University of Portsmouth

School of Film, Media, and Creative Technologies

Abstract

This project investigates the development of a modular programming model for Boss Enemy AI, an area that is currently under researched. The project delves into established techniques for game AI, such as behaviour trees and utility theory. The programming model utilizes a behaviour tree for structuring actions, alongside a utility-based decision system to create advanced and varied behaviour. An agile methodology was employed throughout the development process as to allow for iterative decisions and refinements during development. The project critically reflects on the advantages and challenges experienced while using behaviour trees and utility systems for Boss Enemy AI. Concluding by evaluating the quality of the model as an adaptable and scalable approach for the future design of Boss Enemy AI.

Tables of Contents

Chapter 1: Literature Review

1.1 Boss Fight Introduction

Boss fights in games can be described as memorable and climactic (Siu et al., 2016), usually existing as harder, unique encounters that differ from what a player usually experiences during the rest of the game (Wood & Summerville, 2019). (Agriogianis, 2018) defines a boss to be a unique character or creature that is more complex and challenging than a regular enemy, proceeding to mention that they typically occur in marked encounters at specific milestones in a game. Creating an interesting method to mark story beats or open up new aspects of the game to players, an idea mirrored by (Wood & Summerville, 2019), who claim that bosses often pose as a roadblock to players, testing their ability before they are given the opportunity to progress further through the game.

1.2 The Appeal of Boss Fights

A major aspect of boss fights that appeals to players, is feeling and emotion you receive upon defeating one, the overwhelmingness and excitement when you find a powerful boss, or the feeling of achievement when you figure out how to beat them. (Agriogianis, 2018) claims that these are some of the main components that makes a good boss fight. (Siu et al., 2016) also makes the point that the sense of accomplishment granted to the player upon defeating a challenging boss is a valuable component in creating a good boss fight, implying that the level of difficulty a boss is on, can impact the sense of achievement a player is granted upon completion. (Vorderer, 2003) states that “Video game players expect a given game situation to be more enjoyable if they are confronted with a competitive element than if such an element is missing in the situation.” Suggesting that players themselves might expect harder experiences to be the more rewarding ones, despite this idea, the theory only applies to players of a certain competitive nature, some may only find achievement in player vs player games, and some might be further inclined to a more relaxing experience instead, finding an overly competitive scenario to be too overwhelming.

(Vorderer, 2003) Continues to discuss “social competition” in games, the idea that the perceived relationship between a player and an enemy, be that enemy a normal non-player character(NPC), player, or a boss fight, can lead to different emotional reactions depending on the perceived relationship between the two parties. Swinging the artificial relationship so that the player begins in a comparatively weaker position, could increase the sense of accomplishment gained from beating the boss; the harder the boss, the better you feel after beating it. (Vorderer, 2003) also discusses the idea that some players might experience a differing level of enjoyment from a highly competitive or overly challenging experience, depending on the individual’s preferences towards these scenarios. They also suggest that one’s desire to maintain self-esteem, alongside their desire to seek out harder challenges, can affect how well they respond to an overly competitive situation. A challenge raised by (Agriogianis, 2018) is the difficulty in finding the right balance between making a boss too challenging or too easy, whilst still positioning it to be a good skill or progress check for a game.

A topic raised by (Agriogianis, 2018) is the importance of the presentation of a boss. Even before the encounter itself, there are other things you can do to make the boss more interesting. This could be as simple as mentioning the boss before the encounter, building the boss up in the mind of the player or making them think the boss might offer a more significant experience. There is an implied correlation that the more a boss is mentioned prior to the fight, the more a player will build the boss up in their mind. This idea is also discussed by (Wood & Summerville, 2019), who instead suggest that while the presentation is important, the mechanics, difficulty, and fairness are much more important factors. Should a boss be irrelevant to the story of the game, typically will mean less to the average player than a boss that is highly active within the plot. Obviously, there will be exceptions to this rule; in some cases, word of mouth/interest in a certain boss can be raised if the boss is either interesting or difficult enough.

The topic of freedom in a boss fight is a subject discussed by (Vorderer, 2003), who suggests that a game scenario is often more enjoyable if the player is given a sense of freedom stating that “Video game players expect a given game situation to be more enjoyable if they are offered different possibilities to act than if they are offered only a few such possibilities”. They claim that a challenge is more desirable, the more possibilities there are to complete it, suggesting that players don’t want to be restricted in what they can do to solve a certain problem; if a scenario is overly restrictive to the player, it could be considered boring or lacking replay-ability. (Agriogianis, 2018) mentions that one of their personal favourite aspects of a boss fight is discovering a way to complete it; by restricting a problem to one solution, you remove a lot of the ability to discover how to win, as well as removing a larger reason to replay.

1.3 Game AI

Game AI is described by (Steve Rabin et al., 2017) as lacking any real deep intelligence, instead creating the illusion of it, where there are three things working together in the players mind in order to unknowingly achieve this goal. They go on to state that these things are: one, the fact that players want to believe that agents in their games have a real-world intelligence, two, that humans have an innate desire to anthropomorphise non-human entities, and finally, that if the player has expectation going in, they can more easily become reality in the mind of a player. (Bourg & Seemann, 2004) suggest one possible definition of AI be “The ability of a computer or other machine to perform those activities that are normally thought to require intelligence” before raising the question “what is intelligence?”. They conclude that game AI has a broad and flexible definition, further suggesting that game AI would be considered weak AI anyway when compared to more powerful artificial intelligence, while stating that it involves a broader range of purposes and technologies to give machines specialised intelligent qualities. AI within games has been a staple since the beginning of the industry, (DaGraca, 2017) brings up an early use of game AI being computer chess programmed to play against humans. They mention how single player games with AI enemies started to appear as early as the 1970s, many cases taking the form of arcade machines. (DaGraca, 2017) continues to discuss how having enemies with randomised elements as opposed to strictly defined patterns made the games more enjoyable to many players, as it allowed for a unique experience when you would replay the game. (Bourg & Seemann, 2004) raise the question of how smart nonplayer characters should be? They go on to suggest that within games we do not need typically need to try and replicate human level intelligence in AI, continuing to pose the idea that having a variety of levels of intelligence could create a more unique and richer experience.

1.4 Different Types of AI

There are numerous different techniques and styles of game AI that can be used to create remarkably similar experiences, often indistinguishable from each other by the average player, but while holding up matching facades, there can be a lot of variety within the backend of the system. However (Bourg & Seemann, 2004) go into detail, discussing the difference between “Deterministic” and “Nondeterministic” AI. They explain that deterministic behaviour is set and predictable, suggesting that there is a lack of uncertainty in how the AI will react to a scenario. Contrarily, nondeterministic AI is the opposite, being unpredictable depending on the methods used to create it. This branch of AI is a lot more complex, sometimes allowing the AI to learn and adapt through the use of neural networks, Bayesian techniques, or genetic algorithms. (Bourg & Seemann, 2004), suggests that game developers have often been wary of using nondeterministic AI as it can provide a lot of difficulties, in testing and debugging, as well as taking away a degree of control.

(Steve Rabin et al., 2017) discuss the use of deterministic modular AI in games. They emphasize the importance of recognising repetition in code to maximise efficiency and reduce what could be a long and convoluted code into effective object-oriented code that allows for the reuse of a pattern. This technique has many advantages including decreasing executables size, decreasing the areas in which a bug can be introduced, increasing the number of ways in which the code can be tested and finally reducing time taken to implement it. Working this way allows for inheritance and polymorphism, meaning that a single parent class can encapsulate a wide variety of tasks simultaneously despite each requiring different content. (Steve Rabin et al., 2017) suggest that modular AI is fundamentally about the transition between these states, “it is about enabling you to rapidly specify decision making logic by plugging together modular components that represent human level concepts”. They explain that there should be a collection of modular components that are only implemented once but can be used repeatedly. It is this concept that encapsulates the way we program deterministic AI in games.

1.5 Finite State Machines

A finite state machine is one type of structure used to create deterministic game AI, involving several different states the AI can possess, controlled by a central class, the state machine. (Steve Rabin et al., 2017) claim that state machines are great for composing behaviours, as each state can be parameterised and reused for multiple different characters, e.g. an attack class that can hold different animations for different characters, emphasising the extreme modularity and flexibility granted when using a state machine. They also accentuate how simple the system can be to implement and maintain, an aspect that’s importance cannot be understated. Due to this simplicity (Steve Rabin et al., 2017) do however mention that a finite state machine might remain best suited to games that have smaller AI needs. These states will typically govern over a ‘mindset’ or an action for the AI, (DaGraca, 2017), in the context of a common first person shooter(FPS) game, gives the examples of “Walk slowly to find cover, wait for the player, and shoot him”, “Run for cover and then fire from that position”, “Defend while running to a cover position”, or “Fire against the player, running towards him, and keep firing”. They later add the idea of having states more mindset focused such as “PASSIVE”, “DEFENSIVE”, or “AGGRESSIVE”. They go on to discuss how depending on the type of game you are making; the same states can be reused in order to shape the game into a different genre. An interesting point made by (DaGraca, 2017) is that the personality of the character you are trying to create needs to be shown through the AI. They give the example that (in the previous context), a robot won’t be afraid to keep shooting the player even if the odds are against them, however if the character is meant to be an inexperienced soldier, they might react differently. State machines excel for AI characters that change actions less regularly, typically remaining in a state for longer than a single action, waiting for a specific situation to change what it is doing.

