CSC3094 Dissertation

Sentinel Tactics System: Exploring Modular Behaviour for Real-Time AI

Kyle James Connolly (200982713)

May 2025

‘Master of Computing with Honours in Computer Science with Industrial Placement (Game Engineering)’

Supervisor - **Dr Giacomo Bergami**

Word Count: 12,409 words

Abstract

A common challenge in game development is creating combat effective, real-time AI system to contribute towards the overarching requirement of balance in a collection of systems within a game. Combat effectiveness is expressed through a set of AI behaviours, which is inherently linked to the AI techniques used for the implementation. It is unclear how breaking down a behaviour set into modular components - distributing them across multiple AI opponents - affect AI combat effectiveness compared to a single opponent with the full set of behaviours. This dissertation explores and then evaluates the impact of modular, distributed AI behaviours in real-time games through developing an AI system utilising a Finite State Machine coupled with utility theory. The projects concludes that a distributed AI model can have combat and technical performance benefits compared to a singular AI opponent but notes limitations in the evaluation while discussing opportunities for future work to reinforce the project's findings and reasoning.

Acknowledgements

I would like to thank my supervisor Dr Giacomo Bergami for his guidance during the project's inception and throughout its development, where he provided essential expertise on challenging technical elements to ensure the overall direction was intriguing and the scope remained focused and achievable.

Declaration

I declare that this dissertation represents my own work except where otherwise stated.

1 Introduction

1.1 Context & Motivation

A common challenge in game development is creating a collection of systems within a real-time game that balance skill and challenge required by the player to keep them engaged [11]. Many genres of games use real-time systems, such as the FPS (First Person Shooter) genre. These games often offer real-time combat experiences against AI. It is therefore necessary to create a system to deal with this in such a way to satisfy and contribute towards the overarching requirement of balance. Balance in this context implies that the AI is first combat effective and proficient enough for it then to be balanced to present acceptable levels challenge to the player. Playtesting may find these interactions to be too easy or too difficult and subsequent balancing may be needed, but the aim remains the same: create combat effective AI.

Combat effectiveness can be expressed through the AI technique chosen for implementation (encompassing behaviours, actions and mechanics). The implementation is translated from meaningful game design requirements created for a given project. The technique for implementation that is chosen for a given project is entirely up to the needs of the developers who, for example, may desire AI systems that afford more designer control or offer better dynamic decision making. Existing implementations in games often follow a logic-based approach, or a more dynamic approach.

Another implementation technique is modular AI [25]. Existing explorations into this concept often focus on its impact on development and maintenance, in addition to making more dynamic AI. However, it is unclear how breaking down a behaviour set into modular components - distributing them across multiple AI opponents - affect AI combat effectiveness compared to a single opponent with the entire set of behaviours. This then involves the concept of single agent versus multi-agent systems. Existing implementations suggest that when coordinating as a team multiagent systems have found success in games. Though rather than focusing on creating an effective team, what happens if we take the behaviour meant for a single agent and distribute it among two or more agents? This project focuses on exploring how making these behaviours modular and distributing them to two agents, without any active communication but sharing a common goal affects combat effectiveness versus a single agent, with the aim to evaluate and reason which is the better model.

1.2 Aim & Objectives

Aim:

To analyse and evaluate the impact of modular, distributed AI behaviour in real-time games through developing an AI system and comparing a singular AI opponent with a split/distributed behaviour model, resulting in being able to conclude which is the more effective model.

Objectives:

1. **Assess AI adaptability between AI models**

It is important for AI to be able to adapt to a wide range of scenarios. The adaptability or rigidness for each AI model, may affect the outcome of a combat scenario, especially impacting how predictable these outcomes are. I will assess the adaptability of AI combat behaviour between the single-agent and distributed-agent models by observing their reaction to various

scenarios and by analysing attack-based metrics including attacks used and behaviour transition frequency. How these adaptability factors can be attributed to the system implemented should be used to evaluate if this system is applicable to other real-time games.

2. **Evaluate AI combat effectiveness between AI models**

To assess which model is more effective overall, not only is it necessary to observe which model wins against the other, but also what aspects of behaviours make an AI model more deadly (i.e. does a certain attack appear to be more dominant, does the movement style play a large factor). I will evaluate the combat effectiveness of the single-agent and distributed-agent AI models by analysing data including health remaining, hits landed, lifetime, attack usage, damage output, and frequency of transitions in behaviours over several combat scenarios.

3. **Explore how splitting behaviours between agents affects synergies between behaviours**

This includes evaluating whether certain behaviours become noticeably weaker or stronger when isolated and whether agents can complement each other to create emergent opportunities during combat, as well as if unintended strategies arise. An exploration into this should this result in being able to reason if this can be extended to more AI agents with and expanded set of behaviours. This can be done by observing combat interactions, logging key events, and comparing these observations across a controlled set of combat tests.

4. **Assess technical performance of the single AI agent model compared to the distributed model**

This will involve measuring and comparing computational cost, focusing on CPU usage during runtime. Evaluating this will determine which implementation appears more resource intensive and will be particularly important if both AI models demonstrate comparable effectiveness in combat, enabling reasoning about if either model justifies any additional performance overhead.

1.3 Methodology

1. **Define game design requirements to ensure meaningful, balanced behaviour for combat scenarios**

There is the need to establish game design requirements to create a controlled environment so that AI behaviour can be meaningfully compared. This will involve designing combat mechanics, defining actions (such as movement and attacks), and ensuring that these mechanics provide both AI models with equal opportunities for success. This ensures that any differences observed in AI performance between models can be attributed to the underlying AI model rather than inconsistencies in the game design, where actions and behaviours will be as generic as possible to reason effectiveness and applicability to other real-time games.

2. **Research AI techniques to implement the actions and behaviours required**

A comprehensive review of AI techniques suitable for real-time combat scenarios will be conducted to determine the most appropriate implementation for this project. Existing industry practices and their advantages or limitations will be researched, where the research will inform the structure of both the single AI agent, which has access to all behaviours, and the distributed model, where behaviours are split between agents. This ensures both implementations are built on justified industry proven principles appropriate for this project's scope.

3. **Implement an AI system based on selected technique(s) informed by research**

Resulting from the research into suitable AI techniques, the selected approach will be implemented within Unity. This will involve translating the previously defined game design requirements into a functioning implementation, ensuring that both the single-agent and distributed-agent models are constructed to allow for a fair and controlled comparison. The process will be guided by important factors necessary for success identified during research, including factors such as ability for behaviour distribution, clear ability to control behaviour, potential for combat adaptability, as well as performance considerations.

4. **Set up metrics to track aspects of combat**

To evaluate the AI models effectively, a series of metrics will be implemented within the appropriate scripts to record aspects of each combat scenario. Metrics will include the number and type of attacks used, total damage dealt and received, frequency of behaviour transitions, AI health over time, enabling me to gain deeper insights into the AI's behaviour. CPU usage data will be collected using Unity's profiler window, which will enable to filter CPU usage by different fields including scripts. The collected data will enable quantitative comparisons between the single agent and distributed agent models, supporting arguments in the evaluation of combat effectiveness, adaptability, and technical performance.

5. **Explore methods to dynamically inform AI actions where appropriate**

To enhance the decision-making abilities of both AI models, methods for dynamically informing AI actions will be explored. Different functions will be designed and tested to determine how real-time variables like distance to the player, remaining health, and any cooldowns influence combat behaviour. The impact of these functions on AI combat effectiveness, adaptability, and emergent behaviours will be evaluated. The findings will inform whether dynamically driven decision making offers a clear improvement over a purely logic-based approach. Dependent on how and which behaviours are split and distributed, this might limit how dynamic each AI operating with the distributed system can be.

1.4 Project Proposal Differences

Within my project proposal the key challenge I identified was the implementation of AI (Artificial Intelligence) in modern games development for real-time PvE (Player versus Entity) combat situations - where the AI is referred to as the entity – that provides the optimal balance of entertainment and challenge. This remains as the overarching problem this project aims to provide a solution for, albeit slightly reworded. The skill and challenge found presented through a game's systems with may provide entertainment through offering an engaging experience, and so the project focuses on contributing to the balance of systems in games through skill and challenge, as opposed to entertainment. Entertainment in the context of AI can be highly subjective for players, and in the context of the paper being referenced [11], was incorrect.

The methodology, aim and objectives of this project have changed from the project proposal. This was due to time constraints due to the level of technical expertise required to explore existing AI techniques. To properly solve the problem, this would involve much more investment into researching each key AI technique. The solution proposed exploring how to create a system that maximises control for game design while maximising flexibility to display emergent, less predictable behaviour in such a way so that it has the desirable qualities of being designer friendly whilst being performant. In practical terms, the initial project aim was to combine

advantages of a structured AI system with the flexibility and emergent decision making of GOAP AI. As for implementing GOAP principles, the goal was to implement more dynamic AI. Perhaps not to the extent of GOAP but integrating utility theory may help achieve this and would be more doable given the time constraints. In this way, the initial aim of from the project proposal is still being explored to an extent.

1.5 Dissertation Structure

Following on from the Introduction, the remaining dissertation is organised into four sections. Continuing from the Introduction is the Background Research section which presents an overview of the most valuable factors that help to achieve the project's objectives, and an exploration of notable AI implementation techniques used within the games industry which are evaluated in the context of the project's requirements to determine the best options to move forward with.

The next section is the Technical Work which includes the project's development. This will cover the design for gameplay mechanics, level design, and an overview of the AI design and implementation. The development (implementation) portion of the section will present the topics shown in the design stage, albeit realised and expanded while still being discussed at a higher level, as well as any related key algorithms and problems that required resolving. The testing portion concludes the Technical Work section, presenting all tests conducted and the results.

Proceeding the Technical Work is the Evaluation section. An analysis and evaluation of the test results presented here, in addition to addressing how each project objective was achieved and to what extent, including any relevant data to support this.

Finally, the Conclusion section discusses the project's successes and limitations and any future work that could be done to extend or improve the project. Lastly, there is a reflection on personal growth during the project.

1.6 Repository

All work produced within this project can be found at the GitHub repository in the link below. Please note that the Library folder has been removed but should be regenerated upon opening the project in Unity. To resolve any anticipated errors, please see the README file.

- https://github.com/KyleC-989/CSC3094_Project_200982173

2 Background Research

2.1 Research Approach

My research involved exploring existing literature on different game AI techniques including related literature on multi-agent vs single-agent systems with the aim to improve my understanding of them to decide on which one(s) I could move forward with, critically evaluating them to make the best choice for my requirements. This involved reading research papers, articles, relevant chapters from books, as well as watching informative videos from reputable sources.

2.2 Key Background Sources

2.2.1 Implementation Motivation

As discussed previously, the technique chosen to create the AI can have effect on the overall behaviour of the AI and its performance and so should be chosen based on the needs of the developer(s) [4]. Due to this, I have researched the most common/popular techniques used with the games industry and evaluated them against my most important factors: predictability/adaptability (how static or dynamic is the AIs perceived behaviour – directly influences combat effectiveness), and designer control (how much does the chosen implementation affect this control). In addition, I should I ensure both models remain feasible to develop within the available time. To ensure the scope of the project remains achievable, only a small set of behaviours will be designed, and so any AI technique should enable this and not over complicate the solution.

These factors were chosen because they directly support the aims of this project which attempts to investigate the differences between a single-agent AI model and a distributed behaviour model. Predictability/adaptability is an essential factor to consider in this case as less dynamic behaviour may make my conclusions be too specific to this game/project, and the ideal scenario allows me to reason the outcome for other real-time games. Also, too deterministic implementations could be limiting potential in the case of the single AI agent, as my hardcoded logic may not cover a wide enough range of possibilities during real-time. Designer control is essential to allow me to design and implement fair and balanced combat mechanics that doesn't give one AI model an advantage over the other, as well as enabling me to fairly compare models without relying on random decision making that might obscure differences in a combat performance evaluation.

2.2.2 Multi-Agent Systems vs Single-Agent Systems

Single-agent systems in artificial intelligence involves a single entity/agent making decisions and performing tasks independently [22]. These systems operate with centralised control, where one AI agent makes all the decisions. Advantages of these systems include simplicity, consistency, and easier maintenance. As there is only one agent, developers can focus on designing a single set of algorithms, not needing to concern themselves with coordinating multiple agents. In addition, single-agent systems are consistent as the same logic is applied to every decision, which is beneficial in scenarios where standardised decision-making is critical, Also, single-agent systems can be more resource efficient, requiring less computational power and memory compared to multi-agent systems.

In contrast, multi-agent systems consist of multiple agents that interact and collaborate to solve problems [22]. Each agent can have distinct skills and responsibilities. These systems work well in dynamic environments as agents can adapt to changes collaboratively. Scalability is another benefit where agents can be added to handle increasing demand. Additionally, by dividing the workload among specialised agents, multi-agent systems enable more efficient and resilient systems ensuring that if one agent fails, other agents can adjust and/or take over.

Within the field of game engineering, numerous use cases of multi-agent systems suggest that these systems can yield success for various genres [23, 24] such as simulation, RTS and sport where multi-agent systems contribute to team dynamics and structure, resource management and coordination, and planning. A multi-agent system will be the inspiration for the distributed behaviour model, except where in this case the AI won't be able to communicate since the idea is to split a single agent's behaviour set, not to create an effect team but to explore how this affects combat effectiveness. A multi-agent system with no communication at first appears to be obviously less effective and doesn't present any of the benefits of MAS, however I was intrigued by the AI in the game F.E.A.R. [14] which uses GOAP (Goal Orientated Action Planning). The AI are known to be a great factor in F.E.A.R.'s success [20], contributing to it winning an AI related award in 2005. GOAP is beneficial when dynamic A.I. behaviour is required due to planning, decoupling actions from goals. While offering less control for the designer, the A.I. can exhibit more unique, adaptive, and undetermined behaviour. This creates immersive A.I. combat experiences from the player's perspective, with the AI. The AI can be grouped into squads, where collective behaviour appears as simple or complex, with complex squad behaviours being more of an illusion thanks to dynamic situations coupled with the planning element [13, 14]. In F.E.A.R. AI agents can technically share the same goal without direct communication yet still appear to synergise through the emergent behaviour in combat scenarios. In my distributed behaviour model for this project the AI agents will similarly be assigned a common target, aligning their goals but without explicit coordination. Evaluation will be performed to assess whether these agents can exhibit synergistic behaviours through shared intent.

2.2.3 Modular AI

It was essential to investigate existing solutions to aid or informs my project. As briefly discussed previously, existing explorations into modular AI often focus on its impact on development and maintenance.

Modular AI systems provide flexibility, scalability, and maintainability by structuring AI behaviours as reusable components [25]. This allows developers to easily add, adjust or remove behaviours without reworking the entire system at the same time reducing code repetition and the risk of bugs. Faster development can be enabled through encapsulating code into a human-level concept which can be reused and repurposed throughout a project.

2.2.4 Rejected AI Techniques

The set of behaviours for this project would be small, so in investigating techniques I ruled out a few which I identified as both being overly complex in meeting the project requirements as well as being too time consuming for the time allocated to design and development.

HTNs (Hierarchical Task Networks) can enable the AI to display specific behaviours [12] but can struggle in real-time PvE situations since it cannot respond effectively enough to changes in the environment [7]. A solution to this problem is the topic of a research paper, and places it beyond

the scope for this project. This AI technique seems too complex to deal with such a small set of behaviours.

