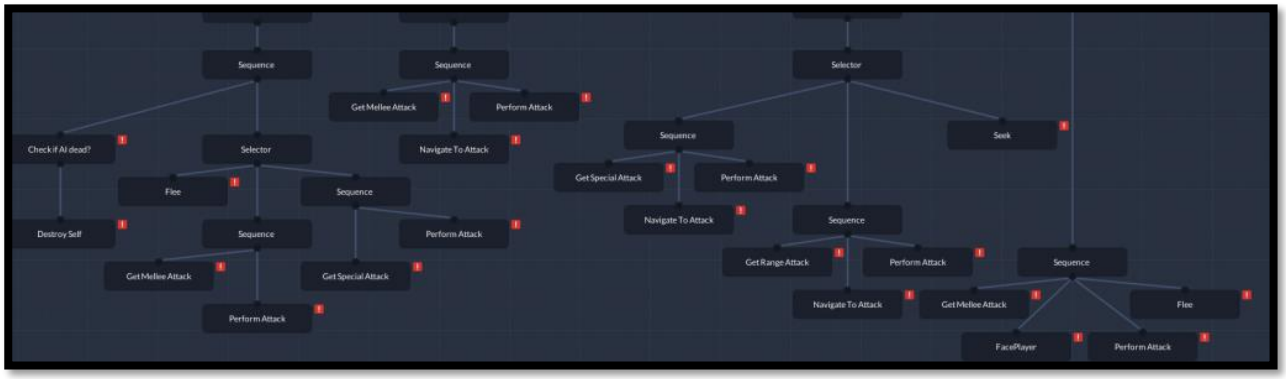
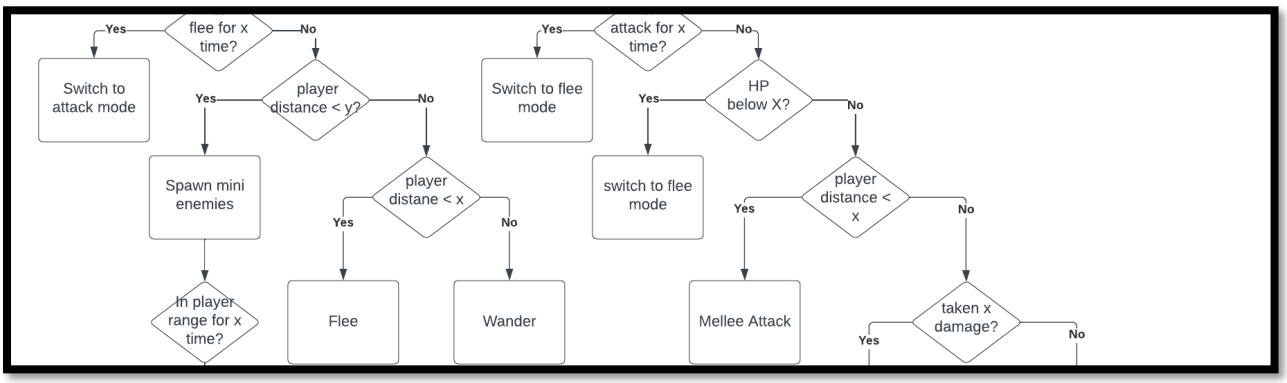
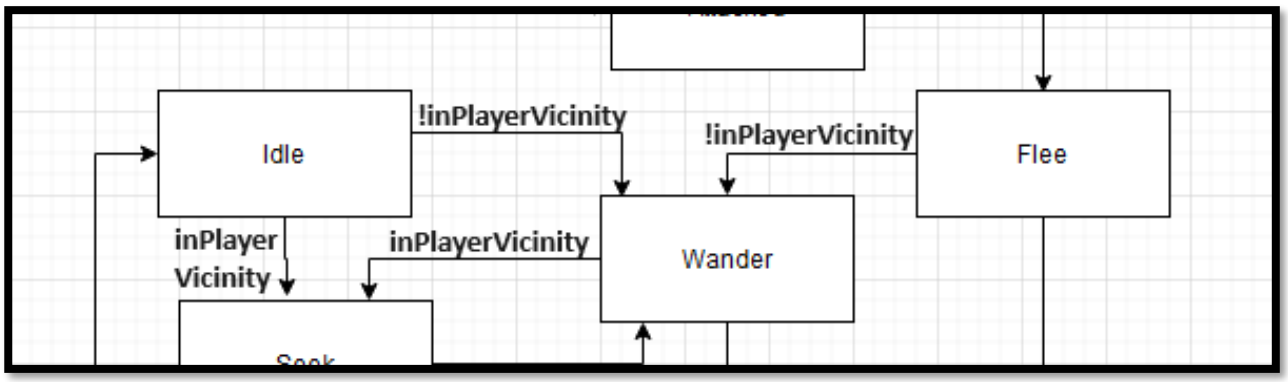


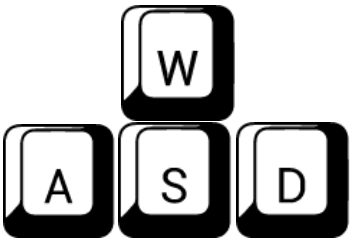




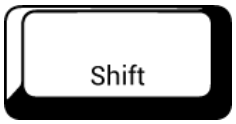
Decision making algorithms in Battle AI creation.