1.6 Behaviour Trees

Another kind of structure used when making game AI is the behaviour tree structure; described by (Steve Rabin et al., 2017), behaviour trees are an architecture for controlling NPCs based on a “hierarchical graph of tasks”, where a task is either atomic (a task or action the NPC can complete) or composite (a behaviour linking to other nodes in the graph). According to (Steve Rabin et al., 2017), a behaviour tree can provide a “cleaner decomposition of behaviour” than alternative systems such as finite state machines. They discuss how behaviour trees can be very modular and customisable due to the way they are made up of lots of smaller components that can be individually overridden and parameterised. (Steve Rabin et al., 2017) emphasises the importance of finding the right balance between how complex nodes should be, suggesting that by making the tree too “tightly coupled” it can lack customisability. On the other hand, if the nodes become too large, it will lack variability and modularity.

(Steve Rabin et al., 2017) does raise a few potential issues one might face using behaviour trees including overcomplicating or overusing certain systems, concluding that it is important to heavily plan out the system before you begin developing it, to prevent the first issue raised; it is best to figure out what the system will need to do so it can be made as efficient as possible. They emphasise the importance of refining abstractions to a point where the functionality of the system is extremely simple. (Steve Rabin et al., 2017) points out that it is important to look carefully for repeating work as to make it as modular and hierarchical as possible, trying to keep code constricted to classes as small as a page in size, sometimes less, then finishing by claiming that behaviour trees can be surprisingly complicated, despite often being very regular in structure.

1.7 Behaviour Decision System

There are other ways of doing this instead of a behaviour tree or a finite state machine. (Steve Rabin et al., 2017) discuss the use of a “Behaviour Decision System” (BDS) used in “Dragon Age: Inquisition (BioWare, 2014), which was used instead of either a behaviour tree or finite state machine. They suggest that the architecture of the decision system is based upon assumptions that; “at any given time, there is a finite set of actions which and AI character can perform.”; “An AI character can only perform one action at a time.”; “Actions have differing utility values; some actions are more useful than others.”; “It is possible to quantify the utility of every action”. It is claimed that if all assumptions are considered, there is naturally an algorithm to decide the AIs best action in any situation; first the AI must identify all actions it can take at any time, then evaluate each action and assign it a score based on its current utility, afterwards, the action with the highest score is the one that can be completed. (Steve Rabin et al., 2017) simplifies the “BDS” framework to be “a framework that allows gameplay designers to impart knowledge to AI characters” specifically “which actions can an AI character perform? Under what circumstances can they perform those actions? How should those actions be prioritized relative to each other? And finally: How can those actions actually be performed?”.

The architecture used data structures they termed “behaviour snippets”, which contained all of the data the AI needed for each action. (Steve Rabin et al., 2017) state that they used a modified behaviour tree called an “evaluation tree” to represent the utility of each action. The tree would be used to pit two actions against each other in order to rank utility. In the context of Dragon Age: Inquisition (BioWare, 2014), this type of modified behaviour tree could also be used to select what target to attack. The evaluation tree is run on the AI characters update, where the score produced, and the target found are stored in a summary table. When everything has finished running, the behaviour snippet with the highest score is executed and the cycle repeats.

(Steve Rabin et al., 2017) emphasises the modularity of the behaviour snippets system. In Dragon Age: Inquisition (BioWare, 2014) the players are able to customize the AI characters that use the BDS framework, suggesting that the framework is very effective at creating AI characters that can be manipulated easily by the player at runtime.

1.8 Utility Theory

Utility theory is a way for an AI to make decisions, driven by mathematical variable and equations determining the value of a certain action. (Lebedeva & Brown, 2020), state that “Utility function maps some parameters of the environment onto a value that describes how desirable the action is”, further suggesting that this value is normally stored as a number between zero and one. In the context of a companion AI, (Lebedeva & Brown, 2020) claim that without the use of utility calculations, the NPCs had no consideration for the players, the enemies or their own states when deciding what task should be completed next. They give the example where, if the AIs actions don’t depend on this data, the companion would ignore the player if they were on low health because it didn’t know it was supposed to help instead. Adding utility to an NPC can add the guise of intelligence fairly easily, additionally being implementable into most kinds of game AI architecture. (Steve Rabin et al., 2017) evidences this by suggesting that a utility system can break decisions down into factors that affect whether or not the decision is the optimal one in any given scenario, applying rationale to what the AI does.

An example provided by (Lebedeva & Brown, 2020) poses a “protect the player” action: in this case, they use the players maximum and current health to determine how desirable it would be to use the action. The equation used made it so the utility value increased as the players current health went down. (Lebedeva & Brown, 2020) continue to provide examples of utility functions, including examples with more complex equations having multiple affecting factors. (Steve Rabin et al., 2017) discusses the value in using utility theory in AI programming, emphasizing that the system enables a large amount of flexibility and configurability, additionally giving more control to designers and easing the load on programmers. (Steve Rabin et al., 2017) suggests the use of a “decision score evaluator” (DSE). This DSE compares the utility scores for each action to find the best option. The utility values are normalised to sit in the range zero to one, they are processed in this form and then normalised again to be their final score. (Steve Rabin et al., 2017) states that a key trait of the design is that any single value can disqualify an action entirely if its value is equal to zero. If a factor for an action is zero the whole action can be disqualified, allowing the system to stop processing it at this point. This means that unnecessary actions don’t need to be fully processed, increasing the efficiency of the AI.

1.9 Existing Framework

There is a lack of research for a boss battle specific game AI framework; however, one example is (Siu et al., 2016) who designed a programming model for boss encounters in 2d action game. The model utilises a finite state machine alongside “component-based systems such as physics and health”; the boss’s behaviour is defined by a set of state machines, each controlling the boss in different ways. The type of bosses described in the framework are “2d action adventure” bosses; (Siu et al., 2016) explain that the model describes a more reactive system, most of the “component-based” decisions made by the AI were made based on collisions with the player or health values. The boss itself deals damage to the player on collision with its body instead of a more advanced combat system. Because of this, the states are movement based over anything else, changing direction or speed. This framework is fairly basic, utilising state machines and collision detection as its main features; it revolves around more passive style bosses that don’t aggressively attack the player, nor do they use complex move sets or combo attacks. (Siu et al., 2016) extensively cover the systems for this specific style of boss fight, but does not suggest applications for varying genres or styles. They discuss extending the model but insist on remaining within the basic set of parameters used initially.

Chapter 2: Methodology:

2.1 How Behaviour Trees Work

(Steve Rabin et al., 2013) defines behaviour trees as describing a “data structure starting from some root node and made up of behaviours, which are individual actions an NPC can perform.” The algorithm gains tree like qualities as each behaviour can have child behaviours. Each behaviour has a condition or set there of that allows it to be run; the tree starts at the root node and works its way down the tree until it hit a behaviour with the correct conditions. If a behaviour’s conditions aren’t met, it moves onto the next until it finds one that does; if no actions can be completed it returns to the root and runs again until one can be. It is stated by (Steve Rabin et al., 2013) that an advantage of behaviour trees is that they are stateless, meaning that behaviours don’t need to be aware of each other: “This alleviates the problem common with FSMs, (finite state machines) where every state must know the transition criteria for every other state. Behaviour trees also have the ability to use different composite nodes, where a parent behaviour could dictate that all of its children get run. One is randomly selected to be run, or one is chosen based on utility. (Steve Rabin et al., 2013) emphasises the importance of not overusing conditional statements as it can slow the system down a lot, especially as the tree gets run every time a new action needs to be taken.

