USE OF MACHINE LEARNING TECHNIQUES IN VIDEOGAMES

Javier Villanueva Forner

Advisor: Dr. Raúl Montoliu Colás

Universitat Jaume I

This dissertation is submitted for the bachelor's degree of

Video Game Design and Development

ABSTRACT

This degree's Final Project consists in the implementation of machine learning techniques and its adaptation to a video game. The game has various types of AI which include traditional approaches and newer ones, like Montecarlo Tree Search and Deep Reinforcement Learning trained models. The game its a real-time strategy game built using Unity3D engine.

Keywords: Machine-learning, video games, adaptability, deep reinforcement learning, montecarlo tree search.

Index

LIST OF FIGURES

LIST OF TABLES

1 TECHNICAL PROPOSAL

1.1. Summary

This document explains the basics of the project that will be developed by the author during the following months based in finding ways of applying machine learning to video games. Machine learning is a group of algorithms and techniques that allow computers to learn to solve problems efficiently and effectively. The main idea is to explore some machine-learning mechanisms and apply them to a video game, adapting them to its flow and performance necessities.

1.2 Introduction and motivations

This project main objective is to implement machine-learning[i] mechanisms like Montecarlo Tree Search[ii] and Deep Reinforcement Learning[iii] in a video game. This techniques present some challenges in terms of adapting them to this type of applications that hopefully will be overcome.

The main motivation is the interest of the author in the subject. During the course of the current academic year, the concept of machine learning has captivated his attention and thus, encouraged him to explore it further trough this project. One of the objectives is the gain of knowledge in this area.

Another motivation is the experience of the author as a player. Hundreds of hours of gameplay have made the writer notice that the current IA development systems, although complex, are not as good as they could be when presented to players away from the average. Either because their skill level is too high or to low, the AI fails to adapt to them and making the game fun. An example would be Ratchet & Clank[iv], where enemies too easy to beat reduce the fun factor of the game. Segmented difficulties often leave players in the gaps between the steps they provide (easy – medium – hard). Progressive ones (those that can be scaled by the player) provide the player with the control of the world, and that not only affects the immersion, it also delegates the balancing difficulties to the player. Bottom line, a game should push the player to his or her limits, this limits being a difficulty hard enough to be challenging but not frustrating.

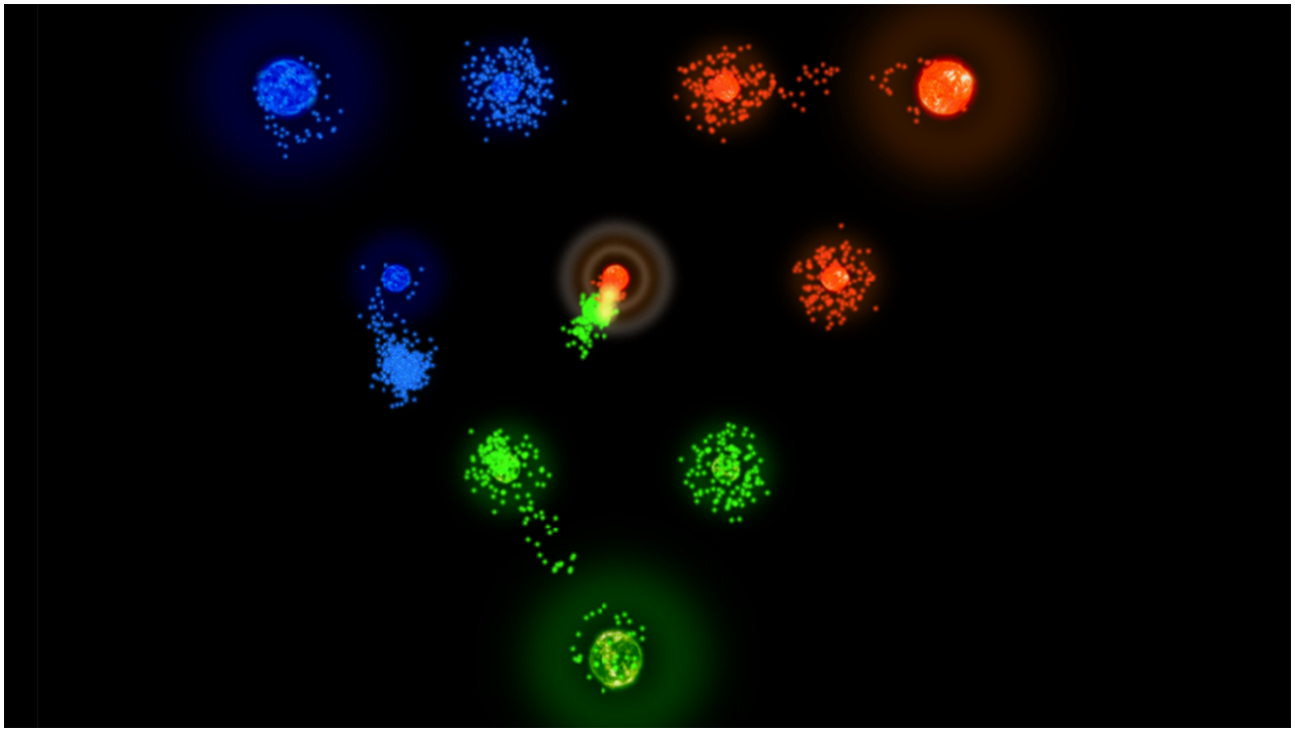


Figure 1: Image of the game Auralux.

The project will be developed following 3 main steps: developing the base game in which everything will be tested, implementation of the Montecarlo Tree Search and implementation and training of the machine learning AI.

Auralux[v] has been chosen as the base game in which this systems will be developed. The project will use a simplified version with the same main mechanics. We believe that this kind of games is perfect for the purpose of the project. They provide a simple set of rules and direct confrontation between players, in other words, an ideal environment to develop and test machine-controlled agents.

The game is simple: each level contains a determined number of stars. Some of them belong to a player (IA or human). The remaining ones are empty. The players can conquer stars sending enough units to their location. They can also fight each other the same way. When a unit is used, either fighting or conquering a star, it is destroyed. When a star is conquered, it will start producing units for the player that conquered it. To win the game, a player must be the only one with stars in its possession.

1.3 Related Courses

- Advanced Interaction Techniques (VJ1234) – This subject introduced different ways of interaction with the user. One of them was machine learning. It has served as the main inspiration source for the project.

- Algorithms & Data Structures (VJ1215) – The techniques that will be implemented will require appropiate data structures and its optimization, besides the implementation of the algorithms. This is why this course is related with the project.
- Programation I & II (VJ1203 – VJ 1208) – The main focus of the project will be the implementation of a game and AI/Machine Learning techniques, so programming is a basic skill required.
- Artificial Intelligence (VJ1231) – AI techniques will be used in the project to compare them with more modern machine learning aproaches.
- Game Engines (VJ1227) – The game will be developed using Unity game engine[vi], which usage was explained in this course.

1.4 Objectives

- Using machine learning techniques in the game.
- Using Montecarlo Tree Traversal and Deep Reinforcement Learning as AI in the game.
- Create a basic game based on *Auralux.*
- Create a fun and challenging game.

1.5 Expected Results

- Research of methods of adapting machine learning techniques to video games.
- Implementation of Montecalo Tree Search and its integration in the game flow.
- Usage of Deep Learning as an AI in the game.
- Difficulty adaptation.

1.6 Tools

- Unity 3D (2017.2.0f3)[vi].
- Tensorboard (1.6.0)[vii].
- TensorFlow (1.4.0)[viii].
- ML – Agents plugin for Unity (0.3.1a for Unity – 0.3 for Python)[ix].
- Python (3..6.4)[x].

1.7 Scheduling

Table 1: Scheduling

ID	Task	Description	Estimated Hours
G01	**Game design**	Implementation and testing of a simplified version of the game *Auralux*.	40 hours
G02	**Training version of the game**	A simplified version of the core game designed to execute without graphical elements as fast as possible.	20
MT01	**Montecarlo Tree Search Research**	Research to learn the basics of the method and to prepare its implementation	10 hours
MT02	**Montecarlo Search Implementation**	Implementation and testing of this machine learning technique in the game.	40 hours
MT03	**Montecarlo Search Game Adaptation**	Investigation of methods to adjust the method to gaming to obtain adapted and fun behaviours.	30 hours
DL01	**Neural Network Research**	Research to learn the basics of the model and to prepare its implementation.	15 hours
DL02	**Neural Network implementation**	Implementation and testing of this machine learning technique in the game.	75 hours
DL03	**Neural Networks Adaptation**	Investigation of methods to adjust the method to gaming to obtain adapted and fun behaviours.	30 hours

Note: The remaining 40 hours will be used for the following tasks:

1. Redaction of project documentation. This task will be done continuously during the project. That is the reason of its exclusion from the planification table (ID DOCU).

2. Security margin: If one of the mentioned tasks took longer than expected, this hours will be used to complete it without affecting others.

Note 2: If the project development is faster than the predictions, the spare hours will be dedicated to finding new adaptations that use both methods.

Note 3: This note has been redacted after the end of the project. The documentation task took 79 hours. This has been longer than expected, but necessary in order to keep the memory accessible to most people, even if they don't know about programming or machine learning.

2 DESIGN

2.1 Introduction

Lux Aura is a simplified version of Auralux[v], launched in 2013 by E. McNeill and ported to different platforms by WardRum Studios. The version proposed is a simplification of the game in terms of game play mechanics and in graphics. This is due to the objective of the game: the objective is to create a controlled environment with simple but firmly defined rules to train and execute the machine learning mechanisms that will be implemented later. Because of this, elements that would be included to make it more attractive to users (like fancy graphics) will be ignored. Our approach will also include the possible implementations of the AI techniques that will be programmed, as well as how they will be tweaked to adapt them to the player level.

In this document the concept "player" (or "players") will be used. Unless it is explicitly specified (case in which the term "human player" will be used), the expression will refer to all the agents with capacity to play the game. The term "agent" will also be used, referring the concept of an autonomous agent with a certain behavior that interacts with the space within the game where it is.

2.2 Gameplay

Lux Aura is a real-time strategy game in which the players fight for the control of a planetary system. The objective is to be the last entity alive, that is, the last player that controls at least one planet. To achieve this, the players have to conquer and attack each other by ordering their units what to do.

These units are generated periodically in the conquered planets. Each planet will produce a certain number of units every time a certain time passes. These units will belong to the player that controls that planet. If a planet is not controlled by anyone (i.e. that planet is neutral neutral) no units will be produced. The number of units a planet produces depends on the level that planet has. All planets start at level one, in which they will produce two units per second. Some planets can be upgraded by ordering units to do so. If a planet levels up, the amount of units produced per second will increase.

