

Swarm Intelligence for Forest Fire Boundary Detection

Jumaira Miller

983101

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Bachelor of Science



Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

May 05, 2021

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed  (candidate)

Date

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed  (candidate)

Date

Statement 2

I hereby give my consent for my thesis, if accepted, to be made available for photocopying and inter-library loan, and for the title and summary to be made available to outside organisations.

Signed  (candidate)

Date

*I would like to dedicate this work to nature: our source of shelter, inspiration,
intelligence, spiritual satisfaction, and life.*

In the words of Jules Verne:

“Nature’s creative power is far beyond man’s instinct of destruction.”

Abstract

The devastating ecological, socio-economical, and atmospheric damage of forest fires are widely acknowledged. Over the last decade, the effects of forest fires have only exacerbated by the feedback loop of climate change and forest fires. The need for fast and reliable systems for early forest fire detection has become ever more urgent. Whilst extensive research has been done in fire confirmation, relatively insufficient work has been done to develop efficient fire search algorithms. This gap in research is the area to which our work contributes.

Our paper proposes the use of Swarm Intelligent algorithms, based on the flocking algorithm, to create composite behaviours for effective forest fire boundary search. Using Swarm Intelligence based algorithms for search, we are able to coordinate a team of un-manned aerial vehicles using a decentralised and fully automated system. Two such composite behaviours have been presented in our paper.

Our proposed behaviours have been implemented and tested through simulation using Unity's Game Engine. Based on the criteria of improving search time, and producing precise results, a search behaviour based on thermal sensors resulted in faster and more reliable results than a behaviour which relied on visual sensors. Ways of further optimising forest fire boundary search have also been identified.

Acknowledgements

I would like to thank everyone who played a role in my academic accomplishments. First and foremost, my parents, who have supported me with love, understanding, and encouragement throughout the entirety of my existence. I would especially like to acknowledge Parwin Celia – the sole motivation for all my efforts, my second greatest blessing, and without whom I could not dream to be who or where I am today.

Secondly, I would like to thank my dearest friends of the last three years, and forever more, Chukwuka Ikponmosa Ajeh. I am deeply grateful for your constant patience, love, faith in my ability even through my darkest times, and for being that little duckling that pulled the other duckling along.

Finally, I would like to express my most sincere gratitude towards my supervisor, Sean Walton, for all his time, guidance, and reassurance. During such a chaotic year, under unprecedented circumstances, and even amidst his own responsibilities, he has gone above and beyond to support many students in their education. His passion towards teaching and education, his kindness, and academic ambition will always be admirable.

Table of Contents

Declaration	3
Abstract	7
Acknowledgements	9
Table of Contents	11
Chapter 1 Introduction	14
1.1 Motivation	14
Chapter 2 Background Research	16
2.1 Related Work	16
2.1.1 Evolutionary Trajectory Planner for Multiple UAVs in Realistic Scenarios	17
2.1.2 Swarm Intelligence based Approach for Real Time UAV Team Coordination in Search Operation	18
2.2 Swarm Intelligence	19
2.2.1 Overview	19
2.2.2 Flocking	20
2.2.3 Steering	21
2.2.3.1 Context Behaviours	22
2.2.3.2 Seek	22
2.3 Spatial Optimisation	23
Chapter 3	23
3.1 Core Implementation	24
3.1.1 Basic Flocking Algorithm	24
Alignment	24
Avoidance	25
Cohesion	25
3.1.2 Resolve Flickering Motion of Agents	26
3.1.3 Composite Behaviour	26
3.1.4 Filtering Neighbouring Objects	27
3.1.5 Heat Track Behaviour	29
3.1.6 Sight Track Behaviour	30
3.1.6.1 Problem: Simulating Line of Sight	30

3.1.7 Different Formations	31
3.1.7.1 Circle Formation	31
Problem: Overlap on Flames	31
3.1.7.2 Line Formation	32
3.1.8 Stop at Distance and Timer	32
3.2 Results	33
3.2.1 Scenario 1	34
3.2.1.1 Description	34
3.2.1.2 Observations	35
3.2.2 Scenario 2	37
3.2.2.1 Description	37
3.2.2.2 Observations	37
3.2.3 Scenario 3	38
3.2.3.1 Description	38
3.2.3.2 Observations	39
3.2.4 Other Interesting Behavioural Observations	40
Lemniscate Bug	40
Balancing Flock Size and Avoidance Behaviour	40
Chapter 4 Conclusions and Future Work	41
4.1 Conclusion	41
4.2 Evaluation	41
4.2.1 Technical Reflection	42
4.2.2 Personal Reflection	42
4.2.2.1 Change to Project Scope	42
4.3 Future Work	43
Bibliography	45
Appendix A Code Implementation	49
Appendix B Supplementary Data	55

Chapter 1

Introduction

The devastating ecological, socio-economical, and atmospheric damage of forest fires are widely acknowledged. A recent example of ecological damages includes the near extinction of many endangered species as a result of Australia's 2019-2020 bushfire [1] [2]. This particular bushfire season is recorded as one of the country's costliest to date, leaving over 5000 homes destroyed across multiple fire zones. Apart from the displacement of civilians, other socio-economic effects include telecommunication networks being cut, blocking of roads and transportation, and civilian casualties [3].

Although it is widely acknowledged that forest fires have a dire impact on climate change, however, what is often overlooked is the insidious relationship between the two. There is an underlying feedback loop as climate change also affects the propensity for forest fires which, as a result, exacerbates the threats with each cycle. Increased emissions of harmful gases from forest fires results in rapid global temperature rise, which in turn dries soil and vegetation. Drier environmental conditions and drought accommodate more frequent and intense fires across prolonged wildfire seasons. With a quarter of the [4] [5]year 2020 remaining, the frequency of forest fires had already escalated by a staggering 13% when compared to the entirety of the previous year. An increase in fire intensity is evident in the multitude of countries and states – such as the Arctic Circle, Australia, and the state of California – reporting record fires this year.

To reduce the effects of the aforementioned concerns, many countries worldwide heavily invest in forest fire detection systems to successfully suppress forest fires whilst they are still manageable [6]. Most methods of forest fire detection rely on satellites and airborne systems [6] [7] [8] rather than other tools and technologies – some of which will be briefly highlighted in the *Background Research* Section.

As Sherstjuk et al. identify, the process of forest fire detection can be split into the two sub-tasks of *fire search* and *fire confirmation* [8]. Fire search refers to the surveillance of a forest region, and fire confirmation refers to the actual process of determining whether a fire exists within the segment of the area currently under surveillance.

Whilst extensive research has been done in fire confirmation, relatively insufficient work has been done to develop efficient fire search algorithms. This gap in research is the area to which our work contributes.

1.1 Motivation

This practical urgency to address and mitigate the impact of forest fires is the first motivation for this project. As Allison et al. highlights, one of the persisting challenges - even amongst the technological advancements in the hardware of UAVs - is the need for efficient detection algorithms [6]. The urgency demands fast and reliable detection.

This urgency is also what encouraged and motivated the use of drones over satellites as the main technology for our research. There are two main satellites, the Moderate Resolution Imaging Spectroradiometer (MODIS) and the Advanced Very High-Resolution Radiometer (AVHRR), launched specifically for forest fire detection. However, whilst UAVs are of sending signals simultaneously which can be received by users in real-time, MODIS and AVHRR receives data every 48 hours. The infrequent reception of data cannot accommodate the goal of fast detection. On top of this, geostationary (GEO) and Low Earth Orbit (LEO) satellites cannot compare with UAVs in terms of early fire detection. This is because GEO and LEO satellites are further away from land than UAVs, and because the intensity of the physical parameters of fires decrease with distance, satellites are unable to detect fires in stages as early as UAVs could. UAVs fly at a lower altitude when compared with traditional imaging techniques using satellites for fire detection, which proves a higher quality of imaging platform which can more accurately distinguish between different light sources and wavelengths [6]. To further add to our motivation, this field of research surrounding the development of designing algorithms for coordinating teams of UAVs “is still in its infancy” [9]. Our aim is to contribute to this exciting gap in research.

A field of research which has long contributed to the advancement of wildfire science and management, specifically fire detection and mapping, is Machine Learning (ML) [10]. There are two main approaches in this field, data-driven and model-driven, both of which have their associated limitations. Traditional model-based methods limit the problems that may be solved using ML to those that can be decomposed into either continuous or differentiable functions. In order to do so, a deep understanding of the problem is required. However, this may not always be possible. Using Swarm Intelligence (SI) can be a solution to this limitation to ML approaches as SI does not require that we have prior knowledge. SI is a type of meta-heuristic algorithm which require little to no prior assumptions. This means they are able to handle high-dimensional and dynamic problems for which it is either difficult or not possible to obtain derivatives, including discreet, non-continuous, or noisy functions [11]. The potential solution to more-commonly used methods of forest fire detection is a what motivates our interest in developing SI algorithms for this problem. Further to this, SI solutions to search will result in a system of decentralised, self-organised, locally communicating agents. As Haksar and Schwager have indicated, again there exists a gap in research in this area where we lack methods to control forest fires with autonomous agents. The few published methods that do exist focus on centralised control [12]. Contributing to a niche gap in research further adds to the importance of our work.

Finally, our interest in using Unity’s game engine arises from the recent rise in advancements in computer vision, processor power, and computer parallelisation. These recent developments allow us to overcome previous barriers in researching solutions for fire front boundary search using teams of UAVs. Some barriers in research include the difficulty in testing systems in real-world scenarios, testing with large teams of UAVs and their sensors may be expensive even if their individual UAVs have become significantly more affordable [6], and limitations to how extensive experimentations can be in scaled-up scenarios. Two such examples of these mentioned limitations can be seen in the limited number of UAVs, four and twenty respectively, used in evaluating the systems proposed by Sherstjuk et al. [8] and Harikumar et al. [13]. Using a game engine to run simulations for testing and evaluating proposed systems evades the limitations of real-world simulations.

Following the above motivations, the main contribution and aim of our paper is to create an effective automated rule-based algorithm, using swarm intelligence, for efficient forest fire boundary search. The algorithm must be able to coordinate a *team of UAV* (often alternatively

referred to as *drones*, *flock*, or *fleet* within this paper) to outline the fire front of an active forest fire. In order to achieve this aim, the goals of this project are as follows:

- Create an effective tracking behaviour to combine into a composite behaviour which can identify the boundary of a forest fire.
- Find the optimal flock size for given forest fire formations
- Find the optimal hyperparameters (which will be defined within the *Methodology* Section) to optimise the portion of the forest fire front being found

Chapter 2

Background Research

Besides swarm intelligence and drones, forest fire detection can be done using various other approaches and technologies. Some commonly used operational forest fire detection tools and technologies include watch towers or fire towers, optical smoke detection, infrared, spotter planes, mobile/smart phones, satellites, and wireless Sensor Networks (WSN) [14]. A multitude of studies also alternatively developed image processing algorithms, evolutionary algorithms (EAs), and some researchers have also published very novel approaches using radio-acoustic sounding systems to detect early forest fires [15, 16]. The motivation for using image processing lies on the rapid advancements of digital cameras and sensors. Advancements in image processing allows for higher image quality at lower costs, ability to cover larger areas with higher accuracy, and the costs to employ processing systems is lower than already existing systems.

2.1 Related Work

Kinaneva et al. proposes a method more closely tied to our approach, compared to the multitude of other technologies and algorithms mentioned above, in the sense that both our studies involve the use of a multi-UAV system using artificial intelligence. Their paper proposes the use of two drones, at two different altitudes, in order to reduce the false positive results. Results show that the use of multiple drones in this way had improved the precision and accuracy of the search and detection. A similarity between their proposed solution and ours is that their system is also fully automated with decentralised control. However, where our project has two distinct sensors for thermal and optical sensors, this paper detects fire based on a single sensor - thermal cameras - which are sensitive to infrared radiation. Our project aims to differentiate the effectiveness of these two sensors individually rather than considering a single sensor which considers both thermal and visual data [17]. This will allow us to compare which sensor is superior for faster and more reliable search times.

Sherstjuk et al. identifies that indeed fire detection can be split into two sub-tasks: *fire search* and *fire confirmation* [8]. Both are addressed within this work. For fire search, their paper suggests a fixed formation throughout the search, thus their drones are hard coded to have a pre-planned path. This is different from our project's fire search as ours, instead of a pre-planned path, relies on the emergent behaviour of compositions of simple behaviours. Thus,

although our system of drones will not have a fixed formation, we enforce fixed spawn formations but the formation through the search mission will have an emergent behaviour instead of a planned path. Our paper will propose two such composite behaviours and evaluate their efficiency and performance (i.e., where this paper proposes a pre-planned path, our paper proposes a pre-planned behaviour for the UAV's). Similar to our system, the number of drones used by Sherstjuk et al.'s system for search may vary based on the size of the surveillance region. Another difference between this project and ours is that ours is tested entirely in simulation instead of being tested in the real world with real hardware. This is a result of constraints of our work such as time, hardware skill, finance, and health and safety precautions in terms of test set-ups. However, an advantage gained through simulation is the ability to use a large number of UAV's instead of being limited (by example of this this paper) to three [8].

2.1.1 Evolutionary Trajectory Planner for Multiple UAVs in Realistic Scenarios

Besada-Portas et al. proposes a path planner for UAVs based on EAs, which measures the fitness of each solution of the whole population [5]. These algorithms deduce the combination of elements that maximises system fidelity with evaluation of fitness functions [18].

The first difference is that our model will be a lot more simplified than the model presented in this paper. Where this paper presents/incorporates a complex mathematical model which considers physical properties of UAVs, radars and missiles to represent a realistic path planner for military UAVs. However, this paper proposes a similar approach to ours where movement of UAVs are based on certain rules as well as defining no-flying zones for military reasons (minimising risk of damaging UAV and sensors). An example of a rule which is similar to behaviours in our system is ensuring the UAVs do not collide (similar to our *avoidanceBehaviour*) or get too far from the search space (which is the functionality provided by our *keepWithinRadius*). Although this paper defines no-flying zones to prevent damaging the UAVs, our system enforces a safe stopping distance for this purpose.