Tree search algorithms such as Minimax and Monte Carlo Tree Search were ruled out for this project due to potentially greater computational demands with the nature of the real-time combat scenarios within this project. While time and memory allocated to these algorithms can be limited, they can still be more expensive than static evaluation methods [21]. My AI models will operate within a real-time environment and even with a small set of behaviours, applying these techniques could create unnecessary overhead and exceed the project's development time and resource constraints due to overcomplicating the solution. Furthermore, decision trees can become error prone during development as changes to the certain aspects of the game may easily break decision tree logic unless the access to the game state is not carefully protected [5], where such insulation/abstracted may demand more development time dedicated to this. This can be especially notable given that experimentation and lack of experience may be a common factor throughout the course of this project. As such, tree search algorithms were deemed likely unsuitable for the scope of this project.

Evolutionary algorithms are optimisation based and aim to find the best solution in a search space [27]. This project focuses on combat effectiveness and adaptability in a real-time game environment with a small set of behaviours, and so optimisation methods may be overcomplicating the solution. Evolutionary algorithms involve randomisation which can introduce unpredictability and make the combat scenario outcomes less controllable. Since the aim of this project focuses on being able to assess the difference between the two AI models, the inherent randomness of evolutionary algorithms could undermine the controlled nature needed for evaluating the AI.

Machine learning techniques including supervised learning, reinforcement learning, and unsupervised learning offer powerful capabilities for AI systems but may not be appropriate for this project due to a few considerations. Supervised learning requires a dataset with labelled inputs and outputs to train a model [28]. For real-time combat AI this would potentially involve collecting a large amount of data from various combat scenarios for effective training. Gathering this could be time consuming for the project's scope and may detract from the project's main aim. Reinforcement learning involves an agent learning optimal behaviour through trial and error through rewards [29] and requires considerable amount of time to train the agent as they explore various strategies, potentially leading to inconsistent combat effectiveness. Unsupervised learning aims to search for patterns in data without a target output [30], but it may not directly apply to this project where specific combat behaviours need to be executed precisely. The behaviours and combat will be designed explicitly in a way to ensure models are comparable, where one is simply a distributed version of the other. This necessitates specific behaviours. This means reinforcement learning and unsupervised learning using development time to ensure the models are trained well enough to have predictability in combat may detract from the project's main aim.

2.2.5 AI Techniques Exploration

Finite State Machines:

FSMs are composed of a finite number of states where each behaviour is implemented as a state [9]. Transitioning between them is controlled using a trigger which can be an event or condition. There exist numerous variations of FSMs. A hardcoded switch statement centralises

state logic within a conditional or switch block, making it quick to set up but difficult to maintain as complexity increases. The hard-coded state pattern improves on this by using a base state class with individual states inheriting. This distributes transition logic across the states; when transitions trigger occurs, they replace the active state with the new state. Optionally an FSM controller can be created to manage state transitions. The interpreted state pattern is a data-driven FSM, which operates by storing state connections and transition trigger conditions in data files which are read at runtime. This allows FSM structures to be changed without recompiling the code, though adding entirely new behaviours (a new state) or transition trigger this requires additional coding.

Finite State Machines (FSMs) are easy to design, implement, and debug; however, as complexity of the system grows FSMs become increasingly difficult to maintain in general, not scaling well [17] and often leading to predictable behaviour due to their static logic and structure [3]. Hierarchical FSMs address this [17] by having FSMs as states of a higher-level FSM, improving modularity and reducing overall system complexity. HFSMs are organised as collections of states where the collections execute as one cluster, with specific transitions into and out of a cluster [16]. Fuzzy State Machines (FuSMs) further extend FSMs by utilising fuzzy logic (as opposed to binary logic), where states to hold intermediate values instead of being on or off [9]. This enables multiple states to be active simultaneously, reducing the total number of states needed and creating more interesting and less predictable AI, offering greater flexibility to a static system.

Behaviour Trees:

A Behaviour Tree (BT) is a modular system [18] used to model transitions between a finite set of behaviours, organised in a hierarchical tree structure beginning at the root node and traversing via parent and child nodes [26]. Each child node may return one of three statuses - running, success, or failure - at regular intervals. BTs are composed of three node types. Sequence nodes execute child behaviours in order and only succeed if all child behaviours succeed. Selector nodes either select a child node based on defined probabilities or priorities and succeed as soon as one child behaviour succeeds. If the child behaviour fails, the next child in the priority order is selected or the selector fails (in the case of probability selectors). Decorator nodes modify how a single child behaviour acts, such as limiting execution time or the number of times it can run. The BT is useful due to its flexibility, modularity [1, 18] and ease of testing in comparison to FSMs but despite being more scalable and maintainable than FSMs, BTs are still static with limited dynamic adaptability [26].

Goal-Oriented Action Planning:

GOAP (Goal-Oriented Action Planning) involves AI dynamically planning sequences of actions to achieve a given goal based on the current world state [14]. Each action has preconditions and effects. The AI selects actions that combine to reach its objective. This decouples goals from fixed behaviour logic, enabling enemies to react adaptively in dynamic combat situations. Since the AI can plan sequences of actions dynamically instead of following a pre-scripted procedure, this enables both emergent and more strategic long-term behaviours. In F.E.A.R. [13] a replanning system is utilised which involves running validation checks on the current plan to make sure it will satisfy the goals, having a function to continually check whether the current plan should be abandoned, and a replacement be obtained, and a procedure during execution for the plan to be validated again forcing a re-plan if not satisfied. This replanning system ensures the AI is reactive and adaptable to sudden and notable events.

Utility AI:

Utility theory operates with the concept that possible actions or states within a system can be described with a single, uniform value called utility representing how beneficial an action can be within a given context [8]. This enables the AI to evaluate different possible actions by scoring them with utility and to then choose one of the top scoring actions. Ensuring fair comparison between utility scores is an important element and so scores must be kept on a consistent scale across the system. This is usually achieved through normalisation with scores being converted to a standard range such as 0 and 1.

Where environments are often nondeterministic, evaluating a single, exact utility score doesn't always describe the best outcome and it is more desirable to guess the expected utility of an action. The principle of maximum expected utility often involves multiplying the utility of each action by the probability of each possible outcome and summing these weighted values to calculate expected utility.

Decisions rarely rely on one single piece of data. Instead, multiple decision factors are often used, where each one can be modified with weights determine how much the AI considers that factor which reflect the AI's personality. The utility of each factor is calculated and combined, usually by averaging if the values are normalised which form the final utility score for a decision. Calculating the initial utility score often involves mapping input values to utility scores through mathematical curves. Curves such as linear, quadratic, and the logistic function define how the AI perceives the desirability of different inputs, directly shaping its decision making.

Once utility scores have been calculated, various methods can be used to select an action with the simplest being just choosing the highest scoring option, though this can lead to predictable behaviour. An alternative is to use utility scores as and randomly choose one of the actions based on the weights, introducing more varied selection but occasionally causing irrational decision making. A balanced approach involves taking a subset of the highest-scoring actions and choosing one of them with a weighted random selection from that group. Another technique is bucketing, where actions are grouped into categories each with a defined priority, with higher priority buckets evaluated first.

Frantic or oscillating behaviour may be caused by highly frequent decision making if actions are scored closely and are recalculated quickly. Solutions to this can involve adding weight to an ongoing action, adding cooldown periods after decisions, or stalling until a current action is completed or a cooldown expires.

2.3 Research Application & Justification

While exploring methods of enhancing dynamic behaviours in planning, I started investigating fluents. As discussed in [15], fluents can be used to track the world state over time; [2] explains the planning problem, describing it as an initial state (comprised of predicates), actions (comprised of parameters, preconditions and effects), and goals. Fluents can be represented by predicates (in first-order logic), or functions when predicates have been converted. While this combination (fluents with GOAP) could potentially enable highly adaptive AI by dynamically tracking world state and making informed decisions over time, it introduces additional complexity for managing distributed behaviours across multiple agents. This project involves comparing a single-agent model with a distributed multi-agent model. Implementing this system risks obscuring the differences in AI combat effectiveness with the planner's emergent

decisions, which could make evaluation of individual behaviours and their impact more difficult since they could possibly change from scenario to scenario.

Existing AI implementation approaches, such as BTs, provide a clear and modular method to implementing behaviours. However, BTs can introduce predictable behaviours over time as they are ultimately deterministic [4]. Existing implementations attempt to enhance BT adaptability by giving priority to certain actions in particular scenarios [6], though this falls short of the long-term strategic and dynamic actions as observed with GOAP. Enhancements to BTs have been explored to reduce some of these issues. In [10] integrating utility theory enhances selection logic and avoids duplicating sections of the tree for one or more behaviours. This is beneficial for ensuring a more performant BT, also enabling the AI to be more adaptable and reactive by making more informed choices.

Given my planned relatively small behaviour set and need for clear behaviour separation - especially when distributing behaviours across agents - a BT may unnecessarily complicate behaviour organisation. In this context FSMs offer clearer and more deliberate control over behaviour distribution when splitting behaviours across multiple agents. FSMs, require explicitly defined state transitions and behaviours, in this case making things easier to assign, track, and isolate individual behaviours during distribution. BTs operate as hierarchical structure. Splitting parts of a BT across multiple agents could either require duplicating parts of a tree or carefully managing interdependencies, potentially overcomplicating development by attempting to make the BT compatible with my project's aims, detracting from the main work.

FSMs offer clear separation of behaviours and high amounts of control over AI logic and should be easy to implement when managing a small set of states, aligning with my project's scope and time constraints. FSMs offer designers full control over which behaviours can occur, avoiding the risk of overly emergent or unexpected AI actions obscuring combat outcome evaluation. FSMs can become harder to maintain as complexity increases [17] and can result in predictable behaviours due to fixed transition rules. HFSMs are useful for managing a larger set of behaviours but would likely be overcomplicating the solution given my limited behaviour set. FuSMs on the other hand may reduce clarity in the separation of behaviours, potentially making it difficult to distribute behaviours across agents as required for this project. I concluded I should use a hardcoded state pattern FSM [9] because as it shows clear separation of behaviours by structuring them into states. This makes it easier to assign and split behaviours between agents.

The FSMs limitations in dynamic decision making can be mitigated by integrating utility theory to influence transitions or action priorities within states. While integrating utility theory is a documented process in GOAP and BT systems, it can also be effectively combined with FSMs. Though seemingly less widely documented in game AI, sources suggest this pairing is compatible and useful for enhancing FSM adaptability [19]. By incorporating utility functions, this transforms the static FSM into a more adaptable and strategic system, improving the AI's overall combat effectiveness.

3 Technical Work

3.1 Methodology Overview

To achieve the project's aim, I decided to develop a Unity based, real-time action game prototype utilising an FSM, with two AI models for comparison: a single AI opponent possessing the full set of combat behaviours, and group of two AI opponents, each handling a subset of behaviours, forming a collective strategy. The two models will fight each other and against their own implementation to produce replicable and results. The project's success will hinge on firstly designing and implementing an FSM, where the states and contained behaviours are translated from the game design, in addition to designing utility functions and the level design. Secondly being able to effectively evaluate the two models using metrics that measure different areas of combat within the context of game design such as attack frequency, as well as more technical aspects including tracking state transitions and agent lifetime.

3.2 Design

3.2.1 Mechanics & Behaviour Design Overview

To develop combat-effective AI, I first needed to design meaningful game mechanics that serve the objectives of this project while being as widely applicable as possible to other games. These mechanics would shape the behaviours of the AI agents while also serving as the foundation for exploring distributed behaviours through an FSM.

The initial step involved identifying core AI behaviours and experimenting with splitting a single behaviour into modular components that could be distributed which aligns with the project's overarching aim to investigate distributed AI behaviours in real-time combat scenarios. The clearest and most easily measurable approach would be to separate combat behaviours into distinct attack and movement styles, distributing them between two AI agents. I decide two agents was the minimum viable number of agents for testing the distributed behaviour concept while keeping the implementation scope manageable. This would provide a solid and expandable foundation for reasoning about scaling for more agents and more complex modular strategies in future work.

Each AI agent, referred to as a Sentinel Agent, will operate using an FSM which will contain four states: Patrol, Search, Evade, and Combat. After careful consideration I decided these were collectively, core pillars of general combat functionality in AI by ensuring the agent can dynamically navigate the environment, seek targets, engage combat effectively, and manage risk. Each Sentinel Agent will start with 1000 HP (Health Points), and any actions developed concerning the health of an agent should be designed with this in mind.

3.2.2 Sentinel Agent Behaviour States

Patrol State

The agent follows a series of predefined checkpoints while monitoring the environment for the target(s). Upon acquiring line of sight on the target it transitions to the Combat state. This behaviour ensures the AI is active within the level/environment and can respond quickly to enemy presence, contributing to an adaptive combat system by transitioning between passive and aggressive behaviour.

Search State

If the AI loses sight of the target for more than two seconds, it transitions to the Search state where the AI will select random positions within a defined area and navigates to them, effectively covering ground and attempting to reacquire line of sight on its target. If at any point the AI regains line of sight on its target, then transition to the Combat state. This behaviour supports combat by preventing static AI behaviour.

Evade State

The Evade state will involve the AI agent actively attempting to evade its attackers by first picking a hiding spot. When it has reached its destination (hiding spot) then it should check to see if any of its attackers have line of sight on it, if so, pick a new hiding spot and if not then stay in that position. Otherwise, it remains hidden for 20 seconds before transitioning back to Patrol. This mechanic introduces a defensive behaviour into the combat system, breaking up frequent aggression and enabling the AI to reposition and display richer behaviour. The Evade state is limited to a single use per agent per combat scenario to maintain good pacing and prevent overly passive behaviour.

Combat State:
The Combat state should be the most complex behaviour, featuring two distinct attack and movement styles. Each set should be designed to operate well individually or collectively. Both attack types rely on resource management via a system called Elemental Charges (often referred to in the codebase as charges), where each agent starts with 10 charges, regenerating 1 charge per second after a charge is depleted. This system will enable utility-based decision making through resource management.

Attacks:

- Behaviour Set 1: Storm (CQC - Close Quarters Combat)
    - Attack name: “Storm”
    - Charge Cost: 2
    - Max Range: 5 units
    - Damage: 50
    - Cooldown: None
- Behaviour Set 2: Tempest (Medium to Long Range Combat)
    - Attack name: “Tempest”
    - Charge Cost: 3
    - Max Range: 15 units
    - Damage: 75
    - Cooldown: 1.5 seconds

Behaviour Set 1 encourages aggressive, close-range engagements. The AI will attempt to close the distance to the target and fire only when within Storm range. This behaviour mirrors the general role of SMGs in FPS games utilising the concept of close, quick engagements with lower damage per-shot but greater mobility (enabled by the lack of cooldown for fast firing). Behaviour Set 2 encourages more cautious, ranged combat. The AI will fire projectiles from any distance but will only move towards the target when outside its maximum attack range. This behaviour is inspired by the general role of assault rifles in in FPS games, dealing higher damage at longer ranges but with a slower rate of fire, usually more effective at medium to long range.

Both Storm and Tempest attacks are projectile based attacks, each with distinct visual and functional properties. Projectiles will self-destroy on impact or after traveling their maximum range, reinforcing their intended tactical roles. Using projectiles ensures that attack range limits are visually and mechanically enforced, making combat outcomes more clearly evaluable for testing and debugging.

Other abilities:

- Teleport Dodge:
    - To add dynamic defensive capabilities for each Sentinel Agent and to add to the adaptability of the AI overall, they will be equipped with a dodge mechanic. If the agent loses more than 100 health within four seconds, it should attempt a teleport dodge. This will involve searching for an available space to its left, right, or behind. If no valid space is found, the agent remains at its current position.

3.2.3 Modular & Distributed Behaviours

By implementing these modular behaviour sets within the Combat state, this system enables distributing combat roles between two Sentinel Agents. One agent can specialise in aggressive, CQC (Storm), while another maintains ranged pressure (Tempest). The singular agent will dynamically switch between behaviours based on utility scores produced by utility functions considering factors including health, distance, and available charges. The setup of this system facilitates a collective combat strategy formed by individually modular behaviours, which is the central concept this project aims to test and evaluate.