2.2 Drawbacks of Behaviour Trees

(Steve Rabin et al., 2017) discusses the possible drawbacks to using a behaviour tree system. The relevant issues are as listed: the creation of too many organizing classes, and the overuse of a “blackboard” system and routing everything through it. The first pitfall discussed by (Steve Rabin et al., 2017) is the overuse of organizing classes, they initially suggest that it can be a bad idea to create a separate “architectural category” for everything you need the system to do, as it makes it harder to develop the features needed. A system that implements too many different types of behaviour or action becomes very complex when other components get added later. It is suggested by (Steve Rabin et al., 2017) that a system using one organizing class could be the most effective, by addressing every task under one name: task. By doing this they claim that any task you would have needed could be built from an inherited version of task; this in turn also massively simplifies the ability to navigate and manipulate the tree itself.

Another issue raised by (Steve Rabin et al., 2017), is the of overuse of a systems blackboard. They suggest that it could present as an issue if the behaviour tree logic and the blackboard logic are too closely related, or if simple tasks from the tree rely on the blackboard to be actioned. An example shared by the writer claims that one of the systems they were working with raised issues when a change to the Application Programming Interface (API) for one of the systems was made. Due to the way in which both systems were designed to be linked together, a change in one meant a change in the other would be required. The alternative is to have a system where the tree and the blackboard are decoupled, either using a different hierarchical blackboard or none at all. It is suggested that it is harder to build a data driven behaviour tree when using no blackboard, however they state the ease at which the tree could be updated later, due to the change. Despite this, depending on the project in question, this issue might not pose a problem at all.

It is concluded that a detailed plan should be made before development of the system begins to reduce the effect of the aforementioned drawbacks, isolate what is needed from the system and use abstractions to simplify the functionality of the system. By doing this, you also make it easier to locate where the system can be more modular and reusable, an important feature for a programming model intended for use in different circumstances.

2.3 Utility Theory

Utility theory is a very useful tool for decision making in game AI. Both (Lebedeva & Brown, 2020) and (Steve Rabin et al., 2013), discuss advantages and examples for the use utility theory in game AI; (Steve Rabin et al., 2013) claims that restricting decision making to the use of Boolean selection can cause the AI to have very discrete and predictable decision making. Utility theory allows for the use of multiple affecting factors without having to use more complex Boolean equations, this both means that you are less restricted for what can affect decisions, but it can also be less discrete and determinate. (Steve Rabin et al., 2013) suggests that this way of decision making can instead provide the AI with a range of possible options for any given scenario allowing for the possibility of more randomised and variable decisions. If a score is created to decide the utility of an action, this can be manipulated with random values or be changed based on the actions repetition, so that the AI character can have more variety in what they do.

2.4 Other Potential Option: Finite State Machine

(Steve Rabin et al., 2017) provides a design architecture for a finite state machine system beginning by discussing the “StateMachine class”. This class owned by the AI game object, is what manages the behaviours for the AI. It owns a list of every possible state the object can be in, as well as a pointer or reference to the state the AI is currently in. They propose that the state machine is updated periodically (either every frame or a period that works well for performance), where it uses an interface for setting the state. A system is then advised where each state has an entry, exit and update function that can be called on these periodic updates.

This system should be built in a base state class that allows all of the “concrete” states used to inherit and override where needed. For the transitions themselves, (Steve Rabin et al., 2017) recommended the use of a “StateTransition class” owned by the state machine, that holds a Boolean indicating whether or not a transition needs to be made. Furthermore, each transition possible in the game should be its own child of this transition class, i.e. ‘PassiveToAggressive’ or ‘AggressiveToDefensive’. Every update the state machine would check through a list of these transitions to see if any of them return true on the Boolean that defines if a change should be made. This system of transitions allows more control over certain states, as an example given “you might not want to allow an enemy to transition out of the death state”. In this architecture, this can be prevented by not including a transition class beginning with the death state. (Steve Rabin et al., 2017) additionally raise the idea of “Metastates”, claiming that one way to expand the system would be to implement a state with its own state machine inside it, where there are substates that can be changed between within the “major” state. This adds more complexity to the system but creates a lot more flexibility in what you can do. For example: granting the ability to have different types of aggressive states that can be swapped through whilst still remaining under the umbrella state of aggressive.

Unlike behaviour trees that focus on a single or short string of actions, state machines tend to dictate an AI character’s mindset; the AI will just keep the state running until it needs changing instead of running through an array of checks every time to see what it should do. It is however because of this that I think a finite state machine system to be less desirable when creating a framework for boss fights as the boss will not have overarching states but a list of actions to pick from.

2.5 Systems Conclusion

Despite the potential issues that can occur when using behaviour trees, I believe that the benefits of the structure make it more suitable for the development of boss AI than the alternatives. Firstly, the boss will be an exclusively aggressive AI and therefore will not act under different mindsets as is encouraged in the state machine structure. In addition to this I feel as though a boss AI is more suited to using single actions or sequences of actions. As is stated by (Steve Rabin et al., 2013), behaviour trees are often comprised of individual actions, which I feel is more appropriate than a state machines more long form state system. Another reason for the use of a behaviour tree over a state machine in this scenario is the advanced modularity that is offers: a state machine is comprised exclusively of states and metastates, where every different action or sequence need to be unique. In comparison, behaviour trees can have different composite or reusable nodes that can be repeated where relevant., This more modular system allows composite nodes to be reused in multiple scenarios such as creating multiple different action sequences or selections. This also allows for a wider array of different and more advanced actions. I have also concluded that the most appropriate option for deciding what action to use is utility theory through the use of utility score for each action. This is because in creating a utility score for each action, you can use a greater number of factors in the decision process, in addition to granting the ability to use different deciding factors for each individual action.

2.6 Software Development Methodology: Agile

Agile Software Development Methodologies (SDMs) are concluded by (Charles Edeki, 2015) to hold multiple benefits over other methodologies such as waterfall. They posit that “Agile attempts to simplify the software planning and estimation process by decomposing large requirements into small individual tasks” (Livermore, 2007). They additionally suggest that Agile SDMs can be used to produce higher quality software over a shorter time frame, emphasising that these types of methodology can streamline the development process and can remove certain restrictive barriers especially when working alongside clients.

(Charles Edeki, 2015) discusses the way that many Agile SDMs break down the development timeline into “sprints”, they suggest that at the beginning of each sprint the development team should have a meeting to discuss the processes, and features are of highest priority for the following sprint. Whilst this project is a solo one, I would still consider that breaking the process into sprints and reflecting after each one would still be good practice. Both (Charles Edeki, 2015) and (Livermore, 2007), describe Agile methodologies to be a form of iterative development, allowing for changes and updates to the design of the software as development progresses. (Charles Edeki, 2015) additionally outlines the way that an Agile methodology can help to increase the accuracy of expectations for a project as it progresses and allows for potential rescoping of projects midway through development dues to the fast iterative sprint system.

2.7 Timeline for Development

According to (Charles Edeki, 2015), lack of sufficient planning is one of the leading causes of delays within software development. In other SDMs developers are forced to predict the timeline of a project at the start of the project with little room to adapt later. However, by utilizing a form of Agile, there is the capacity to restructure and re-estimate, at a later stage of development.

For my development timeline, I have structured the overall task into sub-tasks that are more comprehensible, as to view the overall development process with more ease. My initial timeline (see below) breaks down all different aspects of the development process into chunks, including the timeline for the production of the report before and after development. As is customary when using an Agile style SDM, the development process has been split into sprints. In this case I have split development into four two-week sprints. Due to the individual nature of the project I felt that reducing the sprint duration to a single week would likely result in multiple tasks running over multiple sprints. This issue is still evident in the timeline, however overlapping tasks will be further decomposed at the time of sprint planning to ensure that they align with the set sprint timeline. Additionally, I have set aside an extra six week post development to allow adequate time to produce the final report. This time I feel is necessary for the completion of the report and therefore will be attempting to strictly hold the to the initial period of time allocated for development to ensure that the report itself is of a higher standard.