A similarity between the system proposed in this paper and our proposed system is that they both execute extensive tests through simulation.

Optimisation using cartesian mathematical models and use of cubic-spline curves, results are statistically analysed to reach the convergence and deducting optimal region of the space.

This allows for greater flexibility and resources for testing. If the tests were executed using real technology, it would have been far more costly, and we would have been limited to less extensive and extreme tests. However, the nature of our simulations is entirely different. Whilst their paper took a more mathematical and statistical approach, which lack the ability for real-time visualisation of the simulations, our system makes use of Unity's game engine which result in similarly near-real-world physics as well provide real-time visualisation of simulation tests.

The most prominent difference to note is the method used to create path finding algorithms. This paper presents a model using EA, where the team of UAVs share their optimal parameters, indexes, and constraints with each other to coordinate the evolution of EAs.

I will be using swarm intelligence (SI). Whilst in EAs the optimal result for one agent directly affects the paths of other agents, the decentralised nature of SI algorithms ensures the goals

are achieved regardless of failure of some agents. In the case of EAs, these failures can be undetected if only statistics are considered.

2.1.2 Swarm Intelligence based Approach for Real Time UAV Team Coordination in Search Operation

Gervasio et al. have proposed a system that is the closest to that of ours in terms of our environmental goals as well as them using SI systems for search. This paper proposes a three-stage operation in the given order: initialise optimal starting positions for exploration, simultaneous search and broadcast data to other agents, seek source of pollution once found. A flow diagram of their proposed system's behaviour and its three stages of operation can be found in Section III of their paper [9]. The first stage of finding an optimal starting location consists of maximising the minimum distance between other UAVs in the team whilst also keeping within a given search radius. Our project has taken from this and has implemented a behaviour, *keepWithinRadius*, to keep agents from going off screen (in terms of simulation) and going beyond the effected woodland range. We also ensure UAVs do not collide, however, instead of maximising distance between them, we focus on surrounding drones not invading a minimum radial space around each agent. Finally, instead of searching and spreading out (to optimise exploration time) prior to actually sensing for targets, we have instead investigated two spawn formations for the team of UAVs.

Another difference between their proposed system and our is that our model is a lot more simplified than the model presented in this paper. This paper's model considers movement constraints of airborne objects, such as disabling the system from performing an instantaneous 180 degree turn as this is not possible for a real plane. This paper also utilises sensor fusion to analyse information regarding temperature, humidity and sensors for fire pollutant chemicals. They also had the luxury of being able to test their model not only in their simulation but also confirm results in the real-world.

In contrast, our model only considered visual and thermal sensors for the drones, and the simulation model's environment simply has dots representing burning areas or burnt regions. This high level of abstraction of our model is a result of constraints of time and resources of our project.

This paper's approach is similar to our work in a way that both systems use Game Engines for simulations. Where this paper uses JAVA and Jmonkey 3 OpenGL Game Engine (jME) for its simulation, our project opted to use the popular Unity game engine. Unity allows rapid prototyping and an effortless exporting of projects on more than 20 platforms where jME can only work with (Win/Mac/Linux/Android). Unity forums are also much more active than jME which consists of a smaller community. Finally, Unity runs seamlessly even in systems that are considered "weak" by today's standards [19]. Although jME may enable more limitless implementation, it is much more complex and developer orientated, making it difficult for other researchers to develop on existing work unless they have sufficient knowledge in using the tool [20]. The aforementioned benefits of Unity allow for our work to be more accessible by a wider community for further development and contributions. Furthermore, we have made an effort to make our work generalisable and modular, in efforts of making it easy to augment additional behaviours and compose composite behaviours.

2.2 Swarm Intelligence

The disciplines of Mathematics and the Sciences have long adopted algorithmic solutions and fields of studies from observing, modelling, and analysing nature and its natural phenomena. One such discipline, in the study of 'artificial life', was inspired by the collective behaviours of social insects and high order living animals. Examples of such animals include ants, a flock of birds, schools of fish, and bees. The collective behaviour of such animals is called '*swarm behaviour*' [21, 22, 23]

2.2.1 Overview

Swarm behaviour refers to the aggregation motion of a collection of decentralised, self-organising individual systems to accomplish a given task [22]. From modelling and analysing such behaviours, scientists have found that a collection of simple individuals, using simple local communication rules, can achieve goals which an individual may not have been capable of or achieve goals far more efficiently.

Swarm Intelligence (SI) is the application of the observed *robustness*, *scalability*, and *distributed self- organisation principles*, observed in natural swarm systems, to artificial systems [23].

Aligning with that of natural swarm systems, SI systems are characterised as follows [21]:

- Extensive collections of simple individual systems.
- Individuals must be near homogenous (not much, if any, variety in the type of individual systems).
- Individuals' actions use a limited number of simple behavioural rules to locally interact with other individuals and the surrounding environment.
- Self-organisation of the overall SI system is achieved as a result of the collective actions of the individual systems

In SI systems, the collective group of individual systems is considered the primary system. Thus, the individual systems themselves are the subsystems. The central concept behind SI systems (as a group of simple autonomous systems with limited capabilities) is that the collective knowledge of the system can obtain insights which can be used to solve complex problems (both continuous and discrete non-linear problems) which the simple subsets of the collective system would not have been able to solve themselves [24].

As a result of the characteristics specified above, SI systems have the following properties [23, 21]:

- **Scalability** The systems can introduce more subsystems and maintain all original functionalities without the need to redefine the interaction rules between the subsystems. In other words, increasing the overall size of the system will improve performance. However, reducing the number of subsystems within the SI system will not affect the functionality of the system.
- **Robustness** As a result of the scalability property and the way the subsystems interact, the system is tolerant to failure. The failure of a subsystem does not affect the functionality of the overall system; the failing subsystem can be removed or replaced with another one.
- **Distributed Coordination** Each subsystem is restricted to local and simple sensing and communication abilities. The parallel actions (every subsystem simultaneously performing different actions at different locations) of the subsystems combined with the lack of a ‘leading’ subsystem or a central control system gives the SI system this property of having distributed Coordination

Evaluating the performance of SI systems have proven to be a challenge in the study of SI. Due to a lack of analytic studies in this discipline, a general criterion on evaluating the performance SI systems is yet to be established. Instead, researchers and developers have taken to evaluate the performance based on other properties. An example of an alternative is assessing the flexibility of such systems, which is essentially a robustness property [22].

SI has been applied to a variety of different problems, including optimisation, communication networks, and robotics. Some prominent swarm-based algorithms include Ant Colony Optimization Algorithms (ACO), Particle Swarm Optimization Algorithms (PSO), and Artificial Fish Swarm Algorithm (AFSA) [25].

2.2.2 Flocking

The flocking algorithm is another example of a swarm-based algorithm and it was made to simulate the behaviour of birds. Our final system is based on this algorithm.

A bird-like object (boid) is defined by its position and a velocity vector. The overall flocking behaviour is the result of a set of simple behaviours applied to a set of boids such that, with each timestep, each boid moves by a predetermined distance along the velocity vector. The overall effect is that the boids are able to move sensibly in a group. Patterns begin to emerge, making the system of boids appear to be coordinated by some common controller – similar to the behaviour of a flock of birds. However, in practice, each boid makes decisions independently by observing its immediate neighbouring boids (without directly communicating with them) and changes its position and velocity vector based on the result of applying the set of behaviours on the given boid in relation to its current surroundings [26, 27, 28].

The three key simple behaviours composing the overall flocking behaviour are listed below:

- o **Separation:** All boids are expected to avoid collisions with each other and their environment. After an interval of time, all boids will check their personal spaces. Should any boid fall within this personal circle

(smaller than the local peripheral grey space) the affected boid will move away.

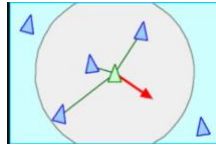


Figure 1: Avoidance or Separation [26]

o Alignment:

This behaviour rule ensures the boids steer towards the collective direction. At each timestep the boids will re-assess and readjust their alignment based on the average alignment of the boids within their vicinity.

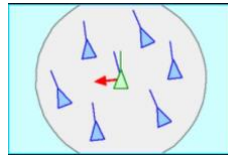


Figure 2: Alignment [26]

o Cohesion:

In this behaviour, the boids move as a group. All boids follow alongside their neighbours. At each timestep, a component, ΔV , is added towards the centre of gravity for boids in the local vicinity.

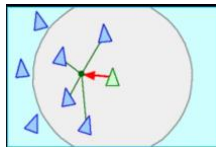


Figure 3: Cohesion [26]

Like many other swarm intelligent algorithms, an interesting side effect of the flocking algorithm is that the behaviour of the flock can seem random or unpredictable as a whole. In actuality, every movement decision is completely calculated and predictable, but too many decisions are made in each timeframe for a human to predict the overall motion of the flock.

2.2.3 Steering

Steering behaviours, such as cohesion, avoidance, and alignment, are algorithms which use local information, such as neighbouring objects' forces, to compute the movement of an individual agent. The 'movement' is the force applied to an agent.

In order to expand on the basic flock algorithm and design a more specific composite behaviour for our system, other steering behaviours had been considered and some had been investigated. Some examples of steering behaviours include flee, pursue, evade, arrival, and containment. For further reading on other steering behaviours, the reader is referred to Shiffman's book chapter [28]. Within our work, we have considered the idea of context behaviours, seek behaviour, and flow field following behaviour.

2.2.3.1 Context Behaviours

Context behaviours are a solution to a limitation of steering behaviours. For steering behaviours, the different behaviours cannot ever be merged with integrity (i.e., for example, a few agents in a flock may occasionally collide) because steering behaviours only return a direction and magnitude [29]. Low integrity in systems is especially introduced when steering behaviours such as avoidance and collision are introduced – which is the case with our system. This may not be very noticeable in large simulation systems, however, in the real-world such an occurrence would be very costly as we are dealing with potentially expensive technology (UAVs and their specialised sensors).

Context behaviours improve steering behaviours by aiming to give our boids/UAVs enough information about the wider simulation environment to be able to make more informed decisions on how to move. This is done by using two context Concept Maps, one for ‘interests’ and one for ‘danger’, which aid to improve the accuracy of making decisions on a combined move for an array of individual behaviours. For a deeper understanding on the topic, the reader is redirected to Fray’s blog post [29], or Keats’ blog post [30].

Although our system did not use context behaviours, a solution for our stopping condition was inspired by its concept. Instead of using Context Maps to make our behaviours aware of one another, we use a Boolean flag within the abstract *FlockBehaviour* class. This allows individual behaviours to change their flag if they are to impose the system to stop (upon meeting a stopping criteria). The *CompositeBehaviour* class simulates the idea of the behaviours being aware of one another by iterating through each behaviour and causing the system to stop if even a single behaviour has updated their flag to ‘True’. Our solution, inspired by the idea of context behaviours, is further discussed within section 3.1.8.

2.2.3.2 Seek

Seek behaviour intelligently steers an agent towards a given target with respect to its current position, vector, and surrounding environment [28]. An agent is required to consider the offset between its current vector and its target vector, as well as consider how to adjust its current velocity, before applying the desired force towards its target. This behaviour has been incorporated within the two tracking behaviours (*HeatTrackBehaviour* and *SightTrackBehaviour*) which are used to compose two distinct composite behaviours.

As later discussed, in sections 3.1.5 and 3.1.6, our implementation of seek behaviour did not use Unity’s built-in method *Transform.LookAt*. Although this may have been viable to use, we decided against it as it automated the behaviour, which we intended to define ourselves. Thus, we instead manually computed the offset vector. Furthermore, *Transform.LookAt* would not work unless the direction of the target is perpendicular to the agents current heading [31]. This is an issue because we have made the assumption of having and wished to simulate 360 rotation cameras which would have the ability to rotate a UAVs current heading towards a target at any offset direction.

2.3 Spatial Optimisation

Spatial optimisation is the use of mathematical and computation techniques to find the best solutions to geographic decision problems using strictly defined parameters. It is a sub-field under the broader field of optimisation [32].

Optimisation problems consist of objective functions, decision variables and other constraints. Spatial optimisation focuses on the representation of space as an essential part in determining the decision variables and objective functions when solving the problem. The functions themselves will tend to represent physical relations such as shape, distance or whether two or more areas overlap. The general form of spatial optimisation is outlined below:

Given a set of geographic decision variables where $x \in X$ represents the location:

$$\text{Optimise}(f(x))$$

With n constraints in the form:

$$\begin{aligned} g_n(x) &\leq C_n \\ g_n(x) &\geq C_n \\ x &\geq d \end{aligned}$$

Where C_n is the limit value for the n th condition.

In spatial optimisation, the decision variables represent geographical locations on a grid (x, y) and contain any number of dimensions concerning the problem at hand.

The challenge is in defining these decision variables. Depending on the problem, it may require that the distance be spread in a uniform way to maximise coverage. Alternatively, clustering may be required. Each requirement comes with its complexity. Using clustering as an example, I would need to address issues of cells being adjacent or connected. This would, of course, require algorithms such as nearest neighbour [32].

For our project, we had considered using spatial optimisation techniques to optimise the spawning location for our swarm of UAVs, in hopes that it may reduce the average search time. However, considering the constraints of time on our project, and to reduce the number of hyperparameters for simplicity (as testing has already become very complex), we instead decided to fix spawn location to roughly the centre of the game scene.