Robyn Botham

Player controls



Move



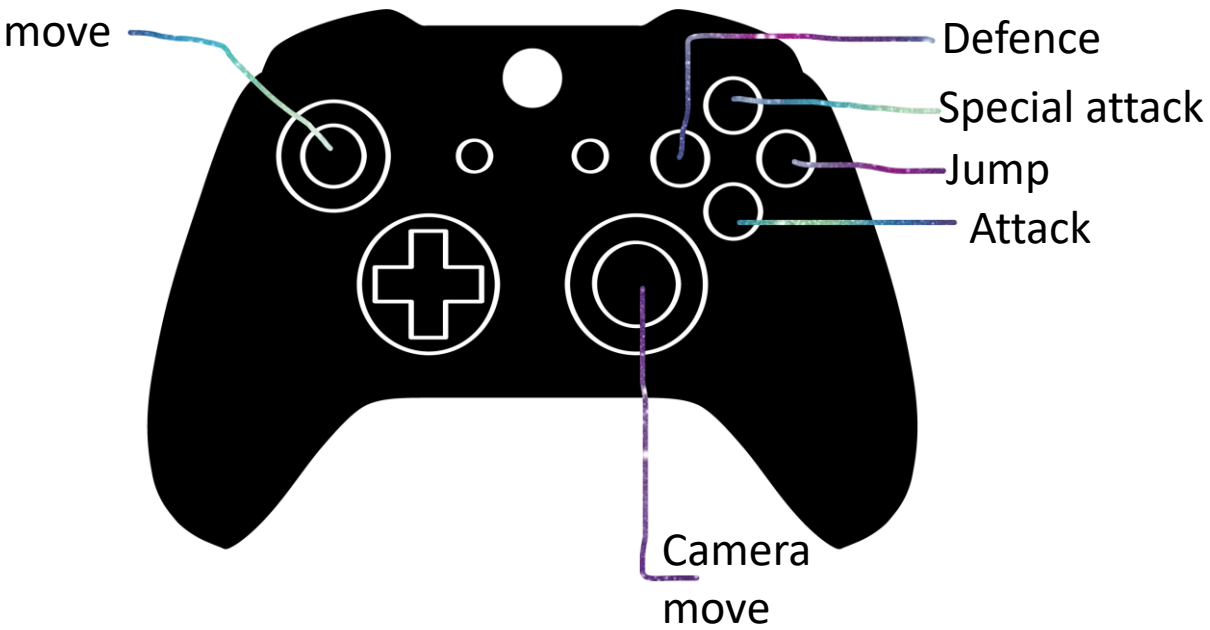
Attack

Jump

Defence

Special Attack

Move camera



Introduction

My project was about looking into decision making algorithms for Ais in battle situations. I then created a battle system and combat based AIs which could reasonably play against a human player using these algorithms.

The inspirations for the project and the battle style I was emulating is the battle style commonly found in JRPGs such as: Tales of Vesperia (Bandai Namco, 2008) and Kingdom Hearts (Square Enix, 2002). These include more closed battle spaces where strategy can be focused on more than pathfinding and navigation of complicated surroundings, and often very narrative based battles.

At the beginning of the project, I set out a list of aims I wanted to achieve by the end:

❖ **Steering behaviours**

The AIs will have the ability to navigate their surroundings.

❖ **Decision making algorithms**

To explore multiple decision-making algorithms to create an AI which can be fought against by a human player.

❖ **Player controls**

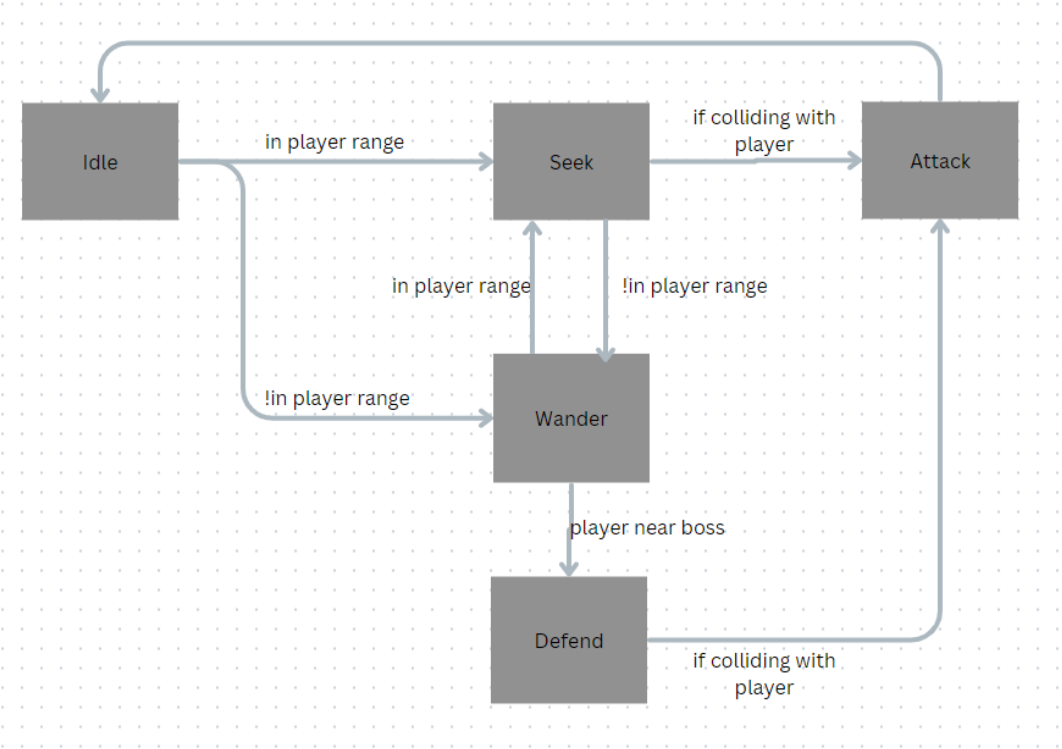
The player should be able to navigate the scene, attack and interact with enemies.

