

Flocking Simulator

ELE00005C

Analysis	2
REQUIREMENTS	2
PRINCIPLES OF FLOCKING	2
STEERING BEHAVIOURS	2
COMPUTATIONAL COMPLEXITY	3
Design	3
SYSTEM OVERVIEW	3
STEERING BEHAVIOURS	4
UML CLASS DIAGRAM	4
GUI DESIGN	5
Implementation	5
PROJECT STRUCTURE	6
CHANGES FROM DESIGN	6
OPTIMISATION STRATEGY	7
Conclusion	8
FINAL SPECIFICATION OF APPLICATION	8
FURTHER DEVELOPMENT	8
FINAL SUMMARY	8
Bibliography	8

Note: application must be compiled and run with Java 8.

ANALYSIS

Requirements

The requirement specification is extracted from the assessment document. [1]

Required

- Simulate flocking behaviour in a 2D world
- Display simulation graphically on computer screen
- Each individual should interact with others using simple rules
- Number of individuals and flocking parameters should be set by the user

Optional

- User control of simulation speed
- Obstacles
- Collision detection
- Other types of individuals to interact with

Principles of Flocking

Flocking is when steering behaviours applied to many individuals in a group result in emergent behaviour for the entire group. [2]

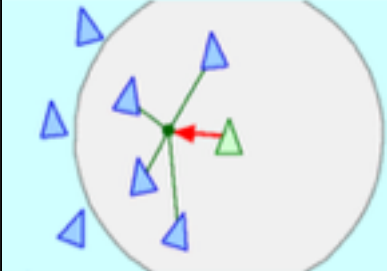
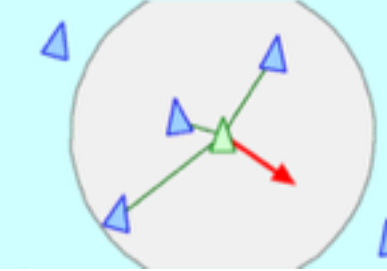
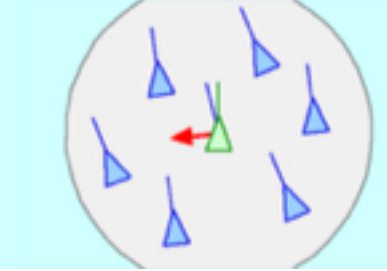
Each flock appears to act as one body, even though they appear only through the interaction of simple rules in the individuals.

For the purposes of this assessment, I will refer to the individuals as 'agents' and to the entire group of individuals in the simulation as a 'swarm'. 'Flocks' refers to the emergent groups within the swarm.

Each agent is only aware of the agents within a visible radius (neighbourhood) around it. By updating its own state based on the state of its neighbours, large changes can cascade throughout the entire swarm.

Steering Behaviours

Some basic behaviours that cause emergent behaviour are shown below (images are sourced from [2]).

Cohesion	Separation	Alignment
		
The agent turns towards the average position of its neighbours. This tends to cause agents to flock towards a central locus.	The agent turns away from the average position of its neighbours. This tends to cause a flock to cover more area with more dispersed agents.	The agent turns towards the average angle of its neighbours. This tends to cause a flock to travel in one direction.

Computational Complexity

In order to determine the local neighbourhood of an agent, the simulation must traverse all other agents for each individual agent. This places the computational time complexity of the algorithm at $O(n^2)$ – polynomial time.

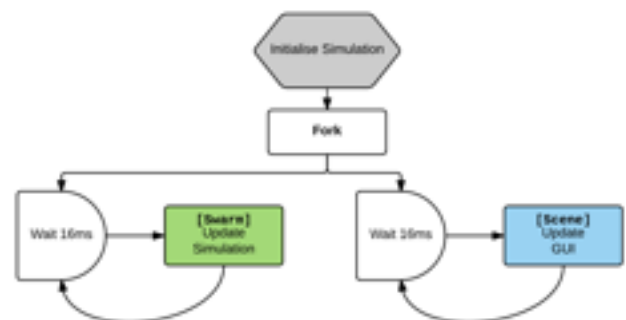
In practical terms this means the number of operations needed to run the simulation increases polynomially with the number of agents. For example: 100 agents would take 10,000 iterations. 1,000 agents it would take 1,000,000.

