Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

# Simulation of artificial inteligence for games

Matěj Gargula

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
May 2023

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Gargula** Jméno: **Matěj** Osobní číslo: **492145**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačové hry a grafika**

## II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

**Simulace herní umělé inteligence**

Název bakalářské práce anglicky:

**Simulating artificial inteligence in games**

Pokyny pro vypracování:

Zmapujte metody pro simulování umělé inteligence postav v počítačových hrách. Soustřeďte se na metody vhodné pro tvorbu modelů schopných reagovat na aktuální stav hry, které zároveň poskytují konzistentní výsledky, jako jsou například stromy chování (behavior trees).
Implementujte nástroje pro vytváření modelu umělé inteligence postav a její následné vyhodnocování v herním enginu Unity. Pro implementaci využijte UI Toolkit a soustřeďte se na intuitivní vytváření modelu, jeho snadnou modifikaci, znovupoužití a vizualizaci vyhodnocování modelu. Pomocí implementovaných nástrojů vytvořte knihovnu nejméně pěti často se opakujících modelů chování nehráčských postav. Vytvořenou knihovnu využijte ve vlastním herním projektu, který bude jasně demonstrovat různé konkrétní modely chování umělých postav.

Seznam doporučené literatury:

[1] Ian Millington, John Funge. Artificial Intelligence for Games, 2nd edition. CRCR Press, 2009.
[2] Colledanchise, M., & Ögren, P. (2018). Behavior trees in robotics and AI: An introduction. CRC Press.
[3] Colledanchise, M., Parasuraman, R., & Ögren, P. (2018). Learning of behavior trees for autonomous agents. IEEE Transactions on Games, 11(2), 183-189.
[4] Marcotte, R., & Hamilton, H. J. (2017). Behavior trees for modelling artificial intelligence in games: A tutorial. The Computer Games Journal, 6(3), 171-184.
[5] Sekhavat, Y. A. (2017). Behavior trees for computer games. International Journal on Artificial Intelligence Tools, 26(02), 1730001.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

**doc. Ing. Jiří Bittner, Ph.D. Katedra počítačové grafiky a interakce**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **17.02.2023** Termín odevzdání bakalářské práce: **26.05.2023**

Platnost zadání bakalářské práce: **22.09.2024**

| doc. Ing. Jiří Bittner, Ph.D.<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

# Acknowledgements

I would like to thank my friends for providing me with their time for needed distraction. I would also like to thank my family and girlfriend for their mental support. Most of all, I would like to thank my supervisor, doc. Ing. Jiří Bittner, Ph.D., for enabling me to work on a video game-related topic for my project and for pointing me in the right direction when needed.


# Declaration

I hereby declare that this project represents my own work, and I have cited all the used literature and sources I have used.

Prague, May 22, 2023

# Abstract

Almost every game today contains some sort of artificial intelligence simulation. One of the most popular is the simulation of artificial intelligence for non-player characters. These simulations are about creating algorithms that can make the characters appear human-like or animal-like. My project is focused on researching the possible techniques used to simulate the decision-making process of the non-player characters and implementing one of these techniques to create an artificial intelligence decision-making system in the Unity engine.



**Keywords:** NPC, AI simulation, behavior trees, AI decision-making, video games, Unity



**Supervisor:** doc. Ing. Jiří Bittner, Ph.D.
Praha 2,
Karlovo náměstí,
E-421



# Abstrakt

Téměř každá dnešní hra obsahuje nějaký druh simulace umělé inteligence. Jednou z nejpoužívanějších je simulace umělé inteligence nehráčských postav. Tyto simulace jsou o vytváření algoritmů, díky nimž mohou tyto postavy působit jako lidé nebo jako zvířata. Můj projekt se zaměřuje na zmapování možných způsobů používaných k simulaci rozhodovacího procesu nehráčských postav a na implementaci jedné z těchto technik k vytvoření systému rozhodování umělé inteligence v enginu Unity.



**Klíčová slova:** NPC, AI simulace, stromy chování, AI rozhodovací procesy, video hry, Unity

**Překlad názvu:** Simulace herní umělé inteligence

# Contents

# Figures

# Tables

# Chapter 1

# Introduction

Artificial intelligence (AI) is a branch of computer science that focuses on developing algorithms and software that enable machines to perform tasks normally requiring human-level intelligence. As defined in [1], AI refers to the ability of computers to perform thinking tasks that humans and animals are capable of.

In game development, AI simulation plays a vital role in creating realistic and immersive game experiences. By implementing AI algorithms for non-player characters (NPCs), game developers can create characters that appear to behave like humans or animals. This can include NPCs that exhibit intelligent decision-making, emotional responses, and natural movement patterns.

Today almost every game contains some artificial intelligence simulation. Each game might require a different approach. Games use artificial intelligence commonly for [12]:

- Non-player Opponents

  Opponents should exhibit realistic attack behavior. This can include decreasing or increasing the opponent's aggression or retreating when a big threat is located.

- Non-player Teammates

  Non-player teammates should act in a coordinated, supportive manner. This means that the AI teammate characters should be able to act in a synchronized way with the player and not act as an obstacle.

- Support and Autonomous Characters

  This may include things like generating realistic crowd behavior, which can interact with players in a realistic manner.

- Commentary or Instruction

  This includes systems that make sure the player can move through the game world smoothly. An example of these systems can be determining if the player is stuck and is in need of a hint on how to progress further.

## 1.1 Non-player Character

Non-player character (NPC) [11] is any character or game object in a game that is not directly controlled by a player. NPCs can be used to simulate ally or enemy characters.

A good AI system for NPC simulations has a set of requirements [12]:

- Smart but not omniscient behavior

  NPC should behave as a real human being would. This creates an immersion for the player. When the AI starts to make decisions that make no sense, the immersion starts to break.

  The AI also should not cheat or have access to all information about the game state. For example, Guards in a game should not be able to see through walls.

- Consistency

  The NPC should behave consistently to generate the impression that it embodies a believable character. To do this, we want the NPC to act in a controllable way. Unpredictable AI is difficult to work with. It also helps during the debugging process. This is also why most games do not use advanced AI techniques like machine learning for decision-making.

- Effectivity

  Games often use a large number of NPCs. Because of this, the AI must be fast to handle multiple instances of NPCs.

- Fast Adaptability

  NPC needs to adapt quickly to the game's current state when the state of the game changes, the NPC should be able to adapt.

## 1.2 Important Games in History of Game AI

These are some of the games that have been important to the history of AI in game development [1] [13] [14] [15]:

**Pacman** [Midway Games West Inc., 1979] was one of the first games with a non-player character that started behaving like a thinking creature. This was achieved by a simple AI technique: a state machine. At each junction, the ghosts (enemies in the game) took a semi-random route, depending on their current state. For example, if the ghosts were in chase mode, they would try to choose a path leading to the player.

**Goldeneye 007** [Rare Ltd., 1997] brought many improvements with the same technique of using state machines with a small number of states for each non-player character and adding a sense simulation system. Characters would now also react to the changes in their environment. Characters could see if some of their colleagues were missing or dead.

**Creatures** [Cyberlife Technology Ltd., 1997] was one the first games to have AI as the game's main point. Creatures still have one of the most complex AI systems seen in games. Each creature in the game had its neural network-based brain to control it. But even with the highly complex systems for AI simulation, the individual creatures would still be acting stupidly or nonsensically.

**Halo 2** [Bungie Software, 2004] was one of the first successful high-profile games for which the behavior tree technique was described in detail. This inspired other game developers to use them in their games as well.

**F.E.A.R** [Monolith Productions, 2005] was one of the first games to use a Goal Oriented Action Planning (GOAP) AI. This first-person shooter psychological horror game was able to simulate very believable human-like actions, which resulted in a very immersive and memorable experience.

**XCOM: Enemy Unknown** [Firaxis Games, 2012] introduced the utility-based decision-making system in game development. This system gave each action a measure of "usefulness" to every possible action. With this system, each NPC can select the most effective action based on various specified factors such as the number of nearby enemies, distance to an objective, etc.

**The Sims 4** [Maxis, 2014] is famous for its AI simulation. Like **Creatures**, **The Sims 4** has an AI simulation as one of the game's main points. Each character in the game can reach almost human-like behavior. Each character in the game has particular needs and tries to satisfy them. The AI system in the game weighs all potential results of each possible action to find a new action most fit for the given situation. This creates a very believable experience.

**Red Dead Redemption 2** [Rockstar Games, 2018] utilized the behavior tree technique and several other methods to improve overall AI decision-making. The individual non-playable characters in this game each have their own personality, with mood states, and are capable of remembering and 'experiencing' past events to some extent. For example, if a player has attacked a certain character. that character can next time behave differently around the player.

## 1.3 Aim of this thesis

This thesis aims to conduct a research study on the various techniques used to simulate the artificial intelligence of video game characters. The ultimate objective of this study is to examine the underlying mechanisms of these techniques and gain a deep understanding of their potential applications in video game development.

One of the primary goals of this research study is to show the advantages and disadvantages of the various techniques. Additionally, I will also analyze the ability of these techniques to provide consistent and predictable results, which is essential in creating immersive and engaging game experiences.

To provide a practical demonstration of how AI characters are simulated in games, I plan to create an AI decision-making system in the Unity engine

based on behavior trees (one of the discussed techniques). This decision-making system will allow us to simulate the behavior of AI characters in a game-like environment, providing valuable insights into how AI simulation can be used in game development.

Furthermore, I also plan to develop an AI modeling tool that can be used to create, edit, and visualize the behavior tree models used in the decision-making system. This modeling tool will enable users to create and modify complex AI models more easily.

# Chapter 2

# Decision Making Techniques

Decision-making refers to [12] intelligent selection of an action for the agent to interact with the world. There are many different decision-making techniques, but each work on a couple of the same principles. Each agent processes a set of input data to generate the following defined action of the agent. Input data is the internal knowledge a character can possess or external knowledge the agent can acquire from the world, as seen in figure 2.1 [1]. The generated actions can then perform internal changes (changes concerning the agent) or external changes (changes within the game world)

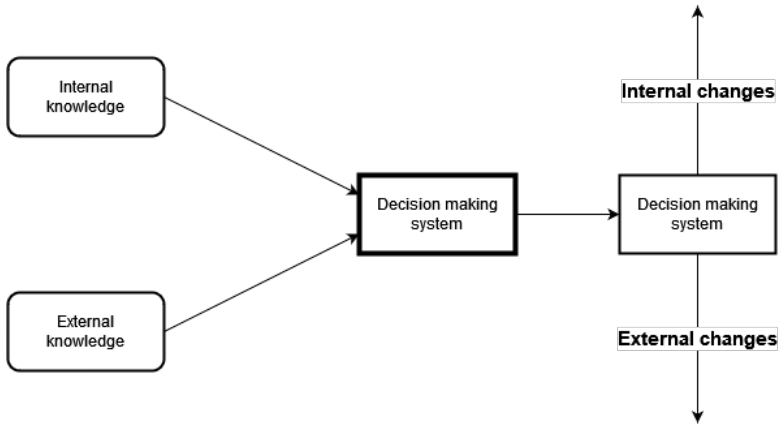



**Figure 2.1:** Decision making schematic

## 2.1 Decision Trees

Decision trees [1] are fast and easy to implement and understand. They have directed trees representing a list of nested if-else clauses used to derive decisions (i.e., produce a new action).

### 2.1.1 Structure of Decision Trees

Decision trees have two types of nodes. Non-leaf nodes or decision points are described with a condition. Each decision point has two child nodes or

outcomes. Leaf nodes represent decisions, conclusions, or actions to be carried out.

In figure 2.2, we see an example of non-player character behavior defined by a decision tree. This example defines the behavior of a guard whose job is to catch criminals. At the beginning of this decision tree, the guard runs a test to determine if a criminal is visible. If the guard cannot see a criminal, he continues patrolling around by selecting the "Patrol" leaf node. If he does see a criminal, he proceeds to the next decision point. This decision point calculates the distance to the criminal. The "Chase Criminal" leaf node is selected if the criminal is more than a meter away. We continue to the last decision point if the criminal is less than a meter away. In the last decision point, the guard checks to see if the criminal has been arrested. If not, the guard selects the "Arrest Criminal" node. Otherwise, he brings the criminal to jail by selecting the "Go to Jail" node.

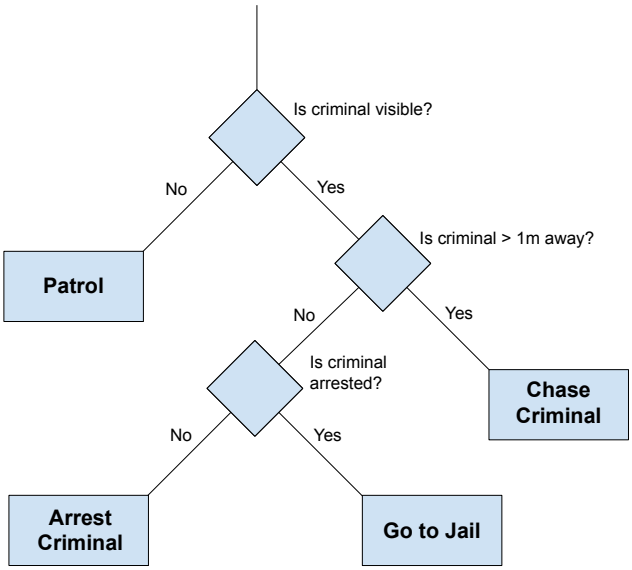



**Figure 2.2:** Decision tree example.

### 2.1.2 Advantages and Disadvantages of Decision Trees

The technique of decision trees has its advantages and disadvantages [3] [1].

The main advantage of decision trees is their modularity. Meaning that individual subtrees can be developed independently from the rest of the tree and then added later where needed. They are also easy to implement and offer good readability because of the hierarchical structure.

The main disadvantage is the mapping between the input and output of a decision point, which means that any small change in the input can make a big difference in the output of the decision. Because of that, the decision tree never has a consistent inner state. This is a big issue in game development because this can easily create problems where the AI can appear stupid.

Because of that, decision trees are usually used only on very simple models of behavior.

### 2.1.3 The Algorithm of Decision Trees

The algorithm of decision trees is simple and straightforward [1]. The tree starts with a root node which is a decision point. When a decision is required, the algorithm moves through the tree by decision points. At each decision point, a condition is evaluated from data known to the agent. This data is usually stored within the agent or is gathered from the game world. The condition check contains no boolean logic (i.e., checks cannot be joined together by AND, OR, etc.). The condition could, for example, be a boolean variable or calculation if the numeric value is within a given range, as seen above in Figure 2.2. The algorithm continues moving through the tree until it reaches a leaf node. After that, an action attached to the leaf node is carried out immediately.

## 2.2 State Machines

Game characters will often act in one of a limited set of ways. State machine [1] best represents this type of behavior with the help of additional scripting. State machines use information from the world around them and their internal information (state).

### 2.2.1 Structure of State Machines

A state-machine agent occupies only one state [1] [3]. Each state has a set of actions or behavior. While the agent remains in the same state, he will keep carrying out the same action or actions defined by the state. States are connected by transitions. Each transition moves the agent from the current state to the next target state, and each has a set of conditions. If the conditions are met, the transition to the target state occurs.

Figure 2.3 shows an example of a simple state machine. This state machine defines the same behavior used in a previous chapter 2.1.1. This time the guard character has three defined states: Patrol, Chase, and Go to Jail. At the beginning of the game, the guard starts in the Patrol state. If the guard spots a criminal, the condition "Can see Criminal" is met, and a transition occurs. This moves the guard from state Patrol to state Chase. This state has two different transitions. If the guard can no longer see the targeted criminal, a transition back to the Patrol state occurs. But if the guard manages to chase down the criminal and arrest him, a transition to the state Go to Jail takes place. In the Go to Jail state, the guard simply takes the criminal to jail. Once completed, the condition "Criminal Jailed" is met, and the guard transitions back to the Patrol state.

Figure 2.3 shows an example of a simple state machine with states: On Guard, Fight, Run Away. Each of these states has a set of transitions, for

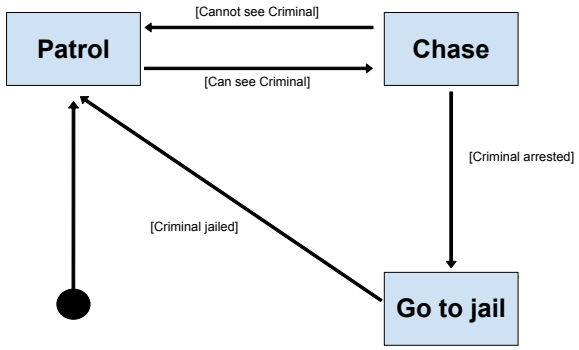



**Figure 2.3:** State machine example.

example, transition "See small enemy" from state On Guard to state Fight.