❖ **Health system**

The player and AI can both attack each other and cause damage if the attack hits.

❖ **Attack implementation**

Multiple types of attack which can be used by the AI in different situations, implemented in a way which makes it easy to add more if wanted.



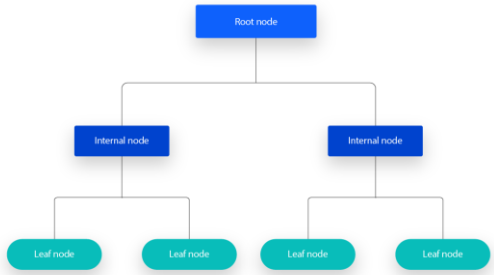
The preproduction phase of the project was used to investigate different decision-making algorithms.

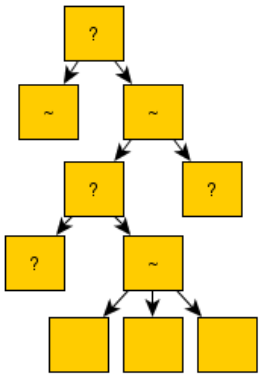
Finite state machines (FSM) separate possible behaviours done by an AI into states which all have their own functionality. Each state has transitions that when certain conditions are met the AI will swap to a different state [1].

FSMs can only be in one state at a time which can lead to predictable behaviour without the inclusion of **fuzzy logic**, which can adjust the chances of transitioning to another state, so that it's not always definite. This may be done by randomising values used in the transitions between instances of the AI or multiplying the values, or having multiple states which could be transitioned to with different chances of which will be, and the chances could be altered between instances of the AI to further randomise it [2].

There was also the possibility of using **fuzzy state machines** which allow for more complex behaviour by allowing the AI to be in multiple states at once, at different degrees called “active levels”[3]. These active levels are how committed to the state the AI is at any given time, and the update for all states which are currently active will be called [4]. This gets rid of transitions between states, which means adding states and transitions can be less complex than basic FSMs but can make it harder to get the desired behaviour.

Decision trees use a basic binary tree pattern to decide behaviour based on conditions. The AI traverses down the tree from a root node, checking whether conditions are true or false until it reaches a root node which will perform an action. This approach shares many of the same issues as basic FSMs in that it can create very predictable behaviour, and the more lead nodes added the more conditions will have to be checked before an action can be performed, causing possible performance issues [5].





Behaviour trees follow a tree pattern like decision trees but have more functionality. The tree also does not need to be traversed from the root each time, as each node can return one of three states: success, failure or running, running will recall the node until it returns either success or failure, or is aborted.

Behaviour tree nodes have different types: control flow nodes, decorators, conditions and execution nodes.

Control flow nodes decide how the tree is traversed. Selector nodes traverse their children either left to right or randomised and will continue to try each child node until one succeeds, it will only return failure if all children fail. Sequence nodes run all their children, usually from left to right but this can also be randomised, as soon as one child fails it will stop and return failure [6].

Decorators allow for actions to be rerepeated, affecting the return value of their children, cooldowns and time limits to stop nodes running if they don't return success of failure after a set time [7].

Conditions checks whether something in the current state of the world are true or false and decide actions based on this [6].

Execution nodes are the actual behaviours the AI will perform [ibid].

Goal oriented action planning (GOAP) is an approach where an AI can plan a sequence of actions to reach a goal, based on its current state and the end state. Pathfinding is used to find the best path of actions to the desired state, but it means different agents and different situations can complete different sequences of actions, creating more realistic behaviour, this approach was used in F.E.A.Rs AI, alongside an FSM [8].

The algorithm works backwards from the goal state, looking at what conditions need to happen to get there compared to the Ais current state, it will then try to find actions which will produce the correct conditions. E.g. if the goal is to go into a room, you may need to: check if the door is unlocked, see if they have a key, find a key if not, etc. None of these actions would be prearranged but instead arranged by the AI, actions will be weighted by how hard they are to do, and the AI will pick the path with the least actions to achieve the goal, or the path with the least weight [9].

Decision Algorithms used

The different algorithms discussed have different best use cases that must be considered in development.

Finite state machines for example are extremely common but as prementioned can lead to predictable behaviour, but as well as this they are difficult to use for a complex AI, due to the number of transitions which must be programmed to add a new state. They can be extremely good for testing, because each state can be tested on its own, but testing transitions can be harder and handling transitions to make them look natural can get harder the more there are. Therefore, FSMs are better suited to simpler enemies, often with a shorter lifespan to stop the player having the chance to learn their pattern.