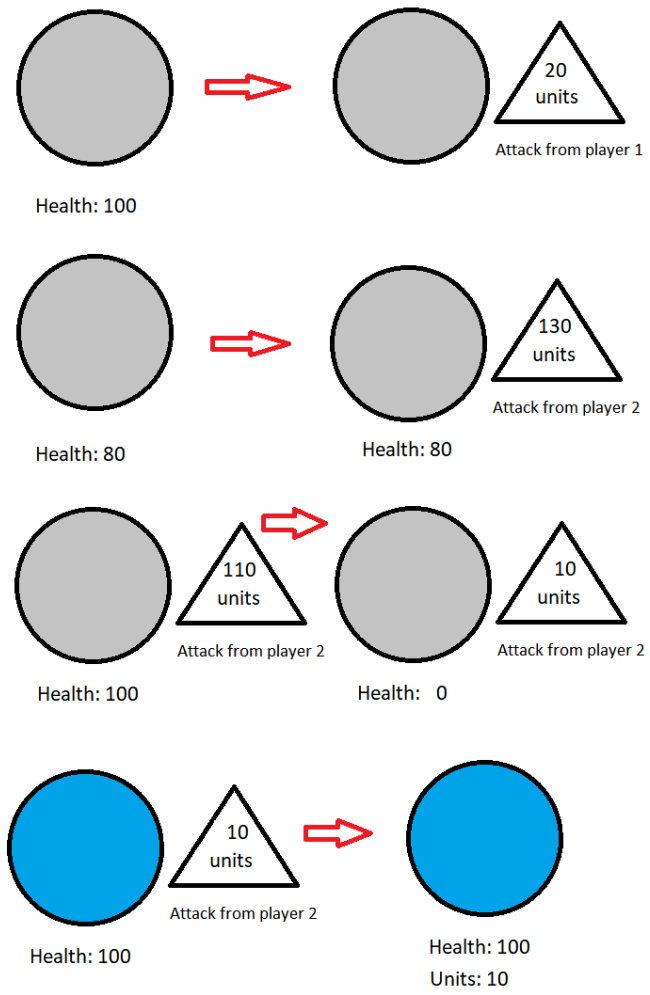


Figure 2: Attack to a neutral planet (grey) step by step. Player one attacks first but does not conquer the planet. Player 2 attacks and "overpowers" player one attack (by restoring the health player 1 attack took) and then conquers the planet.

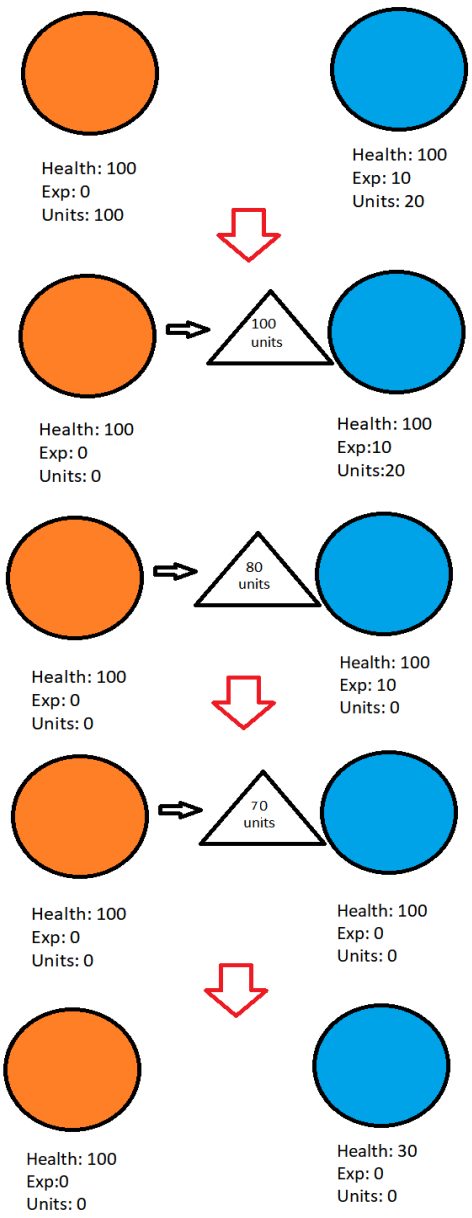


Figure 3: Example of an attack

Each player has at least one planet under its control while it's still alive. The units that are currently in those planets can be controlled and have to be used by the player to achieve his goals. Units can be sent to enemy planets by selecting the desired number to conquer them. They can also be sent to a planet that belongs to the player to upgrade it or heal it or simply wait there until the player decides to use them.

Conquering, attacking, healing or upgrading planets has a cost. A unit can destroy another one, decrease by one the health points of a planet, increase its health points by one (if the planet belongs to the same player as the units) or increase by one the experiences points of a planet (again, if the planet belongs to the same player). Once one of this actions is performed, the unit will be destroyed. For example, let's assume that we have a planet controlled by player one with one hundred of units in it and

another controlled by player two with only twenty. If player one attacks player two with his one hundred units, the units in player's two planet will be destroyed and twenty of the one hundred player's one units will be as well. That means that the eighty remaining units will decrease the same amount of health points to player's two planet and then they will be also destroyed. This example can be seen in Figure 3.

Planets have three main attributes: health, experience and their current units. The first one is used to conquer the planet, the second one to upgrade it and the third one stores the units that are currently in that planet. Units increase periodically, but to modify the other two values, players must spend units in a planet. The health will always increase before the experience and this order will be inverted when the numbers decrease. In other words, if a planet has some experience and full health and it is attacked, the experience will suffer damage first and, if it reaches zero and the attack continues (i.e. there are still some units attacking), the health will decrease. If the player spend units to upgrade a planet, first the health will be restored and only if the health is at its maximum value the experience will increase.

Planets can belong to a player or be neutral. Neutral planets won't produce units and will have health points that players will need to reduce (sending units) to conquer them. Planets that belong to a player will produce units and are harder to domain, because before reducing their health points, the units and experience points have to be eliminated first.

It is important to notice that if a player attacks a planet but does not conquer it, the other player will have to "overpower" their attack before trying to conquer it. This translates to what can be seen in *Figure 2*. The health points that other players reduced have to be restored before conquering the planet.

To make the game more friendly for the player, each player will have its own representative color. Planets and spaceships will be of this color, to identify them easily. When a planet is neutral, its color will be grey until it is conquered (Figure 2).

2.3 Mechanics

The mechanics of the game define what can happen or be done while playing it. In our case, these are centered in the units. The main mechanism players have to interact and change the state of the world are their units. Thus, the main mechanics are:

- Unit Generation: Periodically, units will be generated in every planet that belongs to a player. The number of units that planet has will increase depending on that planet level. No player interaction is required for this to happen.
- Upgrading Planets: If a planet can be upgraded, once enough units have been sent to it and the experience reaches the required amount, the planet will level up. This will imply a minor visual change (the planet will grow in size) and will

increase its units production ratio. This ratio, as well the experience needed to level up the planet, will evolve as follows:

  - Level 0: 2 units per second (initial level).
  - Level 1: 4 units per second (100 experience points required).
  - Level 2: 6 units per second (150 experience points required).

- Unit Selection: Each player will be able to select the number of units that needs from the planets that controls. The number of units that can be selected goes from zero to all the units available in the planet, and this can be extended to every planet the player has under his domain.
- Unit movement: Once some units have been selected, they can be sent to any planet in the system.
  - If that planet belongs to the player, the units will heal it, increase its experience points if the health it's at its maximum value or just be stored there if the health and experience can't be increased.
  - If the planet belongs to another player or it is neutral, the sent units will destroy the units that are stored in it, decease its experience points and lastly, decrease its health points, again, as seen in Figure 3.

  These attacks will manifest in a spaceship that will move from the attacker to the objective. They will become effective once the spaceship reaches the objective. It is important to notice that, while in the original game units can collide with each other and are shown independently, in *Lux Aura* the units move in huge space crafts and do not collide with each other.

- Conquer planets: If there are enough units sent to a planet that belongs to another player that the health points of it reach zero, the planet then will become neutral and, again, if its health points reach zero (*Figure 2*), it will be conquered by the player that sent the units. When a planet is conquered, it will turn the color of the player that conquered it.
- Camera movement: To be able to see the entire system, the camera moves in a plane parallel to the plane that contains the planets and can be zoomed in and out.

A diagram of these mechanics and how they relate to each other can be seen in Figure 4.

