Comparison of Different Artificial Intelligence Techniques Applied in a Multiplayer Shooter

Final project of video games' design and development degree

AUTHOR: ELISABET LARGO IGLESIAS

TUTOR: RAÚL MONTOLIU COLÁS

July 3, 2017

UNIVERSITAT JAUME I

Abstract

Lately, video games sector is experiencing an exponential growth based on the appearance of a great quantity of new multiplayer video games. In fact, this feature has turn basically into a requirement. Most video games provide a story mode in which all narrative action is performed and in which the player learns the mechanics and learn how to master the controls and a multiplayer mode, in which players carry out the playful or ludic action based on the competition and the showing of their skills. This duality may be caused by the fact that in multiplayer mode, players match up each other without the intervention of any agent controlled by an artificial intelligence. This supposes a greater challenge due to the fact that, usually, is very common to determine the patterns with which an enemy has been built after a few attempts to beat it. Once a player has discovered its behavior, the complexity of the battle is drastically reduced, and so, the fun degree.

This paper constitutes the memory of the Final Project in the Game Design and Development degree and proposes a solution for this existent problem, by means of the design and implementation of machine learning techniques, specifically, reinforcement learning. With this solution, NPCs (non-playable characters) are able to learn from the player's actions and modify its behavior to provide a better experience to the gameplay.

In this project, two different types of enemies have been developed with Unreal Engine 4 for a shooter video game called *Hive: Altenum Wars*, which is expected to be released in a few months. On the one hand, there are the agents built up with predefined rule-based artificial intelligence techniques, specifically, behavior trees. On the other hand, analogous agents have been developed based on reinforcement learning to provide them the ability to adapt their behavior to the player's gaming experience.

Keywords: *Artificial Intelligence, Machine Learning, Reinforcement Learning, Q-Learning*

Contents

List of Figures

List of Tables

Nomenclature

BB	Blackboard
DFT	Depth First Traversal
FSM	Finite State Machines
IDE	Integrated Development Environment
LVM	Latent Variable Models
MC	Monte Carlo
ML	Machine Learning
NN	Neuronal Networks
NPC	Non Playable Character
RL	Reinforcement Learning
TD	Temporal Difference
UE4	Unreal Engine 4

Videos

SpeedersBT
https://youtu.be/bR2vavfzBPQ

SpeedersRL_RperActionState
https://youtu.be/ajl2leogEYw

DyggersBT
https://youtu.be/-GlmJPPx5fY

DyggersRL
https://youtu.be/vmbGEQb19Uc

Chapter 1

1. Technical Proposal

This Section presents an overview of the project, including the motivation of the research. This summary lists the principal objectives to be fulfilled during the elaboration of the project together with the planning associated and tools employed.

1.1 Introduction

Nowadays, multiplayer games are in an exponential growth period. In fact, this feature is practically a requirement in video games destined to a massive audience. It can be found a large variety of multiplayer games according to their genre, from the widespread shooters such as *Overwatch* [14] (Blizzard) to casual games for mobile devices as *Clash Royale* [4] (Supercell).

Commonly, these games are constituted by a story mode in which the narrative action is elaborated, and a multiplayer mode, in which players develop their skills with greater dexterity and interact between them. This duality is caused by the fact that enemies in story mode are established by patterns, which are easily predicted by the players after various attempts to beat them, so the confrontation between the human and the machine stops being a challenge. This project is born by the necessity of the creation of autonomous agents capable of modifying their behavior to adapt themselves to the player's actions in order to be reactive and less predictable. The proposed solution is to apply machine learning techniques, so the NPCs could be trained with information extracted from the game session. decision-making would be based on the skill level of the player and the state of the game. For this project, two different types of NPCs are going to be designed and programmed with Unreal 4 Engine: ones with a *classic* behavior, based on predefined rules and others capable of learning by themselves due to the application of machine learning techniques, in particular, reinforcement learning.

This NPCs are going to be included in a MOBA (Multiplayer Online Battle Arena) shooter named *Hive: Altenum Wars* [2] which will be released in a couple of months. Its gameplay consists of battles of two teams with 5 players each one. These players control a character with different weaponry and abilities, including special attacks bound to the type of the character. The idea of applying machine learning techniques is due to the fact that in the first months after the launch of a game, there is not a large amount of players, so there is the possibility that a game could not be created, because the number of players at that moment is not enough to make a team. This problem can be solved by introducing NPCs to the teams.

Furthermore, when a someone plays a game for the first time, it is essential that he or she had available a suitable training mode in order to avoid the issues which happen when the matchmaking system makes unbalanced teams, that is when the algorithm selects users with different experience to play the same battle. In this cases, the most probable option is that the player could not enjoy his first experience, existing the possibility that he leaves the game and would not play it anymore. To prevent this from happening the second part of this project is proposed, the creation of enemies which could learn by themselves and be capable of modifying their actions so that the learning curve of the player could be as most satisfying as possible.

1.2 Related subjects and justification

In this Section, a justification of the project is presented relative to different competences obtained during the study of specific subjects taught in the video games' design and development degree.

1.2.1 VJ1203, VJ1208 - Programming I, II

Programming is a fundamental aspect which is learned in different subjects throughout this bachelor's degree. Programming skills such as problem solving, algorithm understanding and translation of algorithms expressed in natural languages into specific functions are taught in both subjects. As this proposal consists of designing and implementing different NPCs, programming is the core of the project, so proficiency in all these aspects is indispensable and justified with the skills acquired during the degree.

Applied competences:

- IB03 - Capacity to understand and domain the basic concepts of discrete maths, logic, algorithmic and computational complexity, and their application for the resolution of problems in the field of engineering.
- IB04 - Basic knowledge about the use and programming of computers, operating systems, databases and computer programs with applications in the field of engineering.

1.2.2 VJ1231 - Artificial intelligence

This project is oriented to program different intelligent agents, so is essential to know how artificial intelligence works and the keys and limitations of different techniques to program rule-based NPCs, such as decision trees, behavior trees, state machines, etc. to achieve the best goals in this project.

Applied competences:

- IR06 - Knowledge and application of basic algorithmic procedures to design solutions to problems, analyzing the suitability and complexity of the proposed algorithms.
- IR15 - Knowledge and application of fundamental principles and basic techniques of intelligent systems.

1.2.3 VJ1234 - Advanced interaction techniques

One of the aims of this subject was to explain different techniques of machine learning, so it has been a useful introduction to be able to go into detail in this area of artificial intelligence. This aspect is a fundamental basis of the research addressed in this paper.

1.2.4 VJ1227 - Game Engines

Although this subject focused on the use of Unity, the principles and inner functioning of a game engine can be applied to Unreal Engine 4, as well. Auto-didactic capacity was enhanced with this subject and it is an essential generic skill which has been applied in the learning process of Unreal Engine 4, during my stay in *Catness*, company developer of *Hive*.

Applied competences:

- E12 - Capacity to evaluate, manage and understand game engines.
- G09 - Autonomous learning.
- IR07 - Knowledge, design and utilization of the most suitable data structures to problem solving.

1.2.5 VJ1224 - Software engineering

At this subject, project planning has been learned together with class design techniques, use cases, activity diagrams, etc.
Applied competences:

- G01 - Ability to analyze and synthesize.
- IR01 - Ability to design, develop, select and evaluate applications and computer systems and ensure its own reliability, security and quality.
- IR02 - Ability to plan projects and computer systems in the video games scope.

1.2.6 VJ1215 - Algorithms and data structures

This subject teaches different complex algorithms and data structures and provides knowledge about their performance, such as their computational cost.
Applied competences:

- IB01 - Capacity to solve mathematical problems that can appear in the engineering.
- IB03 - To use maps and understand its implementation by means of trees.

1.3 Tools

This Section lists the tools that will be used during the design and development of this application, grouped by its function.

1.3.1 Programming

1. **Unreal Engine 4**
   This is the game engine employed to develop the video game *Hive*, in which the results of the agents will be tested. It allows the combination of visual scripting in blueprints and C++ scripting. This provides a wide flexibility to create complex behaviors for the agents.
2. **Visual Studio**
   It is the main integrated development environment (IDE) used to program scripts in C++. The reason to use it is that it is considered the best IDE for UE4 due to the fact that its building system is based around it.

1.3.2 Debugging, Testing and Profiling

1. **Unreal Engine 4 Developer Tools**
   Unreal Engine 4 provides several information gathering tools such as debuggers, analyzers and profilers.

1.3.3 Documents

1. **Overleaf**
   Overleaf is an online LaTeX and Rich Text tool.
2. **Google Slides**
   It is an online application for creating and editing slide show presentations.
3. **Creately**
   This is a free online tool to make different diagrams. In this case, it has been used to create the UML diagrams.

1.3.4 Version Control System

1. **Git**
   It is a free and open source distributed version control system which handles large projects with efficiency.
2. **GitHub**
   It is the repository hosting service used to manage the project.

1.3.5 Video

1. **Action!**
   This tool captures any area of the screen. Its use its justified due to it is very intuitive and offers several options for professional screen capture.
2. **Adobe Premiere Pro**
   It is a powerful video editing application.

1.4 Objectives

The main objective of this project is to compare different techniques from rule-based systems to machine learning methods to develop intelligent agents while implementing them to create NPCs which simulate a human behavior.

This large objective can be broken down in the implementation of behavior trees and Q-learning, with the purpose of being compared, testing both in a shooter video game called *Hive* in order to determine the viability of implementing machine learning in video games. This analysis will indicate if machine learning would solve all the problems mentioned previously related to the ease of discovering the patterns in which the enemies are built around.

In fact, this goal includes the creation of different NPCs capable of interacting with their environment and with other agents and the development of NPCs capable of modifying and adapting their behavior to the player skills during the gaming session which means developing an online training.

Concretely, all the objectives can be listed in the following way:

- Comparison of the different artificial intelligence techniques.
- Implementation of artificial intelligence techniques to create NPCs which simulate a human behavior.
- Creation of NPCs capable to interact with different agents and environment elements.
- Development of enemies which modify their behavior to adapt to the player skills and his playing style.

1.5 Project Planning

1.5.1 Project Schedule

This Section presents all the phases and its planning for the development of two intelligent agents, both including two versions, one rule-based and other with machine learning. The totality of the project can be divided into five different phases, each one containing a different set of tasks. Figure 1.1 summarizes the contents of this Section in a Gantt chart.

- **Phase 1 - Documentation**
  All the project is documented in different papers, including a technical proposal, a technical memory and a presentation which contains different videos for the defense before the evaluation court. The planning of this phase is shown in Table 1.1.

Table 1.1: Documentation phase - Tasks Breakdown

ID	Task	Period	Hours
TPC	Technical Proposal Courses	30-01-2017 to 02-02-2017	8
TP	Technical Proposal	07-02-2017 to 10-02-2017	6
TM	Technical Memory	20-05-2017 to 11-06-2017	40
PDV	Project Defense Video	12-06-2017	2
PDP	Project Defense Presentation	13-06-2017	4
Total Hours	60		

- **Phase 2 - Research**
  As this project is a comparison between different artificial intelligence techniques, a thoughtful research is essential, specially, a study in machine learning methods to select the most suitable one for this project. In total, 60 hours should be spent in this research. Its breakdown is shown in Table 1.2.

Table 1.2: Research phase - Tasks Breakdown

ID	Task	Period	Hours
BT	Behavior Trees and UE4 AI System	13-02-2017 to 18-02-2017	10
ML	Machine Learning Techniques	19-02-2017 to 05-03-2017	40
MLV	Machine Learning and video games	19-02-2017 to 05-03-2017	10
Total Hours	60		

- **Phase 3 - Design**
  Once the appropriate method of machine learning is selected, the design of the different NPCs takes part. This phase can be divided into 3 sub-phases for each enemy to develop as shown in Table 1.3. First, the conceptual design of two agents will be carried out. Next, it is time to the design of the rule system based version and to design the proper machine learning method.

- **Phase 4 - Development**
  The implementation of each of the sub-phases designed in the previous Section and a system to gather information about the game state so the agents can recognize it and a mechanism to save that information in a server. Table 1.4 shows a breakdown of the development phase.

Table 1.3: Design phase - Tasks Breakdown

ID	**Task**	**Period**	**Hours**
CD1	Conceptual Design enemy 1	06-03-2017 to 07-03-2017	4
CD2	Conceptual Design enemy 2	07-03-2017 to 08-03-2017	4
R1	rule-based enemy 1	09-03-2017 to 11-03-2017	10
R2	rule-based enemy 2	12-03-2017 to 15-03-2017	15
ML1	Machine Learning enemy 1	19-03-2017 to 23-03-2017	15
ML2	Machine Learning enemy 2	24-03-2017 to 05-04-2017	20
Total Hours	68		

Table 1.4: Development phase - Tasks Breakdown

ID	**Task**	**Period**	**Hours**
IR1	Implementation rule-based enemy 1	06-04-2017 to 09-04-2017	15
IR2	Implementation rule-based enemy 2	10-04-2017 to 17-04-2017	25
SGS	System to get game state	27-04-2017 to 29-04-2017	10
IML1	Implementation Machine Learning enemy 1	30-04-2017 to 06-05-2017	20
IML2	Implementation Machine Learning enemy 2	06-05-2017 to 13-05-2017	30
S	Saving information in server	20-05-2017 to 25-05-2017	15
Total Hours	120		

- **Phase 5 - Testing and Results**
  Once, the agents are developed, they will be tested in the video game *Hive*, which is under development for the moment. At this phase, different experiments will be performed in order to compare the results of the different techniques implemented. Table 1.5 shows the planning for this phase.

Table 1.5: Results phase - Tasks Breakdown

ID	**Task**	**Period**	**Hours**
TR1	Testing rule-based enemy 1	15-05-2017	3
TR2	Testing rule-based enemy 2	16-05-2017	3
TML1	Testing Machine Learning enemy 1	15-05-2017 to 16-05-2017	5
TML2	Testing Machine Learning enemy 2	16-05-2017 to 17-05-2017	5
RA	Results analysis	18-05-2017 to 19-05-2017	5
Total Hours	21		

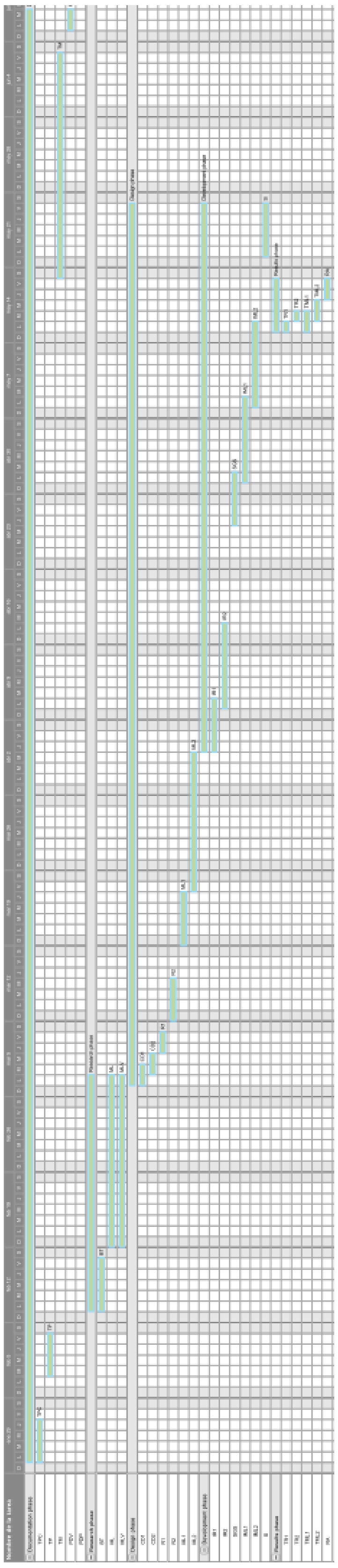
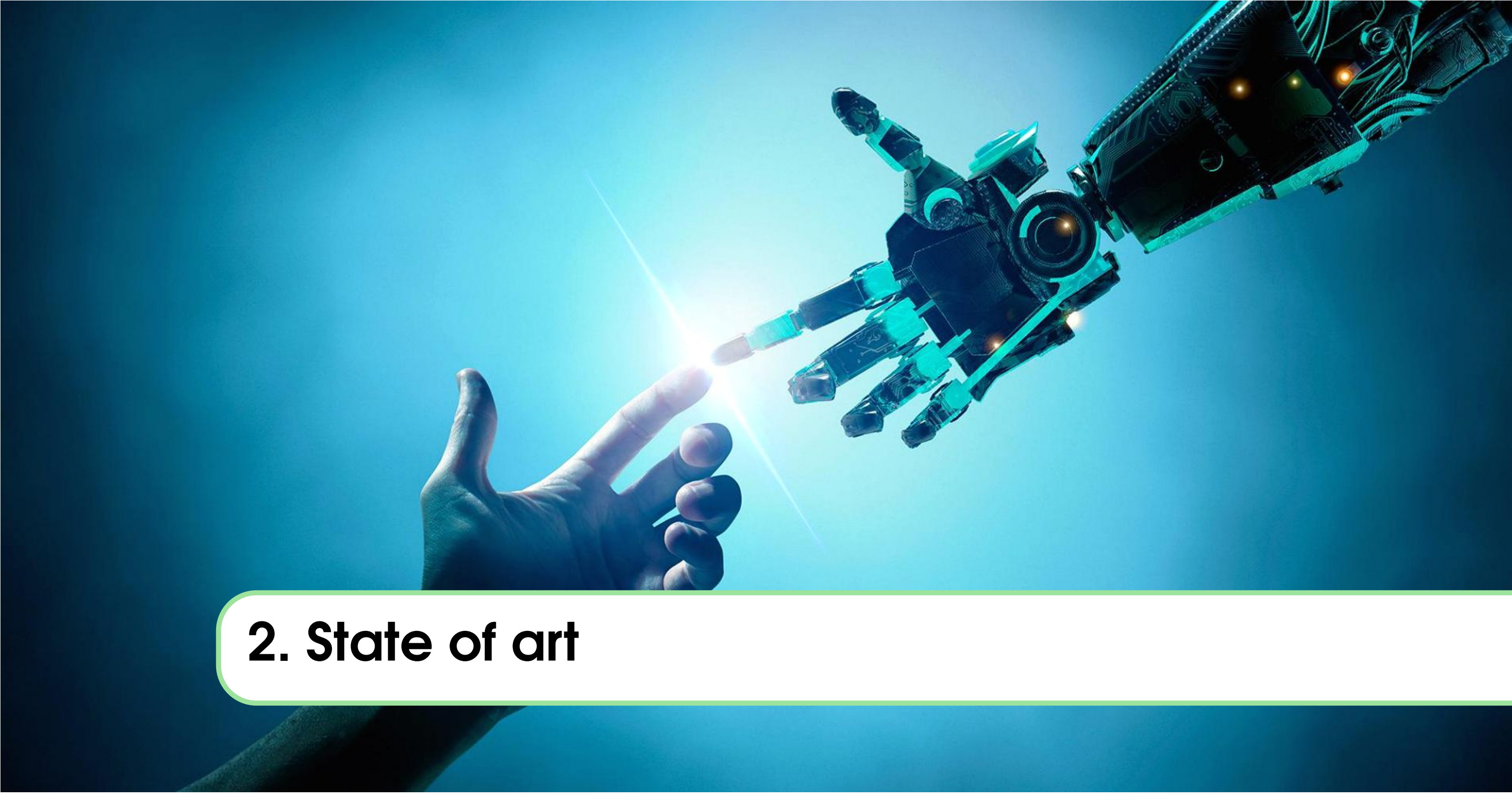


Figure 1.1: Project Gantt Diagram

1.6 Expected Results

With this project it is expected to design and develop different NPCs to be able to contrast results between *classic* artificial intelligence techniques and machine learning, so that this investigation could be a reference to decide if the application of machine learning techniques would be suitable for the development of a video game. In particular, it is expected to develop a rule-based behavior tree system and machine learning techniques, with a previous deep investigation about different algorithms.

Chapter 2



2. State of art

John McCarthy [11], one of the fathers of artificial intelligence (AI) defines it as:

"It is the science and engineering of making intelligent machines, especially intelligent computer programs."