Decision trees can appear even more predictable than finite state machines and are overall less efficient. The project originally explored use of decision trees, which was dropped due to it creating rigid behaviour, unlike behaviour trees it also doesn't allow for behaviour sequencing which can make it difficult to get an AI to perform a single goal, and due to the fact that the tree is run every frame, behaviour thrashing was common due to less control over transitions between states (e.g. in a state machine transitions can be delayed or some states can't be transitioned to at all without an intermediate state, this is not true for decision trees, unless this is included in a decision node. Decision trees may be more useful in simpler contexts with less possible variables, such as for picking most suitable attacks for in a situation, which is something the project will likely be extended to include in the future.

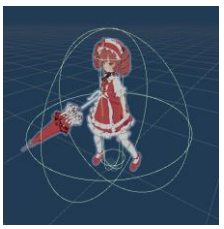
Behaviour trees work well in an AI setting because they can be very easily adjusted as a response for testing. Behaviour trees were used for the main AI in the artefact. Their biggest advantage was their visual nature, their logic could very easily be changed and rearranged to create better strategy or feeling for the player. Having control nodes and decorators allowed for far more controlled behaviour and the tree could easily handle more complex behaviours, and more randomised seeming behaviour. This approach was better than FSMs because a behaviour could be repeated in different parts of the tree but have different conditions leading up to it. It could also be picked randomly by a selector or sequence making behaviour feel less deterministic, whereas FSMs have fixed transitions that don't change based on the world state (e.g. behaving different based on HP to TP, as the main enemy does).

Goal oriented action planning was not used in the project, it has some pros and cons in the context of the artefact in terms of less deterministic behaviour and more realistic actions, but does give the AI programmer less control over the actions of an AI, and is better suited to larger and more complex environments, many of the actions in the artefact are quite simple, or don't require a large number of tasks in between, they're also performed symmetrically most times, would lead to the actions still appearing deterministic. GOAP may therefore be a too complex implementation for the context and may in some cases create worse behaviour due to that lack of control.

Player

The player has been given the functionality to use player input in multiple ways to fight the AIs in the scene.

Move – the player can move left, right, forward and backwards, the direction of movement is linked to the position of the camera, so they will always move relative to the camera rotation, rather than the worlds axis.

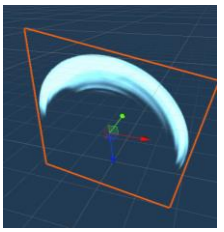


Camera Move – the camera is done with *Cinemachine* to create less jarring movement, as it moves relative to the player but not at the same speed. The player can spin the camera at a set distance from the player to view different parts of the scenery.

Jump/ double jump – the player can jump and double jump. Jumping is limited to two jumps until they collide with the floor again.

Attack – the player can melee attack the enemy, if an attack is performed while colliding with the enemy then it will apply damage to them and increase some of the players TP.

Special attack – special attacks take up the players TP but can be used to attack the enemy from a range and does more damage on average than a melee attack.



Defence – the player can defend themselves when attacked, this will reduce the effect of an attack on the players HP by a certain percentage. Its active so long as the defence button is being pressed down, but while it's active the player cannot also move or jump.

Steering behaviours

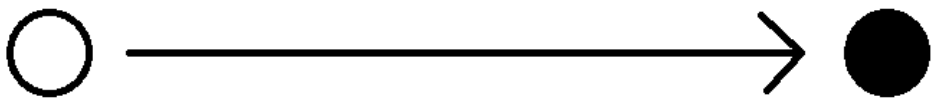
All of the steering behaviours are contained in a component which can easily be added to any Game Object which needs to navigate around the world and contain a NavMeshAgent component. To change which steering behaviour is being used, another component just has to call SetNewNavigation(), which starts the relevant coroutine.

SetNewNavigation has 4 overloads:

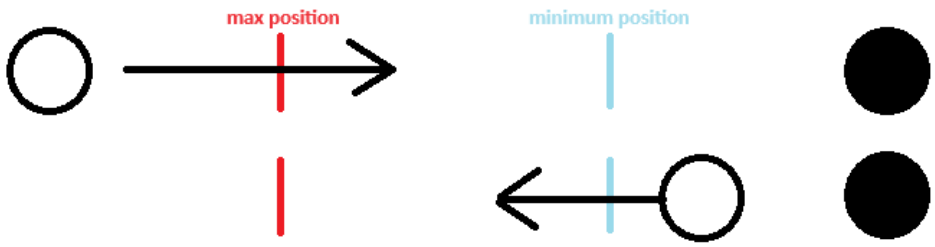
- `SetNewNavigation(steering behaviour, GameObject object)` - used to steer relative to an object.
- `SetNewNavigation(steering behaviour, Vector3 target)` – used to steer relative to a position.
- `SetNewNavigation(steering behaviour)` – used to start steering which doesn't require an object or position reference (e.g. wander and idle)
- `SetNewNavigation(Attack attack)` – used to start steering for an attack.

Seek object - seeks towards an object, this function updates the location it is seeking every frame so it can account for the object it's seeking's movement.

Seek location - seeks towards a set, vector3 location which is set at the beginning of the function.

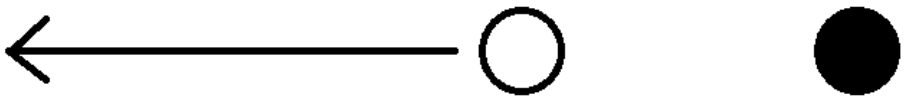


Seek to attack - each attack has a minimum or maximum distance from the player in order to perform it, this function navigates to somewhere between that range if necessary.