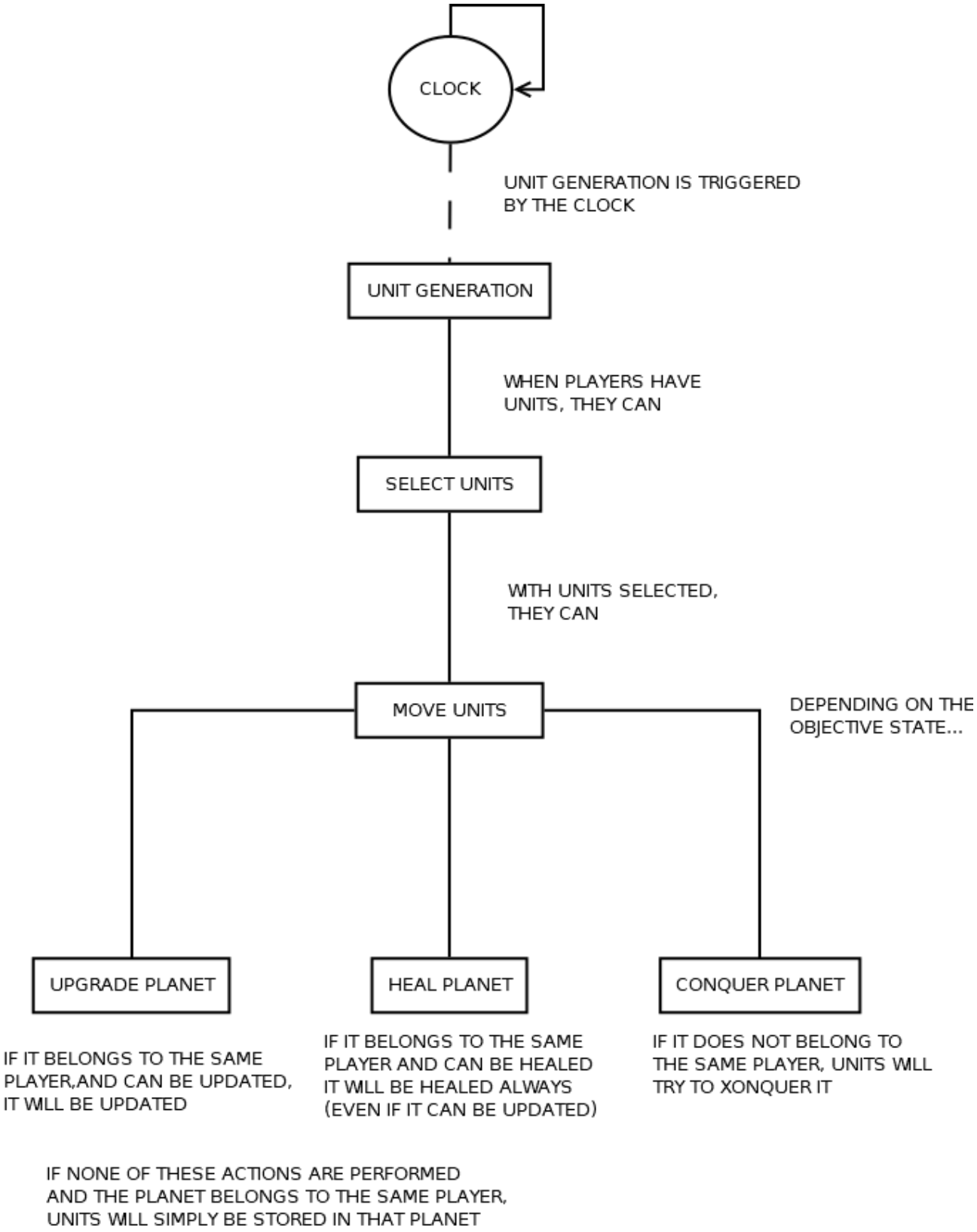


Figure 4: Mechanics scheme

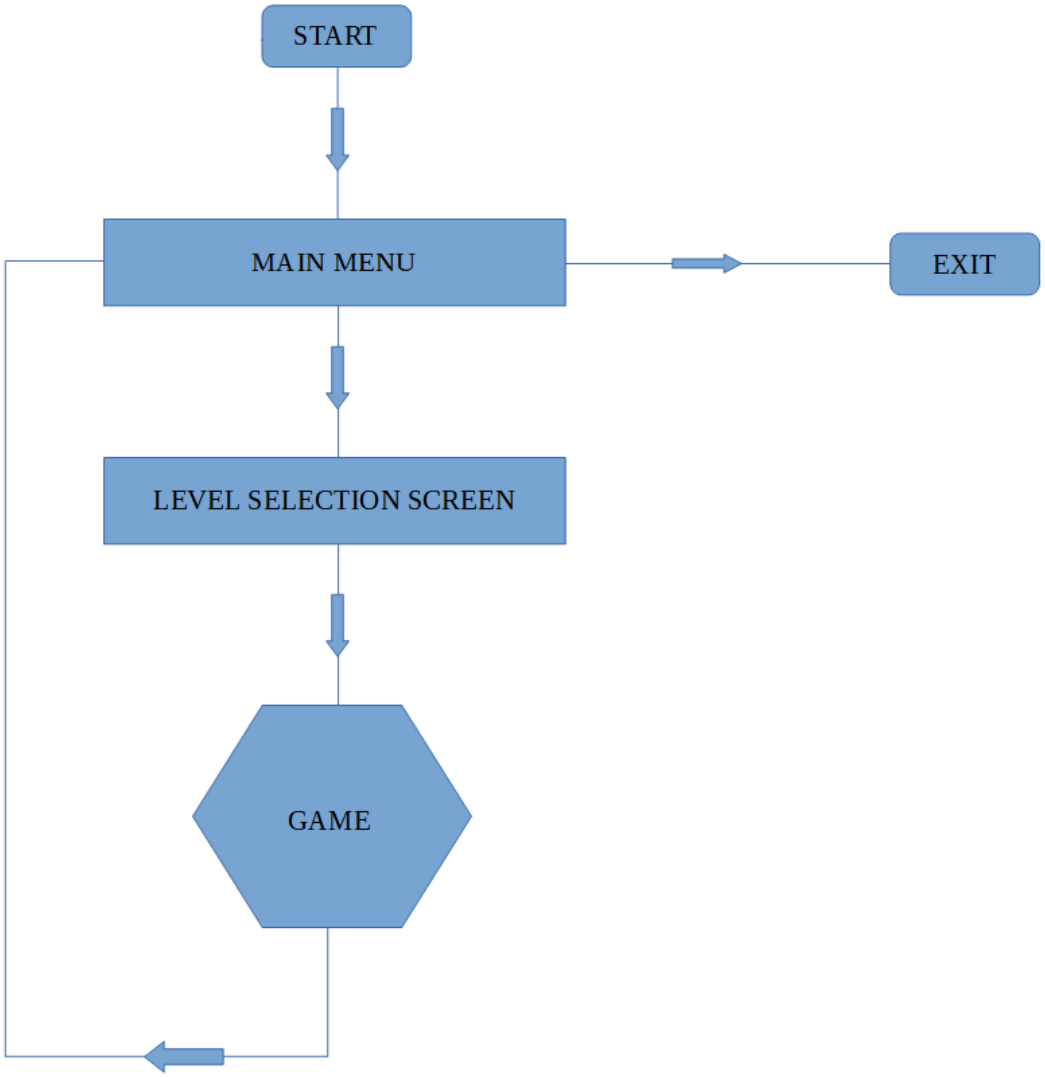


Figure 7: Flow of the game

2.4 Controls

The human player will control the game using mouse and keyboard. This will allow for displacement and unit selection and control.

Moving the mouse to the edges of the screen will make that camera move in that direction. This will allow the player to move freely around the map. If the left control button is pressed, the camera won't move.

If the player scrolls the mouse wheel up and down or presses the "w" or "s" keys, the camera will zoom in and out respectively. This will allow the player to control the amount of the map visible at any moment.

If the player left-clicks in a planet that belongs to him, 1/5 of that planet units will be selected Those unit will be subtracted from the planet and added to a text that will be

placed near the cursor, so the amount of selected units is visible at any moment. If he or she clicks with the right button, all the units of the planet will be selected.

If there are units selected and the player clicks in an empty area, they will be deselected and returned to their planets. If a planet that belongs to another player is clicked, each planet will send the units that were selected to that planet. To do the same with a planet that already belongs to the player (to upgrade or heal it), it has to be left-clicked while holding the left control key.

2.5 Levels

The levels that can be found in the game are quite similar. Set in space, they represent systems of planets and stars. The planets that can be found there are the only intractable elements in the game. This are the agents that will store the units and that will be conquered and upgraded.

Their composition is simple: a group of planets that are distributed through the map. Some of these planets will belong from start to a player while other will start being neutral. The distribution itself is quite meaningless. Stars, background and other elements besides from planets that can be found are only there for decoration.

Figure 8 shows examples of levels in the original game and in our own game.

2.6 Flowchart

In Figure 7 the main flowchart of the game can be seen. The game will start in the main menu. There, the human player will be able to choose to play the game or to exit. If he or she wants to play, the level selection screen will be shown. This screen will display the different levels of the game, including the training one. Once one is chosen, the level will load and he game will start. The player will be able to exit the game and return to the main menu at any moment if he pauses the game and chooses to do it. If the game is won or lost he will also be redirected to the main menu.

2.7 Camera and graphics

In this section discusses the graphics of the game and how the player will see the world. It is necessary to remember the reader that, since our game is a copy of another one and its main purpose it's not to entertain, but rather to serve as a support for our machine learning techniques to develop, the graphical part of the game will be simplified.

The colorful scenarios of *Auralux*[v] will be substituted with spheres and numbers, because the main objective of the project is the implementation of machine-learning

techniques, not the creation of the game itself. This will allow for a better demonstration of the decision making level that the machine has been able to achieve, because the total number of units will be easily readable at all times. This will be harder if the original approach would have been followed.

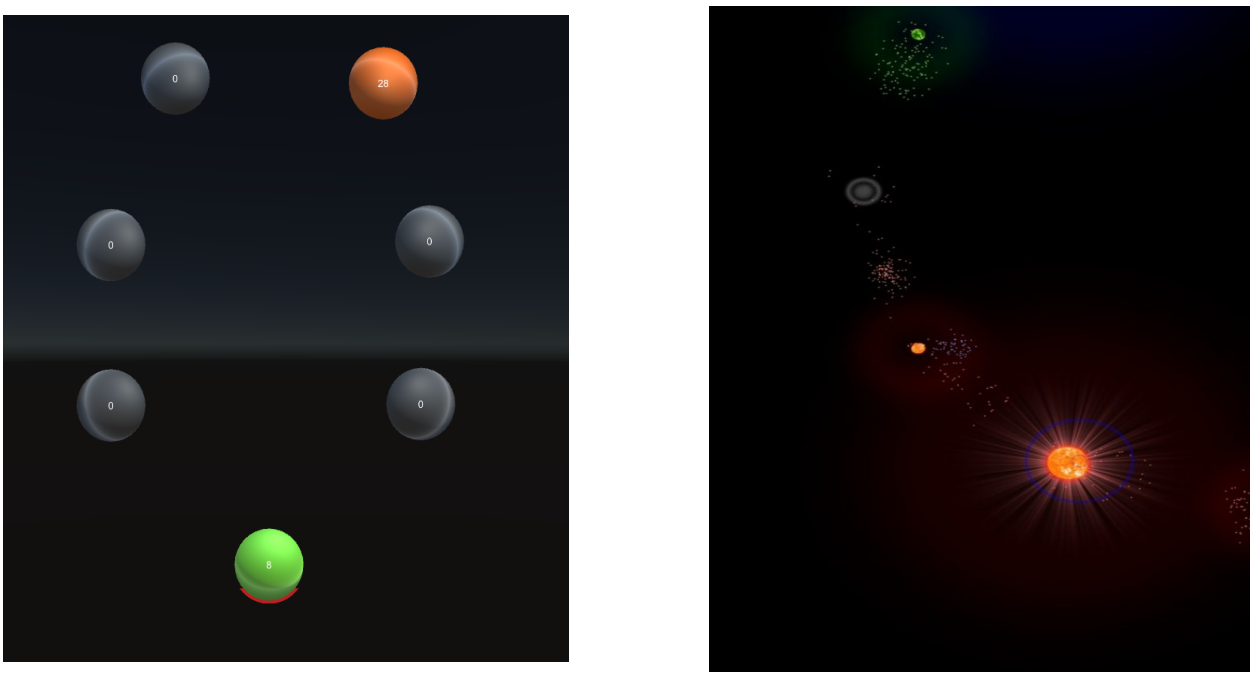


Figure 8: Example of AuraLux level (right) and a recreation (left)

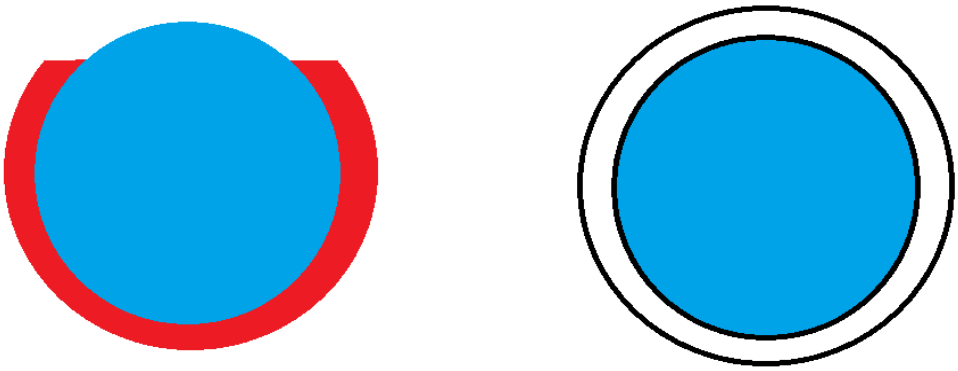


Figure 9: Experience and health indicators wraping the planet in blue. The experience one is not full like the health (right) one.