To achieve a large number of agents in the simulation will therefore require heavy optimisation during implementation, and is likely to require performance profiling.

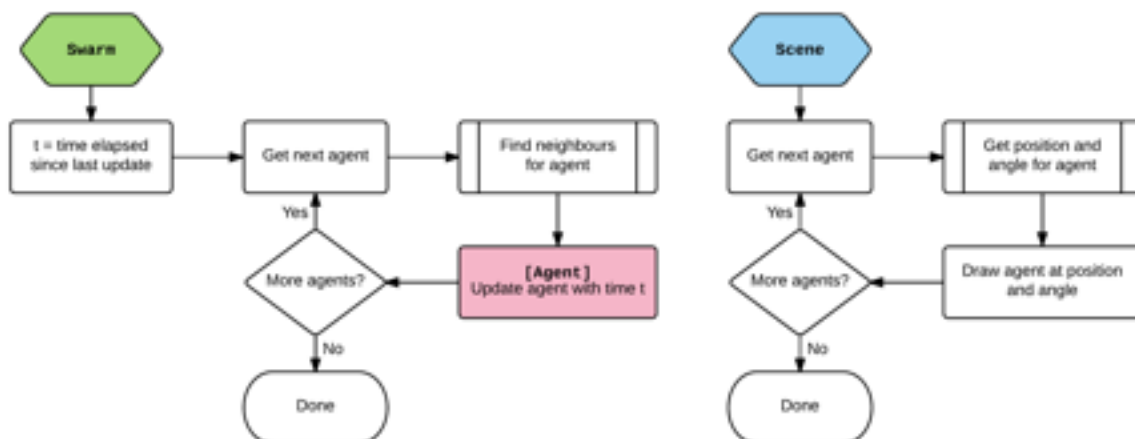
DESIGN

System Overview

The system will operate the simulation and the interface concurrently (see diagram to the right). This is to maintain responsiveness even when the simulation becomes heavy (as it is predicted to be, with thousands of agents). The following diagram illustrates this. 16ms is chosen as it approximates 60 frames per second (most monitors are unlikely to be able to update faster than this, so there's no reasons to go higher).

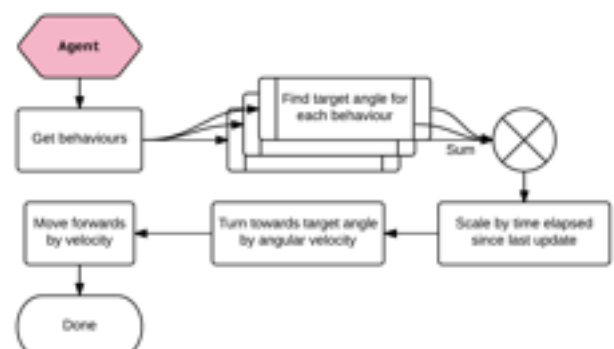


The Swarm and Scene objects are then responsible for their respective tasks (see below).



Each individual agent is updated with the following process, which delegates the determining of a target angle to steering behaviour objects. This design allows for additional behaviours to be added in a straight-forward way, with each behaviour in its own class and separate from the others.

The algorithms for the standard steering behaviours in the design (cohesion, separation, and alignment) are listed in pseudo-code below.



Steering Behaviours

Finding Neighbours

```

for every agent
    for every other agent
        distance := square root of difference between x + difference between y positions
        if distance < view_distance, then these agents are neighbours
  
```

Cohesion

```

for each agent
    neighbours := all other agents within view distance
    centre of mass := sum of neighbours / number of neighbours
    angle to turn to := angle from this agent relative to centre of mass
  
```

Separation

```

for each agent
    neighbours := all other agents within view distance
    centre of mass := sum of neighbours / number of neighbours
    angle to turn to := angle from this agent relative to centre of mass + 180 degrees
  