### 2.2.2 State Machine Algorithm

Individual states are typically implemented with an interface [1]. This way the state can include any specific code. The state machine itself keeps a record of all possible states and a reference to the current state. The state machines also store a series of transitions for each possible state. Each transition is also generic and usually is implemented as needed. The state machine has its update function, typically called every frame or iteration.

At each update function call, the state machine checks if any transitions have been triggered. Then the first transition that has been triggered is executed. After that, a list of actions for the current state is prepared. If another transition has been triggered, the transition is executed. Transitions can also have actions to be called or scheduled while transitioning to the next state. For example, equip a weapon before switching to the state Fight.

### 2.2.3 Advantages and Disadvantages of State Machines

State machines are one of the popular techniques for modeling decision-making for the AI agent. But this technique is still not perfect for every possible use. State machines have their own advantages and disadvantages [3].

The main advantage of state machines is that they have a very common structure used in many fields of computer science. They are very intuitive and easy to understand. Each state can also be implemented separately. Also, unlike decision trees, state machine always has a consistent inner state (the current state). This means that, unlike decision trees, small changes in the agent input data will not significantly impact the agent's overall behavior.

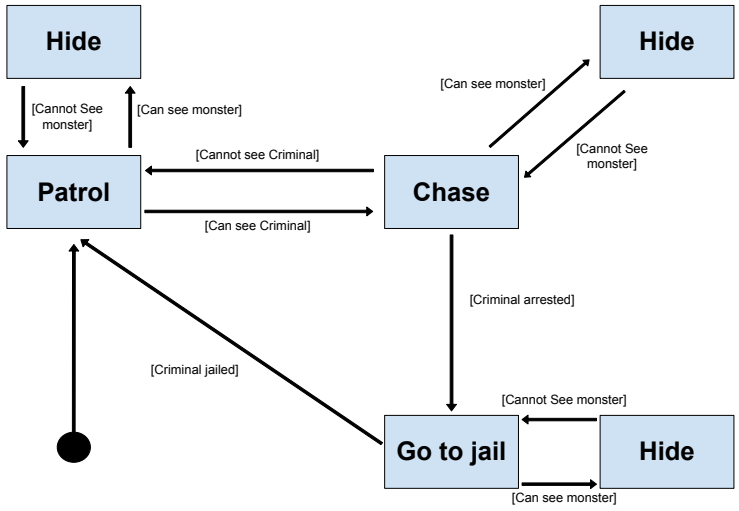
However, state machines have many drawbacks. A complex reactive agent needs many transitions, which means difficult maintainability and scalability. Adding or removing a single state often requires re-evaluating and re-implementing the whole state machine with many changes to other states and their corresponding transitions. This makes state machines susceptible to human design errors. Also, transitions often depend on internal variables. Because of this, they are impractical to reuse in a different sub-state machine or project.

### 2.2.4 Hierarchical State Machines

Hierarchical state machines [3] or State Charts [4] resolve issues of the standard state machines. They improve modularity and introduce a more organized hierarchical structure. They also introduce concepts of the parent state machine and child state machine that work as a sub-state of the parent state machine. The child state machine starts when the parent enters a state attached to it and stops when the transition to a different state is called.

State machines are useful tools, but it can be difficult to express some specific behaviors. One of the most commonly tricky behavior [1] is the "Alarm behavior".

Alarm behavior is a type of behavior that interrupts any currently running behavior and moves the character to a new state to handle some urgent task. After the task is done, the character transitions back to the original state. To demonstrate the problem of simulating an Alarm behavior, we will again use the simple state machine seen in Figure 2.3 and add a new Alarm behavior.




.

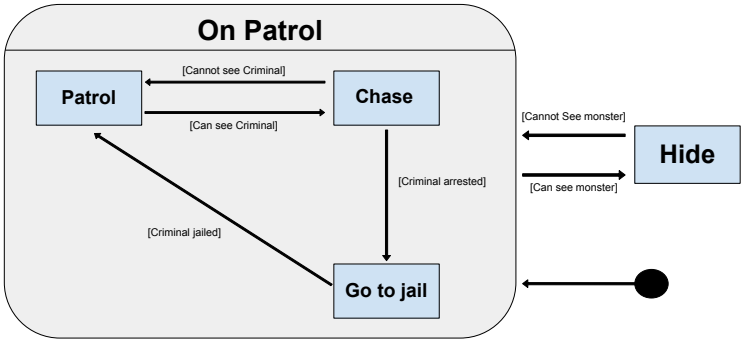
**Figure 2.4:** Basic state machine example with alarm behavior

Figure 2.4 shows the new extended simple state machine. In this state machine, we wanted to add a new behavior: If the character sees a monster, he should hide from it no matter his state, and when the monster leaves, the character should return to his last state. This can be difficult to express with a basic state machine, as seen in figure 2.4. To add the "Hide behavior", we must add three new states with the same functionality and six new state-dependent transitions (two for each state). Even more, problems are created when we want to introduce more Alarm behaviors. To add more alarm behaviors, we must specify which behavior is more important.

Information that is known to the agent.

To fix this issue, we can use hierarchical state machines and create a nested child state machine for the patroling behavior and a separate state for the hiding behavior. This behavior is defined in the hierarchical state machine in Figure 2.5.

When the character sees a monster, he will only transition from the state On Patrol to the state Hide. When the monster leaves and the transition back to the On Patrol state occurs, the character returns to the last state in the nested child-state machine since child-state machines keep track of the




.

**Figure 2.5:** Hierarchical state machine example with alarm behavior

last state. With this approach, we only needed to add a single state instead of the three in the previous example with a basic state machine. We can now also add even more alarm behaviors. To specify the importance of the behavior, we need to place it into the hierarchy (the highest state in the hierarchy has the highest priority).

We can see that thanks to these additions, it is possible to separate individual tasks in the state machines into sub-tasks. However, these sub-tasks may still depend on each other through state-dependent transitions. Because of this, maintainability is still an issue. Also, the state machine hierarchies must still be user-defined and can be challenging to edit. This means that even hierarchical state machines are still highly susceptible to human error.

## 2.3 Behavior Trees

Behavior trees [1] combine several decision-making techniques like hierarchical state machines, scheduling, planning, etc. Their most significant advantage is their ability to implement these techniques in a way that is easy to understand and create, even for non-programmers. This way, more team members can contribute to the creation of game AI.

### 2.3.1 Types of Tasks in a Basic Behavior Tree

Tasks in the behavior tree all share the same basic structure [5]. Each task contains an execution function. After this function is called, the task will schedule, execute or update actions defined by the task. After a given time

(at the start of a new frame, for example), the task returns a status code. The status code [5] will return either a code for success, failure, or often "running", indicating that the task is still not completed in any way. Some developers often add extra codes such as "error" for debugging. Each task can also call another sub-task to run its execution function.

| Task Type | Succeeds | Fails | Running |
|---|---|---|---|
| **Composites** | Depends on the child's return code | Depends on the child's return code | one or more child tasks are running |
| Decorator | Varies | Varies | Varies |
| Action | Upon completion | When impossible to complete | During execution |
| Condition | If condition is met | If condition is unmet | Never |

**Table 2.1:** Return codes of basic task types [2].

The basic Behavior tree consists of four main types of tasks [1]: Conditions, Actions, Composites, and Decorators. The table 2.1 shows the return codes of individual basic tasks and when the tasks return them. The following sub-sections explain the functionality, common implementations, and how are these types of tasks used [1] [5] [6] [2].

### Conditions

Condition tasks are a fundamental component of behavior trees and are essential in determining agents' behavior. Conditions allow the agent to test some properties and decide based on the result. For instance, a condition can be used to determine whether the agent is close enough to a target to perform an action or whether a certain enemy is in the field of view of the agent.

Conditions can be designed to check any property that can be evaluated, such as the distance between objects, the state of a game variable, or the number of enemies in the vicinity. Moreover, conditions can be created and parametrized to make them more reusable. This means the same condition task can be used in different parts of the behavior tree with different input parameters.

Each condition task in the behavior tree has a specific implementation that checks the property and returns a success or failure code but never a running code which can be seen in the table 2.1. If the condition is met, the task returns a success code, indicating that the behavior associated with that task can proceed. Conversely, if the condition is not met, the task returns a failure code, which can trigger the agent to move to the next task in the behavior tree.

Condition tasks are always placed as leaf nodes in the behavior tree, as they do not need to control sub-tasks. Their role is to determine the outcome of the condition and pass it to the parent node, which can then decide which

branch of the tree to execute next.

### Actions

As the name suggests, an action task contains an action, which can be anything the agent should be able to do. In the context of artificial intelligence in video games, actions refer to the actions that an agent can perform in the game world. These actions can be used to control various aspects of the game, such as movement, animation, and the inner state of characters. The implementation of each action is unique and requires individual task creation. However, unlike conditions, actions do not always have to return a code for success or failure. Instead, actions can also return a code "running" to indicate that the action is still being performed, shown in the table 2.1. This allows the behavior tree to handle actions that require multiple frames to complete.

As mentioned earlier, actions are connected as leaf nodes of the behavior tree. This means that they are always the final nodes in the tree and do not have any child nodes. The execution of an action is straightforward - once the action node is reached during tree traversal, the corresponding action is performed, and the control is returned to the parent node. If the action is successful, the parent node can continue traversing the tree. On the other hand, if the action fails, the parent node can try a different branch of the tree. If the action returns "running," the parent node will continue to execute the child nodes until the action returns a success or failure code.

In practice, actions can be quite complex and can involve several sub-tasks that need to be executed. These sub-tasks can also be organized into another simple behavior tree, which is referred to as a sub-tree. For instance, an action that controls the movement of a character may involve pathfinding, collision detection, and animation control. These sub-tasks can be implemented as child nodes of the action node, allowing for the creation of more complex behaviors.

Actions are a crucial component of behavior trees in video game AI. They allow agents to interact with the game world and perform tasks required to achieve their goals. Using actions and their associated child nodes, behavior trees can create complex, reactive, and intelligent behavior for the in-game agents.

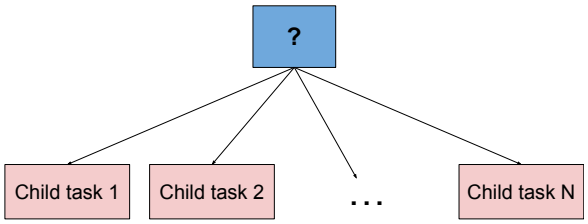
### Composites

Composites are used to control the flow of the behavior tree decision-making. They keep a record of child nodes. Child nodes can have any task attached to them (i.e., condition, action, decorator, or another composite).

| Composite type | Succeeds | Fails | Running |
|---|---|---|---|
| **Selector** | If one child succeeds | If all children fail | If one child returns running |
| **Sequence** | If all children succeed | If one child fails | If one child returns running |
| **Parallel** | If N children succeed | If M-N children fail | If all children return running |

**Table 2.2:** Return codes of common composite tasks [2].

Their behavior is based on the behavior of the child nodes. Composite tasks mainly group other tasks to create a specific behavior. Unlike action or condition tasks, composite tasks usually have only a handful of implementations because we don't need many of them to create a sophisticated behavior. There are three most common types of composite tasks [1][3]: selectors, sequences, and parallel.




.

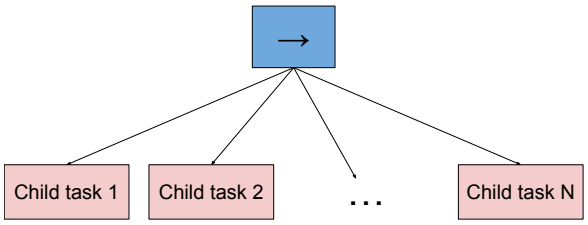
**Figure 2.6:** The structure of a selector task.

The selector task, also called the fallback task [3] [2], is a simple composite task. On each iteration, the selector runs one of its child tasks. The structure of a selector task is depicted in Figure 2.6 (the selector is usually marked by the "?" symbol [6]). As we can see in the selector algorithm 1 and also in table 2.2, if the child's task returns a "success" code, then the selector returns a "success" code. If the child returns a "failure" code or a "running" code, the selector returns a "running" code. If all child states return a "failure" code, then a "failure" code is returned. Each child task is commonly called from

**Algorithm 1** Algorithm of the Selector task

```
1:  function RunSelector
2:      for all childTasks do
3:          childReturnCode ← childTask.Run()
4:          if childReturnCode == Running then
5:              return Running
6:          else if childReturnCode == Success then
7:              return Success
8:          end if
9:      end for
10:
11:     return Failure
12: end function
```

left to right (in figure 2.6, child task 1 will be called first). This way, we can specify the priority of each child task of the selector. Selectors are commonly used to select the first successful from a pool of tasks based on the task's priority [2].




.

**Figure 2.7:** The structure of a sequence task.

The sequence task works in the opposite way of the selector task [2]. The structure of the sequence node is shown in Figure 2.7 (the sequence is commonly marked by the arrow symbol [6]). Like the selector, the sequence runs one of its children on each iteration. As we can see in the sequence algorithm 2 and table 2.2, if the child task returns a "failure" code, the sequence also returns a "failure" code. In the same way, if the child task returns a "running" code, the sequence also returns a "running" code. The sequence returns a "success" code only if all its children return a "success" code. The sequence node is usually used for tasks that should be executed or completed in a sequence.

**Algorithm 2** Algorithm of the Sequence task

```
1:  function RunSequence
2:      for all childTasks do
3:          childReturnCode ← childTask.Run()
4:          if childReturnCode == Running then
5:              return Running
6:          else if childReturnCode == Failure then
7:              return Failure
8:          end if
9:      end for
10:
11:     return Success
12: end function
```

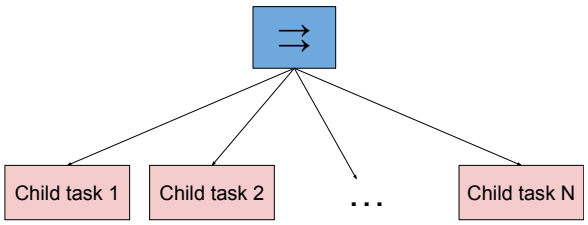



**Figure 2.8:** The structure of a parallel task.

The last commonly used composite task is the parallel task. The parallel task again shares the same structure as the selector and sequence tasks, as seen in Figure 2.8. The parallel task is commonly marked by the double arrow symbol [6]. Unlike the selector or sequence task, the parallel task executes each child task simultaneously. As we can see in the selector algorithm 3, while the child tasks are running, the parallel task returns a running status code just like the selector or sequence task. The difference starts when any child task returns a status code different from the running status code. The parallel task either stores this code in an array or counts the number of successful child status codes(depending on the implementation of the task). Then if M number of child tasks (M can be set to any number lesser than the number of child tasks as seen in the algorithm 3) and table 2.2 returns a successful status code, the parallel task also returns a successful status code. Otherwise, the task returns a failure status code instead. If the number M is set to the number of child tasks, the parallel task behaves similarly to a sequence node and is usually called a parallel sequence. If the number M is set to one, the task behaves similarly to a selector and is called a parallel selector.

**Algorithm 3** Algorithm of the Parallel task

**Require:** $M \leq numberOfChildTasks$

```
1:  function RunParallelTask
2:      SuccessCount ← 0
3:      for all childTasks do
4:          childReturnCode ← childTask.Run()
5:          if childReturnCode == Running then
6:              return Running
7:          else if childReturnCode == Success then
8:              SuccessCount ← SuccessCount + 1
9:              if SuccessCount ≥ M then
10:                 return Success
11:             end if
12:         end if
13:     end for
14:     return Failure
15: end function
```

Since the selector task is used to "select" one new successful task rather than execute a group of tasks, the parallel selector often stops or interrupts the still-running child tasks when the first successful task is found.

As the name suggests, parallel tasks are usually used when we want to execute some behaviors simultaneously in parallel. For example, with the parallel task, the agent can decide where to go and what to say to the player simultaneously.

### Decorators

Decorators [1] are tasks with a single child. They are used for modifying their child's behavior in some way. They do this by changing the returned status code of their child. For example, if we want to make a certain node return only the "failure" code, we can achieve this with decorators. Decorators have many different useful implementations.

## 2.3.2 Common Types of Decorators

Decorators can be implemented in many ways, making designing an AI agent's behavior easier. But some basic types of decorators are used very often. The following list contains examples of commonly used types decorators [1] [5]:

**Algorithm 4** Algorithm of the Inverter

```
1:  function RunInvertor
2:      childReturnCode ← childTask.Run()
3:
4:      if childReturnCode == Running then
5:          return Running
6:      else if childReturnCode == Success then
7:          return Failure
8:      end if
9:
10:     return Success
11:
12: end function
```

- **Inverter**

  Inverter decorator is one of the most simple decorators. As the name suggests, the Inverter simply inverts the result of the child task. As we can see in the algorithm 4 above, if the child task returns a "success" code, the Inverter returns a "failure" code, and vice versa. If the child task returns a "running" code, the Inverter also returns a "running" code. This decorator is useful for simulating a behavior where we expect a certain task to fail.