Human player will be able to see the world through a camera that will cover the whole level. It will be situated in front of the level and will point at it perpendicularly. As it has been mentioned before, It can be moved by moving the pointer close enough to the limits of the screen. The camera will have limitations to avoid moving too far away from the planets.

The units will never be visible. They will be shown as numbers in the planets (a number that will display the total units a planet has at the moment) and as a spaceship with

another number when they move (the number showing how many units are being transported in the ship).

Health and experience of the planets will be shown using a circle that will wrap it. This circle will fill as the health and experience increase or decrease. There will be two: one for the health (white) and another one for the experience (red). These indicators will only be displayed when the health of a planet is not full or when the experience of a planet is not zero. Figure 9 showcases this aspect.

3 DEVELOPMENT

Having explained the elements that compose the game now we will expose how these have been implemented to construct the final game. We will follow a bottom-top road in which we will start explaining the most basic elements of the game and advance towards those more complicated.

The most basic elements are the classes. These form the basic objects from which the game is built. After that, classes that derive from the basic ones will be treated, as they form the complex objects that are present in the game world. Finally, the relations between these objects, and thus, how the game works, will be treated. The obtained results will also be commented in their respective sections.

3.1 Normal Game Classes

These conform the most basic entities of the game. Most of them are designed to be expanded through heritage by other classes. They conform the very core of the game and are responsible of the communication between entities of higher complexity. Because of that, they are very simple and wide-purposed, they accomplish simple functions and have to be used in contexts that can differ drastically between them.

Some of these have *MonoBehaviour*[xi] as their parent class. This implies that they cannot be instantiated. Instead, they have to be inside an object in the scene and will execute some functions following the *Unity* execution order[xii]. Despite this, they can have their own methods and attributes and those can be accessed and called like a normal class. These methods whose call is controlled by Unity will be marked with the characters "MB", so the reader can easily identify them.

The classes will be presented in an order that we believe is optimal for its comprehension, since they are introduced as they relate to each other, starting with the central class and switching to the ones more closely related. We hope that this helps the reader to understand easily how the game has been implemented.

The classes related to the machine-learnings techniques that have been implemented will be covered in their respective section. A diagram of the classes can be found in Figure 10. Structs[xv] and Enums[xxi] have not been included to make it more easily readable.

Table 2: ClockEventReceiver methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
Tick	`EventType`	-	To be called when a clock event is fired.

Table 3: Clock mehtods

NAME	ARGUMENTS	RETURN	DESCRIPTION
AddListener	ClockEventReceiver	-	Subscribes the provided element to the events, so when they are fired, the object will be notified.
Awake (MB)	-	-	Two timers are created: the AI and the main clock. The first one ticks every 4 seconds (4000 milliseconds) and the second every 0.5 seconds (or 500 milliseconds). Both reset automatically once the event is fired and are active until the game ends.
OnApplicationQuit (MB)	-	-	Stops the timers.
OnMainTick / On AI Tick	Object ElapsedEventArgs	-	When one of the timers ticks, the corresponding function is called. These start a new thread that spreads the event trough all the entities that have subscribed to them, so they can perform the corresponding actions in response to the tick. The first one fires a MainTick event, whilst the second one fires an AITick one.
RemoveListener	ClockEventReceiver	-	Deletes a listener. Since the moment this fnction is called, the object will stop being notified of the new clock-related events.
Stop	-	-	Same as OnApplicationQuit, but is not called by the engine.

GlobalData

We start with this one because it is used almost by every other class. It is simply an information storage. It doesn't have any methods and it's not meant to be instantiated. Contains variables that are used by other classes and thus, is recommendable having in one place accessible by all.

ClockEventReceiver

A base class that derives from MonoBehaviour. This class is abstract, which means that it cannot be instantiated or appear in the scene by itself; instead, another class has to implement it. The purpose of this class is to standardize the relation of the clases in

the game with the clock. No matter what class implements this one, thanks to *polymorphism*[xiii] it can be treated as a `ClockEventReceiver` object and thus, simplify the interaction.

If an object wants to interact with the clock (subscribe to its events, be alerted when these are fired) it needs to implement this class.

Its method can be found in Table 2.

Clock

The name of this class says it all. It is the one in charge of making the game advance and evolve through time. It is basically a clock that ticks every 0.5 and 4 seconds. It extends the MonoBehaviour of Unity, so it needs to be in the scene to be used. There is only one Clock per scene.

We understand a tick as an event that triggers a response in the internal logic of the game and that fires when a determined interval of time has passed since it was activated. This class uses the *C# Timer*[xiv] class.

Its methods can be seen in Table 3.

EventType

An enumerator used[xxi] to identify the two types of events that can be fired:

- Main Tick: Fired every 0.5 seconds by the Main Timer.
- AI Tick: Fired every 4 seconds by the AI Timer.

The main idea is that the main tick is used to create units and other basic operations. Their cost is cheap and can be executed without overloading with tasks the main flow of the engine. It also controls the peace of the gameplay, as explained before.

The AI timer triggers the execution of the different AIs. This event has been separated for a couple of reasons: first, these process are very expensive and, like in the case of Montecarlo, can can take several seconds; second, it makes no sense to trigger them so frequently as the other event because the AI will most probably react taking the same actions and thus, no new behavior will be produced.

In further detail: the changes in the state of the game between the main ticks are so little that the AI will react by performing the same actions over and over until enough events have passed to change the world in a sufficient manner for the AI to react differently. These times the AI is triggered and reacts by doing the same it is already doing (in other words, by doing nothing) are a waste of resources.

Table 4: EventEntity attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
Conquerable	`Boolean`	True - False	Indicates if the object can be conquered by a player
CurrentContestantId	`Int`	-1 – 5	When a planet is neutral, a player has to send enough units to conquer it by itself. If another player send units, he has to eliminate all the rivals units first before being able to begin the conquest. This variable stores which player is currently claiming the planet so other attacks can be treated accoridingly.
CurrentExp	`Int`	0 - ExpForNextLevel	Current experience points
CurrentHealth	`Int`	0 - MaxHealth	Current amount of health
CurrentLevel	`Int`	0 – MaxLevel	The level the entity has at a given moment
CurrentPlayerOwner	`Int`	-1 - 5	Id of the player in control of the entity
CurrentUnits	`Int`	0 – 9999	Units the entity has at a given moment.
ExpForNextLevel	`Int`	0 – 1000	Necessary units to upgrade the entity to its next level.
GenerationRatio	`Float`	0 – 10	Units produced per second.
MaxHealth	`Int`	0 - 300	Maximum amount of health possible in a given moment for this entity.
MaxLevel	`Int`	0 - 5	The maximum level achievable in an entity.
UnitsToConquer	`Int`	0 - 1000	Number of units necessary to conquer the entity.

EventEntity

Holds the basic and common information to all the entities that can be placed in the map. The basic idea behind this class is to provide methods to interact between entities without specifying how each type will react and thus, allowing for different behaviors. Because of this, `EventEntities` can't be spawned in a level, but instead serve as a base class for the rest of the entities.

For example, we can have stars and planets in our level. Both have a certain amount of health and can be attacked and conquered. However, the star can react differently to an attack than the planet. By deriving from this class, both will have a "`SufferAttack`" method, but each one can perform different actions.

`EventEntity` derives from `ClockEventReceiver`, which implies that will react to the clock events. However, it does not implement the Tick function.

Table 5: EventEntity methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
UseUnits	`AttackInfo`	-	It spawns an `Attack` using the amount of units necessary to fulfill the requisites given by `AttackInfo`. If it hasn't got enough, it will use all the units that has.

Table 6: EventEntity input-related attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
SelectedUnits	`Int`	0-9999	Stores the number of units selected by the player. These units (and only these ones) will be used when an attack is fired.

Table 7: EventEntity input-related methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
DeselectUnits	-	-	Returns all the units that were selected to the entity
SelectUnits	-	-	Adds: - 1 unit to `SelectedUnits` if the entity has 5 or less in total. - 1/3 of the total units - 0 units if the Entity has none. - All if the entity is interacted while holding the left control button.
Useunits	`Gameobject`	-	Spawns an `Attack` that contains the `SelectedUnits` headed for the provided `Gamobject`.

Its attributes can be found in Table 4 and its method in Table 5.

This class is also responsible to handle user interaction. Given that all the elements in a level are susceptible of having to react to player input, that aspect is handled at this level to, again, simplify its implementation. The attributes and methods related with this aspect can be found in Table 6 and Table 7, respectively.

It is important to note that selected units will be discounted from the total units that entity has when they are selected. If an entity is attacked, this units will behave like if they were not on it. That means they won't be destroyed, but if the entity is lost, these units will disappear.