On his behalf, Millington [**AIvideo games**] specifies that intelligent machines should be able to perform the thinking tasks that humans and animals are capable of. Even so, Stuart Russell and Peter Norvig [19] defend that AI concept can be defined around four categories, shown in Table 2.1, combining the reasoning and behavior processes dimension with the fidelity to human acting and a perception of ideal rationality.

Having these definitions of AI in mind, as video games developers, it has to be differentiated academic AI and game AI. The first one tries to solve a problem optimally with less emphasis on hardware and time limitations, so it is flexible in terms of performance. On the other hand, game programmers have to work with limited resources and the real goal is to simulate intelligent behavior. In addition, video games are a battlefield for academic AI, due to the fact that advancements in research might be tested there.

Since the 90s, progress in AI is so immense that the leap from classical rule-based procedures to systems which can learn by themselves, has been taken. This is the point in which this project is focused, due to the fact that it deals with a comparison between techniques of both milestones for developing the decision-making of an NPC for a shooter video game.

The explanation above justifies the following Sections. At first, definitions of classical AI techniques and Machine Learning techniques are provided. Secondly, state of art of ML in video games is summarized.

2.1 Classical Artificial Intelligence Techniques

This division refers to systems which are mainly based on rules. It includes the majority of techniques used at this moment to develop autonomous agents for video games. Despite the existence of a wide variety of techniques, behavior trees (BT) have been selected for the development of the decision-making in this project, due to the fact that they are a very powerful, flexible and intuitive methods to create complex behaviors.

Table 2.1: AI definitions according to the four categories by Stuart Russell and Peter Norvig [19].

	Humans	**Rationally**
Think like	"The exciting new effort to make computers think ... machines with minds, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..."(Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
Act like	"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 199 1)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Specifically, BT have a few advantages over finite state machines (FSM): they provide lots of flexibility, are very powerful, and they are really easy to make changes to. Nevertheless, they do not replace the functionality of FSM. This is the reason why they should be combined to achieve robust agents. In addition, it is easier to create a BT that will adapt to all sorts of situations whereas it would take a lot of states and transitions with a FSM in order to have similar behavior.

As stated above, one final BT advantage is that they are really intuitive and easy to make changes to. Lots of different visual editors for BT creation have appeared, so it seems reasonable that they have grown in popularity for modeling the artificial intelligence in computer games such as *Halo* [10] and *Spore* [9]. Using this tools, changing the priority of the tasks consists of dragging them from one branch to another. On the contrary, changing it in a FSM involves changing the transitions between states. Also, it is really easy to completely change how the AI reacts to different situations just by changing the task's priority or adding a new parent task to a branch of tasks.

Behavior trees consist of a tree of hierarchical nodes that control the flow of decision-making of an AI entity. At the leaves, commands that control the agent are placed and forming the branches are various types of utility nodes that control the AI's walk down the trees to reach the sequences of commands best suited to the situation.

The trees can be extremely deep, with nodes calling sub-trees which perform particular functions, so it allows to combine complex behaviors. Development is highly iterative because the programmer can start creating a basic behavior and then, adding new branches ordered by their desirability, allowing the AI to have backup plans against a particular behavior fail.

According to execution, BT are depth-first traversal (Figure 2.1). This means that the first time they are evaluated or when they are reset, they start from the root and explore as far as possible along each branch before backtracking, checking the conditions of the nodes. Each child is evaluated from left to right, so they are ordered based on their priority. If all of a child node's conditions are met, its behavior is started. The tree is then reset and ready to go again. When a node starts a behavior, that node is set to 'running' and it returns the behavior. The nodes that failed or completed are returned to 'Ready'. Then the next time it checks, it starts again with the highest priority node.

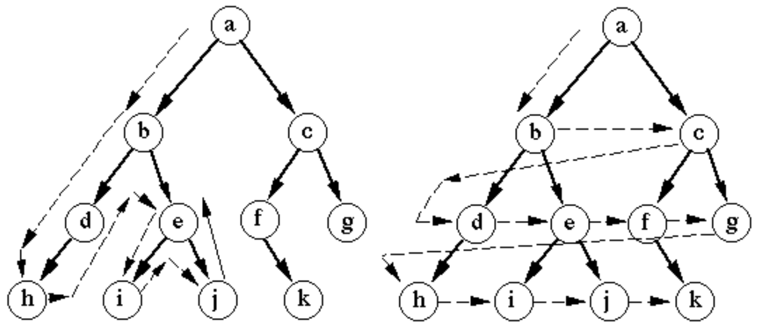


Figure 2.1: Depth-first search VS Breadth-first search

If any condition fails, the traversal returns to the parent and then moves on to the next priority child.

2.1.1 Unreal Engine 4 Artificial Intelligence System

As this project is developed in Unreal Engine 4 [6], its AI System based in a Behavior Tree Editor has been used. Unreal Engine 4 BT consist of different nodes: tasks, composites, decorators and services.

- **Tasks**
  They are the actions the agent can perform. They are placed at the bottom part of the BT, called leaves. Unreal Engine 4 provides different ready-to-use tasks such as *MoveTo*, *PlaySound*, *Wait*, etc.
- **Composites**
  These nodes define the rules for how the branch is executed and they can have decorators attached to them to modify the execution flow. Three types of composites are provided by the UE4 BT Editor: selectors, sequences and simple parallels. The first ones, execute their children from left to right and stop executing its children when one of their children succeeds. If a selector's child succeeds, the selector succeeds and if all the selector's children fail, the selector fails. Sequences can be seen as the opposite nodes of the mentioned before, due to the fact that they stop the execution of the branch when one of their children fail. When a child succeeds, the children in the right are executed too and if all the children succeed, then the sequence succeeds. Finally, simple parallel nodes allow to perform two tasks at the same time, normally, the main task and a secondary sub-tree or task. When the main task finishes, the property Finish Mode is set, dictating if the node should finish immediately, aborting the secondary tree or if it should delay for the secondary tree to finish. This node type is very useful in cases when it is desired the agent to perform different actions such attacking while it is moving, for instance.
- **Decorators**
  They are commonly known as conditionals and are attached to other nodes such as composites or tasks. UE4 Editor offers different decorators such as getting values from the blackboard, conditional loops, etc., but it also offers the possibility to implement custom made conditionals.
- **Services**
  These are special nodes which execute at their own defined frequency, which can be modified by the user. Their common use is to update the blackboard values.

Also, it is necessary to mark the differences between this BT and the conventional ones, because there are different critical ways in which the UE4 implementation of BT differs from standard BT:

- **Event-Driven**
  UE4 BT is event-driven, what means that they avoid doing lots of work each frame, because

they just wait for events which make changes in the tree, instead of being constantly checking if a relevant change has occurred. Apart from being an advantage in the performance of the trees, this feature makes visual debugging easier because the BT visual editor only shows the differences in the execution.

- **Conditionals are not leaves**
  In the standard BT, conditionals are task leaf nodes, which simply do not do anything else than succeeding or fail, but UE4 BT system uses conditionals as decorators which provide different advantages. Firstly, conditional decorators make the BT visual editor more intuitive and easier to read. Also, since all leaves are action tasks, it is easier to see what actual actions are being ordered.

2.2 Machine Learning Techniques

In this subsection, firstly, part of the content of the subject VJ1234 related to Machine Learning and ML types is summarized. After that, a deep research in Reinforcement Learning techniques will be presented, to justify the next Section about the design of our agents.

In 1959, an American pioneer in the field of computer gaming and artificial intelligence called Arthur Samuel, coined the term "machine learning" and defined it as:

"giving computers the ability to learn without being explicitly programmed." [12]

In particular, ML is defined as a set of methods that can automatically recognize patterns in data and then, use the uncovered patterns to predict future data or to perform other kinds of decision-making under uncertainty. [13]

Arthur Samuel developed a Checkers-playing Program which is considered as the world's first self-learning program. [17] He was the first person to propose and implement a learning method that included temporal-difference ideas, a technique that will be explained afterwards.

In machine learning, predictions are data-driven and probability theory is applied to solve this uncertainty related problems. This is the reason why the probabilistic approach to ML is closely related to the field of statistics, but differs slightly in terms of its emphasis and terminology. [21]

Machine learning can be divided into three different areas: predictive, descriptive and reinforcement learning. Despite some literature describes reinforcement learning as a specific type of descriptive learning [16], it will remain in a separated Section for the purpose of making a deep research about that concept.

2.2.1 Predictive Learning

Predictive or supervised learning is the form of ML most widely used today. It is based it the ability to learn a mapping from inputs x to outputs y, given a labeled set of input-output pairs:

$D = \{(x_i, y_i)\} N_i = 1$, [13]

where D is called the training set and N consists of the number of training examples.

Inputs x are called features and they are a D-dimensional vector of numbers that represent the attributes. Furthermore, inputs can be complex objects such as images, sentences, messages, graphs, etc. Outputs y are a variable which the algorithm will try to predict. If there exists a finite set for that variable, it consists of a categorical problem, which is solved with classification. On the contrary, when y is a scalar, the problem is known as regression.

2.2.1.1 Classification

As said before, the goal of classification is to learn to map from inputs x to outputs y, where:

$y \in \{1, \ldots, C\}$ [13].

C is the number of classes of the variable that is going to be predicted. If there are only two classes for the problem, it is called binary classification, for instance, true or false, man or woman, etc. When C is greater than two, it is considered a multiclass problem. In these cases, normally, a single output is predicted, but it can occur that labels are not mutually exclusive; this is the case of multi-label classification.

Mapping is formalized as a function approximation. Considering $y = f(x)$ for some unknown function f. The goal of f is to make predictions given the discrete labeled training set.

Classification is probably the most widely used form of ML and has been used to solve many interesting and often difficult real-world problems such as classification of documents, email spam filtering, image classification and handwriting recognition or face detection and recognition, among others.

There exist a wide variety of classification algorithms, the most common are kNN, SVM, Naive Bayes, logistic regression and decision trees.

2.2.1.2 Regression

Regression is just like classification except the output variable is continuous. Some examples of real-world regression problems are predicting tomorrow's stock market price given current market conditions and other possible side information, predicting the temperature at any location inside a building using weather data, time, door sensors, etcetera [21].

2.2.2 Descriptive Learning

The second main type of machine learning is the descriptive or unsupervised approach, which does not require training samples. It only receives data and its goal is to find patterns which relate them, which in literature is known as knowledge discovery [13].

$D = \{x_i\}N_i{=}1$

Unsupervised learning does not calculate the desired output for each input, as supervised methods do. Instead, it calculates a density estimation in an iterative process. Also, it does not require a human expert to manually label the data.

One of the most popular examples of unsupervised learning is clustering.

2.2.2.1 Clustering

This technique consists of grouping different data in different collections called clusters, so that data in the same group are more similar to each other than to the other clusters.

When clustering is required to solve a problem few steps have to be taken into account. Firstly, depending on the data set, an appropriate algorithm has to be selected and its parameters, such as a number of clusters (K) or the distance function to use, have to be set. Then, it is time to proceed to the estimation of the cluster each point belongs to.

Clustering is, essentially, employed in data mining and analysis and it is used in many fields, including pattern recognition, image recognition, information retrieval, bioinformatics, data compression and computer graphics [13].

2.2.3 Reinforcement Learning

This approach is based in the theory [21] that humans and animals learn by interacting with their environment, so it defends that computers should do a similar procedure to acquire knowledge. This technique is much more focused on goal-directed learning from interaction than other approaches to ML are, and it bases the learning process in numerical reward and punishment signals.

Reinforcement learning consists of the decision of what to do to maximize the reward signal. This means that it relies on mapping situations to choose the best action in each moment. To discover so, the learner agent has to try the actions to contrast which of them generates the most reward, not only the immediate reward, but long-term bonuses. So, RL is constituted by two distinguishing characteristics: trial and error and delayed rewarding.

RL separates itself from other learning methods like Supervised Learning, which makes decisions based on previously experienced states provided by an intelligent and external supervisor. As data from previous experiments are not available for RL, an agent needs to be able to learn from its own experiences and not others, so it has to interact with its environment to achieve a goal which is related to the actual state. This means that at each state of the environment, a different action should be chosen to achieve the best reward possible.

As RL is based on learning- by doing and trial and error, a balance between exploration and exploitation [5] is a must. The first term, as its name indicates, is the act of selecting random actions to discover their impact on the environment in a specific state. This states the process of learning. On the contrary, exploitation is employing those learned experiences. Here, the agent prefers actions that it has tried in the past and were effective in producing rewards, but this step requires the discovery of those actions by means of exploration and this is the point where a balance is required: how to explore in order to make better action selections in the future. The problem here is obvious neither of these both actions can be performed exclusively, so they must be combined to provide a proper learning.

"One of the most important problems in machine learning—and life—is the exploration-exploitation dilemma. If you've found something that works, should you just keep doing it? Or is it better to try new things, knowing it could be a waste of time but also might lead to a better solution?" [7]
-Pedro Domingos, 2015

This exploitation-exploration dilemma has been studied by mathematicians for many decades. As this process relies on random determinations, it is considered a stochastic task, so many iterations in the decision of which action to perform it are needed to obtain a reliably expected reward.

As it has been said before, RL is a goal-directed technique and this is a key feature, because other approaches consider subproblems such as supervised learning does [21]. RL considers a complete goal.

In a Reinforcement Learning system, five important elements exist: a policy, a reward function, a value function, states space and an actions space [8].

2.2.3.1 Policy

The policy defines how the learning agent should behave at a specific time. It is a mapping from the actual state of the environment to an action.The policy can vary from a simple function to a complex algorithm, which involves a lot of calculations and searches. Sutton defines policies as:

It corresponds to what in psychology would be called a set of stimulus-response rules or associations [21].

The policy citeRLRTSis written as: $\Pi : s \rightarrow a$, where Π is the policy, S is all possible states of the environment, s is a specific state of S where s $\in$ S, A is all actions, and a is a specific action, where a $\in A$.

The policy is the core in RL an agent, due to the fact that it is the mechanism which decides the behavior of the agent.

There are three main policies [8] to select the desired behavior: ε-greedy, ε-soft and softmax.

- **ε-greedy**
  In this policy, the most of the time the action with the highest estimated reward (greediest action) is chosen, but, with a small probability ε, an exploration is taken into account to discover better results by picking random actions.
- **ε-soft**
  We can consider this policy as the contrary to ε-greedy. Exploitation is performed with probability 1-ε .
- **Softmax**
  Softmax assigns a rank or weight to each of the actions, according to their action-value estimate, so the worst actions are unlikely to be chosen.

2.2.3.2 Reward Function

The reward is a numerical value that represents the degree of desirability for an action in a specific state, so this mapping indicates how good is to take that action.

The reward function is written as: $R : s \rightarrow r$, where R is the reward function, s is a specific state being mapped into a specific reward r [**RLRTS**].

As the purpose of an RL agent is to maximize the total reward, it bases its selection in the reward function which determines which actions are good or bad to take immediately in a specific state. It is important to mark that the reward function remains the same after the learning process, it can not be changed by the agent, but it is used to modify the policy because if a policy determines to take a bad action, it must change to avoid picking it often in future.

2.2.3.3 Value Function

Despite evaluating which action is better to take for immediate results, as reward functions do, value functions consider the future states and their rewards, so they choose actions for long term benefits in a particular policy. This means that a value function provides the total amount of reward that an agent should expect in the future when it is in a specific state.

The value function can be written: U as: $s \rightarrow R$, where U is the value function, s is a specific state being mapped into a total amount of reward being R [**RLRTS**].

There are two type of value functions: $V\pi(s)$ and $Q\pi(s,a)$.

- $V\pi(s)$: represents the value when starting in the state s and following π policy.

- $Q\pi(s,a)$: this is the value of taking action a in state s under π policy.

To define these value functions, there are two methods: Monte Carlo Method and Temporal Difference Method.

- Monte Carlo Method requires experience from states, actions and rewards got by the interaction with the environment. This technique divides the experience into episodes, which will finish independently of which actions were taken. MC updates the values only when an episode is completed, getting the final reward and taking the path back from the final state to the first one. this is the reason why MC method is incremental in an episode-by-episode sense, but not in a step-by-step sense [21].
  MC method is expressed as:
  $V(s_t) \leftarrow V(s_t) + \alpha[R_t - V\ (s_t)]$ [8]

- Temporal Difference Method is a combination of Monte Carlo ideas and dynamic programming. TD methods update estimations based in part on other learned estimates, without waiting for an outcome. The difference with the method above is that TD estimate the value functions after each step. An estimate of the final reward is calculated at each state and the state-action value updated for every step. This reflects a more realistic assignment of rewards to actions compared to MC, which updates all actions at the end directly. [21]
  TD method is expressed as:
  $V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(st{+}1) - V(s_t)]$ [8],
  where r_{t+1} is the reward at time r_{t+1}.

In these functions, there are different parameters that are really important in the learning process:

- Learning Rate α: This parameter defines the proportion of the old estimation that will be replaced with the new value. If α is 0, the agent will not learn anything, because it relies on its decisions in the old estimation, which is 0 from the start. On the contrary, values near to 1 will completely change the previous value.

- Discount Factor γ: It defines what fraction of the new reward is taken into consideration for the updating. If γ is 0, the new rewards will be ignored and for values near to 1 the agent will consider the reward as equal weight as the learnt value.

- Exploration Rate ε: This parameter appears in the policy and it determines the proportion of exploration and exploitation.

2.2.3.4 State Space

The set of states S consists of all the states that can take part in the environment of the problem. In the next chapter, we will see that for a video game this can become really huge, so discretization of the state space is a key task to develop autonomous agents for video games by means of reinforcement learning.

2.2.3.5 Action Space

The set of all actions A the agent would be able to perform in the environment.

2.2.3.6 Markov Property

Previously, it has been set that in RL an agent bases its decisions in the environment's states, so it is essential to know what kind of information should provide the state signal, which is called the Markov property. This property states that a state signal should include immediate sensations, but at the same time they should not provide everything about the environment, only the useful information to take decisions. On the other hand, the information provided by a state signal must be something that the agent could have received by its sensations. Summarizing, a signal which can be accepted by Markov property is the one which represents relevant past sensations. Markov property dictates that if an environment model has this feature, given a state and an action, is possible to predict the next state, so this is a key in reinforcement learning problems.

2.2.3.7 Markov Decision Process

In 1960, Ronald A. Howard's published the book Dynamic Programming and Markov Processes [1], where he treated the Markov decision processes. when a RL problem satisfies the Markov property, it is called Markov decision process.

A Markov Decision Process is defined by the state and action sets and the dynamics of the environment. The probability of each possible next state are called transition probabilities.

2.2.3.8 SARSA

This algorithm for learning a Markov decision process policy works with an action-value called Q-value, Q(s,a), where s is a state and a, the best action possible for the state s. These values represent the possible reward received in the next state s' taking the action a in the state s. The name of this algorithm summarizes itself, because it is formed by the quintuple (s,a,r,s',a'), being r the reward associated to the next state s' and the best action a' for s'.

SARSA algorithm is called an on-policy learning algorithm, because the agent updates the policy π based on the interaction with the environment at the same time it calculates the Q-value. As shown in Figure 2.2, the Q-value is updated taking the learning rate α into account. This update is done after every state transition when the state is not the final state.

```
Initialize for all states $s \in S$ and actions $a \in A$ the value $Q(s,a)$
        to a constant value
Repeat for each episode:
        Initialize $s$
        Choose $a$ for $s$ using the policy $\pi$
        Repeat for each step in the episode, until $s$ is terminal:
                Execute action $a$ and save reward $r$ and next state $s'$
                Choose $a'$ for $s'$ using $\pi$
                $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
                $s = s'$ and $a = a'$
```

Figure 2.2: SARSA algorithm

2.2.3.9 Q-learning

Q-learning was introduced by Watkins [21] in 1989. This algorithm differs from SARSA by not using the action-value of the next state. It uses the value of the action with the highest value and Q-learning does not wait until a second action is chosen before updating the value. Furthermore, it is an off-policy algorithm, because it works independently of π; this means that the policy is given directly instead of obtaining by itself by means of the model parameters of a transition and reward functions.

As shown in Figure 2.3 Q-learning updates the Q-value of a state-action pair after the action has been taken and an immediate reward has been received.