**Algorithm 5** Algorithm of the Limiter

```
1:  function RunLimiter
2:      if executedTimes ≥ limit then
3:          return Failure
4:      end if
5:
6:      childReturnCode ← childTask.Run();
7:
8:      if childReturnCode! = Running then
9:          executedTime ← executedTime + 1
10:     end if
11:
12:     return childReturnCode
13:
14: end function
```

- **Limiter**

  A Limiter decorator is another widely used decorator. With this decorator, we can limit how often the AI agent can run a child task. The Limiter keeps track of how many times the child task was executed (i.e., how many times the task returned either a "success" or a "failure" code). As we can see in the algorithm 5 above, if the child task gets over this limit, the decorator returns only a "failure" code. The Limiter is often used to ensure the agent doesn't get stuck on the same behavior for an unnecessarily long time and instead tries a differently defined behavior.

**Algorithm 6** Algorithm of the Time Limiter

```
1:  function RunTimeLimiter
2:      if executedStartTime − currentTime ≥ timeLimit then
3:          return Failure
4:      end if
5:
6:      childReturnCode ← childTask.Run();
7:      return childReturnCode
8:
9:  end function
```

- **Time Limiter**

  Time Limiter decorators work similarly to the Limiter decorator. The Time Limiter keeps track of how much time has passed since the child task started running. As we can see in the algorithm 6 above, if the child tasks run time exceeds a given limit, the Time Limiter stops the child task by returning a Failure code. This can be used to limit how long can a child's task run. We can ensure the agent won't get stuck in a loop

trying to finish a never-ending task, or it can also be used to change the agent's behavior more frequently.

**Algorithm 7** Algorithm of the Repeater

```
1: function RunRepeater
2:
3:     childReturnCode ← childTask.Run();
4:     if numberOfRepeats ≥ limit then
5:         return childReturnCode
6:     end if
7:     numberOfRepeats ← numberOfRepeats + 1
8:     return Running
9:
10: end function
```

- **Repeater**

  Repeater decorator is used in the opposite way to the limiter. With this decorator, we make the child state repeat itself a set number of times. The algorithm 7 above shows how the Repeater node execution function works. The Repeater keeps track of how many times the node has been repeated. Instead of returning the child's status code, this decorator returns a "running" code to ensure the child task is executed again. After the set number of repeats, the Repeater returns the status code of the child task.

  This decorator is also often used at the root of the behavior tree with an unlimited number of repeats which creates an endless loop.

**Algorithm 8** Algorithm of the Repeat Until Fail

```
1: function RunRepeatUntilFail
2:
3:     childReturnCode ← childTask.Run();
4:     if childReturnCode == Failure then
5:         return childReturnCode
6:     end if
7:     return childReturnCode
8:
9: end function
```

- **Repeat Until Fail**

  The Repeat Until Fail decorator is very similar to the Repeater. Repeat Until Fail decorator again makes the child state repeat itself but with a different condition. As we can see in the algorithm 8, every time a child task returns a status code, the decorator reruns the child task unless the return code is "failure". Otherwise, the decorator returns the child tasks return code. It is also common to use the inverted version of

this decorator called Repeat Until Success. As the name suggests, the decorator functions the same way, except that the decorator stops when a "success" code is returned by the child task.

**Algorithm 9** Algorithm of the Condition Decorator

```
1:  function RunConditionDecorator
2:      conditionMet ← TestCondition()
3:
4:      if conditionMet == false then
5:          return Failure
6:      end if
7:
8:      childReturnCode ← childTask.Run();
9:      return childReturnCode
10:
11: end function
```

- **Condition decorator**

  Condition decorator is a different way to implement the condition task. The decorator checks if a child task should or can be run. As the name implies, this decorator has its own defined condition or test. As we can see in the algorithm 9 above, when the decorator task is called, a test is evaluated. If the condition is unmet, the decorator returns a "failure" code, and the child task is not executed. But if the condition is met, the task runs as normal. Just like Condition tasks, the condition for the decorator can be anything from checking the value of a variable to more complex checks defined with a separate function.

  The main difference between a decorator condition and a standard condition (as a leaf node) is that the decorator evaluates the condition while the child task is running. This means that the decorator's condition can interrupt or stop the child task during the execution (i.e., while the child task returns a "running" status code). While with the standard condition, we can only test the condition before the following task starts executing or after.

### 2.3.3 Structure of Behavior Trees

Behavior trees work in a similar way to hierarchical state machines [1] [3]. Instead of the state, the main building blocks are tasks. Behavior trees are mainly created with composite tasks as inner nodes, which create the tree's structure, and action tasks as leaf nodes which cause the agent to interact with the game world as explained in the previous section 2.3.1. This way, the tree can be separated into sub-tasks or sub-trees to represent more complex behaviors. Because each task is self-contained and shares the same interface, they can be built into hierarchies without any information on how each sub-task is implemented.

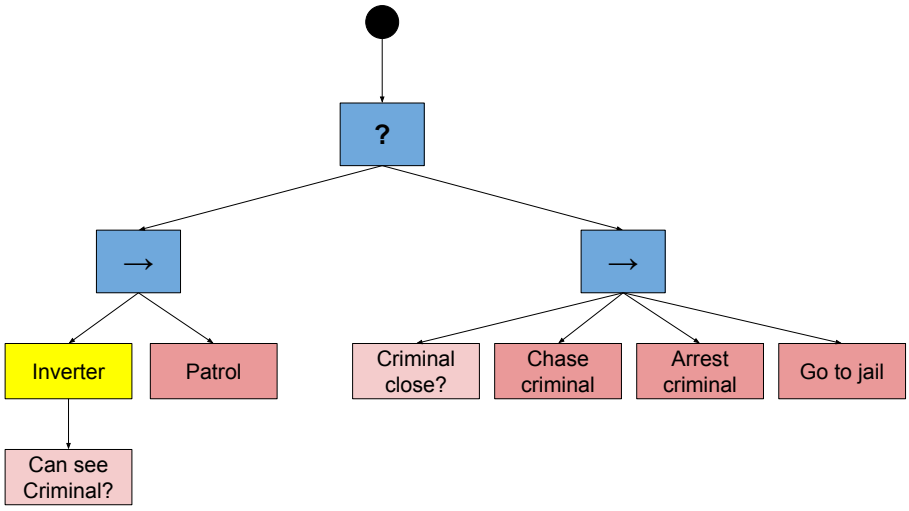



**Figure 2.9:** The example of a simple behavior tree.

### 2.3.4 Blackboards

To be able to create more complex behaviors, behavior trees require to have some form of inner communication between the individual tasks. In the example shown in Figure 2.9, we want the agent to interact with a criminal. If there is more than one criminal in the scene, the agent cannot know which criminal he is supposed to interact with. This problem is caused by the lack of data [1]. Because of this, the behavior trees require a communication system. A blackboard is the most common form of an inner communication system for behavior trees.

Blackboard is an external data store [1]. Individual tasks can write or save data on the blackboard. Other tasks can then later read or withdraw data

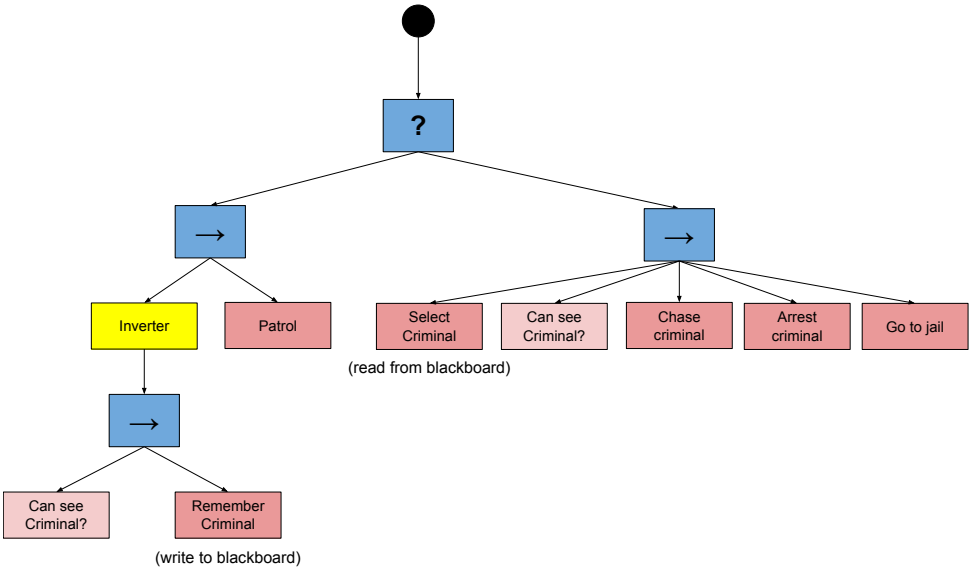



**Figure 2.10:** Behavior tree with Blackboard communication.

from Blackboard. This way, individual nodes can communicate with each other.

Figure 2.10 shows the same behavior tree from Figure 2.9 extended with blackboard communication. Now when the agent sees a criminal during the "patrol" sequence (the sequence on the left side), the action task "Remember criminal" saves the information about the criminal into the blackboard. Then when the agent executes the "chase" sequence (the sequence on the right side), the action task "Select criminal" reads the saved information about the criminal from the blackboard and selects the criminal as a target. After this, the agent continues with the sequence.

Blackboard doesn't have to be used just for communication between individual nodes but also for receiving data from the game world. For example, some actions can be dependent on the time within the game world. This time can be stored in the blackboard and used when needed. Or we can store useful information about the agent's surroundings, such as nearby cover, nearest enemy, etc.

# Chapter 3

# Behavior Trees in Unity

Unity is a cross-platform game engine developed by Unity Technologies based in San Francisco. The engine can be used to create 3D and 2D games, as well as other interactive experiences. It supports a variety of desktop, mobile, console, and virtual reality platforms.

The Engine itself doesn't contain an AI decision-making system. Because of this, I had to create my own system along with an editor for managing and visualizing AI decision-making.

## 3.1 Scriptable Objects

The scriptable object [10] is a type of data container included in the unity engine. The object can store large amounts of data independent of class instances. This is useful for storing unchanging data attached to custom scripts in the engine. The scriptable object can also be instantiated to create a unique copy of the original data.

## 3.2 Behavior Tree Asset

The behavior tree asset is implemented as scriptable objects. Figure 3.1 shows the class diagram of the behavior tree asset. As we can see, the behavior tree itself contains a list of its nodes along with a reference to the root node of the tree. This list stores each node as a scriptable object in the behavior tree asset, which is then stored in the unity asset folder. It also stores the current state of the behavior tree and its blackboard. Each part of the Asset or class associated with the behavior tree is explained in the following sections.

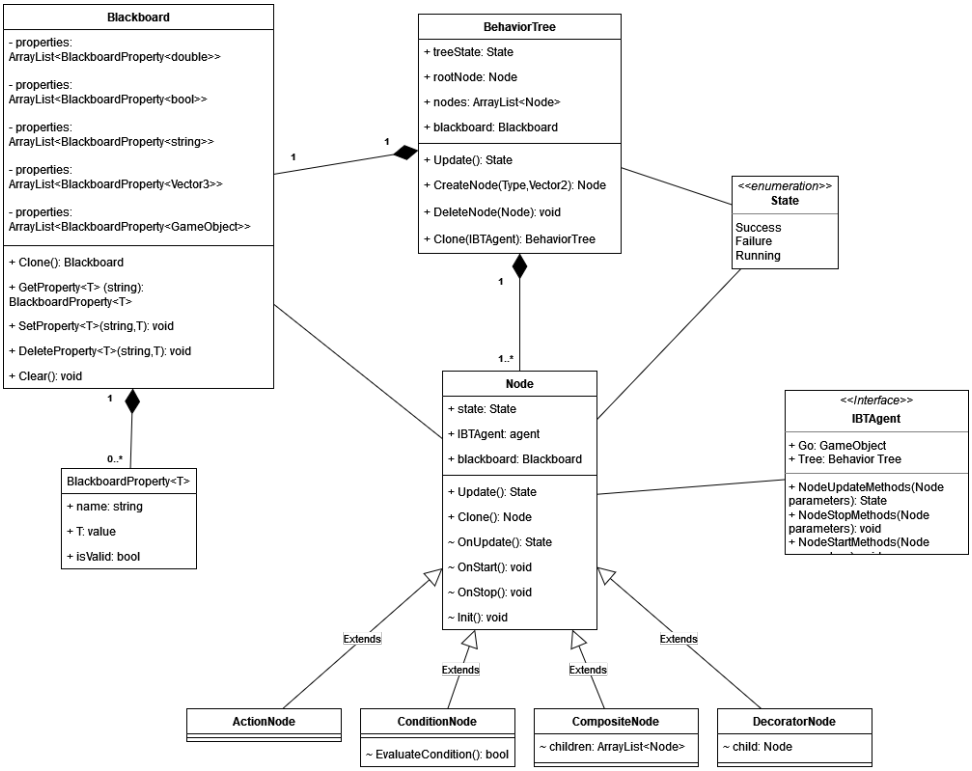



**Figure 3.1:** Class diagram of the behavior tree asset.

### 3.2.1 Blackboard

Blackboards are implemented as a data store for the individual behavior trees and used for inner communications between the individual nodes and communication with the game world, as discussed in the previous chapter 2.3.4.

In my project, the blackboard holds five arrays of properties, one for each of the individual data types: numeric(double), boolean, string, vector, and game object. Every record within the array stores. Every property stores the name of the property, which is used for locating the given property, the value of the given property, and a validation boolean property. The validation property indicates if the given property has a valid value and if it should be used.

The blackboard has four functionalities. Each node (or an object within the game world with access to the blackboard) can retrieve a property by a given name. If the property doesn't exist within the blackboard, the blackboard doesn't return a default value (set for individual types) and a boolean false value, signaling that the property search has failed. Nodes can also create new properties or update the value of existing ones. When a node tries to update a value of a non-existing property, the blackboard creates a new property with the given name and value. Lastly, nodes can invalidate or delete individual properties within the blackboard. When a property is supposed to be invalidated or deleted, its validation property is false. This way, individual

nodes can signal to other nodes that the value of a certain property needs to be updated.

## 3.3 Task Nodes

As mentioned before, behavior tree nodes or task nodes are also implemented as scriptable objects. Because of this, they can be easily stored within the behavior tree asset.

Each node type inherits its behavior from an abstract base class called Node. As we can see in figure 3.1, this class contains the node's current state (as explained in the Behavior tree section 2.3), a reference to the agent attached to the behavior tree, a reference to a blackboard, and an update (also called tick [1]) method used for the execution of the task node. The class also has four abstract methods:

- OnStart

  This method is executed at the beginning of the node execution.

- OnUpdate

  This method is executed on each update call during the execution and returns the node's current state.

- OnStop

  This method is executed at the end of the node execution when the node returns either a Success or Failure state during the update call.

- Initialize

  This method is executed within the node's constructor and is meant to be used for initializing non-changing variables within the node and for initializing data for visualization (explained in the following chapter).

### 3.3.1 Composite Nodes

Composite task nodes connect other nodes in the behavior tree to create more complex behaviors, as explained in the previous chapter 2.3.1. Each composite task node inherits from an abstract class called CompositeNode. This class contains a list of child nodes.

Figure 3.2 shows the implementation of a sequence task node. The node implements the three abstract methods of the base Node class. At the start of the node execution, the index(current) of a current child is set to zero to reset the node. The method OnUpdate shows the implementation of the behavior of the sequence node explained in the composite task section 2.3.1. The method OnStop is not needed, so it stays empty.

Other composite task nodes have been implemented similarly and according to the definition explained in the previous chapter 2.3.1. My project contains several different implementations of the commonly used composite nodes:

```
public class SequenceNode : CompositeNode
{
    public int current;  "1"
    Frequently called  0+1 usages
    protected override void OnStart()
    {
        current = 0;
    }
    Frequently called  0+3 usages
    public override void OnStop()
    {
    }
    Frequently called  0+1 usages
    protected override State OnUpdate()
    {
        Node child = children[current];
        State childState = child.Update();
        switch (childState)
        {
            case State.Running:
                return State.Running;
            case State.Failure:
                return State.Failure;
            case State.Success:
                current++;
                break;
        }

        if (current >= children.Count)
            return State.Success;

        return State.Running;
    }
```

**Figure 3.2:** Implemetation of the sequence task node.

- Sequence
- Parallel Sequence
- Selector

### 3.3.2 Action Nodes

Action nodes are implemented as leaf nodes of the behavior tree as explained in the previous chapter 2.3.1.

