



# Design and implementation of an artificial intelligence for a horror videogame in Unreal Engine 5

Miguel Martínez Martínez

Final Degree Work

Bachelor's Degree in  
Video Game Design and Development  
Universitat Jaume I

July 3, 2023

Supervised by: Jorge Sales, PhD.





To those who are chasing their dreams



## ACKNOWLEDGMENTS

First of all, I want to thank my parents for always supporting me, without them I would not have made it this far. My friends who have accompanied me during this trip, especially to Adri, Pepe, Guillem, Javi, Ferran and Víctor.

I would also like to thank Jorge Sales for his supervision of this project.

I also want to thank Karen Calanni (a.k.a DEMONDICE), her music has helped me in the worst moments and has reminded me why I should keep working hard.

I also would like to thank Sergio Barrachina Mir and José Vte. Martí Avilés for their inspiring LaTeX template for writing the Final Degree Work report, which I have used as a starting point in writing this report.



# **ABSTRACT**

This document presents the Final Degree Work report of Miguel Martínez Martínez in Bachelor's Degree in Video Game Design and Development.

The main concept is how to implement a main enemy for a horror video game using Unreal Engine 5, employing the various systems that this game engine provides for the creation of non playable characters. This enemy will be able to learn according to its state and interactions with the player, allowing it to react to certain events in different ways to achieve unpredictable behavior.

In addition, some of the behaviors, actions and systems obtained after the creation of the enemy are reusable for other video game genres.

# **KEYWORDS**

Unreal Engine, artificial intelligence, horror, asset, EQS



# CONTENTS

<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>ix</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Work Motivation . . . . .	1
1.2 Objectives . . . . .	2
1.3 Environment and Initial State . . . . .	3
<b>2 Planning and resources evaluation</b>	<b>5</b>
2.1 Planning . . . . .	5
2.2 Resource Evaluation . . . . .	7
<b>3 System Analysis and Design</b>	<b>11</b>
3.1 Requirement Analysis . . . . .	11
3.1.1 Functional Requirements . . . . .	12
3.1.2 Non-functional Requirements . . . . .	13
3.2 System Design . . . . .	13
3.3 System Architecture . . . . .	19
3.4 Interface Design . . . . .	20
<b>4 Work Development and Results</b>	<b>21</b>
4.1 Work Development . . . . .	21
4.1.1 First steps: Creating the classic behavior . . . . .	21
4.1.2 Adding the lockers . . . . .	31
4.1.3 Hierarchical pathfinding . . . . .	37
4.1.4 Working in more detail on the design . . . . .	40
4.1.5 Cowardly behavior: The Environment Query System . . . . .	44
4.1.6 Pulling the player out of hiding . . . . .	53
4.1.7 Strafing the player . . . . .	55
4.1.8 Modifying the agent's statistics . . . . .	60
4.1.9 Food searching behavior . . . . .	61

4.1.10	Patrolling using the Environment Query System . . . . .	67
4.1.11	Modifying the NavMesh navigation cost . . . . .	68
4.1.12	Agent reacts to light: custom sense of sight . . . . .	69
4.1.13	Switching between agent's states: Modular AI . . . . .	71
4.2	Results . . . . .	74
<b>5</b>	<b>Conclusions and Future Work</b>	<b>77</b>
5.1	Conclusions . . . . .	77
5.2	Future work . . . . .	78
	<b>Bibliography</b>	<b>79</b>

## LIST OF FIGURES

2.1	Gantt chart depicting the initial project planning . . . . .	8
2.2	Gantt chart depicting the tasks performed in the project . . . . .	9
3.1	Agent's case use diagram . . . . .	18
3.2	Agent's state machine diagram . . . . .	19
3.3	Debugging interface . . . . .	20
4.1	Assign AIController to the agent . . . . .	22
4.2	Senses available through the AIPerception component . . . . .	24
4.3	Sense of sight setting options . . . . .	24
4.4	On Target Perception Updated node . . . . .	25
4.5	Logic behind the player entering and leaving the agent's field of view . . . . .	26
4.6	Surface where the agent can walk . . . . .	27
4.7	Specify BT to be used in the AIController . . . . .	27
4.8	BTD_iAmPursuingThePlayer? details . . . . .	30
4.9	BTT_FindAndChasePlayer . . . . .	30
4.10	BTT_SearchForPlayer . . . . .	31
4.11	First version of the agent's behaviour tree . . . . .	32
4.12	Lockers used in the project . . . . .	34
4.13	Acceptable Radius parameter . . . . .	34
4.14	Event DidEnemySee . . . . .	35
4.15	Event LeftHiddingSpot . . . . .	36
4.16	Kill Hidden Player sub tree . . . . .	37
4.17	Design of the test environment . . . . .	38
4.18	BP_ZoneNode BeginPlay event . . . . .	38
4.19	Representation of the graf created by the nodes of the rooms . . . . .	39
4.20	Hierarchical Patrol sub tree . . . . .	40
4.21	Stat Struct . . . . .	41
4.22	Data Table . . . . .	42
4.23	BeginPlay event of the GameState . . . . .	43
4.24	Loop to every row in the table by name . . . . .	43
4.25	Store the data in a local array in the AI agent . . . . .	44
4.26	Adding a delay in the BeginPlay event . . . . .	44
4.27	EQSC_Player . . . . .	45

4.28	Different types of Generators . . . . .	46
4.29	Actors Of Class details . . . . .	47
4.30	Different types of Tests . . . . .	47
4.31	Environment Query EQ_FindPlayer testing . . . . .	48
4.32	Grid produced by the Generator "Points: Grid" . . . . .	49
4.33	Result of EQ_HideFromPlayer after applying a Trace Test from the Player .	50
4.34	Environment Query EQ_HideFromPlayer testing . . . . .	51
4.35	Result of EQ_HideFromPlayer after applying a Pathfinding Test to the Agent . . . . .	52
4.36	Coward Behaviour sub tree . . . . .	53
4.37	Store the reference to the locker on the blackboard . . . . .	54
4.38	Teleport player out of the locker . . . . .	55
4.39	Bring the player out of hiding sub tree . . . . .	55
4.40	Circumference where the new positions of the agent will be generated . . . . .	56
4.41	Forward and right vectors . . . . .	57
4.42	Straf positions . . . . .	57
4.43	BTS_IsInRangeToPlayer . . . . .	58
4.44	Straf behaviour sub tree . . . . .	59
4.45	BTT_ModifyStat . . . . .	61
4.46	Register Perception Stimuli Source . . . . .	62
4.47	EQ_FindFood . . . . .	63
4.48	EQ_FindSafeRoomNode . . . . .	64
4.49	Testing EQ_FindSafeRoomNode . . . . .	64
4.50	EQSC_SafeRoom . . . . .	64
4.51	EQ_FindSafeRoomPath Item grid . . . . .	65
4.52	Sorted item grid by distance to the node . . . . .	65
4.53	EQ_FindSafeRoomPath . . . . .	66
4.54	Items generated for the patrol through EQS . . . . .	68
4.55	Sub-tree for patrolling through EQS . . . . .	69
4.56	NA_Cheap . . . . .	69
4.57	Invisible component for light detection . . . . .	70
4.58	Distracted behaviour sub tree . . . . .	71
4.59	Evolution sub tree . . . . .	73
4.60	The different behaviors for the agent patrol . . . . .	73
4.61	The different behaviors the reaction of the agent to the player interacting with the locker . . . . .	74
4.62	The different behaviors when the agent reaches the player . . . . .	74

## LIST OF TABLES

2.1	Total project cost . . . . .	10
3.1	Case of use «CU01. Movement» . . . . .	13
3.2	Case of use «CU02. Interaction» . . . . .	14
3.3	Case of use «CU03. Use flashlight» . . . . .	14
3.4	Case of use «CU04. Hide» . . . . .	15
3.5	Case of use «CU05. Sprint» . . . . .	15
3.6	Case of use «CU06. Agent Movement» . . . . .	16
3.7	Case of use «CU07. Reaction to stimulus» . . . . .	16
3.8	Case of use «CU08. Kill the player» . . . . .	17
3.9	Case of use «CU09. Agent states» . . . . .	17



CHAPTER



# INTRODUCTION

## Contents

---

1.1	Work Motivation . . . . .	1
1.2	Objectives . . . . .	2
1.3	Environment and Initial State . . . . .	3

---

This chapter shows the main motivations for the development of this project, as well as the objectives to be achieved and its first stage of development.

### 1.1 Work Motivation

Horror is undoubtedly one of the most famous and relevant genres nowadays. From paintings such as the portrait of Pope Innocent X by Francis Bacon, the music created by the British Brian "Lustmord" Williams, through literature, cinema and, of course, video games.

Currently, the horror video game is at one of its highest moments in history, where the great sagas of the genre are becoming relevant again, such as Resident Evil (Capcom, 1996), with its latest entry has sold more than 4 million copies in two weeks [30], and Silent Hill (Konami, 1999), where a remake of its second game is expected to be released this year 2023 [36]. On the other hand, the indie scene has seen an increase of developers interested in this genre, and at present, on itch.io<sup>1</sup>, the most popular genre of games developed on the platform are horror games with a total of 34.682 results [26] at the time of writing this report. Because of this, a lot of works are appearing that leave much

---

<sup>1</sup>itch.io is an open marketplace for independent digital creators with a focus on independent video games. It's a platform that enables anyone to sell the content they've created [25].

to be desired by AAA<sup>2</sup>/AA<sup>3</sup> studios and indie developers. Games with bad mechanics, level design, atmosphere and, what I think is one of the most important factor in this genre, artificial intelligence.

For this reason I have decided to focus this project on the implementation of various techniques and algorithms to achieve an agent that does not falls into the same failures of some of the games that are in the current market.

Apart from the horror genre and artificial intelligence, I have developed this project in the Unreal Engine 5<sup>4</sup> game engine. I decided to develop it in this software because this engine is taking more and more weight in the gaming industry and some companies are beginning to replace their own engine by Epic Games engine [21], so I think it is a good point to force myself to learn this engine that will open some doors in my future job as a game developer. Also, Unreal Engine has more features and AI tools than the Unity engine<sup>5</sup> [33], which will make it easier for me to reach my goals.

## 1.2 Objectives

Considering the motivations behind the development of this project, here I present the objectives that I seek to achieve with this work:

- Learn Unreal Engine from scratch.
- Learn and understand the various system provided by Unreal Engine to implement artificial intelligence.
- Learn how to combine the artificial intelligence systems provided by Unreal Engine.
- Implement artificial intelligence techniques to create an unpredictable agent.
- Create an agent capable of evolving through three states.
- Make each agent state a different level of difficulty for the player.
- The agent reacts to stimulus that it perceives through his sight.
- Understand reinforcement learning and how I can use it for my project.

---

<sup>2</sup>In the video game industry, AAA (pronounced triple-A) is an informal classification used to categorise video games produced and distributed by a mid-sized or major publisher, which typically have higher development and marketing budgets than other tiers of games.

<sup>3</sup>AA or Double-A games are mid-market video games that typically have some type of professional development though typically outside of the large first-party studios of the major developers; these may be from larger teams of indie developers in addition to larger non-indie studios.

<sup>4</sup>Unreal Engine (UE) is a 3D computer graphics game engine developed by Epic Games, first showcased in the 1998 first-person shooter game Unreal. Initially developed for PC first-person shooters, it has since been used in a variety of genres of games and has seen adoption by other industries, most notably the film and television industry.

<sup>5</sup>Unity is a cross-platform game engine developed by Unity Technologies, first announced and released in June 2005 at Apple Worldwide Developers Conference as a Mac OS X game engine.

- Learn how to implement reinforcement learning in Unreal Engine.

### 1.3 Environment and Initial State

One of the things that worried me the most when I started this project was to learn a new engine from scratch. Throughout my studies I have been taught the basics of Unity and how to implement various techniques and algorithms in it, and having to learn how to do all this in a new engine I knew it was something that was going to take up some time. So, I started looking for documentation and tutorials about Unreal Engine, and to my surprise Unreal Engine's official YouTube channel [19] has a good variety of fairly exhaustive tutorials on their various artificial intelligence systems. I also looked for some tutorials to learn the basic functions of Unreal Engine such as how to move around the software and its blueprints system.

For the most basic things I look up for tutorials on the Udemy website<sup>6</sup> hoping to find some cheap tutorials about Unreal Engine. In a brief search, the page gave me 3.744 results for Unreal Engine, but knowing everyone can upload content on this website, I decided to keep only those with a high rating to avoid poor quality content. Now with a search of 1.434 results I immediately found a tutorial called "*Desarrollo de juegos con Unreal Engine de 0 a profesional*" [3] that caught my attention. This course taught by Carlos Coronado teaches notions of game design, Unreal Engine blueprints, materials creation, particles, interfaces and lights to create 3D games with a AAA finish. Moreover, I found another tutorial called "Unreal Engine: Ultimate Survival Horror Course" [35] created by Aidan Perry which teaches how to create a horror video game from its basic mechanics to artificial intelligence.

With much of the material to start learning Unreal Engine, I decided to start expanding my idea for the agent I was going to create. My initial concept is based on the video games Alien: Isolation (Sega, 2014), Resident Evil 2 Remake (Capcom, 2019) and Outlast (Red Barrels, 2013). In these games, some of the enemy agents patrol the scenery looking for the player and when they find him they will start pursuing the player. In Resident Evil 2 Remake you are able to defend yourself from this type of NPCs, and you can incapacitate him to escape. The point with this enemy is that it is quite good at finding the player and has even been proved that the developers gave him the ability to teleport to a position close to the player if they makes a noise while the agent is off camera [6]. On the other hand, in my other two references you can hide from agents by sneaking in lockers or under some furniture such as tables and beds. This mechanics is well thought in both games although it has some problems to detect if the player is hiding in front of the AI or to find it. My initial idea was to merge certain concepts from each of these games to get a similar agent, but I didn't want to stay with such a simple idea. During my learning of Unreal Engine, I saw that the behavior trees have a tool

---

<sup>6</sup>Udemy is a massively-open online course (M.O.O.C.) website where anyone is free to create and promote courses in the style of traditional post-secondary education. Users can also take courses to earn credit towards technical certification, or just to pick up or improve various job-related skills [38].

called Services which allows to perform operations before the execution of a tree or sub tree, allowing to modify the flow of the tree according to the result of the service in order to archive a more unpredictable behaviour. With this feature and providing the agent with statistics, which are modified according to its interaction with the environment and the agent, I arrive at a more complex idea where my agent can change its behavior or actions according to its parameters.

This new idea was starting to grow on me, but I still felt gutted about it. I defined this new idea that my agent is able to evolve and I knew that this definition meant machine learning. After some research and consultation with Carlos Marin Lora, I came to the conclusion that to achieve a good result I had to implement reinforcement learning. Carlos suggested me to search the Epic Games marketplace<sup>7</sup> [18] for a machine learning plugin, and as soon as I clicked the search button I found a free plugin called MindMaker<sup>8</sup> [2] created by Aaron Krumins. I started looking at the documentation and playing around with the plugin, and after a week of work I had an agent which moved around a small environment looking for food every time its energy level drops to 0. The last thing I needed to have the basics of this part was to store the observations and their respective rewards in a CSV<sup>9</sup> or JSON<sup>10</sup> file so that I could use this data in my agent. Unreal Engine does not have nodes to be able to export data directly to these formats, searching in the marketplace I found the Blueprint FileSDK plugin<sup>11</sup> [23], which allows to export data directly to the formats I need.

With everything I needed covered, I was ready to start developing the project.

---

<sup>7</sup>The Epic Games marketplace is an online platform where users can upload their own assets or plugins for free or at a price.

<sup>8</sup>The MindMaker AI Plugin is an open-source plugin that enables games and simulations within UE4 and UE5 to function as environments for training autonomous machine learning agents. The plugin facilitates a network connection between an Unreal Project containing the learning environment, and a standalone machine learning library used by the agent to optimize whatever it is attempting to learn.

<sup>9</sup>A comma-separated values (CSV) file is a delimited text file that uses a comma to separate values.

<sup>10</sup>JSON (JavaScript Object Notation), is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects.

<sup>11</sup>FileSDK is a simple plugin that provides blueprint nodes for disk IO (input/output) operations. The provided nodes will allow you to read, write, copy, and delete files. The plugin also provides nodes for creating, deleting, copying, and searching directories.

CHAPTER



## PLANNING AND RESOURCES EVALUATION

### Contents

---

2.1 Planning . . . . .	5
2.2 Resource Evaluation . . . . .	7

---

This chapter shows the planning for the project as well as the software and other resources used.

### 2.1 Planning

At the beginning of this project I did not have much knowledge about Unreal Engine, for this reason, my initial planning was based on an estimation of what I thought it would cost me to implement my initial idea and trying to adjust to the 300 hours. In addition, I did not consider the need to research information and look for documentation. Also the tasks I specified do not match with the final ones since some of the algorithms I wanted to implement are already in Unreal Engine's systems. In combination with the research and study of a new engine and some other problems, my work has ended up exceeding the estimated hours for this subject.

Not all tasks have been performed in the order in which they are shown; most of them have been performed in parallel.

- **Information gathering and study (60 hours):** look for research articles to improve my AI knowledge. Besides, taking Unreal Engine courses as well as looking for tutorials or information to learn the software.

- **Player character design (1 hour):** create the playable character with which the player will be able to move around the game environment.
- **Mechanics concept design (2 hours):** define the rules and actions that the player will be able to perform.
- **Mechanics design (15 hours):**
  - **Flashlight (3 hours):** creation and programming of an actor that will allow the player to illuminate the environment as well as distract the NPC on certain occasions.
  - **Lockers (9 hours):** creation and programming of an actor that will allow the player to enter on it to hide from the NPC.
  - **Doors (3 hours):** creation and programming of an actor the player will be able to interact to make his way through the various rooms of the level.
- **AI concept design (15 hours):** define the objectives and requirements for the AI, as well as the appropriate systems and algorithms to use.
- **AI design (230 hours):**
  - **AIController/Perception (35 hours):** programming the various responses that the agent will have to certain stimulus (sight and noise) perceived in the environment.
  - **Behaviour Tree (80 hours):** the construction and programming of various behaviours for the NPC.
    - \* **Services (15 hours):** programming several functions for the behaviour tree that will allow the tree flow to be modified according to certain given values or environmental events.
    - \* **Tasks (75 hours):** programming the multiple leaf nodes that will contain the agent's actions.
  - **Pathfinding (25 hours):** add the different components to the game level to use Unreal Engine's NavMesh as well as the creation and programming of actors that will have the function of pathfinding nodes.
  - **Environment Query System<sup>1</sup> (EQS) (45 hours):** programming Environment Query Tests to allow the AI to make decisions based on its environment.
  - **Interaction system (25 hours):** programming the various actions that the AI will be able to perform when interacting with other elements of the game level.

<sup>1</sup>The Environment Query System (EQS) is a feature within the Artificial Intelligence system in Unreal Engine that is used to collect data from the environment. We can ask questions about the data collected through a variety of different Tests which produces an Item that best fits the type of question asked.

- **AI statistics (20 hours):** create a data table with the agent’s statistics and parameters that the AI can access at any time.

## 2.2 Resource Evaluation

In the video game industry we find a wide variety of professional profiles, and depending on the worker’s experience, their salary can vary drastically. Doing a quick research on job search portals, the average salary of a junior video game programmer is 32,100 € gross per year [28], leading to a salary of 1,740.00 € net per month. Since this project has been developed over an interval of 4 months, it can be estimated that the final cost would be around 6,960.00 € net.

On the other hand, this project has been developed with my personal computer. It is a desktop computer that cost me about 1,000 €. Its components are outdated, but they have been more than enough for the realization of the project.

- **Motherboard:** ROG STRIX X470-I
- **CPU:** AMD Ryzen 7 2700 Eight-Core Processor 3.20 GHz
- **GPU:** NVIDIA GeForce GTX 1060 6GB
- **RAM:** 16GB DDR4

Moreover, all software used is freeware or has a free version. Whiles the Unreal Engine tutorials purchased on Udemy each cost around 15 €.

- **Unreal Engine 5:** the game engine used for the development of the project.
- **Trello:** a web page that allows to keep track of the tasks to be performed, being useful for the organization of the project [29].
- **GanttProject:** an open-source project management application that has been used for the realization of the gantt diagrams [22].
- **Google Docs:** an online word processor that allows you to create and format documents. Used for the initial project submits such as the technical proposal and the GDD.
- **Overleaf:** online LaTeX editor that has been used for the realization of this report [34].
- **Lucidchart:** application used to make the various diagrams shown in this report [32].

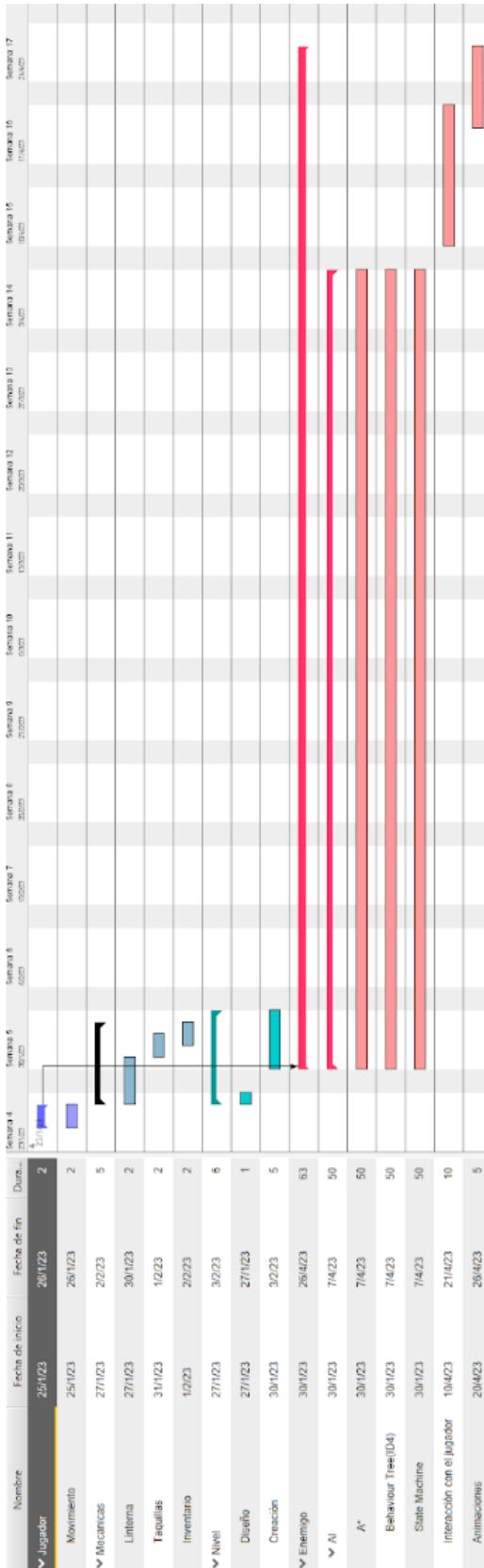


Figure 2.1: Gantt chart depicting the initial project planning

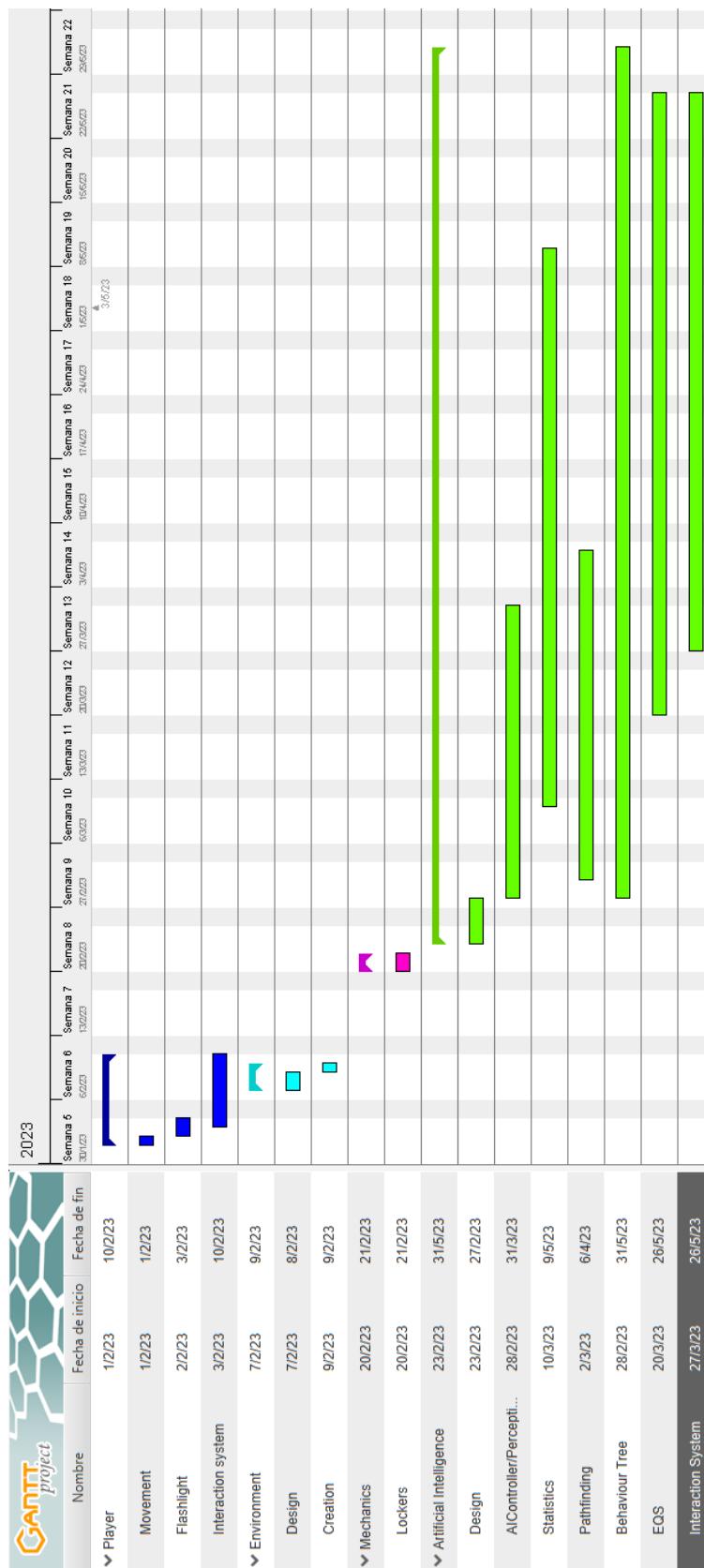


Figure 2.2: Gantt chart depicting the tasks performed in the project

Table 2.1: Total project cost

<b>Software Costs</b>	
Development software	0 €
<b>Hardware Costs</b>	
Computer	1,000.00 €
<b>Other Production Costs</b>	
Udemy courses	30.00 €
Salary	6,960.00 €
<b>Total</b>	<b>7,990.00 €</b>

C H A P T E R



## SYSTEM ANALYSIS AND DESIGN

### Contents

---

3.1	Requirement Analysis . . . . .	11
3.2	System Design . . . . .	13
3.3	System Architecture . . . . .	19
3.4	Interface Design . . . . .	20

---

This chapter presents the requirements analysis, design and architecture of the proposed work, as well as, where appropriate, its interface design.