Literature shows that Q-learning converges earlier than SARSA, because it utilizes a greedy method determining the action-value rather than selecting a random action with a certain probability [21].

```
Initialize for all $s \in S$ and $a \in A$ the value $Q(s,a)$ to a constant value
Repeat for each episode:
    Initialize $s$
    Repeat for each step in the episode, until $s$ is terminal:
        Choose $a$ for $s$ using $\pi$
        Execute action $a$ and save reward $r$ and next state $s'$
        $Q(s,a) = Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$
        $s = s'$
```

Figure 2.3: Q-learning algorithm

2.3 Video games and Machine Learning

After describing what machine learning is, its principles and some techniques, let's see how ML is being done in video games today.

First, ML has to be divided in two areas: off-line learning and on-line learning.

2.3.1 Off-line Learning

This type of ML refers to the techniques which are applied during the design phase of the video game, to find the best behavior for the agents and, then, the results of this techniques are applied before releasing the game, by means of other *traditional* techniques.

In this case, reinforcement learning is used during the development of several video games, specifically during the testing phase, to balance properly the enemies. Also, it is during this phase when the agents have the most information possible to learn. When the learning algorithms converge, developers find the strategies of those results, and then, hard-code them.

One example of a game with off-line learning is *City Conquest* [20] (Figure 2.4), a real time strategy game and tower defense, developed in 2011 by Intelligence Engine Design Systems. It is defined by its developers as:

"City Conquest *is an epic mobile strategy game that combines innovative strategy with classic tower defense gameplay. It evolves the core tower defense concept by adding an offensive gameplay element and territorial expansion and control mechanics..."*

During the development phase, they used genetic optimization techniques and evolutionary algorithms for gameplay balancing and design optimization. They developed a tool named *Evolver* [3] that played 1-2 million missions against itself per day, to find patterns and then balance the game elements such as buildings of buildings updates.

Figure 2.4: City conquest screenshot

2.3.2 On-line Learning

In this case, learning process happens during the game session, so it adjusts itself to the player. Reinforcement learning is hardly ever used for online learning due to two main reasons:

- **CPU resources**
  Usually, reinforcement learning tends to require a lot of CPU power. This issue is disappearing as technology advances so fast.
- **Control**
  Reinforcement learning puts the agents out of control for the programmers and this is the most drawback of implementing RL techniques in video games since it requires a shift in the video game's design paradigm. Game developers and players are used to agents that can not take unexpected paths, for instance, because they have not been programmed to it, but RL can break this rule.

Already in 2001, Lionhead Studios released a video game with on-line learning called *Black and White* (Figure 2.5). It used a variety of techniques, integrating them into an impressive overall AI system. Belief–desire–intention models, rule-based systems, decision trees and neural networks. [22] The video game was a god simulator, where the player teaches the creatures how to act, by interacting with them. It was based on the concept of good and evil, so the behavior and personality of the creatures were shaped by the player.

Although Belief-Desire-Intention is not a ML technique, it can be expanded to incorporate learning. The BDI technique is like a bunch of complex if-then-else sentences. In *Black and White*, BDI is used to determine if certain thresholds have been exceeded and thereby expressing the desire of the creature. The agent then reviews its desire and choose actions to satisfy them. Black and White used a simple feedback learning technique in order for the creature to learn which actions were the most appropriate to take.

Next summer, *Hello Neighbor* [15] will be released (Figure 2.6). It is a first-person tactical puzzle developed by Dynamic Pixels, which is programmed with machine learning. The agent observes the player's actions, learning where they succeed and where they make mistakes on any given level and he reacts to that information, so the more one plays, the smarter and harder the

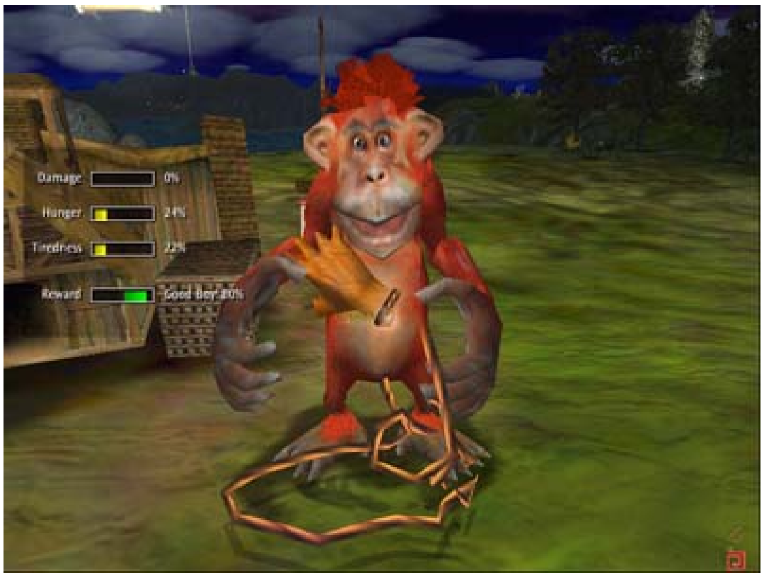


Figure 2.5: Black and White capture

enemy becomes.

Figure 2.6: Hello Neighbor

2.4 Hive: Altenum Wars

Hive [2] is a horizontal scroll shooter video game which is in development at the moment, by the video games studio *Catness*. This video game has two modes: multiplayer and survival.

Regarding the first mode, an online battle of two teams takes part. Each team is formed by five players, who control different characters. Depending on the character selected, the players can choose among a wide variety of weapons and special abilities, including bombs, lasers... In the survival mode, one single player or a team fight against hordes formed by different enemies, that will appear during the game, which lasts until the player dies. The scoring consists of as more time the player survives and more hordes defeat, the more score he will achieve.

One distinctive feature of this game before other video games of the same genre is the environment. The maps, settled into a futuristic atmosphere, consist of one or more interconnected hexagons divided into six sectors, each one with different gravity. This feature is a key in the development of the agents. As it will be explained later, the gravity affects their movement and their axis are exchanged.

Chapter 3

3. Design

In this chapter, the design phase of the agents is addressed. In relation to *traditional* techniques, Unreal Engine 4 behavior trees have been employed, using the editor presented previously. The machine learning agents have been implemented with Q-learning, comparing two value functions *Q(s,a)* and *(s)*. Furthermore, as the game in which the agents are introduced is a survival shooter with the apparition of different hordes, it has been decided that the learning process will be on-line, so each horde should be more complex than the previous ones because they will learn from them.

As a premise for the development of our enemies, we need to remember that to embrace the fun processes of the player, the agents must be:

- **Intelligible:** The AI should be easy to understand.
- **Interactive:** All in-game actors should focus on the player
- **Unpredictable:** Gameplay should not become repetitive.

3.1 Speeders

3.1.1 Concept Design

Firstly, a basic physical and behavioral description is approached in this Section. Lately, the tasks this type of agent must perform will be introduced.

Speeders are a type of futuristic spider, covered by an exoskeleton. They have three pairs of limbs, with powerful claws. These agents have the skill of jumping and its main feature is their high speed and their ability to move in groups. Once they have reached their target, part of the group tries to catch the prey. Also, the bite of these enemies is toxic, so it makes damage per time. The rest of group will jump to reach the upper parts of the victim. If the speeder reaches the target in the jump, it sticks its claws on the skin of the prey, to tear it up and cause a plentiful hemorrhage. When they attack in a group there is no hierarchy, they have no leaders or followers, all the members behave in the same way. The model and some animations have been provided by the artist of the game Hive. The physical appearance of speeders is shown in Figure 3.1.

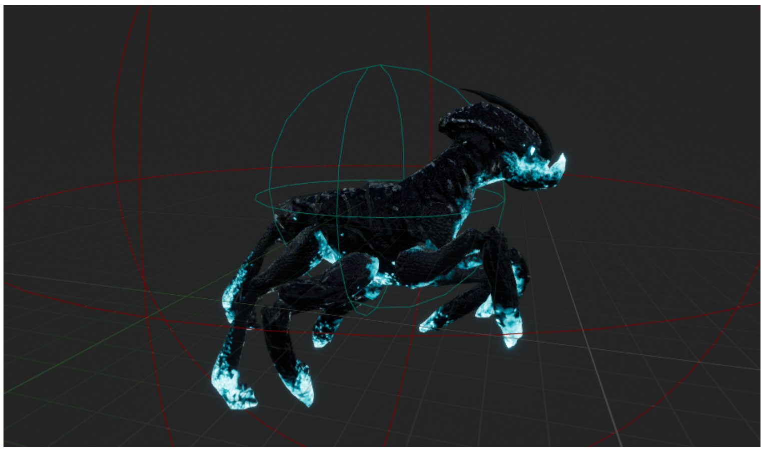


Figure 3.1: Speeder appearance

Speeders statistics are presented in Table 3.1:

Table 3.1: Speeders - Statistics

Speeders	
Feature	**Level**
DAMAGE	From 1 to 5
AI LEVEL	Simple
MOVEMENT	Very Fast
SPAWN NUMBER	Groups from 4 to 16
HEALTH POINTS	100
BASIC ATTACK	Toxic bite
ATTACK SPEED	High

Regarding its abilities, they are summarized in Table 3.2. Speeders have two abilities, *TOXIC BITE*, which is their common attack, and *SAW JUMP*, which is their special ability. The first one, is a fast, non ranged, attack which poisons the prey, making damage per second (DPS), during 10 seconds. The jump has more range than the basic attack, so speeders can jump from the distance. This attack makes a high immediate damage and a continuous bleeding for 15 seconds.

3.1.2 Behavior Trees

Once the conceptual design of the speeders' behavior is completed, let's proceed to design the behavior tree and its different nodes. As well as having the two skills mentioned previously, the agents must have the ability to move. This task causes several problems due to the fact that the Hive video game, contains gravity changes. The problems and its solutions related to the movement dimension will be detailed in next chapter.

To accelerate the BT calculations, is necessary to employ a blackboard, which will be updated in a node placed at the root of the BT, so when a node needs a value it has just to pick it from the blackboard, instead of making all the calculations.

Each agent has a blackboard with the following information: one reference to the player (if the speeder has sensed it), the distance between them, two booleans to indicate if the abilities are

Table 3.2: Speeders - Abilities

TOXIC BITE	
TYPE	Poisonous
IMMEDIATE DAMAGE	From 1 to 5
EFFECT	Poison: From 1 to 2 DPS/10 sec
COOLDOWN	5 sec
SAW JUMP	
TYPE	Range sweep
IMMEDIATE DAMAGE	From 5 to 15
EFFECT	Bleeding: 1 DPS/15 sec
COOLDOWN	10 sec

available or they are in cooldown, two variables to store the last time each ability was performed, a counter for the attacking speeders at the same time and a random position.

All of them are updated each 0.5 ± 0.1 seconds, in a service placed in the BT root. They are updated at that time because it is the most suitable referenced in the UE4 documentation.

Next, the explanation of the five branches of the task leaves of the BT, which is shown in Figure 3.2, is presented. As said before, the tasks priority increments from left to right.

The first four tasks depend whether the agents have noticed the player and have stored its reference in their blackboard. The process of detecting the player is performed by a UE4 event called *OnPawnSensing*, which works with a raycast from the agent forward its direction, so if another agent intersects with this ray, for instance, the player, this event will be triggered.

3.1.2.1 FLEE FROM THE PLAYER

This is the task with the highest priority. When the agent has been hurt, he must flee from the player, to avoid more damage. To do so, there is a branch with a decorator which checks its health level, followed by a sequence which consists of a task that calculates a point in the opposite direction of the player and the task *moveTo* that point.

3.1.2.2 TOXIC BITE

This task has the second priority and it constitutes the basic attack of the agent. It is a hand to hand attack, and because of this, it is necessary that the distance between the agent and the player to be less or equal to 200 units of measurement in UE4. Furthermore, to avoid the agents taking this task continually, it can only perform it if there are less than 4 speeders attacking the player already, so there can only be 5 speeders performing this ability at the same time.

3.1.2.3 SAW JUMP

This ability is the action with the next priority. As it is a ranged attack, the agent can perform this task if the distance between it and the player is less or equal than 600 units. Also, this ability will be enabled if there are already some speeders attacking the player with the ability mentioned previously, so this branch has to check the number of attackers.

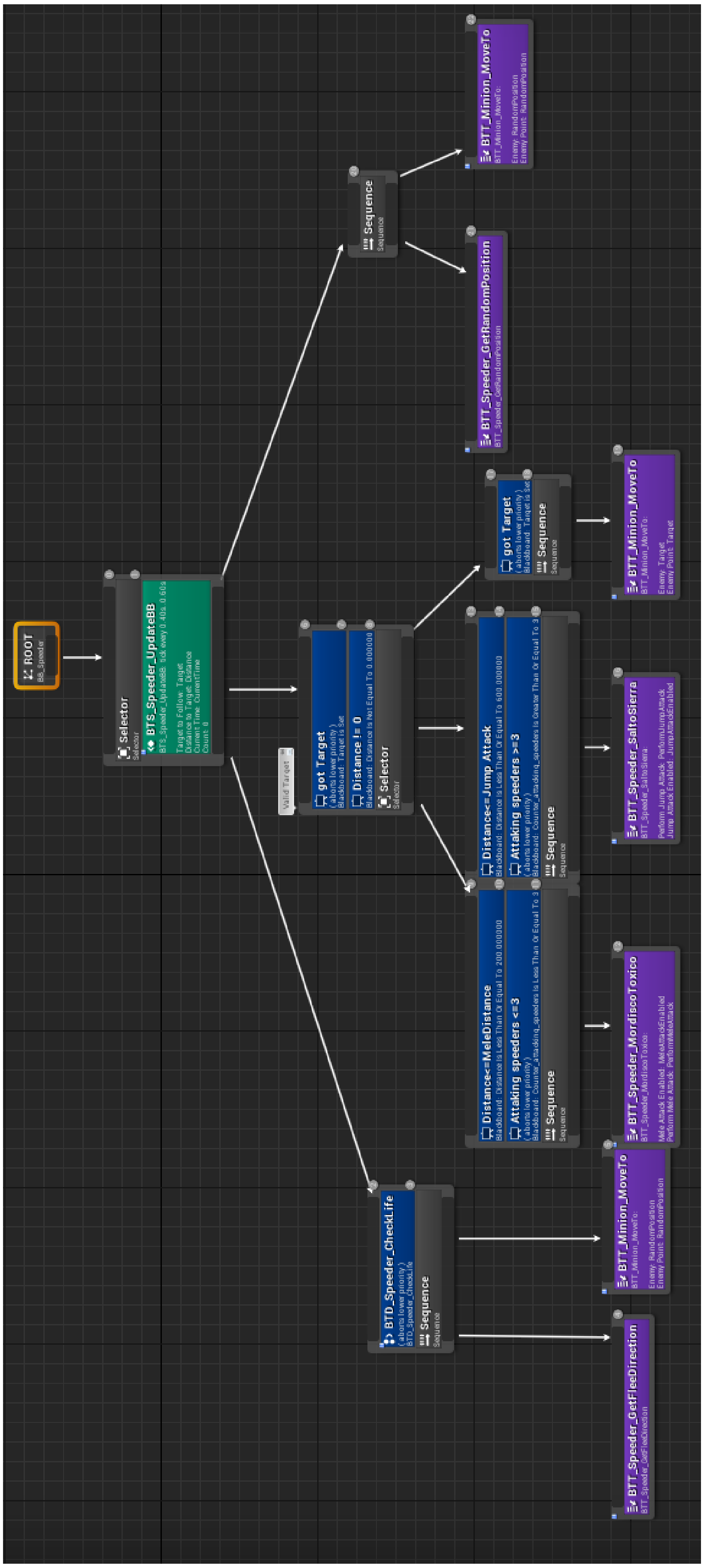


Figure 3.2: Speeders behavior tree

3.1.2.4 MOVE TO PLAYER

The agents' movement causes several problems which will be detailed in Section 4. Nevertheless, this function works with a target point to reach, so in this case, the speeder needs to know the player's position.

3.1.2.5 MOVE RANDOMLY

At the moment when the agent has not detected the player, it should move randomly. Therefore, the BT contains a branch set up with a sequence with a task which gets a random point beneath a radius and the function *MoveTo* to reach that.

Summarizing, the following nodes have to be implemented for the speeders' behavior:

- **TASKS**
  - TOXIC BITE
  - SAW JUMP
  - MOVE TO PLAYER
  - FLEE FROM PLAYER
  - MOVE RANDOMLY
- **DECORATORS**
  - IS TARGET SET?
  - IS NEAR DISTANCE?
  - IS Med. DISTANCE?
  - ARE THERE LESS THAN 4 SPEEDERS ATTACKING THE PLAYER?
  - IS TARGET SET?
  - IS SPEEDER HURT?
  - IS RANDOM POINT REACHED?
- **SERVICES**
  - BLACKBOARD UPDATE VALUES

3.1.3 Q-Learning

First, the five most important elements of a reinforcement learning system have to be designed: a policy, a reward function, a value function, states space and an actions space.

3.1.3.1 Action Space

As this is the first learning agent that it has developed, the complexity of its behavior has been reduced in pursuit of the investigation. Also, this type of enemy has a constant tracking of the player position, so it will always know where he is. This way, the agent will have three possible actions: SAW JUMP, MOVE TO PLAYER and FLEE FROM PLAYER.

3.1.3.2 State Space

The environment state model is constituted by two features: the speeder-player health proportion and the distance between them. Regarding the first feature, it can be: the speeder has health advantage, the speeder and the player have the same health proportion, the speeder has less health than the player and the player is almost dead, which means that if he receives one attack more, it would die. Regarding distance measurement, all the possibilities have been discretized to: the speeder is near, they are at a medium distance or the speeder is far away from the player. The combination of these features leads to twelve different states which can be seen in Tables 9 and 10. The exact proportions and values of these features will be detailed in the fourth chapter.

3.1.3.3 Value Function and Reward Function

These two functions are presented together, due to the fact that the reward function depends on the value one. Q-learning algorithm works with a value function which provides values per state and action. Regarding the reward function, the results of a reward function for pairs state-action and other with only rewards per state are compared in Section 5.

The first reward function is shown in Table 3.3 and the second custom reward function for only states, in Table 3.4.

In both reward functions, a value of -10 or -1 identifies the actions and states less desirable, 0 to the no influent, 100 to the desired state (the final state) and a range of 1 ,10, 20 and 60 for the rest of the actions and states depending on its desirable level.

Table 3.3: Speeders - Reward function for pairs state-action

STATE	**ACTIONS**		
	Move	**Attack**	**Flee**
Near and more health	0	1	0
Near and same health	0	1	1
Near and less health	-10	-10	1
Near and player is almost dead	1	100	-1
Med. distance and more health	10	1	0
Med. distance and same health	10	0	0
Med. distance and less health	-1	0	1
Med. distance and player is almost dead	100	1	0
Far and more health	10	0	-1
Far and same health	1	0	0
Far and less health	-1	-1	1
Far and player is almost dead	10	0	-1

Table 3.4: Speeders - Reward function for states only

STATE	**REWARD**
Near and more health than player	10
Near and same health than player	0
Near and less health than player	-10
Near and player is almost dead	100
Med. distance and more health than player	10
Med. distance and same health than player	0
Med. distance and less health than player	-10
Med. distance and player is almost dead	60
Far and more health than player	10
Far and same health than player	0
Far and less health than player	-10
Far and player is almost dead	20

3.1.3.4 Policy

A ε-greedy policy has been used, with ε equal to 0.2, which means that 2 from 10 actions will be randomly taken, so exploration can discover better choices.

3.2 Dyggers

3.2.1 Concept Design

Regarding their physical appearance, which is shown in Figure 3.3, this type of agents is greater than the speeders, more than the double of its size. They have a heavy shield, which allows them to

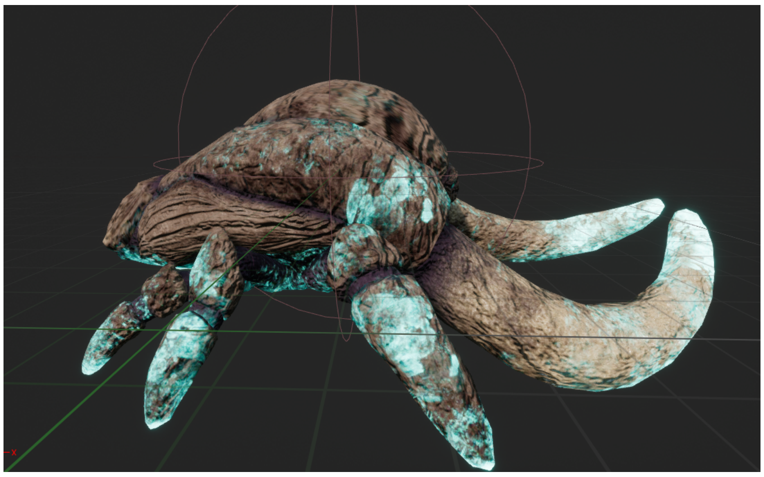


Figure 3.3: Dygger appearance

resist the pressure under the ground. Regarding its physiognomy, two parts can be differentiated: its exterior thick shield and the weak structure covered by it. Furthermore, its head counts with its own protection, which seems to be like a helmet for them. They have three pairs of armored legs.