```

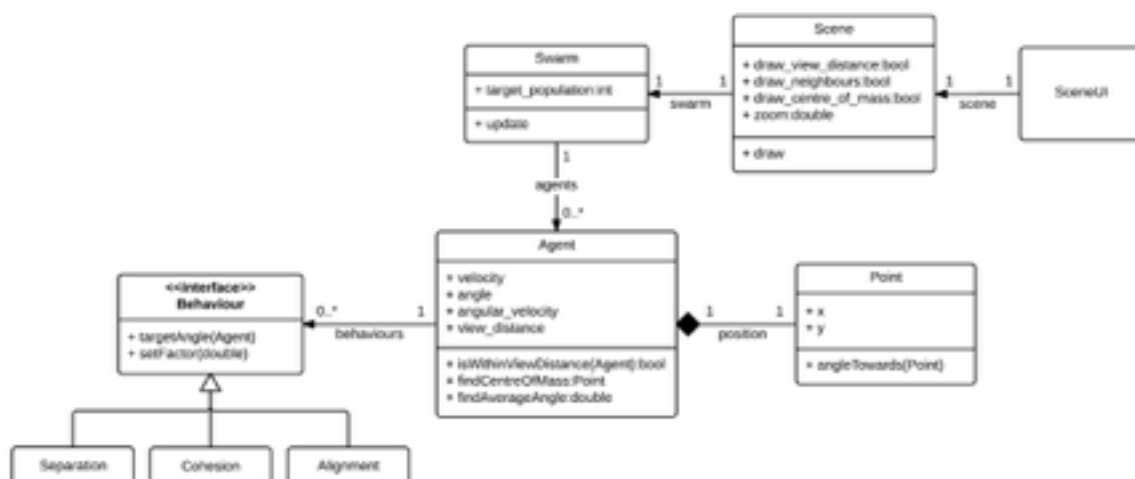
Alignment

```

for each agent
    neighbours := all other agents within view distance
    angle to turn to := sum of neighbours angles / number of neighbours
  
```

UML Class Diagram

Moving towards a class hierarchy, the objects can be structured below. All of the properties and methods relating to the agent are held in the agent itself which handles its own position. Behaviours are given a common interface so that more can be added without specifying their structure up-front. Two additional objects have also been introduced to this structure: *SceneUI*, which the user will interact with to control the parameters for the *Scene*, and *Point*, which represents a current position in the simulation space.

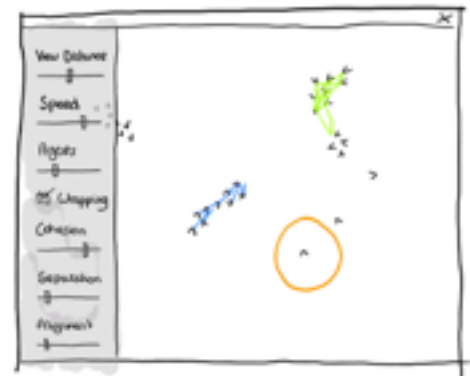


GUI Design

To design the UI I drew mock-ups of the interface. The final design I settled on is pictured on the right.

My aims were to have as much of the window dedicated to the simulation as possible, therefore I am planning to implement a translucent panel to display the controls.

Sliders seem like a natural fit where possible, they are very intuitive for a user to adjust parameters - much more so than entering values into text boxes or clicking buttons to increase/decrease values. Some things which are a binary choice can be represented more simply with a checkbox, so I will utilise both in the design.