Individual actions can have a single implementation or implementation through the agent interface. Single implementation is good for generic actions with the same implementation for every agent, like moving the agent or searching for the closest target. Implementation through the agent's interface is good for agent-specific tasks. Several agents can then share the same behavior but have a different implementation for certain actions (for example, attacks, target selection, etc.).

```
public class AttackNode : InputActionNode
{
    [Header("Input")] [SerializeField] private string targetGameObjectProperty = "AttackTarget";  Unchanged

    private GameObject _target;

    Frequently called  0+1 usages
    protected override State OnUpdate()
    {
        if (_target != null)
            return State.Failure;

        return Agent.AttackUpdate(_target);
    }

    Frequently called  0+1 usages
    protected override void OnStart()
    {
        if (!blackboard.TryGetProperty(targetGameObjectProperty, out _target))
            return;

        if (_target == null)
            return;

        Agent.AttackStart(_target, targetGameObjectProperty);
    }

    Frequently called  0+3 usages
    public override void OnStop()
    {
        if (_target == null)
            return;

        Agent.AttackStop(_target, targetGameObjectProperty);
    }
```

**Figure 3.3:** Example of implementation of the attack action task node.

Figure 3.3 shows an example of an attack action task node implementation through the agent interface. The agent interface has to define three methods, one for each node execution method of the attack action. Since the attack action depends on a blackboard property, the property's value has to be passed as an argument of the interface methods. Alternatively, the agent could only pass the name of the required property, and the agent could fetch the value himself.

Figure 3.4 shows a single implementation of a move to action. This way, every agent shares the same implementation of the action. The agent's interface contains a reference to the agent's game object. With this reference, the action can access other components connected to the agent. In this example, the move to node uses a navmesh agent component, which needs to be present on the agent's game object.

### Conditions

The condition node can then test the given property value as explained in the previous chapter 2.3.1.

In the project, every condition task node inherits from an abstract class called ConditionNode. This class defines an abstract method for evaluating the condition and returns a boolean value. If the defined evaluation check passes (i.e., returns a true value), the node returns a success state. Otherwise,

```
public class MoveToNode : InputActionNode
{
    [Header("Input")] [SerializeField] private string vectorPropertyName = "target position";  "TargetPosition"
    [SerializeField] private float closeToValue = 1.5f;  Unchanged
    private Vector3 _targetPosition;
    private NavMeshAgent _navMeshAgent;
    Frequently called  0+1 usages
    protected override void OnStart()
    {
        _navMeshAgent.isStopped = false;
    }
    Frequently called  0+3 usages
    public override void OnStop()
    {
        _navMeshAgent.isStopped = true;
        if (state == State.Success)
            blackboard.DeleteVector3Property(vectorPropertyName);
    }
    Frequently called  0+1 usages
    protected override State OnUpdate()
    {
        if (!blackboard.TryGetProperty(vectorPropertyName, out _targetPosition))
            return State.Failure;

        _navMeshAgent.SetDestination(_targetPosition);
        if (_navMeshAgent.pathStatus == NavMeshPathStatus.PathInvalid ||
            _navMeshAgent.pathStatus == NavMeshPathStatus.PathPartial)
            return State.Failure;

        Vector3 position = Agent.Go.transform.position;
        if (Vector3.Distance(a: position, b: _targetPosition) < closeToValue)
        {
            return State.Success;
        }
        return State.Running;
    }
```

**Figure 3.4:** Example of implementation of the move action task node.

the condition node fails.

Figure 3.5 shows an implementation of a Close to condition node. This node uses two blackboard properties, a game object property target and a numeric property boundary (in code called "closeToDistance"). The node calculates the distance to the target game object (obtained from the blackboard) and checks if the distance is smaller than the given boundary(also obtained from the blackboard).

```
9 asset usages
public class ConditionCloseTo : ConditionNode
{
    [Header("Input")] [SerializeField] private string gameObjectPropertyName = "Target";  Changed in 5 assets
    [SerializeField] private string numericCloseToPropertyName = "distance";  Unchanged
    [SerializeField] private float defaultCloseToDistance = 10.0f;  Unchanged
    [SerializeField] private double _closeToDistance = 0;  Unchanged

    Frequently called  0+1 usages
    protected override bool EvaluateCondition()
    {
        if (!blackboard.TryGetProperty(gameObjectPropertyName, out GameObject target))
            return false;

        if (target == null)
            return false;

        if (!blackboard.TryGetProperty(numericCloseToPropertyName, out _closeToDistance))
            _closeToDistance = defaultCloseToDistance;

        Vector3 agentPosition = Agent.Go.transform.position;
        float distance = Vector3.Distance( a: target.transform.position,  b: agentPosition);

        return distance <= _closeToDistance;
    }
```

**Figure 3.5:** Example of implementation of the condition close to task node.

### 3.3.3 Decorator Nodes

As explained in the previous chapter 2.3.1, The decorator is used to modify the return state of the child node. Each decorator inherits from a DecoratorNode class and has exactly one child node.

My project contains several different implementations of decorator nodes:

- Repeat

  *Repeat* node has been implemented according to the definition of a *Repeater* decorator which can be found in previous chapter 2.3.2

- Repeat Until Fail

  *Repeat Until Fail* node has been implemented according to the definition of a *Repeat Until Fail* decorator which can be found in the previous chapter 2.3.2

- Limiter

  *Limiter* node has been implemented according to the definition of a *Limiter* decorator which can be found in the previous chapter 2.3.2

- Inverter

  *Inverter* node has been implemented according to the definition of a *Inverter* decorator, which can be found in the previous chapter 2.3.2.

- Pre Condition

  *Pre Condition* node works as a condition decorator 2.3.2. This node holds a name of a blackboard property. The node then performs a check of the node's condition with the value of the blackboard property. If

the condition is met, then the child node is executed. If not, the node always returns a Failure state.

### 3.3.4 Sub-tree Node

The sub-tree node holds a separate behavior tree, which can be created using any combination of nodes that make sense for the specific task. With this node, the user can create a nested behavior tree that can be reused in various parts of the main behavior tree, thus increasing the modularity and readability of the overall tree. This allows users to create complex behavior trees that can be easily reused and modified for different scenarios.

```
public class SubTreeNode : ActionNode
{
    [SerializeField] public BehaviourTree subTreeSource;  Changed in 8 assets

    private bool _subTreeIsValid = false;

    Frequently called  0+1 usages
    protected override void OnStart()
    {
        if (!_subTreeIsValid && subTreeSource != null)
        {
            subTreeSource = subTreeSource.Clone(Agent, blackboard);
            subTreeSource.blackboard = blackboard;
            _subTreeIsValid = true;
        }
    }

    Frequently called  0+1 usages
    protected override State OnUpdate()
    {
        if (!_subTreeIsValid)
        {
            Debug.Log( message:$"Subtree: {subTreeSource.name} is not valid.");
            return State.Failure;
        }

        return subTreeSource.rootNode.Update();
    }
```

**Figure 3.6:** Example of implementation of the sub-tree task node.

Figure 3.6 shows the implementation of the sub-tree node. When a sub-tree node is executed for the first time (In Figure 3.6 when the OnStart method is called for the first time), the node creates a personal working copy of the supplied behavior tree and stores it within itself. This cloned tree is then used in every following execution of the node.

On later calls for execution (i.e., when an OnUpdate method in figure 3.6 is called), the node runs its own behavior tree and returns the state of the nested root node. This means that the state of the nested behavior tree is propagated up to the main behavior tree, allowing the agent to make decisions based on the current state of the game. This also allows for greater flexibility in creating behavior trees, as designers can create sub-trees for specific scenarios and conditions and reuse them as necessary throughout the main tree.

### 3.3.5 Root Node

The root node is a node in which every behavior tree starts. The node doesn't have any special functionality. Its purpose is purely to distinguish the root node from other nodes. It's also created for easier implementation of the behavior tree editor.

## 3.4 Behavior Tree Agent

The behavior tree agents are a mono-behavior script component. To create a behavior tree agent class, a class must implement an interface called IBTAgent. The agent is connected to the behavior tree with an interface. This interface defines the agent's actions. As explained in the actions section 3.3.2, thanks to the interface, each agent action set can be implemented differently. To implement a new action for the given action node, three new methods need to be created:

- Starting method

  Starting methods are called when the action node is first executed. This method should be used for initialization if the action requires one.

- Update method

  Update methods are executed on each update call. This method is used for the main functionality of the action.

- Stopping method

  Stopping methods are called when the node returns either a Success or Failure state. This method should be used to update the agent's data or to end certain processes.

These are then called from the behavior tree during the simulation.

The agent also contains a behavior tree asset. This asset must be instantiated or cloned at the start of the game. This way, each agent that shares the same behavior tree definition can have their own copy of the behavior tree asset for simulating the defined behavior. Each agent also creates a personal copy of the blackboard.

The agent can also use other components for executing the actions.

# Chapter 4

# Behavior Tree Editor

Behavior trees are frequently used with custom editors [1]. Editors help with the debugging process of the AI decision-making system. Since the structure of the behavior trees is usually very intuitive, even non-programmers can assist in creating complex behaviors for the game agents with a custom editor.

Since Unity doesn't support any behavior tree system or an editor, I had to create my own that uses the data from the behavior tree assets. The editor was created with Unity's UI toolkit and other tools for creating custom editors in the engine.

## 4.1 UI Toolkit

UI toolkit [8] is a collection of features, resources, and tools for developing a user interface (UI). The toolkit can create an in-game UI system, custom extensions for the Unity Editor, or run-time debugging tools.

This toolkit provides a UI system based on commonly used web technologies. It supports the creation of complex interactive views. Each UI can be created with three main types of files:

- UXML

  *Unity Extensible Markup Language* (UXML) files are text files defining the overall structure of the elements within the UI. They offer a more intuitive way of creating the layout for UI than the other methods supported by the Unity engine.

- USS

  *Unity Style Sheet* (USS) files used to apply visual styles and behaviors of the elements in the UI structure defined in the UXML files. The USS files are inspired by the Cascading Style Sheets (CSS) from HTML. Because of this, USS shares a lot of similarities with CSS.

- CS

  Each element of the UI can be connected to a CS script. This script contains the logic of the UI structure or individual elements. The CS

script can also create or modify the UI structure and apply visual styles and behaviors.

UI toolkit was created for easier creation of UI systems [9]. With this toolkit, even less technical users can easily define a layout and visual style in the UXML and USS files while the developers can focus on the more technical tasks, such as creating the logic for the UI system.

## 4.2 Custom Editor

I created a custom Behavior tree editor with a UI toolkit and other custom editor tools supported by the Unity engine. The editor is used to create or edit new behavior trees and visualize the agent's decision-making process during the game runtime. The first main issue was to create a way of displaying an oriented graph on the screen. This was solved by using the graph view feature from the UI toolkit. The graph view is a visual element that offers a way to create a visualization for graphs. An example of the use of graph view is shown in figure 4.1.

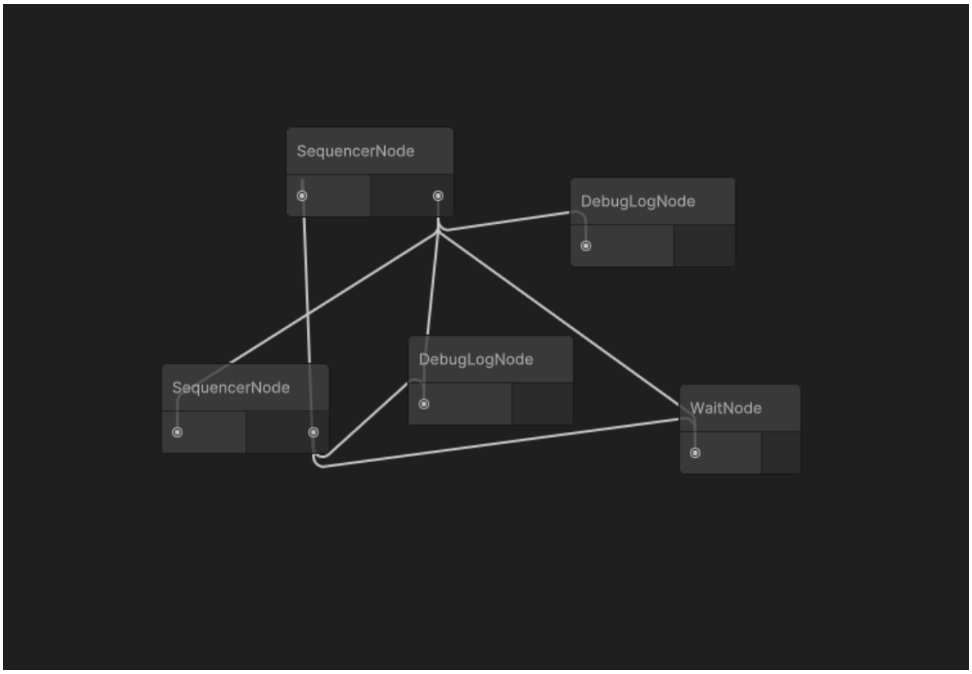



**Figure 4.1:** Example of a UI toolkit graph view

The graph view also holds a definition for the individual graph nodes stored in the graph. Each node can be created with input and/or output ports. These ports can then be connected to other nodes to create edges.

I created a custom behavior view by inheriting from the graph view. This behavior tree view can process the data from any behavior tree asset and display it as a graph, as shown in figure 4.2.

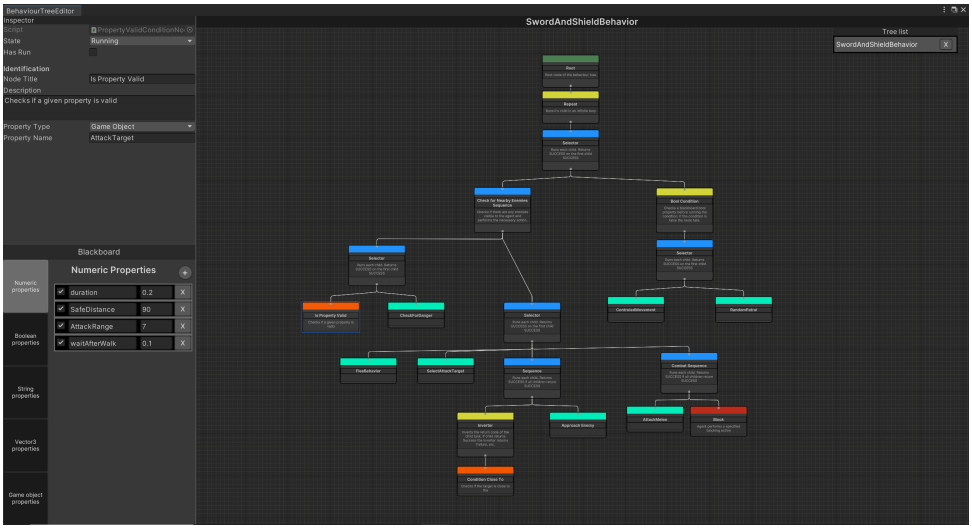



**Figure 4.2:** Behavior tree editor.

To display the node data from the behavior tree, I also created a custom behavior tree node view by inheriting from the graph view node view. The behavior tree node views hold data from the node of the behavior tree asset. This data is then displayed with the node view. Each node view can have output ports and/or input ports depending on the type of node. For example, an action node can have only input ports because it's always a leaf node. They also show the node name and a description of the node functionality. Each description and name can be changed in the editor for better readability. Each main type of node is also color-coded as seen in figures 4.2:

- Green - Root nodes.
- Yellow - Decorator nodes.
- Blue - Composite nodes.
- Red - Action nodes.
- Orange - Condition nodes.
- Teal - Sub-tree nodes.

The behavior tree view is the main part of the custom editor. After that, I added a way for the user to create individual nodes. With the help of the Unity custom editor toolset, I added a simple menu that appears when the user right-clicks anywhere within the behavior tree view. This menu is organized by the available types of nodes. After the user selects a node, the selected node type is created on the cursor's position, and a new node is stored with the behavior tree asset.

Along with the view, I created a custom inspector view for more detailed editing of individual nodes. The inspector is created with an Immediate Mode Graphical User Interface (IMGUI) [9] system from the unity editor toolset.

This system offers an easy way of displaying and editing variables of custom classes.

Figure 4.2 shows the whole editor in edit mode with a behavior tree view, inspector, and a blackboard view.

The Editor also contains a separate behavior tree view called sub-tree view, visible in figure 4.3. This view is used for visualizing sub-trees stored within the sub-tree nodes. When a user selects a certain node within the behavior tree view, the view checks if the selected node is a sub-tree node. If yes, the sub-tree view becomes visible and shows the structure of a tree found within the sub-tree node.