The dygger's tusks are the most highlighted feature of their physic. These have disproportionately dimensions, so they offer the dyggers, a lot of attack power. Furthermore, these agents can move with facility under the ground. That is the reason why when they feel threatened, they prefer to dig and protect themselves. Also, they can attack from the ground, taking advantage of the surprise factor. When they perceive the player over their position, they make a powerful attack, which consists in drilling the ground and jumping vertically to catch him.

This type of enemies try to work together to destroy their victim, so when one dygger finds a human, if it is alone, it will immobilize him to make time for other diggers to arrive at that position.

Despite its powerful ability to dig, their tusks can only resist rock. This means that dyggers can only dig where the floor is rocky, not when it is made of metal. Dyggers statistics are presented in Table 3.5:

Table 3.5: Dyggers - Statistics

Dyggers	
Feature	**Level**
DAMAGE	From 20 to 40
AI LEVEL	Advance
MOVEMENT	Slow
SPAWN NUMBER	Groups from 4 to 8
HEALTH POINTS	500
BASIC ATTACK	Tusk attack
ATTACK SPEED	Slow

Regarding its abilities, they are summarized in Table 3.6. Dyggers have five abilities. Its basic attack is the TUSK ATTACK, it is fast and has a small cooldown. As said previously, these agents can dig and undig, but only when they are over a rocky floor, so they can not trespass metal floor. When they are underground, they are invulnerable to the player's attacks. By this mean, they have a special ability, the JUMP ATTACK, which lets the dygger to impulse vertically from the ground to reach and hit the player. Furthermore, dyggers behavior complexity is greater than the speeders, because they try to work together and coordinate to catch the player, so if one dygger is alone and it catches the player, it will execute its BREAK HUMAN attack which will immobilize the player so other dyggers can reach and attack him.

3.2.2 Behavior Trees

In this Section, the dygger's behavior tree and tasks are presented. As it has been done previously with speeders, the basic movement and flee tasks are out of the scope of this chapter. Also, dyggers need a blackboard which updates its values constantly the same way speeders do.

Each agent contains one blackboard with the following information: a reference to the player, the distance between them, a counter for the dyggers near the player, a boolean to determine if the dygger is over or under the ground, the time it has been in that mode, three booleans for the abilities availability and other three more to store the time they were executed.

As in the speeders' blackboard, these values are updated each 0.5 ± 0.1 seconds, in a service placed in the BT root.

Now is time to explain all the branches of the dyggers' BT, which is shown in Figure 10.

3.2.2.1 UNDIG

This branch triggers when the dygger is already under the ground, so here, the two possibilities for the agent to execute the undig task to go over the ground are either the time has expired or the point towards it is moving is located in a metallic floor, which it can be perforated by the dygger, so it must leave the ground. With the first option, it is being referred to the fact that dyggers have a timer of 20 seconds, which means that each 20 seconds they have to change its mode from under the ground to over the ground or vice-versa.

3.2.2.2 DIG

This branch is like the previous branch, but in the opposite way. If the dygger is over the ground and its time has expired, it has to dig as long as the floor is made of rock.

3.2.2.3 JUMP ATTACK

Once the dygger has targeted the player and knows its position, if it is under the ground and near him (about 250 units), he will execute this attack. This task has two different effects. If at the moment of the execution of this task, the player is flying or jumping, the dygger will run under the ground to a predicted position for the player, and then jump vertically to try to hit him. If the player is on the floor, the dygger will just do its attack in that location.

3.2.2.4 BREAK HUMANS ATTACK

This type of attack is performed when the dygger is outside the ground and it catches the player, when any other agent is near. As said before, this ability is about immobilizing the player, so other dyggers can run to catch him in a group.

3.2.2.5 TUSK ATTACK

This is the basic attack, with no range, so it is executed when the dygger is near the player and other agents are near him, too.

3.2.2.6 FLEE FROM PLAYER

When the dygger has been hurt, he must flee from the player, to avoid more damage. To do so, there is a branch with a decorator which checks its health level, followed by a sequence which consists of a task that calculates a point in the opposite direction of the player and the task *moveTo* that point. After reaching that point, the agent will dig to protect himself under the ground.

3.2.2.7 MOVE RANDOMLY

At the moment when the agent has not detected the player, it should move randomly. Therefore, the BT contains a branch set up with a sequence with a task which gets a random point beneath a radius and the function *moveTo* to reach that.

Summarizing, we have to implement the following nodes:

- **TASKS**
  - DIG
  - UNDIG
  - JUMP ATTACK
  - BREAK HUMANS ATTACK
  - TUSCK BASIC ATTACK
  - MOVE TO PLAYER
  - FLEE FROM PLAYER
  - MOVE RANDOMLY
- **DECORATORS**
  - IS TARGET SET?
  - IS UNDER THE GROUND?
  - IS UNDER ROCK GROUND?
  - IS TIMER COMPLETED?
  - IS NEAR DISTANCE?
  - IS MEDIUM DISTANCE?
  - ARE THERE NO DYGGERS NEAR THE PLAYER?
  - IS DYGGER HURT?
  - IS RANDOM POINT REACHED?
- **SERVICES**
  - BLACKBOARD UPDATE VALUES

3.2.3 Q-Learning

As it has been done for the speeders, the five elements of a reinforcement learning system have to be designed for this second agent: a policy, a reward function, a value function, states space and an actions space, but this time they will be more complex, because in this case, the agents will not get their tasks reduced, so reinforcement learning techniques in both simple and complex agents can be compared.

3.2.3.1 Action Space

The unique ease this type of dygger has among the previously mentioned one is that this has a constant tracking of the player position, so it will always know where he is. As with the BT dyggers, this will have seven actions: MOVE TO PLAYER, FLEE FROM PLAYER, DIG, UNDIG, JUMP ATTACK, BREAK HUMANS ATTACK and TUSK ATTACK.

3.2.3.2 State Space

As this enemy is more complex, the environment state model has to be constituted by five features: the speeder-player health proportion, the distance between them, the type of the floor where the dygger is, the mode (under or above) of the ground and the number of dyggers which are near the player.

The two first features are the same as for the speeders, so the health proportion can be: dygger has health advantage, dygger and player have the same health proportion, dygger has less health than player and the player is almost dead; and the distance: the dygger is near, they are at Med. distance or the dygger is far away from the player. The last feature, number of dyggers, has been discretized due to the fact that it only matters if that is zero or different to zero, the dygger is the only enemy near the player or not. The combination of these features leads to 96 different states which can be seen in Table 3.7. This change in the environment has led to an exponential increment of the complexity of the states space, with a difference of 84 states between the two types of enemies. This fundamental rising of complexity, will let us compare both agents in terms of usability of reinforcement learning depending on the complexity of the agent.

3.2.3.3 Value function and Reward function

Again, these two functions are presented together, due to the fact that the reward function depends on the value one. The reward function for the value function $Q(s,a)$, is shown in Table 3.7.

3.2.3.4 Policy

A ε-greedy policy has been used, with ε equal to 0.2, which means that 2 from 10 actions will be randomly taken, so exploration can discover better choices.

Table 3.7: Dyggers - $Q(s,a)$

STATE	ACTIONS						
	Move	**Flee**	**Dig**	**Undig**	**Jump A.**	**Break A.**	**Tusk A.**
Near, more h., rock, over g.	0	-1	1	-1	-1	10	5
Near, more h., rock, over g.	0	-1	1	-1	-1	-1	10
Near, more h., rock, under g.	0	-1	-1	1	1	-1	-1
Near, more h., rock, under g.	0	-1	-1	1	1	-1	-1
Near, more h., metal, over g.	0	-1	-1	-1	-1	10	5
Near, more h., metal, over g.	0	-1	-1	-1	-1	-1	10
Near, more h., metal, under g.	0	-1	-1	-1	-1	-1	-1
Near, more h., metal, under g.	0	-1	-1	-1	-1	-1	-1

Continued on next page

Table 3.7 – continued from previous page

STATE	ACTIONS						
	Move	**Flee**	**Dig**	**Undig**	**Jump A.**	**Break A.**	**Tusk A.**
Near, same h., rock, over g.	0	1	1	-1	-1	10	5
Near, same h., rock, over g.	0	0	1	-1	-1	-1	5
Near, same h., rock, under g.	0	-1	-1	1	1	-1	-1
Near, same h., rock, under g.	0	-1	-1	1	1	-1	-1
Near, same h., metal, over g.	0	1	-1	-1	-1	5	5
Near, same h., metal, over g.	0	0	-1	-1	-1	-1	5
Near, same h., metal, under g.	0	-1	-1	-1	-1	-1	-1
Near, same h., metal, under g.	0	-1	-1	-1	-1	-1	-1
Near, less h., rock, over g.	-1	1	1	-1	-1	1	0
Near, less h., rock, over g.	-1	1	1	-1	-1	-1	1
Near, less h., rock, under g.	-1	1	-1	0	0	-1	-1
Near, less h., rock, under g.	-1	1	-1	0	1	-1	-1
Near, less h., metal, over g.	-1	1	-1	-1	-1	1	0
Near, less h., metal, over g.	-1	1	-1	-1	-1	-1	1
Near, less h., metal, under g.	-1	1	-1	-1	-1	-1	-1
Near, less h., metal, under g.	-1	1	-1	-1	-1	-1	-1
Near, P. dead, rock, over g.	0	-1	0	-1	-1	20	50
Near, P. dead, rock, over g.	0	-1	0	-1	-1	-1	100
Near, P. dead, rock, under g.	0	-1	-1	0	20	-1	-1
Near, P. dead, rock, under g.	0	-1	-1	0	20	-1	-1
Near, P. dead, metal, over g.	0	-1	-1	-1	-1	20	50
Near, P. dead, metal, over g.	0	-1	-1	-1	-1	-1	100
Near, P. dead, metal, under g.	0	-1	-1	-1	-1	-1	-1
Near, P. dead, metal, under g.	0	-1	-1	-1	-1	-1	-1
Med., more h., rock, over g.	1	-1	0	-1	-1	5	1
Med., more h., rock, over g.	1	-1	0	-1	-1	-1	5
Med., more h., rock, under g.	1	-1	-1	0	1	-1	-1
Med., more h., rock, under g.	1	-1	-1	0	1	-1	-1
Med., more h., metal, over g.	1	-1	-1	-1	-1	5	1
Med., more h., metal, over g.	1	-1	-1	-1	-1	-1	5
Med., more h., metal, under g.	1	-1	-1	-1	-1	-1	-1
Med., more h., metal, under g.	1	-1	-1	-1	-1	-1	-1
Med., same h., rock, over g.	1	0	0	-1	-1	5	1
Med., same h., rock, over g.	1	0	0	-1	-1	-1	1
Med., same h., rock, under g.	1	0	-1	0	1	-1	-1
Med., same h., rock, under g.	1	0	-1	0	1	-1	-1
Med., same h., metal, over g.	1	0	-1	-1	-1	5	1
Med., same h., metal, over g.	1	0	-1	-1	-1	-1	1
Med., same h., metal, under g.	1	0	-1	-1	-1	-1	-1
Med., same h., metal, under g.	1	0	-1	-1	-1	-1	-1

Continued on next page

Table 3.7 – continued from previous page

STATE	ACTIONS						
	Move	**Flee**	**Dig**	**Undig**	**Jump A.**	**Break A.**	**Tusk A.**
Med., less h., rock, over g.	0	1	1	-1	-1	1	0
Med., less h., rock, over g.	0	1	1	-1	-1	-1	0
Med., less h., rock, under g.	-1	1	-1	0	0	-1	-1
Med., less h., rock, under g.	-1	1	-1	0	0	-1	-1
Med., less h., metal, over g.	0	1	-1	-1	-1	1	0
Med., less h., metal, over g.	0	1	-1	-1	-1	-1	0
Med., less h., metal, under g.	-1	1	-1	-1	-1	-1	-1
Med., less h., metal, under g.	-1	1	-1	-1	-1	-1	-1
Med., P. dead, rock, over g.	5	-1	0	-1	-1	5	1
Med., P. dead, rock, over g.	5	-1	0	-1	-1	-1	5
Med., P. dead, rock, under g.	5	-1	-1	0	1	-1	-1
Med., P. dead, rock, under g.	5	-1	-1	0	1	-1	-1
Med., P. dead, metal, over g.	1	-1	-1	-1	-1	5	1
Med., P. dead, metal, over g.	1	-1	-1	-1	-1	-1	5
Med., P. dead, metal, under g.	1	-1	-1	-1	-1	-1	-1
Med., P. dead, metal, under g.	1	-1	-1	-1	-1	-1	-1
Near, more h., rock, over g.	10	-1	1	-1	-1	0	0
Near, more h., rock, over g.	10	-1	1	-1	-1	-1	0
Near, more h., rock, under g.	10	-1	-1	1	0	-1	-1
Near, more h., rock, under g.	10	-1	-1	1	0	-1	-1
Near, more h., metal, over g.	10	-1	-1	-1	-1	0	0
Near, more h., metal, over g.	10	-1	-1	-1	-1	-1	0
Near, more h., metal, under g.	10	-1	-1	-1	-1	-1	-1
Near, more h., metal, under g.	10	-1	-1	-1	-1	-1	-1
Near, same h., rock, over g.	10	-1	1	-1	-1	0	0
Near, same h., rock, over g.	10	-1	1	-1	-1	-1	0
Near, same h., rock, under g.	10	-1	-1	1	0	-1	-1
Near, same h., rock, under g.	10	-1	-1	1	0	-1	-1
Near, same h., metal, over g.	10	-1	-1	-1	-1	0	0
Near, same h., metal, over g.	10	-1	-1	-1	-1	-1	0
Near, same h., metal, under g.	10	-1	-1	-1	-1	-1	-1
Near, same h., metal, under g.	10	-1	-1	-1	-1	-1	-1
Near, less h., rock, over g.	1	0	1	-1	-1	0	0
Near, less h., rock, over g.	1	0	1	-1	-1	-1	0
Near, less h., rock, under g.	1	0	-1	1	0	-1	-1
Near, less h., rock, under g.	1	0	-1	1	0	-1	-1
Near, less h., metal, over g.	1	0	-1	-1	-1	0	0
Near, less h., metal, over g.	1	0	-1	-1	-1	-1	0
Near, less h., metal, under g.	1	0	-1	-1	-1	-1	-1
Near, less h., metal, under g.	1	0	-1	-1	-1	-1	-1
Near, P. dead, rock, over g.	10	-1	1	-1	-1	1	1
Near, P. dead, rock, over g.	10	-1	1	-1	-1	-1	1

Continued on next page

Table 3.7 – continued from previous page

STATE	**ACTIONS**						
	Move	**Flee**	**Dig**	**Undig**	**Jump A.**	**Break A.**	**Tusk A.**
Near, P. dead, rock, under g.	10	-1	-1	1	0	-1	-1
Near, P. dead, rock, under g.	10	-1	-1	1	0	-1	-1
Near, P. dead, metal, over g.	10	-1	-1	-1	-1	1	1
Near, P. dead, metal, over g.	10	-1	-1	-1	-1	-1	1
Near, P. dead, metal, under g.	10	-1	-1	-1	-1	-1	-1
Near, P. dead, metal, under g.	10	-1	-1	-1	-1	-1	-1

Table 3.6: Dyggers - Abilities

DIG	
TYPE	No damage skill
EFFECT	Invulnerability under the floor
COOLDOWN	20 sec
UNDIG	
TYPE	No damage skill
EFFECT	To go from underground to over the floor
COOLDOWN	20 sec
JUMP ATTACK	
TYPE	Vertical jump from underground
IMMEDIATE DAMAGE	From 30 to 50
EFFECT	Positioned over the floor, next to the player
COOLDOWN	20 sec
BREAK HUMANS ATTACK	
TYPE	Immobilization
IMMEDIATE DAMAGE	0
EFFECT	Immobilization: From 15 to 30 DPS/5 sec
COOLDOWN	9 sec
TUSCK BASIC ATTACK	
TYPE	Basic attack
IMMEDIATE DAMAGE	From 20 to 40
EFFECT	
COOLDOWN	2 sec

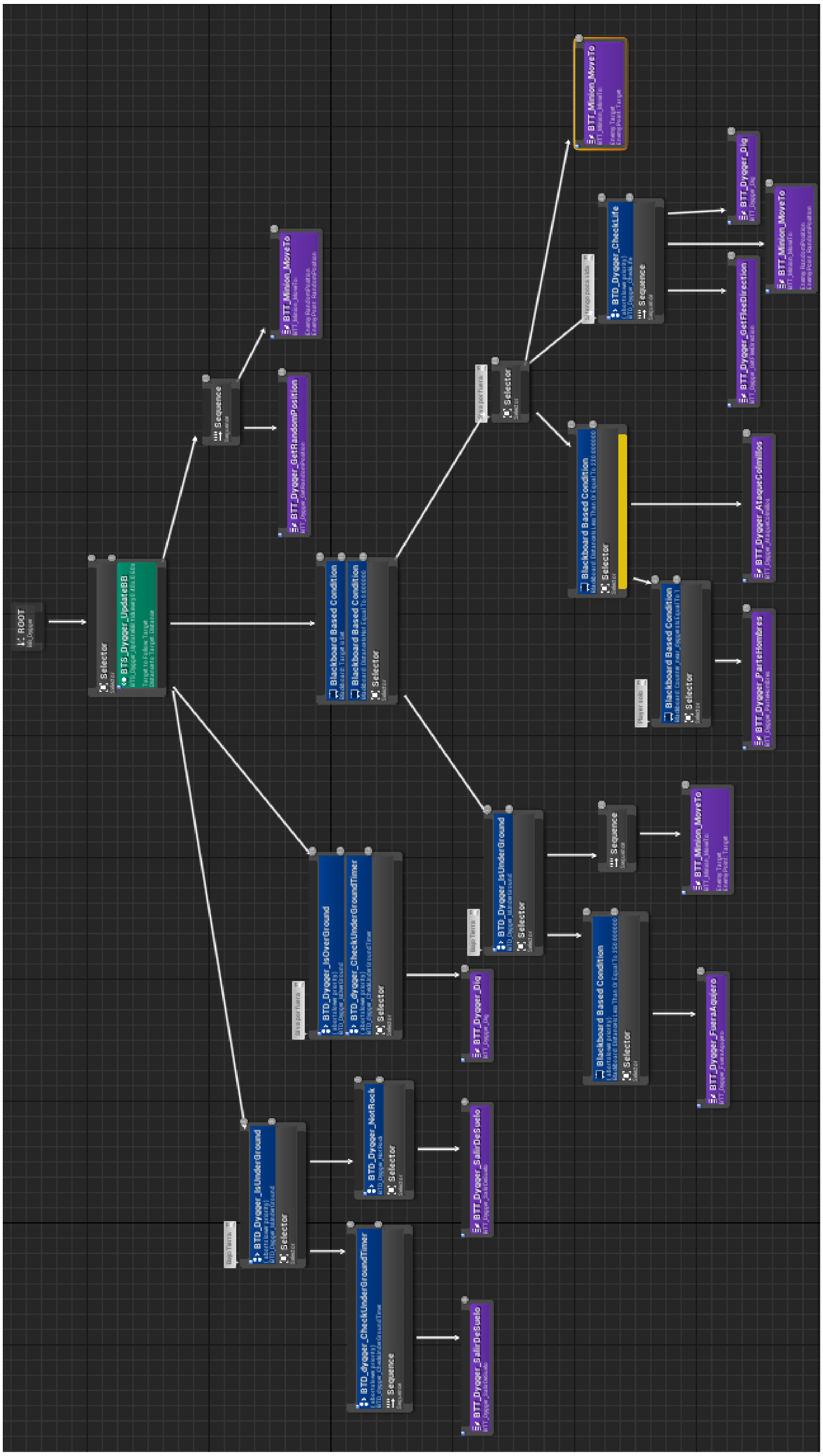


Figure 3.4: Dyggers behavior tree

IV

Chapter 4

4. Development

In this chapter, the development phase of each of the agents described in the design Section is addressed.