Chapter 3

Methodology

Our proposed system will be based on the basic flocking algorithm. As the flocking algorithm is not the main focus of our work, we have taken much of it from an existing algorithm was publicly available on GitHub [33]. Our system can be found at the following link:

After establishing the base system, which was implemented modularly for ease of functionality and augmentation, we implemented two tracking behaviours, *HeatTrackBehaviour* (which we will refer to as *HeatTrack* in short) and *SightTrackBehaviour* (often referred as *SightTrack* throughout our paper). Additional behaviours, such as *KeepWithinRadius* and *SteeredCohesionBehaviour*, were also implemented to enhance the model and improve the overall composite behaviours.

The two main composite behaviours investigated in our paper are *Visual Search Behaviour* and *Seek Heat Source Behaviour*. Both composite behaviours are a combination of *AvoidanceBehaviour*, *AlignmentBehaviour*, *Steered Cohesion Behaviour*, and *KeepWithinR15* (which is an object created from the *KeepWithinRadiusBehaviour* with the radius set to 15 units), where the difference between the two composite behaviours is the tracking behaviour being added to these four other behaviours. The composition of these four behaviours and *SightTrack* creates *Visual Search Behaviour*, and the latter of the two composites is the product of the four behaviours and *HeatTrack*.

A series of test cases were created to investigate and compare the effectiveness of each of our composite behaviours. Our evaluation of the systems considers the following three criteria:

- Time Our main objective is to identify the boundary at the shortest time, in seconds, to achieve our overall goal of contributing to early forest fire detection.
- Completeness Completeness refers to the ability of our composite behaviours to coordinate a team (of a given size) of UAVs to find the entire boundary of a forest fire front and not just a certain segment of it
- Precision We are concerned with how reliable our search times are, rather than the accuracy because our system is implemented in such a way that the wrong target is never identified. The accuracy of our system always identifying the burning regions is ensured by the use of Unity's LayerMask.

3.1 Core Implementation

3.1.1 Basic Flocking Algorithm

In a basic flock algorithm, the emergent coordination is a result of the combination of three behaviours: cohesion, alignment, and avoidance. Each behaviour takes into account all the neighbouring agents within a fixed neighbouring radius. Thus, the neighbouring agents refer to other agents within a given radius from a particular individual agent.

Alignment

Alignment encourages the agents to move in a common direction. Each agent will set its heading to the average heading of its neighbouring agents. Refer to Appendix A Figure A.1 for the following description of the implementation. To implement the alignment behaviour,

the overridden *CalculateMove* method first checks for the case where there are no neighbouring agents, in which case the given agent's current direction will remain the same. The benefit of using *agent.transform.up* is that the returned vector also has a magnitude of one, allowing for a forward motion even in the case of having no neighbouring agents. *item.transform.up* needed to be cast to a *Vector2* to resolve the error of “Operator ‘+=’ is ambiguous on operands of type ‘Vector2’ and ‘Vector3’” because *transform.up* returns a *Vector3* as (0,1,0). Similar steps were taken to resolve a similar error on ‘=’ when computing the offset position from the agent's current heading. Another benefit of using *transform.up* is that it returns normalised values. This means the computed average is also a normalised value. In this case, the alignment is independent of the current position or alignment of the given agent, as such, the offset is not required.

Avoidance

Avoidance, or also referred to as ‘Separation’, helps avoid collision and overlapping of agents. Agents will look at neighbours within a region that is more restricted than the neighbouring area but larger than the agent itself – which it deems to be its personal space – and navigate away from any agents within this space. It can be seen as a behaviour that is more restrictive and opposing the cohesion behaviour. Refer to Appendix A Figure A.2 for the following description of the implementation. To implement this behaviour, we needed to consider the agents within a drone's neighbouring surroundings as well as keep a record of the number of agents within an agent's avoidance radius (which is a more constrained region around the agent than the neighbouring radius). A counter is created outside the foreach loop to keep these counts. An if-statement within the foreach loop checks whether the square distance between a game object and the agent is less than the square of the avoidance radius because this would imply the game object is too close to the agent, in which case we add the offset between the agent and the game object to *avoidanceMove* and increment the counter. If there are no neighbouring agents, the current agent's move will not be adjusted, otherwise the returned move will equal to the average offset distance between the agent and its neighbouring agents within its *personalRadius*.

Cohesion

Cohesion compels each agent in the flock to stay in groups with its neighbours. The agent does this by finding the midpoint between all its neighbours and navigating towards it. Cohesion behaviour typically needs to be mitigated with some steering or smoothing in order to avoid extreme changes in direction. Refer to Figure A.3 in Appendix A for the code for cohesion behaviour. Cohesion behaviour is very similar to the avoidance behaviour script but reversed in functionality – cohesion behaviour's *CalculateMove* method first checks the number of transforms within its neighbouring region. If there is no object surrounding the agent, then no adjustment is made – this translates to returning a vector with no magnitude. Otherwise, the position of each neighbouring agent is summed and then averaged. As with the other behaviour scripts, *transform.position* returns a *Vector3* and thus, to resolve issues with ambiguity, it was cast to a *Vector2* before adding it to the current sum of positions (i.e. *cohesionMove*).

An issue with this implementation of cohesion behaviour, when combined with avoidance and alignment behaviours to create the basic flocking behaviour, is that it causes an unwanted flickering movement of the drones within the team. As the individual drones attempt to align with their neighbours, the cohesion behaviour causes them to immediately correct their

positions if any of the neighbouring agents are too close. An alternative version of the cohesion behaviour can smooth out this flickering motion

3.1.2 Resolve Flickering Motion of Agents

The flickering motion was easy to resolve and only took three additional lines, however, a new script was created for this fixed version of cohesion behaviour. The script with the adjustments is called *SteeredCohesionBehaviour* and it can be found with all the Behaviour Scripts in the link directing the user to our developed system's GitHub. The reader may also wish to refer to Figure A.4 in Appendix A for quick access to view the code.

Our solution makes use of Unity's provided *Mathf.SmoothDamp* method. In order to use this method, we created two variables, one which is a utility variable, another is a public variable which can be set from Unity's inspector window to adjust the steering. A placeholder is created on line 9 in Figure A.4 for a variable required by the *SmoothDamp* method. The second variable on the next line is a floating value stored in *agentSmoothTime*. This variable determines how long it should take for an agent to get from its current state to the calculated state. We've set the value of *agentSmoothTime* to half a second (as seen on line 10 in Appendix A Figure A.4), however, as *SmoothDamp* occurs at every single frame so it may not ever actually run for a full half second. Thus, *agentSmoothTime* is just a value that can be adjusted depending on how quick we want the fix (of the flickering motion) to be; A higher value will increase the flowing movement of the drones, whereas a low value would still have some flickering motion to the movement of the individual drones.

A single line of code is added within the *CalculateMove* method immediately after offsetting the original position of each drone as seen on line 27 (Figure A.4). Once again, on line 28 of Figure A.4, *cohesionMove* is set – this time it is directly set to *Vector2.SmoothDamp*. *SmoothDamp* takes in several parameters, the first of which is the current vector to the current velocity – we retrieve this as *agent.transform.up*. Another parameter is the target of where we want the drone to move to – this is set as the calculated *cohesionMove*. The placeholder variable from line 10, *currentVelocity*, is passed in as a reference ('*ref*' in code) for the third parameter – passing the variable as a *ref* updates and stores the velocity of the drone at each timeframe in the *currentVelocity* variable. Finally, *agentSmoothTime* is passed in as the final parameter to determine the time each drone takes to get to the destination vector [34].

3.1.3 Composite Behaviour

The composite behaviour script allows for the combination of multiple behaviour objects (created using the simple behaviour scripts) with weights to control the overall movement of each individual agent within the flock or team of UAVs.

Like the simple behaviour scripts, this script also inherits from *FlockBehaviour* and provides an implementation for and overloads the *CalculateMove* function. This class uses two arrays: an array of *FlockBehaviours* which are to be combined together, and another is an array of floating-point values for weights which uniquely correspond to each individual behaviour. The arrays are made to be public so we can manually assign to them rather than initialise them upon definition, making the system more generalisable for any composite behaviour. Within the *CalculateMove* method, we ensure that the two arrays have the same amount of information

- i.e., the arrays are of the same length, otherwise we risk running into errors when calculating the composite move. These cases of data mismatch were handled using a simple if-statement to return an error message with the name of the scriptable object (i.e., the composite behaviour which does not have a corresponding weight value for each simple behaviour object) and highlight the object (using the 'this' keyword) to indicate where the problem is so it can be fixed. In addition to throwing an error message, if the number of weights and simple behaviours do not match, we also stop the flock from moving by returning a vector of zero.

The *calculateMove* method of the *CompositeBehaviour* script will act as a middleman to pass the relevant parameters to the behaviours that actually calculate the movements. Lines 24-27 in Figure A.5 of Appendix A will be discussed further down in section 3.1.8. It is to account for the stopping condition.

To calculate the overall move of the composite behaviour, we iterate through each of the behaviours and multiply their returned moves with their corresponding weight. A for-loop was used instead of a foreach loop to ensure that the same index is used for the behaviours and the weights. The partial move, which is the vector of a particular behaviour contributing to the overall composite move, is limited to not exceed the weight of the given simple behaviour. To implement this bounding (referring to lines 31-39 of Figure A.5) the partial move is normalised to 1 and multiplied by the weight so the partial move is capped at the value of the weight as the maximum.

3.1.4 Filtering Neighbouring Objects

Before expanding on the basic flock functionality (which only consists of alignment, avoidance, and cohesion) to give the UAVs in the flock the ability to track specific objects in its surroundings, we needed to first create a functionality which allows the agents to filter their neighbouring objects. We used Unity's physics layers, and *LayerMasks* to determine the type of object a transform represents. *LayerMask* is a convenient way to access the physics layers in Unity's inspector window [35].

To add the filtering functionality to a behaviour script, we need to apply a surroundings filter which will process the list of neighbouring transforms around a particular agent and determine which to keep and which to discard based on some established criteria. Thus, these surroundings filters were created as scriptable objects which can be applied by any behaviour script. For this functionality we created a new C# script, called *SurroundingsFilter*, which is an abstract class and is a *ScriptableObject*, as seen in Figure 4. This class contains a single public method, called *Filter*, which takes in a flock agent (so the method can compare this agent to other objects within the list of transforms) and the original list of surrounding transforms as its parameters. *Filter* will then return a new list of filtered surrounding transforms. Next, we created our filter script which will provide an implementation for this abstract *Filter* method. It is important to note that implementing a filtering functionality this way allows for our system to be easily expandable because it is generalised to have the ability to create multiple different filters by simply creating more filter scripts which implement this abstract class (*SurroundingsFilter*)

```

5 public abstract class SurroundingsFilter : ScriptableObject
6 {
7     public abstract List<Transform> Filter(FlockAgent agent, List<Transform> original);
8 }
9

```

Figure 4: SurroundingsFilter Abstract class implementation

For organisation purposes, we created a folder called *Filter Objects* and a folder called *Filter Scripts* in the *Assets* folder. Another C# script, called *PhysicsLayerFilter* was then created in the *Filter Scripts* folder. Referring to Figure 5 for the implementation, *PhysicsLayerFilter* inherits from the *SurroundingsFilter*. This filter iterates through each of the transforms within an agent's neighbouring radius and if it is on a specific physics layer then we take it into account. This filter can then be applied to any behaviour script. First, we created a *LayerMask* variable, called *mask*, to store the physics layer specified from the Unity inspector window. The implementation of the abstract method, *Filter*, uses a new list of transforms to hold the transforms of the filtered objects in an agent's surrounding space. Then, we iterate over each agent within the original list of transforms, and check if it is on a specific physics layer (i.e., if the item's layer is on the same *LayerMask* as the variable on line 7) in which case we add the transform to the filtered list of transforms. We do this check on line 14 by using a process of bit shifting and logic gates.

```

4 [CreateAssetMenu(menuName = "Flock/Filter/Physics Layer")]
5 public class PhysicsLayerFilter : SurroundingsFilter
6 {
7     public LayerMask mask;
8     public override List<Transform> Filter(FlockAgent agent, List<Transform> original)
9     {
10         List<Transform> filtered = new List<Transform>();
11         foreach (Transform item in original)
12         {
13             // if the value of the layer mask is on the same layer as the Transform item, add it to the filtered list
14             if ((int)mask == (mask | (1 << item.gameObject.layer))) //Left-side of or-statement converts item's layer into a layermask
15             {
16                 filtered.Add(item);
17             }
18         }
19         return filtered;
20     }
21 }

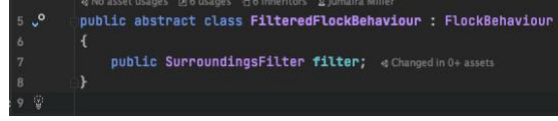
```

Figure 5: PhysicsLayerFilter implementation

In the filter objects folder, we created two new *PhysicsLayerFilter* objects called *Fire Layer Filter* and *Burnt Land Filter* within the *FilterObjects* folder. Their *Mask* variables were set to the *Fire* layer and *Burnt Land* layer, respectively, from the Unity inspector window.

Not every behaviour will require a surroundings filter. For this reason, we created a middleman abstract class called *FilteredFlockBehaviour* which inherits from *FlockBehaviour*. Behaviour scripts can either inherit from *FilteredFlockBehaviour* or directly from *FlockBehaviour* depending on whether they are concerned with filtering an agent's neighbouring objects or not. Again, this has been done to make our system generalisable and easily expandable. Referring to Figure 6, as *FilteredFlockBehaviour* is an abstract class, it does not need to provide an implementation for *FlockBehaviour*'s *CalculateMove* method. Instead, we just simply added a public *SurroundingsFilter* variable called *filter*. Now the behaviours that require a filter are able to inherit from *FilteredFlockBehaviour*. However, to actually apply the filters in these classes, we needed to add a conditional operator to check whether the list of surrounding transforms need to be filtered or not (in some cases we may not have a filter object passed through the Unity Inspector, in which case this filter would be null). This check is done just before iterating through the surrounding objects to calculate an agent's move. A single line of code, as seen in Figure 7, was used to handle this within the behaviours that may require

a filter. This line of code creates a new list of transforms called *filteredSurroundings*, which will simply equal to the existing list of surroundings if no filter has been passed (i.e., *filter == null*), otherwise it will equal to the output of the given Filter applied to the agent and its surroundings (*filter.Filter(agent,surroundings)*).