### 3.1 Requirement Analysis

As I mentioned in previous sections, my goal with the artificial intelligence to be developed is to achieve a dynamic behavior that gives the feeling that the agent learns as the game progresses.

For this project, the player can move around the environment with the WASD keys. While moving, if the SHIFT key is kept pressed, the player will run. If the player runs for a long time in a row, he will be unable to run for a period of time. With the E key the player can interact with those objects in the game that he is allowed to interact with, such as doors or lockers. With the F key, the player can turn his flashlight on and off.

On the other hand, the agent who will take on the role of main enemy will go through three different states, which we will call initial, intermediate and final. In its initial state, the agent will move around the environment in an erratic manner and every time it sees the player, it will run away to hide. In the intermediate state it will be able to move around the game environment choosing which area and room it wants to move to, and if

it sees the player, it will start chasing him and when it reaches a certain distance from the player the agent will start stalking him in a circle. In its final state the agent, will move around the environment in a more efficient way and without retracing its steps, if it sees the player it will start to follow him and once it reaches him it will kill the player.

The agent has a series of statistics which will dictate which state the agent is in. These statistics are: Energy, Intelligence and Bloodlust. Depending on the value of certain of its statistics, its way of interacting with certain elements of the game will be different. Regardless of the state the agent is in and the value of their statistics, whenever it performs a displacement the energy of the agent will decrease by a certain amount. When its energy reaches 0, the agent will start searching for food, which will restore all the lost energy. In case the agent does not find food in a certain period of time, it will move to its initial point where it will stay until it recovers its energy.

### 3.1.1 Functional Requirements

A functional requirement defines a function of the system that is going to be developed. Let's take a look at the requirements of this project:

- **R1.** The player can move around the environment.
- **R2.** The player can interact with game elements by pressing the E key.
- **R3.** The player can use a flashlight by pressing the F key.
- **R4.** The player will be able to hide from the agent.
- **R5.** The player can sprint by pressing the SHIFT key.
- **R6.** The agent can move around the environment.
- **R7.** The agent can react to stimuli from the environment.
- **R8.** The agent will be able to interact with certain elements of the environment.
- **R9.** The agent will be able to chase the player.
- **R10.** The agent will be able to kill the player if it reaches him.
- **R11.** The agent will be able to go through three states.
- **R12.** Depending on the state in which the agent is, its way of interacting with certain elements of its environment will be different.
- **R13.** The agent will have a series of attributes and statistics that will define its current state.

### 3.1.2 Non-functional Requirements

A non-functional requirement is an attribute that dictates how a system operates. It makes applications of software run more efficiently and illustrates the system's quality [24].

- **R14.** Communication between scripts must be efficient through the use of interfaces.
- **R15.** Artificial intelligence will perform the expected actions at all times.
- **R16.** This artificial intelligence is compatible with Unreal Engine version 4.27 and future versions of Unreal Engine 5.
- **R17.** Certain aspects of this artificial intelligence are reusable in other types of video games.

## 3.2 System Design

This section presents the logical and operational design of the system to be carried out. The following pages define the use cases for the player and the artificial intelligence agent, as well as a case of use diagram for the agent (see Figure 3.1).

<b>Requirement:</b>	R1
<b>Actor:</b>	Player
<b>Description:</b>	Each time the player presses one of the movement keys, they can move around the environment by pressing W, A, S or D keys.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The player must not be inside a locker.</li> <li>2. The player must not be caught by the agent.</li> <li>3. The player must not be at the game over screen.</li> <li>4. The player must not be bumping into anything.</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The player press W, A, S or D keys.</li> <li>2. The player moves in the direction assigned to the key.</li> </ol>
<b>Alternative sequence:</b>	None

Table 3.1: Case of use «CU01. Movement»

<b>Requirement:</b>	R2
<b>Actor:</b>	Player
<b>Description:</b>	The player uses some of the objects scattered around the environment by pressing the E key.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The object must implement the interface BPI_Interaction.</li> <li>2. The player must be close to an interactive object.</li> <li>3. The player must be looking at an interactive object.</li> <li>4. The player must not be caught by the agent.</li> <li>5. The player must not be at the game over screen.</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The player presses the E key while looking closely at an interactive object.</li> <li>2. The object executes its internal logic.</li> </ol>
<b>Alternative sequence:</b>	None

Table 3.2: Case of use «CU02. Interaction»

<b>Requirement:</b>	R3
<b>Actor:</b>	Player
<b>Description:</b>	The player can use their flashlight by pressing F key.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The player must not be caught by the agent.</li> <li>2. The player must not be at the game over screen.</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The player presses the F key to activate their flashlight.</li> <li>2. The player presses the F key again while the flashlight is activated, the flashlight will be turned off.</li> </ol>
<b>Alternative sequence:</b>	None

Table 3.3: Case of use «CU03. Use flashlight»

<b>Requirement:</b>	R4
<b>Actor:</b>	Player
<b>Description:</b>	The player can hide in the lockers scattered around the environment to avoid detection by the agent.
<b>Preconditions:</b>	1. The player must be able to interact with the lockers (See CU02 in 3.2)
<b>Normal sequence:</b>	1. The player presses the E key to enter a locker. 2. The player presses the E key while inside the locker, he will get out of the locker.
<b>Alternative sequence:</b>	None

Table 3.4: Case of use «CU04. Hide»

<b>Requirement:</b>	R5
<b>Actor:</b>	Player
<b>Description:</b>	The player can increase his movement speed by pressing the SHIFT key.
<b>Preconditions:</b>	1. The player must be able to move (See CU01 in 3.2). 2. The player must be standing.
<b>Normal sequence:</b>	1. The player holds down the SHIFT key to start sprinting. 2. The player releases the SHIFT key to stop sprinting.
<b>Alternative sequence:</b>	None

Table 3.5: Case of use «CU05. Sprint»

<b>Requirement:</b>	R6
<b>Actor:</b>	Agent
<b>Description:</b>	The agent moves through the game environment
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The agent must have a behavior tree.</li> <li>2. The environment must be within the NavMesh volume.</li> <li>3. The agent must not be distracted.</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The agent's behavior tree is started at the beginning of the game.</li> <li>2. The agent decides the point to move to.</li> <li>3. The agent moves by tracing a path through the NavMesh.</li> </ol>
<b>Alternative sequence:</b>	None

Table 3.6: Case of use «CU06. Agent Movement»

<b>Requirement:</b>	R7
<b>Actor:</b>	Agent
<b>Description:</b>	The agent perceives different stimuli found in the game environment and reacts accordingly.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The agent must be able to move around the environment (See CU06 in 3.2).</li> <li>2. The agent must have the component AIPerception.</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The agent's behavior tree is started at the beginning of the game.</li> <li>2. The agent perceives a stimulus.</li> <li>3. The AIPerception modifies the values of the variables in the behavior tree.</li> <li>4. The flow of the behavior tree is affected according to the stimulus perceived by the agent.</li> </ol>
<b>Alternative sequence:</b>	2.1. In the case that the stimulus comes from an interacting object, it must implement the interface BPI_UsableActor.

Table 3.7: Case of use «CU07. Reaction to stimulus»

<b>Requirement:</b>	R10
<b>Actor:</b>	Agent
<b>Description:</b>	The agent is capable of killing the player if they get close enough to him.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The agent must be able to move around the environment (See CU06 in 3.2).</li> <li>2. The agent must be able to perceive stimuli (See CU07 3.2).</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The agent sees the player.</li> <li>2. The agent starts chasing the player.</li> <li>3. The agent reaches the player.</li> <li>4. The agent kills the player.</li> </ol>
<b>Alternative sequence:</b>	<ol style="list-style-type: none"> <li>2.1. If the agent is in its initial state, instead of chasing the player, it will run away from him.</li> <li>4.1. In case the agents does not have enough blood lust, instead of killing the player, it will start strafing him.</li> </ol>

Table 3.8: Case of use «CU08. Kill the player»

<b>Requirement:</b>	R11
<b>Actor:</b>	Agent
<b>Description:</b>	The agent will go through three states in which its way of reacting to the various stimuli in the environment will be different.
<b>Preconditions:</b>	<ol style="list-style-type: none"> <li>1. The agent must be able to move around the environment (See CU06 in 3.2).</li> <li>2. The agent must be able to perceive stimuli (See CU07 3.2).</li> </ol>
<b>Normal sequence:</b>	<ol style="list-style-type: none"> <li>1. The agent meets the requirements to change state.</li> <li>2. The agent moves to his initial room.</li> <li>3. The agent changes state.</li> </ol>
<b>Alternative sequence:</b>	None

Table 3.9: Case of use «CU09. Agent states»

Another system that deserves to be analyzed in depth is the one referred to in the use case CU09 (See CU09 in 3.2), the agent state machine<sup>1</sup> (see Figure 3.2). Let's look at

<sup>1</sup>A state machine is a behaviour model. Based on the current state and a given input the machine performs state transitions and produces outputs [27].

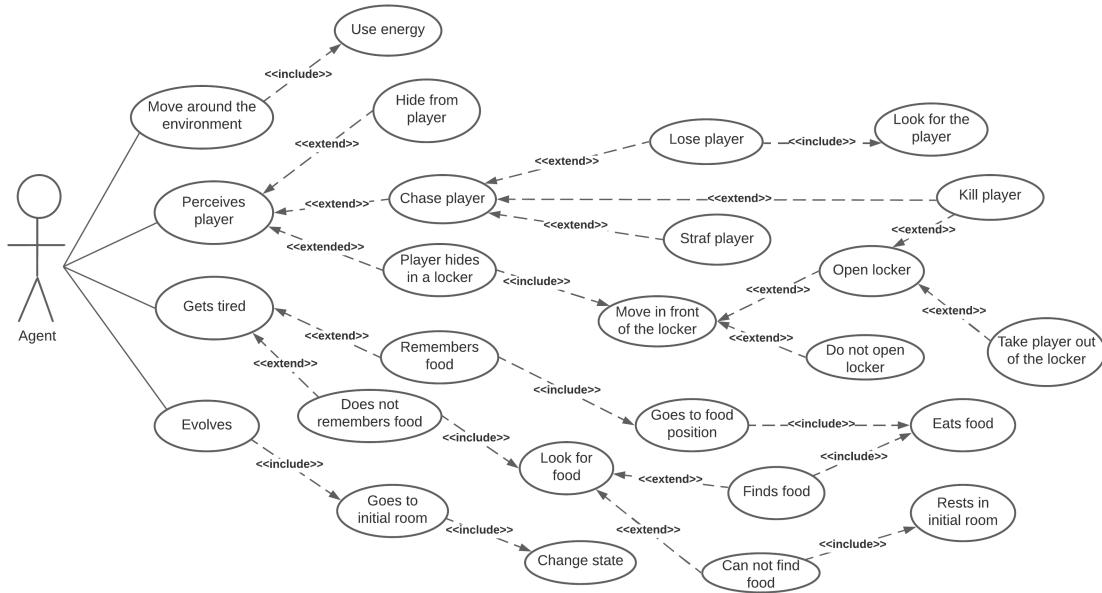


Figure 3.1: Agent's case use diagram

the triggers for each transition and how it affects the behavior of the artificial intelligence:

### Transition 1 trigger

The agent must reach the maximum value of its PTE (Points to evolve) statistic. This will increase the maximum value of all the agent's statistics by twenty-five.

## Transition 2 trigger

Once the agent is in the intermediate state, if it sees the player entering and exiting a locker (not necessarily the same one), the agent will understand how the lockers work and his way of interacting with them will change, allowing his intelligence statistic to improve.

### Transition 3 trigger

The agent must reach the maximum value of its PTE (Points to evolve) statistic. This will increase the maximum value of all the agent's statistics by twenty-five.

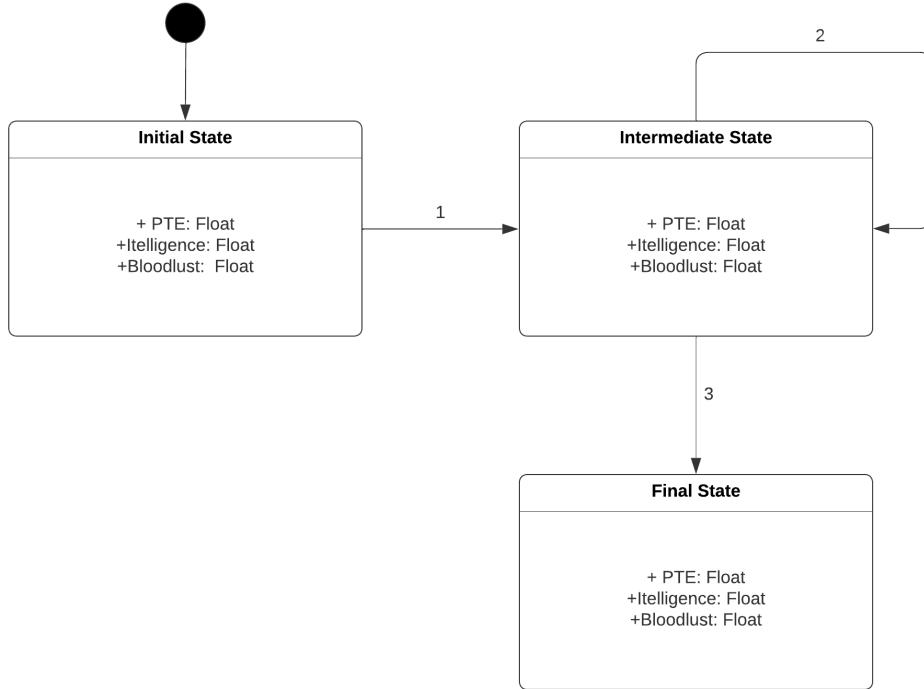


Figure 3.2: Agent's state machine diagram

### 3.3 System Architecture

This section describes the architecture of the projected system. This project has been developed in Unreal Engine 5.1. For running games made with this version of Unreal Engine the recommended system requirements are:

- **Operating System:** Windows 10 64-bit.
- **Processor:** Quad-core Intel or AMD, 2.5 GHz or faster.
- **Memory:** 8 GB RAM.
- **Graphics Card:** DirectX 11 or 12 compatible graphics card.

These are the requirements recommended by Epic Games for games developed in Unreal Engine 5 [16]. However, as this project only develops the artificial intelligence of what would be a horror video game, we can not take these requirements to the letter, since we do not have graphic section.

<b>Energy</b>	0	25	25.0
<b>Intelligence</b>	1	1	25.0
<b>Bloodlust</b>	1	1	25.0
<b>PTE</b>	0	0	25.0

Figure 3.3: Debugging interface

### 3.4 Interface Design

Since this project is about design and implementation of artificial intelligence and not about developing a video game as a whole, it has not been necessary to develop a player interface. However, for a better development of the project a very simple interface has been created to have a constant knowledge of the value of the agent's statistics (see Figure 3.3). This way the developer will be able to have a better control of the agent and the debugging task will be easier.

CHAPTER



## WORK DEVELOPMENT AND RESULTS

### Contents

---

4.1	Work Development	21
4.2	Results	74

---

This chapter shows in depth the creation of the various behaviors and actions that the agent can perform, as well as the design process. It also illustrates some of the problems that the project has gone through and how they have been solved.

### 4.1 Work Development

The work developed is going to be explained in chronological order, this way, the process and certain changes, additions and problems that the project has gone through can be fully appreciated.

#### 4.1.1 First steps: Creating the classic behavior

The first objective we were looking to accomplish was the bread and butter of artificial intelligence: an agent that patrols the environment and when it detects the player starts to chase him. If it catches the player, it goes to the game over screen, and if the player manages to escape from the agent, it will start looking for him for a while, if it finds him again it will start chasing him back and if not, it will return to the patrol. We start by adding a new character blueprint to be our agent and another one of type AIModule for the AIController. A Controller is a non-physical actor that can possess a character to control its actions. Controllers receive notifications for many of the events occurring for the actor they are controlling, giving the opportunity to implement or suspending a

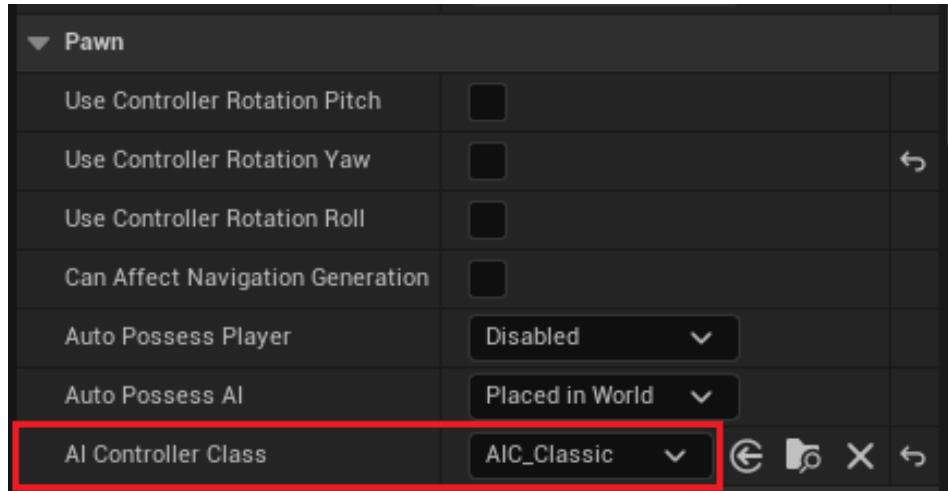


Figure 4.1: Assign AIController to the agent

behaviour in response to the events [11]. More specifically, the AIController is focused on responding to inputs from the environment and game world. Their work is to observe the world around it and make decisions and react accordingly without explicit input from a human player [7]. To assign the controller to our agent it is enough to assign it from its details in the Pawn section (see Figure 4.1).

Now that we have created our agent, it is time to make it capable of perceiving the environment. Unreal Engine allows us to capacitate our agents with senses very easily, for this we must add the AIPerception component in our AIController. The AIPerception component allows us to define which senses (see Figure 4.2) our agent will have and the way it will respond to them [8]. At the moment, for our current objective we are interested in our agent being capable to seeing the game level and perceive the player, so with this new component we will provide our agent with sight. Once it has a sense, we can configure it to suit our needs. The parameters (see Figure 4.3) we are interested in are: Sight Radius to define the area of vision of our agent, Lose Sight Radius to define the area in which the perceived actors will no longer be seen by the agent and PeripheralVisionHalfAngleDegrees to adjust the angle that the peripheral vision of our agent will have. Next, we must specify what our agent's thoughts should be when it sees the player. When we talk about thoughts, we refer to the type of information received through the senses and how it will affect the agent's decisions. For example, when the agent perceives the player, this stimulus will cause the agent to stop patrolling and start chasing the player. Therefore, this stimulus must be able to communicate to the agent's behaviour tree that it must change its flow. In order to accomplish this, we will need to create a blackboard. A blackboard is a memory space where we can store variables which we can read and write to alter the flow of our behaviour tree. We will be able to access the blackboard from our AIController, modifying its information according to the stimulus received by the agent. First of all, let's create the variables that will be necessary for our first objective:

## BLACKBOARD

- **targetLocation**

Vector variable that will store the exact point in the world where the AI is heading.

- **canSeePlayer**

Boolean variable to know if the agent has spotted the player.

- **isPursuingPlayer**

Boolean variable to know if the agent is chasing the player.

- **canAttackPlayer**

Boolean variable to know if the agent is within range to attack the player.

- **inPlayerRange**

Boolean variable to know if the player is in range to the agent.

With the variables created, we must now program the logic so that the AIController of our agent modifies its values according to the stimuli it receives from the environment. Through the AIPerception component we can implement the On Target Perception Updated node (see Figure 4.4), which will be executed every time an actor enters the field of view of our agent or leaves it. As seen in the node, each time it is executed it receives two parameters, the actor that has entered the field of vision and the stimulus, which is a data structure with certain information about the perceived actor. Since for now the only actor that is in the environment is our player, we can leave for later the verification of which actor the agent has perceived, but we will have to check if the actor enters or leaves the field of vision. One of the parameters contained in the stimulus data structure is a boolean called Successfully Sensed, which will return true if the actor is visible and false otherwise. Whether the actor is visible or not to our agent, we must access the blackboard which is accessible from the AIController through the Get Blackboard node, and from there we can update the values of the variables we want. For our current goal we will need to modify the variables canSeePlayer and isPursuingPlayer (see Figure 4.5). Its value will be true when the player enters the agent's field of vision, and only the variable canSeePlayer will become false when the player is out of sight, since when the agent loses sight of the player, it will stop following him, but the agent will start to investigate the last place where it has seen the player.

Our agent is able to perceive the environment, now we must ensure that it can move through it. To give our agent the ability to move we need to implement a pathfinding algorithm. As Xiao and Hao explain very well in their paper [4], pathfinding in modern computer games generally refers to find the shortest route between two end points. This definition can address to a wide range of problems such as characters sent on missions from their current location to a predetermined or player determined destination, how to avoid obstacles cleverly and seek out the most efficient path over different terrain. Using the tools offered by Unreal Engine, we will implement a pathfinding with NavMesh, a

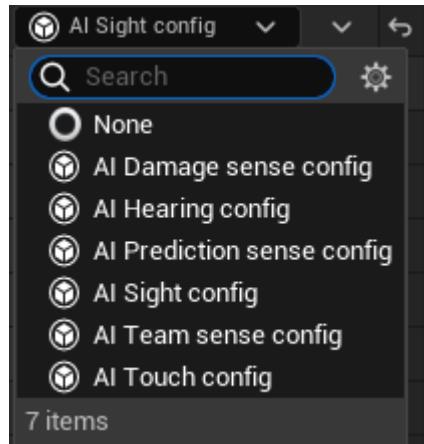


Figure 4.2: Senses available through the AIPerception component

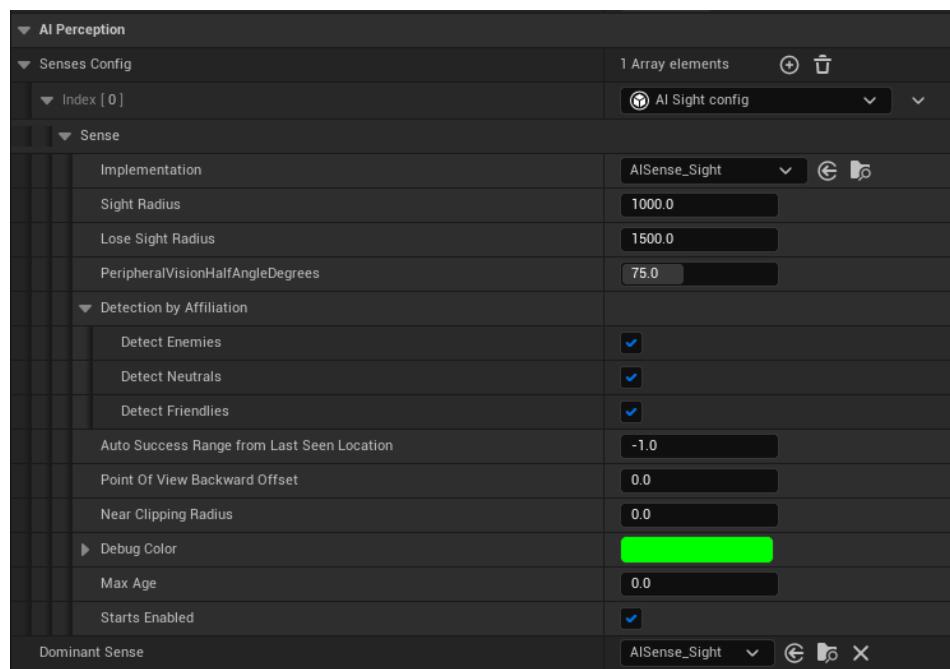


Figure 4.3: Sense of sight setting options



Figure 4.4: On Target Perception Updated node

popular technique for pathfinding in 3D worlds. A NavMesh is a set of convex polygons that describe the “walkable” surface of a 3D environment (see Figure 4.6). It is a simple, highly intuitive floor plan that AI characters can use for navigation and pathfinding in the game world. To add a NavMesh to our environment we will add to our game a component of class NavMeshBoundsVolume, which will create an area where all ground inside it will become a “walkable” surface for our agent.