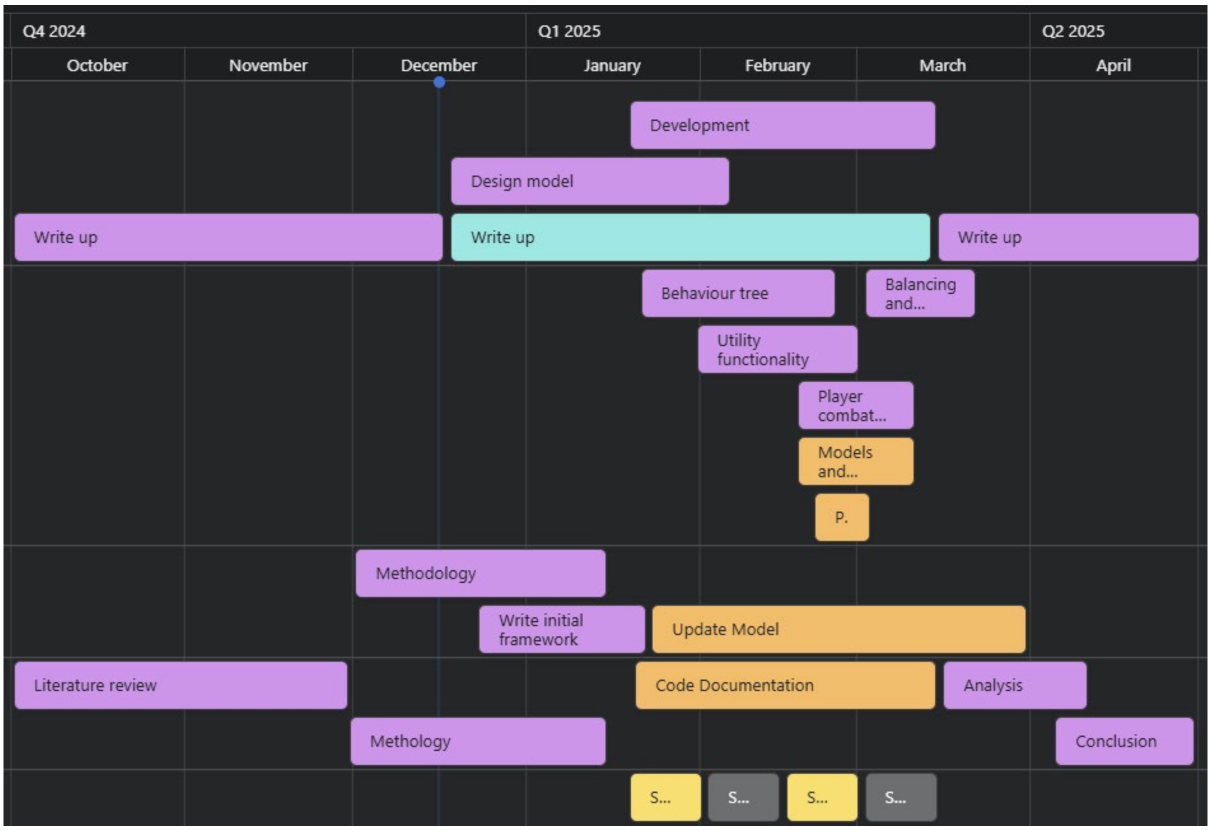


Figure 1: Initial development timeline

Chapter 3: Programming Model

3.1 Behaviour Tree

As discussed previously, the main component of the programming model will be the behaviour tree. For the behaviour tree I plan to mostly keep to convention. As described by (Steve Rabin et al., 2013), a behaviour tree is a data structure beginning from a root node that follows down a tree of behaviours and child nodes. The first node is the basic/atomic node, this node will be parent to all other including composite nodes and individual actions. Below the node class, will be the composite node class that is in turn parent to selector and sequence nodes. The selector node is what will allow the tree to differentiate between different actions and when they should be run. Sequence nodes will be used to create action sequences where some actions might better suit being further decomposed into smaller chunks; this will also allow some code to be reused where relevant, as actions such as move towards player will be applicable for multiple action combinations.

Another component that is commonly used in behaviour trees is the blackboard class that holds useful information for the tree to use. (Steve Rabin et al., 2017) recommends hesitation before overusing the blackboard as it can lead to other unforeseen issues. Due to this recommendation, I plan to only use the blackboard where necessary, specifically for holding information that will be used by the utility system in deciding what action is best.

To reduce the risk of over complicating the system, something (Steve Rabin et al., 2017) suggests can happen if the system is under planned or overly complex, I will be keeping the initial design (below) simple. The design for the tree will only contain a few different actions, designed to demonstrate the different capabilities of the design. The actions will be simple attacks, a move to player then attack and a dash attack, however, I plan to reconsider and potentially add more actions should development run smoother than expected.

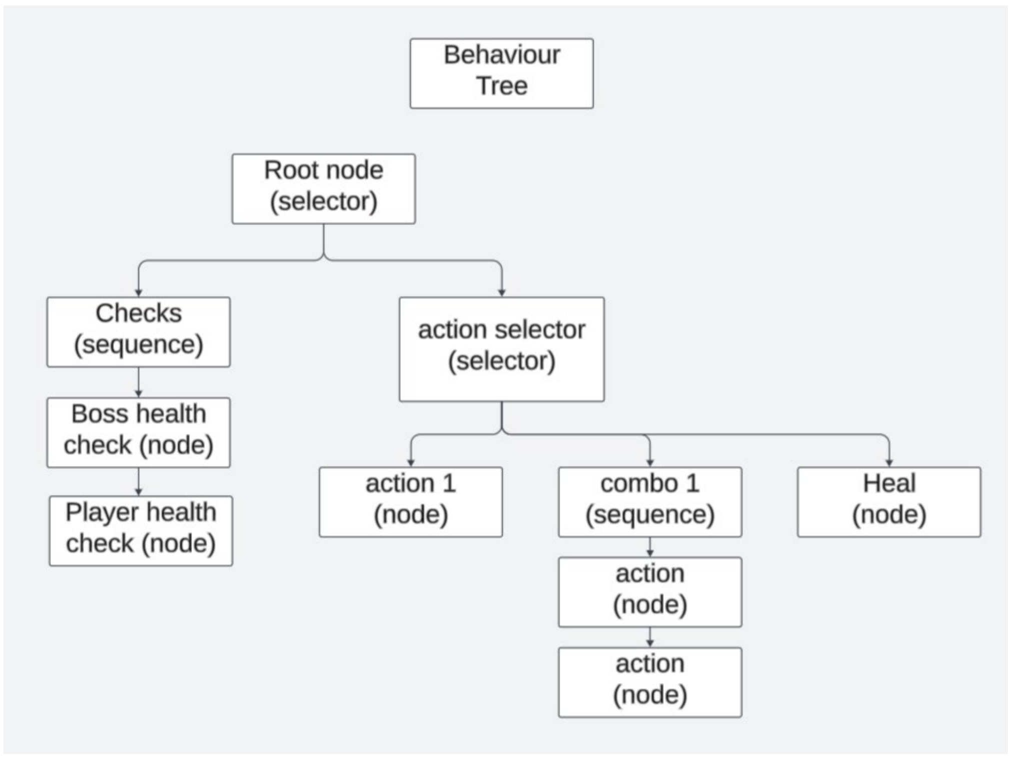


Figure 2: Behaviour Tree design

As is shown in the initial development timeline (figure1), I plan for this task to take a large chunk of the development time, a sizable amount of this time however is likely to be spent on testing and refining the system instead of building it, as well as potentially incorporating additional features into it.

3.2 Utility Decision System

Utility theory is a way of increasing the quality of an AI by making its decision system more advanced. (Lebedeva & Brown, 2020) emphasise how utility theory allows the AI to use an array of world factors to appraise each action before it is selected. This will be done by running a function to give each action or combo a utility score based on a set of ideal values and parameters. Each node class will contain variables that hold their ideal range and player health, which will be set on construction. These values can be used to increase or decrease the challenge, as having more difficult actions occur more when they are most effective will maximise the overall challenge of the fight. Furthermore, when an action is run, it will reduce its score by a set amount, which will then be reduced when other actions are completed. This will therefore reduce the repetition of each action if one is more desirable than the others. On top of a repetition factor, I plan to add a random factor to the decision system to further increase the unpredictability of the boss. Another action I plan to incorporate is the heal action, this will also require the boss's health as a factor in the decision system.

Whilst I believe that programming this system may not take a long time, I believe that it will be more complex to balance than other systems. This is because not all actions will have the same number of factors and some factors will completely halt actions from taking place; this will also become more complex with the addition of further actions, should that occur.

3.3 Player Controls and Experience

Due to a limited development period alongside the focus on the programming of the boss, certain other aspects of player experience, including visuals and context won't be the focus of the artefact. Because of this I plan to use simple transformations to portray the boss and players actions instead of animations, as I believe due to personal skill set this will be simpler to implement while still retaining a good visual experience for the player. Similarly, I do not plan to draw focus to narrative design or context. To add this, would require implementing more game surrounding the boss fight and I do not believe the timeline I have outlined for development will be sufficient to do this.

The player combat/controls will be relatively simple while still providing an adequate set of abilities for the player to interact with the boss. For this I plan to create a reduced set of actions based on other boss focused games which will include basic movement, a dash moves for increased mobility, and an attack for combat with the AI. I believe that this section of development will take a whole sprint to complete, however I have allocated additional time to account for potential issues that occur during production.

3.4 Sprint Outline

Sprint One

The first sprint will be focused solely on the behaviour tree: this will be creating the basic structure and adding in different nodes that will be used. I plan to have in at least the basic and composite nodes, ideally being able to start implementing some of the utility system, or AI actions.