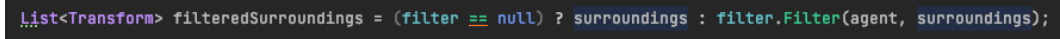


```

5 public abstract class FilteredFlockBehaviour : FlockBehaviour
6 {
7     public SurroundingsFilter filter;
8 }

```

Figure 6: FilteredFlockBehaviour implementation



```

List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);

```

Figure 7: conditional statement to check whether filter must be applied to surroundings

3.1.5 Heat Track Behaviour

HeatTrack simulates the reception of thermal data and seeking to a position of the greatest accumulates temperature. Our system processes the concept of heat using the inverse square law. The inverse square law defines the perceived intensity of a flame, at a certain distance away from the said flame, to be equal to:

$$Intensity = \frac{1}{distance^2}$$

Equation 1: Inverse Square Law [36, 37]

With some simple rearranging, we have decided to use this concept to calculate the perceived heat of an agent to be the accumulated value of $Intensity/distance^2$. We have taken the intensity to be 300 °C (or equally, 572 °F) as this the average temperature required for a tree to be ablaze [38]. We have also decided on take the heat perceived by a drone to be the accumulated temperature as being around more sources of heat will result is a higher overall temperature perceived [37]. Thus, the temperature perceived by a drone will be the summed temperature of all the burning trees around it, where each temperature is computed as $300/distance^2$.

The reader may refer to Figure A.6 in Appendix A for the implementation. The implementation of *HeatTrack*, first has to identify the objects within the range of the thermal sensors. This is an additional stage which *SightTrack* omits because, whilst *SightTrack* can use the neighbouring radius and thus the *surroundings* parameter, the *heatSensorRadius* is different to the neighbouring radius. To do this, we used Unity's physics systems *OverlapCircleAll* to store all transforms around a given agent. An if-condition is used to ensure this list of surrounding objects, are not the drone itself. Although, at this point, the variable which stores all the surrounding objects is called *surroundingFlames*, a LayerMask filter is later applied to remove all objects which are not on the *Flame* LayerMask.

After calculating all the objects within the heat sensor's range (i.e., radius), if it is the case that no objects were detected, the agent does not change its current direction. Otherwise, if flames are detected around an agent, we iterate through each of these transforms and compute the heat perceived from each flame so they can be summed to compute the cumulated perceived heat. To compute the heat and return a move, we first computed the direction to the flame using the difference between the flame's position and the agent's position (*agent.transform.position*) and the direction to the flame using Unity's *Vector2.Distance* method. At this point, we check whether the agent is already too close to a flame, which would mean the stopping condition

for that drone is met, thus the agent's callStop flag will need to be set to 'True'. Otherwise, we carry on seeking the flame by drawing a Raycast to it from the agent's current position. And the total heat is updated using $300/distToFlame^2$. Finally for each flame, the move is computed by accumulating $(300/distToFlame^2)*dirToFlame$.

3.1.6 Sight Track Behaviour

SightTrack simulates the behaviour of agents seeking a target when it is within sight. An agent range of vision is defined by the neighbouring radius. The neighbouring radius is how far an agent can see from its current position. Thus, once an agent detects an object, which is identified to be a flame, the agent will seek (or 'track') it.

To implement a visual sensor-based behaviour was relatively simple - most of the implementation of this behaviour is similar to the Heat Track Behaviour. Each agent already has a neighbour radius, which is used to pass in a list of surrounding objects to the SightTrack behaviour script. If no objects exist in the agent's search radius when surroundings are passed in, this behaviour will just return the agent's current headings to the composite behaviour. This means the direction will remain the same, and this is done by checking if `surroundings.count() == 0` as seen in Appendices A Figure A.7.

If surroundings are not an empty list, the LayerMask filter will be applied to extract only the flames. Then, for each flame in the filtered surroundings, a direction-to-flame vector2 and a float distance-to-flame are calculated so that a Raycast can be drawn. Finally, a move is calculated based off a cumulative value produced from each flame. Inside the for-each flame loop, there is a check to see if the drone is within a safe stopping distance, just to stop the drone moving too close to the flame.

3.1.6.1 Problem: Simulating Line of Sight

In order to differentiate *SightTrackBehaviour* from *HeatTrackBehaviour*, an additional functionality was considered for the former behaviour. Having a Line of sight would mean an agent only seeks a flame if it is within a fixed segment within the neighbouring radius. The fixed segments, enclosed by two lines between an acute angle, represents the area an agent has sight of. The neighbouring radius represents the distance an agent can see around it.

However, as seen in Figure 8, the view angle lines were not in the direction of the agent's heading. Furthermore, the drones kept identifying flames, as indicated by the Raycast lines being drawn, even when they were outside the view angles and beyond the neighbour radius.

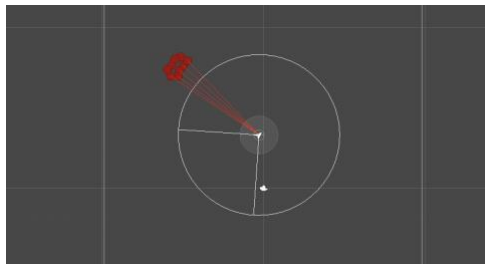


Figure 8: Line-of-sight implementation shown to not be working

As the functionality of having a line of sight was difficult to debug, combined with the pressure of time constraints on this project, an assumption of having rotational cameras was made.

3.1.7 Different Formations

When Implementing different formations for the drones a Boolean variable was added to the flock class. This appears as a checkbox in the unity inspector window and will control how the drones are initiated (or spawned) into the scene. If the Checkbox is ticked 'True' then when the simulation begins, the drones will spawn in a circle formation where otherwise they would spawn in a line. Both implementations (for circle formation and line formation) are all within a single if-else statement. Referring to Figure 10, the if-statement handles circle formation, and the else-statement handles the line statement

```
//Formation, True = Circle and False = Line
public bool CircleFormation = true; // "false"
```

Figure 9: Boolean variable to switch between spawning formations

```
//initialise and instantiate flock
float offset = -(agentCount / 2); //centre offset
for (int i = 0; i < agentCount; i++)
{
    if (CircleFormation == true)
    {
        FlockAgent newAgent = Instantiate(
            agentPrefab, //the object we are instantiating
            position: (Vector3)(Random.insideUnitCircle * FlockDensity), //position at a random points within a circle
            rotation: Quaternion.Euler(Vector3.forward * Random.Range(0f,360f)), //rotation on z axis
            transform //the parent of each agent is going to be this flock instance's transform
        );
        newAgent.name = "UAV/drone " + i;
        flock.Add(newAgent);
    }
    else
    {
        Vector3 position = new Vector3(x:offset + i, y:0, z:0);
        FlockAgent newAgent = Instantiate(agentPrefab, position, rotation:Quaternion.Euler(Vector3.up), transform);
        newAgent.name = "UAV/drone " + i;
        flock.Add(newAgent);
    }
}
```

Figure 10: Implementation of spawning formations

3.1.7.1 Circle Formation

The agents will be instantiated in a circle-like formation if the checkbox condition is true when the simulation is started. To do this, instead of spawning each agent at a new coordinate along the x-axis instead we use *Random.insideUnitCircle*FlockDensity* to give a random position within the circle and place our agent here. The Rotation, or which way the agent will be facing at start, is also randomised by using *Quaternion.Euler(Vector3.forward * Random.Range(0f,360f))* which will return a single degrees offset for each x,y and z.

Problem: Overlap on Flames

After implementing the algorithm for circle formations, it later became clear that something was not working correctly. The error was identified when attempting to get the drones to stop at a safe distance from the fire. The issue was that sometimes, when using the circle formation, some drones would ignore the safe stopping distance and stop on top of a flame, which is bad because the expensive equipment could be damaged and burnt. Furthermore, stopping on top of the flames did not programmatically align with what was implemented. An example of this bug can be seen in Figure 11.



Figure 11: Drones stopping on top of flames

To resolve this, I first tried increasing the stopping distance, but this had no effect on the bug. At this stage it was noticed that the problem occurred more frequently with larger flock sizes (i.e., with more drones) and the problem seemed to disappear when a smaller team of agents were used. Thus, I also tried moving the fires a little further away from the spawn location. This seemed to help by reducing this occurrence, but it did not solve the problem.

When revisiting the code for how an agent is spawned (in Figure 10) inside the circle it became apparent that the mistake was because of the `Instantiate()` method where `Random.insideUnitCircle` is called to pick a position. We were accidentally multiplying (`Random.insideUnitCircle * FlockDensity`) by `AgentCount`. This effectively meant that the size of the spawn circle would also increase with the size of the flock as opposed to being a uniform size for any number of agents. Therefore, if the spawn circle was large enough or if the burning region is too close, then there was a chance that some agents would be initially placed on top of a fire to begin with, hence why they were appearing to have ignored the stopping condition. To resolve this issue, I changed the way the position of the initiate function within the start method is calculated. It was a simple change where I changed the idea of how the drones are spawned: instead of scaling the size of the spawn location from being relative to the number of drones, the spawn area is always the same size regardless of the flock size. This was easily fixed by removing the `*agentCount` from the position vector calculation.

3.1.7.2 Line Formation

The implementation for spawning drones in a line formation works by calculating an offset to the centre of the scene ($- \text{agentCount}/2$) to ensure that no matter how many drones are in the flock, the line formation will always be centred over the middle of the scene. This effectively simulates the flock leaving an initial location and spreading out into formation. This also means that regardless of the size of the flock, each agent will be a uniform distance away from each other to ensure maximum area coverage.

Referring to Figure 10, to instantiate the agents this way, I first use the calculated offset inside a for-loop ($i < \text{agentCount}$) to find a position for the agent by setting the position to `Vector3(offset + i, 0, 0)`. Then we can use this position, the `agentPrefab`, and `transform.up` to place an agent in the desired location for however many agents have been specified at the start of the simulation. Overall, this functionality was fairly simple to implement.

3.1.8 Stop at Distance and Timer

To prevent expensive drone equipment from being damaged by the hazardous environments they are to be deployed in, a stop-at-safe-distance condition was added to the code whilst calculating a move towards the flame. This has been incorporated in both sight and heat tracking behaviours in Figure A.6 (`HeatTrackBehaviour`) and Figure A.7 (`SightTrackBehaviour`) in Appendix A. This condition simply checks if the drone is less than

or equal to 1 unit away from the fire, and if so, update the drone's Boolean variable *stopped* to 'True'.

Then, in the Flock script where we instantiate and update each drone's position (Figure A.8 in Appendix A) we first check if the stopping condition has been met (which indicates all drones have found the fire). *SystemStoppingCondition*, which can be referred to in Appendix A Figure A.8, is a simple private method which iterates over each agent in the flock and returns 'False' at the first instance where an agent that does not call the stop flag (i.e., if even a single agent's *callStop* Boolean equals to 'False'). Only once this check has passed for every drone in the flock (no drones returned 'False') will this for-each loop terminate – at which point method will return 'True' back to the *Update* method thread.

The timer was added for testing purposes – i.e., to measure the effectiveness of the current flock in terms of finding a forest fire boundary. A float variable, *timer*, was declared and instantiated to 0 in the Start method within the Flock script. This variable is then incremented by $1 * \text{Time.deltaTime}$ in the Update method as seen in Appendix A Figure A.8. This *Time.deltaTime* function will return the time it taken to finish the last frame (in seconds), so when we increment the timer by $1 * \text{Time.deltaTime}$ the result is that the simulation will now run independently of the user's current framerate and produce an accurate time across all iterations [39].

Finally, once the *SystemStoppingCondition* method has returned 'True', indicating that all drones have found the flames and thus has stopped searching, the time value will be outputted to console. There was a minor issue where the timer output was continuously returned even after the simulation run had finished. This was identified to be due to the Update method being called every frame and all drones remaining stationary. This bug was easily resolved by adding another bool condition, *continueTimer*, which is set to 'False' upon the first instance of *SystemStoppingCondition* passing.

3.2 Results

Several tests have been executed to examine the performance of the system with different hyperparameters. Successive test scenarios get increasingly complex, each of which consider the varying hyperparameters. The hyperparameters are the test scenes themselves as they represent the different structures of burning regions, the size of the flock/team of UAVs, the weights for each simple behaviour within the overall composite behaviour, and the spawn formation.

Two main composite behaviours will be examined through testing. The behaviours only differ in the flame tracking behaviour being used, which can be either *HeatTrackBehaviour* or *SightTrackBehaviour*. If the reader refers to the provided github link to our system (within Assets>Objects>Behaviour Objects>Composite Behaviour Objects) the first composite is called *Seek Heat Source Behaviour* and the second composite is called *Visual Search Behaviour*.

To begin with, first the flock size has been assessed for each of the three test scenes with a base case for the weightings of each simple behaviour which composes the overall composite

behaviour. The simple behaviours' weights are set such that Alignment is set to 1, Avoidance is set to 15 to stop agents from crashing into each other, Steered Cohesion is set to 1, KeepWithinR15 is a base functionality which sets the search space radius to 15 units of the scene and this behaviour's weight is set to 0.2, and the flame tracking behaviours (Heat Track and Sight track behaviours) are set to 10.

On the second test scene, more extensive experimentation was conducted to finetune the weightings of the individual behaviours within the composite. These tests also provided means for further investigation of interesting observations.