3.2.4 Utility Theory Integration

To create more chances for adaptive decision making for the Sentinel agent, I aimed to integrate utility theory to support action selection during combat scenarios by enabling the AI to assess available actions in current state of the environment and select those most appropriate to the current context based on game state variables.

Utility functions will only be implemented with the singular Sentinel Agent (the agent with the full set of combat behaviours) in mind. The distributed AI pair were excluded as their primary purpose was to test distributed, modular behaviours through the fixed combat roles (behaviours sets 1 or 2); introducing utility driven decision making at this stage may overcomplicate the assessment of the effectiveness of each behaviour as due to the simplicity of each behaviour set there is not much that can influence them to have meaningful effect. Isolating the utility functions to the singular agent would also enable a more controlled test environment to observe the effects of utility based tactical decision making on AI adaptability.

Basic Utility Scores vs Expected Utility

I decided to use basic utility scores instead of expected utility to prioritise simplicity for early development while still preserving extensibility for future development. Given the small set of behaviours, the predictability of combat scenarios would likely not vary too much between each test, though calculating for probability for a real-time scenario would almost certainly yield beneficial results in future work since it isn't just purely reactive as with basic utility scores. In addition, working with more predictable, basic utility scores would allow easier comparison as the results would be less random and would detract less from the focus on assessing the combat effectiveness of the distributed versus singular AI models.

Utility Curves/Formulas

An important step was how to map normalised (between 0 and 1) inputs from variables such as distance to target, health, and resource availability to utility scores. Selecting appropriate utility curves is crucial to produce behaviour that appears natural and tactically appropriate. There are many different formulas to select, so I will be considering a subset of well-established curve types:

- Linear:
    - utility = x/m
    - Can be appropriate when the desirability of an action increases or decreases steadily with a given variable.
- Quadratic:
    - utility = $1 - x^2$
    - Can be appropriate when the desirability of an action declines sharply beyond a threshold.
- The Logistic Function:
    - utility = $1 / (1 + e^{-x})$
    - Can be appropriate when there is the need for a gradual, initial response, then a sharp transition around a midpoint, which then begins to level off again.

Chase Utility

Health was factored into chase distance decision making rather than directly influencing attack selection. The rationale is that the AI should base its aggression primarily on its health affecting its desire to pursue with aggressive behaviour (Storm/behaviour set 1) or maintain distance from the target (Tempest/behaviour set 2).

A linear curve will be utilised here: utility = health / maxHealth. Higher health yields higher aggression (shorter follow/chase distances), while lower health promotes more cautious engagement.

Storm Utility & Tempest Utility

There will be two primary combat behaviours Storm (behaviour set 1) and Tempest (behaviour set 2). Their utility scores will be influenced by two key factors: distance to target and available charges (ammunition), to create more dynamic and more nuanced decisions through aggregation of scores. Distance will mainly determine the desirability of each attack type. The Storm attack has a maximum range of 5 units, while Tempest has a range of 15 units. The AI should favour Storm at close range and Tempest at medium to long range.

Utility scores for both attacks will be calculated as a weighted average of distance utility (weight of 0.7) and charge utility (weight of 0.3), prioritising positioning while still considering resource availability. If the enemy is well positioned for the Tempest attack but charges are low, the utility remains high but slightly reduced. For example, a distance of 11 for the Tempest attack would be 0.81…, but when using the equation: utility = (0.7 * distance utility) + (0.3 * charge utility), with a charge count of 4, the final utility value becomes 0.69…. High amounts of charges on their own do does not justify utilising an attack if the enemy is out of the effective range.

A negative quadratic curve will be ideal for modelling the Storm attack's short-range nature. The utility is highest at zero distance and drops off sharply as distance increases. This shape simulates an aggressive CQC play style.

The logistic curve was selected for Tempest because it allows for smooth prioritisation of an ideal engagement distance.

- utility = 1 / (1 + e^(-k(x - x_0)))
- x is the distance to the target
- x_0 (midpoint) = 10 (midpoint between Storm max range and Tempest max range)
- k (steepness) = 1.5

The parameterised form of the logistic function will be used to enable control over where the utility value transitions most sharply (x_0) and how steeply it changes (k). This was necessary to adjust the AI's desirability for using the Tempest attack within its optimal engagement distance. Without parameterisation the function's transition point and slope would be fixed. This implementation should produce realistic behaviour where the AI favours the Tempest attack when within its ideal range, avoiding over committing at less optimal distances.

The figures below present tables that show the utility scores of the each identified factor over their total ranges.

Health	***Utility Score (normalised)***
0	0.00
100	0.10
200	0.20
300	0.30
400	0.40
500	0.50
600	0.60
700	0.70
800	0.80
900	0.90
1000	1.00

Figure 1: Chase Utility - Health Factor (Linear)

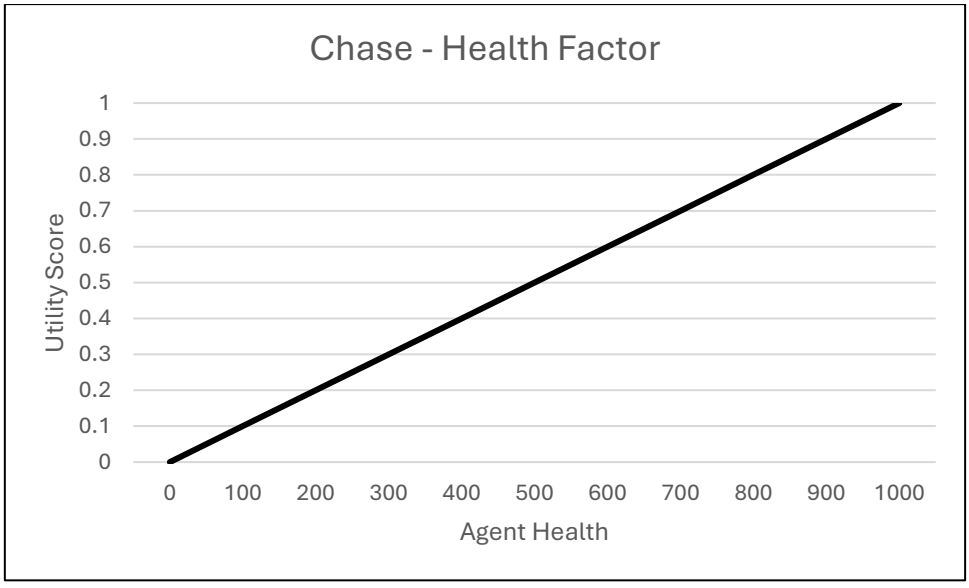

Figure 2: Chase Utility - Health Factor Graph

Distance	*Normalised Value*	*Utility Score*
0	0.00	1.000
1	0.20	0.960
2	0.40	0.840
3	0.60	0.640
4	0.80	0.360
5	1.00	0.000

Figure 3: Storm Utility - Distance Factor (Negative Quadratic)

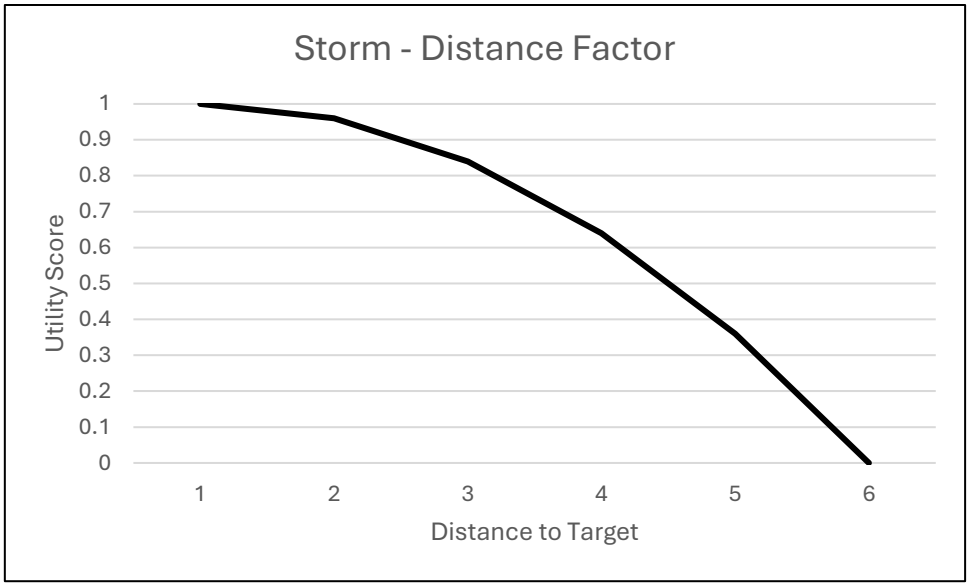

Figure 4: Storm Utility - Distance Factor Graph

Distance	*Utility Score*
0	0.000000306
1	0.000001371
2	0.000006144
3	0.000027536
4	0.000123395
5	0.000552779
6	0.002472623
7	0.010986939
8	0.047425869
9	0.182425528
10	0.5
11	0.817574472
12	0.952574127
13	0.989013059
14	0.997527379
15	0.999447205

Figure 5: Tempest Utility - Distance Factor (Logistic, Midpoint = 10, k = 1.5)

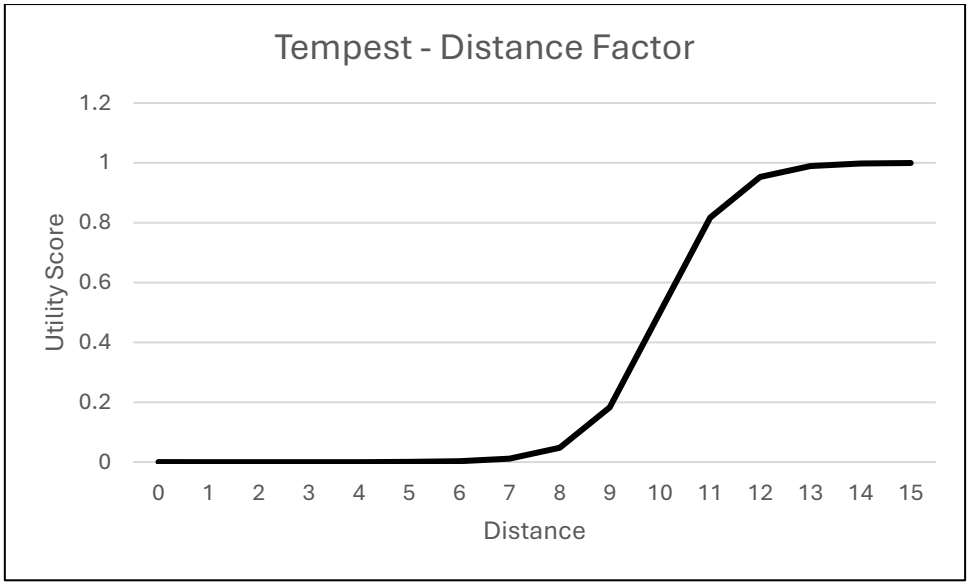

Figure 6: Tempest Utility - Distance Factor Graph

Charges	***Utility Score (normalised)***
0	0.00
1	0.10
2	0.20
3	0.30
4	0.40
5	0.50
6	0.60
7	0.70
8	0.80
9	0.90
10	1.00

Figure 7: Shared Utility - Elemental Charge Factor (Linear)

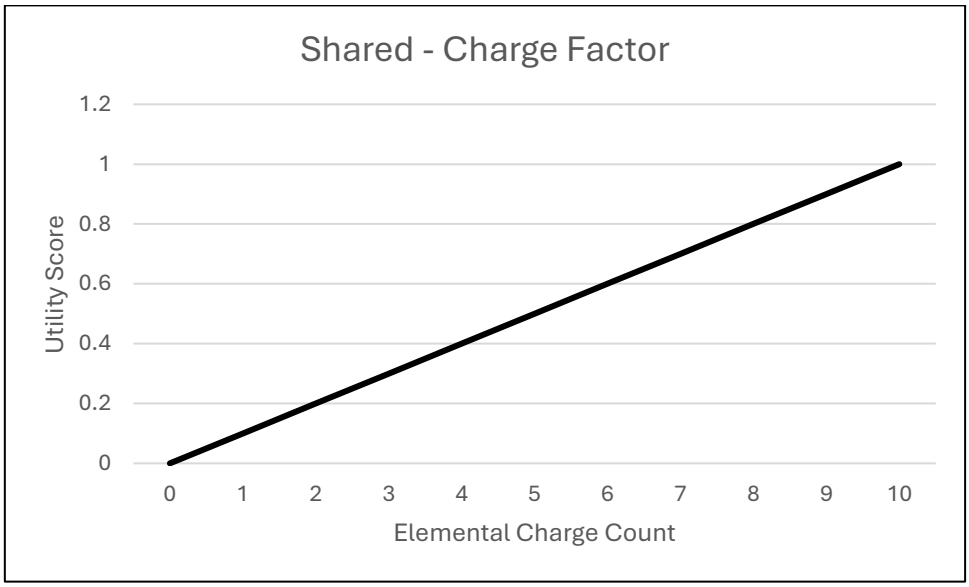

Figure 8: Shared Utility - Elemental Charge Factor Graph

3.2.5 System Design

The AI system within this project will be implemented using a hardcoded state pattern FSM. The main AI object will be implemented as the class SentinelAgent, which will inherit from Enemy. This ensures separation of logic for future development and experimentation with the project. The Enemy class will hold functionality and data that could be expected to apply to most enemy types. The FSM will operate using individual states encapsulated as objects, each state implementing a shared interface. The AI controller will manage state transitions, delegating relevant logic to the active state. Below lists specifications for the planned interface, controller, states, and supporting classes.

EnemyStateMachineController

- Acts as the controller for managing state transitions and updating the currently active state.
- Methods:
    - EnemyStateMachineController() – constructor initialising the controller with a reference to the associated Enemy (Sentinel Agent).
    - ChangeState() - transitions to the given new state and triggers its EnterState method.
    - Update() - delegates update behaviour (through UnityEngine.MonoBehaviour which is called every frame) to the active state through UpdateState().

Blackboard

- A class which each Sentinel Agent will hold an instance of for storing relevant Boolean flags between states. EvasionComplete indicates whether a one-time evasion behaviour has completed. The AI should be transitioning back to the combat state which will hold no previous information. The evade state should only be run once per combat scenario so it is necessary to enable the AI to remember if it has already completed evasion. CombatCycleActive - flags whether a combat cycle is currently active. To stop the combat state(s) UpdateState method from being called frequently that could cause oscillation [9] or jittery behaviour from the concept of inertia [8].

IEnemyState (Interface)

- Defines the interface that all states should implement which ensures consistency across different state implementations.
- Methods:
    - EnterState() - initialises the state with a reference to the associated Enemy/Sentinel Agent (agents inherit/are a subclass of Enemy).
    - UpdateState() - called every frame (from the controller) to process the state's logic.

SentinelPatrolState

- Manages patrol behaviour for Sentinel Agents.
- Methods:
    - EnterState() - initialises patrol variables and checkpoint list. Casts Enemy to Sentinel Agent type.

    - UpdateState() - handles target acquisition checks and transitions to combat states if a target is detected, otherwise continues patrolling.
    - Patrol() - moves the agent to the next patrol checkpoint in the list.

SentinelCombatState (Abstract class)

- Abstract class providing shared functionality and related data for combat states.
- Methods:
    - EnterState() - initialises combat state variables such health and cooldown timers. Casts Enemy to Sentinel Agent type.
    - UpdateState() - virtual and will be overridden by inheriting classes to implement combat logic specific to that state (different states for the singular and distributed models).
    - ChaseTarget() – moves the agent towards the target dependent on the type of distributed agent or the utility score in the case of the singular agent.
    - TeleportDodge() - attempts to teleport the agent to a safe nearby location when under heavy attack in a short period of time.
    - SentinelAttack() – determines which projectile to create based on attack type.
    - CreateProjectile() - instantiates and launches projectile towards the target based on calculated trajectory.