Sprint Two

I plan for this sprint to involve the polishing and completing of the background behaviour tree, as well as the initial utility system. Ideally this will include having a way of calculating utility score, and routing the tree using an action index based on the utility scores.

Sprint Three

The primary focus of the third sprint will be the player experience: provided everything is running in line with the plan, this sprint will involve creating the player combat system as well as making the model and transformations to get the boss into a playable state.

Sprint Four

The fourth and final development sprint will be used to finish up any unfinished programming as well as balancing and small changes to improve player experience. I expect this sprint to potentially include adding in final actions and adjusting transformations to make the system as complete as possible.

Chapter 4: Reflection:

4.1 Agile Methodology and Time/Sprint Management

As I had planned, I stuck to the set development period I set out in the methodology. However, the predictions for each stage of development did not entirely follow suit; despite this it did demonstrate to me the usefulness of breaking development down into separate sprints, and leaving the content of each one open to change based on where the project is at the start of a sprint. Where I spent time at the start of each sprint to analyse where I was at, I was able to reallocate certain tasks more or less time than originally thought without having to change the whole timeline. Specifically, I had set aside a lot of time to make the behaviour tree itself, allocating time in over three of the sprints, even including minor changes and bug fixes, the tree itself took less than two sprints to be completed.

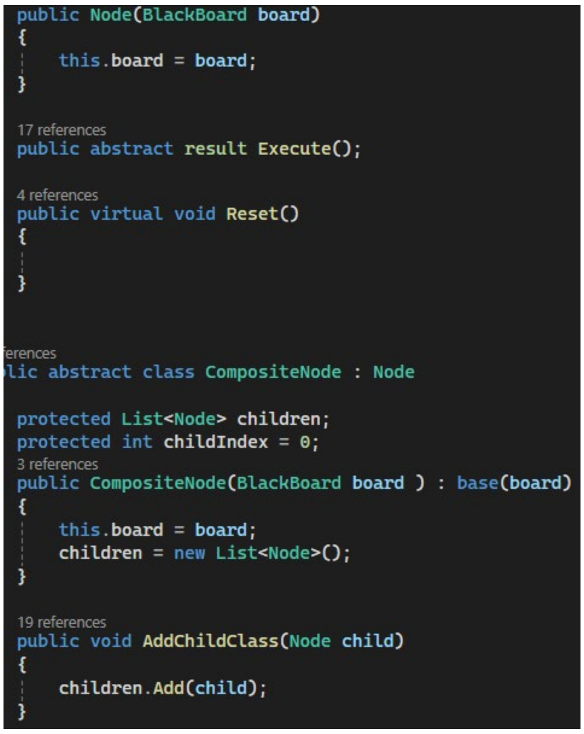
However, the player experience and transformations proved to be a lot more complex and ineffective than I had originally thought. Towards the end of development, I found myself falling behind on the hours I had set aside for this project. This could be in part due to a degree of burnout towards the end of the development period but also due to client deadlines for other modules in progress. This ended up causing a drop in quality to some of the features that were added to the project later into development. Additionally, the original plan for the sprints had to change a lot where certain systems hadn't been implemented or finished in time to move on to something new. The fourth sprint, outlined for bug fixing and balancing to improve player experience was reevaluated and had to be used to finish the implementation of the player combat and the transformations, rather than make minor changes and improvements.

4.2 Behaviour Tree Evaluation

Prior to this project I had not previously used behaviour trees. Due to this, I was hesitant to make predictions for the amount of time taken to implement one, as well as the quality or usefulness of them over alternative AI designs that I have used before. While I ran into a number of issues during this project, one system that seemed to remain relatively sturdy was the behaviour tree itself. By following recommendations from (Steve Rabin et al., 2017), and planning out the system exactly before programming, the process of creating a behaviour tree went smoothly.

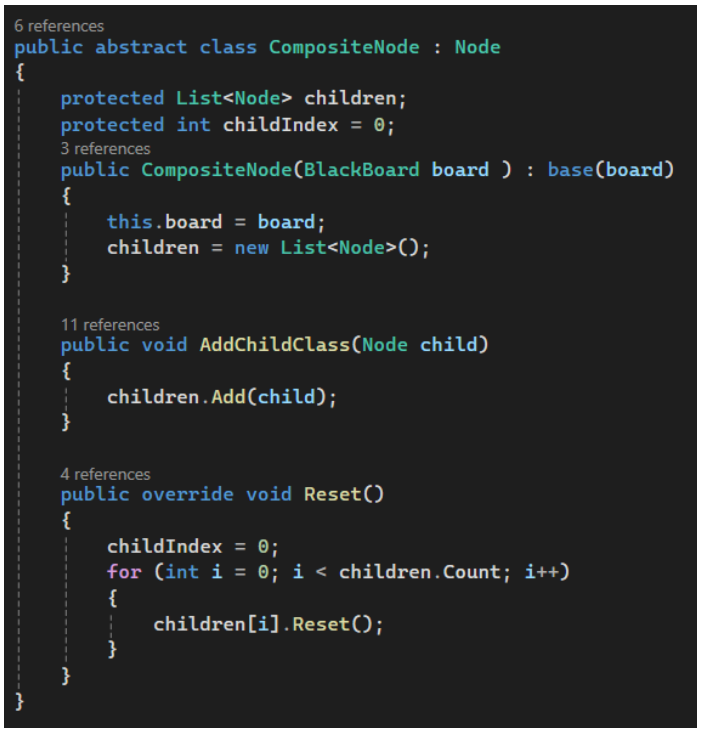
When comparing behaviour trees to other options including finite state machines; an AI design that I have more experience using, I found the basic setup of the behaviour tree to be more confusing. The behaviour tree returns different results based on the outcome of the executed functions, and each composite node outputs different results based on the results of the child nodes under them. At first, I found this concept to be overly complex, and I didn't understand exactly how to make the AI move through the tree how I wanted. However, once I fully understood of the system worked, I realised how useful it was, giving me three different results: referring to a completed action, a running one, and a failed one. This gave me a better idea of how the system was running, in addition to more control for each action.

In a finite state machine, states transition between each other based on checks and parameters in each one, linking to all other possible states, in the behaviour tree I was able to add a single check to see if the action had been chosen before either running it if it was, or failing it if it wasn't, there was no need for actions to be linked to each other beyond being connected as children.



```
public Node(BlackBoard board)
{
    this.board = board;
}

17 references
public abstract result Execute();

4 references
public virtual void Reset()
{

}


erences
lic abstract class CompositeNode : Node

protected List<Node> children;
protected int childIndex = 0;
3 references
public CompositeNode(BlackBoard board ) : base(board)
{
    this.board = board;
    children = new List<Node>();
}

19 references
public void AddChildClass(Node child)
{
    children.Add(child);
}
```

Figure 3: Code snippet: Nodes

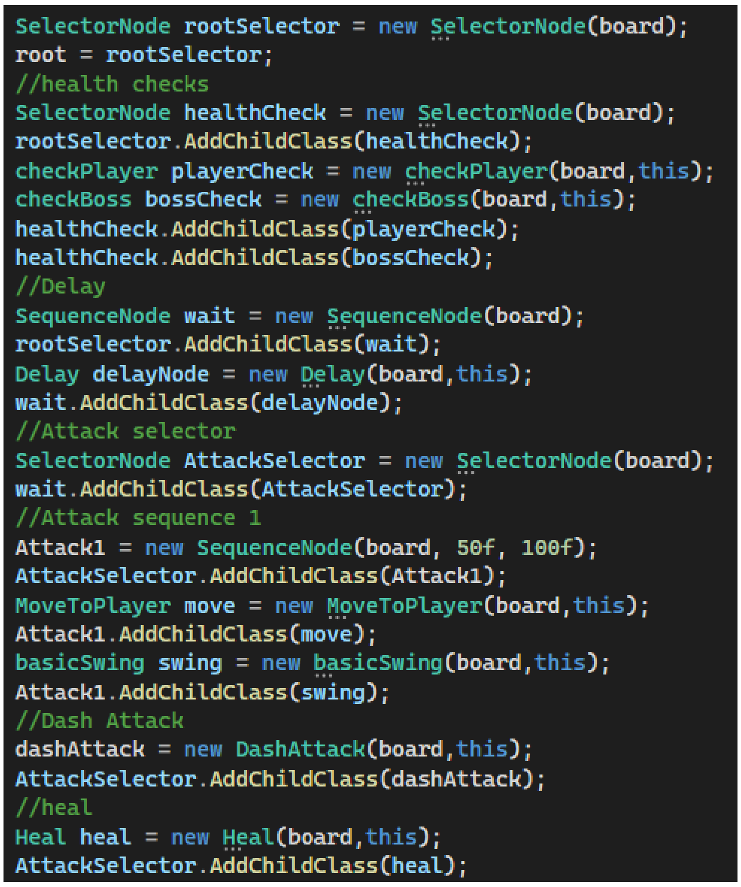


```
6 references
public abstract class CompositeNode : Node
{
    protected List<Node> children;
    protected int childIndex = 0;
    3 references
    public CompositeNode(BlackBoard board ) : base(board)
    {
        this.board = board;
        children = new List<Node>();
    }

    11 references
    public void AddChildClass(Node child)
    {
        children.Add(child);
    }

    4 references
    public override void Reset()
    {
        childIndex = 0;
        for (int i = 0; i < children.Count; i++)
        {
            children[i].Reset();
        }
    }
}
```