Performance of each test is measured in terms of which combination of spawn formation, flock size, and tracking behaviour resulted in the shortest search time. Another measurement used for assessment was completeness (i.e., what proportion of the forest fire boundary was found).

3.2.1 Scenario 1



Figure 12: Test Scene 1

3.2.1.1 Description

In our first scenario, as seen in Figure 12 we simply have a single red circle to indicate a lone burning tree. The flock will spawn at the centre of the scene with the single flame located roughly $(-2,0,4)$. It is almost always found within a couple of seconds and has no boundary to be found as there is only one flame.

3.2.1.2 Observations

Scenario 1

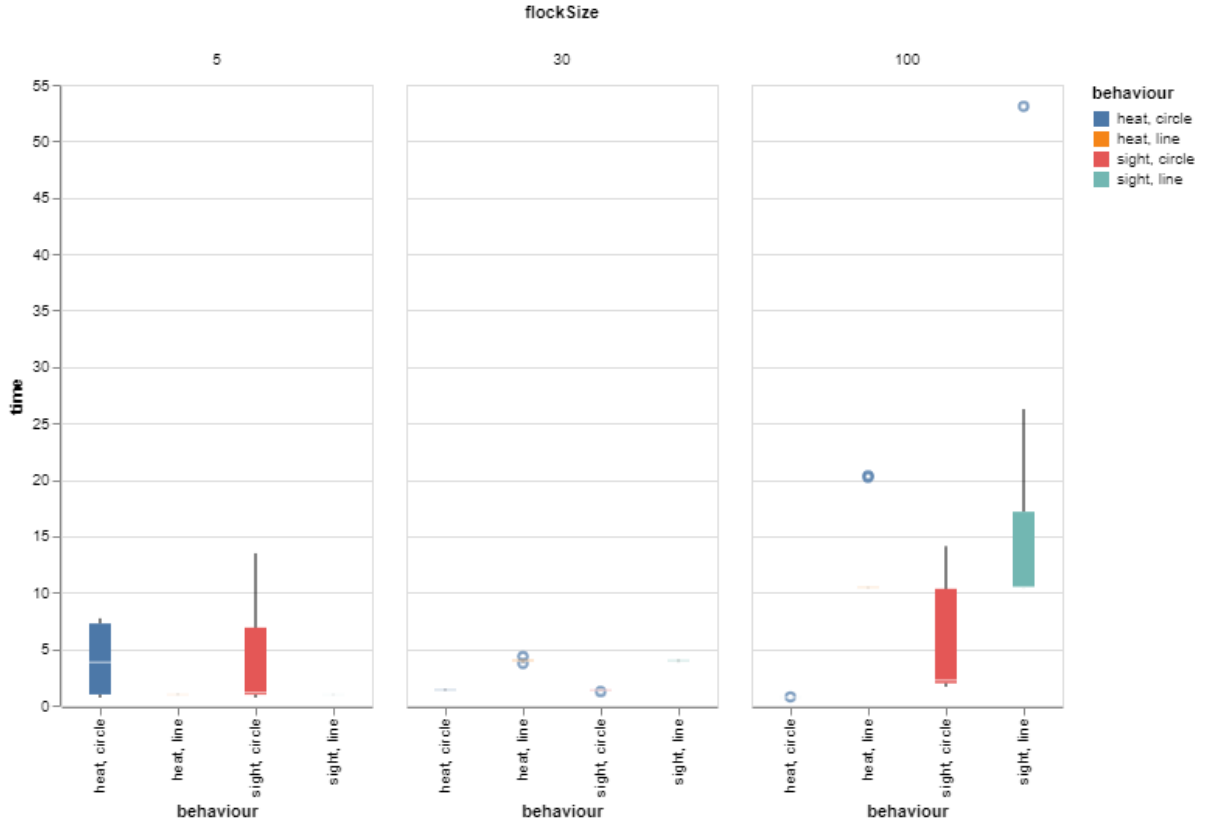


Figure 13: Scenario 1's Results in a Boxplot

For this first scenario, the formation and flock size are less important factors effecting the search time because there is only one flame to be found. The optimal size for this scene is 30 drones in the flock, as all four test cases using 30 drones consistently found the fire in less than 5 seconds (as shown by the interquartile ranges being single lines in Figure 13, with the extra drones helping to steer other nearby drones in the correct direction once they had located the fire. Going beyond 30 agents was only detrimental in most cases, as a larger flock size will only delay the final time produced because the time will only be given once every agent has located the fire and stopped at a safe distance. This is not always true though, as Figure 13 visualises, heat track behaviour tested with 100 drones still performed well - with no visible effect on the time (except for a single anomalous record). However, this was most likely due to the close proximity of the fire to the drones start/spawn locations.

Behavioural weightings also have little effect, because there is only 1 flame, and it was always found quite quickly.

HeatTrack does better with larger flock sizes in a circle formation, as it has a shorter search time, however the opposite is true for HeatTrack in a line formation. For Line formation, as the flock size increases, search time gets longer. This can be seen in Appendix F.1 in the final column, where all the times for flock size of 5 are consistently below 1 second, with a size of 30 the times stay below 5 seconds. Finally, when the flock size is 100 the time is normally somewhere between 10 and 11 seconds with 2 potentially anomalous instances of a 20 second search.

For sight track with line formation, again 30 drones seem to be the optimal flock size as 5 drones are too few to find the entire boundary and 100 is too many. With 100 drones, even after the entire boundary is found, as shown in Figure 14, the system does not stop because

there are still many drones searching. As the behaviour is implemented in a way that the system does not stop until all drones have found the boundary, the time taken for 100 drones is far longer than it needs to be. Instead, an alternative implementation of sight track could be to stop the movement of agents (and thus the timer) after the single flame is enclosed by drones rather than stop the timer once all drones have found the boundary.

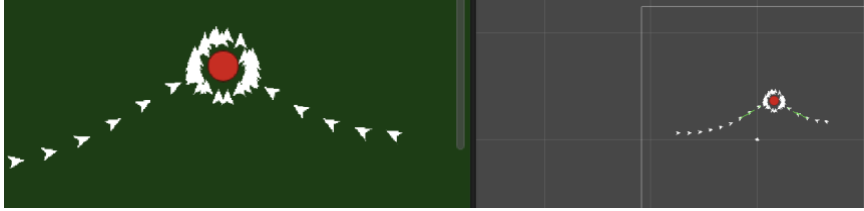


Figure 14: Scene depicting boundary already found with remaining agents being excessive

Circle formation seems to be generally quicker, although this may only be because the circle formation spawns all the drones closer to the fire than the line formation does. However, line formation generally seems to have more consistent times, evident from the smaller interquartile ranges in Figure 13, for finding the boundary. This may be as a result of the drones being more consistently spawn in the same location whereas the circle formation randomly spawns drones within a circular area region (refer to code for instantiating drones in Figure 10 in *Methodology* section 3.1.7). Another reason for line formations consistent results may be as a result of being able to better span across the search space; Line formation evenly spread the agents in the flock in a straight line whereas circle formation initially disadvantages the search by confining the spawning to a smaller search space.



Figure 15: end result of composite behaviour (i.e., Visual Search Behaviour) with sightTrackBehaviour and in line formation

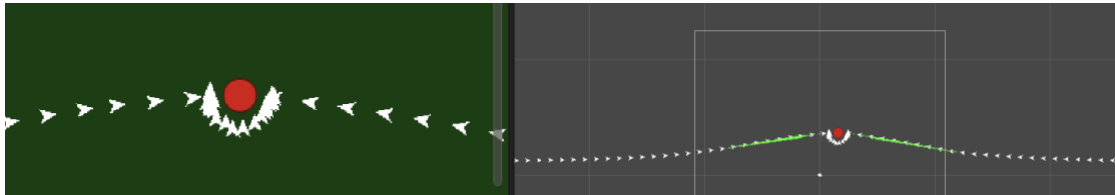


Figure 16: Visual Search Behaviour ceasing to complete top part of single flame's boundary

For 100 drones using sightTrack behaviour and line formation, after a certain portion of the boundary, the drones cease to wrap around the top - prohibiting them from finding the entire boundary. This may be because the drones spawn close to the bottom side of the single flame and thus do not get a chance to spread past a certain point in the search space around the flame. This issue is visualised within Figure 15 and Figure 16.

For 30 drones using heatTrack and the line formation, sometimes for some reason the last two or three drones remaining often miss the fire and continue searching for a very long time. This could be for a similar reason as the previous issue. For some unclear reason, they seem to get stuck in a seemingly irregular loop in pairs or trios. However, the reason for why it takes a long time before breaking out of this loop might be explainable. It may be because the drones get stuck in a loop of very few agents. This could be reducing the chances of having a vastly different alignment of any single drone which could have led the other drones to break free.

They are essentially tangled in each other's moves. This bug does not persist in sightTrack. The only difference between the implementation of the two tracking behaviours is the calculation of what we seek the flames based on - heat (proportional to $1/\text{distance}^2$) or sight (proportional to the distance). Thus, this leads to the hypothesis that the inverse square calculation used to measure heat must be the cause of this unexplainable behaviour.

3.2.2 Scenario 2



Figure 17: Test Scene 2

3.2.2.1 Description

This Scenario, as depicted in Figure 17, is composed of many flames to outline a patch of burning land with the centre region's flames having died out. The drones now have an irregular boundary which can be searched for. This test scenario also includes burnt land, represented by the black dots, which the drones must fly over. The drones are to only detect burning regions of land. However, as the burnt region is surrounded by a burning region, as expected, no drones flew over this region or came too close to the red burning flames.

3.2.2.2 Observations

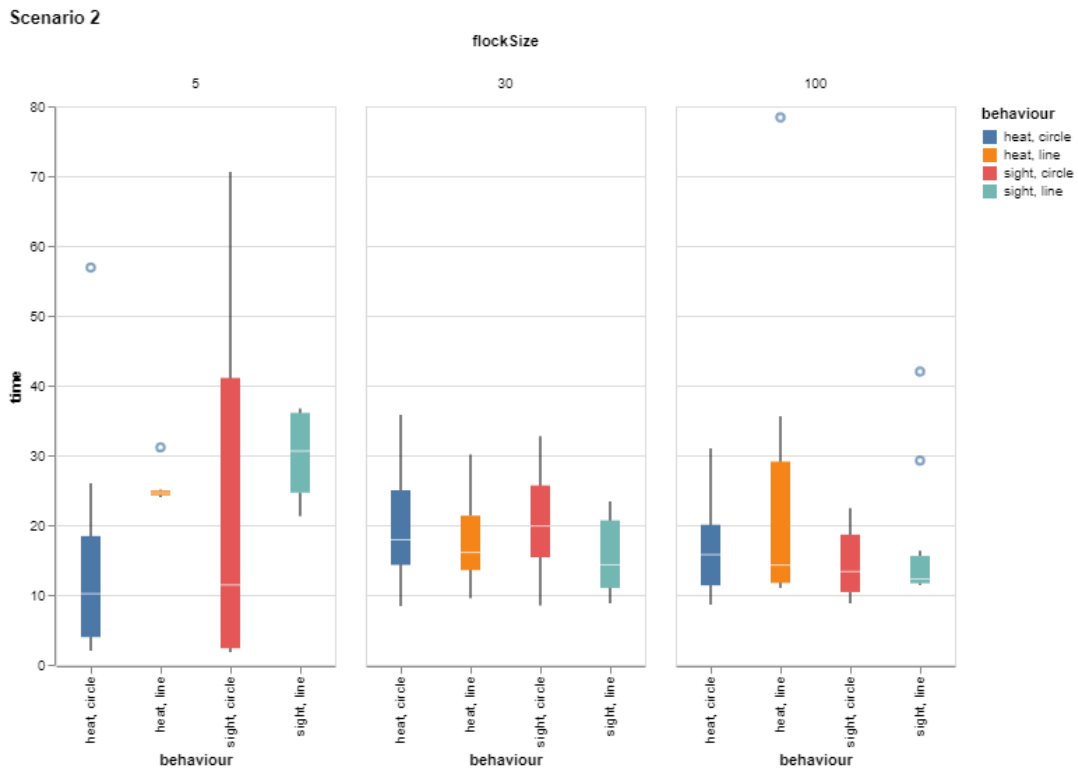


Figure 18: Scenario 2's Results in a Boxplot

Big groups tended to perform better as they will drag each other towards the flames when they find them. However, if small sub-groups of drones are left by themselves near the end of the

search, it can take far longer for them to find the rest of the fire. This is reflected in Figure 18 as both the visual sensor tests with only 5 agents took longer, with the circle formation interestingly having the largest interquartile range but also one of the lowest averages for the whole scenario. This is due to how the circle formation flock is initialised (Figure 10) with a random direction being chosen, meaning that quite simply the cause for such large total & interquartile ranges is down to whether the flock is travelling in the right direction from spawn. Small groups struggled to find the fire boundaries so much in this scenario because the fire itself is located so close to the edge of the search radius that a small flock's total visual sensor region must be very close to find the fire at all. On top of this, as the forest fire in this scene is modelling a disconnected fire (the start and end of the fire do not loop into each other, but instead there are broken into smaller fires with larger areas of burnt area for the drones to fly over) the agents are required to find flames spread across an overall larger search space.

Line formation produced smaller sub-groups more frequently than circle formations did, meaning the line formations can take slightly longer in this scene. However, if part of the line segment finds the fire whilst the majority of the line formation is intact, then it will cause any drones who miss the fire to swing more severely towards the direction that adjacent drones in the formation had taken. This formation produces the most consistent results for a medium or small flock size as seen in Figure 18. However, *heatTrack* with line formation for a flock size of 100 surprisingly performed the worst when it had been favourable in the smaller sized tests. In Appendix B Figure B.2, *heatTrack* with line formation and a flock size of 5 was so consistent that the interquartile range for all 10 tests were only 0.98 seconds, with a single instance taking 31 seconds.