4.1 Speeders

4.1.1 Speeders Behavior Tree

At first point, let's analyze the class (and blueprints) diagram which has been implemented for the first agent (Figure 4.1). To favor its readability the diagram has been simplified.

There are two main blueprints (visual scripting in UE4 for implementing classes), which are *SpeederBT* and *SpeederController*. The first one, is the main class of the agent, due to the fact that it constitutes the actor in essence. It contains the meshes, animations, collisions, movement components and so forth. As said before, it is essential that an agent has also a controller, which is a class that operates as the brain of the agent. This class is linked to the behavior tree and the blackboard and it is the responsible to indicate, at each moment, what should do the speeder: where to go, where to orientate, etc. This controller inherits from an Unreal class called *DetourCrowdController*, which will be explained in detail later with the movement dimension.

The agent's blueprint (*SpeederBT*)inherits from a C++ class called Speeder, which inherits from *MinionCharacter*, which inherits from the Unreal class *Character*. *Character* is a predefined class of the engine, which manages all the components related to the agent. Specifically, an Unreal Character is a Pawn that has a mesh, collisions and movement. It is the responsible of all the physical interaction between the player and the agent. Also, they are designed to provide the ability to walk, jump, fly and swim using the *CharacterMovementComponent*. The Pawn is the base class of all actors that have a controller, which means that they are the physical representation of the agents. An actor is whatever object placed in the game world and it can have several components, which control the movement, the render, etc.

For the development of the game *Hive*, its creators, implemented some classes which inherit from Character, to differ each type of character of the game. for instance, there is a *ConceptPlayer* class, which manages all the player's actions (which are out of the scope of this project) or the class *MinionCharacter*, that manages the autonomous agent's movement taking the gravity changes into account. It also controls how to hurt the player or how to receive damage from him. This class was developed by the Hive team and it has been a starting point for the development of the characters, due to the fact that it manages several actions and functions that remain beyond my reach because of the game's magnitude. Nevertheless, I have had to modify this class in several times to adapt it to the necessities of the proposed agents in this paper, such as adding new different types of permanent damage like immobilization or bleeding.

From this class (*MinionCharacter*), both of the agents designed for the use of behavior trees, inherit.

4.1.1.1 Movement Dimension

Regarding the movement, the UE4 default movement component has been used. This employs a pathfinding with a navigation mesh. In this case, pathfinding parameters have been set, to let the agent avoid collisions with static objects such as the environment, and with the player, but it has not set to avoid other speeders. The reason of this is because speeders will spawn by hordes of several agents, and avoiding all the collisions consumes too many resources, what affected directly in the performance of the game, slowing the framerate. Nevertheless, as the visual effect was not suitable because the agents went through each other and moved into a single file as it was the shortest path to the player. Furthermore, they usually overlapped. The first solution planned, was to elaborate a steering behavior consistent in modifying the agents' trajectory, based on its direction and the others' ones. This was not effective because this solution combined the custom made steering with the default UE4 pathfinding, which can not be controlled anyway. This leads to a problem, because each movement method contradicted sometimes and the movement was not suitable because it seemed that the agents moved in zig zag too fast, because the agent was trying to reach two different positions at the same time.

The next step was to discard that steering and implement another one, which consisted in that the agent jumped over the one which interjected its direction. For this solution, three different mechanisms were implemented:

- **To add a constant impulse:** This solution did not work properly, because it was always the same impulse, but the distances between the agents were different.
- **Parabolic jump:** In this attempt, the angle and the velocity of the jumping impulse, were calculated with the parabolic movement formula. Despite it was a realistic jump, the visual effect was that the speeder never jumped over the others, it jumped but is final position in relation to the intersecting agent was the same. After several attempts to try to solve that issue by augmenting the impulse, the jump remained the same. Again, it was a problem of incompatibility of having two movement methods, but in this case, it was different than the previous one. In this case, the maximum speed of the pathfinding did not let the impulse velocity to be greater, so the jump was not strong enough to pass over the agents. After trying to modify its parameters, this idea was discarded because the conflicts of the movement could not be solved.

After this fails, a thoughtful research in UE4 documentation was made and different solutions were found, but the information was incomplete.

One solution was to enable a parameter of the pathfinding called RVO avoidance, which is an implementation of the Reciprocal Velocity Obstacle algorithm. This algorithm works with velocities, when choosing a new velocity for the agent, it is taken from an average of its current velocity and a velocity outside the velocity obstacle [18].

The main drawback of this method is that it does not care about the navigation mesh and its bounds which in turn means that the agents can end up being pushed outside it. Nevertheless, our

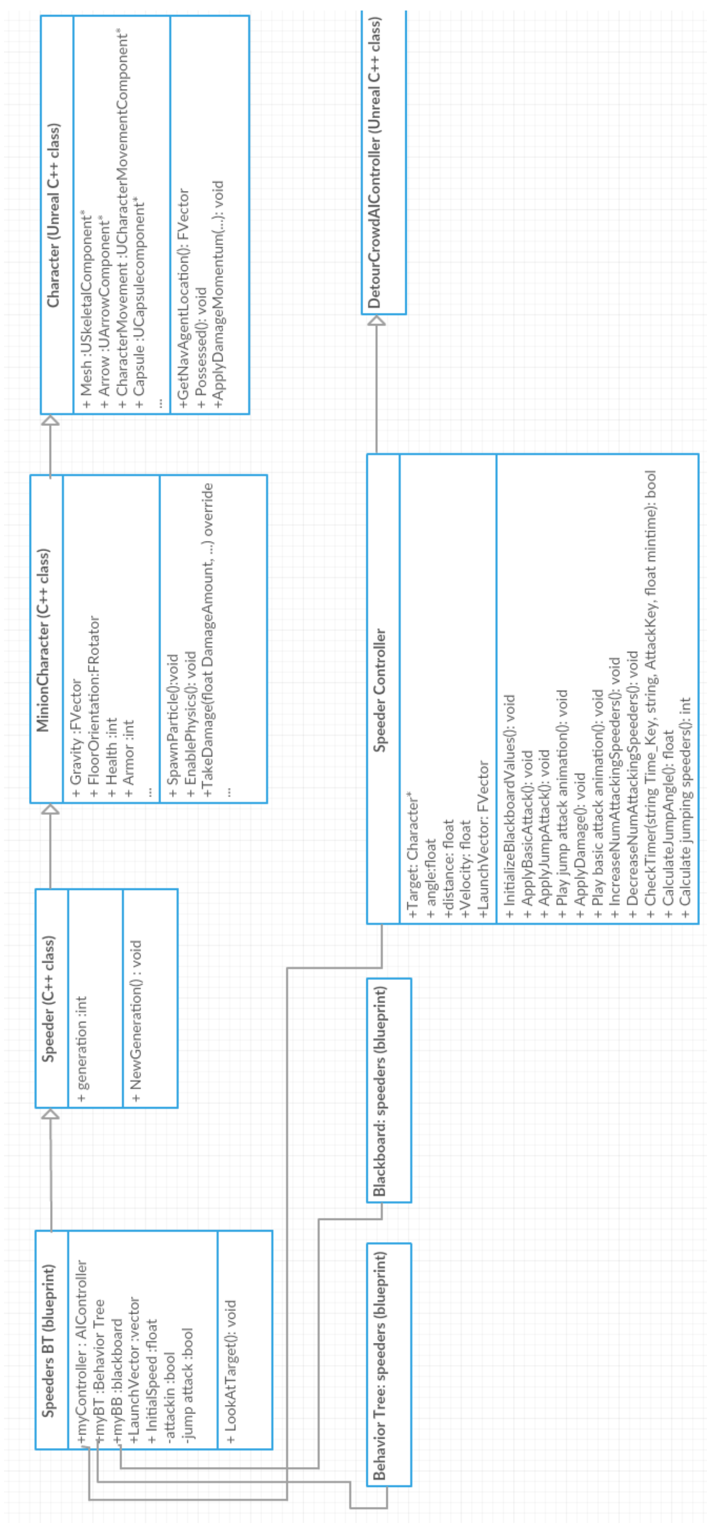

Character (Unreal C++ class)
+ Mesh :USkeletalComponent*
+ Arrow :UArrowComponent*
+ CharacterMovement :UCharacterMovementComponent*
+ Capsule :UCapsulecomponent*
...
+GetNavAgentLocation(): FVector
+ Possessed(): void
+ApplyDamageMomentum(...): void
DetourCrowdAIController (Unreal C++ class)
MinionCharacter (C++ class)
+ Gravity :FVector
+ FloorOrientation:FRotator
+ Health :int
+ Armor :int
...
+ SpawnParticle():void
+ EnablePhysics(): void
+TakeDamage(float DamageAmount, ...) override
...
Speeder Controller
+Target: Character*
+ angle:float
+distance: float
+Velocity: float
+LaunchVector: FVector
+ InitializeBlackboardValues(): void
+ ApplyBasicAttack(): void
+ ApplyJumpAttack(): void
+ Play jump attack animation(): void
+ ApplyDamage(): void
+ Play basic attack animation(): void
+ IncreaseNumAttackingSpeeders(): void
+ DecreaseNumAttackingSpeeders(): void
+ CheckTimer(string Time_Key, string, AttackKey, float mintime): bool
+ CalculateJumpAngle(): float
+ Calculate jumping speeders(): int
Speeder (C++ class)
+ generation :int
+ NewGeneration() : void
Blackboard: speeders (blueprint)
Behavior Tree: speeders (blueprint)
Speeders BT (blueprint)
+myController : AIController
+myBT :Behavior Tree
+myBB :blackboard
+LaunchVector :vector
+ InitialSpeed :float
-attackin :bool
-jump attack :bool
+ LookAtTarget(): void

Figure 4.1: SpeederBT class diagramm

problem was different: it was related to the game environment. As in *Hive* the map is a hexagon, agents must walk through slopes and change from a plane with one orientation to another plane with a different one and when the agents reached the corner formed by two planes, they detected the plane as an obstacle which covered all the path and they got stuck, so they could only move in one plane, which was not acceptable for the gameplay. Furthermore, there is not enough information of this algorithm in the UE4, so it was discarded.

Finally, a solution was found in the UE4 documentation. there is a controller class, named *AIDetourCrowdController*. It is an algorithm that works better than RVO, because it bases the avoidance on the direction of movement and it also affects navigation movement, so the solution was to re-parent the speeder controller to inherit the movement from that class and it provided a desirable avoidance without causing a loss of computing resources.

Other problems have appeared in relation to this dimension. For instance, as the player has the possibility to jump/fly, the pathfinding is not capable of following him, when the player was not colliding with the navigation planes and, therefore, they remained immobile when the player was jumping. To solve this unacceptable issue, at the moment when the player is on air, its position is projected to the navigation plane, and the agents move towards that point, so that the movement is fluent and without errors.

In relation to the sequences movement-attacks, another problem appeared due to some reasons which will be explained lately. It was necessary to create a function which is called *LookAtPlayer* for the agents to fix their orientation towards the player whenever it was required, such as in the movement case. Theoretically, UE4 pathfinding already has it implemented, but due to the shape of the map, this failed, causing the agents to move backwards or sideways on several occasions.

The main problem with this map, is that depending on the area where the agents are placed, they have different rotation in its three axes, what means that they can not rotate in the Z axis, which is the most common case when approaching orientation issues. For instance, when they were located in a vertical plane (90 degrees from what we understand as the common floor), the resulting rotation was incorrect, due to the fact that instead of rotating in the yaw axis, visually it looked like the rotation was in the roll axis. To solve this, an interpolation has been made from the forward vector of the agent to the direction from itself to the player. This interpolation returns a vector, which can be directly converted into a rotator, which is an orthogonal rotation in 3d space that Unreal manages the rotations with. this rotator can be used to realize the orientation properly, because it contains all the axis. All these solutions to the movement issues have been employed for all the different agents, so we are not going to detail the dimension movement of the others.

4.1.1.2 Decision Dimension

As mentioned before, a behavior tree has been implemented for the decisions of the autonomous agents. It has been developed by means of the UE4 visual editor. To build it up, the following nodes and functions have been implemented.

- **DECORATORS**
  - **IS TARGET SET?** This conditional consists in the application of a default decorator of UE4, which accesses directly to the blackboard to check a value stored in it, in this case, the target reference.
  - **IS IT NEAR DISTANCE?**
    This checks if the blackboard value of the distance between player and agent is less or equal than 200 measurement units.
  - **IS IT AT MEDIUM DISTANCE?** it verifies if the distance stored in the blackboard is less or equal than 600 units.
  - **ARE THERE LESS THAN 4 SPEEDERS ATTACKING THE PLAYER?** Again, accessing to the blackboard and getting the value of the counter of attacking speeders.
  - **IS SPEEDER HURT?** Unreal default decorators only make simple comparisons, so this condition has been implemented entirely. It consists of getting some percentages to determine the hurt level of the agent.

- **TASKS**
  - **TOXIC BITE**
    This BT task makes some accesses to the agent's controller and to its blackboard to update some variables which indicate that the agent must perform the function *ApplyBasicAttack* of the *SpeederController*. These variables are controlled in the tick (update function) and they manage the attacks. At first place, the attack animation is managed, modifying the variables that show the transitions between the attack and movement states in a Finite State Machine (Figure 4.2) which controls the animations of the agent. Lately, the counter of simultaneous attacking speeders is augmented and the function *ApplyDamage* is called. In this function, the player's damage is executed only if the distance between agent and player is small enough to consider that they have collided, by means of a function named *ApplyDamage*, implemented in the *MinionCharacter* class, which accesses to the player's variables to update them depending on the attack type and others. This attack is formed by two parts: an immediate damage and a continuous one in form of poison, which has been programmed in C++, with the combination of different timers to get the effect of making X damage points per second, during Y seconds.
    Lately, the variables which control the availability of each attack, are updated. They will return to true when the cooldown time has passed, what is performed in the *CheckTimer* function that can be observed in the UML diagram.

  - **SAW JUMP**
    This task works in a similar way to the previous one, the attack control parameters are set so the controller can manage them and perform the attack in the proper moment to hurt the player if it is precise. The main difference with the attack explained before, is that at the moment when the animation starts, an impulse towards the player is added to the agent. in this case, the animation was created by myself with a tool of the same engine, so I had to learn how to use it and make the animation, task which was not

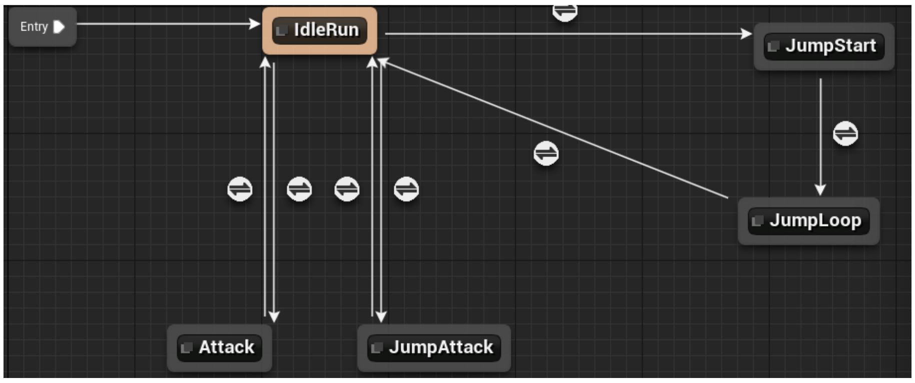


Figure 4.2: Speeder finite state machine for animations

taken into account in the planning of the project. To make this impulse, three different methods were implemented:

∗ Parabolic jump A:
The parabolic jump is calculated taking the distance between the actors into account. The jump angle is got with it and its arctangent. Lately, with the cosine between that angle and the distance, the impulse velocity is calculated, and can be split up in its components, Z (height) and Y (horizontal), to apply the impulse with them. This is made with the UE4 function *AddImpulse* which gives the agent an instantaneous force. This solution did not work properly because the gravity did interfere with the velocities and the final jump was not suitable.

∗ Parabolic jump B:
In this case, the gravity is employed with a parabolic function which works with the maximum time of the jump. The time of the jump in air and the falling time are calculated with the gravity and the maximum height of the jump. With them, we get the total time of the jump, which can be used with the gravity to get the velocity vector needed for the *AddImpulse* function. As this solution requires several calculations, another third one has been implemented.

∗ Final jump: This method gets the impulse vector with the direction vector from the speeder to the player and the up vector of the speeder multiplied by an offset that determines the strength of the jump. Despite this jump is the less realistic, it provides a suitable result and it is not resources consumer, so this has been the one method selected finally.

– **FLEE FROM PLAYER and MOVE RANDOMLY**
These functions are used together with the movement of the agents. Basically, each one returns a point either a random point, inside a radius, or a point in the opposite direction from the player to the speeder.

With all the processes explained previously, the development of the speeders with behavior trees is settled. To summarize, the base class of the speeder, its controller, its BT, its blackboard and a C++ script that inherits from a class which controls the interaction between player-enemy, has been implemented for this type of agent.

4.1.2 Speeders Q-Learning

The class diagram for the learning speeders is shown in Figure 4.3. There, it can be appreciated that the structure of the main blueprint is similar to the one of the previous case, due to the fact that it inherits from the same C++ class *Speeder*, so we are not going to highlight them, because they have been explained in Section before.

This blueprint has a reference to its controller blueprint (*SpeederMLController*), which inherits from a C++ class called *SpeederML*, which is the class in charge of making all the agent's learning process. this class has direct access to another one named *SpeedersSharedBrainMatrix*, which collects the value and reward functions in form of matrix, so there is only one instance of them in the game, which is shared by all the agents, saving, this way, memory resources.

For the development of this agent, it has been necessary to implement the interaction between blueprints and c++ scripts, due to the fact that it is precise to create the actions functions (move, attack and flee) in the class *SpeederML*. Furthermore, all of them have to be implemented in the son blueprint, *SpeederMLController*, because it is easier to implement this kind of actions via the UE4 visual blueprint editor. To do so, in the parent class, the functions should be declared as Unreal functions, specifying that they are going to be implemented in blueprints:

UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "Speeder_ML") void Flee() ;
UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "Speeder_ML") void MoveToPlayer() ;
UFUNCTION(BlueprintNativeEvent, BlueprintCallable, Category = "Speeder_ML") void ApplySaltoSierra();

The next step is to implement those functions in the blueprints as events. Once the communication between scripts and blueprints is clear, emphasize that the core of the learning process is done in the script *SpeederML*. In this class, the identification of the state and the action selection depending on the policy is realized. Most part of these functions access to the *SpeedersSharedBrainMatrix* script to get and update the values of the Q matrix.

4.1.2.1 Main Core Function

All the learning process is made inside the *Tick* function, which is executed once per frame continuously. This learning core is shown in Figure 4.4.

This function works in what can be called learning cycles. Each cycle consists in identifying the state, selecting an action depending on the followed policy which will determine if the agent has to chose a random action of the one that best results had provided up to the moment. After that, the action is performed and one timer starts depending on the action default time. After that time, the new state is identified and the Q matrix is updated with the obtained results, closing the cycle that way and starting the following one immediately.

4.1.2.2 State Identification

The state is identified in this case by the combination of two enumerations:

- **Distance To Player**
  - NEAR: If the distance is less or equal to 100 units.
  - MEDIUM: If the distance is between 101 and 400 units.
  - FAR: If distance is greater than 400.
- **Speeder Health Proportion To Player**
  - PLAYER_ALMOST_DEAD: If the player's health is the 10 % of its maximum health. For the next states, we need to calculate the proportions of the player's health and the agent's health, dividing its current health between the maximum.