Figure 4: Code snippet: Nodes

Despite planning out the tree beforehand I did run into situations where I had to deviate from the plan, for example, when planning I had not taken downtime into account. When I first finished the behaviour tree, I noticed that the boss was just immediately jumping from action to action without waiting. I proceeded to add in a new action that ran after an action is completed. That action caused the boss to slowly approach the player, before they then choose another action and attack the player again. Due to the modular nature of behaviour trees, this process was simple, and I was able to insert this delay action into the tree without causing any issues, demonstrating to me why the behaviour might have been a good choice for this project.

The other option I was considering for the AI was a finite state machine, and in my opinion, the behaviour tree felt more modular and legible when adding in additional actions later in production. For a finite state machine, transitions and parameters would have needed to be added to each state to get a new state in. However, it was as simple as just adding it as a child to a preexisting node in the behaviour tree.



```
SelectorNode rootSelector = new SelectorNode(board);
root = rootSelector;
//health checks
SelectorNode healthCheck = new SelectorNode(board);
rootSelector.AddChildClass(healthCheck);
checkPlayer playerCheck = new checkPlayer(board,this);
checkBoss bossCheck = new checkBoss(board,this);
healthCheck.AddChildClass(playerCheck);
healthCheck.AddChildClass(bossCheck);
//Delay
SequenceNode wait = new SequenceNode(board);
rootSelector.AddChildClass(wait);
Delay delayNode = new Delay(board,this);
wait.AddChildClass(delayNode);
//Attack selector
SelectorNode AttackSelector = new SelectorNode(board);
wait.AddChildClass(AttackSelector);
//Attack sequence 1
Attack1 = new SequenceNode(board, 50f, 100f);
AttackSelector.AddChildClass(Attack1);
MoveToPlayer move = new MoveToPlayer(board,this);
Attack1.AddChildClass(move);
basicSwing swing = new basicSwing(board,this);
Attack1.AddChildClass(swing);
//Dash Attack
dashAttack = new DashAttack(board,this);
AttackSelector.AddChildClass(dashAttack);
//heal
Heal heal = new Heal(board,this);
AttackSelector.AddChildClass(heal);
```

Figure 5: Code snippet: Behaviour Tree

In conclusion, I found using behaviour trees for the creation a boss fight to be very effective. I found it useful to be able to create individual nodes for each ‘sub-action’ in the AI. Keeping movement and attack actions separate would allow for a more modular setup where the move nodes could get repeated as per requirement. With better time management or additional time, I believe I could have better shown how effective the behaviour tree system is for the creation of boss fight. I would attempt to increase the amount of modular reused code in the creation of additional action combinations, as well as possibly breaking actions down into different categories and having an additional layer of selectors for more variety. Despite this, I do feel as though my AI, while unpolished does a good job at demonstrating why a behaviour tree might be the superior choice of system for a boss enemy AI. The behaviour tree is an impressively modular and expandable system, making it an ideal choice for AI of this nature. It is not the most simple system to understand initially, however once one knows how it works, it becomes increasingly simple to expand and adjust the system without having to worry about nodes not fitting in.

4.3 Utility Theory Evaluation

Utility theory was major logical component of the programming model. I found that having an individual score for each action that appraises it against others allowed me to give much more control to the AI than I otherwise would have. It allowed me to make the decision system much more advanced than just checking conditions in each node to see if an action could be completed. It meant that action parameters could overlap, so that multiple action were capable of being performed the behaviour tree runs the most applicable. To provide the utility score to the behaviour tree, I used a function that returns an integer that refers to the appropriate action, this function uses a number of parameters from the blackboard to calculate a score for each action. One issue I found originally, was that I was running the utility score function every time the behaviour tree was run; it took me while to realise that this was causing the behaviour tree to try and change action whilst one was still running, causing actions to either not end or never properly start in the first place. By moving the call for the utility function to after the completion of an action or sequence, this issue ceased.

For each of the attack actions I used player health, distance, random and repetition as factors for their utility score. I did find that making scores like this to calculate the utility of an action, was a lot harder to balance than other more basic ways of selecting a node to run. I found myself spending a lot of time changing numbers around slightly to try and make sure that one action wasn't used exclusively over the others. Despite my original thoughts that the random factor would be the most complicated to balance, I found that the range and health factor were much more of a challenge.

In testing, I found that the ideal range and health for each action had by far the largest impact in the weighting of each action. This however, might be due to the format in which I calculated the scores for attack actions: they are incremented by the difference between the ideal distance to the player and the actual distance to the player, and the same is done for the health: they are then incremented by a random number and a score based on repetition. The lower the score output, the higher the chance it will be run. However, this causes an action close to its ideal range to be prioritised much more than the others, leading to the repetition factor sometimes being the only cause to run a different one. In hindsight, I would instead design the system so that the score is kept within a unit range (0 – 1) where the ideal factors for actions are less significant than they are currently as to produce more variety in the player experience. The two included attack actions are: a dash, and a move to and attack sequence. While the dash can be performed at any distance, the move to and attack has a maximum range. I decided to do this because I found initially that if you kept your distance from the boss when playing it would sometimes just start walking towards you and never change its mind. To reduce the chance of this happening, I added in that maximum range. I was initially worried that this might cause the boss to only use the dash attack. However, I found that the dash would then put the boss within range for the other attack. Another change I made to stop this issue was to add in a catch to the move to node; I decided to add in a condition that if the player isn't reached with a set period, it would cancel the action and select a different one allowing the fight to continue.

The action that I had the most trouble with initially was the heal action. I found that the boss would lean towards self-healing before all other actions once reaching a low enough health. To stop this happening, I decided to add a restriction to the heal action, where once it has been run, it can be run again until a certain number of actions have been run since.

As a whole I would credit the use of utility theory in the model to be a major factor in the modularity and reusability of the model in different scenarios. By creating a utility score system to judge each action, provided you maintain a single format you are able to use as many different parameters and factor as you require. The heal used completely different parameters than the other actions in the system. However, by maintaining a consistent format within the scores, it could slot in with the others fine. It is a useful tool that allows the programmer to grant the AI more appeared intelligence as it is capable of utilising a larger and more diverse set of parameters to help it decide what action to complete.

4.4 Player Controls and Experience Evaluation

The player controls and experience sections were an area of the project that I didn't think would cause many issues to begin with, combined with the fact that the focus of the programming model was on the backend of the AI. I decided that I would not need to allocate as much time to it as the other components of the artefact. This turned out to be probably the largest miscalculation I made during the design phase. I started to make the visual and interactable elements of the artefact during the third sprint, assuming that I would be able to finish it during that time and would be able to focus on making improvements to the rest of the project. I made simple models in engine to be the boss and the player models, where I would then use transformations to provide a playable environment. I began with the player, where I built a simple input system that moved the player object around the game, later adding in rotation to match. This was fairly simple, but once I tried to add in an attack, it got more complex. I had decided to use quaternion rotation to move the players sword and arms to mimic a vertical swing attack, however, I ran into lots of issues with axis alignment and getting the arms to line back up with the model after attacking. After spending the nearly half a sprint attempting to get the transformation working, I decided to change the swing to a horizontal one, which I found to be much simpler to implement. The player is not only equipped with an attack but also a dash to help with dodging the boss's attacks. The dash, whilst providing the movement expected, also doesn't not have an additional transformation. Despite this causing a lack of clarity to the player, the focus was on the boss not the player controls and I felt that the action was clear enough to be understood.

The boss's actions themselves proved equally as challenging to represent without proper animations. The boss has three different actions it can use:

The heal action doesn't have any kind of animation or transformation and if I had extended the development period, I would have liked to have had some kind of animation to represent the heal, as the player currently has no way of telling that the boss has used that action.