Once this is done, we must create what will be our behaviour tree. According to the definition given by Yoones [37], a Behaviour Tree (BT) is a directed tree including a set of nodes and edges. The root of a BT is a node without parents. On the other hand, nodes without children are the leaves of this tree. In a basic BT, a non-leaf node can be a selector node or a sequence node. Selectors are used when we aim to find and execute the first possible child that can run without failure. A selector node succeeds once any child is performed successfully. In a sequence node are evaluated sequentially. A sequence node succeeds only if all children are performed successfully. On the other hand, a leaf node can be an action or a condition. Action nodes include playing animation, changing the state of a character, or any activity that changes the state of the game. Still, a condition node is generally used to test some values. Test for proximity, testing the state of a character and testing the line of sight are examples of condition nodes. A condition returns success if the condition is met; otherwise, returns failure. In order for our agent to receive its behaviors from the tree, we will need to assign it in its AIController through the "Run Behaviour Tree" node through the BeginPlay function, which will execute all the code found in it once the game is started (see Figure 4.7).

Now that our agent has a behavior tree it is time to start working on the tasks, decorators and services that it will need for our current objective.

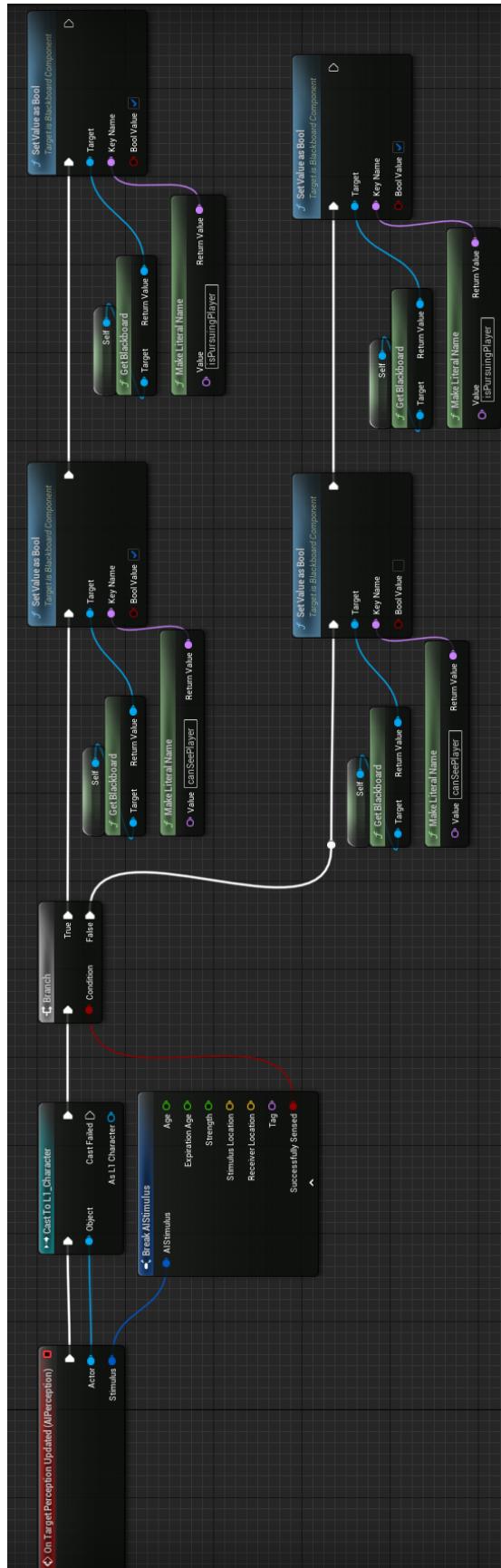


Figure 4.5: Logic behind the player entering and leaving the agent's field of view

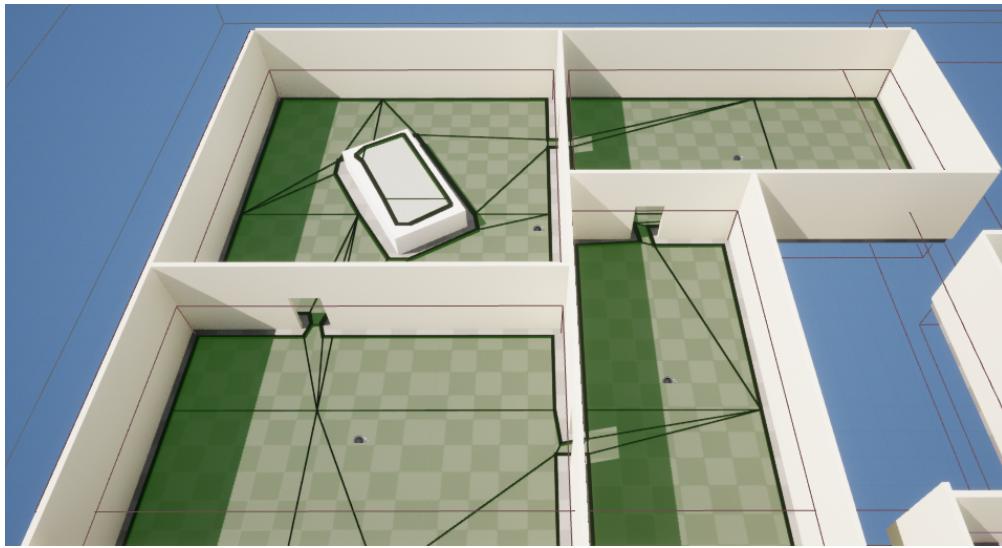


Figure 4.6: Surface where the agent can walk

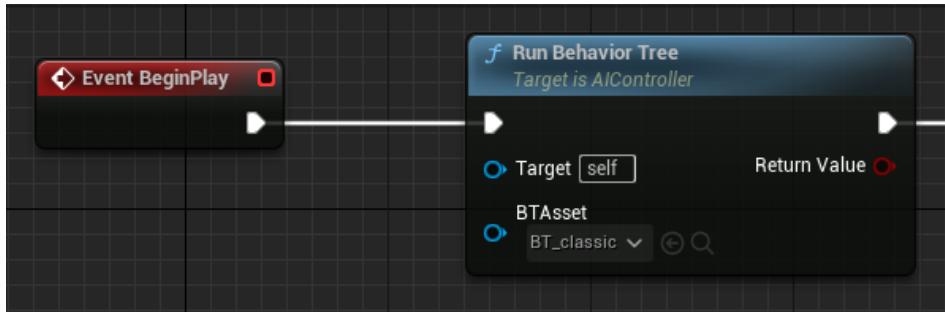


Figure 4.7: Specify BT to be used in the AIController

## TASKS

Tasks nodes are what we commonly refer to as leaf nodes. Unreal Engine also has its own tasks that we will use a lot of times such as `moveTo` which moves the agent to a given position, and `Wait` which will make the agent stop for a certain amount of time.

- **BTT\_SetSpeed**

Allows modification of the agent's speed. This way, for different situations the agent can go at different speeds to give a greater sense of realism.

- **BTT\_RandomMovementNavMesh**

The agent will move to a random point on the navmesh within a certain radius.

- **BTT\_FindAndChasePlayer**

The agent moves to the player's position

- **BTT\_LastPlayerLocation**

The agent moves to the last point in the world where it has seen the player when it loses sight of the player.

- **BTT\_SearchForPlayer**

Taking as center the last point where the agent has seen the player, within a given radius, the agent searches for the player.

- **BTT\_SetIsPursuing**

Modify the value of the isPursuingPlayer variable so that the agent does not remain in the pursuing state once it has lost the player and cannot find it.

- **BTT\_KillPlayer**

The agent kills the player.

## SERVICES

Special nodes that can be attached to composed and task nodes and can be executed at their own frequency as long as the branch they are on is running. They are mostly used to update blackboard values.

- **BTS\_IsInRangeToPlayer**

Compares the distance between the agent and the player, if the distance is less than or equal to a given distance, the value of the canAttackPlayer variable will be modified allowing the agent to attack the player.

## DECORATORS

Conditions attached to other nodes. There are several types of decorators such as getting a value from the blackboard, loops, conditional loops, etc. They are used to control the tree flow.

- **BTD\_isPlayerVisible?**

Checks the value of the canSeePlayer variable.

- **BTD\_iAmPursuingThePlayer?**

Checks the value of the isPursuingPlayer variable.

- **BTD\_isPlayerInRange?**

Checks the value of the isPursuingPlayer variable.

- **BTD\_canIAttackThePlayer?**

Check the value of the canAttackPlayer variable.

Now that we have everything we need for this first version of our agent's tree, let's build it and look at its control flow. As long as the agent is not chasing the player, it will be patrolling the level, the left sub tree of the root will consist of a sequence node with the BTD\_isPlayerVisible? decorator. While patrolling, it will move at a fixed speed through random points of the level and before moving from point to point the agent will wait three seconds. Thanks to the flow control of the decorator nodes, we can modify the flow of the tree if the value or result of the variable being observed in the node changes. In this case, when the value of the variable isPursuingPlayer changes when the agent detects the player through its sight, the decorator will abort the execution of any node that is in its sub tree (see Figure 4.8) and the flow will pass to the right sub tree of the root node. Now, while the agent sees the player, it will start to follow him adjusting his speed to a higher one than patrol speed and moving towards him thanks to the BTT\_FindAndChasePlayer task, which stores the player's position in the targetLocation variable of the blackboard and through the Simple Move to Location node, the agent will perform the displacement (see Figure 4.9). Through the service node BTS\_IsInRangeToPlayer, it will check if the player is within its attack range, if so, it will change the speed of the agent to 0 to stop moving and through the BTT\_KillPlayer task which will execute a series of functions that will rotate the player towards the agent, block its controls and shows the Game Over screen. Finally, if the agent loses sight of the player while pursuing it, the agent will move to the last point where it has seen the player by running the BTT\_LastPlayerLocation node, which simply stores in targetLocation the last point where it has seen the player. To get the feeling that the agent is investigating the area we have used a sequence with a Loop decorator, which will perform the execution of the sequence node a certain number of times, for this implementation it will run a total of three times. Each time the sequence node is executed, the agent will choose a random point within a given radius where the center is the last point where the player was seen with the BTT\_SearchForPlayer node (see Figure 4.10), move to it and wait between two and four seconds. If the agent encounters the player while investigating, thanks to the BTD\_isPlayerVisible? decorator, the flow of the tree will be modified by the agent's sight, allowing it to start chasing the player instantly. In case the agent did not find the player, the flow would exit the loop and execute the BTT\_SetIsPursuing node, which changes the isPursuingPlayer boolean to false so that the agent returns to its patrolling state. With this we have obtained an initial behavior tree for the agent (see Figure 4.11).

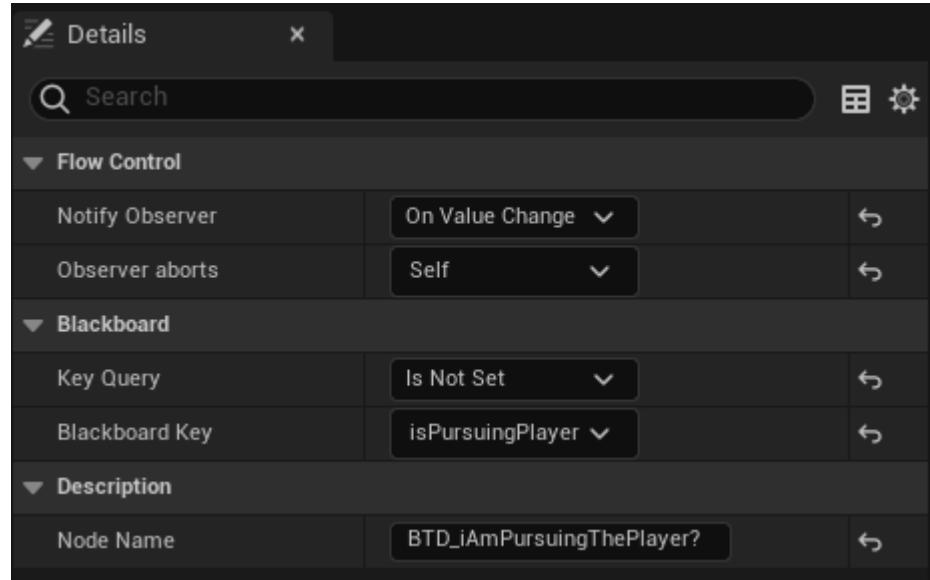


Figure 4.8: BTD\_iAmPursuingThePlayer? details

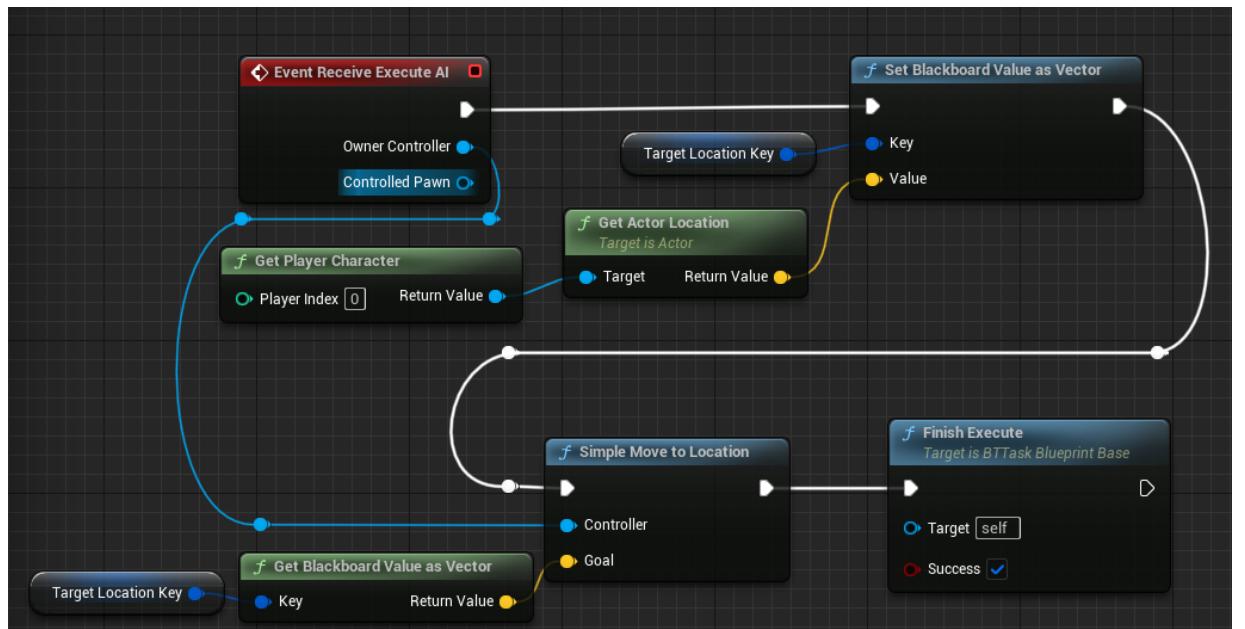


Figure 4.9: BTT\_FindAndChasePlayer

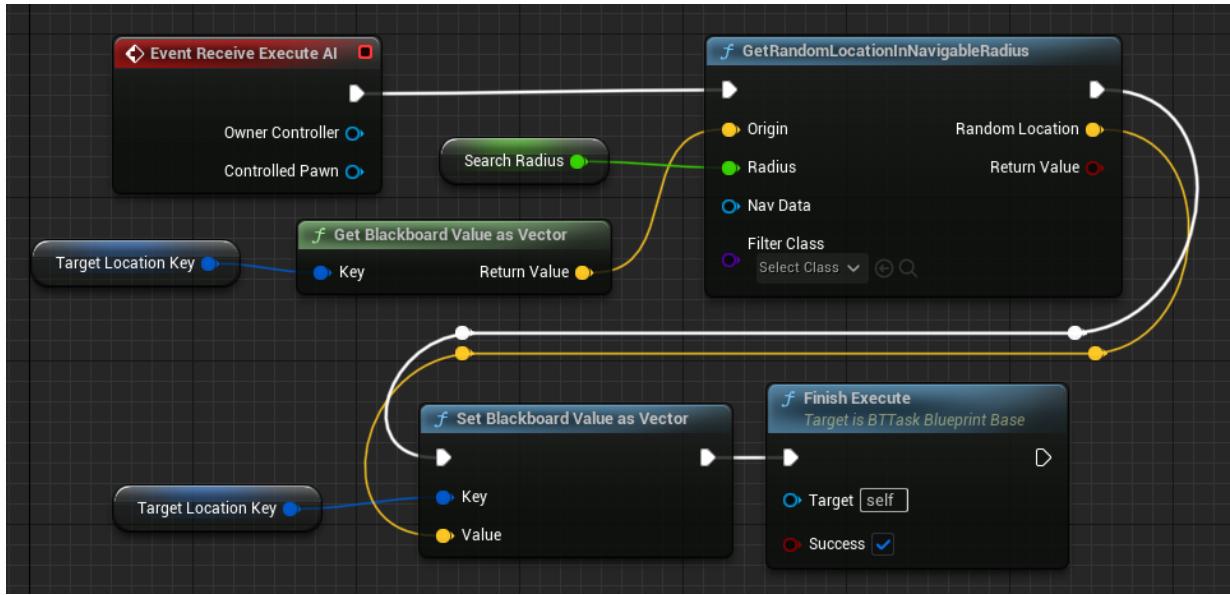


Figure 4.10: BTT\_SearchForPlayer

#### 4.1.2 Adding the lockers

Now that we have an agent capable of moving through an environment and interacting with the player, we are going to start implementing some systems more related to a horror video game. The most famous and iconic that have the vast majority of horror games nowadays are the actors where the player can interact with them to hide from the enemy, and are usually lockers, closets or beds inside the games (see Figure 4.12). So, the next goal to achieve in the project is to add the actors where the player can hide and the agent can interact with them if it sees the player entering one. We will not go into detail about the full implementation of these actors, such as the animation for opening the locker door or the player's movement inside the locker, as these are non-AI related parts of the project.

As we are adding new functionalities to our agent, we will need to add more variables to our blackboard as well as create new task nodes and decoratos for our behaviour tree:

#### BLACKBOARD

- **targetRotation**

Stores the rotation vector of an actor.

- **killHiddenPlayer**

Boolean variable to know if the agent is going to kill the player while he is hiding.

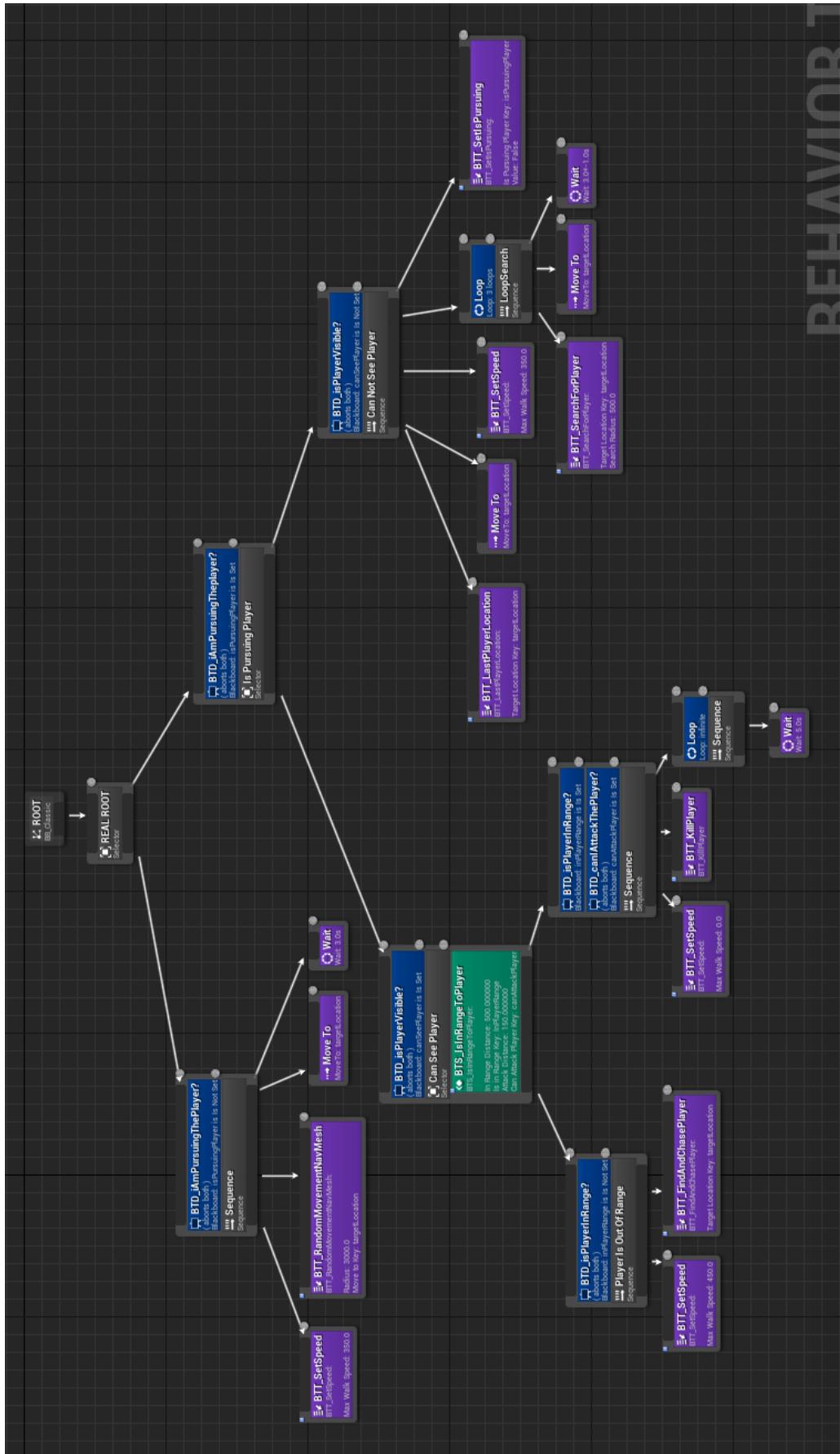


Figure 4.11: First version of the agent's behaviour tree

## TASKS

- **BTT\_RotateAI**

The agent performs a rotation until it is equal to the rotation stored in the variable targetRotation.

- **BTT\_PullOutOfHiding**

The agent opens the locker where the player is hidden.

## DECORATORS

- **BTD\_goingToKillHiddenPlayer?**

Checks the value of the killHiddenPlayer variable.

To get the agent to lose sight of the player when hiding in a locker is a simple task. It is enough for the player to enter inside the locker and close the door, since it will block the agent's vision modifying the canSeePlayer variable. But here we find the first problem that this project has gone through. If the agent is following the player and he hides in a locker, yes, the agent will move to the last point where it has seen the player and start investigating in a radius. The problem is that if the agent is aligned with the player as he opens the door and hides, before the door closes the point in the world that will be stored as the player's last location will be inside the locker because the agent has time to see the player before the door closing animation ends, and when it tries to go to that point to start the search for the player will be impossible. A first idea was to make the player invisible when interacting with the locker, but this solution gave rise to another problem where if the behaviour tree was executing the BTT\_FindAndChasePlayer node just at the moment when the player was made invisible, it would paralyze the agent because it was trying to store the position of an actor that it was not perceiving. Doing some more research while looking for a solution to this issue we saw that it was possible to specify to the agent that it does not need to go to the exact point it was trying to reach. With the moveTo task node provided by Unreal Engine, it is possible to specify an acceptable radius (see Figure 4.13) where the agent can move to consider the goal reached, this way by specifying a radius greater than the five units that come by default in said attribute of the moveTo node, we can get our agent to stop in front of the locker. But this is not the result we want to get to, so now we need to get the agent to approach the locker if it sees the player hiding, open the locker and kill the player. To get the agent to interact with the locker we are going to implement a blueprint interface. A blueprint interface is a collection of one or more functions that can be added to other blueprints. Any blueprint that has the Interface added is guaranteed to have those functions. The functions of the Interface can be given functionality in each of the blueprints that added it. This is essentially like the concept of an interface in general programming, which allows multiple different types of Objects to all share and be accessed through a common interface [10]. We will create an interface called BPI\_PlayerHiding with two functions, DidEnemySee and LeftHiddingSpot and add it to our agent controller. Each time the player interacts

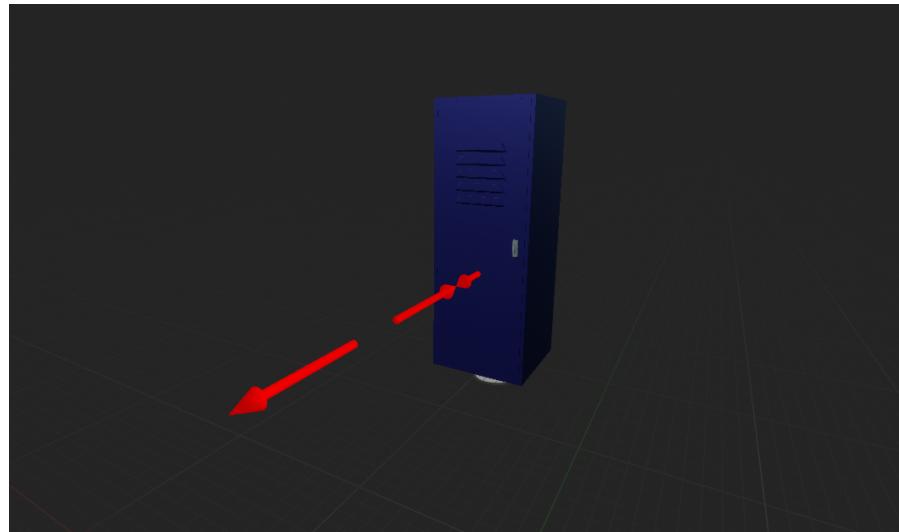


Figure 4.12: Lockers used in the project

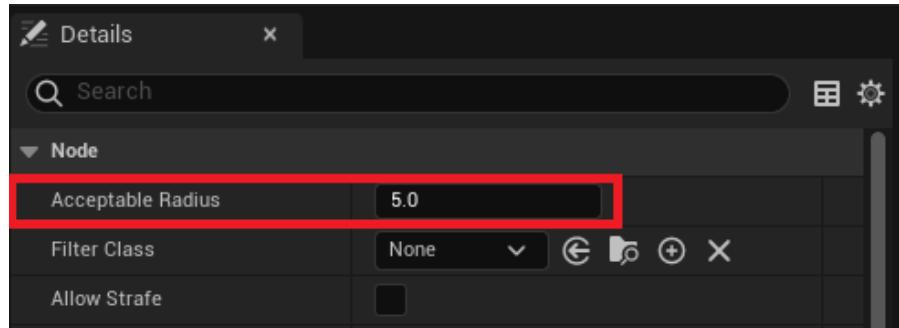


Figure 4.13: Acceptable Radius parameter

with a locker, the DidEnemySee function inherited by the agent's controller from the interface will be called. This function will check if the agent is watching the player by reading the value of the canSeePlayer variable of the blackboard, if the value is true, it will change the value of the variable killHiddenPlayer to true, store a point in front of the locker door in targetLocation and the vector rotation of the locker so that the agent can look towards it in the variable targetRotation (see Figure 4.14). It may be the case that while the agent is on his way to the locker, the player decides to come out of hiding. In this case, we will follow a similar logic to the one we have just seen. When the player interacts with the locker while hiding inside it, the agent's LeftHiddingSpot function will be called. This function will simply modify the value of the killHiddenPlayer variable to false (see Figure 4.15), causing the agent to target the player again.