Flee object - flees from an object until it's a set distance away, accounts for the object's movement.

Flee position - flees from a set vector3 position until a defined distance away.



Wander - picks a random point within the battle circle and navigates towards it.

Finite State Machine

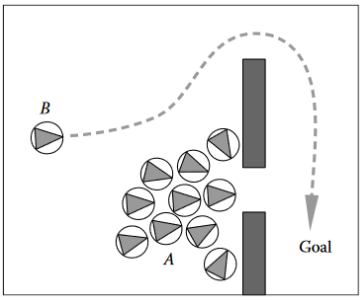
The mini enemies which are spawnable by the main enemy run using a finite state machine with 8 different states: Idle, Seek, Wander, Defend, Attack, Flee and Attacked. There is a 50% chance that the main enemy will spawn in 1-3 mini enemies when the player is playing offensively (damage or attack number over a defined amount in the last minute.) There is a limit on how many mini enemies can be spawned in the scene at any given time.

The mini enemies were designed to be smaller, simpler enemies to distract the player. Their purpose, rather than survival and attack strategy like the main enemy, their only aim is to attack the player and defend the main enemy.

The addition of the three states: idle, flee and wander, were to increase how natural the enemy felt as well as fairness. Idle and flee are both to make it so the mini enemy is not constantly in a state of hunting down the player, wander is there to make it so if you go out of range the enemy ‘forgets’ the player, which thematically would be expected of a clone enemy.

Fuzzy logic

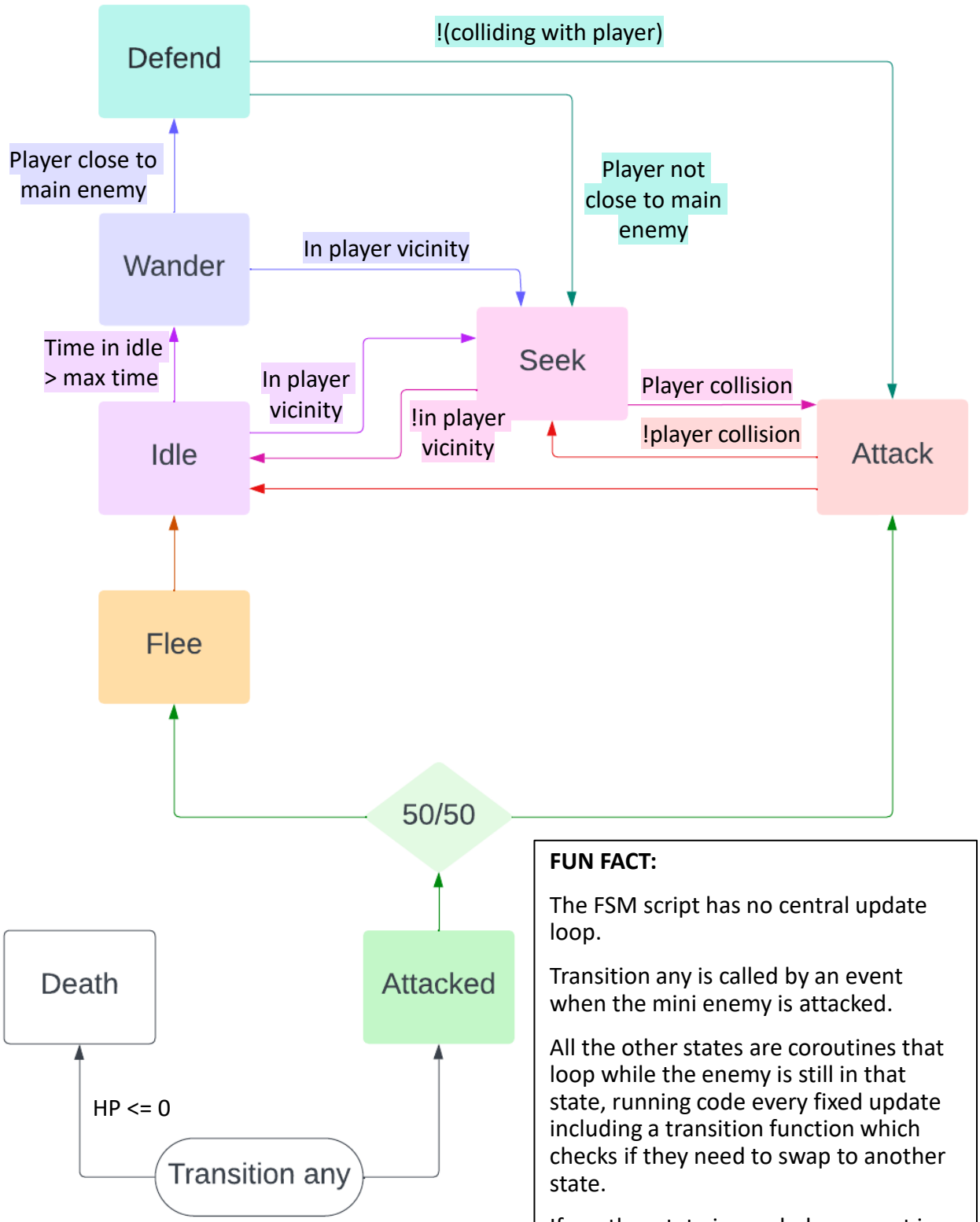
An issue came up in development was the mini enemies behaving symmetrically. There was occasions where all enemies would end up behaving the same at the same time and becoming a hoard following you, rather than seeming intelligent. There was multiple possible solutions to the problem considered:



Congestion maps – instead of following an idealised path the pathfinding the pathfinding considers congestion and discourages movement through crowded areas by increasing the cost in those areas for the A* algorithm [10]. This approach wasn’t feasible within the time limit as it would require making custom pathfinding instead of the type already built into Unity's navmesh agents.