Planet

An extension of `EventEntity`. This one symbolizes a planet and is the standard entity that can be found in the game. It extends `EventEntity` so attacks, upgrades and unit

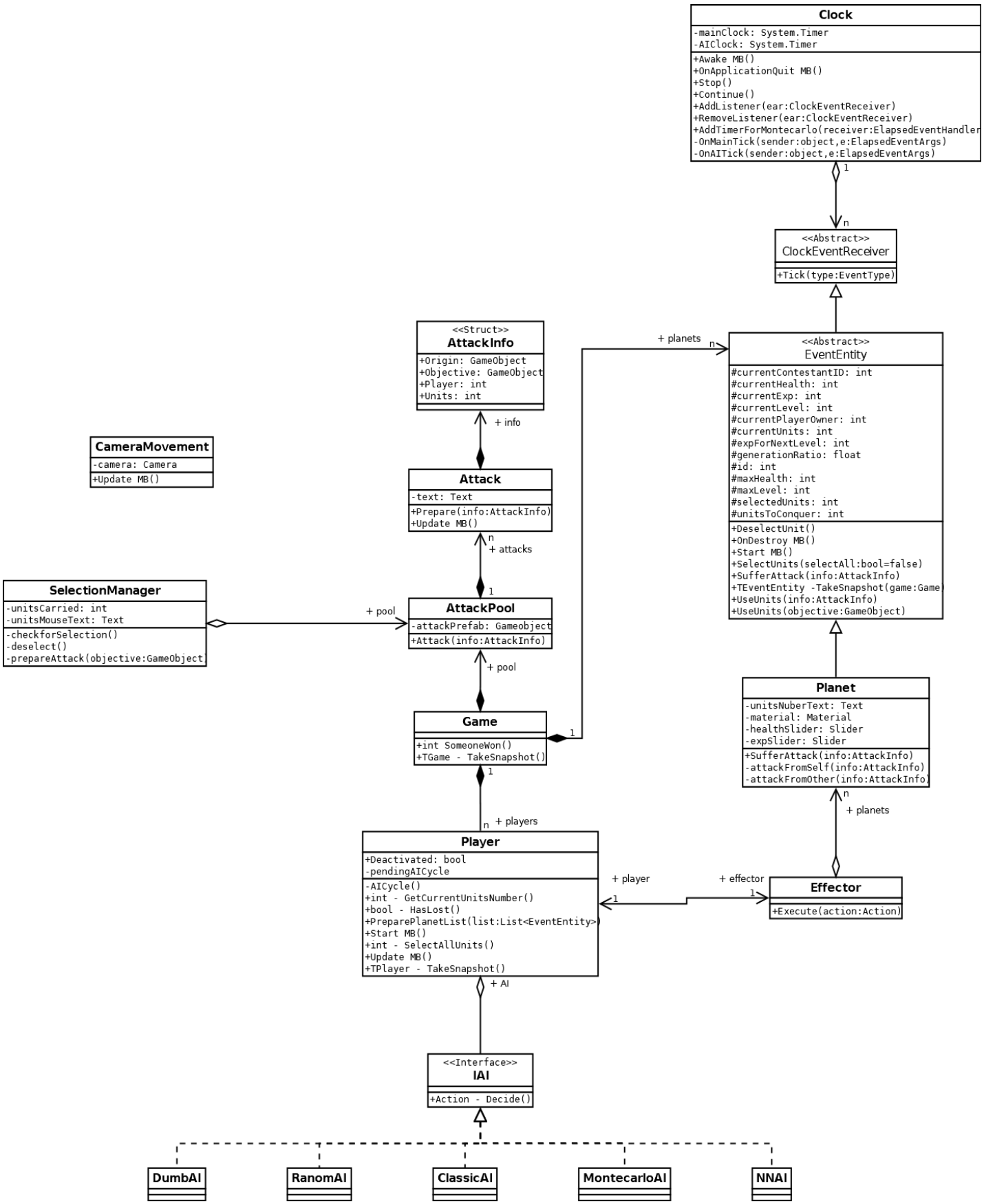


Figure 10: Diagram of classes (higher resolution version here)

Table 8: Planet attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
UnitsNumberText	`Text`	-	Hold a reference to the UI text element where `CurrentUnits` will be displayed.
Material	`Material`	-	Reference of the material of the object in the scene. It is used to change colors when the entity is conquered.
HealthSlider	`Slider`	-	Reference of the UI slider used for displaying health values.
ExpSlider	`Slider`	-	Reference of the UI slider used for displaying experience values.

Table 9: Planet methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
AttackFromSelf	`AttackInfo`	-	Handles attacks of the player that owns the player. - If the planet's health is not at maximum, it will be healed. - If the maximum level of the planet has not been reached and it has full health, the planet is upgraded (its experience points are increased). If it has enough experience points, the planet levels up. - If the planet has full health and can't level up more, the units are added to the units of the planet. More information on this subject in Section 2.2.
AttackFromOther	`AttackInfo`	-	Handles attacks of player that do not own the planet. - If the planet has some experience points but has not leveled up yet, these are reduced. - If the planet contains units and has no experience points, they will be destroyed. - if the planet doesn't have units, its health points are reduced. If these reach zero, the planet is conquered and the owner is the player that produced the attack.
SufferAttack	`AttackInfo`	-	Handles the provided attack. If the planet is neutral, the function will conquer it (if `AttackInfo` has enough units) or damage it. However, if the planet is not neutral: - If the attacker is the owner of the planet, `AttackFromSelf` is called. - If the attacker is not the owner, `AttackFromOther` is called.
Tick	`EventType`	-	Handles the clock events: - Main Tick: Adds 1 + current level to the total number of units. - AI Tick: Does nothing.

Table 10: AttackInfo attributes

TABLE 9: `AttackInfo` attributes			
NAME	TYPE	VALUE RANGE	DESCRIPTION
Origin	`GameObject`	-	The entity from which the attack was fired.
Objective	`GameObject`	-	The entity towards which the attack is headed.
Player	`Int`	0 – 5	ID of the player that ordered the attack.
Units	`Int`	0-999	Number of units contained in the Attack.

Table 11: Attack attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
CurrentAttackInfo	`AttackInfo`	-	The information of the attack that is being transported by the object.
text	`Text`	-	Reference to a text object used to display the units that are being transported.

generation can be processed in a certain way. This design will allow for different types of entities to be easily added to the game in a future. Its attributes and methods can be seen in Table 8 and Table 9 receptively.

AttackInfo

A `struct`[xv] that stores the basic information needed to define an `Attack`. It is used to transfer this information between objects. Its attributes are explained in Table 10.

Attack

This class controls the attacks in the game. It is responsible of its movement and behavior since its spawning until its arrival. It "transports" the units from their home entity to its objective.

It is important to notice that attacks move at a constant speed and do not depend on events. Once they are created, they will start to move until they reach their objective, moment in which the attack will be effective.

Attacks start at the same position as the entity which units are being used. They slowly move until the objective is reached, when the attack is made effective and the object

Table 12: Attack methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
Prepare	`AttackInfo`	-	Prepares the Attack to be spawned using the information that `AttackInfo` contains.
Update (MB)	-	-	Moves the object towards its destination and changes the size accordingly.

Table 13: AttackPool attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
AttackPrefab	`GameObject`	-	Model used to create the attacks that will be used by the pool.
AvailableAttacks	`List<Attack>`	-	List that stores all the attacks in the pool (those that are and that aren't being used at any given moment)
pendingAttacksInfo	`List<AttackInfo>`	-	When an object is instantiated in Unity, a complete cycle must have been passed before it is usable. This list stores the information of those attacks that have to wait a frame to be prepared because all the atacks in the pool were being used and a new one had to be instantiated.

disappears. Its size varies: it starts at size 0 and slowly increases until its completely outside the origin entity. This process is reverted when it is close enough to the objective. This is done to make the game easier to understand.

AttackPool

Generating and destroying `GameObjects`[xvi] in Unity is an expensive task in time and resources. Attacks are so common in our game that generating and destroying them would affect negatively the game's performance. That is why an object pool[xvii] is used.

At the very start of a level, 5 `Attacks` are created and deactivated, so they are invisible and have no effect in the game. When an entity needs to use one, instead of creating a new one, it simply uses one of these (one that is not already being used), changing its position and the information it contains. There is only one instance of this class per scene.

The `AttackPool` class administrates this collection of objects. When an `Attack` is needed, the entity that demands it provides this class with the corresponding `AttackInfo` and this class generates the corresponding `Attack`. Doing thing this way, entities don't need to know how the pool is implemented nor used, simplifying the task, thus benefiting of the principle of *cohesion*[xviii].

Table 14: AttackPool methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
Attack	`AttackInfo`	-	Selects an available `Attack` from the pool and prepares it. If not available `Attack` is found, instantiates a new one, stores the `AttackInfo` in `pendingAttacksInfo` and starts `PrepareNextFrame`.
PrepareNextFrame	-	-	Waits one frame and then iterates over `pendingAttacksInfo` and invokes the Attack function for each one of the elements in the list.

Table 15: SelectionManager attribute

NAME	TYPE	VALUE RANGE	DESCRIPTION
entitiesSelected	List<EventEntity>	-	Stores all the entities that have at least one unit selected by the player.
unitsCarried	Int	0 - 999	Stores the total amount of units selected by the player (counting selected units of all entities selected)
UnitsMouseText	Text	-	Reference to a text that will follow the mouse and will display `unitsCarried`.

Table 16: SelectionManager methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
checkForSelection	-	-	Implements the logic described in the Controls section for selecting units.
deselect	-	-	Deselects all selected units.
prepareAttack	GameObject	-	Prepares the necessary attacks to send all selected units to GameObject.

The pool size (that is, the number of attacks it contains) it's not constant. If all of its elements are being used and a new one is required, the pool size will be increased (new attacks will be created). After that the pool will keep its new size until the game ends or it is increased again.

SelectionManager

This class handles the interaction of the player with the game. It is in charge to select the desired units by the player, to use them or to deselect them again. Together with `CameraMovement` it covers the input needs of the game.

More information about attributes and methods in Table 14 and Table 15.

Table 17: Game attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
attackPool	AttackPool	-	Reference to the scene's `AttackPool`.
Planets	EventEntity[]	-	Array of all the `EventEntities` that exist in the level.
Players	Player[]	-	References to the players that compete in the level.

Table 18: Game methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
SomeoneWon	-	`Int`	Returns the ID of the Player that won or -1 if no one did.

CameraMovement

Moves the player's camera following the logic described in Controls section.

Game

Stores information about the game that is currently being played. Contains level (entities that can be found in the scene), players and attacks information. It is used mainly by players and entities to obtain information about the state of the game and other players and entities.

It is important to notice that this class can be accessed from anywhere in the scene, but not modified.

Player

Represents the entities that can play the game, whether they are controlled by a human or by an AI. They are used basically to keep record of the state of each player and what entities belong to him and thus which units can control.

It derives from `ClockEventReceiver`, because each Player controls its AI in terms of starting their process. This class is also responsible for executing the Actions that the AI orders. For more information, see Table 19 and Table 14.

Table 19: Player attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
AI	`AI`	-	Stores the AI of the player, independently of its type.
CurrentGame	`Game`	-	Reference to the game that is being played.
Deactivated	`Boolean`	True – False	When a player is eliminated, it still receives clock events and, thus, its AI is triggered. This `boolean` is set to true when the last `EventEntity` of the player is conquered, so it won't use its AI any more.
myEffector	`Effector`	-	Stores a reference to the player's `Effector`. More information in its current section.
pendingAICycle	`Boolean`	True – False	The AI executes in a separate thread to avoid affecting the game's performance. To do so, coroutines[xix] are used. However, because the AI is triggered through a separate thread (the thread that spreads the clock events), a coroutine can't be started (because of the way Unity is implemented). This value is used to start those coroutines the next frame after clock event are recevied.
typeAI	`AIType`	None - Dumb – Random – Classic – Montecarlo - NeuralNetwork	Stores the type of AI used by the player (if any).