Straightaway, we are going to make the necessary modifications to our behavior tree. For now the agent will only go to a locker when it sees the player interacting with one, and if it is seeing the player, it is chasing him. So the modifications must

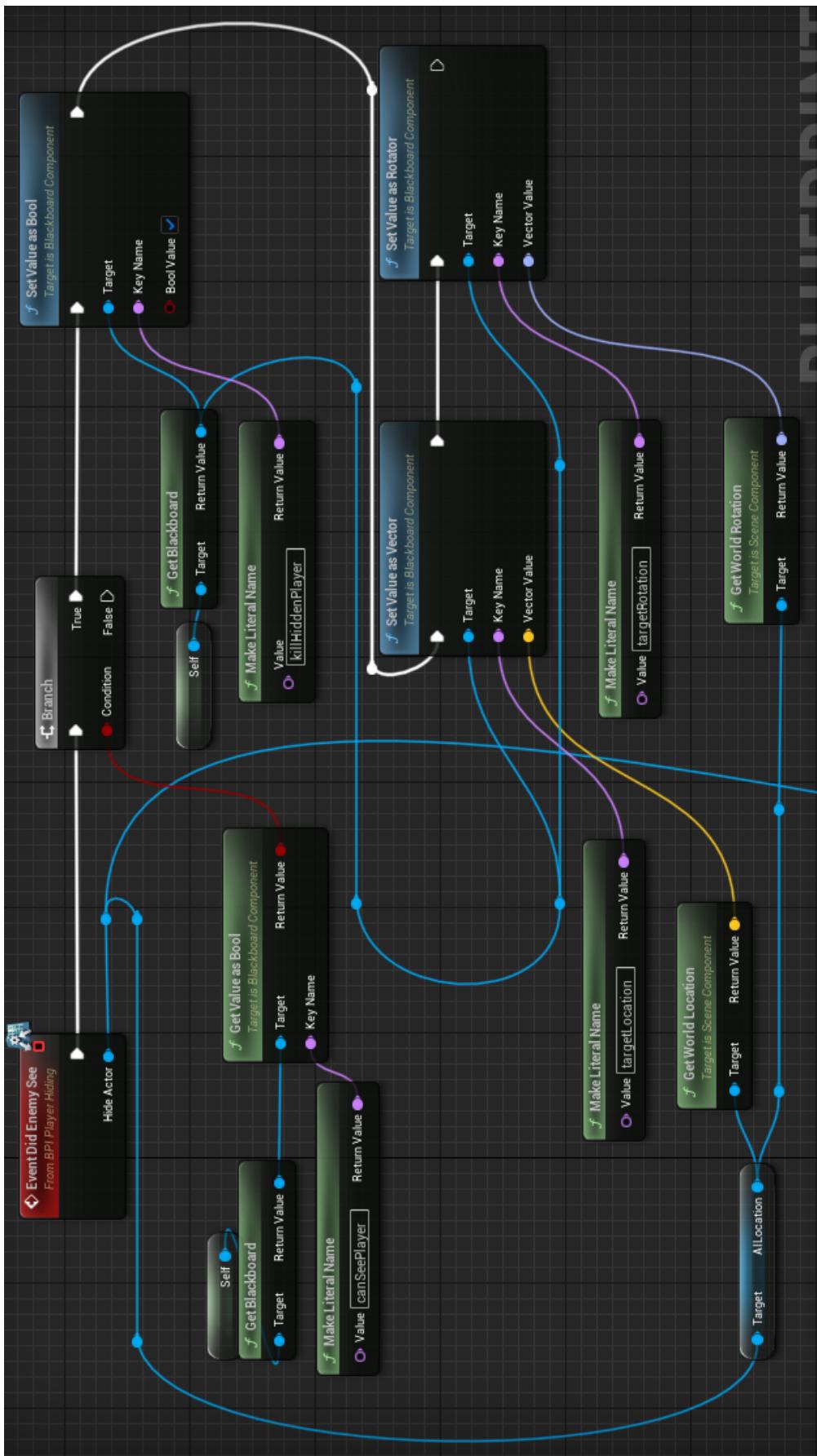


Figure 4.14: Event DidEnemySee

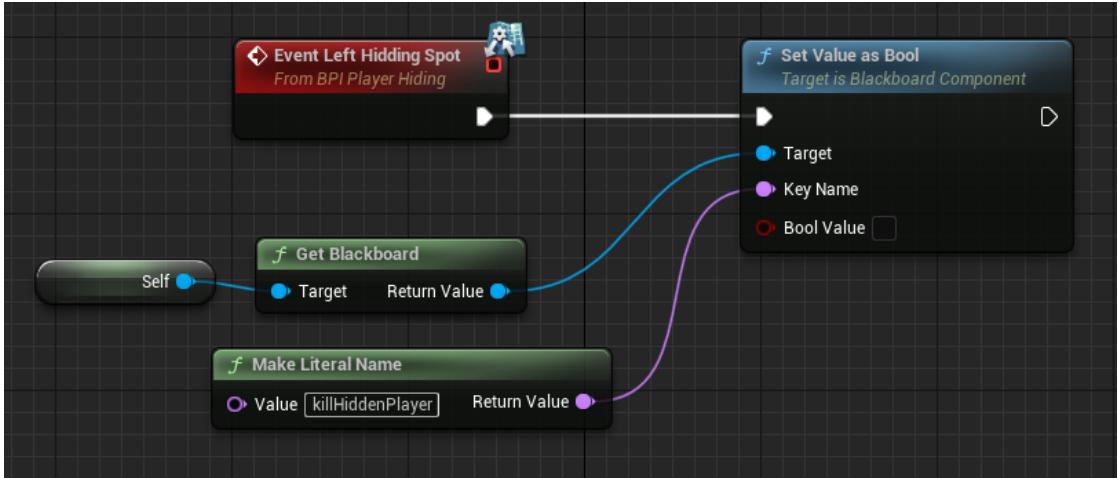


Figure 4.15: Event LeftHiddingSpot

be made in the sub tree in charge of managing the agent's decisions when it is chasing the player. The placement for the new sub tree is very important, when the decorator BTD\_isPlayerVisible? is aborted by the change of value of the variable canSeePlayer because the player leaves the field of view of the agent, the flow will be modified by passing to the next sub tree. As soon as the agent loses sight of the player, we want the BTD\_goingToKillHiddenPlayer? decorator to be checked first, so if the player has not hidden, the flow will go to the sub tree for the agent to start investigating. If the opposite were done, and the decorator BTD\_isPlayerVisible? is checked first, the agent would always investigate the last point where it saw the player even if the player has hidden in a locker, because the flow would enter first in the sub tree of investigation and not in the sub tree of going to kill the hidden player. As a result, the new sub tree (see Figure 4.16) should go between the pursuit sub tree and the investigation sub tree.

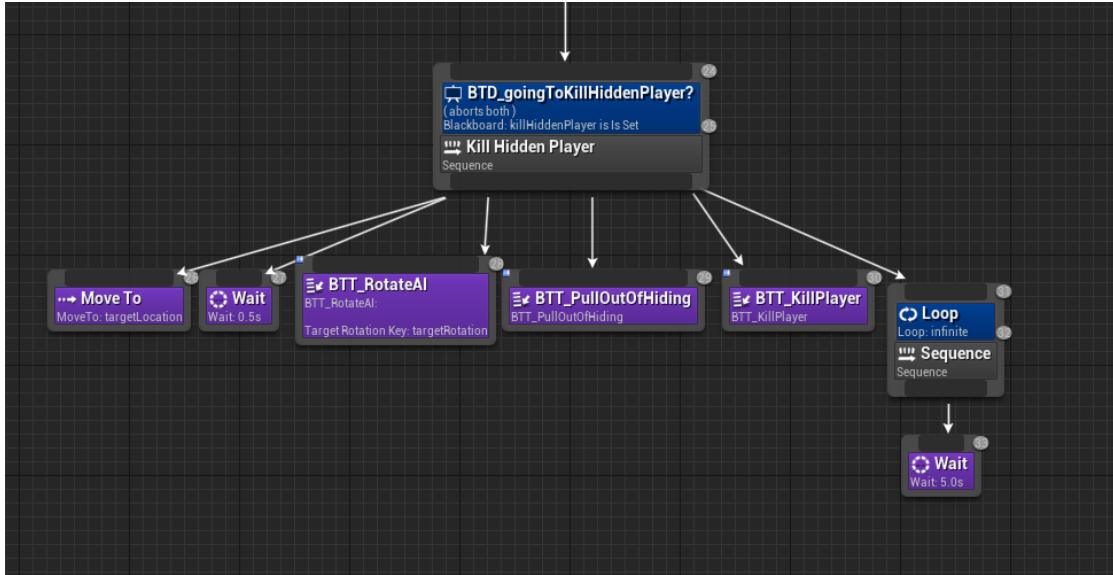


Figure 4.16: Kill Hidden Player sub tree

#### 4.1.3 Hierarchical pathfinding

Another important element in horror video games is the design of the environment. If the agent is going to chase the player, we must ensure that the level can be traversed in a loop path to prevent the player from reaching a dead end. For this reason we spent a bit more time designing the environment (see Figure 4.17), and while working on this, we realized that we needed the agent not to move around a random point in the NavMesh but to be able to go from room to room to give a greater sense of patrolling.

The vast majority of environments in horror video games are divided into zones, each of which is composed of rooms, being the hierarchical pathfinding a good option for this type of design. This technique is highly recommended based on its efficiency and flexibility to handle various types of maps [5]. Allows dividing the level into various sections where each for these will be represented by a node of the graph. Each section node stores which are its neighboring nodes (or in other words, the rooms adjacent to it or which can be accessed from it) and the nodes that can be found within the room itself. Therefore we are going to take advantage of the hierarchical pathfinding and we will divide the environment into zones, for this we will create an empty object called BP\_ZoneNode that will take the role of node, and each room will have its own node which will also be an empty object that we will call BP\_RoomNode. To delimit which rooms will be part of each zone, we will add a box collider to our node and modify its dimensions according to our needs, and using the Get Overlapping Actors node we will search for the actors of the BP\_RoomNode class and store in an array (see Figure 4.18). Figure 4.19 shows the different zones and the connections between the room nodes.

With this node hierarchy we can now instruct our agent to move between rooms instead of traveling between random points. Like the patrol that is currently implemented,

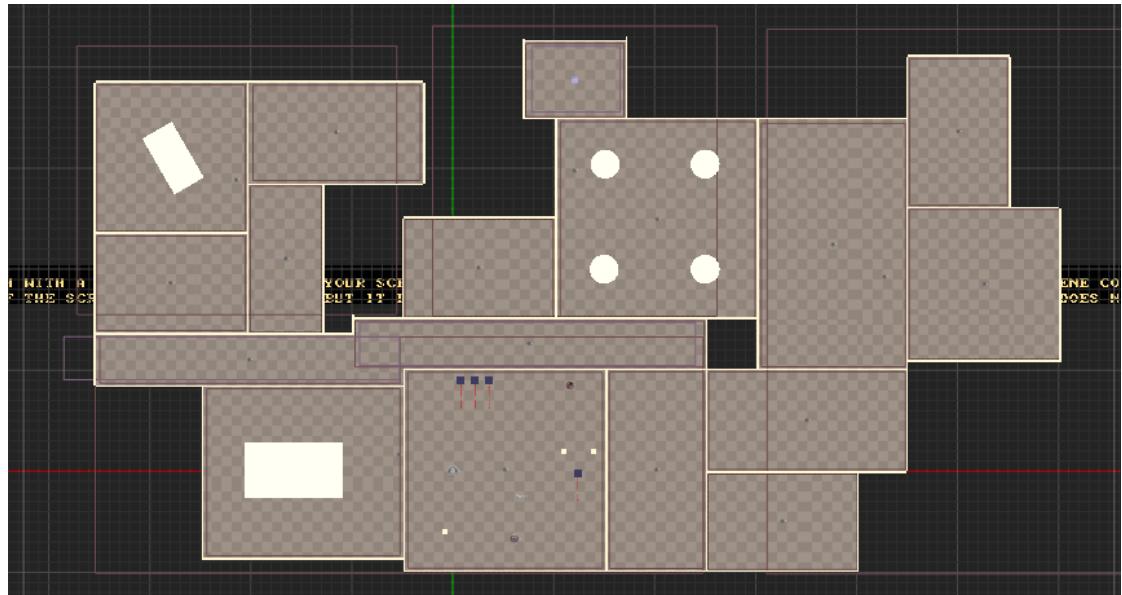


Figure 4.17: Design of the test environment

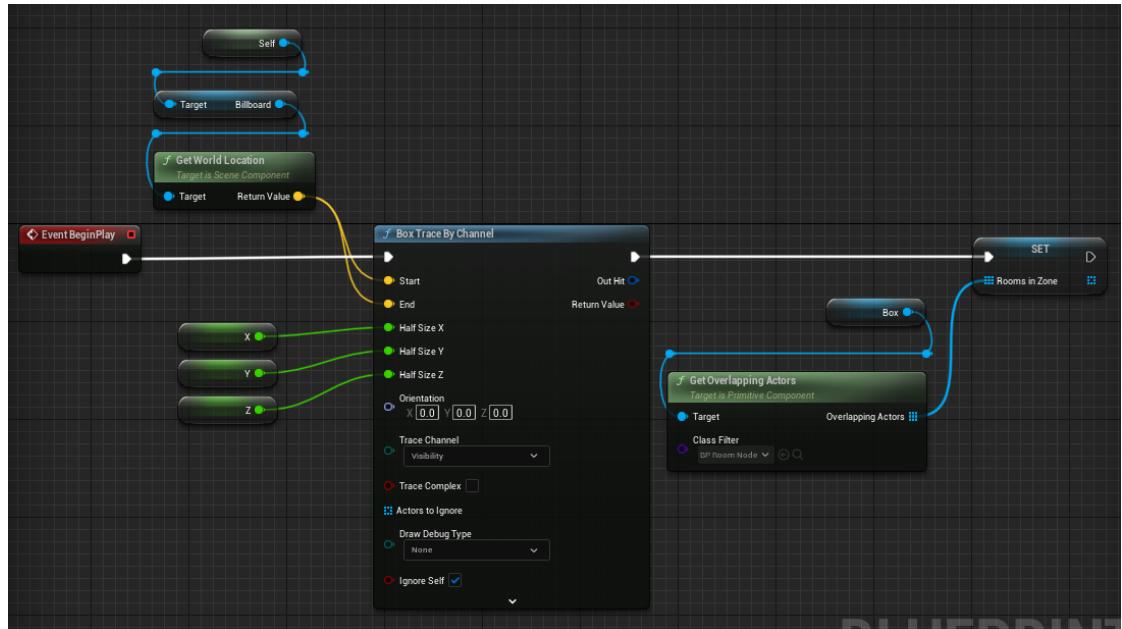


Figure 4.18: BP\_ZoneNode BeginPlay event

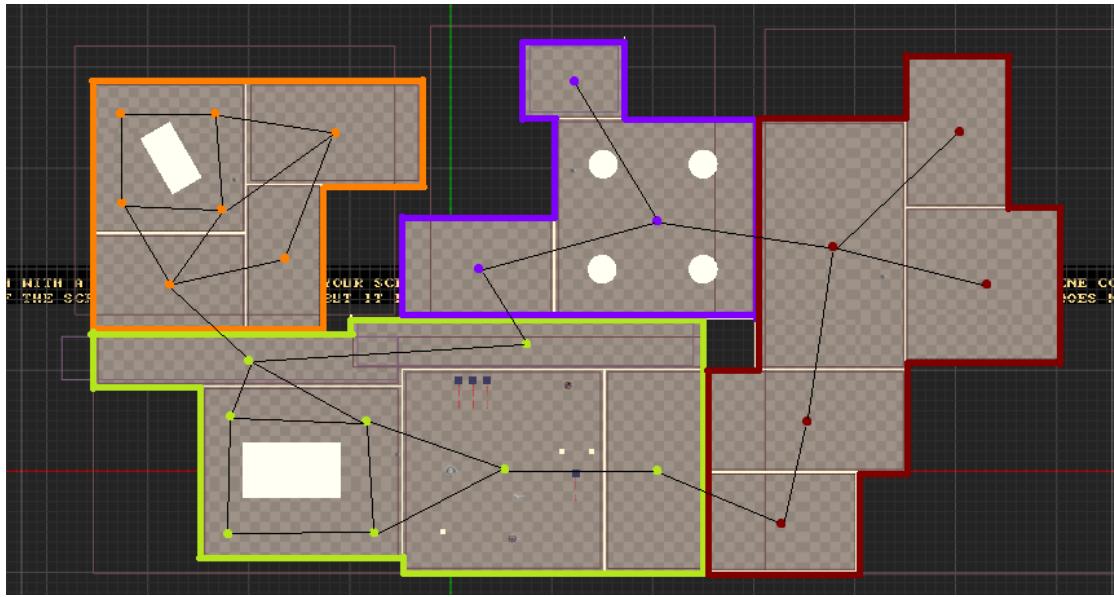


Figure 4.19: Representation of the graf created by the nodes of the rooms

we will create a sub tree that will be in charge of executing the decisions of our agent when deciding which zone and room it should move to. Let's see first what variables for the blackboard and tasks we will need.

## BLACKBOARD

- **desiredZone**

Object type variable where the node of the zone to which the agent is heading will be stored.

- **objectiveRoom**

Object type variable where the node of the room to which the agent is heading will be stored.

## TASKS

- **BTT\_DesiredZone**

When this task is executed, a random zone node will be chosen from all available ones and stored in the desiredZone variable.

- **BTT\_FindARoom**

It reads the value of the desiredZones variable from the blackboard, accesses the array where all the room nodes contained in the zone are stored and chooses one at random. It cannot choose the same room twice in a row.

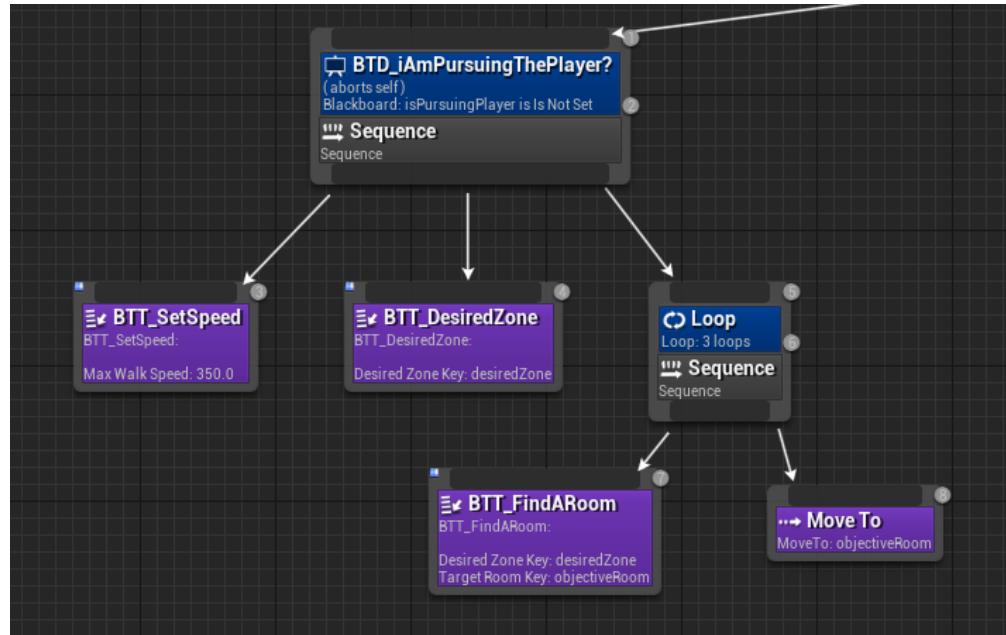


Figure 4.20: Hierarchical Patrol sub tree

We will start the new sub tree with the BTT\_SetSpeed task and then the BTT\_DesiredZone node will be executed. Then, the flow will follow to another sequence node with a loop type decorator with 3 iterations where the nodes BTT\_FindARoom and MoveTo are executed. This way the agent will go to a zone, move a total of three times between rooms in that zone, and once it has moved three times, he will return to the beginning of the sub tree and choose another zone node. Figure 4.20 shows the obtained sub tree.

#### 4.1.4 Working in more detail on the design

As of now, we have a basic artificial intelligence with a behavior tree, capable of interacting with elements of the environment and that moves through the level thanks to Unreal Engine's NavMesh using hierarchical pathfinding concepts. With this base, we are going to start working more in depth on its design. The agent's idea for this project consists of a science fiction monster resembling an animal that is adapting to the environment and the stimulus it receives from it. It will go through three progressive states: initial, intermediate and final. Each state represents a level of challenge for the player and skills for the agent, creating a graded game experience.

- **Initial State:** The agent will move through the level by choosing a random point on the navmesh. Whenever it sees the player, the agent will run away and hide in a point not visible to the player near it. The agent will not be able to interact with the lockers.

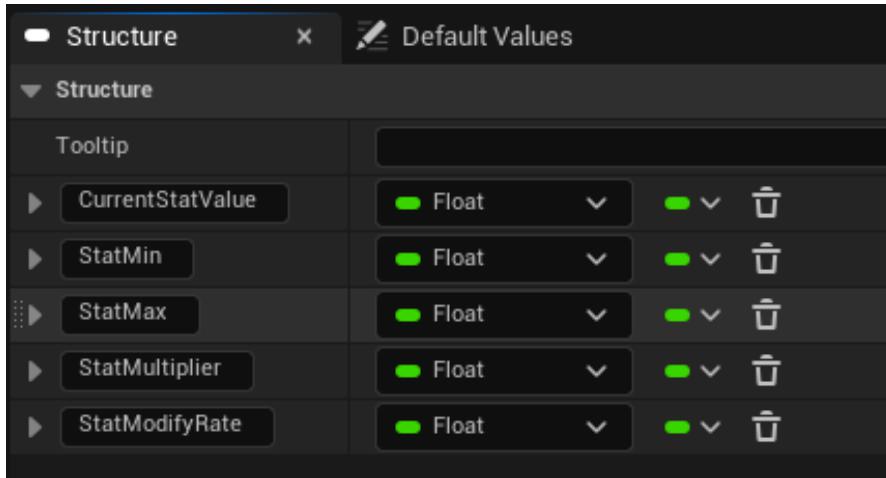


Figure 4.21: Stat Struct

- **Intermediate State:** The agent will patrol the level using the node hierarchy. When it detects the player it will start chasing him, but if it reaches the player, instead of killing the player it will start stalking him in circles until the player runs away. When the agent sees the player hiding in a locker, it will go in front of it and stand in front of the locker's door for a seconds. After several factors that we will see later, the agent's response to this stimulus will change and instead of standing in front of the locker for a few seconds, the agent will open the locker and take the player out of it.
- **Final State:** The agent will patrol the level using the Unreal Engine's Environment Query System. It will chase the player if it detects him and if it catches him it will kill the player. When the agent sees the player hiding in a locker, the agent will go to the locker to open it and kill the player.

In order to give the agent the ability to evolve and change its state, we will implement a series of data structures where we will store a series of values that will take the role of statistics of our agent. First of all, we add a custom struct where we can store the different values of the agent statistics and access them (see Figure 4.21). A struct is a collection of different types of data that are related and held together for easy access. In addition, Unreal Engine allows us to create our own custom structs with different variables and values [17]. Thanks to this, we will be able to specify and store the different characteristics that each statistic will have, such as:

- **CurrentStatValue:** The current value of a statistic.
- **StatMin:** The minimum value of a statistic can reach.
- **StatMax:** The maximum value of a statistic can reach.