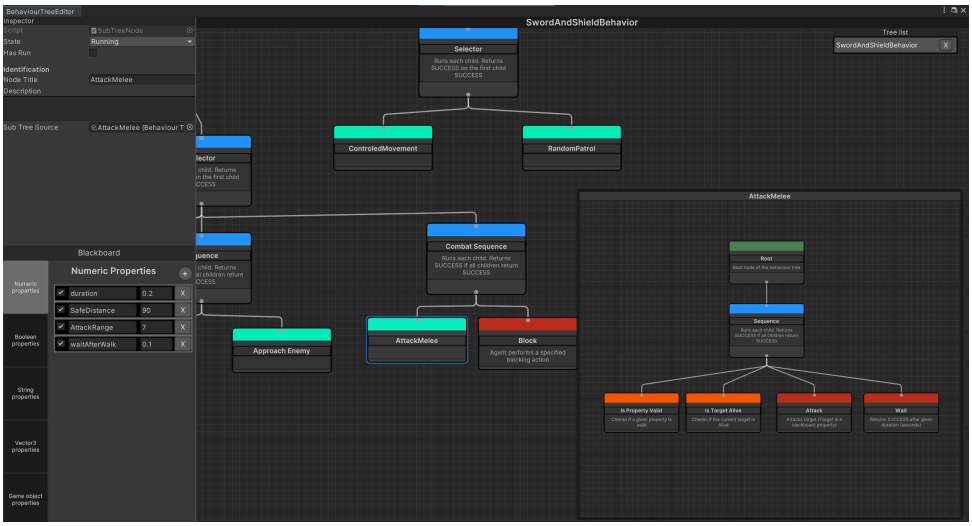



**Figure 4.3:** Behavior tree editor with a sub-tree view visible.

The editor is usable even during runtime. Users can also select a game object with an attached behavior tree agent to display the agent's behavior tree. The execution of the behavior tree is visualized with color borders. Each node starts without a color border. When the agents interact with nodes, the nodes change their color depending on their state, as seen in figure 4.4:

- Orange border

  The node is still running and hasn't finished executing.

- Green border

  The node has executed the action without a problem and returned to the Success state.

- Red border

  The node has failed to execute its action and returned to the Failure state.

For readability, the colored borders fade to a transparent color over time. With this approach, it is much clearer which node is currently running, which

node has recently finished executing, and which node has not been executed in a while.

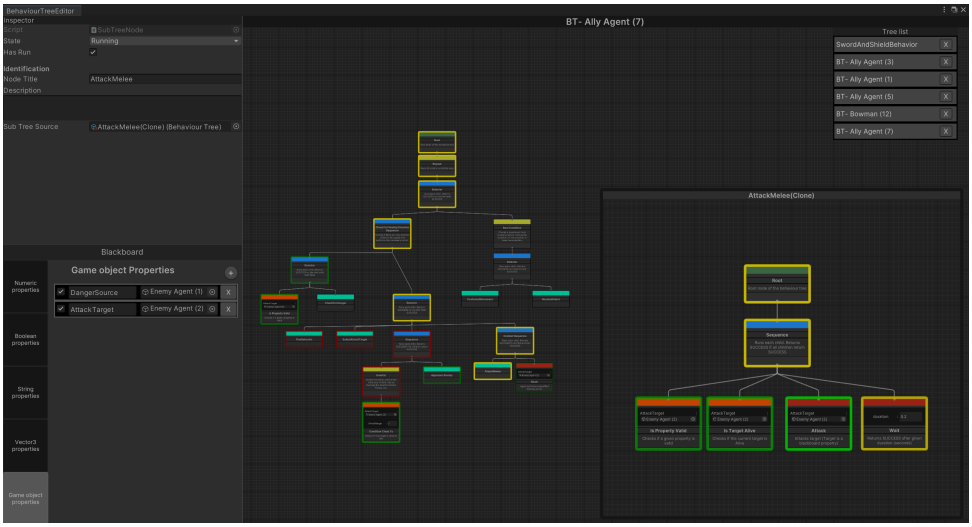



**Figure 4.4:** Behavior tree editor during runtime with a selected in-game agent.

The sub-tree view also visualizes the current state of the sub-tree, which is visible in figure 4.4.

Also, during runtime every node which is using a blackboard property or properties as input and/or as output, the node view shows the currently used blackboard property (visible in figure 4.4). Each action node dependent on a blackboard property inherits from either an abstract class InputAction, OutputAction, or InputOutputAction. These abstract classes define a list of used blackboard properties. InputAction is used for nodes that require data from a blackboard property as input. OutputAction is used for nodes, which upon completion, update the value of a certain blackboard property. InputOutpuAction combines the behavior of an InputAction and an OutputAction. The node has to update these lists to visualize the property in the node view. This way, the user can quickly and easily see which action node uses which blackboard property and the properties value.

## 4.3 Blackboard View

As explained in the previous chapter 2.3.4, blackboards are a significant part of the behavior tree. Because of this, I created a custom behavior tree view for editing and visualizing the individual blackboard properties.

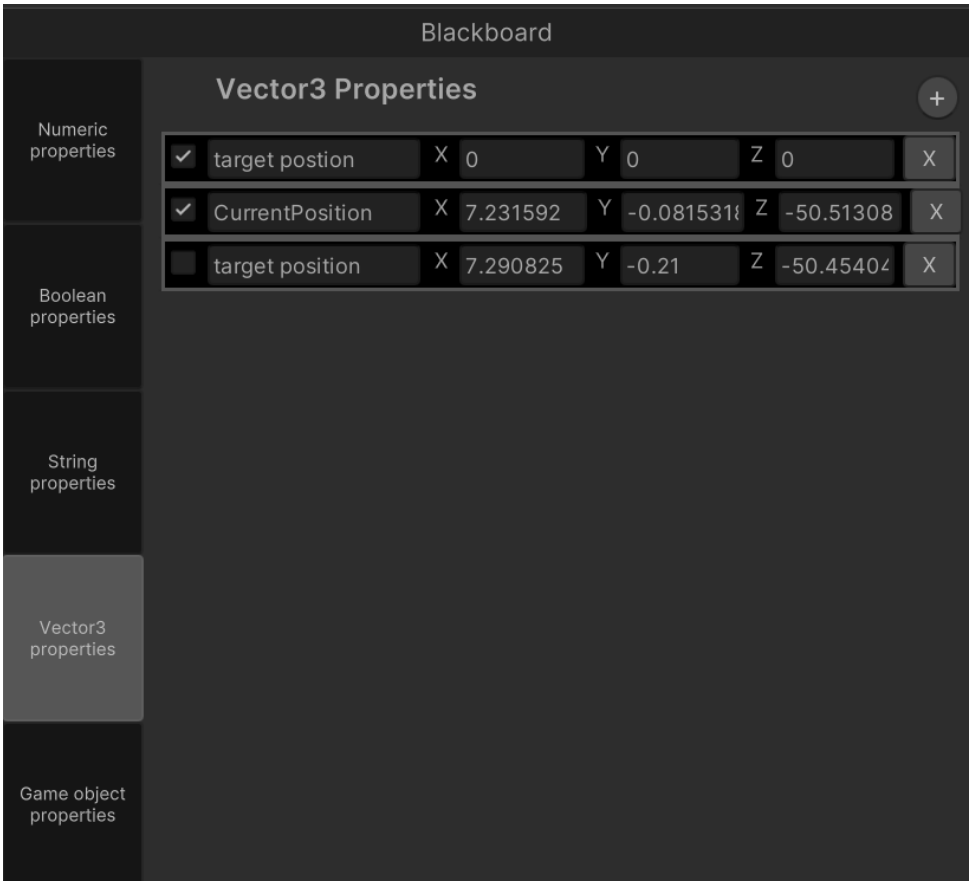



**Figure 4.5:** Blackboard view.

The blackboard view, visible in figure 4.5, is an essential part of the editor window, and it plays an important role in the visualization and editing of blackboard properties. The view is designed with the UI toolkit, and it is implemented as a separate view to ensure maximum clarity.

The left panel of the blackboard view contains buttons that enable users to switch between different lists of blackboard properties quickly. The right panel displays the properties of the currently selected list, enabling users to visualize and edit the properties with ease.

Each blackboard property within the view is divided into four parts. Firstly, a checkmark is present, indicating if the property is valid or invalid. Secondly, the property name is displayed, making it easy for users to identify each property. Thirdly, the property value is shown. After that, the value of a property is visible (figure 4.5 shows the values of a vector property list). Lastly, a delete button is present, allowing users to delete a property completely from the blackboard instead of merely setting it invalid.

All parts of a property can be edited within the blackboard view window, enabling users to modify the values of real blackboard properties quickly. The view also includes a creation button, visible as a circle button with a plus sign in the figure 4.5. This button enables users to create new properties within the currently selected list of properties with ease.

# Chapter 5

# Results

To show how my decision-making system works, I created a demo game project in the Unity engine. Figure 5.1 shows an in-game screenshot from the game. The game is a simple game where the player controls a party of warriors a tries to defeat all enemies in his path.

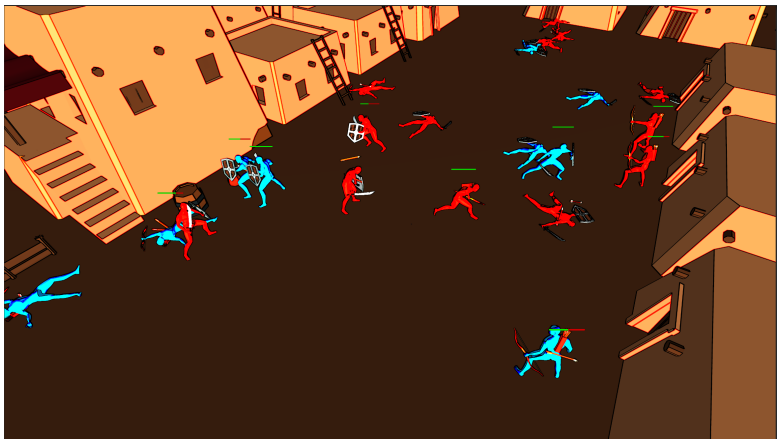


**Figure 5.1:** Screenshot from the demo game.

The project contains several scenes that showcase the functionalities of the present features.

## 5.1 Camera movement

To look around the scene in a game view easily, I created a simple script for camera control. This camera is implemented in a strategy game-style way. This means that the camera is looking at the scene from above at a 45-degree angle. The camera can be moved either by pressing the keys W, A, S, and D, by pressing the arrow keys, or by holding the middle mouse button and dragging the mouse. The camera can also zoom in and out by scrolling with the mouse wheel and rotating around the scene with the keys Q and E.

## 5.2 Characters Agents

Characters within my project are all implemented as NPC agents or, specifically, behavior tree agents. These characters are created as game objects with many useful components. For example, I have used the navMeshAgent component. The navMeshAgent is part of the unity pathfinding system, which uses the NavMesh (present in the scene) to move characters in the game world. This allows characters to move around the game world realistically (avoiding obstacles and etc.) and without any issues. The characters themselves are also marked as obstacles with the Nav Mesh Obstacle component. This stops the character from moving through each other or getting stuck in each other.

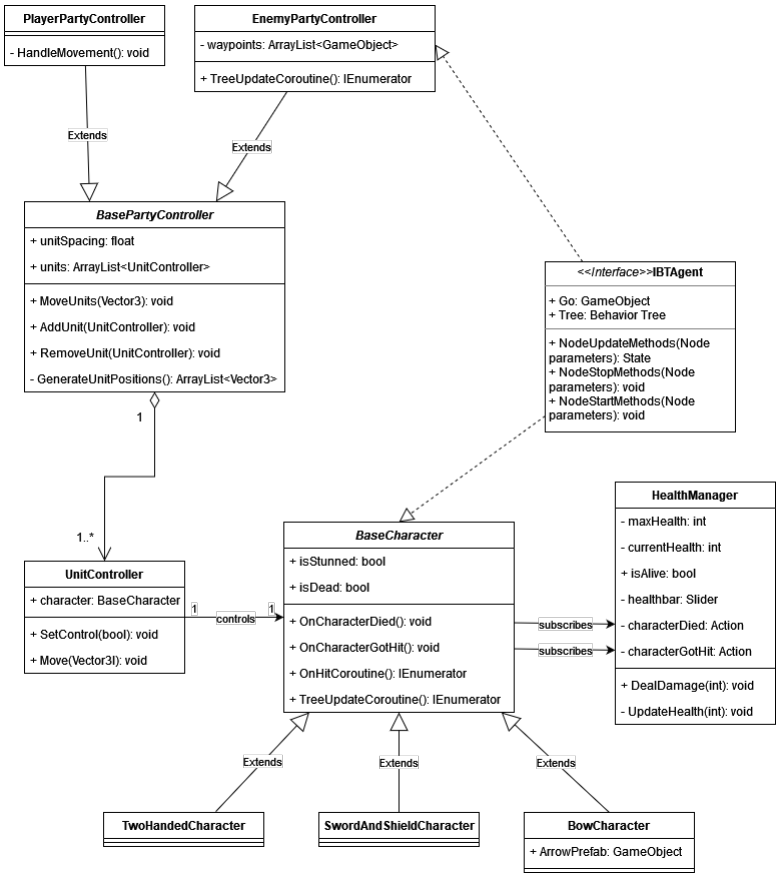



**Figure 5.2:** Class diagram of implemented character components.

Figure 5.2 shows the class diagram of all created components present or used by the character (unity components such as the navMeshAgent are not shown in the diagram).

Each character has a health manager script and a character script component that implements the behavior tree agent's interface, shown in figure 5.2. Characters also have a child object with a mesh renderer component, an animator component and another child object with a UI canvas. This canvas contains a health bar.

Additionally, we can see in figure 5.2, that each character can be controlled by a unit controller. This unit controller is then a part of a party controller. The party controller can then decide where the units should move to. These controllers can be either driven by the player input (**PlayerPartyController** in figure 5.2) or by a behavior tree (**EnemyPartyController** in figure 5.2).

Every character component and possible character actions, along with party controllers, will be explained in the following sections.

### 5.2.1 Health manager

The health manager is a script component responsible for managing all health-related matters, such as health bar display, damage calculation, etc. This enables easy modification of properties of individual characters and separates the health-related logic from other systems connected or used by the specific character.

Each character has a set number of maximum hit points. At the start of the game, the character's current health is set to the maximum hit points. The manager is also responsible for updating the health bar child object each time the character is hit. It also contains a boolean variable to tell if a character is still alive and two events: **characterGotHit** and **characterDied** (the two variables of type Action in HealthManager class in figure 5.2). The first event is invoked when a character is hit, and the second event is invoked when a character has died. The character script is subscribed to these events and handles them with methods **OnCharacterGotHit** and **OnCharacterDied**.

### 5.2.2 Character Actions and Conditions

Each character simulates the behavior defined by their behavior tree. Every behavior tree is composed of generic task nodes shown in the previous chapter 3 and specific actions and conditions created for this game project. The project currently contains several types of actions:

- **Attack**

  This is an agent-specific action that needs to be implemented through the agent's interface. The action uses one game object blackboard property as an input. The main purpose of this action is for the agent to face the target and try to attack him. Since this is an agent-specific action, the attack can be implemented in any way. For example, the sword and shield character attacks the target with a sword, and the bow character shoots an arrow toward the target when this action is executed. This action should always be either in a state of running or success unless the target is not targetable or dead, in which case the action should fail.

- **Block**

  This is also an agent-specific action, which has to be implemented through the agent's interface. The action uses one game object blackboard property as an input. The purpose of this action is for the agent to try

to block incoming attacks from the current target (the blackboard input property) for a small period of time. This action should always be either in a state of Running or Success (unless the agent requires a different implementation)

- **Get Next Waypoint**

  This agent-specific action is used for fetching a new game object waypoint. When the next waypoint is found, the waypoint is stored in the blackboard as a game object property.

- **Get Position From Game Object**

  This action takes one game object blackboard property as an input property. Then it takes the current position within the game world of the game object acquired from the blackboard and stores it in a separate output vector blackboard property.

- **Move to Position**

  This action uses one vector blackboard property as an input (this is the target destination). The action also expects the agent's game object to contain a NavMeshAgent component. This component is used within this action to move the agent to the target destination (acquired from the blackboard). When the agent is still moving toward the target destination, the action returns a running state. If the target destination has been reached, the action returns a successful state. If the target destination is unreachable, the **Move to Position** action fails.

- **Move to Game Object**

  This action works similarly to the **Move to Position** action. Just like the **Move to Position**, this action uses one blackboard property, except instead of a vector, this action uses a game object. The action is also agent-specific, which must be implemented through the agent's interface. The main difference between this action and **Move to Position** action is that this action should move the agent to the target game object's current position. This way, the agent can move to a different moving object. The action changes its state in the same way the **Move to Position** action does.