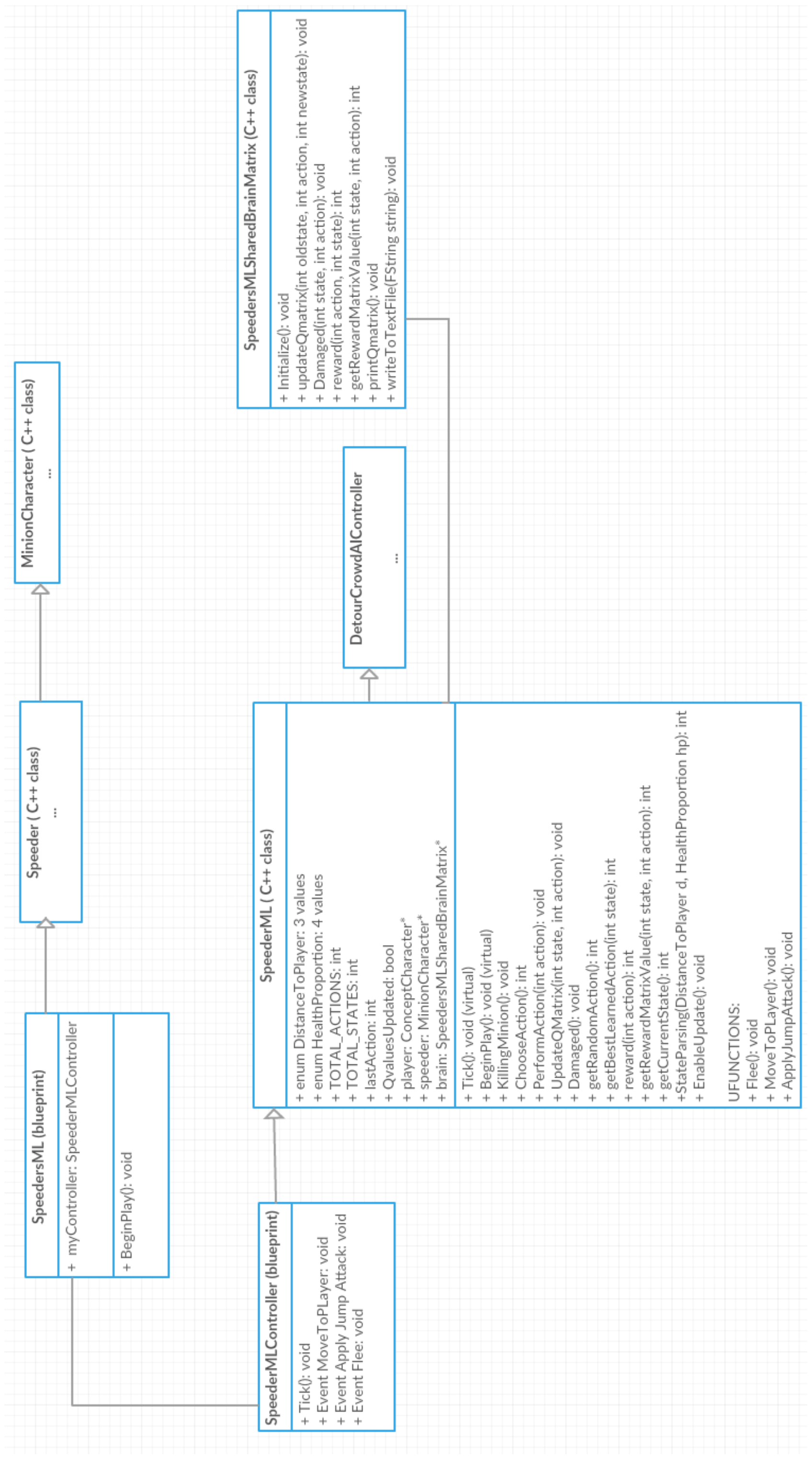
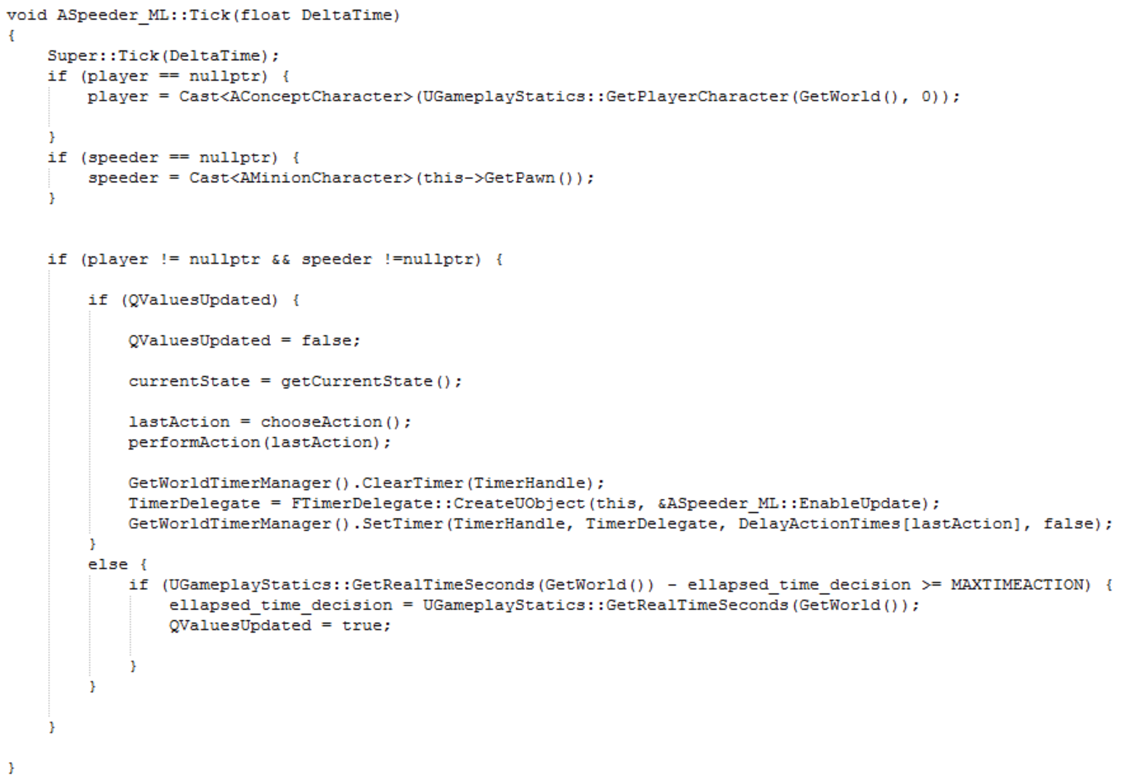

SpeedersMLSharedBrainMatrix (C++ class)
+ Initialize(): void
+ updateQmatrix(int oldstate, int action, int newstate): void
+ Damaged(int state, int action): void
+ reward(int action, int state): int
+ getRewardMatrixValue(int state, int action): int
+ printQmatrix(): void
+ writeToTextFile(FString string): void
MinionCharacter (C++ class)
...
Speeder (C++ class)
...
DetourCrowdAIController
...
SpeedersML (blueprint)
+ myController: SpeederMLController
+ BeginPlay(): void
SpeederML (C++ class)
+ enum DistanceToPlayer: 3 values
+ enum HealthProportion: 4 values
+ TOTAL_ACTIONS: int
+ TOTAL_STATES: int
+ lastAction: int
+ QvaluesUpdated: bool
+ player: ConceptCharacter*
+ speeder: MinionCharacter*
+ brain: SpeedersMLSharedBrainMatrix*
+ Tick(): void (virtual)
+ BeginPlay(): void (virtual)
+ KillingMinion(): void
+ ChooseAction(): int
+ PerformAction(int action): void
+ UpdateQMatrix(int state, int action): void
+ Damaged(): void
+ getRandomAction(): int
+ getBestLearnedAction(int state): int
+ reward(int action): int
+ getRewardMatrixValue(int state, int action): int
+ getCurrentState(): int
+StateParsing(DistanceToPlayer d, HealthProportion hp): int
+ EnableUpdate(): void
UFUNCTIONS:
+ Flee(): void
+ MoveToPLayer(): void
+ ApplyJumpAttack(): void
SpeederMLController (blueprint)
+ Tick(): void
+ Event MoveToPLayer: void
+ Event Apply Jump Attack: void
+ Event Flee: void

Figure 4.3: SpeederML class diagramm



```
void ASpeeder_ML::Tick(float DeltaTime)
{
    Super::Tick(DeltaTime);
    if (player == nullptr) {
        player = Cast<AConceptCharacter>(UGameplayStatics::GetPlayerCharacter(GetWorld(), 0));

    }
    if (speeder == nullptr) {
        speeder = Cast<AMinionCharacter>(this->GetPawn());
    }


    if (player != nullptr && speeder !=nullptr) {

        if (QValuesUpdated) {

            QValuesUpdated = false;

            currentState = getCurrentState();

            lastAction = chooseAction();
            performAction(lastAction);

            GetWorldTimerManager().ClearTimer(TimerHandle);
            TimerDelegate = FTimerDelegate::CreateUObject(this, &ASpeeder_ML::EnableUpdate);
            GetWorldTimerManager().SetTimer(TimerHandle, TimerDelegate, DelayActionTimes[lastAction], false);
        }
        else {
            if (UGameplayStatics::GetRealTimeSeconds(GetWorld()) - ellapsed_time_decision >= MAXTIMEACTION) {
                ellapsed_time_decision = UGameplayStatics::GetRealTimeSeconds(GetWorld());
                QValuesUpdated = true;

            }
        }

    }

}
```

Figure 4.4: Core of speeder learning process

- MORE_HEALTH_PROPORTION: The difference between the speeder's health proportion and the player's one is greater or equal than 0.25.
- LESS_HEALTH_PROPORTION: The opposite of the previous one. The difference between the player's health proportion and the agent's one is greater or equal than 0.2.
- SAME_HEALTH_PROPORTION: The rest of the states.

It has to be remembered, that the value and reward functions have been implemented as two matrices with the same number of rows as states in the environment, and columns like actions as explained in the chapter before. Once these enumerations have been determined according to the game state, the function *stateParsing* is called, to convert that combination of enumerations into an integer that identifies the state row in the matrix. In this case, as the possible combinations are 12 states, it is acceptable to implement a state recognition mechanism with the combination of different switch statements. Nevertheless, this is possible only because this number is not too big, fact which occurs with the second agent and which solution will be discussed later.

4.1.2.3 Action Selection and Performance

To determine the resolution of the policy, a threshold value of 0.2 has been employed, so as before determining the action, a random is performed. If this is lower than the threshold, the exploration is performed by means of the selection of a random action. Otherwise, the best learned action for the current state is selected. the function which performs this, accesses to the value Q matrix int the *SpeederSharedMatrix*, getting the value of the row that identifies the state. In this row, the column with the greater value is chosen, determining that way, the best action. Once the action has been selected, it is performed by triggering the events of those tasks in the *SpeederML* blueprint and the timer is started.

4.1.2.4 Obtaining Values and Updating Q Matrix

Once the timer has finished, the Q matrix is updated depending on the last state, the selected action, the current state and the action's reward. furthermore, to test whether the matrix was updating correctly, a writing to file mechanism has been developed to store the current time, the state, the action selected and the values of the Q matrix, which has been really useful to debug the learning process. This issue will be discussed in the testing chapter.

4.1.2.5 Problems

In this Section, we are going to address all the problems that have appeared during the development of this learning system. The main drawback is caused by two features related to the game genre: real time and shooter. On the first place, aim that previous researches about reinforcement learning by experts in this area, which have been mentioned at the state of art Section were tested in games by turns, such as strategy games. In this case, *Hive* is a real time game, which means that both the player and the autonomous agents take decisions simultaneously and continuously. This causes a problem because decisions can overlap, so after the agents chooses an action for the current state, it does not always lead into the same state, due to the fact that in that decision period, the player has taken his own actions. for example, if the state is that they are near and the player is about to die, the most effective action for the speeder is to attack him, because it would probably kill the player, which is the desirable state. Nevertheless, in hat decision period, the player could have moved, changing the game state, so the result of the same action in a determined state will be different each time. This provokes the learning process to be slow and, also, it complicates the convergence of the value Q matrix.

On the other hand, as the game is a shooter, there is a main feature that affects the learning process. This is the fact that the player can aim with the mouse to shoot to different points of the map, to destroy the speeders. In an ideal learning system, this should be taken into account so as the agent could avoid the player's attacks. Nevertheless, checking these cases, would lead to an exponential increment of the state space, due to the fact that it should register the weapon's type, its states (with ammunition or not and making a shoot or not) and the trajectory of the projectiles to check whether they are going to collide with the autonomous agent. To solve this issue, each time that the agent is reached by a player's projectile, its Q value matrix is accessed at the row of the state in which the last action was taken and in the column of that action and is reduced by one unit, to decrease the probability of taking that action again in the same state. This is very useful, for instance, in the case where the actors are at medium distance so the agent alternates between the actions move towards the player, flee from him and jump (attack), which generates the feeling that it is trying to avoid the player's attacks.

The last issue to outline about the learning process is that it needs a player. As this is an online learning approach its not such problem, because it makes sense inside the technique employed. Nevertheless, if the learning was off-line, this would be a problem because it would require lots of human resources who were taking continuous games with the machine, fact that is a expense of time and financial resources. One possible solution, would be to make the agents learn by fighting with the speeders which have a behavior tree, but as the player's actions differ from the speeders' ones, this would not be a suitable learning. Another solution could be to implement an artificial intelligence likeness the humans with a behavior tree, but this idea has been discarded due to the fact that the project's dimensions are too large, that it would be practically impossible to consider all the possible actions of the player.

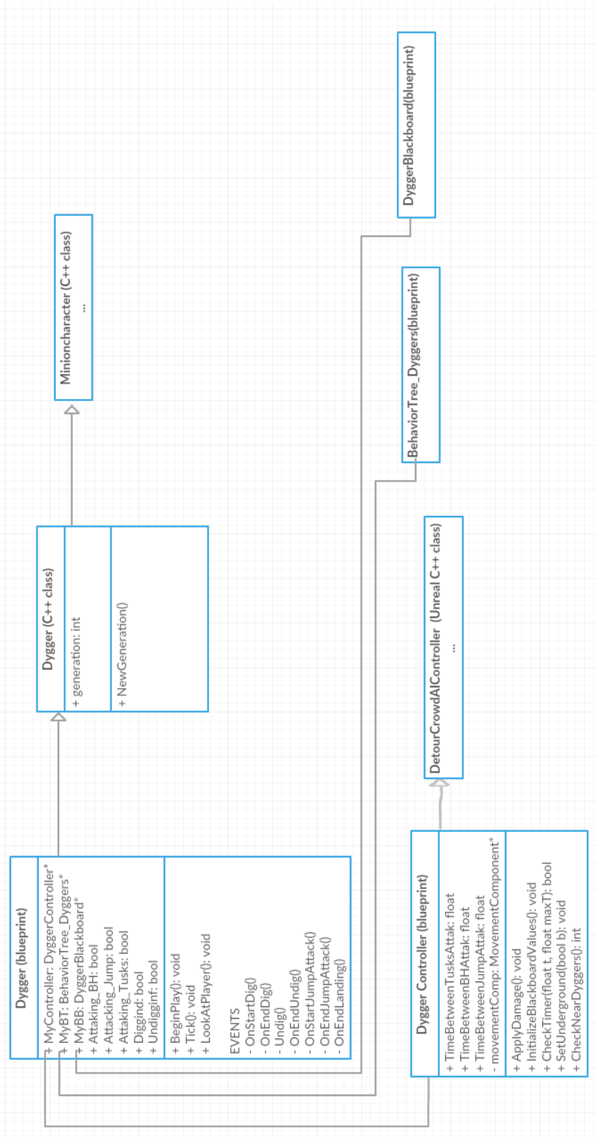

DyggerBlackboard(blueprint)
Minioncharacter (C++ class)
...
BehaviorTree_Dyggers(blueprint)
Dygger (C++ class)
+ generation: int
+ NewGeneration()
DetourCrowdAIController (Unreal C++ class)
...
Dygger (blueprint)
+ MyController: DyggerController*
+ MyBT: BehaviorTree_Dyggers*
+ MyBB: DyggerBlackboard*
+ Attaking_BH: bool
+ Attacking_Jump: bool
+ Attaking_Tusks: bool
+ Diggind: bool
+ Undigginf: bool
+ BeginPlay(): void
+ Tick(): void
+ LookAtPlayer(): void
EVENTS
- OnStartDig()
- OnEndDig()
- Undig()
- OnEndUndig()
- OnStartJumpAttack()
- OnEndJumpAttack()
- OnEndLanding()
Dygger Controller (blueprint)
+ TimeBetweenTusksAttak: float
+ TimeBetweenBHAttak: float
+ TimeBetwentJumpAttak: float
- movementComp: MovementComponent*
+ ApplyDamage(): void
+ InitializeBlackboardValues(): void
+ CheckTimer(float t, float maxT): bool
+ SetUnderground(bool b): void
+ CheckNearDyggers(): int

Figure 4.5: Dyggers class diagram

4.2 Dyggers

In this Section, the implementation process of the second enemy, the dygger, is described.

4.2.1 Dyggers Behavior Tree

The class diagram developed for this type of agent is shown in Figure 4.5. the inheritance is very similar to the speeder's one (Section 4.1.1), so only the two main classes will be focused: the dygger's blueprint and its controller. The blueprint dygger is the one which forms the agent, its mesh, its movement and the implementation of its actions. Furthermore, in this class, the management of the animations is performed by a finite state machine. In this case, the actions dig, undig and the jump attack have been implemented as events, specifically, two events for each attack. The first one is the action's start and this event is triggered from the leaf node of this task in the BT. At this moment, all the animation's control variables are set and it is determined if the agent is under the ground or over it (function *SetUnderground*). As mentioned before, this event controls the start of the animations, which also call the second event when the animation's execution has finished. At that event, all the variables are re-set, so the agent can go back to its *IdleRun* state (Figure 4.6).

To illustrate this sequence, we are going to see a clear example. Let's suppose that the BT determines that the dig node has to be executed. At the first moment, it checks whether the floor is rocky, because if it was the case that the floor is made of metal, the dygger could not perform the dig action and the node would return false to its selector, so the branch failed. If the raycast employed to check the floor's material determines that it is rock, the node triggers the event *OnstartDig*. This event sets the proper parameters to indicate the FSM that the state is going to change from *IdleRun* to Dig, which would provoke the animation to start its execution. This animation is linked to the *OnEndDig* event, which is triggered when the animation reaches its last frame. Finally, this event re-sets that variables again and calls the controller's *SetUnderGround* function to determine that now the agent in under the floor. There, the blackboard is updated and the collisions and visibility of the agent are managed, to make the dygger invulnerable under the ground.

4.2.1.1 Movement Dimension

In this agent, the movement problem has been solved the same way we did in the previous enemy: the agent has a movement component controlled by a pathfinding which avoids collisions with static obstacles and the player. To make the dyggers avoid each other and not overlapping, the AI *DetourCrowdController* has been employed, because it manages dynamic collisions in an optimal way, so computing resources are not wasted. On the other hand, as we mentioned previously, these enemies have two mechanisms to move: above the ground and under it. The first one consists of the normal movement, but to achieve the feeling that the agent is moving under the ground a hack has been implemented. Instead of moving the agent in its vertical axis to set it under the ground and then move it by means of projections of its navigation mesh positions, its visibility, collisions and animations are modified in order to give the feeling that the dygger is moving under the ground, when it is, in fact, moving normally.

4.2.1.2 Decision Dimension

Specifically, the following nodes and functions have been implemented.