Table 20: Player methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
AICycle (CR)	-	-	Coroutine that executes the AI of the player, waits one frame and then uses myEffector to execute its orders.
GetCurrentUnitsNumber	-	`Int`	Returns the total number of units the player has at any given moment but does not select them.
HasLost	-	`Boolean`	Returns true if the player does not has any entities under its domain.
PrepareList	`List<EventEntity>`	-	Used at the start of the game to set the initial entities of each player.
Start (MB)	-	-	Adds the player to the clock's listners, instantiates the corresponding AI and an instance of Effector.
SelectAllUnits	-	`Int`	Selects and returns the total number of units of the player (that is, every unit from every planet the player controls).

Tick	EventType	-	When an AITick is prouced, starts the process for executing its AI.
Update (MB)	-	-	If pendingAICycle is true, the coroutine AICycle is started.

Table 21: AI method

NAME	ARGUMENTS	RETURN	DESCRIPTION
Decide	-	`Actions`	Returns an action that has to be executed.

Table 22: Actions values

Action	Description
NONE	Represents the absence of an action.
WAIT	The player does nothing.
ATTACK ENEMY	The player will attack entities that have been conquered by another player.
ATTACK NEUTRAL	The player will attack entities that have not been conquered yet.
UPGRADE	The player will spend units to level up its own entities.
HEAL	The player will spend units to restore the healh points of its own entities.

AI

Interface[xx] used to simplify the interaction with the different types of AIs. It has only one method (Table 21). Again, this is used to avoid dealing with the particularities of each type of AI. The player only needs to invoke Decide and execute the Action it returns.

Actions

Enumerator that defines the actions that can perform non-human player (Table 22).

There are some classes (`Effector`, AI related…) that have not been mentioned. Those will be explained in greater detail in their corresponding section, because we feel that treating them like a normal class wouldn't be enough to cover them correctly.

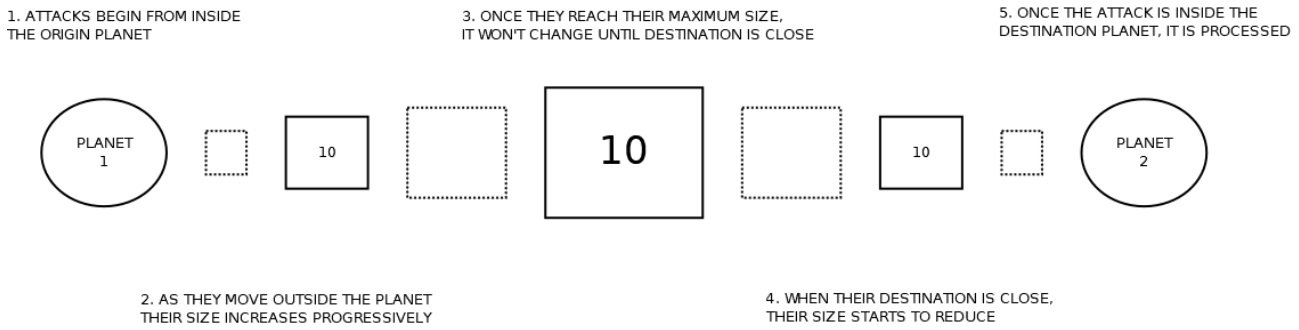


Figure 11: Attack lifecycle

3.2 Game Flow

The game flow and the peace at which game events happen depends on two factors: the clock and the human player. The first one is in charge of all the human-player independent events and the second one controls one of the participants. That has an important implication: the player controlled by a human can execute some actions in an unpredictable manner as long as it has enough units to do so, which implies that those actions can occur at any given time and are unpredictable.

In opposition, the clock has a cyclic and predictable behavior:

1. A tick is triggered
   a) It is a normal tick
      a.1. Non-neutral entities unit number its increased
   b) It is an AI tick
      b.1. For each player in the game, except the one controlled by a human, the AI is invoked
      b.2. When the AI has decided its next move, it is executed.

Attacks that have been issued but have not yet reached their destiny execute their own behavior:

1. One player attacks another
   - The attack is created and moved to the center of the planet (or planets) where the used units were stored.
2. The attack moves at a constant speed towards its objective.
3. The attack reaches its objective
   - Th objective processes the attack depending on who fired it, the amount of units it contains, its own state (the objective state), etc.

Notice that attacks follow unity's execution flow in order to move and update accordingly the objects that represent them in the scene. This will happen until someone wins the game or, in attack's case, until there are no pending attacks moving through the level.

Table 23: Effector attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
map	GameObject	-	Reference to the parent object in the scene that contains all the entities of the level.
myPlayer	Player	-	Reference of the player that owns the instance of the class.

Table 24: Effector methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
Execute	Action	-	Executes the action if possible using the rules above.

3.3 Action Execution

Execution of non-human player actions is done by the `Effector` class. It was created to simplify not only the process of action execution, as said before, but also to do it with the decision making as well. The first part is simple and we have already seen it before: it simplifies the execution of the actions by the principle of *cohesion*[xviii].

By having this class, the players do not need to know anything about how to execute actions or what criteria to follow depending on its state, etc. It only needs to trigger the AI, store the action it chooses as the best and let `Effector` do the job.

But the most important simplification is the one related to decision making. By defining a constant set of rules about how every action has to be done, we can simplify the possible number of actions. In its original state, a player could do nothing or could attack every entity there was with any number of units between 0 and the amount of units the entity that was "using" had. And that is if only one of its own entities were used, because several can be used at the same time.

As the reader can see, that represents a huge number of possible actions. And trying to choose only on using any type of AI would be highly difficult. By having the `Effector`, players now can only perform 5 actions: wait, attack entities conquered by other players, attack neutral entities, upgrade its own entities and heal them.

The `Effector` is in charge of deciding the best way of performing those actions and executing them using the player's resources. Of course, this is done introducing random factors that adapt to the difficulty to perform these actions not as optimal as they could be done.

Here we explain how the `Effector` executes the actions. Again, keep in mind this is the best performance scenario. By introducing variations in the number of units that will be used, the difficulty can be adjusted so the player is presented with a challenge and not overwhelmed with a merciless rival.

- Wait: It simply does nothing. But in a very effective way.
- Attack enemies / Attack neutral entities: Both actions are executed similarly.
    1. The number of units necessary to conquer the objective is counted. That includes (in the case of non-neutral entities) counting the health points, the experience points and the units that the entity possesses. The `Effector` will try to send 1.5 times this numbers, to ensure success.
    2. A list of the player's entities sorted by distance to the objective is obtained.
        - Units of the players entities will be used starting by the closest one to the objective, as it can be seen in Figure 1.
- Upgrade / heal: The process is similar to the followed in the attack actions, however, only the necessary amount of units is sent. Again, the closest units are used first.

This approach has also some limitations. Only one entity can be attacked per execution. This slows the gameplay and makes the game easier, because the player has time to react to the attacks and the AI has to be executed again implying that several seconds have to pass for another attack to be produced.

To be able to function, the class needs two attributes, found in Table 23 and a method, shown in Table 24.

The `Effector` also has security checks. It handles the special cases like, for example, when it receives an Upgrade action and all the player's entities are already at their maximum level. In those cases, the `Effector` behaves the same way that does when it receives a Wait action type.

3.4 Training Version

Until now, we described the "normal" version of the game and how it has been implemented. This version is the one users can play and interact with. It has also a graphical component that represents the state of the game at any given moment in a screen, in an easy and understandable way.

However, this has some limitations. First of all, the execution of the actual "game" has to wait and take care of the graphical part. Attacks and entities have to be updated and that has to be done at a slow enough speed for the human player to be able to interact and react to the events. The second limitation is that the game must follow the Unity execution flow, being very careful to not block it. If it does, the game freezes, temporarily or forcing he user to close the application.

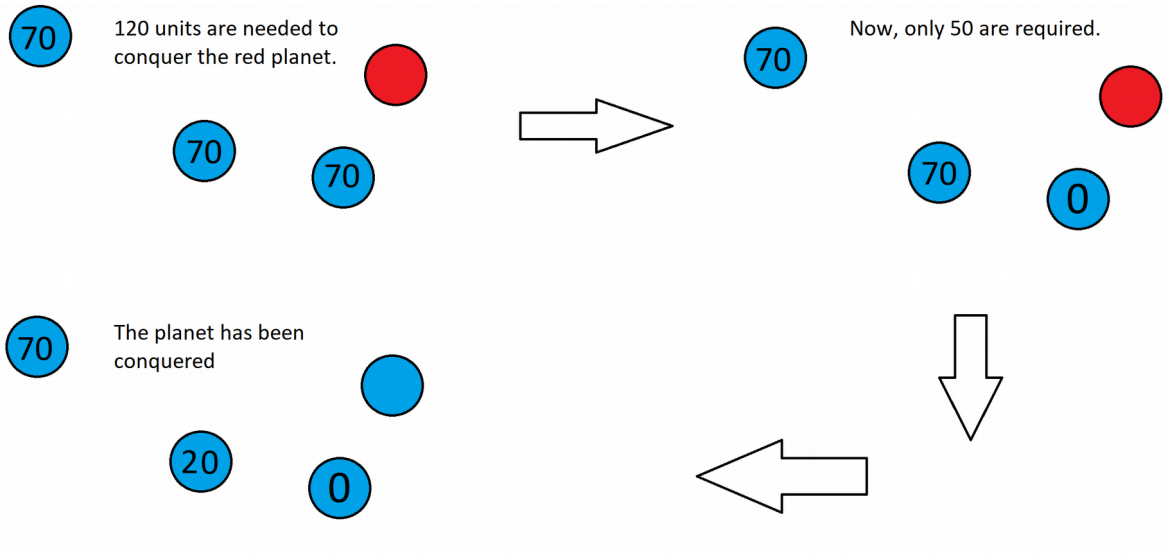
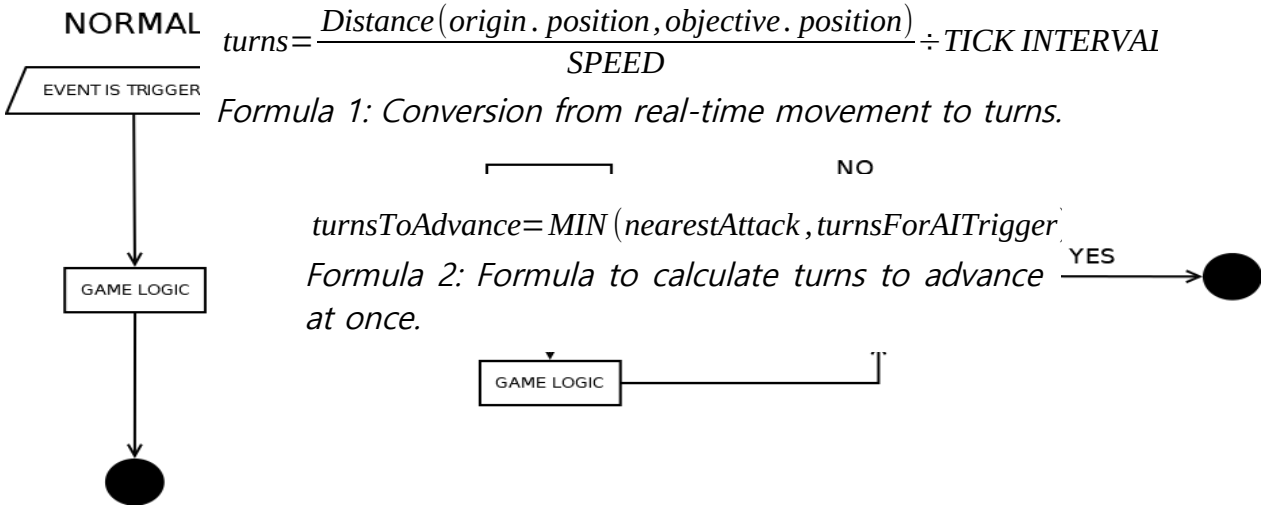