SplitSentinelCombatState (Inherits from SentinelCombatState)

- A specialised combat state for split (distributed behaviour) Sentinel agents. Some functionality will only be available depending on the behaviour set the agent variant should have access to.
- Methods:
    - UpdateState()
        - Monitors health to trigger Evade state transition when health drops to 250HP or less.
        - Manages cooldowns and dodge teleports under specific conditions.
        - Selects appropriate attack and chase type (Storm or Tempest) based on variant, range to target, and available charges.
        - If agent loses line of sight on its current target for more than two seconds transition to Search state.

USentinelCombatState (Inherits from SentinelCombatState)

- A specialised combat state for the singular Sentinel Agent that using a utility-based decision-making system to choose attacks and chase distance, representing the full set of combat functionality.
- Methods:
    - UpdateState()
        - Monitors health to trigger Evade state transition when health drops to 250HP or less.
        - Manages cooldowns and dodge teleports under specific conditions.
        - Selects appropriate attack and chase type (Storm or Tempest) based on utility score from utility functions

        - If agent loses line of sight on its current target for more than two seconds transition to Search state.

SentinelUtility

- Should provide static methods for calculating utility scores to support decision-making for the AI. These utility scores influence combat choices and chase behaviours based on factors including distance, health, and available charges.
- Methods:
    - CalculateStormDistanceUtility() - uses a quadratic curve ($1 - x^2$) that should have higher score when distance to target is low to prioritise targets at close range.
    - CalculateTempestDistanceUtility() applies the logistic function ($1 / (1 + e^{-k(x - x_0)})$) with a higher score when distance to target is ideal (tempest range midpoint).
    - CalculateChargeUtility() - linear curve-based utility score calculation for managing resources (elemental charges) with normalisation of charge level returning a value between 0 (no charges) and 1 (max charges).
    - CalculateStormUtilityScore() - calculates final Storm utility score using weighted utility aggregation, with 0.7 weighting to distance and 0.3 to charge level.
    - CalculateTempestUtilityScore() - calculates final Tempest utility score using weighted utility aggregation similarly weighting distance and charge levels.
    - CalculateHealthUtilityScore() - linear curve-based utility score calculation for assessing health of the AI.
    - CalculateChaseDistanceUtilityScore() - calculates final health utility score to influence aggression during chase behaviours, with higher health promoting more aggression and so closer follow/chase distance.

SentinelSearchState

- Handles searching for the target(s) when line of sight has been lost on the current target.
- Methods:
    - EnterState() - picks an initial random search position. Casts Enemy to Sentinel Agent type.
    - UpdateState() - if the target is spotted again, transitions back to Combat state. If the agent reaches its current search position, it picks a new one.
    - PickRandomPosition() - selects a random position within a specified radius. Sets this position as the new destination. Retry up to a predefined number of attempts if no position found.

SentinelEvadeState

- Temporarily forces the Sentinel to actively run from its attackers and hide.
- Methods:
    - EnterState(Enemy enemy) - increases agent's speed and acceleration for faster evasion. Picks an initial hiding position. Casts Enemy to Sentinel Agent type.
    - UpdateState() - increments the evasion timer. If the evasion duration expires it marks evasion as complete on that agent's blackboard and restores agent's original speed and acceleration, then transitions to either

SplitSentinelCombatState or USentinelCombatState based on the agent's tag. If hiding and the agent reaches its destination it faces the target. If hiding and the player has line of sight it picks a new hiding position.

- SelectHidingPosition() - picks a random position within a large radius and sets the agent's destination to this hiding spot.
- HasTargetLineOfSight() - checks if any of the agent's targets has an unobstructed view (raycast) toward the agent.

FSM Diagram

Below is a diagram to visualise the transitions between state, and trigger conditions:

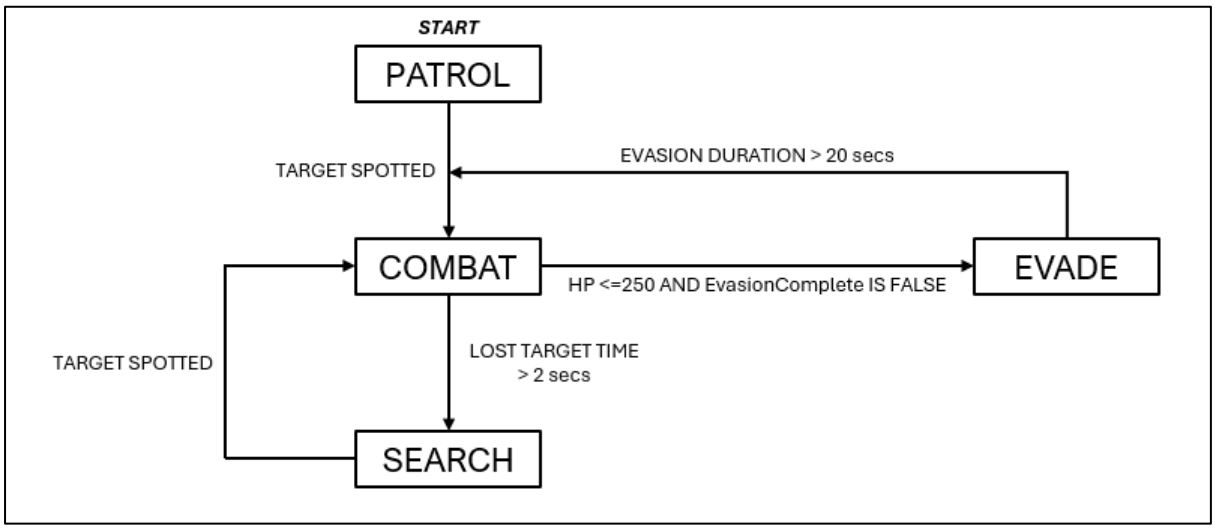

Figure 9: Visual of proposed FSM transitions and triggers

3.2.6 Supporting Algorithms

Contained with the Enemy class, which the SentinelAgent will inherit from, is list of targets, and a variable to hold the current target. These will be used to check line of sight to and from the targets for different purposes. AcquireTarget will search through the list of targets and the first one seen will be set to the current target. CanSeeTarget will verify if the Sentinel Agent can still see its current target. Both will use ray casting to verify line of sight.

3.2.7 Level Design

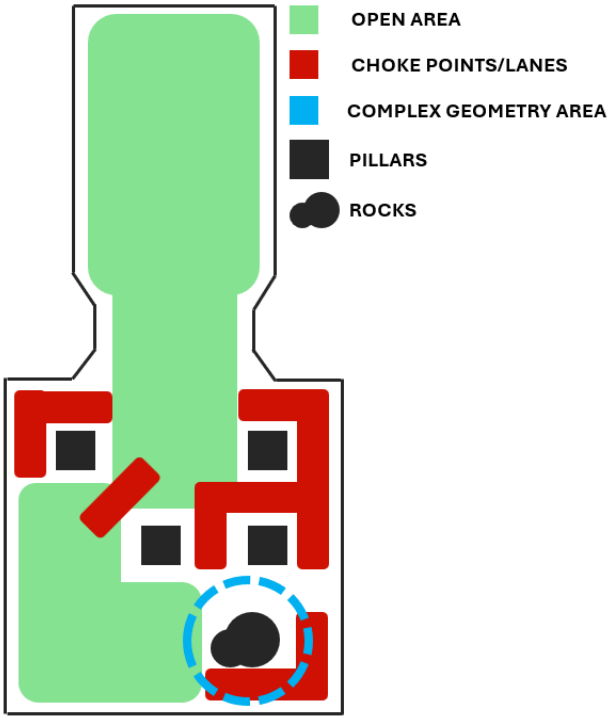

Figure 10: Top view of proposed level design with keys

The level/environment has been deliberately designed to include a variety of spatial features to evaluate different elements of AI behaviour and decision making. Each region within the level serves a specific purpose in testing and validating AI functionality, observing AI reactions across a range of combat, navigational situations ensuring that all core AI systems are tested against appropriate spatial challenges.

Green: Open Areas

- These wide, unobstructed zones are ideal for assessing the AI's navigation, chasing, and ranged combat behaviours in open conditions without environmental interference. They allow for clear line of sight engagement in addition to testing chase logic, attack range evaluation, and decision making at various distances.

Red: Choke Points & Narrow Lanes

- These are confined corridors and bottlenecks designed to test the AI's pathfinding adaptability under more constrained conditions. They are also effective for forcing engagements, observing how AI agents handle tight spaces, and evaluating how well the AI selects alternate paths when faced with limited options. It also allows for the easier testing of evasion or aggressive behaviours in close quarters encounters.

Blue: Complex Geometry

- This region introduces irregular geometry and obstacles (in this case rocks) to test AI line of sight functionality with raycasting, evaluating whether the AI correctly respects environmental occlusion when determining whether a target is visible or not.

3.3 Development

This part of the Technical Work section contains work related to the implementation of the work completed in the design phase, contributing towards the project's objectives and main aim. This part of the Technical Work section contains snippets of code, which for readability purposes is often simplified and is not a direct reference to the actual code is some cases.

3.3.1 Third-Party Assets

The table below contains every third-party asset used in the development phase of this project. Each link provides the source for each asset and contains relevant information such as the original creator/author. The credit for each asset belongs to their respective owners. The following links provides information on usage rights/licences:

- https://unity.com/legal/as-terms
- https://www.textures.com/about/terms-of-use

Category	***Asset Name***	***Source (URL)***	***Usage***
Code/Functionality	StarterAssets	https://assetstore.unity.com/packages/essentials/starter-assets-firstperson-updates-in-new-charactercontroller-pa-196525	Added new methods in StarterAssetsInputs.cs and FirstPersonController.cs to include new movement features for a first-person controller (to observe the AI efficiently)
Code/Functionality	Water Surface Ripple Effect	https://github.com/Parrot222/Unity-Water-Shaders/blob/main/Assets/Scripts/RippleEffect.cs https://github.com/Parrot222/Unity-Water-Shaders/blob/main/Assets/Shader/RippleShader.shader https://github.com/Parrot222/Unity-Water-Shaders/blob/main/Assets/Shader/Add.shader https://www.youtube.com/watch?v=DFwNv1n1Y6Q&t=370s https://www.youtube.com/watch?v=U7pH5lIPZOU	To create the ripple effect in the realistic body of water
Models	Ancient Ruins and Plants	https://assetstore.unity.com/packages/3d/props/exterior/ancient-ruins-and-plants-201914	Used the P_Ancient_Ruins_Sword prefab for environment object and M_Ancient_Ruins_Rocks (Material) - for sword platform material

Models	Boss Of War | Assets	https://assetstore.unity.com/packages/3d/environments/boss-of-war-assets-86108	cube_stone, wall, wall_stone, big_stone, aqueduct - for environment objects
Models	Simple Water Shader URP	https://assetstore.unity.com/packages/2d/textures-materials/water/simple-water-shader-urp-191449	WaterBlock_50 - for realistic water texture
Models	Rock and Boulders 2	https://assetstore.unity.com/packages/3d/props/exterior/rock-and-boulders-2-6947	Rock1_grup1, Rock1A, Rock2, Rock3, Rock4A, Rock4B, Rock5A, Rock5B, Rock6A, Rock6B, Rock6C - for environment
Models	Stylized Rocks with Magic Rune	https://assetstore.unity.com/packages/3d/props/stylized-rocks-with-magic-rune-192933	StylRocksMagic_LOD0 for environment objects
Particles/Effects	Cartoon FX Remaster Free	https://assetstore.unity.com/packages/vfx/particles/cartoon-fx-remaster-free-109565	CFXR3 Magic Aura A (Runic).prefab - for AI teleporting ability visual. CFXR4 Bounding Glows Bubble (Blue Purple).prefab - for environment magic particle system
Particles/Effects	Particle Pack	https://assetstore.unity.com/publishers/1	WildFire, TinyFlames, FireBall - fire particle systems
Particles/Effects	Unity Tutorial - Shoot Projectiles in FPS	https://www.youtube.com/watch?v=T5y7L1siFSY&t=41s	Inspiration for the Tempest Projectile and mechanics
Textures	Free Volcanic Rock Cliff Material - PBR0188	https://www.textures.com/download/free-volcanic-rock-cliff-material-pbr0188/133289	All materials apart from ambient occlusion - for ground texture
Textures	Starfield Skybox	https://assetstore.unity.com/packages/2d/textures-materials/sky/starfield-skybox-92717	Skybox - for the skybox
Textures	Too Many Crosshairs	https://assetstore.unity.com/packages/2d/gui/icons/too-many-crosshairs-126069	Attack.png - for crosshairs (UI)

Figure 11: Table of third-party assets

3.3.2 First-Party Assets

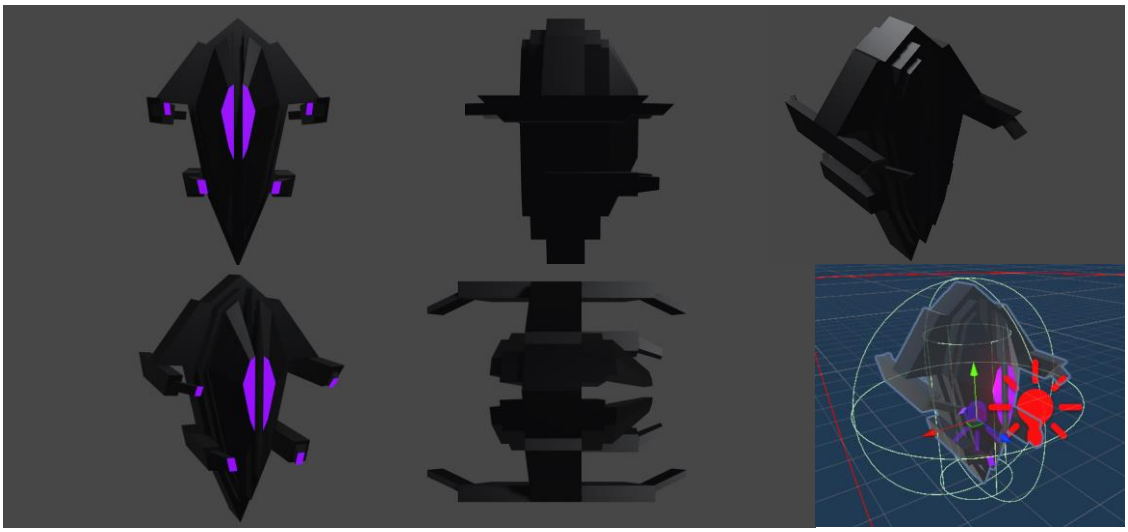

Figure 12: Different views of the Sentinel Agent 3D model

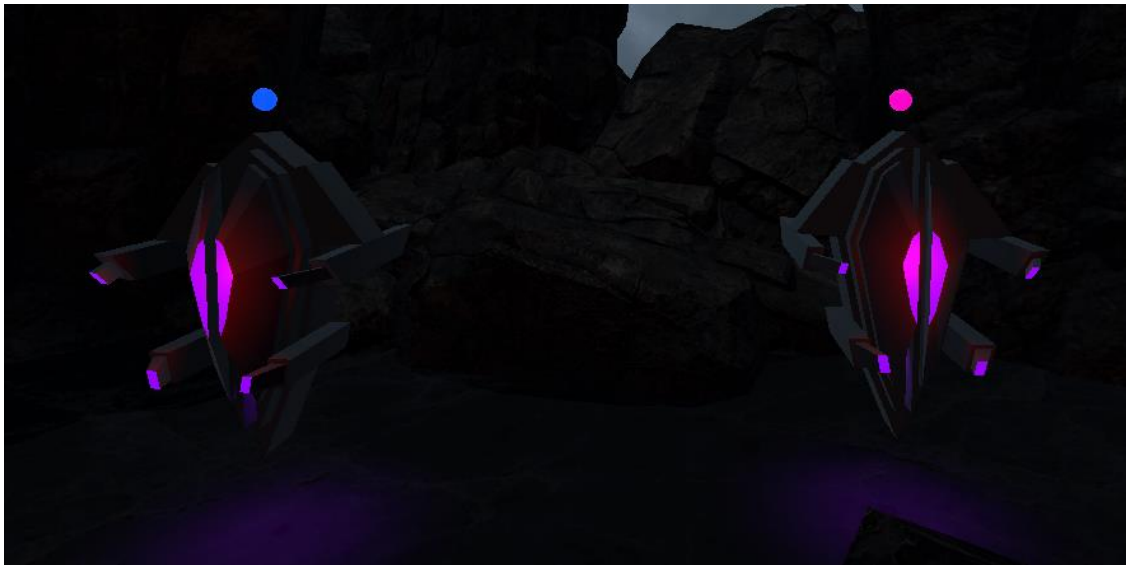

Figure 13: The Storm (blue) and Tempest (pink) variants of the split/distributed AI model

Objects	1 / 3
Vertices	252 / 252
Edges	498 / 498
Faces	248 / 248
Triangles	510 / 510

Figure 14: 3D model statistics in Blender

The above figures show the Sentinel Agent model created using Blender 4.2.0. The bottom right image within Figure 4 shows the capsule collider applied to the model in Unity. I decided to keep the model simple with areas of high contrast for easy visual recognition and a relatively low polygon count to not impact performance significantly (Figure 14). For the two Sentinel Agents belonging to the distributed model, I added markers above to able to differentiate them (Figure 13). Blue represents the Storm variant, with pink/purple representing the Tempest variant.