- **Select Closest By Tag**

  This action finds the nearest game object to the agent using a specific tag, such as "Player" or "Enemy". The tag can be the same for all behavior trees or specified through a string value in a string blackboard property.

  If the action locates a game object with the specified tag, it will store a reference to the object in a game object blackboard property and return a success state. However, if the action cannot find any corresponding objects, it will return a failure state.

- **Select Target**

  This is an agent-specific action. The purpose of this action is to select a new target game object. This game object can be used in other actions, such as **Attack** or **Block** action. When a target has been found, a reference to the target's game object is stored within a blackboard property.

- **Set Movement Speed**

  This agent-specific action is used to change the agent's movement speed. The movement speed is changed based on the selected movement speed enum. The enum has four values: *Crawling*, *Walking*, *Running* and *Sprinting*. When the action is executed, the enum value is passed as a parameter to the agent's implementation. Then the agent can change its own movement speed based on the selected enum value.

- **Flee**

  This action is an agent-specific action that uses a game object blackboard property as an input. This game object is a danger source from which the agent should run away.

- **Generate Random Position**

  This action generates a random vector position and stores it inside a vector blackboard property. This random position is generated around the agent's current position within a given range.

- **Wait**

  This action takes a numeric blackboard property as input. This input is a waiting duration. When a blackboard property is not found, a default value (which can be set in the behavior tree editor) is used. When the node is executed, it stores the current time and then checks on every update if the time passed is bigger than the duration. If the action should still wait, a running state is returned. If the action waited for the given duration, it returns a success state. This action is generally used to delay decision-making. For example, when a character is patroling, we can make the character wait at his patrol destination before he moves to the next one.

- **Set Animator Trigger**

  This action assumes that the agent's game object contains an Animator component. The action sets an animator's trigger inside of the animator. The trigger is accessed by the trigger's name, which can be set as input inside the editor.

- **Custom Action**

  This action contains a unity event. This event takes a game object and calls a method from a component attached. After the call is made, the

node returns a success. This method can be used to directly call any method from the agent or any child game objects of the agent.

- **Debug Log**

  As the name suggests, this action is used only for debugging purposes. The node simply logs a given message. This message can be set inside the behavior tree editor.

The project also contains several conditions:

- **Can See Target?**

  This condition takes a game object blackboard property as an input. This game object is the target. The condition then performs a ray cast from the agent's position toward the target's position. If the ray cast hits the target without being blocked, the condition returns a successful State. Otherwise, it returns a failure state.

- **Is Property Valid?**

  This condition takes property from a blackboard and checks if the property has a valid value (if the property has a validation value set to true). If the given property is valid, it returns a success state. Otherwise, it returns a failure state.

- **Is Target Close?**

  This condition uses two blackboard properties as input. One is a game object property, and the other is a numeric property. The agent then calculates the distance of the given game object. If the distance is smaller or equal to the distance limit (given numeric value), the condition returns a success state. Otherwise, the condition fails.

- **Is Target Alive?**

  This condition uses a game object blackboard property as an input. This game object is the selected target. The condition tries to find a health manager script component connected to the target game object. If the condition fails to find the component, the condition fails. If the component is found, the condition checks if the target is still alive. If the target is still alive, the condition returns a successful state. Otherwise, it returns a failure state.

- **Boolean Condition**

  This condition simply reads the value of a boolean blackboard property. If the property's value is true, the condition returns a success state. Otherwise, if the value is false or the boolean property is not found, the condition returns a failure state.

### 5.2.3 Common Behavior

Common behaviors are behaviors that any character can use. Each of these behaviors is created as a separate behavior tree. These behavior trees are not intended to be used as the main behavior trees used by the NPC characters within the project. Instead, these behaviors should be used inside the sub-tree node 3.3.4. The game project contains several common behaviors:

- **Select Target**

  This simple behavior is composed of a sequence with a **Is Property Valid** condition and a **Select Target** action as seen in the figure 5.3. The Behavior simply checks the currently selected target, which is a game object blackboard property, is valid. If the property is valid, the behavior returns a failure state and ends, signaling that the current target is still valid. If the current target is not valid, the sequence continues to execute the Select Attack Target action, which tries to find a new valid target game object and returns a state based on the result of the action.

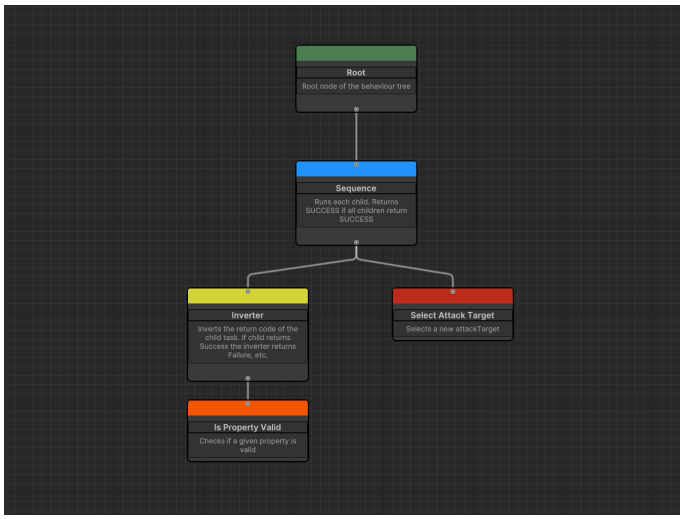



**Figure 5.3:** Select Target behavior tree.

- **Approach Target**

  This behavior is composed of a sequence with two conditions and two actions, as shown in figure 5.4. Additionally, for this behavior to work properly, the blackboard has to contain a specific game object blackboard property, which serves as the target that should be approached. This property is set to all actions and conditions present in the behavior, which require the property to execute properly.

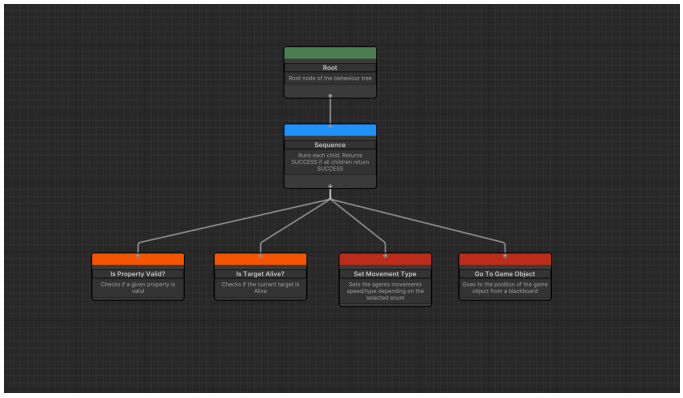



**Figure 5.4:** Approach Target behavior tree.

The behavior first checks if the current target (the game object blackboard property) is valid in the **Is Property Valid** condition. This also tests if the property exists within the blackboard. Next, the behavior tests whether or not the target is alive by executing the **Is Target Alive** condition. If both of these conditions are met, the behavior changes the movement speed of the agent to a running speed and starts executing the **Move to Game Object** action. Otherwise, if the conditions are unmet or the **Move to Game Object** actions fail, the behavior returns a Failure state.

- **Attack Target**

  This behavior is composed of a sequence with two conditions and two actions, which are visible in figure 5.5. Just like the **Approach Target** behavior, this behavior that the blackboard contains a game object blackboard property, which is used as the target. The behavior also uses an optional numeric blackboard property.

  First, the behavior checks to see if the target blackboard property exists and is valid. Then tests if the currently selected target is still alive by running the **Is Target Alive** condition. If both of these conditions are met, the agents start performing an attack. If the attack has been successful, the sequence starts executing the **Wait** action. The **Wait** action can use the optional numeric blackboard property for the duration or use the default value set in the editor. This creates a delay after an attack. The delay can be used for modifying the character's attack speed.

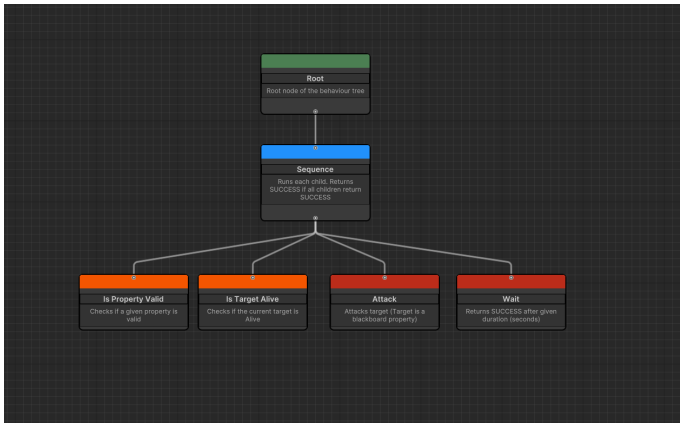



**Figure 5.5:** Attack Target behavior tree.

- **Check for Danger**

  This behavior is composed of a sequence, three conditions, and a single action which is shown in figure 5.6. The behavior can use two optional blackboard properties. One numeric property is the distance limit of a character's safe area. The other one is a string property, which holds the name of a tag.

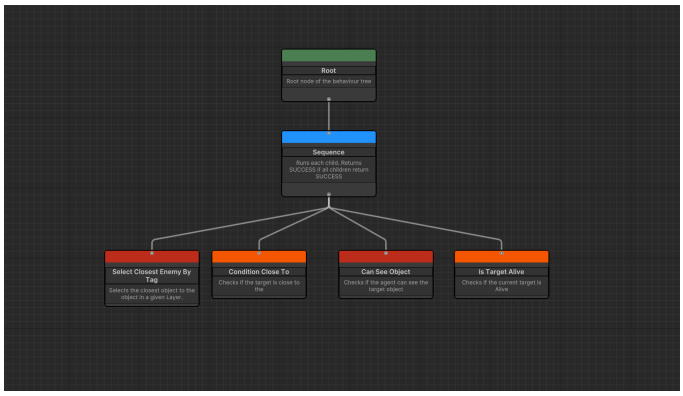



**Figure 5.6:** Check for Danger behavior tree.

First, the behavior finds the nearest game object by a given tag. This tag can be acquired from a blackboard or be set as the default value in the editor. If an object is found, the sequence continues to test the conditions. The behavior checks if the found object is close to the agent by executing the condition **Is Target Close**. Then we check if the target is visible to the agent and, finally, if the target is alive. If all of these conditions are met, the behavior returns a success state. Otherwise, it returns a failure state.

- **Flee from Danger**

  This behavior is composed of a sequence, **Boolean Condition**, two actions, and a **Limiter** decorator. The behavior requires a boolean

blackboard property, indicating if the agent should run away.

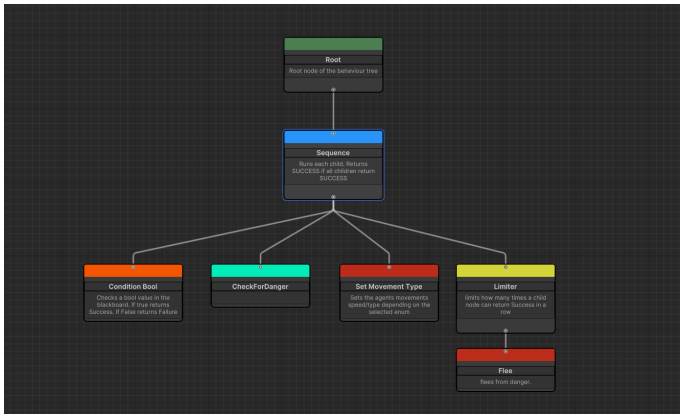



**Figure 5.7:** Flee from Danger behavior tree.

The behavior also uses a sub-tree task. This sub-tree contains the common behavior **Check for Danger**. The sub-tree is used for finding the closest valid game object, which is the danger source.

The behavior first checks the **Boolean Condition** if the agent should run away. This task uses the boolean blackboard property. The property can be set in the behavior tree or by another script with access to the agent's blackboard. In the project, this property signals that the agent is low on health and is set by the health manager component attached to the agent's game object. Then the **Check for Danger** behavior is executed to find the closest source of danger. Then the behavior sets changes the movement speed for running away in the **Change Movement Speed** and executes the **Flee** action. The **Flee** action also has a **Limiter** decorator attached to it. This decorator ensures that the agent, after a set number of tries, returns a failure state. This makes the agent choose a different action from time to time (for example, attacking the danger source) when he is supposed to be running.

- **Patrol**

  The behavior shown in figure 5.8 is composed of a main sequence and two other composite tasks, two conditions, and three actions. and also a sub-tree task. This sub-tree contains a **Check for Danger** common behavior. The behavior also requires a boolean blackboard property. If the property is true, it indicates that a different behavior tree or a script component controls the agent. This is tested in the **Boolean Condition** task. If the agent is not controlled by someone else, the decision-making process continues.

  Next, the behavior executes the sub-tree with **Check for Danger** behavior to test if the agent sees any enemies nearby. The result of the sub-tree is then inverted by the **Inverter** decorator. Now if the agent does not see any enemies in his vicinity, he continues his patrol.

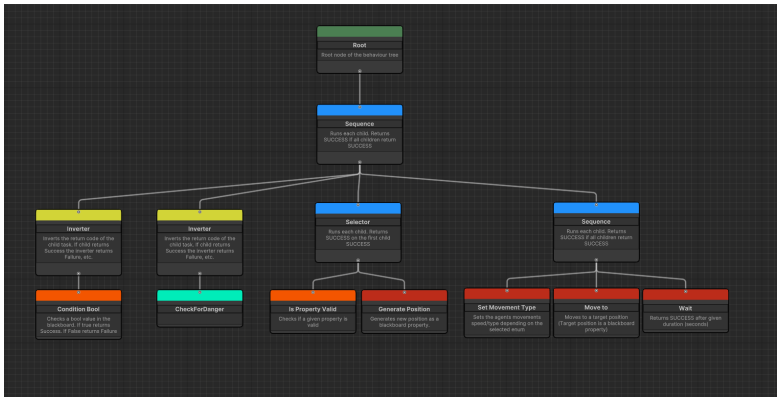



**Figure 5.8:** Patrol behavior tree.

The behavior then runs a selector. This selector is tasked with acquiring a target position to which the agent should move. It does this by checking if the target position blackboard property is valid and, if not, generates a new one with the **Generate New Position** action. Then the behavior moves to a sequence (the nested sequence in figure 5.8). This sequence is tasked by moving the agent to the target position. First, it sets the movement speed of the agent to the walking speed with the **Set Movement Speed** action. Then the **Move To Position** action is executed to move the player to the target position. And finally, a **Wait** action is run to make the agent stay at the target position for a while.

- **Controlled Patrol Movement**

  The behavior Controlled Patrol Movement, shown in figure 5.9, works the same way as the common behavior **Patrol** with two exceptions.

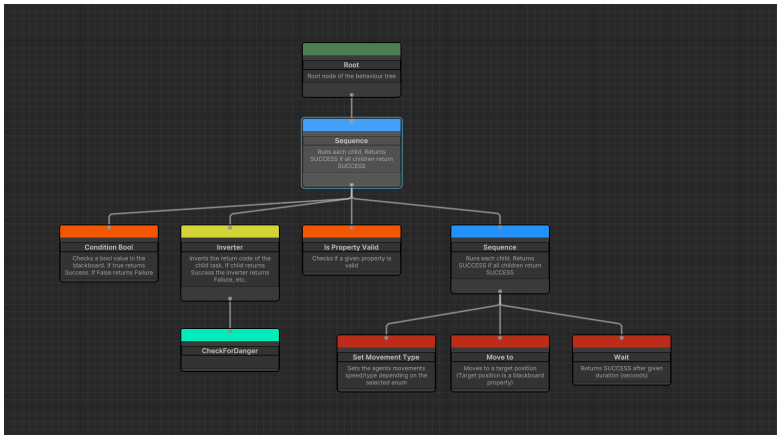



**Figure 5.9:** Controlled Patrol Movement behavior tree.

Firstly, unlike the **Patrol** behavior, this behavior expects to be controlled by another entity. Because of this, the first **Boolean Condition** task doesn't have a **Inverter** decorator attached. And Secondly, since the behavior expects to be controlled, it does not contain a **Generate New**

**Position** action. Instead, the behavior expects that the target position is present in the blackboard as a vector property. If the property is not valid or doesn't exist, the behavior always fails.

### 5.2.4 Character behavior types

As mentioned before, characters are implemented as non-player characters. Every character contains a character script component that uses my AI behavior tree decision-making system explained in chapter 3. To simplify the development process and ensure consistency, every character script has been implemented as a MonoBehavior class that inherits from a base abstract class called BaseCharacter. This base class implements the behavior tree agent's interface and provides a set of shared functionalities and actions that are common to all characters in the game. The game contains three basic types of characters: A sword and shield character, a two-handed weapon character, and a bow character. Each character has their own behavior defined by a separate behavior tree. Each of these behavior trees starts with a **Repeater** (2.3.2) decorator. Without this decorator, the tree would stop when the first composite node connected to the root node completes its execution. The following sub-section explains the individual behaviors of the mentioned characters.