- **DECORATORS**
  - IS TARGET SET? It consists of accessing the blackboard to check the reference to the player.
  - IS UNDER THE GROUND?
    Access to the blackboard to check the boolean that determines whether the dygger is under or over the ground.
  - IS UNDER ROCK GROUND? this condition has been implemented in a node, in which a raycast is projected from the agent towards its down vector, to check if the floor which it is colliding is the type of rock or metal, accessing its tag.
  - IS TIMER COMPLETED? For this node, the dyggers have a timer that indicates if they have to change their movement mode. It accesses the blackboard and checks the float which stores the last time is changed from one mode to the current mode, and it is compared with the maximum allowed value.
  - ARE THERE NO DYGGERS NEAR THE PLAYER? In this verification, the class of all the actors which are at a distance less or equal to 400 from the player is checked, to determine the number of dyggers near the player.
- **TASKS** Most part of the tasks require an animation, so it is essential the use of a finite state machine which controls the transitions between that states. It is shown at Figure 16. In this Section only the specific tasks of the dyggers are focused, because movement and blackboard

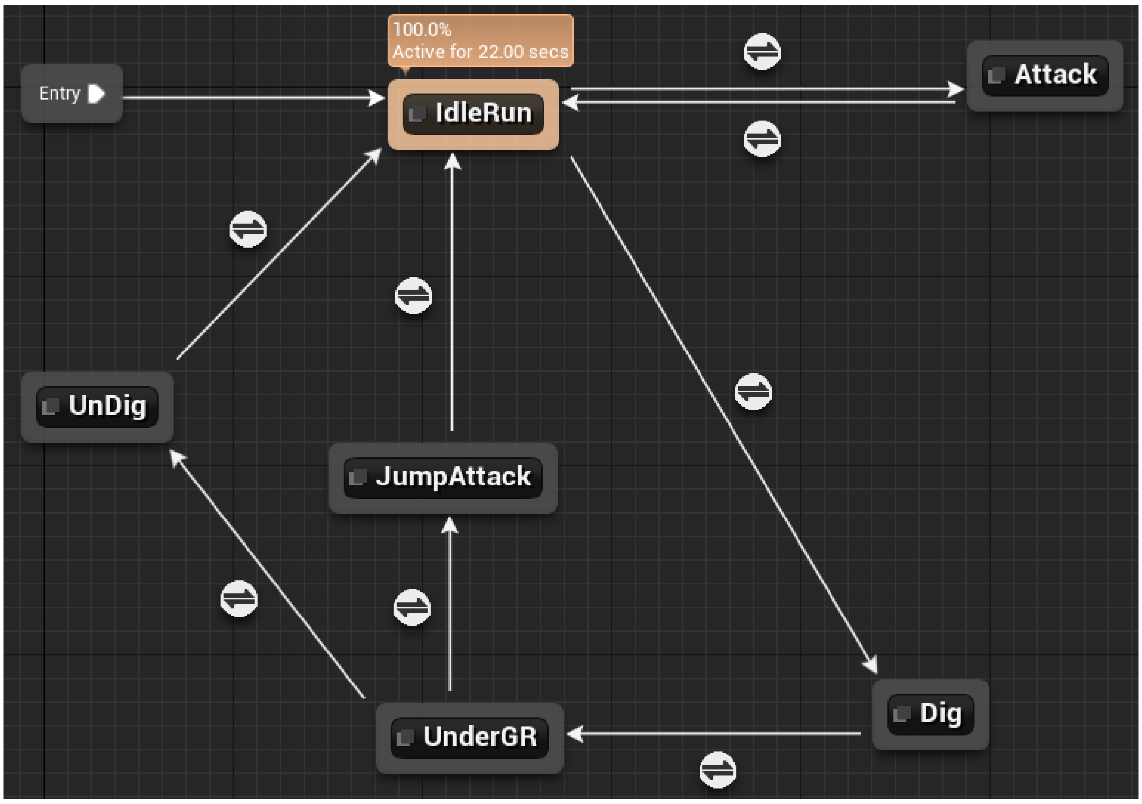


Figure 4.6: Dyggers finite state machine for animations

updating is made the same way that in the speeders.

- DIG
  The conditions to enter in this node are the dygger to be over the ground and its timer had expired. In this BT task, at first place, one raycast is launched from the dygger downwards to check whether the floor is rocky. If the process is satisfactory, in the blackboard, the value which stores the time under the ground is set to the current time. Lately, *OnstartDig* event is triggered and the animation is set. After that, *OnEndDig* event starts, to change again the animation and to update the blackboard boolean which stores the dygger movement state to underground, and modifies the collisions and visibility of the agent to ignore the player's projectiles.
  There is another dig node in the BT with minor priority located in a branch that checks if the dygger has been hurt. If so, it will flee from the player and dig in a safer place.

- UNDIG
  There are two reasons that will make the agent undig: to be under the ground with an expired timer or to be under the ground and its forward floor is metallic, which is impassable, so it must exit from the ground. Regarding the actions flow, it is very similar to the complementary previous action. the node triggers the event *OnStartUndDig* which launches the animation, and manages its collisions and visibility.

- JUMP ATTACK
  This attack is made when the dygger is under the ground and is near the player. This

attack has two different modes whether the player is over the ground or on the air. If it is on air, the dygger predicts the player's next position based on his velocity and direction, and then, projects that point to the navigation plane to move toward it. Whenever it reaches that predicted position, it is triggered an event that controls the jumping animation and also, an impulse is added to the agent in the vertical axis. If during this attack the distance between both actors is small enough to consider they have collided, the dygger damages the player. If the player is over the ground, the dygger just jumps from below him to hurt him.

- BREAK HUMANS ATTACK This task constitutes a constrictor attack, which is executed if and then the agent is located over the ground, at a near distance from the player and with no other dyggers near them. Then, it performs an attack of type constrictor which hurts the player and immobilizes him during a couple of seconds to allow other agents reach them and attack.

- TUSK BASIC ATTACK Finally, the basic attack is triggered if the agent is over the ground and there are more dyggers near.

As we can see, the dyggers behavior is far more complex than the speeders, because they have different cooperative strategies.

4.2.2 Dyggers Q-learning

The diagram class for this agents is shown in Figure 4.7. As we can see, the structure is very similar to the speeders one. The most meaningful differences are located in the class *DyggersSharedBrainMatrix*, in the reward and value functions and in the identifying state system.

The class *DyggerML* is responsible for the learning process and it has direct access to *DyggersSharedBrainMatrix* to get the values of the learning matrix. Note that there is only one instance of this class, so all the dyggers share the same knowledge, which is a save in terms of space resources.

4.2.2.1 Main core function

The learning algorithm is exactly the same as in the other type of agents. As said previously, this algorithm takes place in the tick function, each frame.

4.2.2.2 State identification

This case, the agent has 96 states which are identified by the combination of different enumerations:

- **Distance To Player**
  - NEAR: If the distance is less or equal to 100 units.
  - MEDIUM: If the distance is between 101 and 400 units.
  - FAR: If distance is greater than 400.
- **Speeder Health Proportion To Player**
  - PLAYER_ALMOST_DEAD: If the player's health is the 10 % of its maximum health. For the next states, we need to calculate the proportions of the player's health and the agent's health, dividing its current health between the maximum.
  - MORE_HEALTH_PROPORTION: The difference between the speeder's health proportion and the player's one is greater or equal than 0.25.

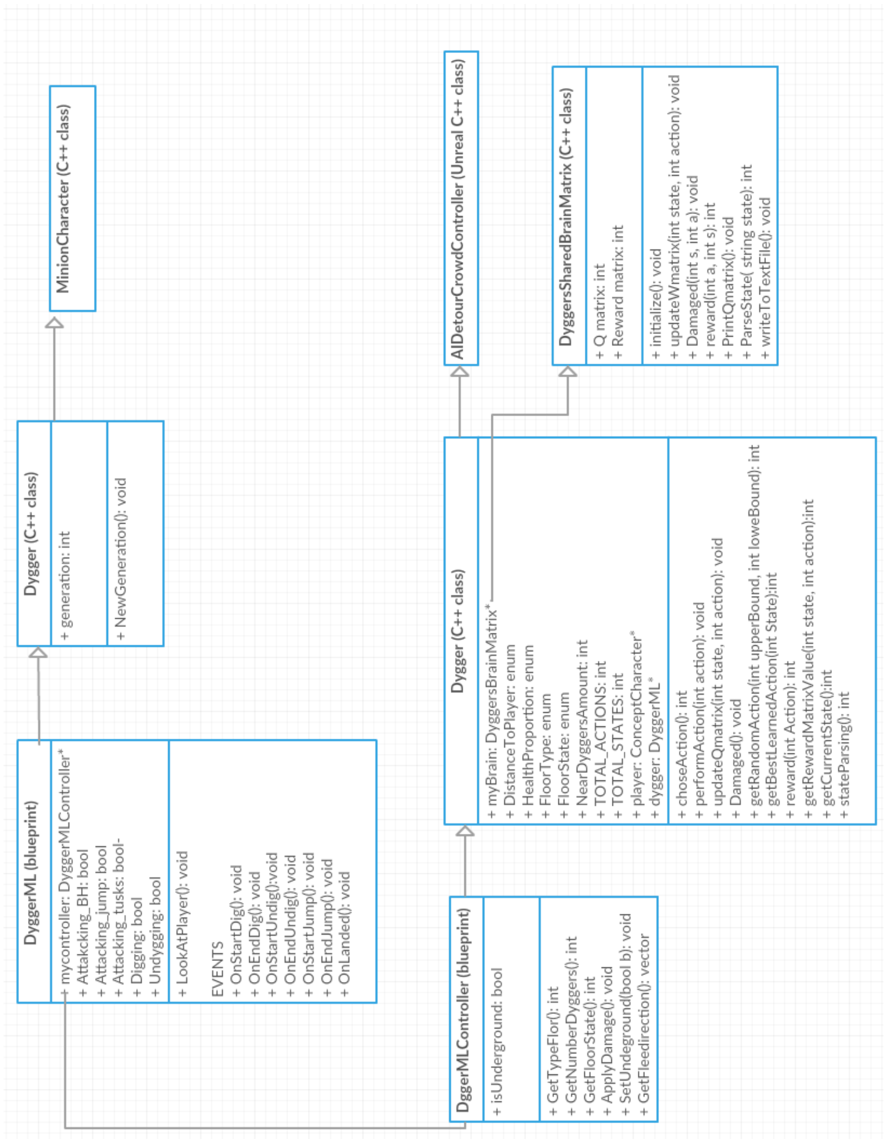


Figure 4.7: DyggersML class diagram

  - LESS_HEALTH_PROPORTION: The opposite of the previous one. The difference between the player's health proportion and the agent's one is greater or equal than 0.2.
  - SAME_HEALTH_PROPORTION: The rest of the states.
- **Floor Type**
  - ROCK
  - METAL
- **Floor Status**
  - UNDERGROUND
  - OVERGROUND
- **Near Dyggers Amount**
  - DYGGER_ALONE
  - MORE_DYGGERS

As said previously, the identifying state mechanism is different than the speeders' one, due to the fact that because of the huge amount of states, it is not suitable to make a bunch of nested switch statements. As we need a system to get fast an integer to identify the matrix's row by related to the current enumerations combination, the solution employed has been to use a map. First, each type of enumeration is set depending on the dygger's state. With that combination, a string is constructed. At the C++ map, there is a list of integers (rows identification) and their key, which is that string that identifies the state. This way, there is a really quick access from a state to the number of the row that it identifies, because maps are balanced binary trees, which means that a lookup to a key value has a cost of $O(\log N)$, with N equal to the number of entries.

4.2.2.3 Action selection, performance, Obtaining values and updating Q matrix

In this case, all this processes are done in the same way than the speeders, so it will be superfluous to explain them again.

4.2.2.4 Problems

In addition to the problems mentioned in the speeders section, the dyggers deal with another issue. As their behavior is way more complex than the speeders, they have several restrictions, such as the impossibility to dig if the floor is metal. These restrictions have been considered with the state identifying system, by giving them a heavy punishment of -100 points in the reward matrix. Nevertheless, as in exploration dyggers take random actions, there exist the possibility that they make "forbidden" actions, which affects the gameplay. This will be discussed thoroughly in Chapter 5.

V

Chapter 5

5. Results, Testing and Evaluation

In this Section, results of the projects are presented. First of all, a comparison between each type of agent will be described and its testing process. Finally, a task overview is going to be evaluated to see whether the planning in the technical proposal has been accomplished.

To evaluate the agents' effectiveness, the number of defeated agents per minute is going to be analyzed. The measurement employed consists in the greater value, the lesser is their difficulty to the player, so the challenge for him is not good balanced.

In an ideal performance of Q-learning, it is expected this value to be the lowest, which means that the agents are more difficult to defeat, so that can prove that they are learning. Furthermore, as it is possible that the average defeated agents per minute did not provide sufficient information to make a statement about the learning agents performance, in different games of five minutes, the number of defeated agents each minute is going to be compared. That way, it can be observed if there exists a difference from the minute 1, to each one of the following minutes. Also, as this paper is treating online learning, each time an agent dies, another one is spawned with the same value Q matrix, so the experience of the agents is going to be accumulative and shared during generations.

5.1 Results Speeders

Firstly, the results of the different techniques employed for the speeders are going to be analyzed.

5.1.1 Speeders: Results Behavior Trees

To determine the number of speeders per minute, five games of five minutes each one have been played. In Table 5.1, results of this games against agents whose behavior are managed by a behavior tree are show. The resultant average is 6.68 speeders per minute and 0.2 player deaths per game, which means that the player hardly ever dies during five minutes.

5.1.2 Speeders: Results Q-Learning Reward Pair Action-State

In Table 5.2, the results of the speeders with Q-learning based in a reward matrix per pairs action-state are shown. To analyze the results lately, they have been organized so that the number of

Table 5.1: Speeders - results behavior trees

GAME ID	Killed speeders	Player deaths
G1	31	1
G2	33	0
G3	36	0
G4	35	0
G5	32	0
Speeders/Minute		6.68
Deaths/Game		0.2

killed speeders each minute in the game can be observed, to then conclude if the number of them decreases with time, what will provide a direct feedback of the learning effectiveness. There is a decrement in the number of killed speeders among the time, from an average of 6.6 agents per minute, to 5.2. Despite this is a very small difference, the learning time has been only 5 minutes and it can be considered that the learning is effective.

Table 5.2: Speeders - results reward pair action-state

GAME ID	Killed speeders					Player deaths
	min 1	**min 2**	**min 3**	**min 4**	**min 5**	
G6	10	15	17	22	28	1
G7	9	15	22	27	32	1
G8	7	11	17	20	22	1
G9	4	7	11	14	23	2
G10	8	12	20	27	30	1
Avg.	6.6	12.4	17	22	27.2	-
Increment	6.6	5.8	4.6	5	5.2	-
Average Speeders/Minute						5.4
Deaths/Game						1.4

Now, an analysis of the value functions of the agents from that games is presented to see if they reach any kind of convergence. At Table 5.3, the first column refers to the movement towards the player action, the second is the attack and the third the fleeing from the player. Let's analyze the data collected, to extract some conclusions. There are five rows whose all column values are 0 which belong to the states:

- NEAR DISTANCE AND MORE HEALTH THAN THE PLAYER
- NEAR DISTANCE AND LESS HEALTH THAN THE PLAYER
- NEAR DISTANCE AND PLAYER IS ALMOST DEAD
- MEDIUM DISTANCE AND MORE HEALTH THAN THE PLAYER
- FAR DISTANCE AND LESS HEALTH THAN THE PLAYER

The possible reasons to that can be, on the first point, that is really difficult to the speeder to be near the player, because he can shoot projectiles and kill him from the distance.

In the sixth row, we can appreciate that the agent has learned that when it is at a medium distance from the player and has the same health proportion, the best action is to move toward him. Also, at the same distance, if it has less health than him, it prefers to attack to try to hurt him instead of catching him. Probably, this is due to the fact that this enemy has been designed to appear in hordes, so when there are various speeders and one is about to die, it worths to take a kamikaze strategy, and try to hurt the player before it gets killed.

The state which presents the biggest values is when they are at medium distance and the player is almost dead. In that case, the speeder prefers to chase him. Furthermore, when it is far and has more health than the player, the speeder prefers to jump and attack rather than chasing the player. This can be caused because when the agent has to go across a big distance to reach the player, the chances to get hit by a projectile are really large, so if he jumps, he can difficult the process to the player, as it can avoid some shoots. In this case, it seems that the agent has discovered a new function to a task that was not designed specifically to do that.

Table 5.3: Speeders - convergence speeders function value Q

Game 1			**Game 2**			**Game 3**			**Game 4**			**Game 5**			**Avg.**		
M	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	2	-2	0	1	0	0	6	0	1	3	-1	0	3.2	-0.6	0.2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	2	13	7	3	12	7	3	12	2	1	13	2	2	12.4	3.6	2.2
0	7	0	0	4	0	0	7	0	0	2	0	0	4	0	0	5.4	0
122	74	98	121	95	0	83	0	0	124	99	98	98	87	98	109.6	71	58.8
-3	98	0	-1	98	0	0	0	0	-5	98	0	-2	98	0	-2.2	78.4	0
0	1	0	2	0	0	2	0	0	5	0	1	1	1	0	2	0.4	0.2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
61	58	0	59	0	0	12	8	0	49	0	0	55	26	0	47.2	18.4	0

5.1.3 Speeders: Results Q-Learning with Reward per State

This time, our approach is to have a reward matrix, with values for states only, without taking into account the actions. Results (Table 5.4) are not very good, as the average number of speeders killed is around 7 per minute. With the pass of the time, the number of speeders killed is not reduced. On the contrary, it increases until four points, so it seems that the agent is not learning properly. Analyzing the convergence of value functions, shown in Figure 5.5 to determine what is happening in this case, it can be seen that the speeder is in the most part, learning to flee from the player and chase him in only two cases: when they are at a medium distance and the player is almost dead, and the same when he is far. Also, he will attack when it is far from the player and has more health. As we can see, this strategy has no sense, because it seems that the agent is waiting until the player has hurt himself with a bomb to give him the last attack.

Table 5.4: Speeders - results speeders Q-learning with reward per state

GAME ID	Killed speeders					Player deaths
	min 1	min 2	min 3	min 4	min 5	
G6	5	9	15	22	30	2
G7	8	19	24	27	32	1
G8	7	20	17	38	46	1
Avg.	6.67	16	18.66	29	36	-
Increment	6.67	9.33	2.66	10.34	7	-
Average Speeders/Minute						7.2
Deaths/Game						1.33

Table 5.5: Speeders - convergence for reward per state

Game 6			Game 7			Game 8			Avg.		
M	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**	**M**	**A**	**F**
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	-1	0	0	-0.33	0	0
-3	0	0	0	0	0	0	0	0	-1	0	0
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0
-1	11	0	-1	43	0	-1	-1	1	-1	17.66	0.33
0	0	0	-3	0	0	-1	0	0	-1.33	0	0
118	70	91	71	71	93	48	0	63	79	47	82.33
-1	114	0	0	118	0	0	0	0	-0.33	77.33	0
-2	15	0	-1	0	29	-1	-2	18	-1.33	4.33	15.66
-8	2	0	-6	0	0	-4	0	0	-6	0.67	0
77	32	39	66	50	52	0	33	38	47.67	38.33	43

5.1.4 Speeders: Results Random Agents

In Table 5.6, the results of different games with speeders which take random actions are shown. The average of kills of this agent is 8.5 per minute and the player has a probability to die around the 30% each game with a duration of 5 minutes.

Table 5.6: Speeders - results random agents

GAME ID	Killed speeders	Player deaths
G9	42	0
G10	46	0
G11	39	1
Speeders/Minute		8.5
Deaths/Game		0.33

5.1.5 Speeders: Results Conclusion

At this Section, a comparison of the results obtained with each type of speeder is presented. Regarding the average number of killed enemies per minute, the BT produced a result of 6.68 speeders per minute. The Q-learning system leads to 5.4 agents per minute. On the other side, the attempt of using Q-learning with a reward function that only had values per state, gives an average of 7.2 speeders per minute. Finally, random artificial intelligence produces an average of 8.5 killed speeders per minute. In order from the most useful method to the less one, it has been determined that:

Q-learning > the behavior tree > our custom Q-learning > random agents

On the other hand, regarding the times that the player had died per game:
Q-learning (1.4) > custom Q-learning (1.3) > random (0.33) > Behavior tree (0.2)

With these results, it can be confirmed that applying reinforcement learning via the Q-learning algorithm can be useful in video games, at least, in those where the enemies are not too complex.

5.2 Results Dyggers

At this section, we will compare the results of the second enemy implemented. It must be said, that as the attempt to use a reward matrix to only states failed in the previous agent, it has not been implemented in this type of enemy. For each type of dygger, another five games of five minutes have been carried through.

5.2.1 Dyggers: Results Behavior Tree

First, Table 5.7 shows the results of the implementation of a complex behavior tree. The average of killed dyggers is 2.88 per minute and also, the player dies 2 times per game.