3.3.3 Level & NavMesh

After acquiring all assets, as seen in the figures below, I recreated my level design in Unity.

Figure 15: Screenshots of the level in Unity

To simplify development of the AI system to focus on the main objectives, I decided to utilise Unity's NavMesh package for AI pathfinding. This involved created a NavMesh surface for the NavMeshAgent (the Sentinel Agent) to traverse on. This surface defines all surfaces that the AI use to navigate the environment. A notable art of this process involved assigning Nav Mesh Obstacle components to objects around the level such as the boundaries, pillars, and the rocks. This component contains a field called 'Carve' (Figure 18). This enables an adjustable area around that specific object that it is applied to in order affect how close the NavMesh can be to the model, directly affecting how close the AI can be when traversing around this object. In turn this then ensures that the AI does not get stuck on any geometry and when performing its various functions. Below are screenshots of different angles of the level which illustrate the entire process. The blue areas show the NavMesh surface, also presenting the carved areas around objects (Figure 18).

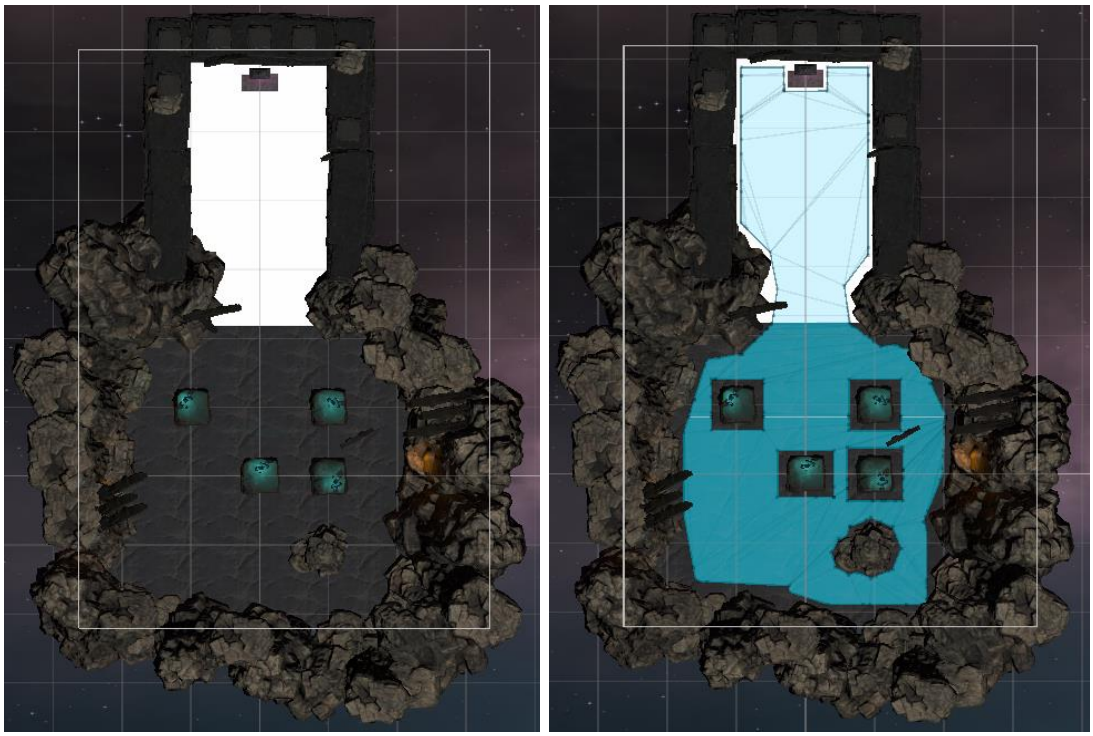

Figure 16: Top view of the level with the right image showing the NavMesh area

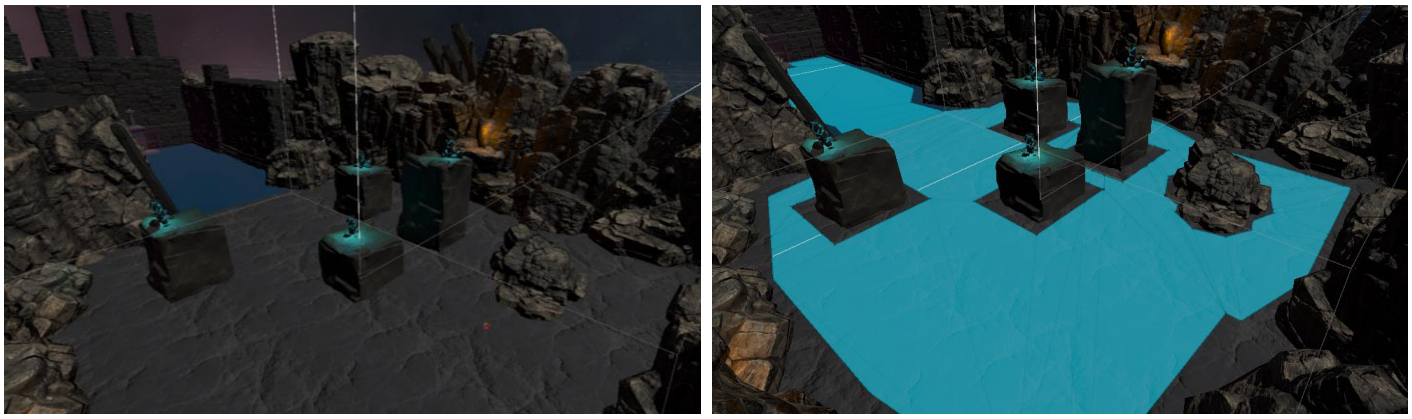

Figure 17: Alternate views of the level and NavMesh surface

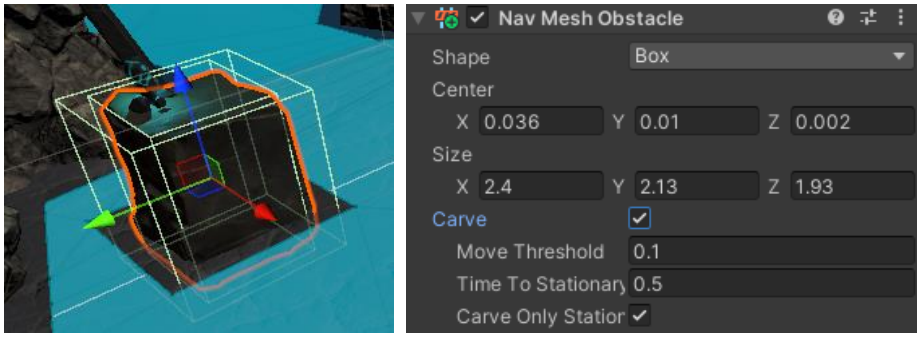

Figure 18: NavMesh Obstacle component being added to the pillar object, with carve enabled

3.3.4 Projectile Setup

Before creating the attacks for the Sentinel Agents, I needed to explore how to create the projectile. To do this, I developed the Tempest projectile from a first-person perspective. This was done before implementing any attacks for the AI to ensure the projectile physics worked and wasn't affected by any potential issues with the AI.

This involved utilising the FirstPersonController script (see Figure 11), creating a prefab for the projectile object and exploring how to fire it. Taking inspiration from the Unity Tutorial (Figure 11), I first created an empty game object, placing it slightly in front of the character model to act a point to fire from, then creating the projectile prefab along with an impact prefab. An additional sphere collider component was added with it set to be a trigger, which will used to check if it has collided with the any of the agents when fired as an attack. I then worked out how to instantiate the projectile prefab from the firing point game object I made previously, setting the velocity. Finally, I added script to the projectile prefab to destroy itself after a certain period, as required by the game design defined in the design phase.

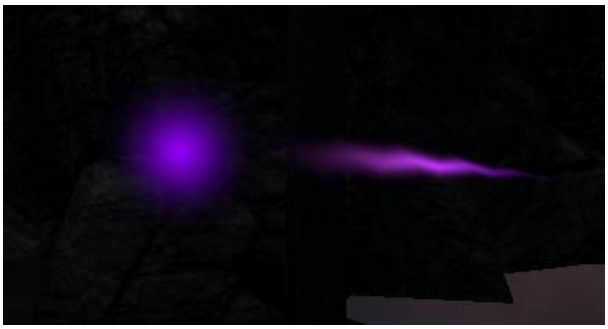

Figure 19: The Tempest projectile in flight after being fired

Figure 20: The Tempest projectile impact after travelling the maximum distance

```
public void CreateProjectile()
{
    var projectile = Instantiate(tempestProjectile, projectileOrigin.position, Quaternion.identity) as GameObject;
    projectile.GetComponent<Rigidbody>().velocity = (_target - projectileOrigin.position).normalized * tempestSpeed;
}
```

Figure 21: Initial code to create the projectile and set its velocity

```
public GameObject impactPrefab;
protected Vector3 _startPosition;
protected float _travelDistance = 0f;
private float _tempestMaxRange = 15f;

protected void Start()
{
    _startPosition = transform.position; //set to initial spawn position to assess distance travelled
}

private void Update()
{
    _travelDistance = Vector3.Distance(_startPosition, transform.position);

    if (_travelDistance >= _tempestMaxRange)
    {
        CreateImpact(); //method to destroy the projectile and create the impact effect
    }
}
```

Figure 22: Projectile code shows how the projectile will destroy itself after travelling the max distance

I also used this as an opportunity to modify StarterAssetsInputs.cs and FirstPersonController.cs to include new movement features. These files concern the movement of a first-person character, using keyboard and mouse as an input to traverse and look around. Included an increased jump height and 'boost' mechanic which lets the user jump higher (separate input to the existing jump mechanic) and speed up movement for short period respectively. This was to observe the AI efficiently via improved traversal, which was essential of quickly moving to other parts of the level or jumping on top of the pillars to observe the AI interactions.

3.3.5 FSM Interface & Controller

As detailed in the design phase, I implemented the hardcoded state pattern FSM with an interface (Figure 23) for creating more states (Figure 25) and a state controller (Figure 24). The Enemy class holds and instance of this controller to access the FSM, with the SentinelAgent class that inherits from it changing the state to the specified one upon starting the game (see Figure 18).

```
public interface IEnemyState
{
    void EnterState(Enemy enemy);
    void UpdateState();
}
```

Figure 23: Interface code for states

```
public class EnemyStateMachineController
{
    private IEnemyState _currentState;
    private Enemy _enemy;

    public EnemyStateMachineController(Enemy enemy)
    {
        this._enemy = enemy;
    }

    public void ChangeState(IEnemyState newState)
    {
        _currentState = newState;
        _currentState.EnterState(_enemy);
    }

    public void Update()
    {
        _currentState?.UpdateState();
    }
}
```

Figure 24: FSM controller code

```
public class ExampleState : IEnemyState
{
    private SentinelAgent _sentinelAgent;

    public void EnterState(Enemy enemy)
    {
        this._sentinelAgent = (SentinelAgent)enemy;
    }

    public void UpdateState()
    {
        //do something
    }

    //other methods
}
```

Figure 25: An example of a state implementing the interface

```
public abstract class Enemy
{
    public EnemyStateMachineController stateMachine;

    //Start is called before the first frame update
    protected virtual void Start()
    {
        stateMachine = new EnemyStateMachineController(this);
    }

    //Update is called once per frame
    protected virtual void Update()
    {
        stateMachine.Update();
    }

    //other methods
}

public class SentinelAgent : Enemy
{
    //Start is called before the first frame update
    protected override void Start()
    {
        base.Start();
        stateMachine.ChangeState(new ExampleState());
    }

    //Update is called once per frame
    protected override void Update()
    {
        base.Update();
        //call other methods
    }

    //other methdods
}
```

Figure 26: Enemy and Sentinel Agent class example code

3.3.6 Combat States Implementation

As explored in the design section, there would be two combat states, one for the singular AI, and the other for the split/distributed pair. To avoid code duplication, these two states would inherit from a parent combat state, holding all shared functionality including chase, teleport dodge, attack, and projectile creation/calculation. The figures below show the methods implemented.

```
protected void ChaseTarget(bool isStorm)
{
    NavMeshAgent agent = _sentinelAgent.GetNavMeshAgent();

    _sentinelAgent.SetDestinationLocked(true);

    //if the input chase type is the storm type (close quarters chasing) then set the chase/follow distance to the storm distance
    //else set it to the tempest distance
    float followDist = isStorm ? _sentinelAgent.GetStormRange() : _sentinelAgent.GetTempestRange();

    Vector3 directionToTarget = (_sentinelAgent.GetTargetAgent().transform.position - _sentinelAgent.transform.position).normalized;

    //Target position + attack range
    Vector3 targetPosition = _sentinelAgent.GetTargetAgent().transform.position - directionToTarget * followDist;

    //Move only if AI is out of range of the attack
    if (Vector3.Distance(_sentinelAgent.transform.position, _sentinelAgent.GetTargetAgent().transform.position) > followDist)
    {
        agent.SetDestination(targetPosition);
        agent.isStopped = false;
    }
    else
    {
        agent.isStopped = true;
        agent.ResetPath();
    }
}
```

Figure 27: Chase code

```
//Try dodging by searching for space left, right, or behind. If no space found then do nothing.
protected void TeleportDodge()
{
    Vector3 currentPosition = _sentinelAgent.transform.position;
    float teleportDistance = 2.5f;
    float radius = 2.0f;

    //Available Teleport directions
    Vector3[] directions = new Vector3[]
    {
        _sentinelAgent.transform.right, //Right
        -_sentinelAgent.transform.right, //Left
        -_sentinelAgent.transform.forward //Backwards
    };

    //Shuffle directions list to make sure AI doesn't dodge too predictably
    for (int i = 0; i < directions.Length; i++)
    {
        int randomIndex = Random.Range(i, directions.Length);
        Vector3 tempDirection = directions[i];
        directions[i] = directions[randomIndex];
        directions[randomIndex] = tempDirection;
    }

    foreach (var dir in directions)
    {
        Vector3 teleportSpace = currentPosition + dir * teleportDistance;

        //Check for valid space to teleport
        if (NavMesh.SamplePosition(teleportSpace, out NavMeshHit hit, radius, NavMesh.AllAreas))
        {
            isTeleporting = true;

            _sentinelAgent.GetNavMeshAgent().Warp(hit.position + new Vector3(0, 1.5f, 0));

            //Teleport visual
            var warpEffect = UnityEngine.Object.Instantiate(_sentinelAgent.GetTeleportPrefab(),
_sentinelAgent.GetNavMeshAgent().transform.position + new Vector3(0, -1.7f, 0), Quaternion.identity);
            UnityEngine.Object.Destroy(warpEffect, 1f);

            //Return since successful teleport dodge
            return;
        }
    }
}
```