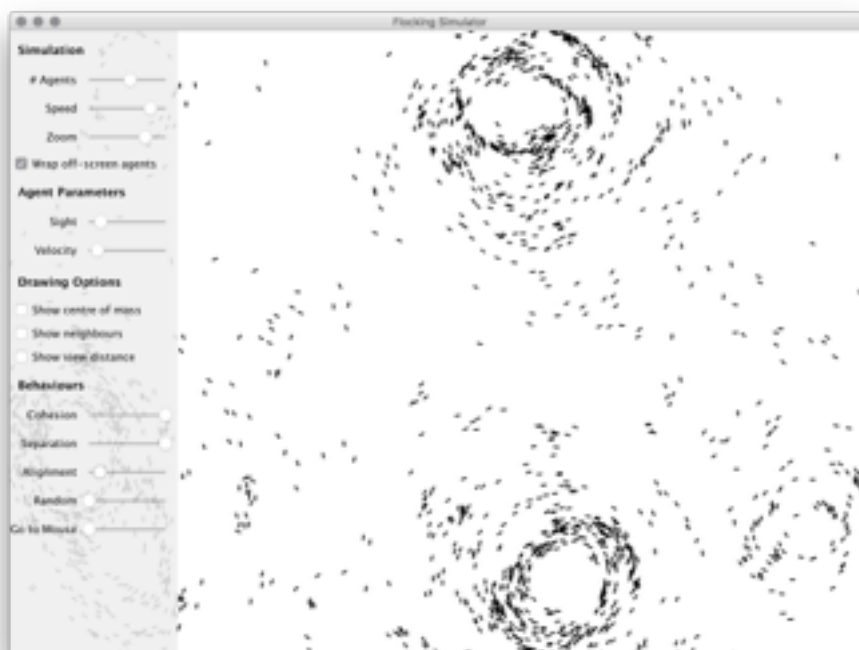


Using a slider for the number of agents allows me to put a limit on the number of agents added - I will do performance testing and then set this limit at where the performance starts to break down.

Features like being able to view view distance (orange circle), centre of mass (blue lines) etc. will help to visualise how the algorithm is working and also to verify it is working correctly.

One additional feature I would like to implement is the ability to zoom in and out of the simulation. At different depths of perspective, different emergent properties become visible, and it would be interesting to be able to see this with my simulation.

IMPLEMENTATION



Pictured: final design of application, in usage

Project Structure

JavaAssessment.java	Execution entry-point for the application - sets up simulation and creates operation threads.
flocking/	
Agent.java	Manages the state of an individual agent
Point.java	For dealing with points and relative positions and angles
Scene.java	A JComponent subclass that renders the actual swarm simulation
SceneUI.java	A JPanel subclass that displays the user interface and forwards changes to parameters to the Scene.
Swarm.java	An ArrayList<Agent> subclass that invokes the simulation of agents it contains, and manages common parameters.
Util.java	Utility class for some angle conversions.
behaviours/	
Alignment.java	Alignment behaviour implementation
Behaviour.java	Provides the interface that steering behaviours must implement
Cohesion.java	Cohesion behaviour implementation
MouseLove.java	This behaviour steers agents towards where the mouse cursor currently is
Randomness.java	This behaviour steers agents in a random direction
ScaledBehaviour.java	Abstract class that implements common functionality for behaviours that scale
Separation.java	Separation behaviour implementation

Changes from Design

When implementing the design there turned out to be many extra considerations that had to be added. Having each agent managing its own properties is important for proper encapsulation of the principles of flocking - that each agent manages itself. However practically this makes it difficult to adjust parameters. Because of this the final implementation has the `Swarm` class manages passing changes to parameters on to all of its constituent agents.

In order to avoid re-implementing functionality in the Java standard library, many of the final classes are sub-classes of standard library classes. `Scene` is a `JComponent`, `SceneUI` is a `JPanel`, `Swarm` is an `ArrayList`, and `Point` is a `Point2D.Double`.

The reason `Swarm` is a subclass of `ArrayList<Agent>` rather than containing one is that many methods involving swarms don't really need to be aware of the particular properties of a collection of agents that involve flocking specifically. It is sufficient for them that it implements the `List<Agent>` interface. Using `ArrayList` provides the implementation for that interface whilst also providing the implementation for much of the behaviour of the swarm.

I ended up implementing additional behaviours (they are outlined in the programme structure above).

I also changed the way agents are displayed. Originally they were planned to be small arrowheads, but I found in testing that when a large number of agents, this rapidly devolves into large black lumps on the simulation where it's not exactly clear what is going on. Therefore I changed the design to something

very reduced, just a small line that indicates the angle an agent is facing. This allows many more agents to be displayed concurrently and still be clear.

Implementing zoom was more complicated than predicted. The zoom behaviour itself is straightforward as it just requires a simple scale and transform when drawing, however as many operations on the swarm depend on the position of the view (e.g. inserting new agents, agent wrap around), I had to implement a circular reference back to the scene from the swarm, and a new method `Scene.getScaledVisibleRegion()`, which can be used to find the effective boundaries the user can see.