3.2.3 Scenario 3



Figure 19: Test Scene 3

3.2.3.1 Description

The final test scenario is the most complex. Although the assumption of only having connected fires was made, this test scenario acts as a stress test for the system. The scene contains two disconnected fires, one is a connected ring of burning trees (burning region on the left side), and the other is in itself also disconnected (burning region on the right side).

3.2.3.2 Observations

Scenario 3

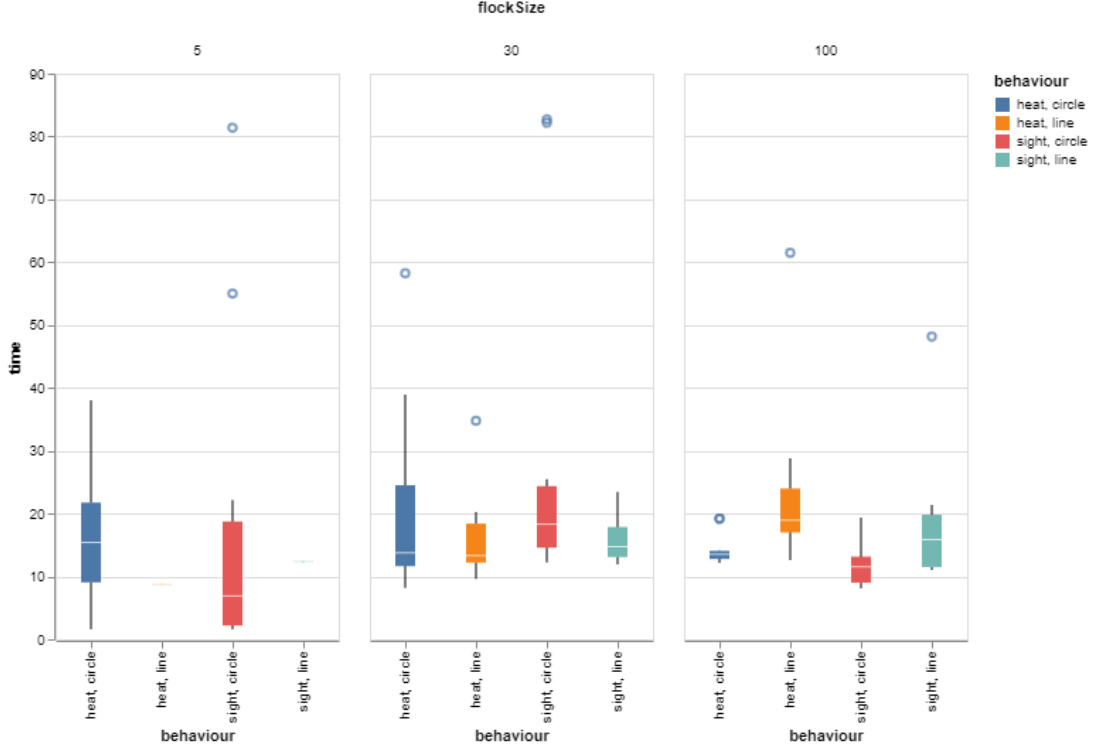


Figure 20: Scenario 3's Results in a Boxplot

The most reliable and accurate test case for this scenario was the line formation with the smallest flock size. Both Heat and Sight tracking behaviours produced very consistent times with sight taking just over 12.3 seconds and had a total range of 0.18, whilst heat tracking was slightly faster at 8.6 seconds and too had a total range of 0.18. This raw data is shown in Appendix B Table 3. Whilst the times for *HeatTrack* and *SightTrack* with a team of 5 UAVs resulted in the fastest and most reliable testcases for this scenario, this is most likely due to the line formation only having to make one sweep of the total search area for this scenario 3 as seen in Figure 20. The 5 agents are initialised in a line formation facing the top of the scene, upon reaching the boundary they make a complete 180 degree turn and fly to the opposite side of the search area where they always found the fires.

The results shown in Figure 20 (Scene 3 boxplot) are very similar to those from scenario 2 (Figure 18). However, each individual testcase seems to have performed slightly better. In previous scenario tests, it seemed to be the case that the medium flock size was ideal because flock sizes that were too large only delayed the final time. This is evident in Figure 13 (Scene 1 boxplot) and Figure 18 (Scene 2 boxplot) which show the highest interquartile time ranges in the largest flock size; Figure 20 (Scene 3 boxplot) shows different results, where the largest flock size had a better spread, lower anomalous values and ranges than when smaller teams of UAVs were used. This could mean that the fire boundary is more reliably detected with a larger flock when the burning areas themselves are disconnected.

3.2.4 Other Interesting Behavioural Observations

Lemniscate Bug

When investigating the balance of weightings for each simple behaviour composing composite behaviours, it was noticed that a specific condition led to SoftLock. This is when the simulation never completes. This occurs when the alignment behaviour's weighting is only slightly lower, equal to, or greater than the avoidance behaviour weighting (i.e., when the weighting for the two behaviours is very closely similar). In such instances, when finding a fire and attempting to fly towards it, we end up in a SoftLock. The error appears as a portion of the agents becoming trapped in a Lemniscate type pattern (infinity symbol) which will either continue indefinitely or for an extremely long time. Figure 21 portrays an example of such an occurrence (image below, infinity pattern).

This is probably because the drones are too busy aligning to the other drones around the fire boundary (which are facing away from the direction towards the flames) to be able to fly close enough to stop themselves properly. Essentially, the drones are conflicted between whether to prioritise alignment or avoidance – the agents still searching attempt to align to the successful agents but begin to avoid these same successful agents before the agents can move close enough to the fire (because they have come too close to the stopped agents).

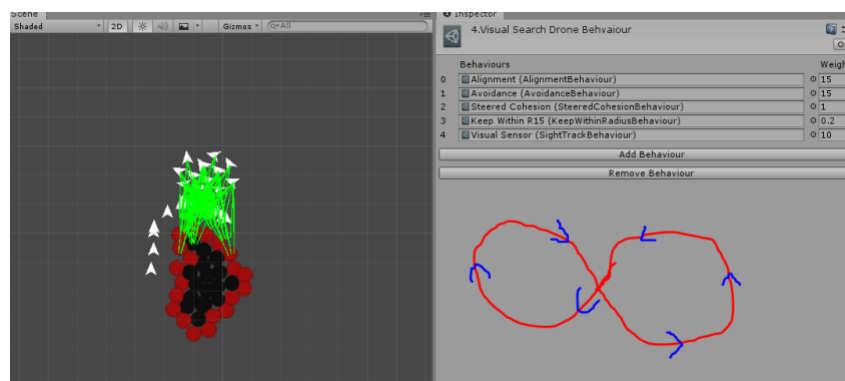


Figure 21: Scene depicting issue of Lemniscate bug in Unity's interface

Balancing Flock Size and Avoidance Behaviour

It is important to find a healthy balance between avoidance and flock size to ensure the full fire boundary is being found. If the avoidance weighting is high enough, when finding a fire, drones will space themselves out nicely along the boundary of the fire. However, if the flock size is too large, there is a greater likelihood that too many agents will already be crowding a section of space around the fire boundary. Having agents with a high avoidance weighting, within a large flock where agents begin to crowd the boundary, will result in some agents getting caught in a SoftLock. In this case, some agents will continue spinning around the boundary. This may never stop because of the high density of agents triggering avoidance behavioural movements before reapproaching the fires in an attempt to reach the safe stopping distance. The main side effect this spinning causes is that the time will never be displayed not all the drones will come to a stop, meaning tests where this error occur are invalid and useless.

Depending on the size of the burning region, the size of the flock and avoidance weighting needs to be balanced in order to space the drones out in such a way that the entire boundary is found without too many trapped agents. Figure 22 shows that increasing avoidance will spread the agents out enough to find the entire boundary, however, because the avoidance is too high some agents never stop moving because they cannot fit between other agents to find space at the safe stopping distance around the fire boundary. Despite this issue in the simulation, in

reality the drones would be able to space themselves out in 3 dimensions whilst finding fire boundaries, so this error is only a minor concern.

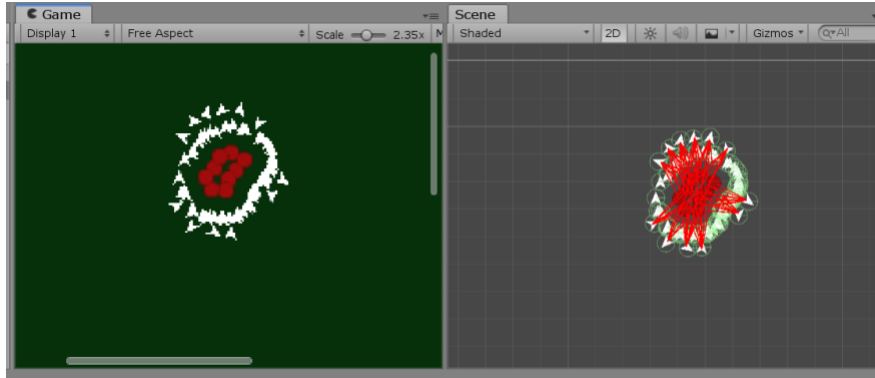


Figure 22: Scene depicting some agents unable to stop due to high weighting of Avoidance Behaviour

Chapter 4

Conclusions and Future Work

4.1 Conclusion

In conclusion, yes swarm intelligence can be used for fire boundary detection, although the overall effectiveness of performance will depend upon the composite behaviours, spawn formation, flock size and weightings.

It appears to be the case that a Line formation is the most reliable, but not necessarily always the fastest. Furthermore, Heat track behaviours made a formed a better composite behaviour in terms of search time due to the heat track behaviour having a longer data reception radius. In general, the optimal flock size will depend largely on the distribution of the fire. In order to find the lowest number of flock agents, based on the distribution density of a given fire, further investigating needs to be done in balancing alignment and avoidance behaviours with the flock size. To a large extent, the accuracy of boundary detection may also depend on the spawn position relative to the fire location and distribution density structure.

4.2 Evaluation

Due to the emergent nature of SI systems, testing was very difficult. Further to this, the large number of parameters also made it difficult to assess. However, based on the evaluation criteria discussed (time, completeness, and precision), we were able to implement a composite behaviour which returned relatively fast results which were very consistent. This passed our criteria for time and precision. However, although we were able to identify the parameters that need to be balanced in order to meet the criteria of completeness, further investigation must be done to satisfy this criterion completely.

A positive about our system is that it is very much generalizable, which allow for future development on our work.

4.2.1 Technical Reflection

As a reminder, aims of our project were as follows:

- Create an effective tracking behaviour to combine into a composite behaviour which can identify the boundary of a forest fire.
- Find the optimal flock size for given forest fire formations
- Find the optimal hyperparameters (which will be defined within the *Methodology* Section) to optimise the portion of the forest fire front being found

Our system was able to achieve the first aim by implementing two tracking behaviours, which were then used to create two composite behaviours. Amongst the two behaviours, HeatTrackBehaviour seems to produce the optimal composite behaviour for boundary detection as it was the most precise, and combined with line formation, it also resulted in the most consistent results, and the fastest search times.

Although we have come to the conclusion that the optimal flock size for our three test scenes were 30, it is sensible to conclude the true optimal will depend largely on the size of the fire.

As mentioned above, we have identified that balancing flock size and avoidance will likely achieve a higher score in terms of our evaluation criteria of completeness. This is because it can aid in finding the minimal number of agents required to find the entire boundary by using avoidance to spread the agents across the said boundary. However, we must also acknowledge that it is difficult to measure the actual percent of boundary being found. It may have been better to instead look at the number of fires found out of the total fires that are currently on screen. An alternative could also be to simply allow a single drone for a certain infected area.

Not being able to find the entire boundary is a clear limitation of our system which requires further work. A wall following behaviours, or flow field following behaviour may have resolved the issues of finding an optimal flock size. Either of these behaviours may have also satisfied our completeness criteria (of finding the entire fire boundary) as it could allow for a single drone to trace the entire boundary. This could mean we could have very small flocks, and as soon as a single agent finds a section of the boundary, it can proceed to trace and find the rest.

4.2.2 Personal Reflection

4.2.2.1 Change to Project Scope

Our initially proposed system was intended to use an off-the-shelf simulation script of a forest fire. Through this simulation, we had intended to train an unsupervised ML model to obtain insights on the propagation of forest fires, to then be able to predict the propagation of an unseen forest fire simulation. The main reason for using a readily available Unity 3D simulation script was to focus our research on optimisation and ML algorithms. Using SI for effective search algorithms and optimisation methods was initially only the first stage of the original project outline.

Listed below is a breakdown of the three identified stages for the original proposed system:

1. Boundary Detection and Optimisation

Using an aerial image of the state of a forest fire at a specific timestamp, we needed to first be able to detect the boundary of the fire. Once our search algorithm captures the boundary, we intended to optimise the algorithm using either spatial optimisation or particle swarm optimisation.

2. **Train an ML Model**

The search algorithm would then be used on then several simulations to curate a dataset where each sample is a series of images at different timestamps of a single forest fire simulation. An ML model would then be trained on this processed dataset of forest fires using an auto-encoder which made use of two Neural Networks.

3. **Predict the Propagation of a Forest Fire**

The resulting weight from the auto-encoder would then be fed into a predictor which should then be able to construct an image of the forest fire at a later timestamp.