Figure 28: Teleport dodge code

```
private void CreateProjectile(Vector3 target, bool isStormAttack)
{
    GameObject projectile;

    //create the Storm or Tempest version of the projectile depending on if isStormAttack is true or not
    if (isStormAttack)
    {
        projectile = GameObject.Instantiate(_sentinelAgent.GetStormProjectile(), _sentinelAgent.GetProjectileOrigin().position,
Quaternion.identity);
    }
    else
    {
        projectile = GameObject.Instantiate(_sentinelAgent.GetTempestProjectile(), _sentinelAgent.GetProjectileOrigin().position,
Quaternion.identity);
    }

    Rigidbody body = projectile.GetComponent<Rigidbody>();

    //Calculate direction + velocity
    Vector3 projectilePosition = _sentinelAgent.GetProjectileOrigin().position;
    Vector3 targetDistance = target - projectilePosition;
    float distance = targetDistance.magnitude;

    //Get height difference
    float heightDiff = target.y - projectilePosition.y;

    //Set initial launch angle
    float angle = Mathf.Deg2Rad * 20;
    float gravity = Physics.gravity.y;
    float velocityMagnitude = Mathf.Sqrt(distance * Mathf.Abs(gravity) / Mathf.Sin(2 * angle));

    //Launch velocity vector
    Vector3 horizontalDirection = new Vector3(targetDistance.x, 0, targetDistance.z).normalized;
    Vector3 launchVelocity = horizontalDirection * velocityMagnitude * Mathf.Cos(angle);
    launchVelocity.y = velocityMagnitude * Mathf.Sin(angle);

    //Calculated velocity to projectile rigid body
    body.velocity = launchVelocity;
}
```

Figure 29: Code for creating the Storm or Tempest projectile and calculations for trajectory

After this was implemented, I began work on the two inheriting combat states (see figures on following pages). As discussed in the design phase, each state belongs to each of the two models being developed for this system. For the split/distributed model, the UpdateState() method locks functionality with conditional statements to either the Storm or Tempest variant, so that they only have access to Behaviour Set 1 or 2 respectively.

```
public class USentinelCombatState : SentinelCombatState
{
    public override void UpdateState()
    {
        //check if health is below 50, transition to Evade state once
        if (_sentinelAgent.GetHealth() <= 250 && !_sentinelAgent.blackboard.EvasionComplete)
        {
            _sentinelAgent.stateMachine.ChangeState(new SentinelEvadeState());
            return;
        }

        if (_sentinelAgent.CanSeeTarget() && !_sentinelAgent.blackboard.CombatCycleActive)
        {
            _sentinelAgent.blackboard.CombatCycleActive = true; //set to true so that update method won't run this again mid cycle

            distanceToTarget = Vector3.Distance(_sentinelAgent.transform.position, _sentinelAgent.GetTargetAgent().transform.position);

            tempestDistMidpoint = (stormRange + tempestRange) / 2f;

            float stormScore = SentinelUtility.CalculateStormUtilityScore(distanceToTarget, stormRange, charges, maxCharges);
            float tempestScore = SentinelUtility.CalculateTempestUtilityScore(distanceToTarget, tempestDistMidpoint, charges, maxCharges);

            //Select attack based on utility scores
            if (stormScore > tempestScore && charges >= 2)
            {
                SentinelAttack(true); //true for storm attack
            }
            else if (tempestScore > stormScore && charges >= 3 && tempestCooldownTimer <= 0f)
            {
                SentinelAttack(false); //false for tempest attack
                tempestCooldownTimer = tempestCooldown;
            }

            //check if too much damage was taken in the last 4 seconds
            if (Time.time - healthCheckTimer >= healthCheckTime)
            {
                float health = _sentinelAgent.GetHealth();
                float damage = previousHealth - health;

                if (damage >= damageThreshold)
                {
                    TeleportDodge();
                }
            }

            float chaseScore = SentinelUtility.CalculateChaseDistanceUtilityScore(health, maxHealth, distanceToTarget, tempestRange);

            if (chaseScore >= 0.5f)
            {

                ChaseTarget(true); //aggressive chase for Storm (close range) attack
            }
            else
            {
                ChaseTarget(false); //more conservative chase for Tempest (long range) attack
            }

            _sentinelAgent.blackboard.CombatCycleActive = false; //set to false as combat cycle has completed so this section of code can run again
        }

        //lost enemy SentinelAgent so need to search but only after a certain period of time has passed
        if (!_sentinelAgent.CanSeeTarget())
        {
            if (lostTargetTimer >= waitBeforeSearch)
            {
                _sentinelAgent.stateMachine.ChangeState(new SentinelSearchState());
            }
        }
    }
}
```

Figure 30: Simplified code for the single AI model's combat state using utility scores

```
public class SplitSentinelCombatState : SentinelCombatState
{
    public override void UpdateState()
    {
        //check if health is below 50, transition to Evade state once
        if (_sentinelAgent.GetHealth() <= 250 && !_sentinelAgent.blackboard.EvasionComplete)
        {
            _sentinelAgent.stateMachine.ChangeState(new SentinelEvadeState());
            return;
        }

        if (_sentinelAgent.CanSeeTarget() && !_sentinelAgent.blackboard.CombatCycleActive)
        {
            _sentinelAgent.blackboard.CombatCycleActive = true;

            distanceToTarget = Vector3.Distance(_sentinelAgent.transform.position, _sentinelAgent.GetTargetAgent().transform.position);

            //Storm attack if within Storm range and Sentinel is the Storm variant (GetSentinelVariantType is true for storm variant)
            if (_sentinelAgent.GetSentinelVariantType() && distanceToTarget <= _sentinelAgent.GetStormRange() && charges >= 2)
            {
                SentinelAttack(true); //true for storm attack
            }

            //Tempest attack if within Tempest range and Sentinel is the Tempest variant
            if (_sentinelAgent.GetSentinelVariantType() && tempestCooldownTimer <= 0f && charges >= 3)
            {
                SentinelAttack(false); //false for tempest attack
                tempestCooldownTimer = tempestCooldown;
            }

            //Check if too much damage was taken in the last 4 seconds
            if (Time.time - healthCheckTimer >= healthCheckTime)
            {
                float health = _sentinelAgent.GetHealth();
                float damage = previousHealth - health;

                if (damage >= damageThreshold)
                {
                    TeleportDodge();
                }
            }

            if (_sentinelAgent.GetSentinelVariantType())
            {
                ChaseTarget(true); //aggressive chase for Storm (close range) attack
            }

            if (!_sentinelAgent.GetSentinelVariantType())
            {
                ChaseTarget(false); //more cautious chase for Tempest (long range) attack
            }

            _sentinelAgent.blackboard.CombatCycleActive = false;
        }

        //lost enemy SentinelAgent so need to search but only after a certain period of time has passed
        if (!_sentinelAgent.CanSeeTarget())
        {
            lostTargetTimer += Time.deltaTime;

            if (lostTargetTimer >= waitBeforeSearch)
            {
                _sentinelAgent.stateMachine.ChangeState(new ASentinelSearchState());
            }
        }
    }
}
```

Figure 31: Simplified code for the split/distributed AI model's combat state

3.3.7 Utility Functions

The figure on the following pages shows the implementation of the utility functions detailed in the design phase. The functions make use of Mathf.Clamp01 for normalisation between 0 and 1, and Mathf.Exp for returns a number raised to a power.

```
//Quadratic curve based utility calculation - (1 - x²). Higher score when distance to target is low.
public static float CalculateStormDistanceUtility(float distance, float maxDistance)
{
    float distanceRatio = Mathf.Clamp01(distance / maxDistance);
    return 1f - (distanceRatio * distanceRatio);
}

//Logistic Function based utility calculation - 1 / (1 + e^(-k(x - x0))). Higher score when distance to target is ideal (tempest range midpoint)
public static float CalculateTempestDistanceUtility(float distance, float midpoint)
{
    float k = 1.5f; //curve steepness/slope
    return 1f / (1f + Mathf.Exp(-k * (distance - midpoint)));
}

//Linear curve based utility score calculation for managing resources (elemental charges/ammunition)
//Normalisation of charge level. Returns a value between 0 (no charges) and 1 (max charges).
public static float CalculateChargeUtility(int currentCharges, int maxCharges)
{
    return Mathf.Clamp01((float)currentCharges / maxCharges);
}

//Final Storm attack utility score using weighted utility aggregation. Weighting of 0.7 for distance since more important than charges
public static float CalculateStormUtilityScore(float distance, float maxDistance, int charges, int maxCharges)
{
    float distanceUtilityScore = CalculateStormDistanceUtility(distance, maxDistance);
    float chargeUtilityScore = CalculateChargeUtility(charges, maxCharges);
    return 0.7f * distanceUtilityScore + 0.3f * chargeUtilityScore;
}

//Final Tempest attack utility score using weighted utility aggregation
public static float CalculateTempestUtilityScore(float distance, float optimalMid, int charges, int maxCharges)
{
    float distanceUtilityScore = CalculateTempestDistanceUtility(distance, optimalMid);
    float chargeUtilityScore = CalculateChargeUtility(charges, maxCharges);
    return 0.7f * distanceUtilityScore + 0.3f * chargeUtilityScore;
}

//Linear curve based utility score calculation for assessing health of the AI.
public static float CalculateHealthUtilityScore(float health, float maxHealth)
{
    return Mathf.Clamp01(health / maxHealth);
}

//Final Chase utility score.Higher health -> more aggressive and so closer follow/chase distance
public static float CalculateChaseDistanceUtilityScore(float health, float maxHealth, float distance, float tempestRange)
{
    float healthUtility = CalculateHealthUtilityScore(health, maxHealth);
    return healthUtility;
}
```

Figure 32: Utility calculation code

3.3.8 Patrol, Search, & Evade States

The figures on the following pages present simplified code for the Patrol, Search, and Evade state implemented as part of the FSM.

```
public class SentinelPatrolState : IEnemyState
{
    private SentinelAgent _sentinelAgent;
    private Transform[] _checkPointList;
    private int _checkPoint;

    public void EnterState(Enemy enemy)
    {
        this._sentinelAgent = (SentinelAgent)enemy;
        this._checkPointList = _sentinelAgent.GetPatrolCheckPoints();
    }

    public void UpdateState()
    {
        if (_sentinelAgent.AcquireTarget())
        {
            _sentinelAgent.stateMachine.ChangeState(new SentinelCombatState()); //return to appropriate combat state based on model
        }
        Patrol();
    }

    private void Patrol()
    {
        NavMeshAgent agent = _sentinelAgent.GetNavMeshAgent();
        _checkPoint = _sentinelAgent.GetPatrolCheckpoint();

        if (!_sentinelAgent.GetDestinationLocked() && agent.remainingDistance < 0.1f) //continue patrolling if no target lock on
        {
            //Move to next checkpoint. Return to start at the end
            _sentinelAgent.SetPatrolCheckpoint((_checkPoint + 1) % _checkPointList.Length);
            agent.SetDestination(_checkPointList[_checkPoint].position);
        }
    }
}
```

Figure 33: Simplified Patrol state code

```
public class SentinelSearchState : IEnemyState
{
    private SentinelAgent _sentinelAgent;
    private Vector3 _searchPosition;
    private bool _foundDestination = false;

    public void EnterState(Enemy enemy)
    {
        this._sentinelAgent = (SentinelAgent)enemy;
        PickRandomPosition();
    }

    public void UpdateState()
    {
        if (_sentinelAgent.CanSeeTarget())
        {
            _sentinelAgent.stateMachine.ChangeState(new SentinelCombatState());
            return;
        }

        if (_foundDestination && !_sentinelAgent.GetNavMeshAgent().pathPending) //agent arrived at the location
        {
            _foundDestination = false; //set to false to try again
        }

        if (!_foundDestination) //haven't found a place to move to so need to search
        {
            PickRandomPosition();
        }
    }

    //mimics searching for the target by selecting random positions to move the agent to
    protected void PickRandomPosition()
    {
        float radius = 8.0f; //search radius
        int attempts = 10; //maximum attempts

        for (int i = 0; i < attempts; i++)
        {
            //generate random direction within radius
            Vector2 randomDirection = Random.insideUnitCircle * radius;
            Vector3 newPosition = _sentinelAgent.transform.position + new Vector3(randomDirection.x, 0, randomDirection.y);

            NavMeshHit navHit; //check if random position is on NavMesh
            if (NavMesh.SamplePosition(newPosition, out navHit, radius, NavMesh.AllAreas))
            {
                //distance is far enough - avoids jitter
                if (Vector3.Distance(_sentinelAgent.transform.position, navHit.position) > 4.0f)
                {
                    _searchPosition = navHit.position;
                    _sentinelAgent.GetNavMeshAgent().SetDestination(_searchPosition);
                    _foundDestination = true;
                    return;
                }
            }
        }

        _foundDestination = false;
    }
}
```

Figure 34: Simplified Search state code

```
public class SentinelEvadeState : IEnemyState
{
    public void EnterState(Enemy enemy)
    {
        _evasionTimer = 0f;
        _originalSpeed = _agent.speed;
        _originalAcceleration = _agent.acceleration;
        _agent.speed *= _speedIncrease;
        _agent.acceleration *= _accelerationIncrease;
        SelectHidingPosition();
    }

    public void UpdateState()
    {
        _evasionTimer += Time.deltaTime; //track the amount of time that has passed, updated every frame

        //Stop evasion after 25 seconds
        if (_evasionTimer >= _evasionDuration)
        {
            _sentinelAgent.blackboard.EvasionComplete = true; //mark as true so agent won't evade again
            _agent.speed = _originalSpeed; //reset speed
            _agent.acceleration = _originalAcceleration; //reset acceleration

            _sentinelAgent.stateMachine.ChangeState(new SentinelCombatState()); //return to appropriated combat state based on model

            return;
        }
        //agent has reached hiding spot
        if (!_isHiding && !_agent.pathPending)
        {
            _isHiding = true;
        }
        //agent is hiding but one of its enemies can see the agent, so find new hiding spot
        if (_isHiding)
        {
            if (HasTargetLineOfSight())
            {
                _isHiding = false;
                SelectHidingPosition(); //keep selecting a new hiding spot while the enemies can see the agent
            }
        }
    }

    private void SelectHidingPosition()
    {
        Vector3 randomDirection = UnityEngine.Random.insideUnitSphere * 30f;
        randomDirection += _sentinelAgent.transform.position;

        if (NavMesh.SamplePosition(randomDirection, out hit, 30f, NavMesh.AllAreas))
        {
            _agent.SetDestination(hit.position); //found viable hiding spot, set destination
            _isHiding = false;
        }
    }

    private bool HasTargetLineOfSight()
    {
        var targets = _sentinelAgent.GetTargets();
        Vector3 sentinelPosition = _sentinelAgent.transform.position;
        //for each of the agent's enemies need to check if they have line of sight on the agent
        foreach (var target in targets)
        {
            Vector3 targetPosition = target.transform.position;
            Vector3 direction = sentinelPosition - targetPosition;

            //If there's a clear line of sight from the target to the sentinel
            if (Physics.Raycast(targetPosition, direction.normalized, direction.magnitude))
            {
                return true;
            }
        }
        return false;
    }
}
```

Figure 35: Simplified Evade State code

3.3.9 Challenge: Search Algorithm

During development I noticed that combat scenarios would start of highly predictable since each Sentinel Agent has a patrol path. The way the nodes are placed on the level appeared to give one AI the advantage allowing the Sentinel Agent to see their target marginally sooner than the other. In addition, with the distributed AI, they would sometimes get stuck on each other since they were patrolling the same route. To fix this I decided the best decision would be to create a more even start and have all AI utilise the Search state to search the environment.