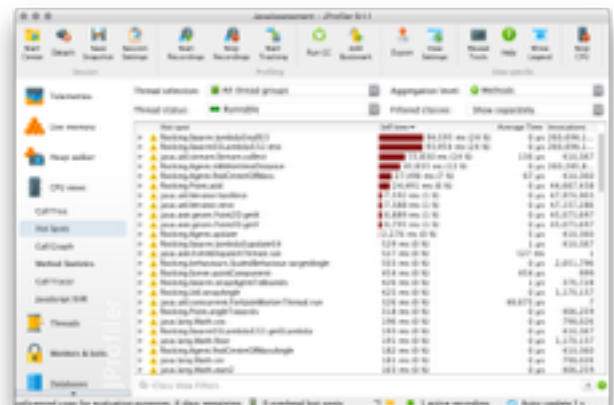
Implementation of User Interface

When implementing the user interface I opted to use lambdas to provide alternatives to anonymous classes for event listeners. This keeps the code much more concise.

To maintain drawing performance at high speeds I removed object initialisation from the draw loop where possible. Disabling anti-aliasing for the additional visual components, such as the centre of mass angle and view distance, also allowed me to keep up the rendering pace.

Optimisation Strategy

As outlined in the analysis section, the application needs to adopt an aggressive optimisation strategy to be able to process a large number of agents. For the implementation of this in particular I used the Java Profiler 'JProfiler' to investigate where the JVM spends most of its execution time.



Multi-Threaded Parallel Execution

The largest gain came from using parallel computation - when the swarm needs to process each agent individually to determine its neighbours, this execution is handed off to Java 8's parallel stream execution implementation (see the implementation in the `Swarm.update()` method). This roughly doubled the amount of agents the application is able to process on a dual-core processor. On processors with more cores this will scale further.

Ignoring Off-screen Agents

A further simple optimisation was to not draw agents that are not currently on the screen.

Rendering Strategy

Using Java's native double buffering implementation (by overriding the `paintComponent` method in a `JComponent` sub-class) gives smooth rendering without screen flickering.

Drawing Accessories

I found during profiling that there was a large bottleneck in the rendering of ovals to draw the agent view distance. I investigated this and found two factors responsible - one was the call to `Graphics2D.drawOval` which will instantiate a new `Ellipse` object in the drawing looping. The other was the use of anti-aliasing which seems to be particularly taxing when drawing arcs. Having a single

Ellipse object to represent all the view distances, and turning off anti-aliasing for this operation, seems to have increased the draw performance significantly.

CONCLUSION

Final Specification of Application

- Stable, high-speed implementation of flocking that supports several thousand simultaneous agents
- Supports cohesion, alignment, separation, as well as randomness and a mouse follow behaviour.
- Allows the simulation to be zoomed out to a very wide macro level view, and scales agents appropriately.
- Optionally allows off-screen agents to wrap around onto the other side of the screen.
- Allows the simulation speed to be slowed down and sped up.
- Allows individual agent velocity and vision to be adjusted.
- Behaviours can be scaled and mixed together.
- Additional drawing accessories such as the centre of mass, neighbouring agents, and view distance display can be enabled.
- Supports a resizable display with a translucent tool panel.

Further Development

I would like to pursue performance even further. To push this more I think a new implementation strategy would have to be considered. A possible alternative approach could be splitting up the simulation into zones where certain zones could be immediately discounted as locations for possible neighbours, therefore cutting down the number of comparisons that need to be done by a large amount.

Another potential optimisation would be to slow down the simulation frame-rate, but leave the display frame-rate the same, and have the scene interpolate the agent's positions based on their previous position, their momentum, and the time since the last update. It should be possible to at least halve the number of updates that need to be done this way, without a noticeable difference to the user.

An additional consideration for flocking could be to allow the radius of each behaviour to be set differently, which could create more interesting interplays between them.

Final Summary

Overall I am quite pleased with the performance I have been able to get out of the implementation, and watching the large number of agents in the simulation is a very satisfying way to pass the time - although sometimes frustratingly distracting when trying to write a report!

BIBLIOGRAPHY

- [1] The University of York, *Digital Electronics, Java Programming Assignment 2015/16 (ELE00005C)*, 2015.
- [2] Craig Reynolds, "Boids", *Background and Update*, September 6, 2001 [Online]. Available: <http://www.red3d.com/cwr/boids/>