The screenshot shows a 'Data Table' window with a search bar at the top. Below the search bar is a table with the following data:

	Row Name	CurrentStatValue	StatMin	StatMax	StatMultiplier	StatDecayRate
1	Energy	25.000000	0.000000	25.000000	1.000000	20.000000
2	Intelligence	1.000000	1.000000	25.000000	1.000000	0.000000
3	Bloodlust	1.000000	1.000000	25.000000	1.000000	0.000000
4	PTE	0.000000	0.000000	25.000000	1.000000	0.000000

Figure 4.22: Data Table

- **StatMultiplier:** Floating value by which the value that will modify the current value of a statistic will be multiplied.
- **StatModifyRate:** Floating value that will determine the rate at which a statistic will increase or decrease if it is being modified by game tick.

Once the struct has been defined, it is necessary to create a data table which will inherit it. Data tables, as the name implies, is a table of miscellaneous but related data grouped in a meaningful and useful way, where the data fields can be any valid type, including asset references [12]. Each row of the table represents one of our agent's statistics and the columns are the variables that we have previously defined in our struct (see Figure 4.22). Let's take a look at the lines of the data table and its role on the agent:

- **Energy:** Each time the agent completes a movement, it will consume an amount of energy according to its current state. Once the energy reaches 0, the agent will start searching for food. If after some time it does not find food, it will move to its initial room where it will stay for a few seconds and recover all its energy. This concept will be present in the three states.
- **Intelligence:** This attribute determines the agent's knowledge of its environment. As it increases and receives certain stimuli, the agent will learn to open the lockers or to be more efficient in moving around the environment.
- **Bloodlust:** Determines the aggressiveness of the agent towards the player.
- **PTE (Points To Evolve):** Value that determines where the agent is in the state. When it reaches its maximum value, the agent will go to its initial room and will evolve, thus changing its state.

While we were trying to implement the custom structure and data table, we ran into a problem that delayed the work. In order to access the data table, we needed it to be available globally, so that we could retrieve its data and use it in our behavior tree and AIController to manage the states. To achieve this, we opted to store this



Figure 4.23: BeginPlay event of the GameState

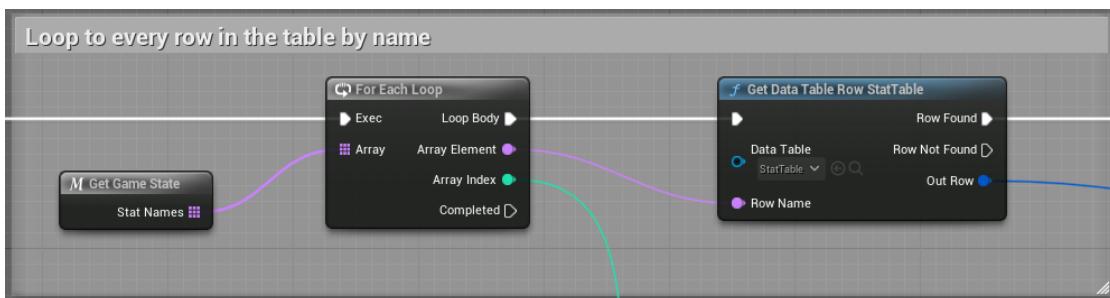


Figure 4.24: Loop to every row in the table by name

information in the project's Game State, as it is responsible for monitoring the game state and tracking game properties. We created a custom Game State for the project, and using the BeginPlay node event, we iterated through the rows of the data table and stored their names in an array of names called Stat Names (see Figure 4.23). With the row names stored, in our agent's BeginPlay event, we could iterate through the table by row name (see Figure 4.24) and store the data in a local array of the same type as our data table (see Figure 4.25).

The problem we encountered with this part of the project was that when we tried to access the array where the names of the rows of our data table were stored through the agent, the array was completely empty. While debugging, we discovered that the issue lay in the BeginPlay of our agent, as checking the length of the array in our GameState gave us the expected value. Searching through the official Unreal Engine forum we found a post from a user with a similar problem to us [1]. In this post a user comments that when using a custom GameState, the engine takes a couple of milliseconds to load it, so if you try to access it from the BeginPlay event of any actor or object in the game, the engine will check its default GameState, and then any other GameState that the user may have created. In order to bypass this bug, it was as simple as adding a delay at the start of the BeginPlay event (see Figure 4.26), so that the engine can load our GameState file.

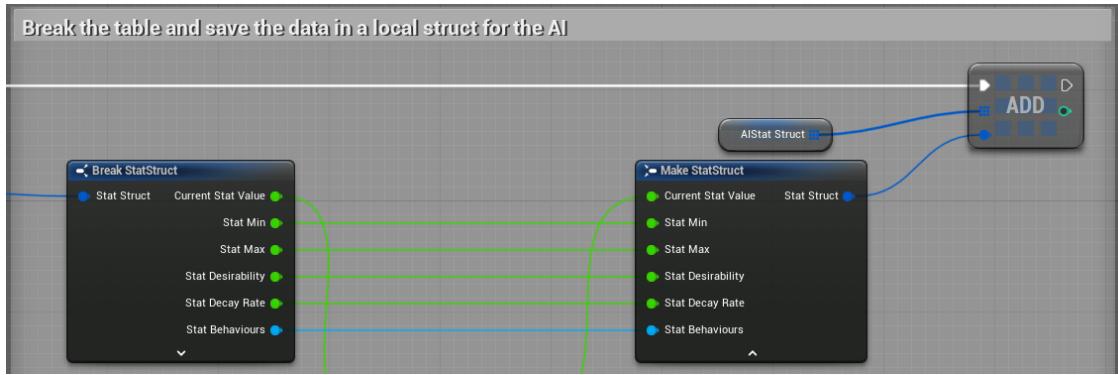


Figure 4.25: Store the data in a local array in the AI agent

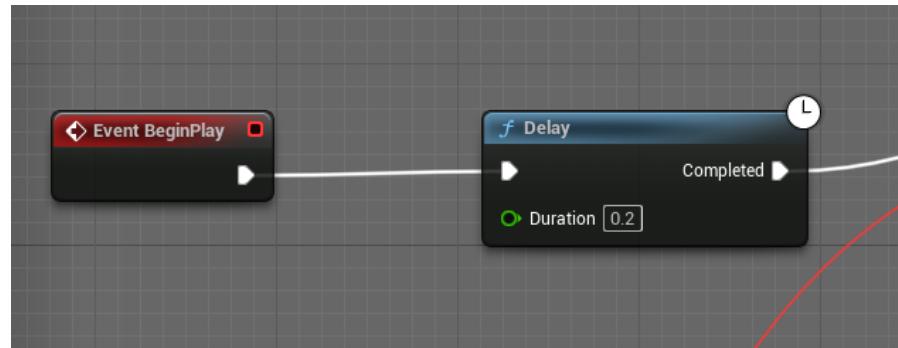


Figure 4.26: Adding a delay in the BeginPlay event

#### 4.1.5 Cowardly behavior: The Environment Query System

With some design done and problems solved, let's start implementing some of these ideas. Let's begin with the initial state of the agent, more specifically, its cowardly behavior mentioned above. For this behavior, the agent will hide every time he sees the player, it can simply run away in the opposite direction to where the player is and wait for a certain time. For this we should take into account certain factors:

- What if the agent is against a wall?
- How far should it move?
- When choosing where to go, should it take into account the cost or the distance of the path?
- If the agent cannot run away from the player, can it run towards him to find a hiding place?
- How does the agent react if while running it perceives the player again?

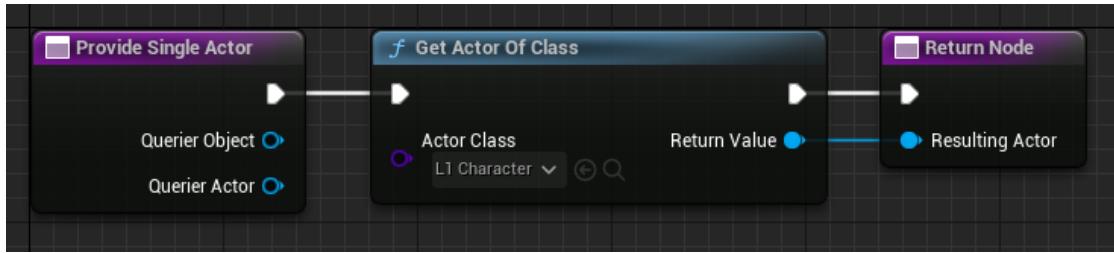


Figure 4.27: EQSC\_Player

With this in mind, we see that the agent must have some knowledge of part of the surrounding environment in order to decide where to flee. The point where the agent is going to move must meet certain conditions, since we want the agent to hide in a place that is not visible to the player and not too far away from him. Luckily, Unreal Engine has a feature within the artificial intelligence system called Environment Query System (EQS) that is used to collect data from the environment. The EQS can be used within behaviour tree to poll the environment through a variety of Test, then based on the results of those Test, the agent can make decisions on how to proceed [13]. An Environment Query is made up a number of different pieces:

- **Generators:** Used to produce the locations of Actors that will be tested and weighted, with the highest weight location or Actor being returned to the behaviour tree [14].
- **Contexts:** A Context would be what is often referred to in programming as the Querier (who is performing the Test).
- **Tests:** A Test can be performed to determine which location or Actor produced from a Generator is the "best" option, given the Context. There are several Tests that are provided with the Engine that cover a good percentage of use cases, such as "can an Item trace (see) another Location" or "is the distance between and Item and its Context within a specified range". We can add multiple Tests to a Generator which can be an effective way to narrow down the results, giving us the best possible option [15].

If we want our agent to be able to detect the player using an Environment Query, first we need to create a Context. So, we will add a blueprint of type EnvQueryContext\_BlueprintBase called EQSC\_Player where we can specify that the Test should be run on the player (see Figure 4.27). Next, we will create our first Environment Query named EQ\_FindPlayer where we will be able to start performing the Tests. When we access our Environment Query the first thing we will find is an interface similar to that of the behaviour tree, with a root node from where we can start creating our Environment Query. If we try to add a node, we will find the different types of Generators that we can add, for now we are only interested in Actors Of Class (see Figure 4.28), which finds all of the Actors of a given class within the specified Search Radius around the Search

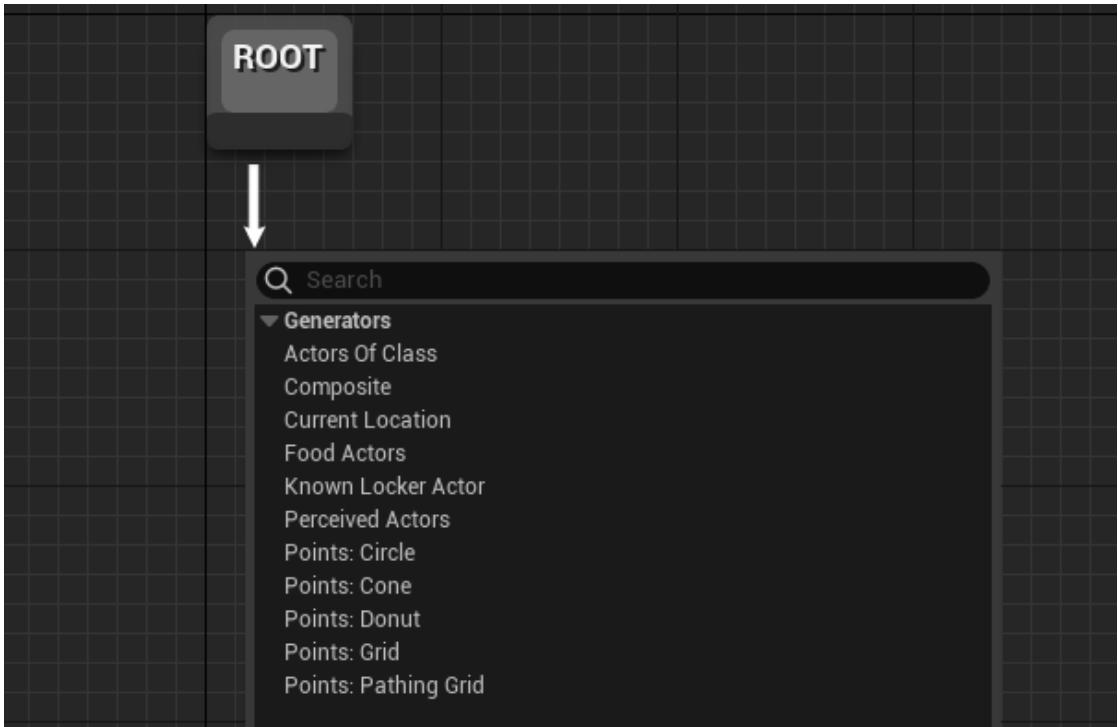


Figure 4.28: Different types of Generators

Center. We will specify that the actor we are looking for is the player, we will assign a Search Radius of 1500 units and the Search Center will be the Actor that executes the Generator (in this case the agent). The Figure 4.29 shows the details of the Actors Of Class Generator. Now that we have the Generator ready, let's add the necessary Tests to detect the player. As with the generators, we have a wide variety of tests available, but for now we are only interested in the Test Trace. The Trace Test will trace to (or from) an Item or Context and return if it hit or not something. To check that the Generators do what is expected, Unreal Engine provides an actor called Testing Pawn, to which we can assign an Environment Query and check from the inspector itself the results. If we add this actor and our player to the scene, we can check that when this Actor has no view of our player the generated Item is blue, meaning that the Test Trace has not been executed successfully (see Figure 4.31a), and if the player is visible to this actor, the Item will be green confirming the correct execution of Test Trace (see Figure 4.31b). This Environment Query will not be used in the final sub tree, since the player detection is performed by the AIPerception component.

With the Environment Query EQ\_FindPlayer finished, the next step will be to create another Environment Query that will analyze the environment near the player and choose a point that best suits our needs. We will create a new Environment Query called EQ\_HideFromPlayer, and the type of Generator we will use will be "Points: Grid", which will generate a grid of Items around the specified Context assigned (see

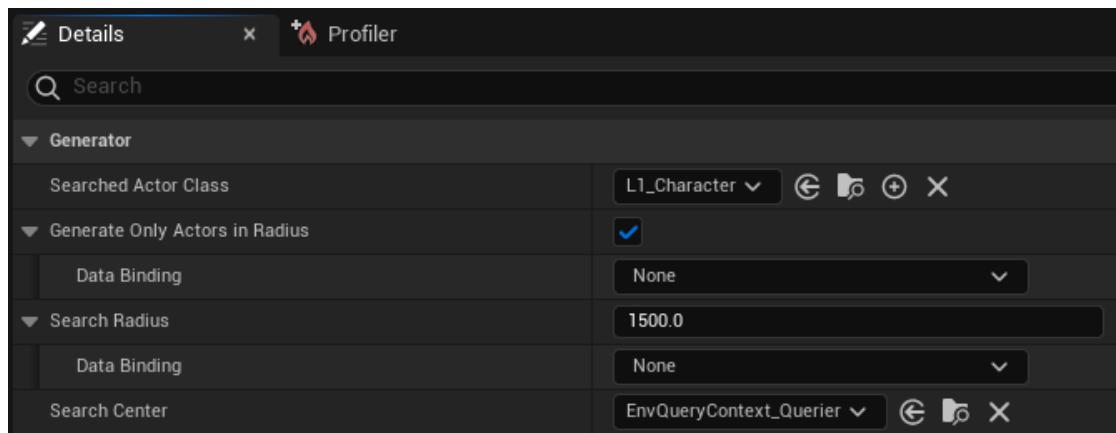


Figure 4.29: Actors Of Class details

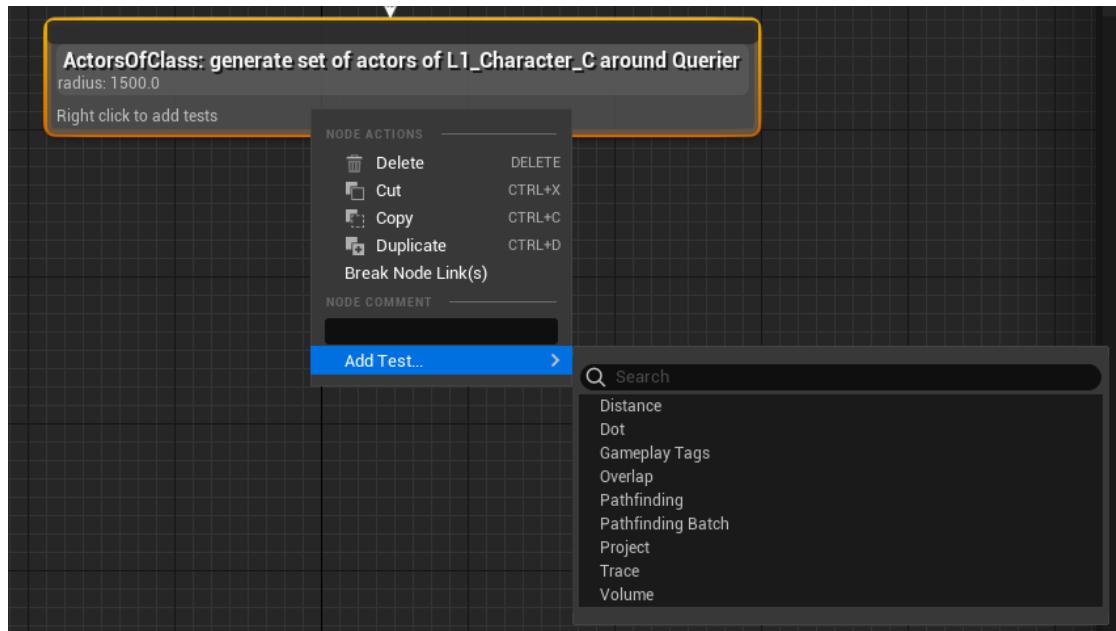
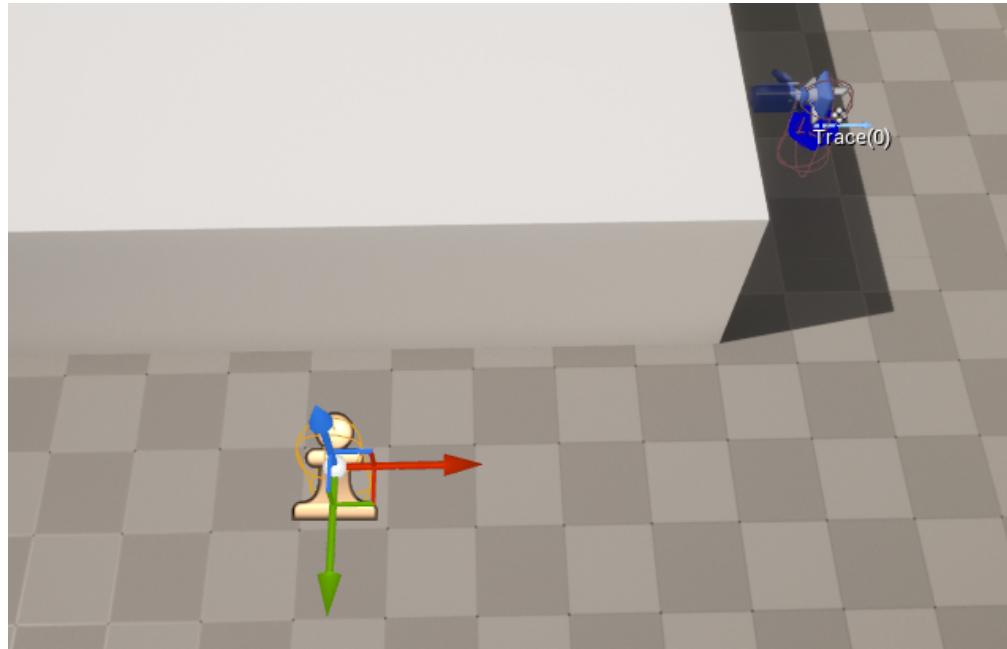


Figure 4.30: Different types of Tests



(a) Testing Pawn does not detect the player successfully



(b) Testing Pawn successfully sensing the player

Figure 4.31: Environment Query EQ\_FindPlayer testing

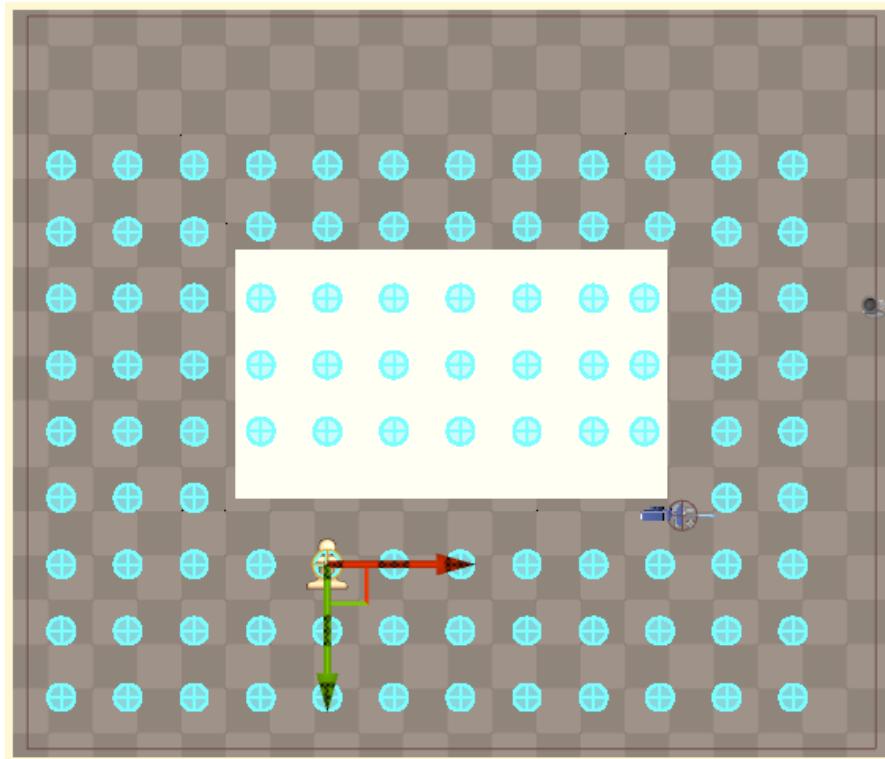


Figure 4.32: Grid produced by the Generator "Points: Grid"

Figure 4.32). This generator allows us to modify the parameters of the grid to adjust it to our needs. Among its customization options we can modify: the size of the grid, the space between the Items, and its projection both upwards and downwards for vertical spaces. We will start filtering the Items to keep only those that are not visible to the player. We will add a Trace type Test, the first thing we will have to do is to change the default Context to EQSC\_Player that we created previously, so the Items that can see the player will return a true value, and those that cannot see the player, false (see Figure 4.33). We have obtained the Items which are not visible from the player's position, the next step is to obtain an Item which is far away from the player but close to the agent. One of the tests we have available is the Distance test, which returns the direct distance between the Item and the chosen Distance To property. If in this property we specify the context of the player, we can see how the Items that were not filtered in the previous step will obtain a value between zero and one, where the Items close to the player will have values closer to zero and those far away closer to one (see Figure 4.34a). As can be seen in the figure 4.34a the Item at the top left is the one with the highest score, but we want the agent to hide somewhere closer to it and, if possible, behind an obstacle. A functionality that the Environment Query Tests have is that every Test that we add will have repercussion with the result of the previously added Tests, therefore if we now add a Test to score the Items that are closer to the agent, the current values of