#### Sword and Shield Character

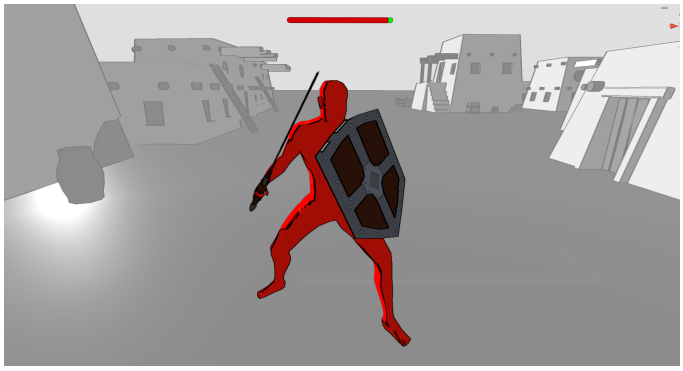


**Figure 5.10:** Sword and shield character.

The sword and shield character is a melee fighter equipped with a sword and a shield. Figure 5.10 shows how this character looks in the game. As mentioned, the character has its own behavior defined by a behavior tree visible in figure 5.11.

This character's behavior starts with a selector to choose an appropriate task depending on the importance. First, the selector tries to execute his left child task. this child task contains a sequence that performs a test to see if there are any visible enemies in the vicinity by executing the common behavior **Check for Danger**.

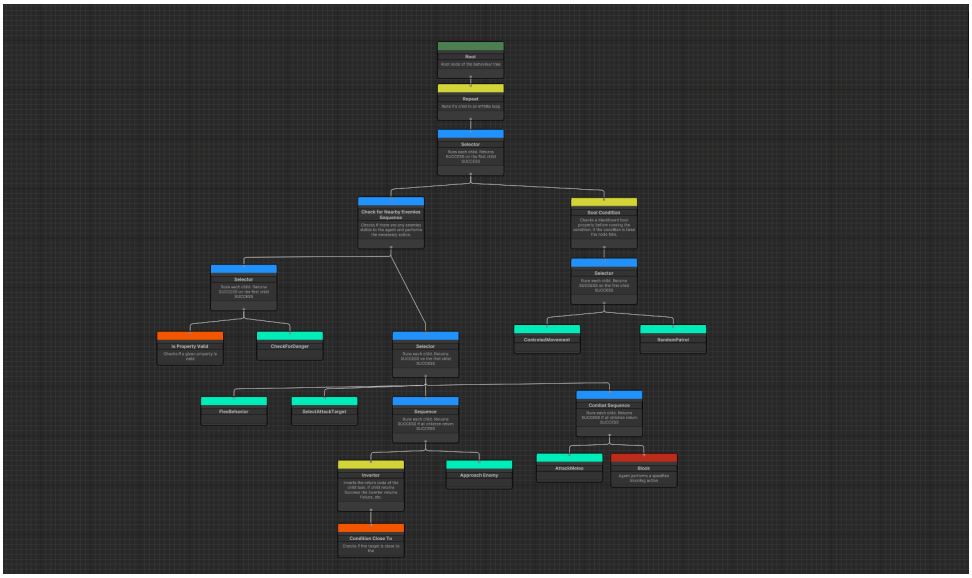


**Figure 5.11:** Behavior tree of the sword and shield character.

If there are no nearby enemies, the agent should start to patrol. The agent can patrol either individually by running the sub-task with a **Patrol** behavior or, if another entity is controlling the character, run the **Controlled Partol Movement** behavior.

If the character can see a nearby enemy, the behavior continues with another selector task to decide what to do next. First, the character checks to see if he should run away in the sub-task with the **Flee** behavior. If not, the behavior continues with the sub-task containing **Select Target** behavior. If the character already has a valid target stored as a property in the blackboard, the selector moves the following sub-task. This sub-task first checks if the selected target is within the attack range with the condition **Close To**, and if the target is too far away, the character starts moving towards the target with the **Approach Target** behavior. When the target is in the attack range, the character runs the **Attack** behavior to attack the target. If the attack is successful, the character then starts the blocking behavior by running the **Block** action.

## Two-handed weapon character

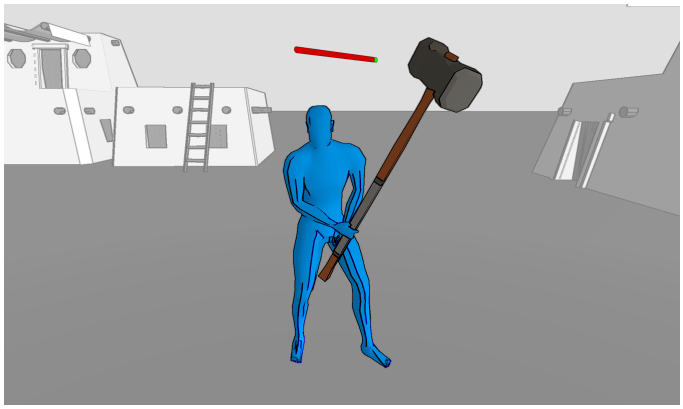


**Figure 5.12:** Behavior tree of the two-handed character.

Like the sword character, the two-handed weapon character, shown in the figure 5.12, is also a melee fighter. The character behaves to the behavior defined by his behavior tree, which is visible in figure 5.13. As we can see, the behavior tree is very similar to the behavior tree shown in figure 5.11. Because of this, the character behaves, in the same way, the sword and shield character does. The only difference is that this character cannot block attacks and holds a different implementation of the **Attack** action. Because of this, the character doesn't use the **Block** action after an attack.

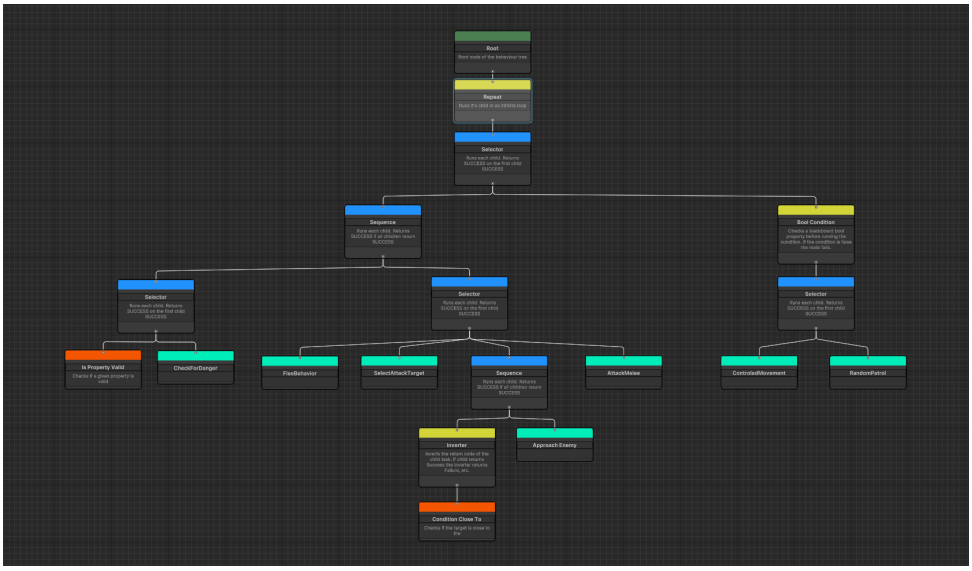



**Figure 5.13:** Behavior tree of the two-handed character.

### Bow character Character

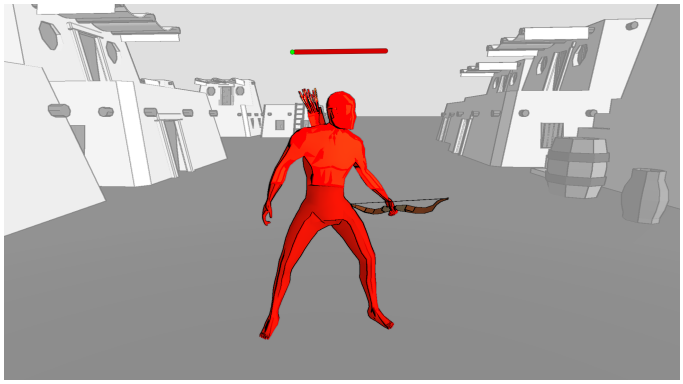


**Figure 5.14:** Behavior tree of the bow character.

The bow character shown in figure 5.14 is a ranged fighter. Because of this, his behavior, defined by the behavior tree in figure 5.15 is different from the sword and shield character or two-handed weapon character. Just like the other characters, this character also first decides if he should pick an appropriate action to handle a nearby enemy or to continue patrolling. It also checks if another entity controls him to pick the correct patrol behavior while testing with the **Pre Condition** decorator if the character has been hit.

When the character spots an enemy or already has a valid attack target, the character checks if he should run away because of low health. This is simulated in the sub-task with a **Flee** behavior. Then the character checks if he should run away because the enemy is too close.

If The target is not running away he continues to choose an attack target in the sub-task with a **Select Target** behavior. When the character has a valid attack target he performs a check to see if the given target is in attack range. If not, the character approaches the target. When the target is in the attack range he starts to attack the target.

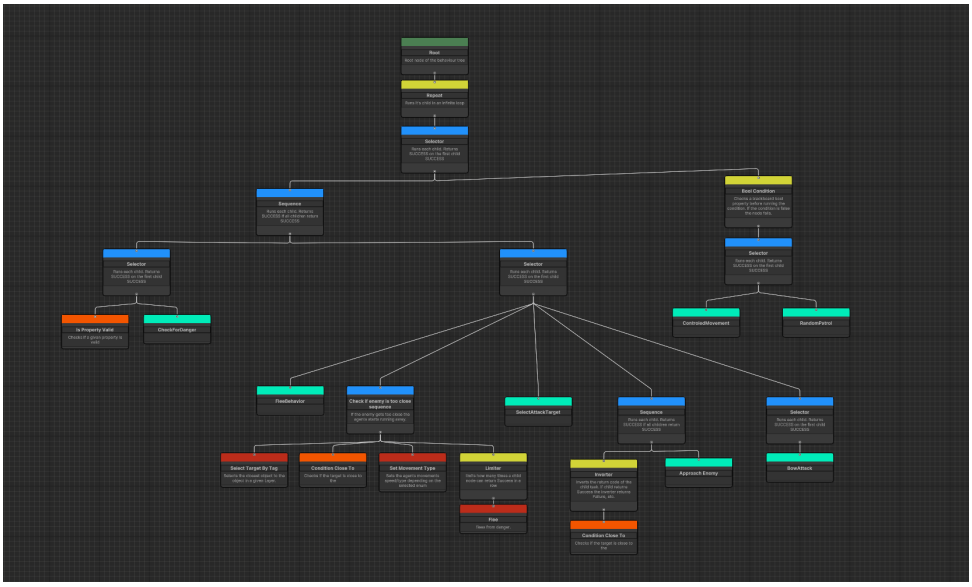


**Figure 5.15:** Behavior tree of the bow character.

## 5.3 Party Controllers

Characters can also be arranged into a party or a squad. The party is implemented as a separate game object with a party controller script component. This component holds a reference to all units (characters in the party). It is responsible for moving and arranging the party in an organized way as shown in figure 5.16. The controller also contains a variable for setting the spacing between each character. For a character to be controlled by a party controller, he must contain a unit controller script component.

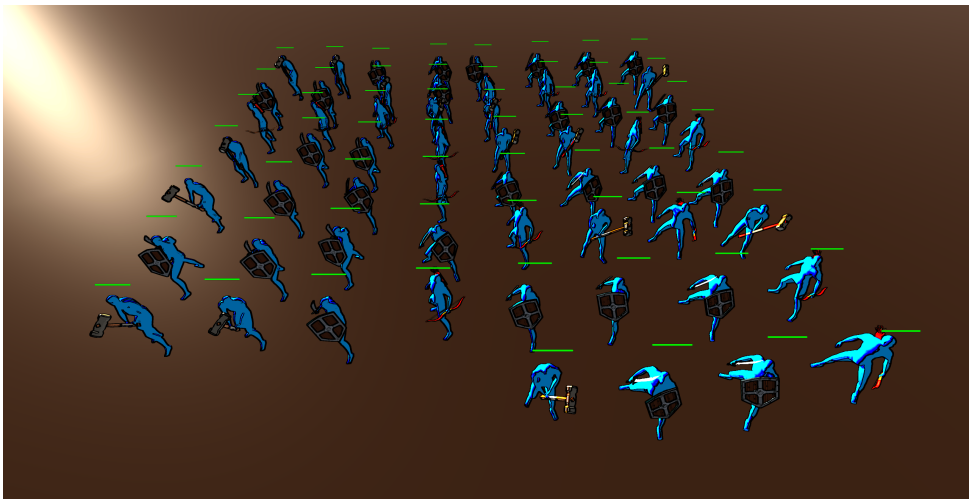


**Figure 5.16:** Characters in a party.

The unit controller marks the character as being controlled. This means

that the character will use a **Controlled Patrol Movement** behavior instead of the standard **Patrol** behavior. The unit controller creates and updates a blackboard vector property. This property holds the position to which the character should move.

There are two types of party controllers: a player party Controller and an enemy (or AI) party Controller. The Player party controller uses the input from a mouse to choose a new position.

The enemy party controller has a list of game objects which serve the purpose of waypoints. It also implements the behavior tree agent's interface. Because of this, the controller behaves according to his behavior tree shown in figure 5.17. This simple behavior uses a sub-tree task with a **Check for Danger** behavior to test if there are any enemies nearby. The outcome of the sub-task is then inverted by the **Inverter** decorator. This way, the task returns a successful state if there are no visible enemies nearby. If there are no enemies in the vicinity, the behavior starts to patrol. First, the behavior moves to the current waypoint with the **Move to Game Object** action. When a waypoint is reached, the party waits for a given duration with the **Wait** action. After that, a new Waypoint is selected with the **Get Next Waypoint** action, and the behavior starts over again.

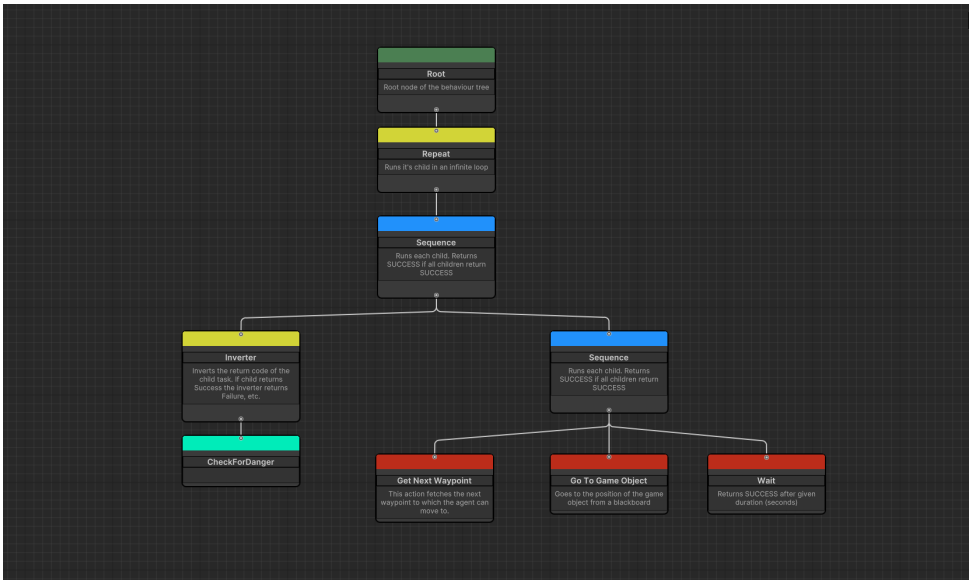



**Figure 5.17:** Enemy party controller's behavior tree.

# Chapter 6

# Optimization and Performance Tests

## 6.1 Optimazation

As explained in the previous chapter 5, each character has its own behavior tree asset that needs to be executed (or updated) in order to simulate the given behavior. When the behavior tree is executed, we need to traverse the tree to find a new appropriate action. This creates a performance overhead. To lower the overhead, I have tried two different approaches to calling the update method for a behavior tree.

First, I simply called the behavior tree update method in the character's **Update** method, which is called every frame. This approach works well but creates an unnecessary performance overhead since the tree is updated every frame.