The sequence attack that uses move to and swing. The boss uses normal movement and directional rotation for moving around, however, I attempted to have the boss spin on the spot for the attack itself. This was not entirely effective. While the boss sometimes spins as intended, it sometimes will begin to spin and snap back to face its original direction. If I had made better predictions as to the time frame of certain systems, I would have liked to have fixed this issue and made sure the attack was better and more consistently telegraphed.

The dash attack is, I feel, the most effectively made transformation. The boss turns fast to face the player and speeds towards them, the purely kinetic nature of this action means that it is more suited to simple transformations the other actions I included.

I had decided not to use traditional animations and models due to personal inexperience with the area as well as feeling it would be out of scope for the time frame. Whilst I am dissatisfied with the player experience at the end of development, I maintain the principle that the project held a majority focus on the backend AI systems of the boss enemy, not the player controls or visual experience. Under that context I do feel that the visual element of the artefact, while not impressive itself, demonstrates the systems beneath it, and does that functionally.

Chapter 5: Conclusion

The intention of this project was to attempt to fill a gap in research surrounding programming models/frameworks specifically for game AI, relating to boss enemies. I wanted to ensure that the system designed would be capable of functioning in a number of different scenarios, making it modular and expansive so that any boss enemy could be based off the same initial backend design. I aimed to outline the system in a way that meant that other programmers might then be able to base their own boss AI off the model that I presented. At the end of development I feel that the AI I made to demonstrate the model could have been made more effectively. Better time management and more experienced estimations would have made the development period smoother. Additionally, creating the visuals elements of the artefact in the manner I did, is not something I would recommend or do again myself. Visual presentation aside, I feel that the creation and evaluation of the backend model itself could be deemed a success. I was able to use the design I produced to create an AI that demonstrated the key features of the model, and I was able to evaluate the systems based on my personal experience in using them for the artefact.

Over the course of the project, I have learned a lot. I now understand the importance of detailed planning and accurate time estimations and management. The biggest change I would make if repeating the project would be to ensure that the decomposed components of the project were more accurately planned, and to ensure that a stricter time management plan be implemented. Despite the issues, I have learnt a lot about the different systems and features that are included in game AI and feel a lot more comfortable implementing and understanding them; I leave the project with a greater desire to continue researching and creating game AI in the future.

References

Agriogianis, T. (2018). *The Roles, Mechanics, and Evolution of Boss Battles in Video Games*.

BioWare. (2014). *Dragon Age: Inquisition* [Computer software]. BioWare.

Bourg, D. M., & Seemann, G. (2004). *AI for Game Developers*. O'Reilly Media, Inc.

Charles Edeki. (2015). Agile Software Development Methodology. *European Journal of Mathematics and Computer Science*, 24–27.

DaGraca, M. (2017). *Practical Game AI Programming*. Packt Publishing Ltd.

Lebedeva, E., & Brown, J. A. (2020). Companion AI for Starbound Game Using Utility Theory. *2020 International Conference Nonlinearity, Information and Robotics (NIR)*, 1–5. https://doi.org/10.1109/NIR50484.2020.9290164

Livermore, J. A. (2007). Factors that impact implementing an agile software development methodology. *Proceedings 2007 IEEE SoutheastCon*, 82–86. https://doi.org/10.1109/SECON.2007.342860

Siu, K., Butler, E., & Zook, A. (2016). A Programming Model for Boss Encounters in 2D Action Games. *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, *12*(2), Article 2. https://doi.org/10.1609/aiide.v12i2.12891

Steve Rabin, Kevin Dill, Christopher Dragert, David Graham, Anthony Francis, Sebastian Hanlon, Cody Watts, & Mike Lewis. (2017). *Game Ai Pro 3* (1st ed.).

Steve Rabin, Michael Dawe, Steve Gargolinski, Luke Dicken, Troy Humphreys, & Dave Mark. (2013). *Game Ai Pro* (1st ed.).

Vorderer, P. (2003). *EXPLAINING THE ENJOYMENT OF PLAYING VIDEO GAMES: THE ROLE OF COMPETITION*.

Wood, A., & Summerville, A. (2019). *Understanding Boss Battles: A Case Study of Cuphead*.