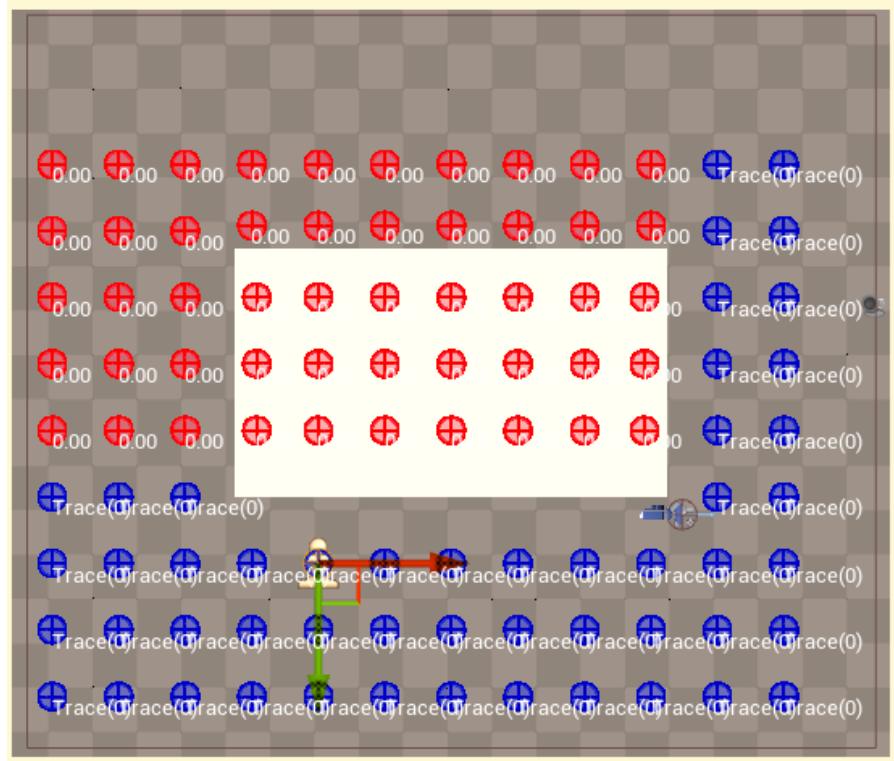
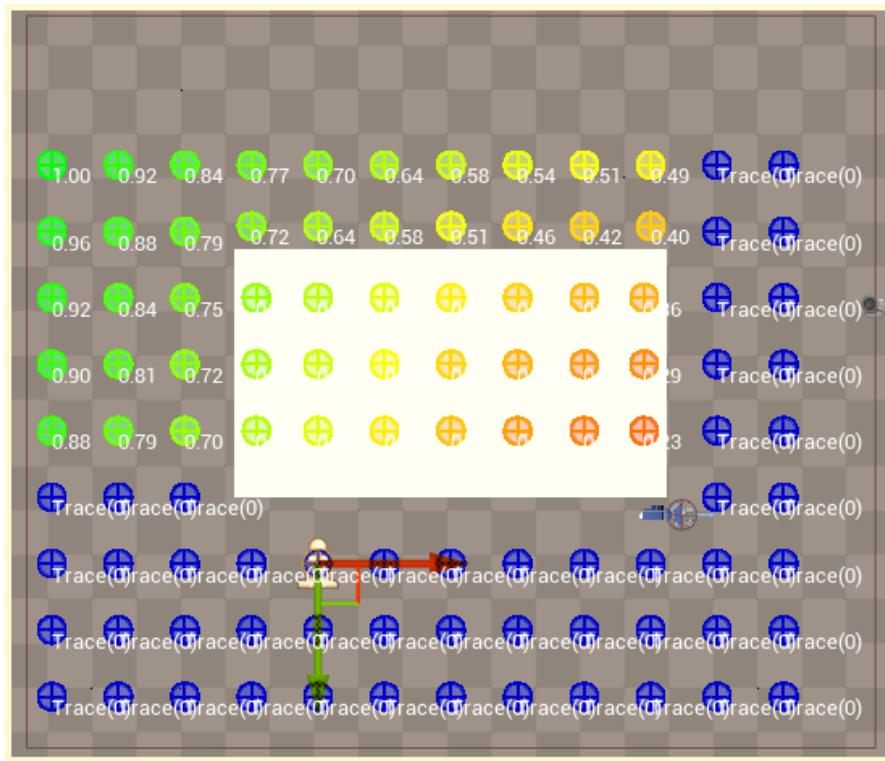
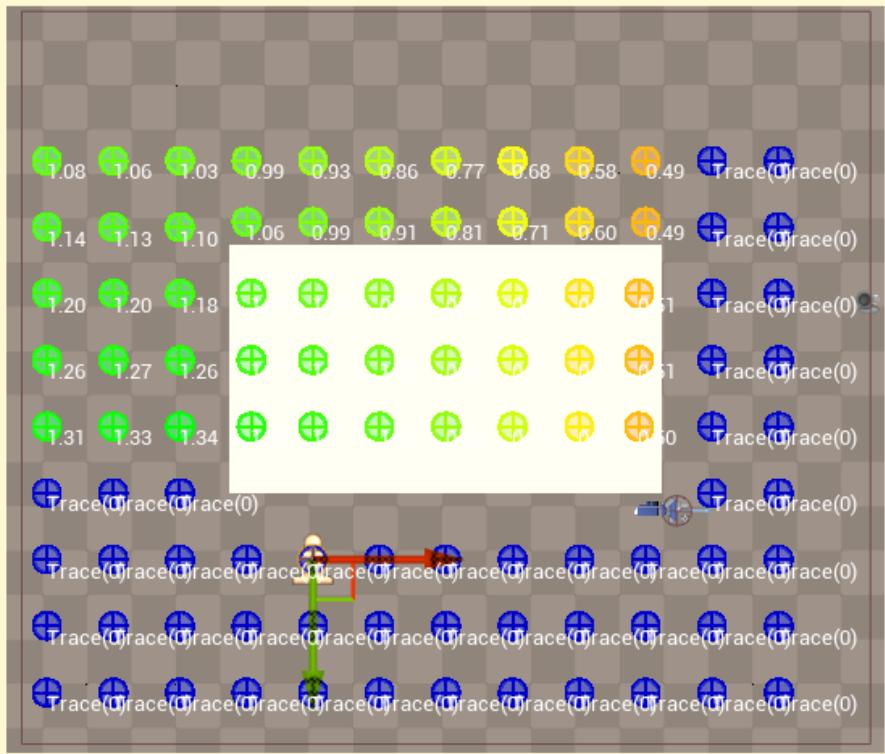


Figure 4.33: Result of EQ\_HideFromPlayer after applying a Trace Test from the Player

the Items will be affected. Hence, if we add another Test of type Distance with target to our agent and invert its result, the score of the Items will be better for those that are far away from the player but close to the agent (see Figure 4.34b). At this point we could finish this Environment Query, but let's add one more Tests to avoid future problems. As can be seen in both images of Figure 4.34, the Items that are above the white rectangle have a score and should not have it because they are in an unreachable position for the agent. To prevent the Environment Query from ignoring these Items we can add a Pathfinding Test, used to determine if a path exists to (or from) the Context, how expensive the path to (or from) the Context is, or how long the path is. For this test we are interested in filtering the items that do not have a path to the agent (see Figure 4.35).



(a) Result of EQ\_HideFromPlayer after applying a Distance Test to the Player



(b) Result of EQ\_HideFromPlayer after applying a inverted Distance Test to the Agent

Figure 4.34: Environment Query EQ\_HideFromPlayer testing



Figure 4.35: Result of EQ\_HideFromPlayer after applying a Pathfinding Test to the Agent

With this Environment Queries finished, let's create the sub tree for the agent's coward behavior. The first node of this new sub tree will be BTT\_SetSpeed to modify the speed of the agent to get the feeling that it is running away in a hurry. Next, we will have to execute our Environment Query EQ\_HideFromPlayer. In order to execute the Environment Queries, Unreal Engine provides a node called "Run EQS Query", to which we can assign an Environment Query to be executed. If we look at the details of this type of node, we will find the parameter "Query Template" where we can assign the Environment Query that we want to be executed, the "Run Mode" parameter that will determine which Item will be stored in the Blackboard (we can choose if it returns the best Item, a random Item among the best 5% or a random one among the best 25%) and "Blackboard Key" which will be a variable of our blackboard where the position in the world of the obtained Item is stored (in this case we will use the targetLocation variable). Once the Item is obtained, through the moveTo node our agent will go to the position stored in the targetLocation variable and finally we will add the Wait node so that the agent waits between two and four seconds to move again. The Figure 4.36 shows the obtained sub tree.

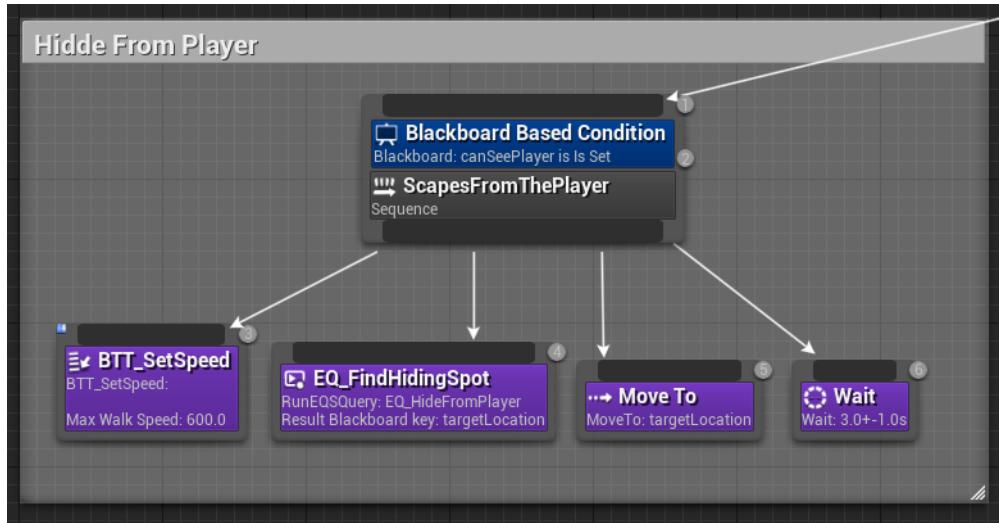


Figure 4.36: Coward Behaviour sub tree

#### 4.1.6 Pulling the player out of hiding

When we implemented the lockers, we got our agent to stand in front of them when it saw the player enter one and to be able to open it to kill him. So regarding the lockers, we only need to get the agent to open the one where it has seen the player hiding and take him out of his hiding place to have everything designed for them. Let's see what new variables and tasks we need to achieve this:

#### BLACKBOARD

- **desiredObject**

Object type variable where a reference to the game element with which the agent will be able to interact or move towards depending on the situation will be stored.

#### TASKS

- **BTT\_UseActor**

Task that will receive a variable of type Object. When executed, the agent will interact with the object that has received the task.

Currently when the agent opens a locker, at code level what it is doing is calling a function of the lockers that only performs the animation of the door opening. Now we want the interaction of the agent with the elements of the environment to be more complex, therefore it will not be enough to call a function of the object itself. For this, we are going to create a new interface called BPI\_UsableActor which will have two functions: AI\_UseActor which will send to our agent the object that is going to use, and UseActor which will execute the logic established in the object that the agent is

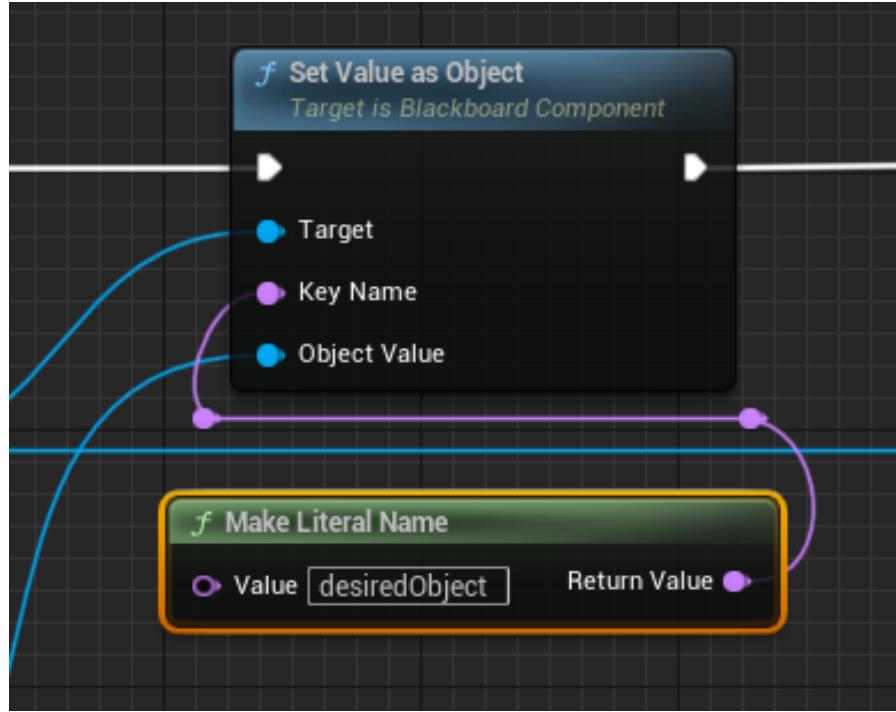


Figure 4.37: Store the reference to the locker on the blackboard

going to use. When the actor is in front of the element with which it is going to interact (in this case a locker) the BTT\_UseActor node will be executed, which will cause the actor to execute a Line Trace. A Line Trace consists of performing a collision using an invisible laser along a given line and returning the first Object that hits the laser [20].

Therefore, when the agent sees the player hiding in a locker, taking advantage of the Did Enemy See function we created earlier, we will add a new node to store a reference to the locker (see Figure 4.37) where the player has hidden in the desiredObject variable of the blackboard. When the agent approaches the locker, the BTT\_UseActor node will be executed in order to launch the Line Trace and detect the object. If the detected object is the same as the one stored in the desiredObject variable, the node will call the function AI\_UseActor that the agent has inherited from the BPI\_UsableActor interface, passing the locker as a parameter. In the agent part, it will check if the object implements the BPI\_UsableActor interface, if it does, it will call the UseActor function that the object will have inherited from the interface in question. In this function, an animation will be executed that consists of a lerp between 0 and 90 degrees to open the door, and once it finishes executing, after a short delay the player will be moved from inside the locker to outside through the Teleport node (see Figure 4.38), which receives the target, a final position to move the target and a rotation vector to rotate the target in said direction.

Now we only need to create the sub tree. We will start with the node BTT\_SetSpeed to be able to modify the speed of our agent, then the flow of the tree will pass to the node moveTo to which we will receive the variable targetLocation so that it goes to the

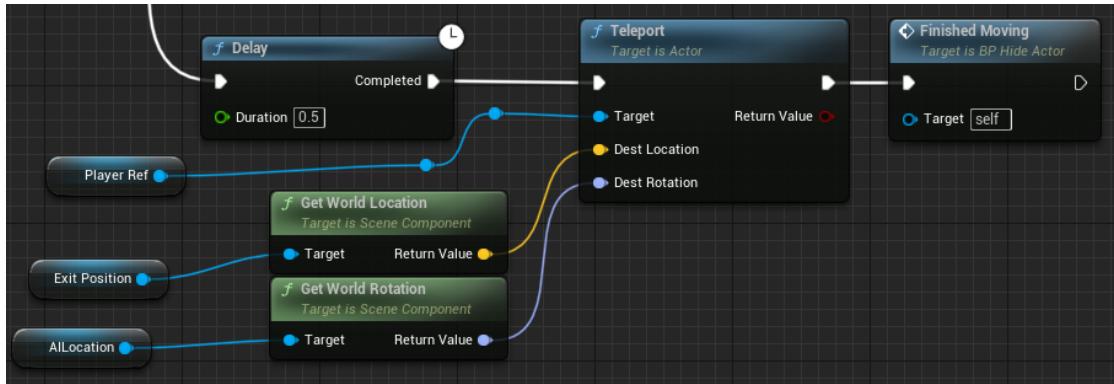


Figure 4.38: Teleport player out of the locker

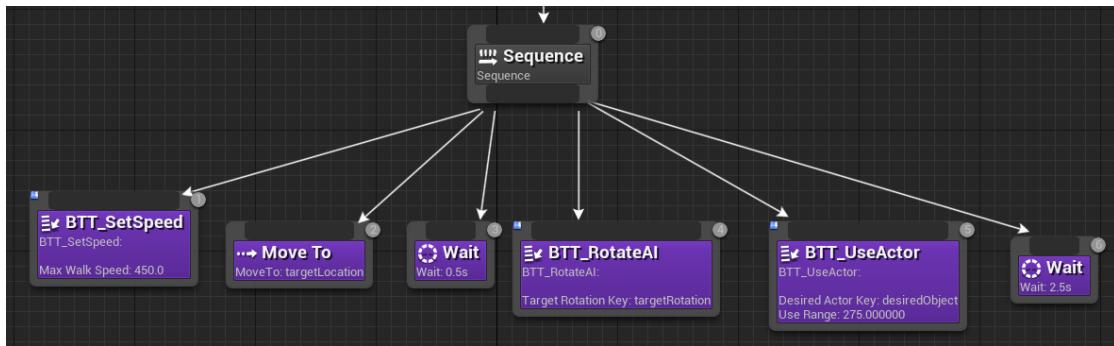


Figure 4.39: Bring the player out of hiding sub tree

door of the locker. Then with the BTT\_RotateAI node we will make sure that the agent turns to look at the locker so that when the BTT\_UseActor node is executed the agent has no problem with the locker detection. And we will finish the sub tree with a Wait node so that the agent will give some time to the player to escape when it takes him out of his hiding place. Figure 4.39 shows the resulting sub-tree.

#### 4.1.7 Strafing the player

When we use the verb Strafe for a video game, we are not referring to its usual meaning. When we talk about strafing in video games, we refer to the act of sliding in a direction, allowing the player or NPC to keep the camera focuses on a target while moving in a different direction. Lancheres mentions in his book [31] that there are several techniques for this mechanic, all of them related to the type of movement or action that is being performed in the video game. For what we are trying to achieve, we are interested in the Circle strafing technique, which consists of moving around an opponent in a circle while facing them.

As mentioned earlier, when the agent is in its intermediate state and chasing the player if it approaches him at a certain distance, it will start strafing the player. Let's

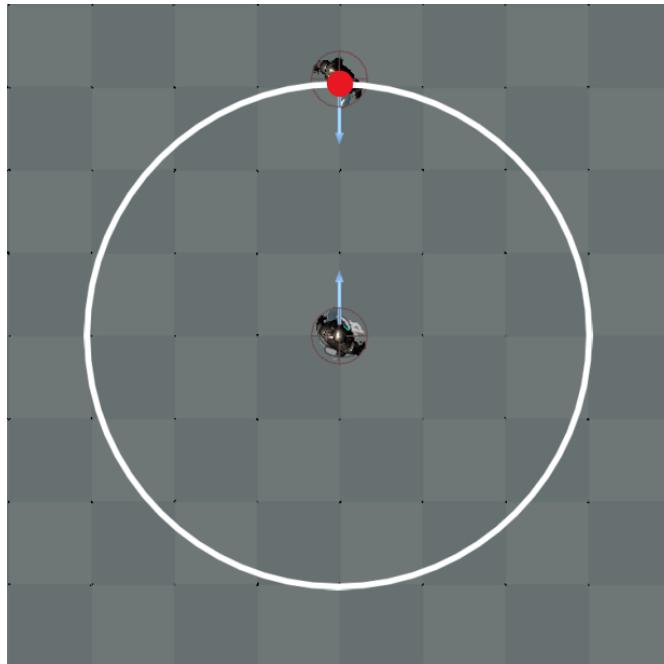


Figure 4.40: Circumference where the new positions of the agent will be generated

start by looking at the mathematics behind this game mechanic. Strafing is based on the mathematical concept of trigonometry. We must take into account two aspects: the orientation of the agent and its lateral displacement. Using angles we can represent the character orientation which we will use to determinate the direction in which the lateral movements occur. The lateral displacements involves moving perpendicular to the character's orientation using trigonometric functions such as sine and cosine. Combining the orientation angle with a constant movement speed we can get horizontal and vertical displacement to achieve strafing motion.

Knowing how this movement is calculated, let's see how to implement it in the game engine. Whenever the agent is going to perform the straf, it will have to "fix" his target on the player, and taking the player's position as the center of a circle with radius the distance between them (see Figure 4.40), the agent will calculate his new position on the circle. To obtain the agent's orientations, Unreal Engine provides us with a pair of nodes called "Get Actor Forward Vector" and "Get Actor Right Vector", which given an Actor return its front and right direction vector respectively (see Figure 4.41). We will multiply the front and right direction vector by a float value, and both will be added to the vector of the agent's current position in the world. This way we will obtain a new position to the right of the agent, but we also want the agent to be able to generate a new position to its left. Unreal Engine does not have any way to obtain a left direction vector, but this is not a problem because if we want to obtain this vector we simply multiply by minus one the right direction vector of the agent. Figure 4.42 shows a possible result of positions given by the straf.

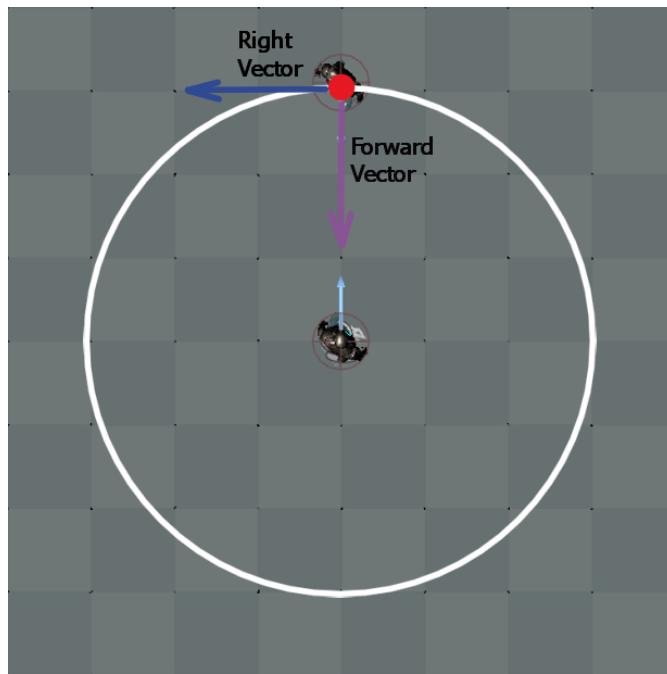


Figure 4.41: Forward and right vectors

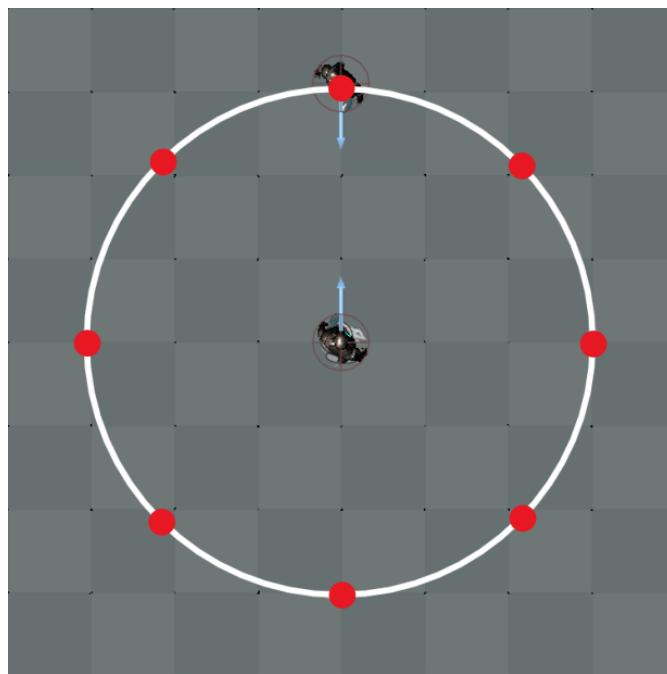


Figure 4.42: Straf positions

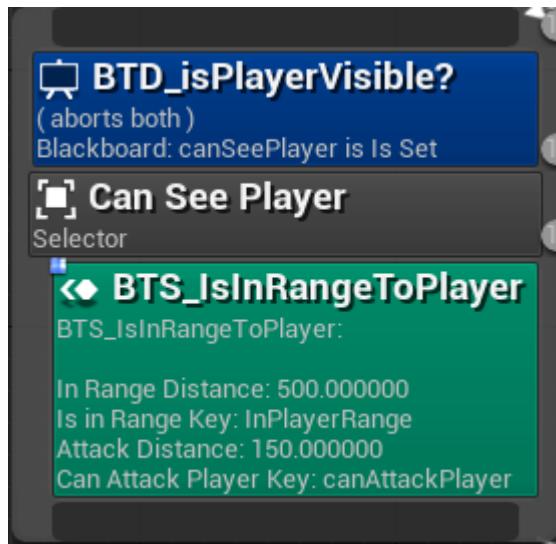


Figure 4.43: BTS\_IsInRangeToPlayer

Now that we know how to calculate the positions for the straf, let's look at the new additions to our artificial intelligence:

## BLACKBOARD

- **strafLocation**

Vector type variable where the new position generated during agent strafing will be stored.

## TASKS

- **BTT\_FocusOnPlayerParallel**

Task that will ensure the agent will look at the player before calculating the new position for the straf.

- **BTT\_GetStrafLocation**

Task that will calculate the new position to be taken by the agent while strafing the player.

Currently, when the agent chases the player, the Service BTS\_IsInRangeToPlayer is executed in order to modify the flow of the behavior tree so that the agent can attack the player when it reaches him. We are going to take advantage of this Service and use the variable inRangePlayer to know if it is at the right distance to start strafing the player. As we did with the canAttackPlayer variable, we will specify the distance at which we want the agent to start performing the straf (see Figure 4.43), and the service itself will change the value of the inPlayerRange variable.

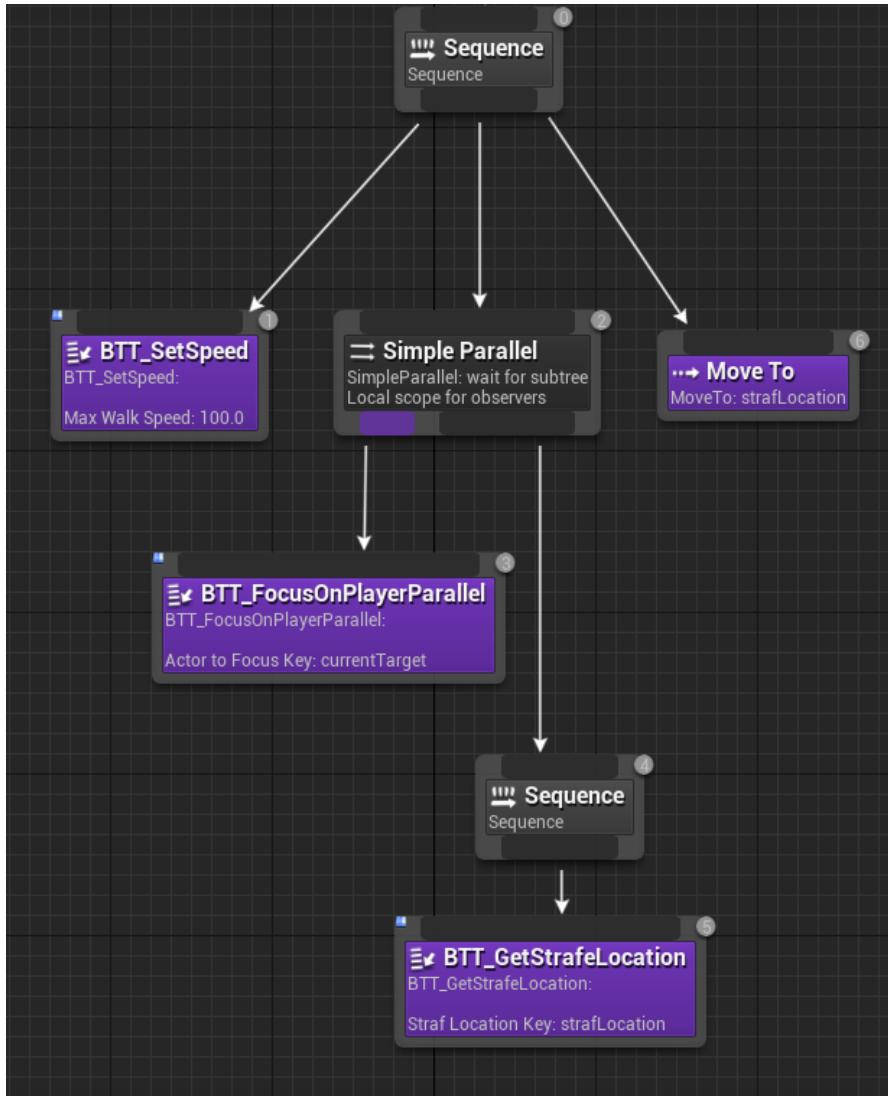


Figure 4.44: Straf behaviour sub tree

With all this we can start creating the sub tree. The first task will be **BTT\_SetSpeed**, to reduce the speed of the agent and give a greater sense of stalking. Now we need that while the agent is looking at the player, it calculates its new position and moves to it. For this we are going to use the composite node called **Simple Parallel**. **Simple Parallel** allows us to handle concurrent behaviors and the parallel node begins execution on all of its children simultaneously [9]. The concurrent behaviour in this case is going to be **BTT\_FocusOnPlayerParallel**, so when the agent calculates the new position with the **BTT\_GetStrafeLocation** task, it will always be with respect to the player. And finally, the agent will move to its new position thanks to the **moveTo** node. Figure 4.44 shows the resulting sub tree.