While observing the AI I noticed that the way in which an agent would search the environment was illogical, checking and moving to places such as behind itself as opposed to exploring what was ahead. I wanted to explore having the AI move more realistically. I concluded that to achieve this, the AI should search each search point (node) in the area that it can see before thinking of new search areas based on what it can see after its current search finishes. To explore this approach, I decided to use the A* algorithm.

A* is a pathfinding algorithm that finds the shortest route between a start and end goal by combining the actual cost to reach a node (g) and an estimated cost from that node to the goal (h). This calculated as f = g + h. The algorithm explores paths with the lowest total f score first, and continues exploring nodes until it reaches the goal, guaranteeing an optimal path. Work for the main A Star algorithm was completed utilising my past experiences of studying and applying the algorithm to a previous project, in which I had a grid-based system and used the Manhattan distance heuristic.

To apply this to my project, my approach was to search an area around the Sentinel Agent and create a list of nodes representing points within that area, where nodes are only added to the list if the AI can see that node (has line of sight) - see Figure 36; then randomly chose an end node from the list and then construct shortest path to that node via the A Star algorithm (Figure 37). I chose the Euclidean distance as the heuristic because it is the most appropriate and efficient method for estimating the straight-line cost between two points in open, 3D spaces with the ability of omnidirectional movement such as with Unity's NavMesh. Calculating Euclidean distance would involve a call to Vector3.Distance() that directly calculates that Euclidean distance. The AI iterates through the list representing the path of nodes, where within the UpdateState method, once the agent has reached the node and index variable is iterated to move the agent onto the next node until it has completed the list at which point the process begins again.

```
//Search a random positions around the SentinelAgent to discover valid nodes to use pathfinding on
private List<Vector3> DiscoverNodes()
{
    List<Vector3> foundNodes = new List<Vector3>();

    for (int i = 0; i < _graphNodeCapacity; i++)
    {
        //Random direction and distance within a circle
        Vector2 searchDir = UnityEngine.Random.insideUnitCircle.normalized * UnityEngine.Random.Range(_minSearchDistance, _maxSearchDistance);

        Vector3 searchPosition = _sentinelAgent.transform.position + new Vector3(searchDir.x, 0, searchDir.y);

        NavMeshHit hit;
        //If a valid NavMesh point within 2 units of the search position can be found
        if (NavMesh.SamplePosition(searchPosition, out hit, 2.0f, NavMesh.AllAreas))
        {
            Vector3 nodePosition = hit.position;

            //If there is clear line of sight from the agent to the discovered node
            if (HasLineOfSight(_sentinelAgent.transform.position, nodePosition))
            {
                foundNodes.Add(nodePosition);
            }
        }
    }
    return foundNodes;
}
```

Figure 36: Method to build a list of viable, discovered nodes

```
private List<Vector3> AStarSearch(Vector3 startingPosition, List<Vector3> nodeList)
{
    Vector3 startNode = startingPosition;
    Vector3 destinationNode = nodeList[UnityEngine.Random.Range(0, nodeList.Count)];

    //Return if no path
    if (nodeList.Count < 2)
    {
        return new List<Vector3>();
    }

    //Dictionary to keep estimated distance from the start node to end node
    //f(n) = g + h (heuristic = actual distance + heuristic distance)
    Dictionary<Vector3, float> estDistance = new Dictionary<Vector3, float>();
    //Dictionary to keep predecessor Nodes, in order for shortest path to be re-traced
    Dictionary<Vector3, Vector3> predecessorNodes = new Dictionary<Vector3, Vector3>();
    //List to keep every Node visited so algorithm doesn't keep repeating forever
    HashSet<Vector3> closedSet = new HashSet<Vector3>();
    //Order the nodes in the openSet by the lowest value
    List<Vector3> openSet = new List<Vector3>();

    //add start node to openSet for consideration
    openSet.Add(startNode);

    //Initialise scores for all nodes to show that the score has not yet been calculated
    foreach (var node in nodeList)
    {
        estDistance[node] = float.MaxValue;
    }

    //startNode heursitic set to (f = g + h)
    estDistance[startNode] = Heuristic(startNode, destinationNode);

    //While openSet isn't empty (still need to consider nodes)
    while (openSet.Count > 0)
    {
        //Sort open set to get node with lowest estimated total cost (f)
        openSet.Sort((a, b) => estDistance[a].CompareTo(estDistance[b]));
        Vector3 currentNode = openSet[0];
        openSet.RemoveAt(0);

        //End node/destination reached so retrace path and output shortest route
        if (currentNode == destinationNode)
        {
            //contruct path from the destination node to the start not in LIFO order
            return RecreatePath(predecessorNodes, startingPosition, currentNode);
        }
        //Node is added to closedSet
        closedSet.Add(currentNode);

        //Create a list to store neighbour nodes of the current node
        List<Vector3> neighbors = new List<Vector3>();

        //Assigning neighbouring nodes to the list if it has line of sight on the current node (it's not blocked)
        foreach (var node in nodeList)
        {
            if (node != currentNode && HasLineOfSight(currentNode, node))
                neighbors.Add(node);
        }

        //Iterate through neighbouring nodes
        foreach (var neighborNode in neighbors)
        {
            //If neighbour is not in the closed set
            if (!closedSet.Contains(neighborNode))
            {
                //Calculate the estimated cost for a neighbour node
                float costToNeighbor = Vector3.Distance(currentNode, neighborNode) + Heuristic(neighborNode, destinationNode);

                //If neighbouring node's new distance from start node is shorter from current node, update distances
                if (!openSet.Contains(neighborNode) || costToNeighbor < estDistance[neighborNode])
                {
                    //Update estDistance with new estimated distance
                    estDistance[neighborNode] = costToNeighbor;

                    //Record current path of previous nodes
                    predecessorNodes[neighborNode] = currentNode;

                    //Add neighbour node to open set for consideration
                    openSet.Add(neighborNode);
                }
            }


        }
    }

    //No path found so return an empty list
    return new List<Vector3>();
}
```

Figure 37: Implementation of A algorithm*

```
//Reconstruct the A* Search path of nodes in the correct order (LIFO)
private List<Vector3> RecreatePath(Dictionary<Vector3, Vector3> predecessorNodes, Vector3 startNode, Vector3 endNode)
{
    //List to store the path of nodes
    List<Vector3> nodePath = new List<Vector3>();
    //Get the end Node and add it it to the list
    nodePath.Add(endNode);
    //While the end node does not equal the start node, there are still more nodes to add so the end of the list has not been reached
    while (endNode != startNode)
    {
        //retrieve predecessor node
        endNode = predecessorNodes[endNode];
        //Add to the front of the list/path
        nodePath.Insert(0, endNode);
    }
    return nodePath;
}
```

Figure 38: Supporting algorithm to recreate the path in the correct order after A algorithm*

3.4 Testing

After completing the implementation, I conducted testing of the two models. This included collecting data on the following fields for each AI: round winner, round time, number of state transitions, health remaining, hits landed, damage output, Tempest attack usage, Storm attack usage. Peak CPU usage for each round was also collected, filtering usage by scripts, where the average peak total time for the UpdateState() method within the combat class was taken. This was chosen as the interest was in the differences between the single and split model combat classes due to the aims of the project. The total time in this case shows the time the method took, including sub functions. I unfortunately was unable to verify the single AI model's CPU usage during the single versus split tests, as I could not observe any calls on the main thread despite searching for script. The testing process was structured into two distinct stages to ensure a fair and meaningful comparison between the models. The first test involved the single AI model fighting against an identical instance of itself. This baseline test helped confirm that the AI's logic and combat abilities were functioning correctly in a controlled, symmetrical scenario, and provided a useful benchmark for interpreting future results. The second test involved the single AI model facing the split/distributed AI model. This was designed to observe how distributing behaviour across two AI agents (Tempest and Storm behaviours separated) would affect overall strategy, coordination, and individual performance compared to the fully capable single model agent. I decide against conducting tests which isolated each half of the split model to identify their individual strengths and weaknesses against the single AI, as I was more intrigued to explore how the single AI model functioned against the split model as a whole in these scenarios. Within the results table, Sentinel_1 is the single model (with the full set/both behaviour sets), and Sentinel_2S and Sentinel_2T refer to the Storm (Behaviour Set 1) and Tempest (Behaviour Set 2) variants of the split/distributed behaviour model.

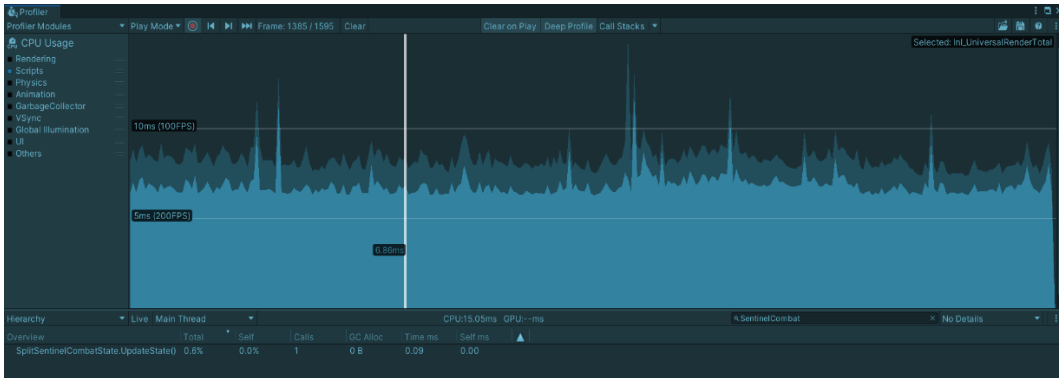

Figure 39: Unity's profiler window filtered to show CPU usage by scripts only

3.4.1 Test 1 Results

Round	Agent	Hits Overall	Tempest Hits	Tempest Uses	Storm Hits	Storm Uses	Teleport Dodges	Damage Output	State Transitions	Transitions to Combat	Transitions to Search	Survival time (seconds)	Outcome
1	Sentinel_1A	16	8	11	8	8	3	1000	11	5	4	61.68	Tie
1	Sentinel_1B	16	8	11	8	8	3	1000	9	4	3	61.69	Tie
2	Sentinel_1A	17	7	11	10	10	3	1025	7	3	2	55.33	Tie
2	Sentinel_1B	17	7	11	10	10	3	1025	7	3	2	55.33	Tie
3	Sentinel_1A	16	7	17	9	9	4	975	13	6	5	74.39	Loss
3	Sentinel_1B	17	7	14	10	10	4	1025	15	7	6	Alive	Win
4	Sentinel_1A	17	6	9	11	11	3	1000	7	4	3	Alive	Win
4	Sentinel_1B	11	2	3	9	9	3	600	6	3	2	44.04	Loss
5	Sentinel_1A	16	10	16	6	6	2	1050	11	5	4	75.49	Tie
5	Sentinel_1B	16	10	15	6	6	2	1050	11	5	4	75.49	Tie

Figure 40: Test 1 Combat Results Table

Round	Peak CPU Usage Total Time (ms)
1	0.42
2	0.91
3	0.94
4	0.17
5	0.39

Average call: 0.57 (rounded)

Figure 41: Test 1 CPU Usage Results Table

3.4.2 Test 2 Results

Round	Agent	Hits Overall	Tempest Hits	Tempest Uses	Storm Hits	Storm Uses	Teleport Dodges	Damage Output	State Transitions	Transitions to Combat	Transitions to Search	Survival time (seconds)	Outcome
1	Sentinel_1	6	4	8	2	2	2	400	8	4	3	27.86	Loss
1	Sentinel_2S	10	N/A	N/A	10	11	0	500	5	3	2	Alive	Win
1	Sentinel_2T	8	8	12	N/A	N/A	2	600	8	4	4	Alive	Win
2	Sentinel_1	13	3	5	10	10	3	725	10	5	4	45.58	Loss
2	Sentinel_2S	18	N/A	N/A	18	18	4	900	9	5	4	Alive	Win
2	Sentinel_2T	2	2	3	N/A	N/A	0	150	1	1	0	Alive	Win
3	Sentinel_1	10	10	13	0	0	2	750	8	4	3	40.43	Loss
3	Sentinel_2S	5	N/A	N/A	5	5	0	250	3	2	1	Alive	Win
3	Sentinel_2T	10	10	13	N/A	N/A	3	750	6	3	2	Alive	Win
4	Sentinel_1	6	6	9	0	0	2	450	8	4	3	29.38	Loss
4	Sentinel_2S	11	N/A	N/A	11	12	0	550	9	5	4	Alive	Win
4	Sentinel_2T	6	6	10	N/A	N/A	1	450	6	3	3	Alive	Win
5	Sentinel_1	6	3	7	3	3	0	375	12	6	5	28.79	Loss
5	Sentinel_2S	8	N/A	N/A	8	8	1	400	7	4	3	Alive	Win
5	Sentinel_2T	8	8	9	N/A	N/A	0	600	6	3	3	Alive	Win

Figure 42: Test 2 Combat Results Table

Round	Single Peak CPU Usage Total Time (ms)	Split Model Peak CPU Usage Total Time (ms)
1	Unavailable	0.18
2	Unavailable	0.26
3	Unavailable	0.17
4	Unavailable	0.10
5	Unavailable	0.34

Average call: 0.21 ms

Figure 43: Test 2 CPU Usage Results Table

4 Evaluation

This section discusses and evaluates the results of testing in the Technical Work section and reflects on the project's objectives and main aim.

4.1 Results Summary

In test 1 the results show a relatively even split between wins, losses, and ties. This suggests a balance in the AI's combat logic when facing an identical opponent. The fact that there are ties (indicating both died) highlights scenarios where neither agent could gain an advantage through their actions. Across the five rounds, there isn't a huge difference in metrics such as hits overall, damage output, and state transitions, indicating a consistent level of engagement and decision-making from the singular AI model across each scenario. While I didn't directly measure the utility scores, the varied values tempest hits and storm hits suggest the utility functions are influencing the AI's choice of attack with the differences in these values across rounds indicating that the AI is not simply executing a fixed, static sequence of actions but adapting based on the game state/environment. The survival times are relatively consistent too, averaging around 60-75 seconds per round, further reinforcing the idea of balanced engagements between the two identical AIs. Lastly, the average peak CPU usage for the UpdateState() method in the combat class (0.57 ms) acts as a benchmark for the computational cost of the singular AI's decision-making process. Overall, the singular AI model appeared to function as anticipated, demonstrating consistent and balanced performance against itself with the variance in attack usage hinting at the utility theory influencing its actions.

