the program meets the design specification fully, making this a successful project.