#### 4.1.8 Modifying the agent's statistics

Previously we created a structure and a data table to be able to store the statistics of our agent and to be able to work with them in a fast and simple way. To do this, we are going to implement a series of functions in our agent that will allow us to work with its statistics data:

- **GetStatCurrentValue:** Function that receives an integer value X. The X position of the agent's statistics array is checked and its current value is returned.
- **ModifyStat:** Function that receives an integer value X and a float value. The X position of the agent's statistics array is checked, its current value is multiplied by the StatMultiplier variable, and the result is added to the value of the statistics.
- **Evolves:** Function that goes through all the agent's statistics and increases the maximum value of the statistic by twenty five.

The next step will be to see how to use the functions we have just created. We want the agent's statistics to be modified after performing a certain action or task, for example, after moving, when it finds the player again after having lost him, when it opens a locker, etc. Seeing these examples, we realize that these are actions that are executed and managed from the agent's own behavior tree, so we can create a series of tasks that allow us to modify these statistics after the agent performs the actions we are interested in.

### TASKS

- **BTT\_ModifyStat**

Task that communicates with the agent to execute its function called "ModifyStat". This task receives an integer value to access one of the positions of the structure that stores the agent statistics. The integer will be stored in the variable "StatTo-Modify". And a floating value which will be the amount to add to the statistics, which we will store in the variable "Amount". Both variables will be editable from the Unreal Engine inspector itself, so we will be able to use this task in different positions of the behavior tree to modify different statistics. Figure 4.45 shows the logic inside the BTT\_ModifyStat task.

- **BTT\_ModifyMultiplier**

Task to modify the value of StatMultiplier variable. Receives an integer value called "Statistic" to access one of the positions of the structure that stores the agent statistics. And a float variable called "Modifier" to update the StatMultiplier value. As in the BTT\_modifyStat task, the variables will be editable from the Unreal Engine inspector.

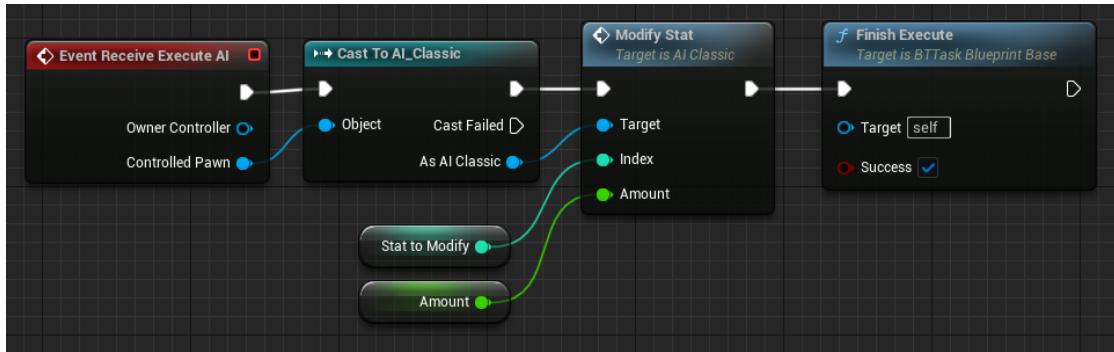


Figure 4.45: BTT\_ModifyStat

#### 4.1.9 Food searching behavior

As mentioned in previous points, the agent has an energy attribute which decreases every time it performs an action, and if its value reaches zero it will start looking for food. If food is found, the actor will interact with the game element and his energy will be restored, and if not, it will go to his initial room where it will rest and his energy will gradually recover. However, by looking in detail at the AIPerception system, we can make it so that instead of the agent searching for food when hungry, the agent remembers food that it has seen around the game environment before running out of energy and when it needs to eat, instead of searching, it goes directly to the food it remembers seeing, and if it did not remember any, it would start searching for food in the area. The AIPerception system has a floating value attribute called "Max Age" which specifies the time that the artificial intelligence remembers a stimulus perceived by any of its senses, with zero being that it will never forget that stimulus, and any other positive value being the time in seconds that it will take to forget the stimulus. On the other hand, the AIPerception system has a series of arrays where it stores all the game stimuli that the artificial intelligence has perceived throughout its life and those stimuli that it remembers.

For the agent to recognize the food as a stimulus, we must specify it in some way in the food object itself. For this, Unreal Engine provides a node named "Register Perception Stimuli Source", which we will add in the BeginPlay event (see Figure 4.46) so all the food objects that we add to the environment will be recognized as stimuli.

When we implemented our agent's sense of sight, we commented that we were not going to take into account which element of the game was being perceived since the agent was only able to recognize the player. Now that we have a new perceptible element for the agent, we will have to modify this part. To do this we will check the class of the perceived element with the "Get Class" node, and if it is of the type BP\_BaseFood (which is the name of the class that we have given to the food), we will store a reference to the perceived food in an array called "Stored Food" of type BP\_BaseFood.

Seeing that we need the agent to search for something in its environment, we are going to resort to the Environment Query System to achieve it. In the chapter 4.1.5

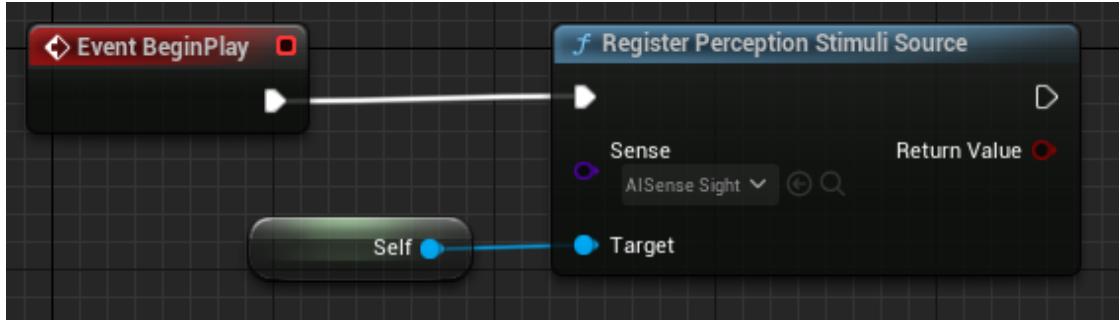


Figure 4.46: Register Perception Stimuli Source

we saw that this system has modules called Generators that we use to search for certain actors. Unreal Engine allows us to create our own Generators and thus be able to search for actors that fit our needs, in this case, food that the agent has perceived and still remembers its position in the environment. To create a custom Generator we will add a new Blueprint of type Env\_Query\_Generator\_Blueprint\_Base. Within this new blueprint that we will call EQSG\_FoodActors, we will go through the "Stored Food" array of our agent and we will check if some of its elements exist in the AIPerception array of stimuli that the agent still remembers (this array is accessible through the "Get Known Perceived Actors" node). Any food actor that exists in both arrays will be added to our Generator using the "Add Generated Actor" node.

The implementation of the Generator EQSG\_FoodActors has been done following the pseudocode in Algorithm 1.

---

**Algorithm 1** Add food to the Generator

---

```

procedure GETFOOD(perceivedFood, rememberedFood)
  generatorFood ← ∅
  for food in perceivedFood do
    if food in rememberedFood then
      generatorFood.append(food)
    end if
  end for
  return generatorFood
end procedure
  
```

---

Next, we will create a new Environment Query called EQ\_FindFood (see Figure 4.47), add our custom Generator to it, and a Distance Test to get the closest food to the agent at the time the Environment Query is executed.

With this, our agent is able to find the closest food to him that it remembers having seen. Now, we are going to make the agent able to go to his starting room in order to recover energy. As a way to specify to our agent which room to go to, we will create



Figure 4.47: EQ\_FindFood

a node that we will call BP\_SameRoomNode which we will place in the initial room. The next step will be to create a Environment Query called EQ\_FindSafeRoomNode to find the exact point in the environment where the node we just placed is located. This Environment Query will consist in a ActorsOfClass Generator that will search for the BP\_SafeRoomNode actor in the game environment, and a Distance Test in order to obtain its position in the environment (see Figure 4.48). Through the Testing Pawn actor we can check how, indeed, the Environment Query that we have just created detects the BP\_SafeRoomNode node (see Figure 4.49).

Next, we are going to create an Environment Query for the agent to move to its initial room. First we must create a context for the node BP\_SafeRoomNode (see Figure 4.50), in order to generate Items around the node so that the agent has a point in the world to move to. The next step will be to create a new Environment Query which we will call EQ\_FindSafeRoomPath, which will be in charge of finding the shortest path from the agent to the safe zone. We will start by adding a Grid Generator to our Environment Query and modify its parameters to adjust the Items generation to the room dimensions (see Figure 4.51). In order to get the agent to be able to move to one of these Items, we need to score them in some way. To do this we will add a Distance Test to the BP\_SafeRoomNode node, so that it gives a value to each one (see Figure 4.52). Lastly, we will add a Pathfinding Test to our Generator, and we will change its configuration so that instead of checking if there is any path to one of the generated Items, it always chooses the shortest path to one of these. Figure 4.53 shows the resultant Environment Query.

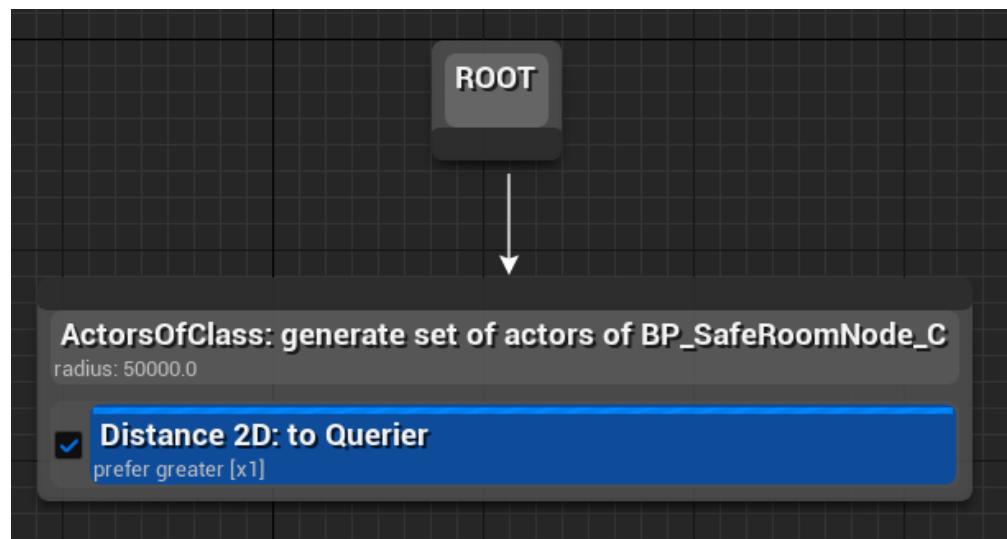


Figure 4.48: EQ\_FindSafeRoomNode

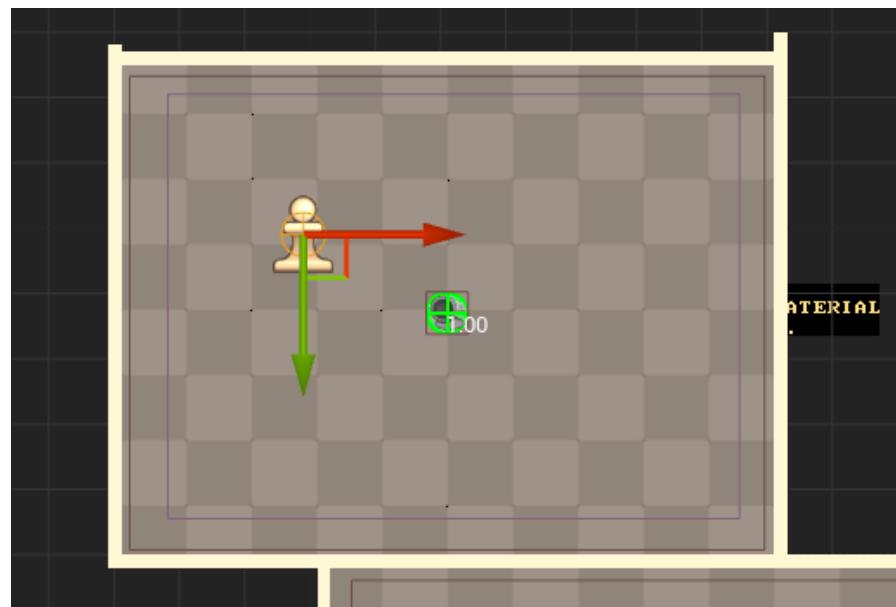


Figure 4.49: Testing EQ\_FindSafeRoomNode

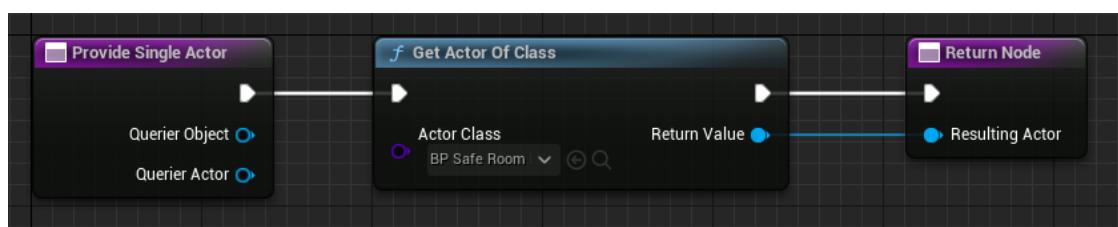


Figure 4.50: EQSC\_SafeRoom

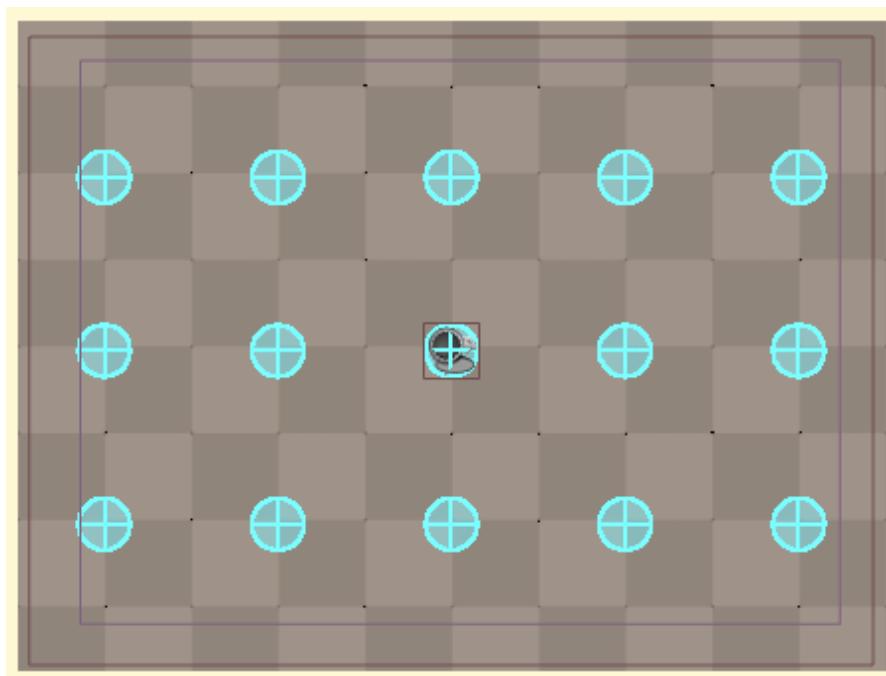


Figure 4.51: EQ\_FindSafeRoomPath Item grid

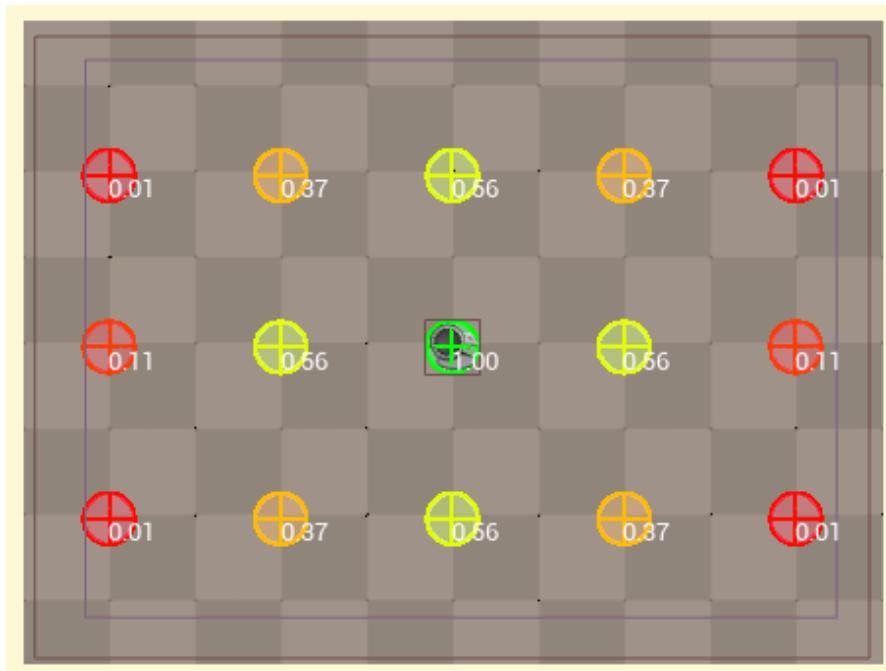


Figure 4.52: Sorted item grid by distance to the node

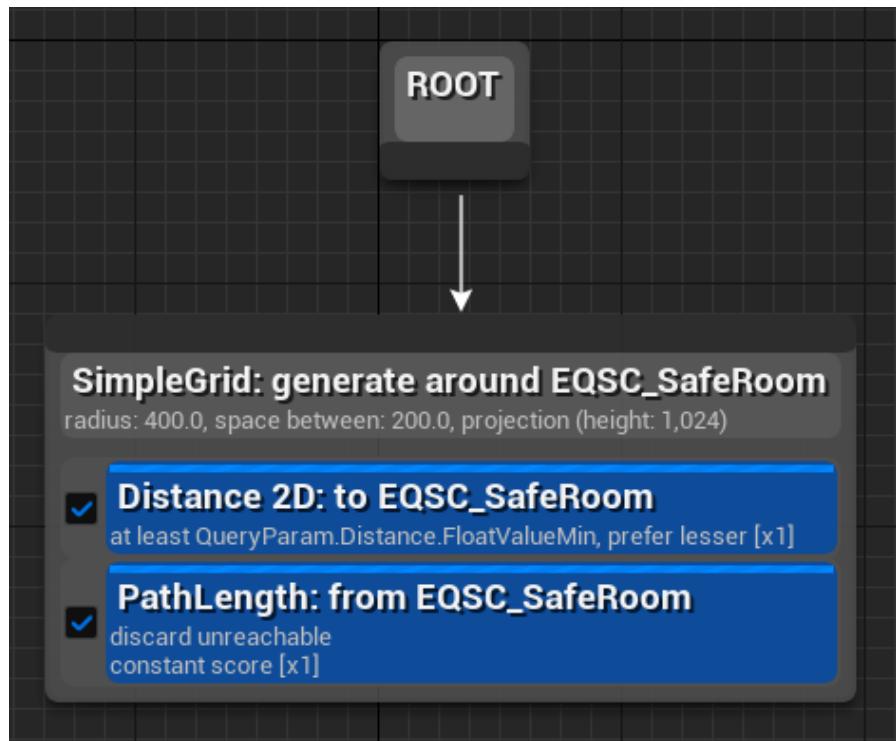


Figure 4.53: EQ\_FindSafeRoomPath

Now let's see what modifications and additions we need to make to our artificial intelligence to implement what we have just seen:

## BLACKBOARD

- **hasLowEnergy**

Boolean variable to know if the agent is at low energy.

- **knowAboutFood**

Boolean variable to know if the agent has seen food at any time.

- **energyRestored**

Boolean variable to know when the agent's energy has been restored.

## TASKS

- **BTT\_CallEventNode**

Task in charge of executing the logic to fully restore the agent's power.

## SERVICES

- **BTS\_EnergyCheck**

Service in charge of checking whether the agent has run out of energy. Each time it is executed, it will call the "GetStatCurrentValue" function of the agent and check if it is less or equal to 0, and the result of the check will be the value of the hasLowEnergy variable.

- **BTS\_RestoringEnergy**

Service in charge of checking if the agent has restored all its energy. Like the BTS\_EnergyCheck Service, in each execution it calls the "GetStatCurrentValue" function of the agent to check if the energy value is equal to its maximum value, and the result of the check will become the new value of the energyRestored variable.

## DECORATORS

- **BTD\_iHaveLowEnergy?**

Checks the value of the hasLowEnergy variable.

- **BTD\_iKnowAboutFood?**

Checks the value of the knowAboutFood variable.

In order for our agent to switch from patrolling to foraging, we will first add the Service BTS\_EnergyCheck and the decorator BTD\_iHaveLowEnergy? so that the flow between these two behaviors will be managed automatically. Then, the sub tree for energy recovery will consist of two sub trees. The first one will be the left one, it will only be executed if the agent has seen food and has low energy. Its first node will be a Run EQS Query to execute the Environment Query EQ\_FindFood. If the Environment Query finds food, then the agent will move to it with the moveTo node and eat the food with the BTT\_UseActor node. In case the agent does not remember having seen food, the execution of the node will fail since the Environment Query is returning a null value, and the flow will pass to the right sub tree. In this part of the tree, we will reuse the random point patrol behavior to make the agent start looking for food. If after three executions of this behavior the agent has not found food, we will execute the Environment Queries EQ\_FindFood and EQ\_FindSafeRoomNode to make the agent go to its initial room. Once there, the BTT\_CallEventNode will be executed so that the agent starts restoring energy. To know if the agent has already recovered all its energy, we will add a Conditional loop decorator in the BTT\_CallEventNode task that will check if the value of the energyRestored variable is true.

### 4.1.10 Patrolling using the Environment Query System

To get our agent to patrol using the Environment Query System we are going to reuse the nodes we created for the hierarchical pathfinding and everything we have applied

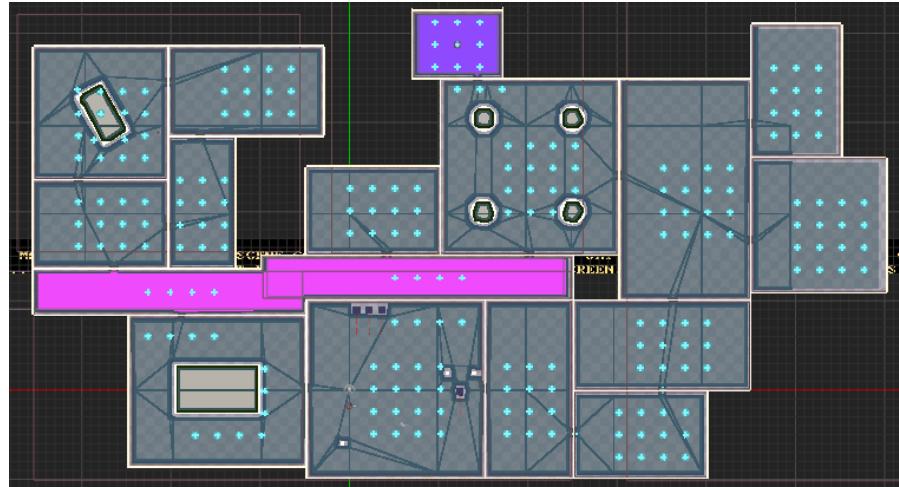


Figure 4.54: Items generated for the patrol through EQS

about the Environment Query System. First we will create a new context to be able to generate Items around the room nodes. We will add a new blueprint of type EnvQueryContext\_BleuprintBase and we will call it EQSC\_RoomNode, which will be in charge of obtaining all the actors of type BP\_RoomNode. Next, we will create a new Environment Query which we will call EQ\_NavigationQuery we will add a Grid type generator, and modify its attributes so that it generates a small amount of Items around the nodes (See Figure 4.54).