Figure 12: Process of unit sending. Units that are in closest planets are sent first.

There are some tasks that cannot be performed under those circumstances. Either because they need to be done as fast as possible and have to be repeated huge amounts of times or because need to use all the resources possible. Using the "normal" version of the game is impossible to perform those tasks.

Is at this point where developing a "training" version of the game becomes a necessity. By "training version" we refer to a variation in which no graphical nor interactive aspects of the original game remain. It is simply the core logic of the game executed as fast as possible and communicating with the exterior through console output. In this section how this variant has been implemented and the most fundamentals aspects of it will be discussed.

The first thing the reader needs to know is that this version will be an exact copy of the games logic. The rules and outcomes will be the very same that those of the normal version. This version modifies *how* the game is executed, not *how* is it played. And by *how is it played* we refer to how the performance of determined actions changes the game's state, we are not talking about input.



$$turns = \frac{Distance(origin.position, objective.position)}{SPEED} \div TICK\ INTERVAL$$

Formula 1: Conversion from real-time movement to turns.

$$turnsToAdvance = MIN(nearestAttack, turnsForAITrigger)$$

Formula 2: Formula to calculate turns to advance at once.

Figure 13: Normal flow VS. training flow.

But in regards to that, the input is eliminated from this version. No human will be able to interact (and of course, to play) this variation. We want the game to execute as fast as a computer is able to do it and adding a human factor will make this impossible. Humans are infinitely slower than any machine and thus, need to be ignored to achieve fast execution.

New flow

In the normal version, the gameplay depends on the clock. The AI and the units are produced as a response of the events triggered by timers. The human player, of course, can perform actions between events, but still needs units to do so, and units are created when these events are triggered. So it is pretty fair to say, the flow of the game is completely bound to the clock.

Because we want to execute the game as quick as we can, this bound has to disappear. Attacks behavior needs to be revised too. In the normal version of the game, they moved from the entity which created them to their objective, but now there are no objects in the scene and thus, nothing to be moved.

The solution we adopted was to use turns instead of time or movement. A turn is equivalent to a MainTick event. There is nothing between turns and they happen constantly, one after the other, as seen in Figure 13.

Now only the attack aspect remains. Our solution was to count how many turns would it take for an attack to reach its objective by dividing the distance by the speed, and then by the time between MainTicks (Formula 1). The first operation returns how many seconds the travel would take and the second translates that into turns. It is not perfect, but because all attacks will follow the same rules, the precision loss is compensated.Every attack will now store this information and update it accordingly as the turns pass. When all the necessary turns have passed, the attack becomes effective.

With this new system, a minor change can be performed to optimize the execution. Instead of advancing turns at a time, which would produce the execution of operations that don't have any important effect on the game whatsoever, we can advance several at a time.

The idea behind this is that a normal turn only increases the number of units in the entities of the level. But as long as an attack reaches its destination or the AI is triggered, nothing happens. In the normal version we had a human player, that could perform actions whenever he wished, but now that element does not exist.

So, to avoid performing operations each time a turn is executed, we execute several turns at once. This implies that instead of having to check and add one to every entities units counter, we can add the corresponding number at once.

This has to respect a simple condition: the number of turns that will be advanced will correspond to the minimum between the remaining turns for the next attack to arrive at its destination and the remaining turns for the next AI tick (Formula 2).

Simplified classes

Until now, the main classes and mechanisms of the game have been explained to the reader. However, all of this classes were designed to create a playable game. This can be seen clearly taking into account how the flow is dependent on a clock. By doing things this way, the game can be tied to real time and allow humans to interact.

But in this version, as it has already been said, we need to free the game flow from human needs. That is why, a lot of classes need to be redesigned. Redesigned, but not rebuilt, because the logic is still the same. So the first pass we are going to take is to modify the main aspects of the game and create new modified classes, designed to execute as fast as possible. We will also suppress any graphical elements from them, because graphics would slow down the game.

All the attributes related to graphical elements and with the Clock class have been suppressed. This has also happened with the mono behavior methods. By doing so, specially the last part, we obtain independence from *Unity's* execution cycle, which allows the game to execute as fast as possible.

Because these classes do the same as their user-friendly sisters, they will be presented by alphabetical order. Their functions are the same that their original counterparts, so if the corresponding section Section 3.1 was read, the reader should understand how they relate with each other and form the game. They have the same name, but whit a capital "T" at the beginning.

Table 25: TattackInfo attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
Destiny	Int	0 – number of entities in the scene	Stores the index of the entity where the attack is headed. With it, the Entities array in the class TGame can be accessed and thus, perform it.
Player	Int	0 – 5	Index of the player in the Players array of the class TGame. Used to identify who launched the attack.
remainingTurns	Int	0 – 99	The amount of turns, obtained the way explained before, that remain until the attack becomes effective. Each turn is reduced by 1.
Units	Int	0 - 999	Amount of units contained within.

Table 26: Teffector attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
player	Tplayer	-	Reference to the player that owns the instance.
planets	TEventEntity[]	-	Reference to all the entities in the level.

Table 27: TEventEntity attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
game	TGame	-	A reference to the game being played. Its function is the same, but the type has been updated.
Position	Vector3	(-1000, -1000, -1000) – (10000, 10000, 10000)	Because there isn't an object in the scene to which position attribute can be accessed, it needs to be stored when the TEventEntities are stored

Table 28: TGame attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
PendingAttacks	List<TAttackInfo>	-	Stores the attacks that have been launched but have not reached their destiny.
Planets	TEventEntity[]	-	Stores the entities of the game.
Players	Tplayer[]	-	Stores the players of the game.
WeHaveAWinner	Boolean	True - False	Set to true when a player wins the game, false otherwise.
Winner	Int	0 – 5	ID of the player that won.

Table 29: TGame methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
AITick	-	-	Triggers an AITick, making all players decide their next action.
GetUnitGenerationRatio	int	Float	Returns the amount of units generated per turn by the player whose id is passed as the argument.
Initialize	Tplayer[], TeventEntity[], List<TAttackInfo>	-	Initializes the game at a given state.
CheckPendingAttacks	Int	Int	When a turn is advanced, all the attacks that have been launched but have no reached its destiny have to be updated (their remaining turns have to be decreased). This function receives the number of turns the game has advanced and updates the attacks accordingly. If an attack, after being updated, has zero or less remaining turns, it is executed and removed from the list. It also returns the maximum number of turns that can be advanced; that is, the turns until the next attack arrives at its destiny.
CreateUnits	Int	-	Advances the provided number of turns for all the entities. The corresponding amount of unities per turn are created in every entity multiplied by the number of turns that have passed.

TattackInfo

Modifies `Attack` and `AttackInfo`.

This enumerator[xxi] replaces both `Attack` and `AttackInfo`. It stores the information needed to track an attack since the moment it is created until it reaches its destiny. Because it will not be an object in the scene any more, only this info is required to keep track of each attack's state. Its attributes can be found in Table 25.

It does not require any methods, because it simply holds the information that represents an attack, but it is processed externally by other classes.

AI classes (dumb, random, montecarlo and NN)

Minor updates to accept the new training types (TEventEntity and Tplayer).

Table 30: TPlayer attributes.

NAME	TYPE	VALUE RANGE	DESCRIPTION
Planets	List<TEventEntity>	-	List of the entities under control of the player.
effector	Teffector	-	Instance of the TEffector that will execute the actions commanded by the AI.

Teffector

Modifies `Effector`.

Again, same as the original, but the attributes suffered minor changes (Table 26).

TEventEntity

Modifies `EventEntity`.

Again, little has changed when adapting this class. Its attributes can be found in 27.

TGame

Modifies `Game`.
This class has suffered some major changes to adapt it to the new flow of execution. It now handles the events and passes them to the corresponding entities, because there is no clock that does that in this version. Its attributes and methods can be found at tables 28 and 29, respectively.

Tplanet

Modifies Planet.
The modifications made in this class only suppress the graphical elements. The logic is still the same and no new elements have been added.

Tplayer

Modifies Player.
Again, this class only modifies the types of some of the variables. The logic is untouched (30). The classes diagram of this version of the game can be found in Figure 14.

3.5 Snapshots

Some classes have a method that hasn't been explained until now: the `TakeSnapshot` method. This method is very simple and returns an instance of the simplified class corresponding to the type of the object that invoked this function.