- Fuzzy logic – reduce the chance of multiple agents having a shared goal or changing how they act while in that role, by changing values between agents or changing the chances of whether you transition to a state [11]. A possible way of implementing this considered was to have randomised intelligence values acting as a percentage, which controlled a thinking time before going into that state or the intelligence value being the chance of picking the best option or dumber option.

In the end, fuzzy logic was used. The distance to seek, defend, the Ais speed, as well as an artificial break before beginning actions to simulate thinking time were used to avoid them acting the same.



FUN FACT:

The FSM script has no central update loop.

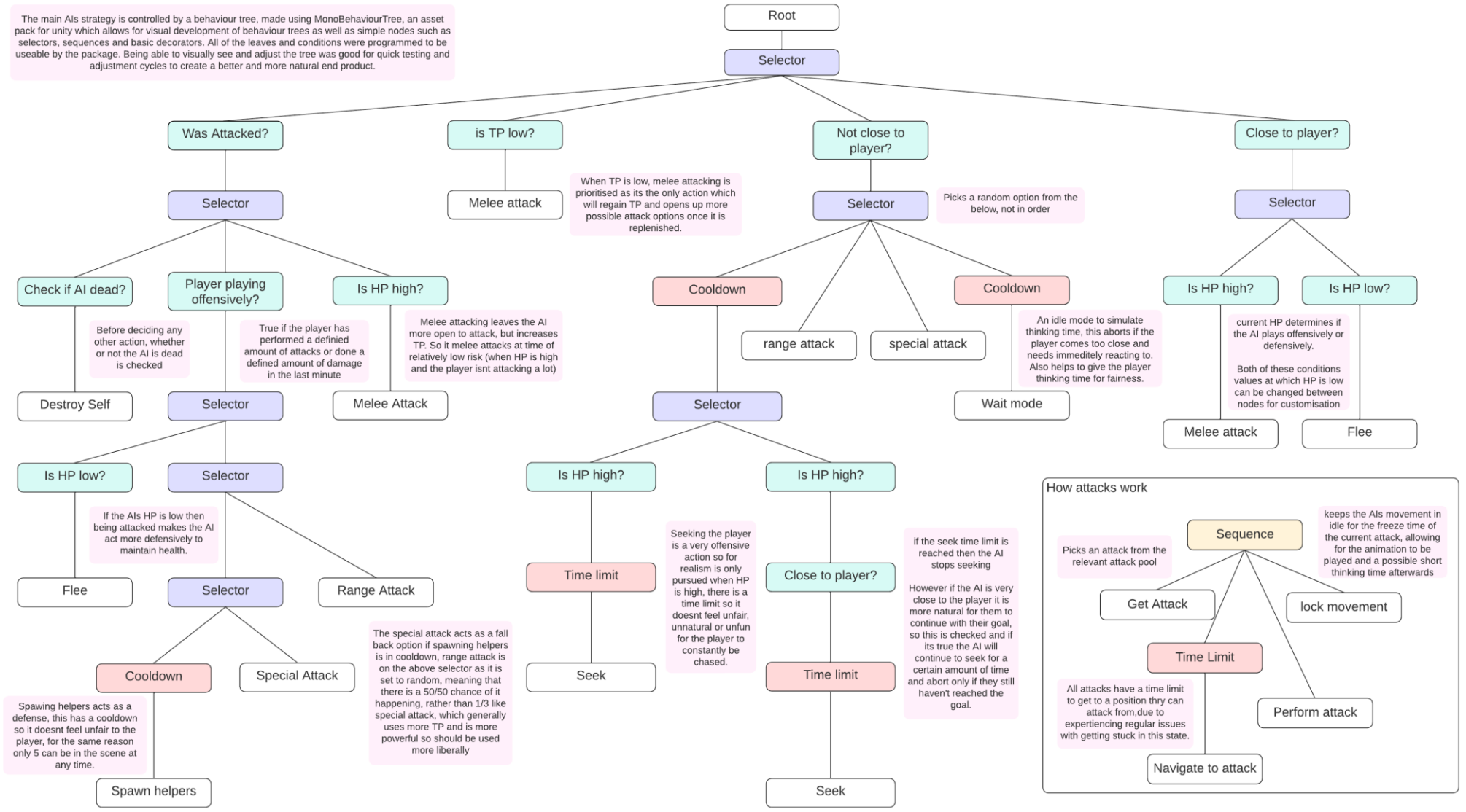
Transition any is called by an event when the mini enemy is attacked.

All the other states are coroutines that loop while the enemy is still in that state, running code every fixed update including a transition function which checks if they need to swap to another state.