Following the initial project outline, the aims were as follows:

Primary Aims	<ul style="list-style-type: none"> - Modify an off-the-shelf forest fire simulation to be able to take in a dataset and continuously generate simulations (the script currently requires manual input of attributes to generate a single simulation). - Successfully implement an intelligent swarm system of drones with effective swarm behaviour for efficient boundary detection using a segmentation algorithm. - Learn the ML.NET library and be able to implement a basic auto-encoder, using neural networks, to train an ML model of forest fire propagation.
Secondary Aims	<ul style="list-style-type: none"> - Optimise the initial location-allocation of a swarm of drones for an optimal segmentation algorithm. - Implement a predictor using the resulting weight form the auto-encoder to be able to approximate the propagation of a forest fire after a given time.

From the very beginning we had anticipated the time constraints may affect our project's scope, thus we had intended to focus on the primary aims first, before attempting the secondary. At a very early stage, we refocused our project to only target the task of boundary detection as this was the first stage of our original project proposal. Our revised project still managed to achieve the second primary aim of successfully implementing a SI system of drones for boundary detection. However, we based our search on the flocking algorithm rather than a segmentation algorithm.

4.3 Future Work

Our projected currently only considers static forest fire boundaries. To develop our project in the direction of our original project's scope, we could begin by augmenting our system to also be able to detect dynamic forest fire boundaries. This would mean the agents would have to remain a safe distance away from the boundary, but also propagate in the way the flames do.

Other formations, such as triangle formations, could be investigated to further optimise the search times. A triangle formation, as proposed by Sherstjuk et al., may force flocks to adapt triangle shaped formation during flight (not just when spawning) like birds do to maximise the flock's search radius and hopefully gain both the best parts of line and circle formations. This could be done by using Sierpinski triangle formations, although this would probably be difficult to implement.

Implement a behaviour for a pinging system to alert the rest of the flock when any single agent finds a fire may benefit in reducing the search time even further. This could be done by broadcasting the direction of the fire found by an agent to all neighbours within its neighbour radius. These neighbouring agents could then work out the direction of the fire from themselves again, send the information to their neighbours, before coordinating themselves towards the fire. There would need to be a check to stop neighbours updating the drone that just pinged them in the first place, but there are ways to do this.

Bibliography

- [1] M. Slezak, “3 billion animals killed or displaced in Black Summer bushfires, study estimates,” ABC News, 20 July 2020. [Online]. Available: <https://www.abc.net.au/news/2020-07-28/3-billion-animals-killed-displaced-in-fires-wwf-study/12497976?nw=0>. [Accessed 2020 October 23].
- [2] D. F. Karin Brulliard, “A billion animals have been caught in Australia’s fires. Some may go extinct,” The Washington Post, 9 January 2020. [Online]. Available: <https://www.washingtonpost.com/science/2020/01/09/australia-fire-animals-killed/>. [Accessed 2020 October 21].
- [3] A. González-Cabán, “Social Impact of Large-Scale Forest Fires,” in *Proceedings of the Second International Symposium on Fire Economics, Planning, and Policy: A Global View*, Riverside, 2008.
- [4] M. Rath, A. Darwish, B. Pati, B. K. Pattanayak and C. R. Panigrahi, in *Swarm intelligence as a solution for technological problems associated with Internet of Things*, Academic Press, 2020, pp. 21-45.
- [5] E. Besada-Portas, L. de la Torre, J. de la Cruz and B. de Andrés-Toro, “Evolutionary Trajectory Planner for Multiple UAVs in Realistic Scenarios,” *IEEE Transactions on Robotics*, vol. 26, no. 4, pp. 619-634, 2010.
- [6] J. M. J. G. C. S. J. Robert S. Allison, “Airborne Optical and Thermal Remote Sensing for Wildfire Detection and Monitoring,” *Sensors*, vol. 16, no. 1310, 2016.
- [7] A. A. A. Alkhatib, “A Review on Forest Fire Detection Techniques,” *International Journal of Distributed Sensor Networks*, vol. 2014, no. 597368, 2014.
- [8] M. Z. I. S. Vladimir Sherstjuk, “Forest Fire Monitoring System Based on UAV Team, Remote Sensing, and Image Processing,” in *2018 IEEE Second International Conference on Data Stream Mining & Processing (DSMP)*, Lviv, Ukraine, 2018.
- [9] P. C. F. O. Á. D. F. L.-P. R. J. D. Gervasio Varela, “Swarm intelligence based approach for real time UAV team coordination in search operations,” in *2011 Third World Congress on Nature and Biologically Inspired Computing*, Salamanca, Spain, 2011.
- [10] e. a. Piyush Jane, “A review of machine learning applications in wildfire science and management,” *Environmental Reviews*, 2020.
- [11] L. Q. Y. S. Y. S. S. C. J. Z. X. S. Jian Yang, “Swarm Intelligence in Data Science: Applications, Opportunities and Challenges,” in *Advances in Swarm Intelligence*, Springer, Cham, 2020, pp. 3-14.

- [12 M. S. Ravi N. Haksar, "Distributed Deep Reinforcement Learning for Fighting
] Forest Fires with a Network of Aerial Robots," *2018 IEEE/RSJ International
Conference on Intelligent Robots and Systems (IROS)*, pp. 1067-1074, 2018.
- [13 J. S. S. S. K. Harikumar, "Multi-UAV Oxyrrhis Marina-Inspired Search and
] Dynamic Formation Control for Forest Firefighting," *IEEE Transactions on
Automation Science and Engineering*, vol. 16, no. 2, pp. 863-873, 2019.
- [14 Z. L. Y. Z. Chi Yuan, "Vision-based forest fire detection in aerial images for
] firefighting using UAVs," *2016 International Conference on Unmanned Aircraft
Systems (ICUAS)*, pp. 1200-1205, 2016.
- [15 H. R. Mubarak A. I. Mahmoud, "Forest Fire Detection Using a Rule-Based
] Image Processing Algorithm and Temporal Variation," *Mathematical Problems
in Engineering*, vol. 2018, no. 7612487, 2018.
- [16 I. T. Sahin Y. Guneri, "Early Forest Fire Detection Using Radio-Acoustic
] Sounding System," *Sensors*, vol. 9, no. 3, pp. 1485-1498, 2009.
- [17 G. H. J. R. P. Z. Diyana Kinaneva, "Early Forest Fire Detection Using Drones
] and Artificial Intelligence," in *2019 42nd International Convention on
Information and Communication Technology, Electronics and Microelectronics
(MIPRO)*, Opatija, Croatia, 2019.
- [18 D. Soni, "Introduction to Evolutionary Algorithms," towards data science, 18
] February 2018. [Online]. Available: [https://towardsdatascience.com/introduction-
to-evolutionary-algorithms-a8594b484ac](https://towardsdatascience.com/introduction-to-evolutionary-algorithms-a8594b484ac). [Accessed March 2021].
- [19 Anurag, "13 Pros & Cons to Know Before Choosing Unity 3D," NewGenApps,
] 30 March 2018. [Online]. Available: [https://www.newgenapps.com/blog/unity-
3d-pros-cons-analysis-choose-unity/](https://www.newgenapps.com/blog/unity-3d-pros-cons-analysis-choose-unity/). [Accessed 16 April 2021].
- [20 PracticalCode, "Unity 4.3 vs jMonkeyEngine 3 (jME 3) — Teaching
] Programming vs Engine Benefits," Unity, 7 February 2014. [Online]. Available:
[https://forum.unity.com/threads/unity-4-3-vs-jmonkeyengine-3-jme-3-teaching-
programming-vs-engine-benefits.226937/](https://forum.unity.com/threads/unity-4-3-vs-jmonkeyengine-3-jme-3-teaching-programming-vs-engine-benefits.226937/). [Accessed April 2021].
- [21 M. B. Marco Dorigo, "Swarm Intelligence," *Scholarpedia*, vol. 2, no. 9, p. 1462,
] 2007.
- [22 K. M. P. Yang Liu, *Swarm Intelligence: Literature Overview*, Columbus, Ohio:
] The Ohio Sate University, 2000.
- [23 F. C. Belkacem Khaldi, "An Overview of Swarm Robotics: Swarm Intelligence
] Applied too Multi- robotics," *International Journal of Computer Applications*,
vol. 126, no. 2, pp. 31-37, 2015.
- [24 J. V. K. P. Alec Banks, "Particle Swarm Guidance System for Autonomous
] Unmanned Aerial] Vehicles in an Air Defence Role," *The Journal of
Navigation*, vol. 61, no. 1, pp. 9-29, 2008.
- [25 B. S. G. d. A. Victor C. Leita, "Particle Swarm Optimization: A Powerful
] Technique for] Solving Engineering Problems," in *Swarm Intelligence - Recent
Advances, New Perspectives and Applications*, IntechOpen, 2019, pp. 9-29.

- [26 C. Reynolds, “Boids,” 6 September 2001. [Online]. Available:
] <https://www.red3d.com/cwr/boids/>. [Accessed 02 November 2020].
- [27 C. Ferguson, “Particle Swarm Optimisation,” Youtube: Churchill CompSci
] Talks, 24 March 2018. [Online]. Available: <https://youtu.be/DzcZ6bP4FGw>.
[Accessed 03 November 2020].
- [28 D. Shiffman, “The Nature of Code Chapter 6: Autonomous Agents,” 2012.
] [Online]. Available: <https://natureofcode.com/book/chapter-6-autonomous-agents/>. [Accessed 03 November 2020].
- [29 A. Fray, “CONTEXT BEHAVIOURS KNOW HOW TO SHARE,” 26 March
] 2013. [Online]. Available:
<https://andrewfray.wordpress.com/2013/03/26/context-behaviours-know-how-to-share/>. [Accessed 12 March 2021].
- [30 J. Keats, “AI COntext Behaviours,” 2016. [Online]. Available:
] <https://jameskeats.com/portfolio/contextbhvr.html>. [Accessed 2021].
- [31 Unity, “Transform.LookAt,” Unity, 2021. [Online]. Available:
] <https://docs.unity3d.com/ScriptReference/Transform.LookAt.html>. [Accessed
2021].
- [32 R. L. C. P. J. Arika Ligmann-Zielinska, “Spatial optimization as a generative
] technique for sustainable multiobjective land-use allocation,” *International Journal of Geographical Information Science*, vol. 22, no. 6, pp. 601-622, 2008.
- [33 boardedtobits, “flocking-algorithm,” GitHub, 14 March 2019. [Online]. Available:
] <https://github.com/boardtobits/flocking-algorithm>. [Accessed 15 January 2021].
- [34 Unity, “Mathf.SmoothDamp,” February 2021. [Online]. Available:
] <https://docs.unity3d.com/2020.1/Documentation/ScriptReference/Mathf.SmoothDamp.html>. [Accessed February 2021].
- [35 Unity, “LayerMask,” 2021. [Online]. Available:
] <https://docs.unity3d.com/ScriptReference/LayerMask.html>. [Accessed 2021].
- [36 Insight Numerics, “FOV Multiplier and Inverse Square Law Theory,” Insight
] Numerics, 2020. [Online]. Available:
https://help.insightnumerics.com/Detect3D/Tutorials/Tutorial_07_Inverse_Square_Law/Tutorial_07_FOV_Multiplier_and_Inverse_Square_Law_Theory.htm.
[Accessed 2021].
- [37 G. Blonder, “Fire away,” Genius Ideas, 9 October 2016. [Online]. Available:
] <https://genuineideas.com/ArticlesIndex/fireAway.html>. [Accessed 12 February
2021].
- [38 B. Gabbert, “At what temperature does a forest fire burn?,” Wildfire Today, 26
] February 2011. [Online]. Available: <https://wildfiretoday.com/2011/02/26/at-what-temperature-does-a-forest-fire-burn/?sfw=pass1620159010>. [Accessed 12
February 2021].

[39 Unity, “Time.deltaTime,” 2016. [Online]. Available:
] [https://docs.unity3d.com/530/Documentation/ScriptReference/Time-
deltaTime.html](https://docs.unity3d.com/530/Documentation/ScriptReference/Time-deltaTime.html). [Accessed February 2021].