The next step will be to add the necessary Tests so that the agent follows the flow of the environment and does not retrace its steps unless it is its only option. We will start by adding a distance test to filter all those Items that are very close to the agent, in this way we will prevent it from moving to an adjacent Item in the same room, and we will make sure that it always moves to another room. To follow the flow of the environment, we will use a Test Dot, to give much more priority to all those Items that are in the direction of the agent's vector direction.

To create the sub tree of this behavior we will not need to create or modify our artificial intelligence. We will only need a node of type Run EQS Query to execute our Environment Query EQ\_NavigationQuery and a node moveTo for the agent to move to the position of the selected Item. Figure 4.55 shows the resultant sub tree.

#### 4.1.11 Modifying the NavMesh navigation cost

Our agent has three types of patrol behaviors and each of them will be performed in the state where it corresponds. The first two are designed so that they are not entirely efficient and effective, where it may be the case that the agent does not move from the same area for a period of time, or retraces his steps. While the third behavior for EQS patrolling is designed to follow the flow of the environment and avoid backtracking. Testing the third behavior for the patrol we saw that the agent did not end up moving

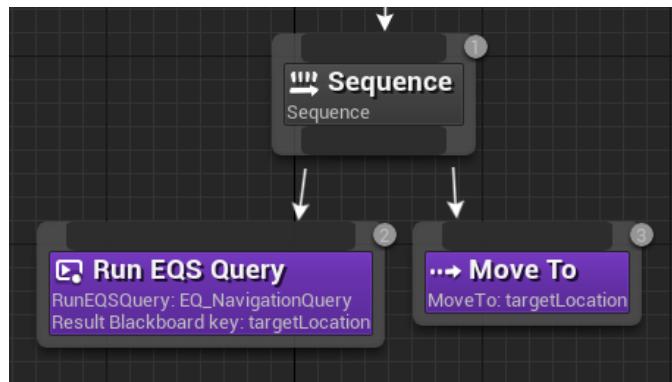


Figure 4.55: Sub-tree for patrolling through EQS

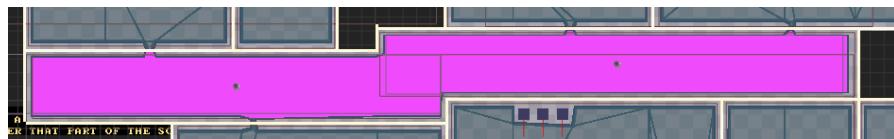


Figure 4.56: NA\_Cheap

as expected, and often ended up veering between the same two points or did not follow the flow of the environment.

To solve this, we can modify the cost of Unreal Engine's own NavMesh using a component called Navigation Modifier Volumes. Through these components we can create customized NavMesh to influence the navigation cost of the NavMesh polygons, so we can get the agent to prioritize moving through that area because its cost is lower, to take it less into account because it has a higher cost, or not to move through the area at all. So we will create a component of type NavMesh Modifier Volume called NA\_Cheap, which we will be able to add as many times as we want to our environment, and we will modify the cost of its nodes so that the agent prioritizes to move through the areas where we place the component. Figure 4.56 shows the area where this component has been added in the environment.

#### 4.1.12 Agent reacts to light: custom sense of sight

One of the drawbacks of the AIPerception system's sense of sight is that it does not have nodes for light detection. So if we want the agent to be able to react to the light of the player's flashlight we will have to do it by other methods. For this, we are going to create an invisible object that will be linked to the light cone of the flashlight, and it will be activated and deactivated along with it (See Figure 4.57). Let's see what additions we should make to our artificial intelligence:

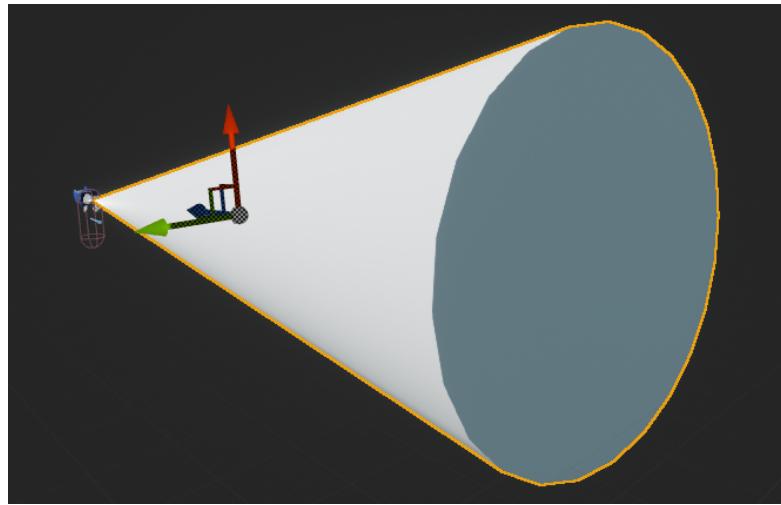


Figure 4.57: Invisible component for light detection

## BLACKBOARD

- **isDistracted**

Boolean variable to know if the agent is distracted.

## DECORATOR

- **BTD\_iAmDistracted?**

Checks the value of the isDistracted variable.

We will create a new function in our agent called LookForLight, which will be executed at all times and will launch a LineTrace in the direction of the agent's direction vector. When the LineTrace detects the component that we have linked to the light cone of the flashlight, we will perform the logic as if it were an event of the AIPerception system. We will access the agent's blackboard, modify the value of the isDistracted variable, and also store the position where the LineTrace collides with the component in the targetLocation variable. Once the value of the isDistracted variable is modified, any behavior will be stopped and the sub tree for the distracted behavior will be executed. This sub tree will consist in a moveTo task where the agent will move to the position stored in the targetLocation variable. Then it will wait a couple of seconds in this position to give the impression that the agent is distracted looking at the light. And finally, with the BTT\_SetBoolValueKey task we will change the value of the isDistracted variable to false so that the agent is no longer distracted. Figure 4.58 shows the resultant sub tree.

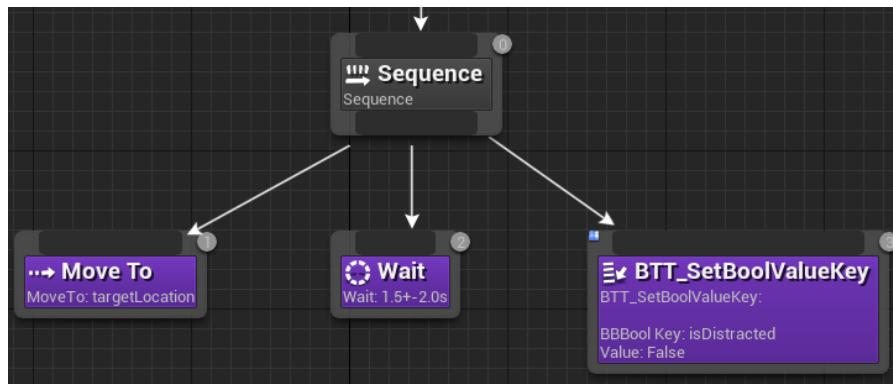


Figure 4.58: Distracted behaviour sub tree

#### 4.1.13 Switching between agent's states: Modular AI

So far we have implemented the different behaviors that the agent will have in each of its three states. In this section we are going to implement the capacity of our agent to evolve and how it will interact in different ways in the same situation depending on its state. As usual, we will start by looking at the modifications and additions that we will have to make to our artificial intelligence:

### BLACKBOARD

- **AIPTESState**

Integer variable that will take a value between 0 and 2, both included, to specify in which state the agent is.

- **AIIIntState**

Integer variable that will take a value between 0 and 2, both included, to specify the intelligence of the agent.

- **AIBloodState**

Integer variable that will take a value between 0 and 2, both included, to specify the aggressiveness of the agent.

- **isEvolving**

Boolean variable that indicates if the agent is going to evolve or not.

- **didPlayerEntersLocker**

Boolean variable that indicates whether the agent has seen the player enter a locker.

- **didPlayerLeavesLocker**

Boolean variable that indicates if the agent has seen the player leaving a locker.

## TASKS

- **BTT\_Evolve**

Task that will execute the agent's Evolves function.

## SERVICES

- **BTS\_PTECheck**

Service in charge of checking if the agent meets the requirements to evolve and change status.

- **BTS\_IntelligenceCheck**

Service in charge of checking if the agent meets the requirements to change their intelligence.

- **BTS\_BloodlustCheck**

Service in charge of checking if the agent meets the requirements to change their aggressiveness.

## DECORATORS

- **BTD\_iAmEvolving?**

Checks the value of the isEvolving variable.

For our agent to evolve we will place the BTS\_PTECheck service in the root node of our tree, this way the agent's PTE value will always be checked. For the agent to evolve, the value of the PTE variable must reach its maximum value, and at each evolution its maximum value, as well as that of all its statistics, will increase by 25 points. When the PTE variable reaches its maximum value, the BTS\_PTECheck Service will modify the value of the isEvolving variable, which will alter the flow of the tree.

The first thing the agent will do will be to return to its initial room, which we have already achieved previously, but now we must use it in another part of the tree and not in the behavior to restore energy. For this purpose, Unreal Engine has a type of nodes called Run Behavior, which allow us to execute a sub tree through a single node, we only need another behavior tree apart from the main one to specify it in the node. For the sub tree, it will first execute the node that we have just seen, then, once the agent arrives at its destination, it will execute the task BTT\_Evolve, wait 5 seconds and finally we will modify the value of the variable isEvolving to false. Figure 4.59 shows the resulting sub tree.

Now that our agent is able to change state, let's see how to make its behavior change for the same stimulus. For this, we will use the Services that we have created that will modify the variables AIPTESTate, AIIntState and AIBloodlustState every time they reach their maximum value. For the patrol, we will check the value of the AIPTESTate

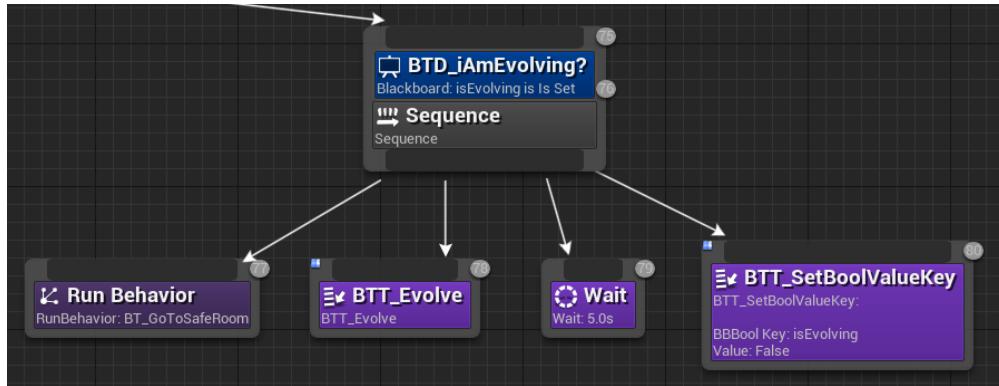


Figure 4.59: Evolution sub tree

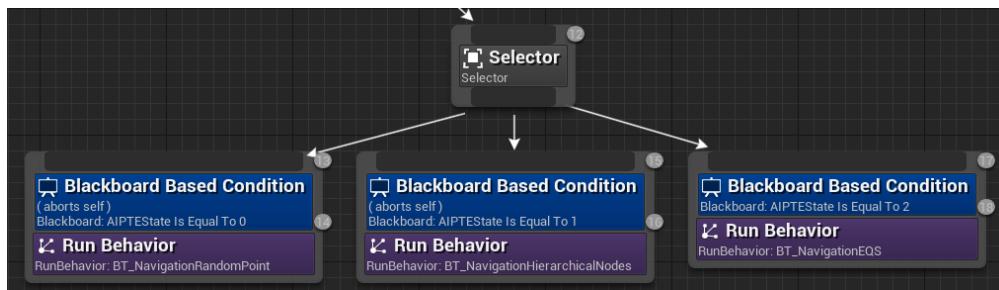


Figure 4.60: The different behaviors for the agent patrol

variable and depending on its value the agent will execute one behavior or another (see Figure 4.60).

When the agent evolves for the first time, it will be able to chase the player and react when the agent interacts with a locker in front of it. The way they react to the player's interaction with the locker will depend on the values of their intelligence and bloodlust statistics. For the agent to modify the way he reacts to the player entering a locker, it is not enough just to check the value of the intelligence variable, but the agent must first see how the player enters and leaves a locker. To know if the agent has seen the player perform these actions, we will use the Boolean variables didPlayerEntersLocker and didPlayerLeavesLocker, and when one of the two situations occurs, the value of the corresponding variable will be changed to true, and once both have this value, the agent will be able to open the lockers. In addition, since interacting with the lockers the agent can kill the player, we decided that the value of the bloodlust variable would also be a factor to take into account for the change in behavior when the player enters the locker (see Figure 4.61).

Finally, we will do the same for when the agent is following the player. In this case, only the value of the bloodlust variable will suffice to check the level of aggressiveness of the agent (see Figure 4.62).

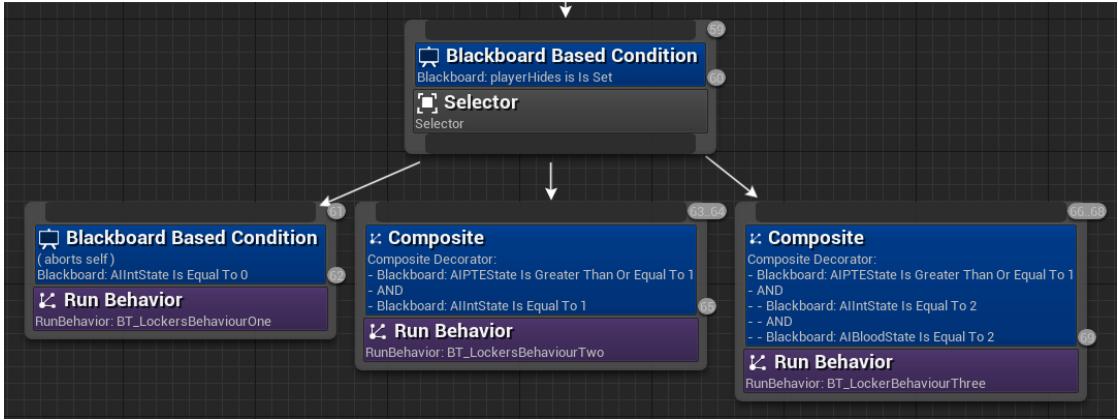


Figure 4.61: The different behaviors the reaction of the agent to the player interacting with the locker

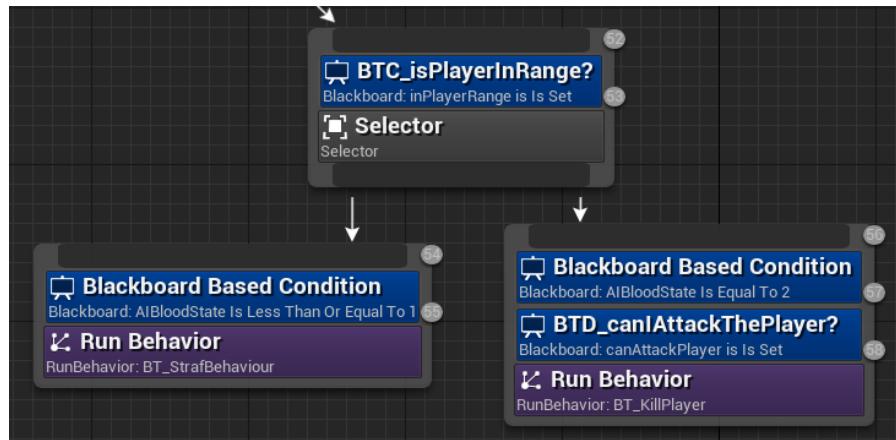


Figure 4.62: The different behaviors when the agent reaches the player

## 4.2 Results

After the completion of this project we have obtained an artificial intelligence based on a behavior tree. This behavior tree is composed of a total of thirteen different behaviors and twenty-seven tasks. This tree takes advantage of the rest of the artificial intelligence systems available in Unreal Engine to modify its flow and provide the agent with intelligence, allowing it to react to environmental stimuli in an unpredictable way depending on its state or other factors.

On the other hand, we went through a series of difficulties to implement the various states through which the agent can go through. However, we solved this difficulty by implementing a simple state machine, where its triggers for changes are based on the agent's own statistics. In addition, these states imply a level of difficulty for the player, being the first one an easy difficulty, the intermediate one a normal difficulty and the

final state a hard difficulty. In his initial state the agent is clumsy, his patrolling is not efficient and every time it sees the player it will be scared of him. In the intermediate state the agent will start patrolling more areas, and will be able to follow the player, and if he catches up with him he will start strafing him to intimidate him. In addition, if its intelligence reaches a certain level, it will be able to open the lockers to get the player out of them. And finally, in the final state, the agent will patrol the environment practically without retracing his steps, allowing him to cover more area, and if it manages to catch the player or opens a locker where he is hiding, it will kill him.

We have also implemented a system that was not planned for the agent. Using the AIPerception system the agent can perceive its environment, and also with a small configuration, we can specify whether the agent can remember and for how long the things it perceives. Through this, we have achieved that the agent can perceive food that is scattered around the environment, and after a few seconds it forgets having seen it. In this way, if it needs to eat and remembers having seen food previously, the agent can go to it directly without the need to look for it.

The main application of this artificial intelligence, which is how it has been conceived, is for a horror video game. However, some of the things that have been implemented for this artificial intelligence can also be used in other types of video games:

- The light perception of the player's flashlight can be applied to stealth video games.
- Patrolling via Environment Query System and the hierarchical pathfinding, can be applied to multi player or single player shooters and stealth video games.
- The searching for food behavior can be applied to intelligent life simulation games. And if we change the food for ammunition we can apply it to shooters.
- The strafing behavior can be applied to adventure/action video games and shooters.
- The structure and data table can be applied to role-playing games.

The project can be found in the following link: <https://github.com/miguelmm95/FDW>



CHAPTER



## CONCLUSIONS AND FUTURE WORK

### Contents

---

5.1 Conclusions . . . . .	77
5.2 Future work . . . . .	78

---

In this chapter, the conclusions of the work, as well as its future extensions are shown.

### 5.1 Conclusions

When I started the development of this work, I was not too sure I was going to get something I was happy with, and learning a new engine from scratch and against time did not help that insecurity. Fortunately, the learning process was not very complex and after a few weeks I was able to defend myself a little bit. It was worth it to have a hard time at the beginning, because thanks to the decision to learn Unreal Engine for my final degree work, I was able to find a company to do my internship that required programming skills in Unreal Engine. On the other hand, I have also been able to see the fruits of my time at college, and how I have been able to apply much of my acquired knowledge to a new engine without the need to seek a lot of outside help.

Unreal Engine is an excellent game engine, quite simple to use and with a little effort it gives you amazing results. Implementing artificial intelligence in it is not an easy task, but once you get to know its various systems, and how they work together, you can obtain results that in other game engines would be very difficult to achieve. The artificial intelligence that I have developed is something that at least surpassed the standards of many horror video games that are currently on the video game market, but it is still far from perfect. I am a bit sad that I could not implement reinforcement

learning in the main project, but seeing that I was against the clock and that I still achieved a good result, it does not bother me that much.

Thanks to this project I have realised that I love artificial intelligence and that I still have a lot to learn, so I will continue to study and improve to be able to better translate my ideas.

## 5.2 Future work

There is still a lot to work on my artificial intelligence. The first thing would be to polish certain things of this artificial intelligence, such as the evolution and the statistics system. On the other hand, I also want to add the sense of hearing so that the agent can hear the player's footsteps and other noises in the environment. I also want to keep digging more into the Environment Query System, as I know I can get more potential out of it and implement more behaviours that will improve the agent. On the other hand, Epic Games has announced that for the 5.3 version of Unreal Engine they are going to add nodes for machine learning, so when this version is released it will be a good time to add reinforcement learning without the need of third party plugins. Also, when I get happy with the finished artificial intelligence, I would like to publish it as an asset in the Unreal Engine's marketplace. And of course, I also want to develop a horror video game to use this artificial intelligence.

## BIBLIOGRAPHY

- [1] Custom “Game State” never works. <https://forums.unrealengine.com/t/custom-game-state-never-works/91931>, 5 2017.
- [2] Aaron Krumins. Mindmaker Machine Learning AI Plugin. <https://www.unrealengine.com/marketplace/en-US/product/mindmaker-ai-plugin>, 11 2020.
- [3] Carlos Coronado. Desarrollo de juegos con Unreal Engine de 0 a profesional. <https://www.udemy.com/course/desarrollo-de-juegos-con-unreal-engine-4-de-0-a-profesional/>.
- [4] Xiao Cui and Hao Shi. A\*-based pathfinding in modern computer games. 11, 11 2010.
- [5] Le Duc, Amandeep Sidhu, and Narendra Chaudhari. Hierarchical pathfinding and ai-based learning approach in strategy game design. *International Journal of Computer Games Technology*, 2008, 01 2008.
- [6] ENXGMA. Does Mr. X Teleport? | Resident Evil 2 Remake (Out-Of-Bounds Discovery). [https://www.youtube.com/watch?v=0tg00-u5jN0&ab\\_channel=ENXGMA](https://www.youtube.com/watch?v=0tg00-u5jN0&ab_channel=ENXGMA), 03 2019.
- [7] Epic Games. AI Controllers. <https://docs.unrealengine.com/5.0/en-US/ai-controllers-in-unreal-engine/>.
- [8] Epic Games. AIPerception. <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/AIPerception/>.
- [9] Epic Games. Behavior Tree Overview. <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/BehaviorTreesOverview/>.
- [10] Epic Games. Blueprint Interface. <https://docs.unrealengine.com/5.0/en-US/blueprint-interface-in-unreal-engine/>.
- [11] Epic Games. Controllers. <https://docs.unrealengine.com/5.0/en-US/controllers-in-unreal-engine/>.

- [12] Epic Games. Data Driven Gameplay Elements. <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/DataDriven/>.
- [13] Epic Games. Environment Query System. <https://docs.unrealengine.com/5.0/en-US/environment-query-system-in-unreal-engine/>.
- [14] Epic Games. EQS Node Reference: Generators. <https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/EQS/EQSNodereferenceGenerators/>.
- [15] Epic Games. EQS Node Reference: Tests. <https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/EQS/EQSNodereferenceEQSNodereferenceTests/>.
- [16] Epic Games. Hardware and Software Specifications. <https://docs.unrealengine.com/5.1/en-US/hardware-and-software-specifications-for-unreal-engine/>.
- [17] Epic Games. Struct Variables in Blueprints. <https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Variables/Structs>.
- [18] Epic Games. Unreal Engine's Marketplace. <https://www.unrealengine.com/marketplace/en-US/store>.
- [19] Epic Games. Unreal Engine's Youtube channel. <https://www.youtube.com/@UnrealEngine>.
- [20] Epic Games. Using a Single Line Trace (Raycast) by Channel. <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Tracing/HowTo/SingleLineTraceByChannel/>.
- [21] Evgeny Obedkov. 5 game studios that switched to Unreal Engine for flexibility and new hiring opportunities. <https://gameworkdobserver.com/2022/03/22/5-game-studios-that-switched-to-unreal-engine-for-flexibility-and-new-hiring-opportunities>, 03 2022.
- [22] GanttProject. <https://www.ganttpoint.biz/>.
- [23] Incanta Games. Blueprint FileSDK. <https://www.unrealengine.com/marketplace/en-US/product/blueprint-file-sdk>, 11 2020.
- [24] Indeed Editorial Team. 9 Nonfunctional Requirements Examples. <https://www.indeed.com/career-advice/career-development/non-functional-requirements-examples>.
- [25] itch.io. About itch.io. <https://itch.io/docs/general/about>.
- [26] itch.io. Top games tagger horror - itch.io. <https://itch.io/games/tag-horror>.

- [27] Itemis. What is a state machine? [https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview\\_what\\_are\\_state\\_machines](https://www.itemis.com/en/products/itemis-create/documentation/user-guide/overview_what_are_state_machines).
- [28] Jobted. Sueldo del Programador de Videojuegos en España. <https://www.jobted.es/salario/programador-videojuegos>.
- [29] Joel Spolsky (Atlassian). Trello. <https://trello.com/>.
- [30] Joseph Yaden. Resident Evil 4 Surpasses 4 Million Copies Sold In Two Weeks. <https://www.gamespot.com/articles/resident-evil-4-surpasses-4-million-copies-sold-in-two-weeks/1100-6513061/>, 4 2023.
- [31] Eric Lancheres. *Fragging Fundamentals*. Eric Lancheres; 1.0.2 edition, 02 2009.
- [32] Lucid. Lucidchart. <https://www.lucidchart.com/pages>.
- [33] Nikolai Baskin Sokolov. Unreal Engine AI vs Unity AI. <https://gamedev.gg/unreal-ai-vs-unity-ai/>, 09 2021.
- [34] Overleaf. <https://es.overleaf.com>.
- [35] Aidan Perry. Unreal Engine: Ultimate Survival Horror Course. <https://www.udemy.com/course/unreal-engine-ultimate-survival-horror-course/>.
- [36] Playstation Blog. Silent Hill 2 remake revealed, first gameplay details and design changes announced. <https://blog.playstation.com/2022/10/19/silent-hill-2-remake-revealed-first-gameplay-details-and-design-changes-announced/>, 10 2022.
- [37] Yoones Sekhavat. Behavior trees for computer games. *International Journal on Artificial Intelligence Tools*, 26, 01 2017.
- [38] TechBoomers. What is Udemy and How Does It Work? <https://techboomers.com/t/what-is-udemy>, 03 2022.