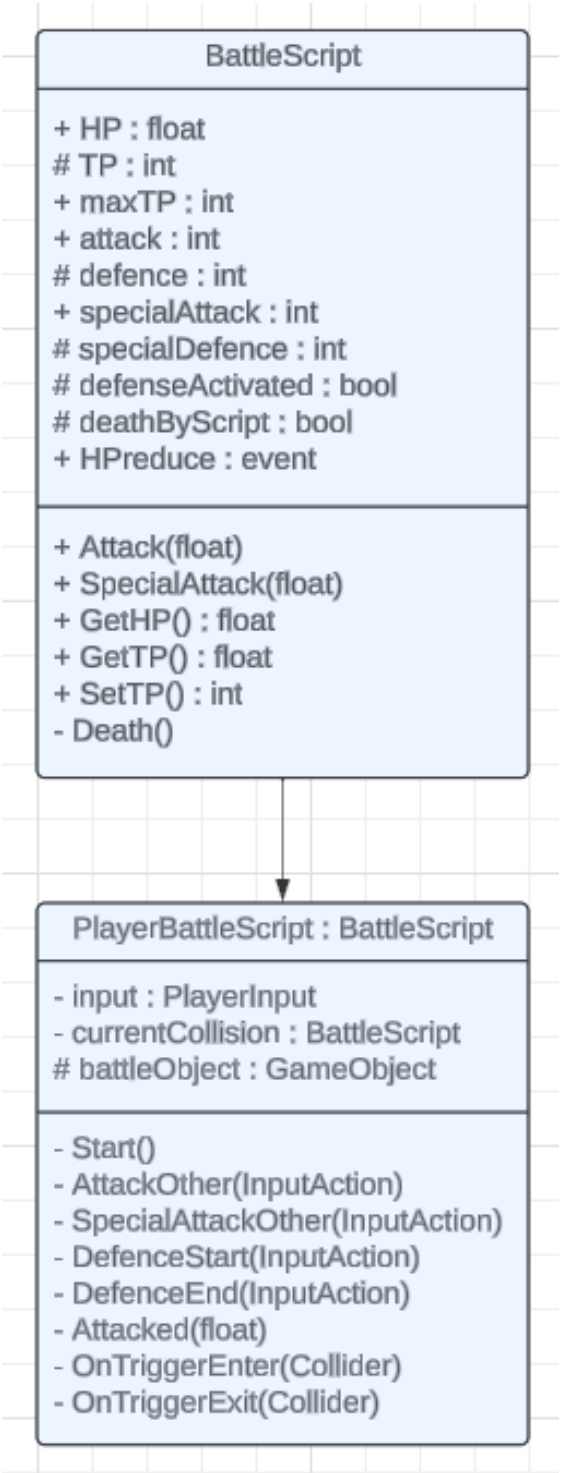
If another state is needed, an event is called which will call a StateChange function within the same script!

Behaviour Tree

The main AIs strategy is controlled by a behaviour tree, made using MonoBehaviourTree, an asset pack for unity which allows for visual development of behaviour trees as well as simple nodes such as selectors, sequences and basic decorators. All of the leaves and conditions were programmed to be useable by the package. Being able to visually see and adjust the tree was good for quick testing and adjustment cycles to create a better and more natural end product.



Battle System



Any damage dealt in the game is done through that objects battle script. All Ais in the scene use the base BattleScript, this allows you to:

- set battle stats

- handling of damage taken

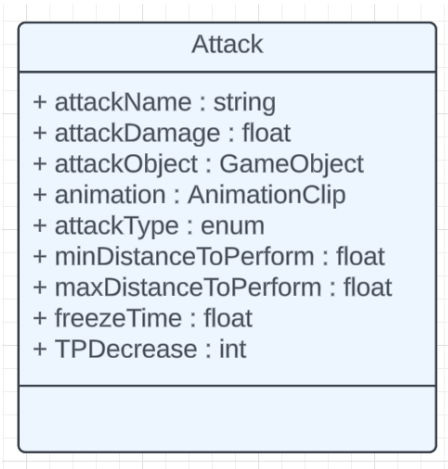
- get and set the AIs current HP and TP.

Attack implementation is done by the AI's relevant strategy script, which only uses battle script to check health, check TP or set TP when an attack is performed. It also shoots off an event when attacked which can be responded to accordingly.

The players battle script class inherits the functionality of its parent class, but with the added functionality of handling the players battle related inputs. This made it easier to check battle related variables and modify damage values without referencing another script as those values were already member variables, unlike with the AIs strategy scripts.

Attacks

Attacks are stored in a scriptable object containing all the information about an attack. Attacks can be easily added to an enemy via simply adding them here, which gives the AI access to them.



Attack name – the name of the attack

Attack damage – how much base damage the attack will cause without any modifiers

Attack object – if the attack spawns an object (common in range attacks) it should be included here.

Animation – the animation performed by the character when the attack is called.

Attack type – an enum with 3 types: melee, range and special which are used to split up the attack pools, different attacks are used in different scenarios.

Minimum distance to perform – the closest to the player you can be to perform the attack.

Maximum distance to perform – the furthest away from the player you can be to perform the attack.

Freeze time – after performing the attack movement and other actions will be locked until freeze time is over, this is usually at least the length of time it takes to perform the attack animation.

TP Decrease – how many tactical points are lost through performing the attack.

Reflection

The project has successfully implemented two decision making algorithms, explored another and investigated other decision-making algorithms, which were not used as they were not the best use case. The Ais can react to their surroundings and the players actions and respond to them reasonably, though not the level of complexity I had originally intended.