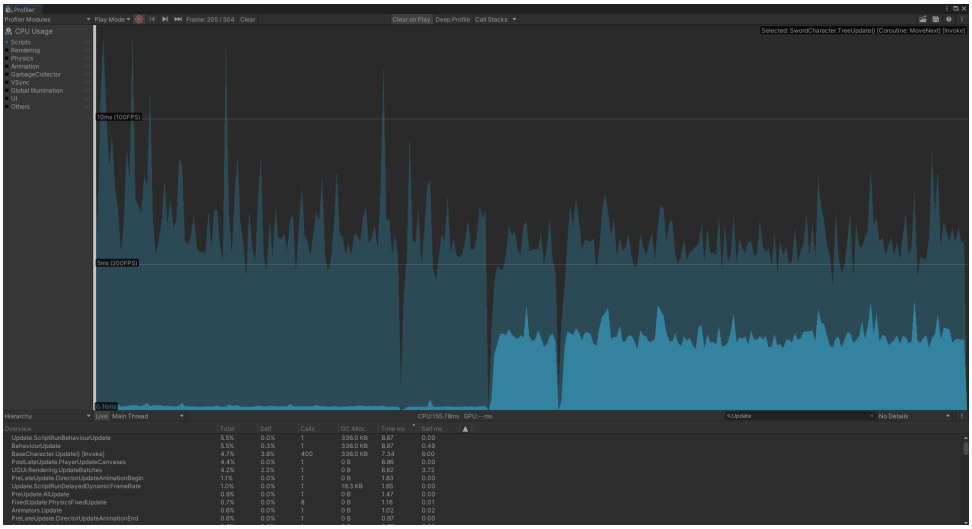
```
Frequently called  1 usage
private IEnumerator TreeUpdate()
{
    float interval = Random.Range(0.03f, 0.09f);

    while (true)
    {
        if (!isStunned && !isDead)
        {
            tree.Update();
        }

        yield return new WaitForSeconds(interval);
    }
}
```

**Figure 6.1:** Behavior Tree coroutine update method.

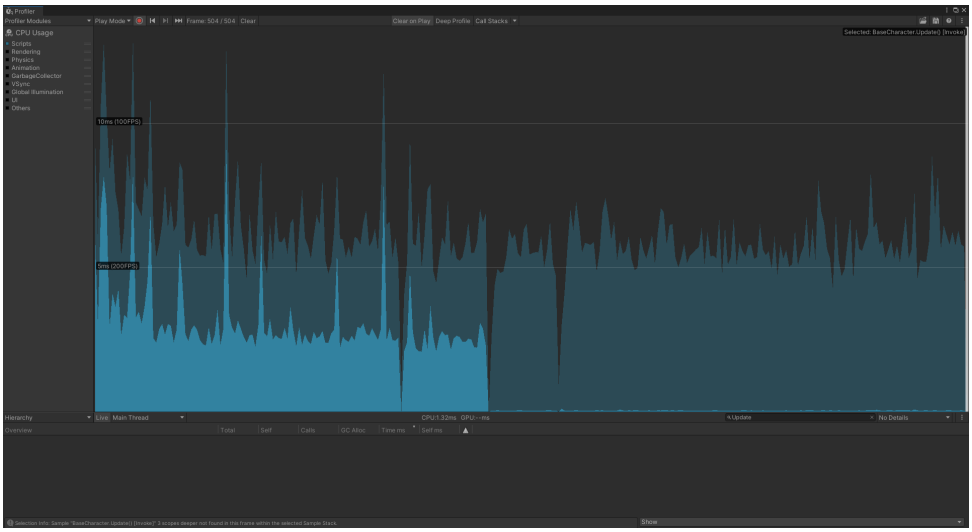
To simulate a believable character, we don't have to update the behavior tree as often as every frame. Instead, the tree can be updated every given interval. This interval can be modifiable and even used as a game design tool when creating new character behaviors (for example: to simulate a slow or stupid character, the tree can be updated less frequently). Because of this, I created a coroutine method within the character base class, visible in figure 6.1. Coroutine is a function that allows pausing its execution and resuming from the same point after a condition is met. This coroutine is responsible

for updating the behavior tree.

The method first generates a random number from an interval of very small values (the float variable **interval** in figure 6.1). This number is the delay time between the behavior tree update calls. The number is generated from a random range to spread the update calls across a period of time. To see the difference between this approach and updating the behavior tree on every frame, I added an option to switch between these approaches to the base character class, and I analyzed the CPU usage with a unity profiler.



(a) Updating behavior tree with coroutine with a random interval



(b) Updating behavior tree on every frame

**Figure 6.2:** Unity profiler showing CPU usage during decision-making simulation

The two sub-figures, shown in figure 6.2 show the CPU usage of the behavior tree update calls with a large number of characters present in the scene. The sub-figure 6.2 (b) shows the CPU usage while the behavior update method

has been called on every frame, and sub-figure 6.2 (a) shows the usage while the coroutine with a random interval between 0.03s and 0.09s was being used. As we can see, the coroutine method offers a small performance improvement.

## 6.2 Performance Test

To test how much the decision-making system impacts the game's performance, I created a simple scene for performance testing. This will only contain a camera, a plane object for the ground, and characters. To make the testing easier, I created a simple script for spawning a given number of **Sword and Shield** characters at the start of the game.

I tested how the system works with 100 (figure 6.3) and 1000 (figure 6.4) characters. Every time I spawned, half of the characters were marked with the tag "Player" (blue characters in figures 6.3 and 6.4) and the other half marked with the tag "Enemy" (red characters in figures 6.3 and 6.4). Because of this, the characters can engage in combat.

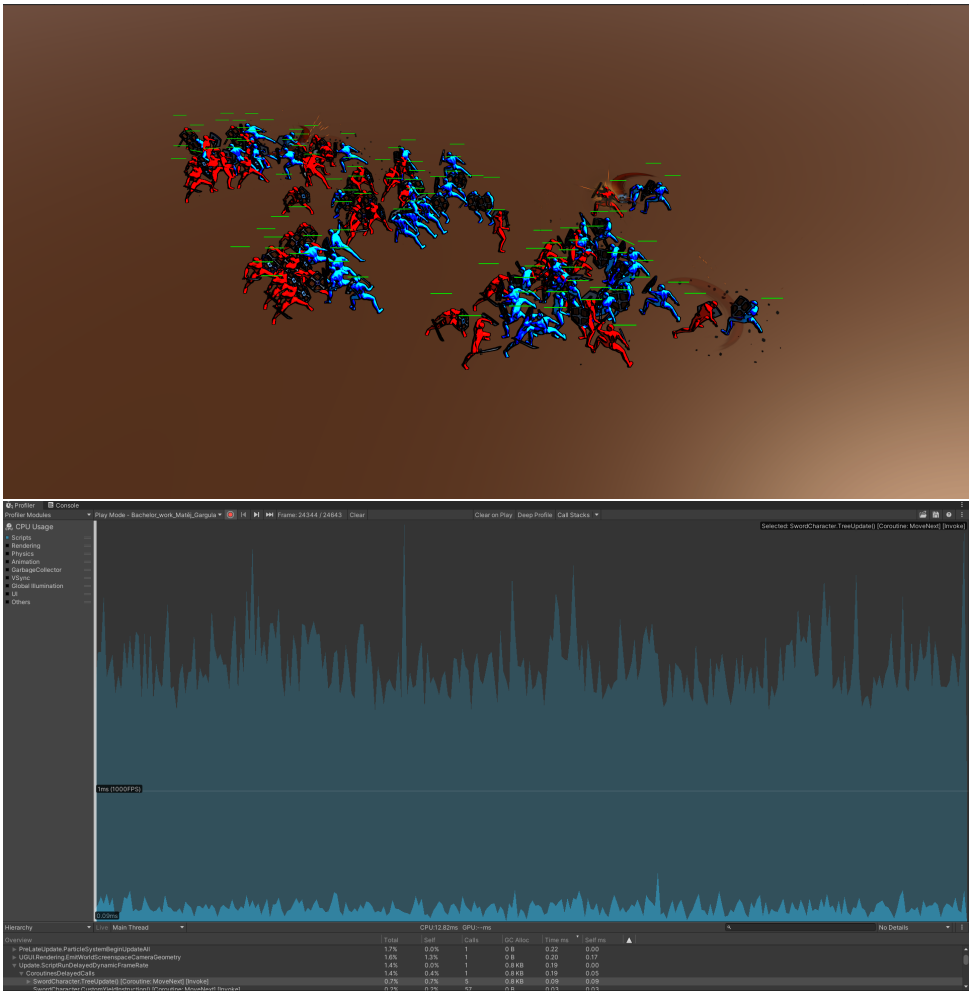


**Figure 6.3:** Performance test with 100 characters

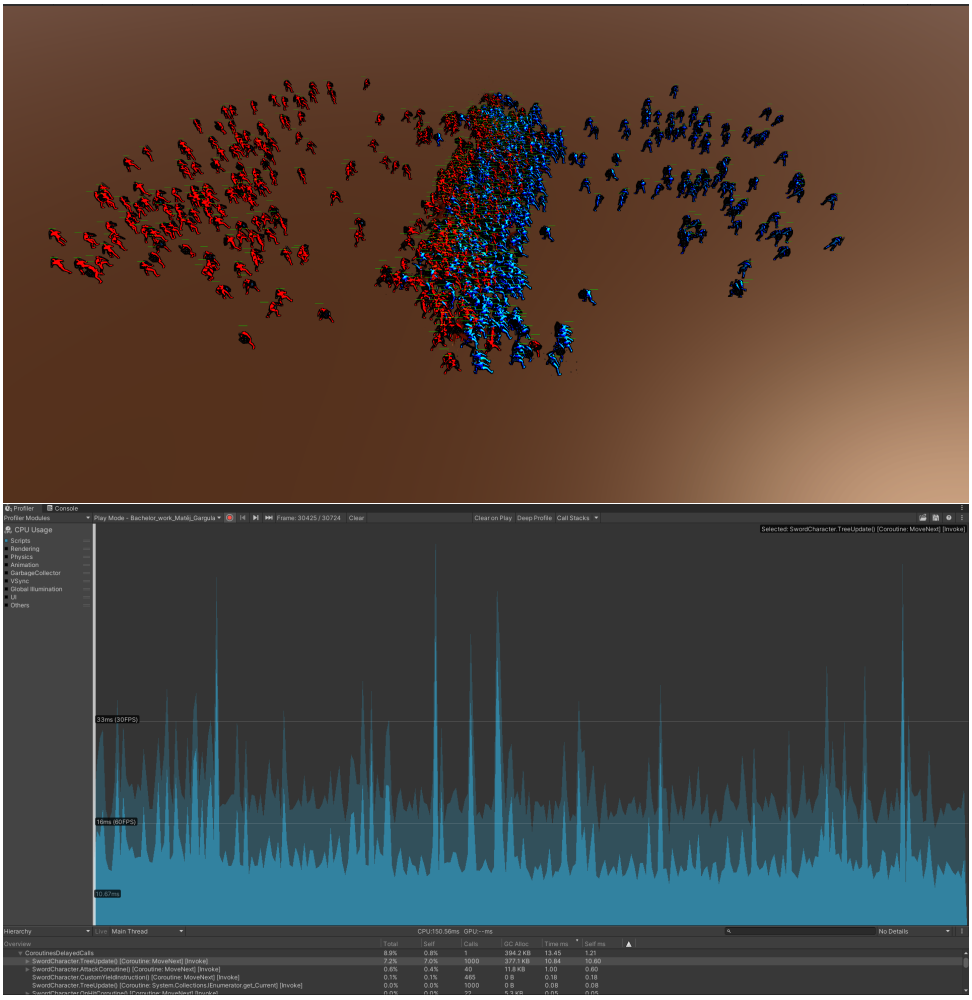


**Figure 6.4:** Performance test with 1000 characters

Figures 6.3 and 6.4 shows a screenshot from the scene and a unity profiler window. The unity profiler shows the CPU usage of script components (dark blue color) and the usage of the behavior tree update call (bright blue color).

As we can see in figure 6.3, the AI system does not have a significant impact on the overall performance of the game. On the other hand, figure 6.4 shows that the system can impact performance when a very large number of characters is present within the game compared to other script components. This is likely caused by actions that require finding all characters in the scene (for example, the **Select Closest By Tag** action). Figure 6.5 shows the overall CPU usage of the behavior tree update calls along with rendering, physics simulation, and animation handling. Compared to the other parts of CPU usage, the AI system can still impact the overall performance of the game slightly.


**Figure 6.5:** Profiler screenshot from a test with 1000 characters

# Chapter 7
# Conclusion

In this project, I successfully researched several possible techniques used for simulating the decision-making of AI characters in games. In unity, I created an AI decision-making system with behavior trees. This system enables users to create and modify behavior trees in Unity and later use them in a game. To show how the system can be used, I created a game project to showcase the capabilities of the system.

In the following sections, I will review and summarize the work done on the project and go through possible improvements for the system.

## 7.1 What Was Achieved

I have successfully created a tool for modeling and storing behaviors for AI characters within games. To use this tool, the user first has to create a behavior tree asset. This asset stores the model of the behavior tree. To edit or modify this asset, the user has to use the created custom behavior tree editor. In the editor, the user can create new nodes, connect existing nodes, or modify the properties of individual nodes through the inspector. The editor also contains a blackboard view used to display, create and modify the values of the properties within the agent's blackboard.

The created behavior tree asset can then be used in the agent script. The agent is connected with the behavior tree by an interface. This way, each new type of agent can be easily adapted for many different situations. It also means that two agents can share the same structure of behavior and have different implementations for each agent.

Users can also edit the currently active behavior trees in the editor even while the game is running. The editor also works as a visualization tool during runtime. While the game runs, each node shows its current state with different color-coded borders.

To showcase the possibilities of the behavior tree system, I created a game project. In the project, I created several different common behaviors that are often used in games. These common behaviors were then used to create three more complex behaviors. To use these complex behaviors, I created three types of characters (agents). Each character uses its own behavior tree for decision-making.

Lastly, I tested the performance of the decision-making system with varying numbers of characters present in the scene.

## 7.2 Possible Improvements

There are still things that could be improved in my decision-making system.

The editor is missing an easy way of creating new actions and more user-friendly functions such as copy and paste. Also, the editor could use a better way of displaying multiple behavior trees at once and an option for hiding and showing individual sub-trees.

The behavior tree itself can also still be improved. The tree still can have problems with simulating interrupting behavior (i.e., moments when we want to stop with the current task and look for a different one). It is possible to implement this behavior with a decorator task, but the design can be challenging in larger, very complex behaviors.

The decision-making system can also still be more optimized. For example, the performance could be improved by using a more event-driven approach, where tasks that require a larger amount of time to execute could only send a message with the task result instead of constantly checking the state of the task with the update/tick method.

# Bibliography


[1] Ian Millington, John Funge. *Artificial Intelligence for Games, 2nd edition.* CRCR Press, 2009.

[2] Petter Ogren. *Increasing Modularity of UAV Control Systems using Computer Game Behavior Trees.* Aiaa guidance, navigation, and control conference, 2012.

[3] Colledanchise, M., and Ögren, P. (2018). *Behavior trees in robotics and AI: An introduction.* CRC Press.

[4] Debby Nirwan. *Hierarchical Finite State Machine for AI Acting Engine.* https://towardsdatascience.com/hierarchical-finite-state-machine-for-ai-acting-engine-9b24efc66f2.

[5] Marcotte, R., and Hamilton, H. J. (2017). *Behavior trees for modeling artificial intelligence in games: A tutorial.* The Computer Games Journal

[6] Colledanchise, M., Parasuraman, R., and Ögren, P. (2018). *Learning of behavior trees for autonomous agents.* IEEE Transactions on Games,

[7] Game Developer. *Behavior trees for AI: How they work.* https://www.gamedeveloper.com/programming/behavior-trees-for-ai-how-they-work

[8] Unity Manual. *UI toolkit* https://docs.unity3d.com/Manual/UIElements.html

[9] Unity Manual. *Immediate Mode Graphical User Interface System Basics.* https://docs.unity3d.com/Manual/GUIScriptingGuide.html

[10] Unity Manual. *Scriptable objects* https://docs.unity3d.com/Manual/class-ScriptableObject.html

[11] NPC Wikipedia. https://en.wikipedia.org/wiki/Non-player_*character*

[12] David M. Mount. *Game Programming Lecture Notes.*

[13] Data Conomy. *Artificial intelligence games: What is AI in gaming?.* https://dataconomy.com/2022/04/29/artificial-intelligence-games/

[14] PC Gamer. *GDC 2013: The AI tricks behind XCOM, Assassins Creed 3 and Warframe*. https://www.pcgamer.com/gdc-2013-the-ai-tricks-behind-xcom-assassins-creed-3-and-warframe/

[15] Arts Management and Technology Laboratory. *How AI Is Used in Video Games: The Sims 4 and Red Dead Redemption 2*. https://amt-lab.org/blog/2023/4/how-ai-is-used-in-video-games-the-sims-4-and-red-dead-redemption-2