Appendix A

Code Implementation

```
6 public class AlignmentBehaviour : FilteredFlockBehaviour
7 {
8     /*
9     * find the new direction of an agent based on alignment behaviour
10    * (move in the average direction of all neighbouring agents)
11    */
12    # Frequently called 0+2 usages Jumaira Miller
13    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
14    {
15        //maintain current direction if there are no neighbouring objects
16        if (surroundings.Count == 0)
17        {
18            return agent.transform.up;
19        }
20
21        //otherwise get the average of the sum of all the directions of each neighbouring object
22        Vector2 alignmentMove = Vector2.zero;
23        List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);
24        foreach (Transform item in filteredSurroundings)
25        {
26            alignmentMove += (Vector2)item.transform.up;
27        }
28        alignmentMove /= surroundings.Count; // normalised direction of all neighbouring agents
29        return alignmentMove;
30    }
31 }
32 }
```

Figure A. 1: Alignment Behaviour

```

6 public class AvoidanceBehaviour : FilteredFlockBehaviour
7 {
8     /*
9     * find the new position of an agent based on avoidance behaviour
10    * (move away from agents in the personal/avoidance radius of the given agent)
11    */
12    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
13    {
14        //No need to make any adjustments if there are no neighbouring objects
15        if (surroundings.Count == 0)
16        {
17            return Vector2.zero; // return a vector with no magnitude = no adjustments made
18        }
19
20        //otherwise get the average of the sum of all the positions of each neighbouring object
21        Vector2 avoidanceMove = Vector2.zero;
22        int nAvoid = 0; // counter for number of objects to avoid
23
24        List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);
25        foreach (Transform item in filteredSurroundings)
26        {
27            // update counter and move if item is within personal radius
28            if (Vector2.SqrMagnitude(item.position - agent.transform.position) < flock.SquarePersonalRadius)
29            {
30                nAvoid++;
31                avoidanceMove += (Vector2)(agent.transform.position - item.position); // sum of offset positions
32            }
33        }
34        if (nAvoid > 0)
35        {
36            avoidanceMove /= nAvoid; // normalise sum of offsets
37        }
38        return avoidanceMove;
39    }
40 }

```

Figure A. 2: Avoidance Behaviour

```

6 public class CohesionBehaviour : FilteredFlockBehaviour
7 {
8     /*
9     * find the new position of an agent based on cohesion behaviour
10    * (move to the mid-point between all neighbouring agents)
11    */
12    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
13    {
14        //No need to make any adjustments if there are no neighbouring objects
15        if (surroundings.Count == 0)
16        {
17            return Vector2.zero; // return a vector with no magnitude = no adjustments made
18        }
19
20        //otherwise get the average of the sum of all the positions of each neighbouring object
21        Vector2 cohesionMove = Vector2.zero;
22        List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);
23        foreach (Transform item in filteredSurroundings)
24        {
25            cohesionMove += (Vector2)item.position;
26        }
27        cohesionMove /= surroundings.Count; // this is the global position
28        cohesionMove -= (Vector2)agent.transform.position; // offset from agents position
29        return cohesionMove;
30    }
31 }

```

Figure A. 3: Cohesion Behaviour

```

6 [CreateAssetMenu(menuName = "Flock/Behaviour/Steered Cohesion")]
7 public class SteeredCohesionBehaviour : FilteredFlockBehaviour {
8 {
9     Vector2 currentVelocity;
10    public float agentSmoothTime = 0.5f; // time taken for an agent to get from its current state to its next state - half a second
11
12    /*
13     * find the new position of an agent based on cohesion behaviour
14     * (move to the mid-point between all neighbouring agents)
15     */
16    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
17    {
18        //No need to make any adjustments if there are no neighbouring objects
19        if (surroundings.Count == 0)
20        {
21            return Vector2.zero; // return a vector with no magnitude = no adjustments made
22        }
23
24        //otherwise get the average of the sum of all the positions of each neighbouring object
25        Vector2 cohesionMove = Vector2.zero;
26        List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);
27        foreach (Transform item in filteredSurroundings)
28        {
29            cohesionMove += (Vector2)item.position;
30        }
31        cohesionMove /= surroundings.Count; // this is the global position
32        cohesionMove -= (Vector2)agent.transform.position; // offset from agents position
33
34        // smooth the flickering motion of;
35        cohesionMove = Vector2.SmoothDamp(current: (Vector2)agent.transform.up, target: cohesionMove, ref currentVelocity, agentSmoothTime);
36        return cohesionMove;
37    }

```

Figure A. 4: Steered Cohesion Behaviour

```

5 [CreateAssetMenu(menuName = "Flock/Behaviour/Composite")]
6 public class CompositeBehaviour : FlockBehaviour
7 {
8     public FlockBehaviour[] behaviours; // flock behaviours to be composited together
9     public float[] weights; // weights of each behaviour
10    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
11    {
12        /* Handle data mismatch;
13         * if the two arrays are not of the same length, highlight the object and keep flock in a stationary position */
14        if (weights.Length != behaviours.Length)
15        {
16            Debug.LogError(message: "Data mismatch in " + name, context: this); // name of scriptable object, and highlight the object
17            return Vector2.zero;
18        }
19
20        Vector2 move = Vector2.zero;
21        //iterate through behaviours using for instead of foreach because we need to refer to the same index of data
22        for (int i = 0; i < behaviours.Length; i++)
23        {
24            if (agent.callStop)
25            {
26                return Vector2.zero;
27            }
28
29            Vector2 partialMove = behaviours[i].CalculateMove(agent, surroundings, flock) * weights[i];
30            //confirm the partial move is limited to extent of weight
31            if (partialMove != Vector2.zero)
32            {
33                if (partialMove.sqrMagnitude > weights[i] * weights[i])
34                {
35                    partialMove.Normalize();
36                    partialMove *= weights[i];
37                }
38                move += partialMove;
39            }
40        }
41        return move;
42    }
43 }

```

Figure A. 5: Composite Behaviour

```

6 //Because its a scriptable object, we need a way to create it:
7 [CreateAssetMenu(menuName = "Flock/Behaviour/Heat Track")]
8 public class HeatTrackBehaviour : FilteredFlockBehaviour
9 {
10     // Frequently called (20+2 usages) Jumalra Miller *
11     public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
12     {
13         List<Transform> surroundingFlames = new List<Transform>();
14         //identify objects which are within a given agents heat sensor's radius
15         Collider2D[] heatSensorColliders = Physics2D.OverlapCircleAll((Vector2) agent.transform.position, flock.heatSensorRadius);
16
17         foreach (Collider2D colliderObject in heatSensorColliders)
18         {
19             if (colliderObject != agent.AgentCollider)
20             {
21                 surroundingFlames.Add(colliderObject.transform);
22             }
23
24             //maintain current direction if there are no neighbouring objects
25             if (surroundingFlames.Count == 0)
26             {
27                 return agent.transform.up;
28             }
29
30             //Otherwise get the sum of distances to each flame object within the heat sensor's range distance
31             Vector2 heatTrackMove = Vector2.zero;
32             float totalAgentHeat = 0;
33
34             List<Transform> filteredSurroundings =
35                 (filter == null) ? surroundingFlames : filter.Filter(agent, surroundingFlames);
36
37             foreach (Transform flame in filteredSurroundings)
38             {
39                 Vector2 dirToFlame = (Vector2) (flame.position - agent.transform.position).normalized;
40                 float distToFlame = Vector2.Distance(a: (Vector2) agent.transform.position, b: (Vector2) flame.position);
41
42                 // if agent is too close to flame, set flag for stopping condition to 'true'
43                 if (Mathf.Abs(distToFlame) <= 1f)
44                 {
45                     agent.callStop = true;
46                 }
47
48                 // otherwise compute move to seek flame
49                 RaycastHit2D pathToFlame = Physics2D.Raycast(origin: (Vector2) agent.transform.position, direction: dirToFlame);
50                 Debug.DrawRay(start: agent.transform.position, dir: (Vector3) (dirToFlame * distToFlame), Color.red);
51                 // heat at drone is 300/d^2 where 300 is the average temperature of tree to ignite
52                 totalAgentHeat += 300.0f / Mathf.Pow(f: distToFlame, p: 2.0f);
53                 heatTrackMove += ((300.0f / Mathf.Pow(f: distToFlame, p: 2.0f)) * dirToFlame);
54             }
55
56             return heatTrackMove;
57         }
58     }
59 }

```

Figure A. 6: HeatTrackBehaviour

```

5 //Because its a scriptable object, we need a way to create it:
6 [CreateAssetMenu(menuName = "Flock/Behaviour/Sight Track")]
7 public class SightTrackBehaviour : FilteredFlockBehaviour
8 {
9     // Frequently called 0+2 usages Jumaira Miller
10    public override Vector2 CalculateMove(FlockAgent agent, List<Transform> surroundings, Flock flock)
11    {
12        //maintain current direction if there are no neighbouring objects
13        if (surroundings.Count == 0)
14        {
15            return agent.transform.up;
16        }
17
18        List<Transform> filteredSurroundings = (filter == null) ? surroundings : filter.Filter(agent, surroundings);
19
20        Vector2 sightTrackMove = Vector2.zero;
21        //find flame
22        foreach (Transform flame in filteredSurroundings)
23        {
24            Vector2 dirToFlame = (Vector2) (flame.position - agent.transform.position).normalized;
25            float distToFlame = Vector2.Distance(a: (Vector2) agent.transform.position, b: (Vector2) flame.position);
26
27            if (Mathf.Abs(distToFlame) <= 1f)
28            {
29                agent.callStop = true;
30            }
31
32            RaycastHit2D pathToFlame = Physics2D.Raycast(origin: (Vector2) agent.transform.position, direction: dirToFlame);
33            Debug.DrawRay(start: agent.transform.position, dir: (Vector3) (dirToFlame * distToFlame), Color.green);
34            sightTrackMove += ((300.0f / Mathf.Pow(f distToFlame, p: 2.0f)) * dirToFlame);
35        }
36        return sightTrackMove;
37    }
38 }

```

Figure A. 7: SightTrackBehaviour

```

85  /*
86  * Update is called once per frame; this is where we iterate through each agent to
87  * apply behaviours based on the context of its surrounding
88  */
89  Event function  Jumaira Miller
90  void Update()
91  {
92      timer += 1 * Time.deltaTime; // increment timer
93      if (SystemStoppingCondition())
94      {
95          if (continueTimer)
96          {
97              Debug.Log(message: "Search Time: " + timer);
98              continueTimer = false;
99          }
100      }
101      foreach (FlockAgent agent in flock)
102      {
103          // List of objects surrounding a given agent in the flock
104          List<Transform> surroundings = GetNearbyObjects(agent);
105
106          // calculate move based on surrounding objects
107          Vector2 move = behaviour.CalculateMove(agent, surroundings, flock: this);
108          move *= forceFactor;
109
110          // if move exceeds the maximum speed then cap it (set it to) at maxSpeed
111          if (move.sqrMagnitude > squareMaxSpeed)
112          {
113              move = move.normalized * maxSpeed;
114          }
115          agent.Move(move);
116      }
117  }

```

Figure A. 8: Flock script's Update method

Appendix B

Supplementary Data

Scenario 1					
Formation:		Circle		Line	
Tracking Behaviour:		Sight	Heat	Sight	Heat
Flock Size	Test #				
5	1	6.76	6.44	0.95	0.93
	2	1.08	7.61	0.94	0.93
	3	1.19	6.41	0.94	0.93
	4	0.81	0.98	0.94	0.93
	5	7.9	1.24	0.93	0.98
	6	13.48	0.72	0.95	0.93
	7	0.95	7.54	0.95	0.96
	8	0.71	0.97	0.95	0.94
	9	6.94	7.71	0.96	0.93
	10	0.96	0.96	0.95	0.96
30	1	1.38	1.48	3.88	4.04
	2	1.33	1.4	4.01	3.72
	3	1.36	1.32	4.01	4.02
	4	1.28	1.34	4.06	3.98
	5	1.37	1.35	3.93	3.93
	6	1.41	1.41	4.04	3.83
	7	1.23	1.42	3.96	3.94
	8	1.32	1.35	3.95	4.01
	9	1.38	1.37	3.93	3.95
	10	1.38	1.27	4	4.33
100	1	14.07	0.7	10.47	10.46
	2	12.97	0.702	10.48	10.46
	3	2.23	0.71	26.2	20.36
	4	13.99	0.71	53.09	10.45
	5	2.22	0.75	10.45	10.45
	6	1.73	0.71	10.47	10.49
	7	1.96	0.7	19.39	20.21
	8	1.99	0.71	10.46	10.42
	9	1.68	0.71	10.49	10.44
	10	2.43	0.71	10.45	10.42

Table 1: Scenario 1 Raw Results

Scenario 2					
Formation:		Circle		Line	
Tracking Behaviour:		Sight	Heat	Sight	Heat
Flock Size	Test #				
5	1	70.5	26	21.34	31.13
	2	10.97	2.07	36	24.222
	3	1.86	12.00007	25.21	25
	4	2.6	2.08	24.43	24.4
	5	2.28	8.35	36.6	24
	6	11.88	56.88	36.05	24.5
	7	44.2	2.6	36.65	24.8
	8	62.2	13.52	25.21	24.7
	9	2.11	20.07	24.46	25.04
	10	31.75	8	36.1	24.06
30	1	27.24	25.7	21.56	13.5
	2	12.03	30.18	9.2	26
	3	18.54	14.3	23.4	16
	4	8.54	21.35	8.87	16.16
	5	20.14	35.76	10.123	23
	6	32.669	14.46	14.33	13.7
	7	31.56	14.3	18.0008	13.5
	8	21	22.9	21.985	9.56
	9	14.36	8.5	14.31	16.43
	10	19.6	8.85	13.6	30.056
100	1	13.9	13.35	12.07	11
	2	20.08	11.16	13.48	11.86
	3	12.8	31	16.27	16.69
	4	9.87	19.6	29.24	17.06
	5	8.86	20.2	11.6	33.12
	6	22.42	18.2	11.87	11.6
	7	10.36	9.12	41.99	11.7
	8	10.57	28.8	11.48	11.8
	9	14.29	12	12.48	35.56
	10	21.04	8.67	11.6	78.41

Table 2: Scenario 2 Raw Results

Scenario 3					
Formation:		Circle		Line	
Tracking Behaviour:		Sight	Heat	Sight	Heat
Flock Size	Test #				
5	1	8.5	13	12.3	8.78
	2	2.35	23	12.42	8.7
	3	2.14	38	12.48	8.76
	4	8.64	1.7	12.45	8.66
	5	5.3	12.2	12.48	8.62
	6	22.1	7.6	12.34	8.62
	7	1.7	18	12.32	8.6
	8	81.4	25.5	12.48	8.76
	9	1.8	8	12.48	8.75
	10	55	17.8	12.47	8.6
30	1	21.111	14.8	14.2	12.1
	2	82.2	13.2	12.5	13.9
	3	82.72	58.24	14	11.65
	4	25.45	11.3	23.43	12.73
	5	12.34	38.9	11.93	12.5
	6	12.3	12.64	18.1	18.13
	7	15.96	9.37	12.8	20.22
	8	20.66	27.75	18.6	9.642004
	9	15.38	8.25	15.27	34.76
	10	14.32	14.32	17.15	18.5
100	1	8.2	12.7	11.2	18.7
	2	17.2	12.2	13.2	19.2
	3	10.23	14.15	11.8	15.1
	4	8.2	13.86	48.17	12.6
	5	12.76	19.2	11.064	28.8
	6	13	13.35	20	16.6
	7	19.3	19.2	18.53	61.5
	8	13.2	13.7	11.4	24.75
	9	8.6	12.2	19.2	21.7
	10	10.3	13	21.32	18.2

Table 3: Scenario 3 Raw Results