Many of the issues affecting the AIs in their final submission state is aesthetically, such as animations, attacks, sound effects etc. These were not the purpose of the artefact, but have a large effect of player experience, as a lot of game AI focusses on the illusion of intelligence rather than actual intelligence, therefore aesthetic issues can be jarring and make the AI feel unnatural, even though it might be acting in the way I intended, it also makes it appear under a professional level[12].

Some issues in why the AIs did not reach my expectations, or not as many algorithms were explored in code as initially intended, it due to decision trees being dropped from the project. The development of the decision tree which is not included took up a large portion of development time, this was a good learning experience and shows how different algorithms can affect the end product, as well as learning more about the characteristics of that form of decision making. It has significantly furthered my own knowledge, knowing when and if to use that algorithm in the future, there is an intention to use it for attack picking, but did lead to a significant loss of time for an AI which was not included.

A large learning curve to the project was learning that often the best way for the AI to act, is not the way that should be implemented. If an AI behaves in the ‘perfect’ way, this can feel unnatural or unfair, or even jarring for the player and reduce enjoyment. Therefore, artificial waiting, thinking time, etc. had to be added to the AI, perfecting how long idle times should be and the fuzzy logic to make the AI decide ‘dumber’ actions while making it still seem natural, was one of the hardest parts of developing the AIs behaviour and continues to have issues in the final implantation, though has vastly improved. This was a large cause of dropping decision trees and is also a large fault in the mini enemies' finite state machines.

Moving forward with development after the module is finished, I want to work on picking better attacks for a situation, likely via using a decision tree. I also want to work on making it more aesthetically enjoyable, adding more attacks and more complex and fun battle strategies for the player. Some of these additions were not part of the core project, as they are not related to decision making, but through the development process I have learnt how important they are to the overall illusion and feeling developers are trying to create for the player

Sources

1. Özer, M. C. *Finite State Machine (FSM) Based Agent*. Available at: http://www.mcihanozer.com/tips/artificial-intelligence/finite-state-machine-fsm-based-agent/
2. Sayantini. 2023. What is Fuzzy Logic in AI and What are its Applications?. Available from: https://www.edureka.co/blog/fuzzy-logic-ai/
3. J. Li, Z. Wang and Y. Zhang, 2011. "An Implementation of Artificial Emotion Based on Fuzzy State Machine," 2011 Third International Conference on Intelligent Human-Machine Systems and Cybernetics, Hangzhou, China pp. 83-86, doi: 10.1109/IHMSC.2011.90.
4. Roberts, P. 2022. Artificial Intelligence in Games, Taylor & Francis Group. ProQuest Ebook Central, https://ebookcentral.proquest.com/lib/staffordshire/detail.action?docID=7007113.
5. What is a Decision Tree?. IBM. Available at: https://www.ibm.com/topics/decision-trees
6. Simpson, C. 2014. Behavior trees for AI: How they work. Game Developer. Available at: https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work
7. Decorators. BehaviourTree.dev. Available at: https://www.behaviortree.dev/docs/nodes-library/decoratornode/
8. Thompson, T. 2020. Building the AI of F.E.A.R with Goal Oriented Action Planning. Game Developer. Available at: https://www.gamedeveloper.com/design/building-the-ai-of-f-e-a-r-with-goal-oriented-action-planning
9. Owens, B. 2014. Goal Oriented Action Planning for a Smarter AI. Envatotuts. Available at: https://gamedevelopment.tutsplus.com/goal-oriented-action-planning-for-a-smarter-ai--cms-20793t
10. Pentheny, G. 2015. Advanced Techniques for Robust, efficient crowds. Chapter 17. Available at: http://www.gameaipro.com/GameAIPro2/GameAIPro2_Chapter17_Advanced_Techniques_for_Robust_Efficient_Crowds.pdf
11. Roberts, P. 2022. Artificial Intelligence in Games, Chapter 7. Taylor & Francis Group. ProQuest Ebook Central, https://ebookcentral.proquest.com/lib/staffordshire/detail.action?docID=7007113.
12. Rabin, S. 2017. The illusion of intelligence. Chapter 1. Taylor & Francis. Available at: http://www.gameaipro.com/GameAIPro3/GameAIPro3_Chapter01_The_Illusion_of_Intelligence.pdf

Assets

AI Navigation 1.1.4 – Unity Technologies Inc. [package]

Cinemachine 2.9.5 – Unity Technologies Inc. [package]

Input System 1.6.1 – Unity Technologies Inc. [package]

Toki no Kagi – Selkione [music] (available at: https://selkione.itch.io/nitro-music-rumble)

Jacob – Jupifox [3D Model] (available at: https://jupifox.itch.io/jacob-game-ready-character)

Asougi – Hitsuji 15 [3D Model] (available at: https://assetstore.unity.com/packages/p/asougi-205096)

Hyper Casual FX – Lana Studio [Particle Effects] (available at https://assetstore.unity.com/packages/vfx/particles/hyper-casual-fx-200333)

MonoBehaviourTree 1.2.0 – Qriva [package] (available at https://assetstore.unity.com/packages/tools/behavior-ai/monobehaviourtree-213452)

Simple Sky – Synty Studios [Skybox] (available at: https://assetstore.unity.com/packages/3d/environments/simple-sky-cartoon-assets-42373)

Lowpoly Environment – Polytope Studio [3D environment model] (available at: https://assetstore.unity.com/packages/3d/environments/lowpoly-environment-nature-free-medieval-fantasy-series-187052)