Table 5.7: Dyggers - results behavior trees

GAME ID	Killed dyggers	Player deaths
G12	15	1
G13	12	2
G14	13	3
G15	17	2
G16	15	2
Dyggers/Minute		2.88
Deaths/Game		2

5.2.2 Dyggers: Results Random Agents

Random agents (Table 5.8) provide unacceptable results for a video game, as the player does not die during the game, and he can kill about 14 enemies per minute, which is a really big amount in this case.

5.2.3 Dyggers: Results Q-Learning

Table 22 shows the results of applying Q-learning to the dyggers. As we can see, the average of dyggers per minute is 5.8. Furthermore, regarding to the dyggers killed at each minute, we can see

Table 5.8: Dyggers - results random agents

GAME ID	Killed dyggers	Player deaths
G17	70	0
G18	68	0
G19	69	0
G20	70	0
G21	72	0
Dyggers/Minute		13.96
Deaths/Game		0

that despite there is an increase from the first minute to the second, from that moment to the final of the game, there is a great decrease of killed dyggers, an average of almost 7 dyggers less, which is a significant prove to show that the dyggers are learning from the players actions, because as time passes it is more difficult to the player kill them. Especial attention to passing from minute 3 to 4, in which the average indicates that not any dygger was killed in that transition, fact which is the first time to appear.

Table 5.9: Dyggers - results Q-learning

GAME ID	Killed dyggers					Player deaths
	min 1	**min 2**	**min 3**	**min 4**	**min 5**	
G22	5	12	17	23	30	0
G23	5	11	14	23	29	0
G24	8	15	22	28	33	1
G25	3	10	12	19	23	1
G26	4	12	18	25	30	1
Avg.	5	17.13	23.6	23.6	29	-
Increment	5	12.13	6.47	0	5.4	-
Average Dyggers/Minute						5.8
Deaths/Game						0.6

5.2.4 Dyggers: Results Conclusions

The testing process has determined that behavior trees provide an average of 2.88 dyggers per minute, nearly 14 random agents are killed each time and with Q-learning the player kills almost 6 dyggers per minute, so in this case, the order is:

Behavior tree > Q-learning > random agents

Despite behavior trees offer a better solution in this case, it is not arguable that the agents have learned from the interaction with the player, as they have become more resistant with time. Furthermore, there is the possibility this type of agents need more time to learn as they have complex behavior.

5.3 Comparison of both agents

All the results above, justify that Q-learning can be used in online learning for real time video games. In this case, it seems that it works better in simple agents, but it is essential to remark that the state space for both agents has been discretized as much as possible, instead of working with thousands of states as it is usually done in the training process.

Apart from observing that the number of killed agents per minute was decreasing as the time passed, determining that the agents were learning, we have seen that one of our agents has discovered a new function for a task which was not designed for: the speeder attacks to use the jump for trying to avoid getting hit by the player's projectiles.

Another advantage that provides reinforcement learning is this type of games is that hordes are more dynamic. For example, in hordes made with 10 speeders implemented with behavior trees, as they follow the same rules, when certain states are accomplished in the game all of them decide to make the same action such as attacking simultaneously. This was partially solved, by using randoms and delays to avoid them make the animation at the same time, but they did one after other. On the contrary, this issue does not happen when employing reinforcement learning, because despite the agents use the same matrix, they have their own process of action selection following the policy, which leads to a more realistic result.

Finally, despite behavior trees have a great performance regarding the number of agents destroyed per minute, they do not progress over time. From the start until the end of a game, the NPCs stay the same, so the solution to increase the difficulty of the agents in this case is to modify their attributes such as health or attack power, which can moderately complicate the game for the player. Nevertheless, this alternative does not modify the agents' behavior unless new rules are implemented. However, machine learning provides the solution to this problem, without the need of designing and programming different behaviors.

5.4 Videos

In this Section, different videos of the agents are shown. Each video is constituted by several clips that have been selected by its importance in the study.

- **Speeders Behavior Trees:** This video is named *SpeedersBT* and shows a horde compound by various speeders controlled by the behavior tree. The collision avoidance and the poisonous attack, which makes the player's health decrease during a couple of seconds can be observed during the fist part of the clip. Also, the simultaneity issue is shown.
- **Speeders Q-learning Reward per pair action-state:** *SpeedersRL_RperActionState* shows Q-learning working with the reward matrix per pairs action-state. During the first seconds, the agent is exploring random actions: It goes back and forth to the player, it stays near the player without attacking, etc. In minute 0:34, the agent tries to attack the player for the first time and, after that, it tries to reach the player and attack him another time. Even, it kills the player at minute 1:07. Once the player starts to shoot the agent more precisely, it discovers that the jump of the attack can be useful to avoid getting hit by the projectiles (minute 2:22). At the last part of the video, some speeders are put together and it can be seen that the horde is more dynamic than the one controlled by BT, because each agent takes the most suitable action for its state.
- **Dyggers Behavior Trees:** This clip is named *DyggersBT* shows the ferocity of this type of agents, as they make the player to be continually jumping to avoid their attacks and shooting them when they are over the ground.
- **Dyggers Q-learning Reward per pair action-state:** The last video *DyggersRL* shows the same agent implemented with reinforcement learning. It can be seen that the learning process is slower than for the speeders, as they have more states. After trying different options, the

first thing it learns is to cover itself by digging. This can be observed by the fact that after dying several times during 2 minutes, the first thing the dygger does when respawning, is digging, so it can avoid getting killed by the player. Nevertheless, it needs a lot more time to learn to attack from underground. At the last part of the video, some dyggers are put together with the player and it can be seen that they are not as difficult to beat as the dyggers controlled by a behavior tree. Probably, because they have not achieved the convergence for their value function in only five minutes.

5.5 Testing

The testing has been a great part of the project, as it has been an iterative process until their behavior was completely satisfactory. This has been an arduous task due to the fact that the game Hive is still under development, so it has been subjected to several modifications which directly have an impact on the proper functioning of the agents, which have caused several bugs that had to been solved. Once this was achieved, the second testing process was carried out to get the results of the different agents. This has been a really tedious process, as several games have been performed, and on top of that, much more games should have been played in order to get more realistic results and results from different people are needed, but this has not been possible due to the nature of the project. Despite having only five samples of each agent, as we have shown previously it has been possible to extract some conclusions.

Also, for Q-learning agents, a testing method has been implemented by means of storing the value Q matrix in a file, which has been really useful to analyze the learning process.

5.6 Evaluation of the project

In this chapter, the development process of the final project of the degree in video games' design and development is exposed. To clarify that, an overview task to task from the planning section in the first chapter is going to be detailed, to contrast if all of the tasks have been accomplished.

5.6.1 Phase 1 - Documentation

This phase consists of all the tasks needed to document the research done in this paper.

- **TPC** At the Technical Proposal Courses, which took place in the last week of January, the structure of that article was presented. Also, different tips for how to approach the project were provided.
- **TP** During the week after the assistance to that courses, the technical proposal was elaborated. The motivation of the selected theme was that I wanted to do a research in a currently open issue as machine learning in video games, because it was a greater challenge to try to implement online learning in a real time game, which has not been exploited in the industry than simply making another video game.
- **TM** The task related with the elaboration of this paper, has taken the month of June. To remark that this process duration has exceeded the estimated time in twenty more hours.
- **PDV and PDP** The project defense videos and presentation have been realized together, the las week of June.

5.6.2 Phase 2 - Research

As said previously, one of the largest processes has been the research, due to my lack of experience in the field, also the planning for this task has been suitable, because it has needed around the predicted 60 hours. The results of this phase have been exposed in the second chapter of this paper, state of art and it consists of different tasks:

- **BT** Despite having theoretical knowledge in behavior trees, as they were part of the set of themes in the artificial intelligence subject, I had not the chance to implement them before, so I needed to document about them. Also, as this project has been developed in Unreal Engine 4, a game engine which I had never used before an underlying learning process about this tool has been done in a short period of time. This process has been shown in Section 2.1.
- **ML** The most part of the research has treated machine learning, specifically, reinforcement learning, due to the fact that this was a completely new area for me.
- **MLV** The last part of the process was to investigate how video games deal with machine learning nowadays. For my astonishment, there are only a few video games which make use of learning techniques and if they do, the most part is for offline training. This matter magnified my interest in investigating if online learning can be useful to games.

5.6.3 Phase 3- Design of the agents

Once the different techniques to implement selected, it was time to design how to implement them in the agents. This issue is shown in chapter three, design. The process was divided into three subtasks for each type of enemy. The followed steps have been to decide the desirable behavior (CD1) for the first enemy, the speeder, and then design it with behavior trees (R1) and implement it. When the result was acceptable, it was turn to design the Q-learning (ML1) one and implement it. At the same time, improvements in the design of the behavior took place. After I was satisfied with the result of both agents I repeated the same process for the second enemy, the dygger. This time, the Q-learning design process (ML2) was easier, because I based it on the work I had previously done, but the conceptual design (CD2) and the design of the behavior tree needed more time, due to the wanted complexity searched for the behavior.

As we can see, the order of this phase has differed from the planning, but in this way, we have taken benefit from time, because as we waited to have one complete enemy in all its forms, before designing the second one, a lot of problems have been avoided, because of the experience. This can be proved because the process took less than the estimated time, between 15 to 20 hours less.

5.6.4 Phase 4- Development of the agents

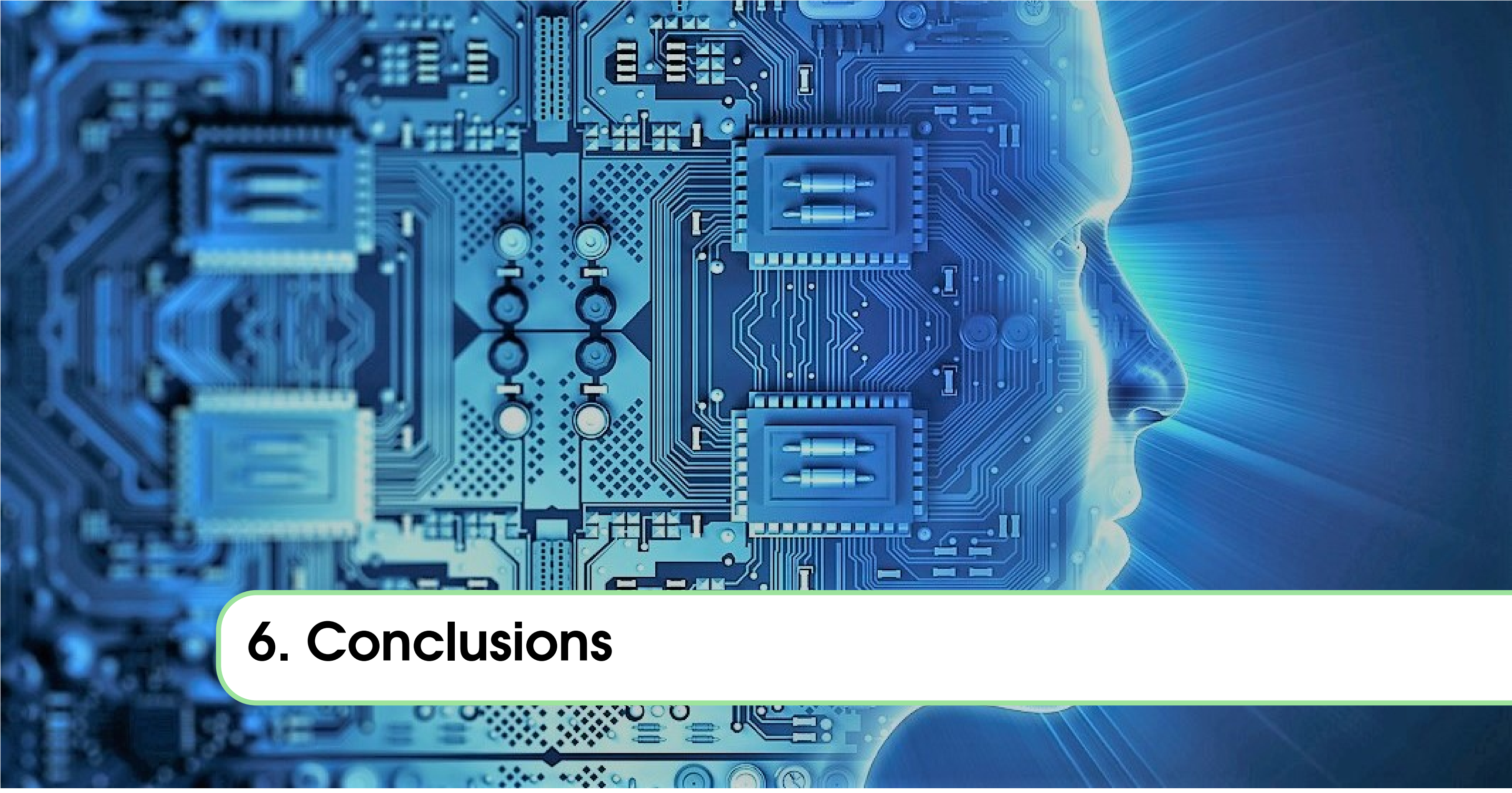
Just like it has said previously, this process has supposed a great effort, due to my inexperience with Unreal Engine 4. As this engine is far more complex than the one I had used before (Unity) at the same time I was implementing the agents, I was getting over a heavy learning curve of the tool in a very short time, because I could not delay the development of the agents. Also, a programming language that I had not too much experience with was used. Both reasons made that the implementation of the autonomous agents have taken about 40 hours more than I had predicted. Another drawback was that I had to familiarize with a large project such as Hive. All the tasks have been thoroughly explained in Section 4. As we said in Section before, their order has been different from the planning one. The actual order has been: development of behavior trees for speeders (IR1), implementation of the learning one (IML1) at the same time with the system to get the game state for it (SGS, which had resulted different for both enemies), implementation of BT for dyggers (IR2) and implementation of the learning dyggers (IML2) and its state identifying system (SGS). In the planning for this phase, which is shown in Table 4, another task was proposed: saving the learned information in a server (S). It has not been possible to achieve this task, because I needed to master network processes and I run out of time to learn it. Nevertheless, I have it prepared for future work, by saving the value matrix in a file, which could then be uploaded to the server.

5.6.5 Phase 5- Results

All the tasks related to the testing and results phase have been explained previously in this chapter.

VI

Chapter 6



6. Conclusions

This paper presented the comparison between the artificial intelligence behavior trees technique and Q-learning. This project not only included a deep research on the techniques and a comparison of them by means of related works, but it also covered the design and implementation of two different agents employing both techniques, extracting results and comparing them. All of this, made under a tight time schedule around about 300 hours.

First, we presented the Technical Proposal which described the work meticulously task by task to be done during this project. As we have seen in the chapter before all those tasks have been accomplished except the server storing mechanism. Also, objectives were described, so let's check if they have been achieved:

- **Comparison the different artificial intelligence techniques.**
  In chapter 5, results and comparison between behavior trees and Q-learning are shown.
- **Implementation of artificial intelligence techniques to create NPCs which simulate a human behavior.**
  This consists in a really ambitious and abstract objective, but if we consider that human behavior is not predictable by simple patterns as traditional AI is, we can aim that our learning agents accomplish at least in part, this objective.
- **Creation NPCs capable of interacting with different agents and environment elements.**
  Both agents interact with the environment and other agents, for instance, they interact with the player by attacking him, they also are capable of avoiding each other to avoid collisions between them and between static objects, dyggers can detect the floor in which they are over, to decide whether they can dig or not, etc.
- **Development enemies which modify their behavior to adapt to the player skills and his playing style.**
  This objective was related to the implementation of machine learning techniques, so this has been properly accomplished with the Q-learning as we have analyzed their value functions to determine the strategies that the agents had learned.

With all of this, we can conclude that the objectives of the project have been successfully accomplished.

Also, summarizing all the knowledge obtained:

- **Planning**
  As shown in the technical proposal, it is a good idea to make the planning of the project to end before the deadline, so we have some extra time if a delay it is produced, in this case, this time has been really useful to end the agents, as the UE4 learning process, has difficulted some tasks.
- **Behavior trees**
  This project has been really useful to master the behavior design of agents by means of behavior trees and also their implementation with the Unreal Engine 4 BT tool.
  - Performance and flow
  - Types of nodes
- **Reinforcement learning**
  This implies the major apprenticeship as this area was completely new to me. We can outline the following items:
  - The two most important solution methods using Temporal Difference are Q-Learning and SARSA.
  - The state space consists in the model of the environment containing all relevant information regarding the possible states.
  - An action space needs to be defined.
  - The reward function which gives a positive or negative reward when executing a specific action in a specific state.
  - The value function determines for each state which action yields the best reward over time.
  - The policy decides which action to take in a specific state.
  - The exploitation-exploration dilemma.
  - Despite literature discards reinforcement learning in shooter games, Q-learning can be in fact used in real time shooters if the agents have the same behavior and share their experience.
- **Problem solving**
  As this project was a bit ambitious, several problems have appeared during its development, and all of them have been solved by making research.

With all of this, personally, I think that I have successfully completed my goals, because I have created autonomous agents, which can learn in games of 5 minutes of duration.

Bibliography

[Ber11] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*. 2011 (cited on page 33).

[Cat17] Catness. *Hive: Altenum Wars*. 2017. URL: `http://hive.catnessgames.com/` (cited on pages 15, 36).

[Cha12] Alex J. Champandard. *Making Designers Obsolete? Evolution in Game Design*. 2012. URL: `http://aigamedev.com/open/interview/evolution-in-cityconquest/` (cited on page 34).

[17a] *Clash Royale*. 2017. URL: `https://clashroyale.com/es` (cited on page 15).

[Des15] Deven R. Desai. *Exploration and Exploitation: An Essay on (Machine) Learning, Algorithms, and Information Provision*. 2015 (cited on page 30).

[Doc] Unreal Engine 4 Documentation. *How Unreal Engine 4 Behavior Trees Differ*. URL: `https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/HowUE4BehaviorTreesDiffer/index.html` (cited on page 27).

[Dom15] Pedro Domingos. *The Master Algorithm: How the quest for ultimate learning machine will remake our world*. 2015 (cited on page 30).

[HP15] Amit Patel Harshit Sethy and Vineet Padmanabhan. *Real Time Strategy Games: A Reinforcement Learning Approach*. 2015 (cited on pages 30–32).

[Hec14] Chris Hecker. *My Liner Notes for Spore*. 2014. URL: `http://chrishecker.com/My%5C_liner%5C_notes%5C_for%5C_spore` (cited on page 26).

[Isl05] Damian Isla. *Proceeding: Handling Complexity in the Halo 2 AI*. 2005. URL: `http://www.gamasutra.com/view/feature/130663/gdc%5C_2005%5C_proceeding%5C_handling%5C_%7B%7D.php` (cited on page 26).

[McC] John McCarthy. *WHAT IS ARTIFICIAL INTELLIGENCE?* (Cited on page 25).

[Mun] Andrés Munoz. *Machine Learning and Optimization* (cited on page 28).

[Mur12] Kevin P. Murphy. *Machine Learning a Probabilistic Perspective*. 2012 (cited on pages 28, 29).

[17b] *Overwatch*. 2017. URL: `https://playoverwatch.com/es-mx/` (cited on page 15).

[Pix] Dynamic Pixels. *Hello neighbour*. URL: `http://www.helloneighborgame.com/-kickstarterru` (cited on page 35).

[Rab] Steve Rabin. *AI Programming wisdom* (cited on page 28).

[Sam59] Arthur L. Samuel. *Some Studies in Machine Learning Using the Game of Checkers*. 1959 (cited on page 28).

[Sna] Jamie Snape. *The Hybrid Reciprocal Velocity Obstacle* (cited on page 60).

[Stu03] Peter Norvig Stuart J. Russell. *Artificial Intelligence: A Modern Approach*. 2003 (cited on pages 25, 26).

[Stu] Intelligence Engine Studios. *City Conquest Game*. URL: `http://www.intelligenceenginestudios.com/cityconquest.htm` (cited on page 34).

[SB12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2012 (cited on pages 28–32, 34).

[Wex02] James Wexler. *Artificial Intelligence in Games: A look at the smarts behind Lionhead Studio's "Black and White" and where it can and will go in the future*. 2002 (cited on page 35).