Appendix

Figure 1: Initial development timeline

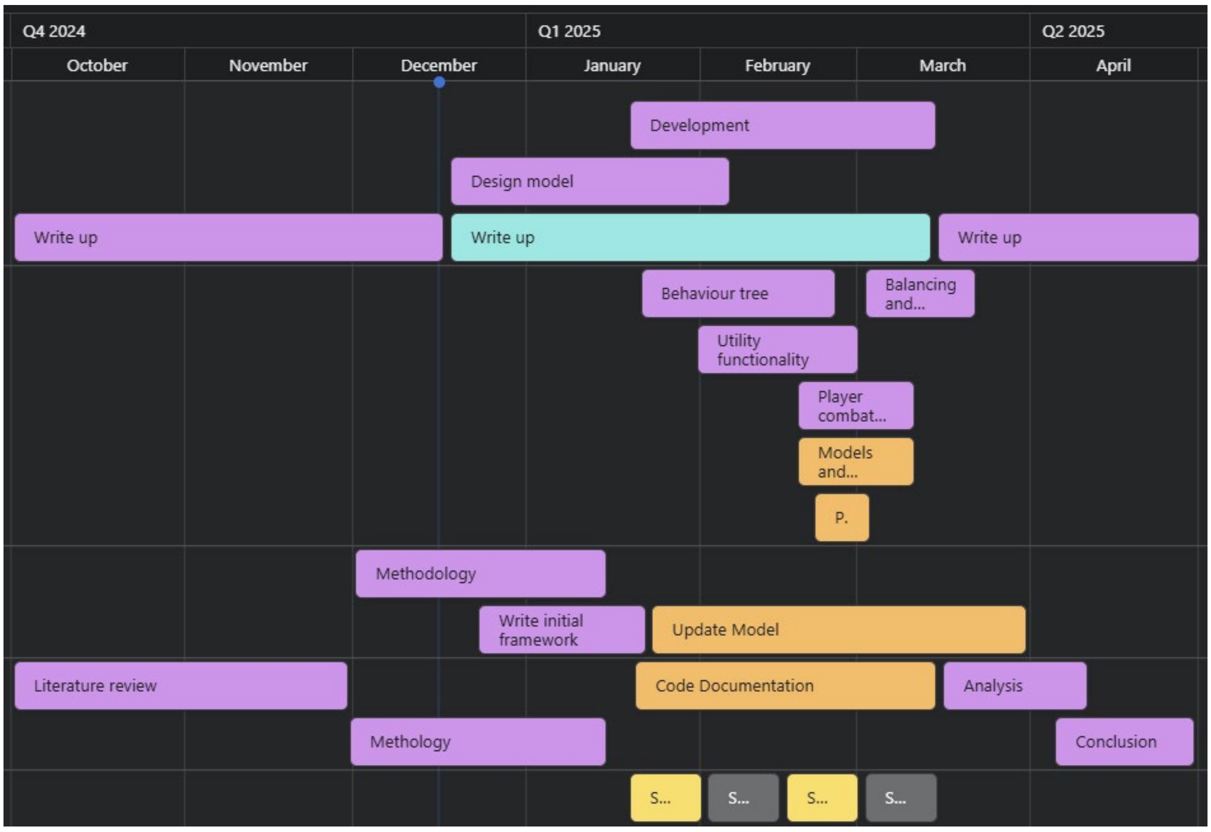


Figure 2: Behaviour tree design

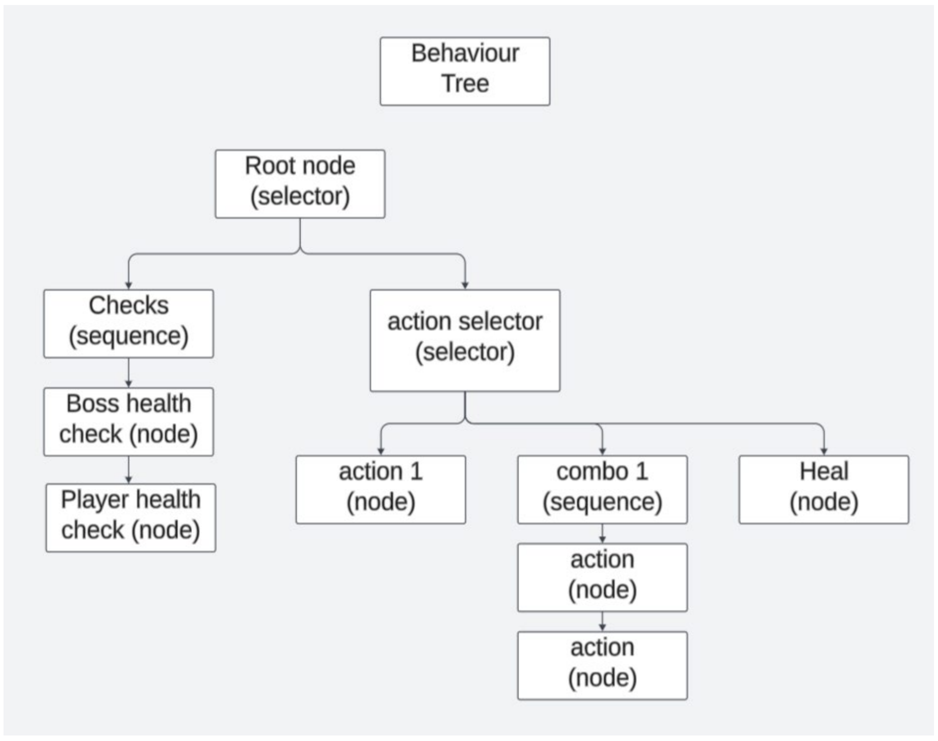
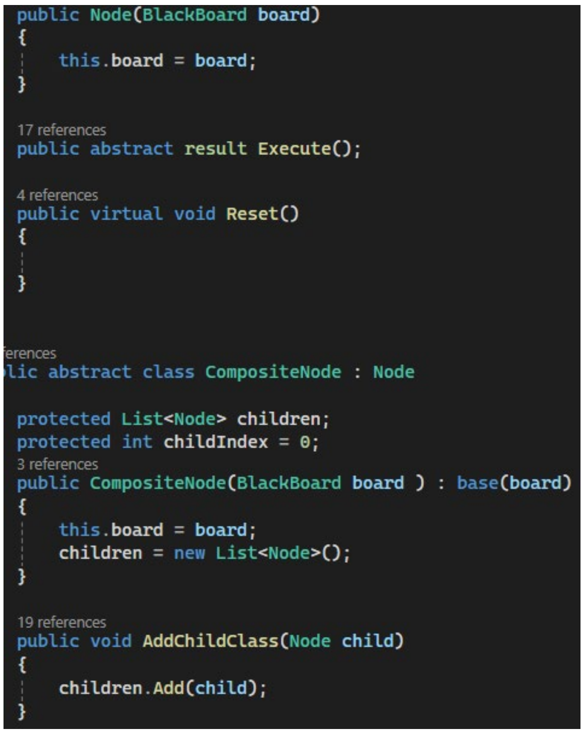
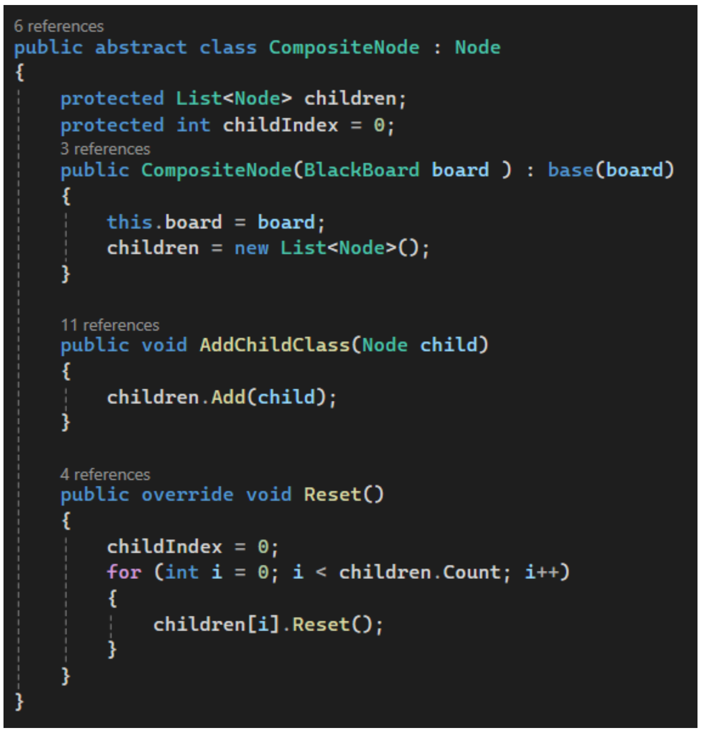


Figure 3: Code snippet: Nodes



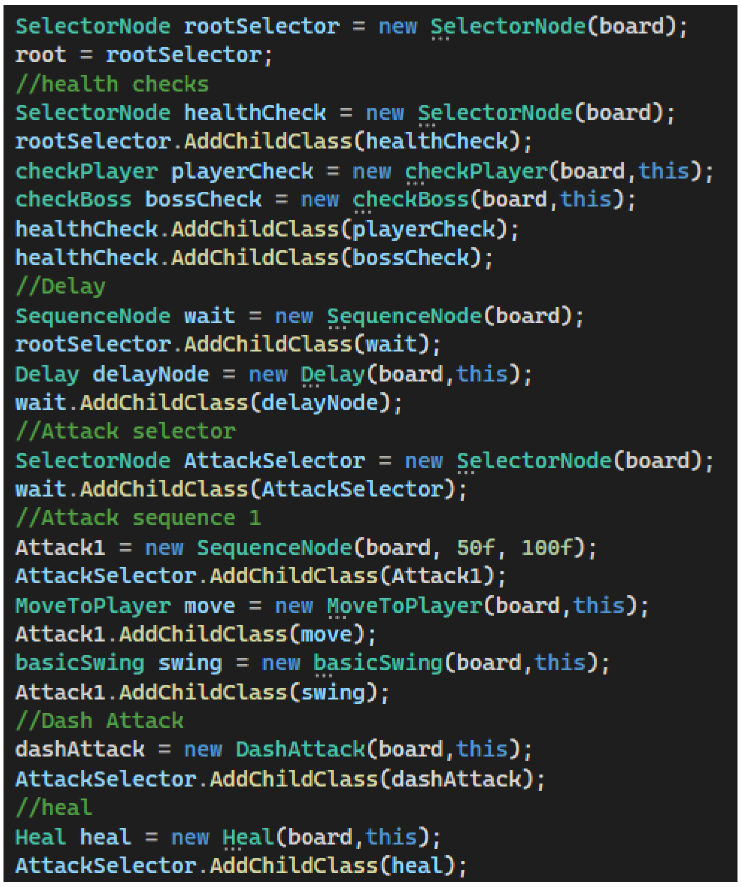
```
public Node(BlackBoard board)
{
    this.board = board;
}

17 references
public abstract result Execute();

4 references
public virtual void Reset()
{

}


ferences
lic abstract class CompositeNode : Node

protected List<Node> children;
protected int childIndex = 0;
3 references
public CompositeNode(BlackBoard board ) : base(board)
{
    this.board = board;
    children = new List<Node>();
}

19 references
public void AddChildClass(Node child)
{
    children.Add(child);
}
```

Figure 4: Code Snippet: Nodes



```
6 references
public abstract class CompositeNode : Node
{
    protected List<Node> children;
    protected int childIndex = 0;
    3 references
    public CompositeNode(BlackBoard board ) : base(board)
    {
        this.board = board;
        children = new List<Node>();
    }

    11 references
    public void AddChildClass(Node child)
    {
        children.Add(child);
    }

    4 references
    public override void Reset()
    {
        childIndex = 0;
        for (int i = 0; i < children.Count; i++)
        {
            children[i].Reset();
        }
    }
}
```

Figure 5: Code snippet: Behaviour tree



```
SelectorNode rootSelector = new SelectorNode(board);
root = rootSelector;
//health checks
SelectorNode healthCheck = new SelectorNode(board);
rootSelector.AddChildClass(healthCheck);
checkPlayer playerCheck = new checkPlayer(board,this);
checkBoss bossCheck = new checkBoss(board,this);
healthCheck.AddChildClass(playerCheck);
healthCheck.AddChildClass(bossCheck);
//Delay
SequenceNode wait = new SequenceNode(board);
rootSelector.AddChildClass(wait);
Delay delayNode = new Delay(board,this);
wait.AddChildClass(delayNode);
//Attack selector
SelectorNode AttackSelector = new SelectorNode(board);
wait.AddChildClass(AttackSelector);
//Attack sequence 1
Attack1 = new SequenceNode(board, 50f, 100f);
AttackSelector.AddChildClass(Attack1);
MoveToPlayer move = new MoveToPlayer(board,this);
Attack1.AddChildClass(move);
basicSwing swing = new basicSwing(board,this);
Attack1.AddChildClass(swing);
//Dash Attack
dashAttack = new DashAttack(board,this);
AttackSelector.AddChildClass(dashAttack);
//heal
Heal heal = new Heal(board,this);
AttackSelector.AddChildClass(heal);
```