Test 2 represents the core investigation in this project, designed to evaluate the effectiveness of the distributed, modular behaviour AI model. The split/distributed model (Sentinel_2S and Sentinel_2T) clearly dominates the singular model (Sentinel_1), with all rounds resulting in victory for Sentinel_2S and Sentinel_2T. This strongly suggests that this distributed, modular approach to behaviour in these test scenarios is more effective at defeating the singular AI model. Sentinel_2S (Storm) displayed consistent Storm hits however Sentinel_2T presented a higher number of Tempest uses compared to Tempest hits. This was likely because of the utility score having a high value when it hits maximum attack range, not considering that it may not be effective outside the maximum range and wasting charges attacking outside this and making it less combat effective once it got to its target. To improve the Tempest behaviour set, a solution would be to piecewise linear curve which is a custom-built curve where I could simply set the usefulness to zero or a small value after the maximum range. The overall combined damage output of Sentinel_2S and Sentinel_2T surpasses the overall damage output of Sentinel_1. This indicates that the division of labour and potentially focused attack strategies of the split, modular model leads to greater overall damage and in turn combat effectiveness. Lastly, the significantly lower average peak CPU usage for the UpdateState() method in the split model (0.21 ms) compared to the singular model (0.57 ms) is notable. This strongly suggests that distributing the AI's behaviours leads to a more computationally efficient system, at least regarding the core combat decision-making loop.

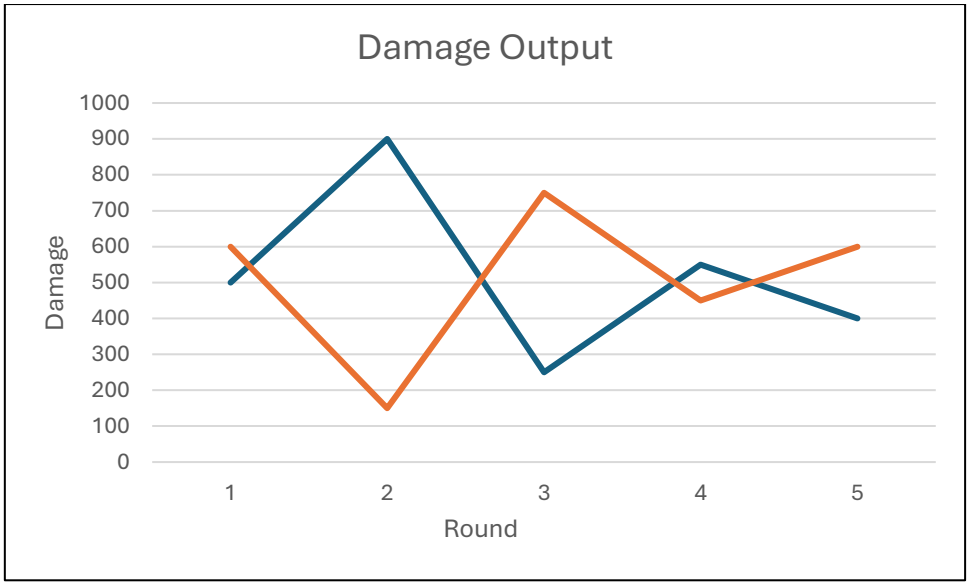

Figure 44: Storm (blue) & Tempest (orange) damage over rounds

Overall, the distributed model appears to be more combat effective than the singular AI, with the higher win rate in Test 2 strongly supporting this conclusion. The success of the split model highlights the potential benefits of modularity regarding combat effectiveness.

4.2 Objective 1 Evaluation

Objective 1 was to assess AI adaptability between AI models. To a certain extent, this objective was met. By observing the attack usage and the frequency of state transitions, I gained insight into how each AI model reacted to the dynamic combat scenarios. The variance in these metrics across different rounds in both Test 1 and Test 2 suggests adaptability, seemingly driven by the utility functions in the case of the singular AI and the inherent reactive nature of the individual, modular behaviours in the distributed AI model.

However, my assessment of adaptability was primarily based on quantitative data that I collected. While the metrics provide some indication of adaptability, a more qualitative approach of combat encounters would have provided a better understanding of the nature of adaptability in this project's context. While the adaptability of the singular AI can be attributed to the utility functions, the adaptability of the distributed AI is more emergent from its static conditions therefore the applicability of this specific distributed, modular system to other real-time games with other requirements might be limited without further development. One potential way in which this could be solved would be with some form of dynamic role assignment.

4.3 Objective 2 Evaluation

Objective 2 was to evaluate AI combat effectiveness between AI models. I believe this objective was mostly well met. The data collected on each combat scenario outcome directly enabled for quantitative comparison of combat effectiveness of both two models. The clear dominance of the split/distributed model in Test 2, evidenced by the higher win rate and often greater damage output, strongly indicates its superiority. In addition, the analysis of individual agent (the variants) contributions within the split model through the attack type metrics provided insight into which aspects of the model (e.g., the consistent damage from the Storm behaviour) contributed to the model's success.

While I identified the more effective model based on wins and performance factors, a greater focus and resulting analysis on why certain behaviours were more effective could have been more comprehensive and beneficial for this project's aim. For example, analysing and reviewing the specific sequences of attacks observed successful outcomes could have provided more granular insight and could enable me to better apply these findings in future work on this project.

4.4 Objective 3 Evaluation

Objective 3 was to explore how splitting behaviours between agents affects synergies between behaviours. My testing primarily focused on metrics for combat outcomes rather than observation of tactical coordination or emergent behaviour. On one occasion I observed the Storm variant taking a route around one of the pillars since the Tempest variant was engaging the single Sentinel Agent in lane, effectively sandwiching the single model and attacking from two different sides. This appeared to demonstrate clear emergent synergy in the combat behaviour. Other times the Split Tempest variant would be late to the engagement and so would start firing from afar, acting as backup for the Split Storm variant. The fact that the combined power of the Tempest and Storm agents consistently outperformed the singular AI suggests some synergy arising from their separate roles.

4.5 Objective 4 Evaluation

Objective 4 was to assess technical performance of the single AI agent model compared to the distributed model. Despite being unable to verify the single AI model's CPU usage during the single versus split/distributed tests, this objective was clearly met. The comparison of average peak CPU usage for the UpdateState() method within both Combat state types measured the computational cost associated with the core decision-making process of each model. The observation that the distributed model exhibits lower CPU usage, directly addressed the objective of comparing the computational cost of the two models in their core function.

5 Conclusion

5.1 Overview of Successes & Limitations

The aim of this project was to analyse and evaluate the impact of modular, distributed AI behaviour in real-time games through developing an AI system and comparing a singular AI opponent with a split/distributed behaviour model, resulting in being able to conclude which is the more effective model. Overall, I believe my project has largely met this aim. I successfully developed both a singular AI model and a split/distributed model, enabling a direct comparison of their performance. The two-stage testing process provided valuable data on their combat effectiveness and technical performance, with the analysis of this data enabling me to draw conclusions in the potential benefits of a modular, distributed approach to AI design.

However, while the project successfully identified the more effective model, a deeper exploration of the nuanced impact of modularity and distributed behaviour could yield more benefits to apply to future work and to reinforce conclusions made in this project. As discussed in the objective evaluations the assessment of adaptability was mainly quantitative, lacking detailed qualitative analysis. In addition, while instances of emergent synergy were observed with the distributed model, a more systematic approach into the coordination and the specific

strengths and weaknesses of these isolated, modular behaviours would provide a better understanding of aspects contributing to the distributed model's success.

5.2 Future Work

The implementation of utility theory still demands more exploration for enhancing the adaptability of both AI models. Exploring expected utility and incorporating a wider range of decision factors could lead to more dynamic decision-making. This could address the observed inefficiency in the Tempest behaviour's attack usage by implementing a piecewise linear utility curve as previously mentioned.

Expanding the complexity and variety of modular behaviours and increasing the number of agents within the distributed system could yield valuable insights into emergent behaviours and tactics. Finally, conducting tests to isolate individual behaviours and qualitatively analyse them in combat scenarios would provide a deeper understanding of the strengths and weaknesses of each AI design, which could be useful to explore to move towards the most adaptable, distributed modular AI system as feasibly possible so that this system can be used in a variety of real-time games.

5.3 Project Reflection & Personal Growth

Over the course of this project, I have developed my understanding and appreciation of different AI techniques used within the games industry, and related topics in the wider AI community such as single versus multi agent systems. Regarding game AI, my research revealed new techniques that I previously had no knowledge of while also developing my understanding of previously known techniques including FSMs, Behaviour Trees, and GOAP. It also became clear to me through reading academic sources, the advantages and disadvantages of these notable industry techniques, as well as how limitations in these can be mitigated or worked around through combining techniques to obtain equivalent or comparable benefits to another implementation.

The implementation of the FSM coupled integration of utility theory to introduce dynamic decision-making, and the considerations involved in designing meaningful and combat-effective AI have provided invaluable experience in developing effective AI. I have enhanced my proficiency within the Unity game engine, also solidifying my technical skills. Overall, this project has equipped me with a more comprehensive understanding of AI principles and practical implementation experience.

While I had great success in development of my skills, I believe I would’ve benefited from devising a more concrete plan to tackle development, such as utilising a development methodology to ensure an efficient and timely project. In addition, I believe development would have been made easier and wouldn’t resulted in a more robust solution if I had used my past placement experience to create and run unit tests.

References

[1] - Colledanchise, M. and Ögren, P. (2018) *'Behavior Trees in Robotics and AI: An Introduction'*. arXiv. 1st edn. Boca Raton, FL: CRC Press.

[2] - de Lima, E. (2020) *Automated Planning*. Available at: https://edirlei.com/aulas/game-ai-2020/GAME_AI_Lecture_04_Automated_Planning_2020.html (Accessed: 19 February 2025).

[3] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, p. 33. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).

[4] - Waltham, M. and Moodley, D. (2016) 'An Analysis of Artificial Intelligence Techniques in Multiplayer Online Battle Arena Game Environments', *SAICSIT '16: Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, South Africa. New York, NY, USA: Association for Computing Machinery (ACM). Available at: https://dl.acm.org/doi/pdf/10.1145/2987491.2987513 (Accessed: 19 February 2025).

[5] - Millington, I. & Funge, J. (2009*) Artificial Intelligence for Games* (2nd ed.). Boca Raton, FL: CRC Press, p. 303. Available at: https://spada.uns.ac.id/pluginfile.php/629724/mod_resource/content/1/gameng_AIFG.pdf (Accessed: 29 April 2025).

[6] - AI and Games (2015) *The Behaviour Tree AI of Halo 2 | AI and Games #09*. 11 May. Available at: https://www.youtube.com/watch?v=NU717sd8oUc&list=PLokhY9fbx05ddpisDpZQ9QfQQsumxn46c&index=3 (Accessed: 19 February 2025).

[7] - Namiki, K., Mori, T., Miyake, Y., Sakata, S. and Martins, G. (2021) Advanced Real-Time Hierarchical Task Network: Long-Term Behavior in Real-Time Games, *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 17(1), pp. 208-212. Available at: 10.1609/aiide.v17i1.18910 (Accessed: 20 February 2025)

[8] - Graham, D. (2014) An Introduction to Utility Theory, in Rabin, S. (ed.) *Game AI Pro: Collected Wisdom of Game AI Professionals.* 1st edn. New York: A K Peters/CRC Press, pp. 113-126. Available at: https://doi.org/10.1201/b16725 (Accessed: 19 February 2025)

[9] - Ushaw, G. (2016) 'Artificial Intelligence 1: Finite State Machines', *Newcastle University: Game Engineering Masters Degree*. Available at: https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/previousinformation/artificialintelligence1finitestatemachines/2016%20Tutorial%208%20-%20Finite%20State%20Machines.pdf (Accessed: 22 April 2025).

[10] - Merrill, B**.** (2014) Building Utility Decisions into Your Existing Behavior Tree, in Rabin, S**.** (ed.) *Game AI Pro: Collected Wisdom of Game AI Professionals*. 1st edn. New York: A K Peters/CRC Press, pp. 127-136. Available at: https://doi.org/10.1201/b16725 (Accessed: 19 February 2025)

[11] - Hunicke, R. and Chapman, V. (2004) AI for Dynamic Difficulty Adjustment in Games. *Proceedings of the AAAI Workshop on Challenges in Game AI*, 2. Available at: https://users.cs.northwestern.edu/~hunicke/pubs/Hamlet.pdf (Accessed: 21 February 2025).

[12] - AI and Games (2016) *HTN Planning in Transformers: Fall of Cybertron | AI and Games.* 10 June. Available at: https://www.youtube.com/watch?v=kXm467TFTcY (Accessed: 21 February 2025).

[13] - Thompson, T. (2024) *Building the AI of F.E.A.R. with Goal-Oriented Action Planning*. Available at: https://www.aiandgames.com/p/building-the-ai-of-fear-with-goal (Accessed: 25 February 2025).

[14] - Orkin, J. (2006) *Three States and a Plan: The A.I. of F.E.A.R.*. Available at: https://www.gamedevs.org/uploads/three-states-plan-ai-of-fear.pdf (Accessed: 25 February 2025)

[15] - Wikipedia (2023) *Fluent (artificial intelligence)*. Available at: https://en.wikipedia.org/wiki/Fluent_%28artificial_intelligence%29 (Accessed: 25 February 2025).

[16] - AI and Games (2018) *The AI of Doom | AI and Games*. 5 August. Available athttps://www.youtube.com/watch?app=desktop&v=RcOdtwioEfI&t=7s (Accessed: 27 February 2025).

[17] - AI and Games (2019) *The AI of Half-Life: Finite State Machines | AI 101*. 29 May. Available at: https://www.youtube.com/watch?v=JyF0oyarz4U (Accessed: 27 February 2025)

[18] - Lovino, M., Förster, J., Falco, P., Chung, J., Siegwart, R., and Smith, C. (2024) *Comparison between Behavior Trees and Finite State Machines*, arXiv preprint arXiv: 2405.16137v1. Available at: https://arxiv.org/html/2405.16137v1 (Accessed: 27 February 2025).

[19] - Aversa, D. (2022) *'Utility-based AI for Games'*, Davide Aversa, 25 November. Available at: https://www.davideaversa.it/blog/utility-based-ai/ (Accessed: 29 April 2025).

[20] - Furnish (2009) Shiver me bones. An Awesome Horror FPS along with Intense Gameplay which will blow your mind. Available at: https://www.gamespot.com/fear/user-reviews/2200-445092/ (Accessed: 23 February 2023).

[21] - Sturtevant, N. (2015) Monte Carlo Tree Search and Related Algorithms for Games, in Rabin, S. (ed.) *Game AI Pro 2: Collected Wisdom of Game AI Professionals*. 1st edn. New York: A K Peters/CRC Press, p. 280. Available at: https://doi.org/10.1201/b18373 (Accessed: 28 April 2025)

[22] - De Ridder, A. (2024) *Multi-Agent Systems vs Single-Agent Systems*. Available at: https://smythos.com/ai-agents/multi-agent-systems/multi-agent-systems-vs-single-agent-systems/ (Accessed: 28 April 2025).

[23] - Ocio Barriales, S. and Brugos, J.A. (2009) *Multi-agent Systems and Sandbox Games*. Available at: https://www.researchgate.net/publication/228740692_Multi-agent_Systems_and_Sandbox_Games (Accessed: 26 April 2025).

[24] - Barambones, J., Cano-Benito, J., Sanchez-Rivero, I., Imbert, R. and Richoux, F. (2022) 'Multi-Agent Systems on Virtual Games: A Systematic Mapping Study', *IEEE Transactions on Games*, PP, pp. 1–15. doi:10.1109/TG.2022.3214154.

[25] - Dragert, C., Dill, K. (2017) Modular AI, in Rabin, S. (ed.) Game AI Pro 3: Collected Wisdom of Game AI Professionals. 1st edn. New York: A K Peters/CRC Press, pp. 87 - 88. Available at: https://doi.org/10.4324/9781315151700 (Accessed: 27 April 2025).

[26] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, pp. 34 - 35. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).

[27] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, pp. 49 - 52. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).

[28] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, pp. 57 - 58. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).

[29] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, pp. 71 - 73. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).

[30] - Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. 1st edn. Cham: Springer, p. 77. Available at: https://link.springer.com/book/10.1007/978-3-319-63519-4 (Accessed: 27 April 2025).