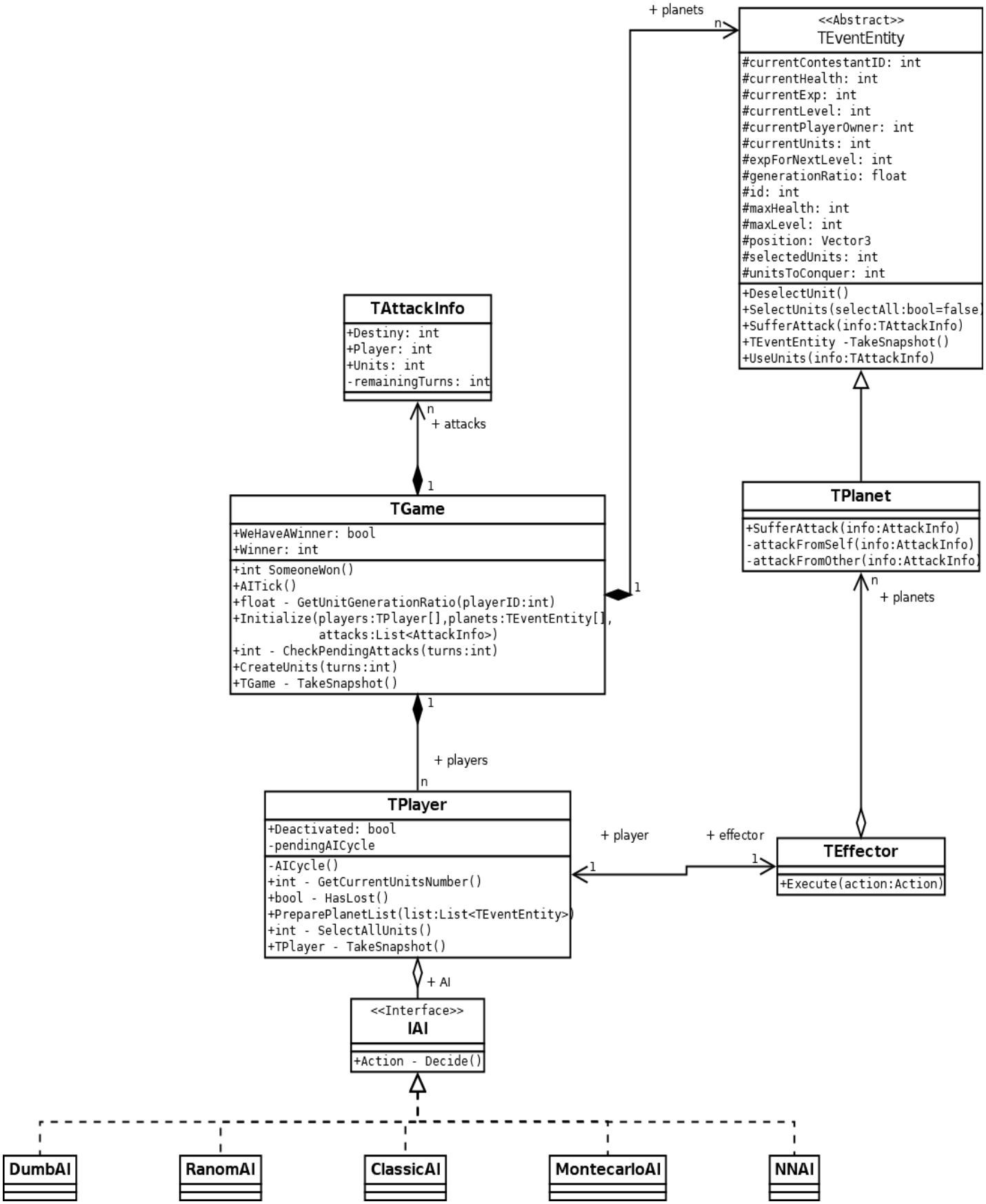


Figure 14: Diagram of training classes

3.6 Training Game Execution Cycle

The training has been designed with the idea of playing more than one game consecutively. A diagram of the full cycle can be found in Figure 15. Here its explained in detail:

1. The information from the scene is loaded. The distribution of entities, and players is read. A new instance of the class `TGame` is created with this information.
2. The training instances of objects that represent the entities are created and initialized following the information that was found in the scene. That includes:
   2.1. Creating training instances for every entity, specifying its position, current level, maximum level, etc.
   2.2. Initializing the entities by providing them with the player that owns them (or none if that's the case).
3. Generate the players:
   3.1. Instances of `TPlayer` are created and its AI its initialized.
   3.2. The planets that belong to each player since the beginning of the game are added to each player's Planets list.
4. A snapshot of the current state of the game is taken. This will be used to restart the game when needed.

At this point we have all that we need to start playing, the entities are prepared and the players have an AI and some initial planets. The game can start.

5. If the required amount of games have been played the execution ends.
6. If not, the game is restarted using snapshots. That means that the snapshot taken at step 4 is loaded, to ensure a fresh start.
7. Main loop of the game:
   7.1. The decided amount of turns (or 1 by default) is advanced. This implies that, the operations that will be performed in this iteration of the main loop will do so for this amount of turns at once.
   7.2. The attacks are checked. Their remaining turns are updated, that is, the amount of turns advanced in step one is subtracted for each attack `remainingTurns` attribute. If one or more has reached its destination (`remainingTurns` = 0), the attack is executed.

This step also provides the amount of turns that can be advanced until another attack reaches its destination. This number will be used as a candidate for the next execution of step 7.1.

7.3. If someone has won, the game ends and the execution returns to step 5.

7.4. If nobody won, units are created. The number of units that will be created in each entity corresponds to the units that will be created in one turn times the number of turns that have been advanced.

7.5. It is checked if the AI should be triggered.

A) If it has to be, we continue to step 7.6.

B) If not, the remaining turns for it to be triggered is compared to the number of turns obtained in step 7.2.

(a) If the number of turns remaining until the AI is triggered is less than the turns obtained in step 7.2, the game will advance the number of turns remaining for the AI to be executed.

(b) If not, the number of turns obtained in 7.2 will be advanced.

After that, the flow returns to step 7.1.

7.6. Number of turns until next AI execution is reset.

7.7. For every player, its AI its triggered and the action it decides to execute is passed to the player's `Teffector` for execution.

7.8. After all actions have been executed, the flow returns to step 7.1.

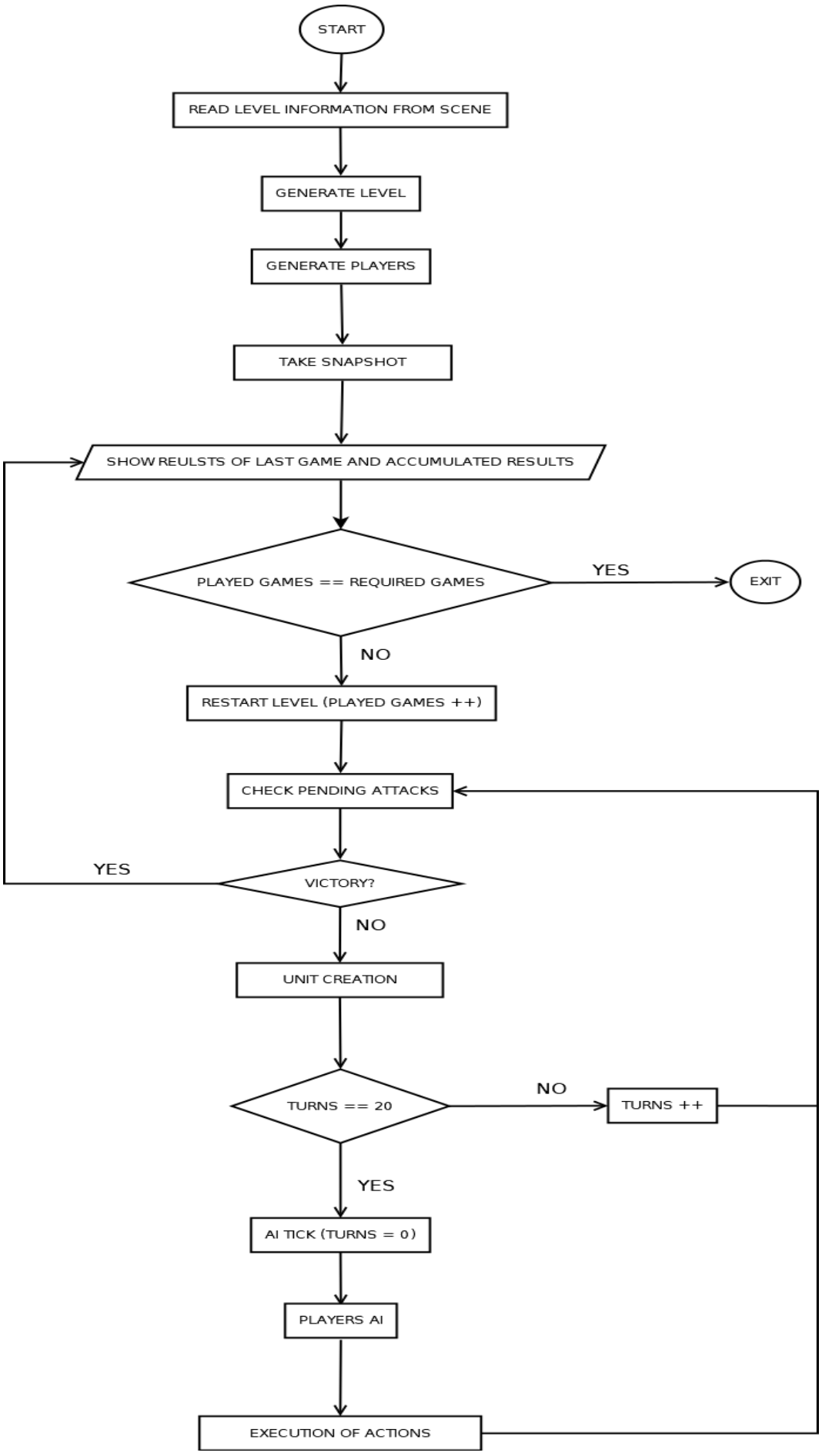


Figure 15: Training Execution cycle

Simulations

In the following sections, the term "simulation" will be used various times. We would like to clarify here what are we talking about when we use that term. A simulation is a complete execution of the training game execution cycle where the total numbers to play is 1 that can be started using a provided state of the game.

In other words, is the execution of a full game in training mode until it is finished that starts either from the state read from the information found in the scene or another state that can be provided to the algorithm as the starting point.

3.7 AI

In this section the basic AI mechanisms will be explained in detail. Both Montecarlo Tree Search and Deep Learning will be treated in their own section. This techniques are more simple approaches (trivial in some cases) that allow a game to take place, with or without human player. These AI techniques have been implemented mainly to be compared with the more complex ones, main subject of this project.

Again, we want to remember that all of these techniques implement the interface AI. They all implement a different version of the Decide method, where the work is done. After finishing, an action (passed as a variable of type Actions) is returned, and the AI work is done.

There will be explained three AI types in this section: *dumb*, *random* and *classic*.

We start with the *Dumb* AI. The most simple and easy to defeat. It simply waits, forever. When the Decide function is called, it immediately returns the action wait, and does nothing else. It is useful for checking the main strategies of other types of AI.

The *Random* AI is a little more complex. Instead of returning always the same action, it chooses one arbitrarily. Very useful for checking the reaction of other types of AI to its behavior.

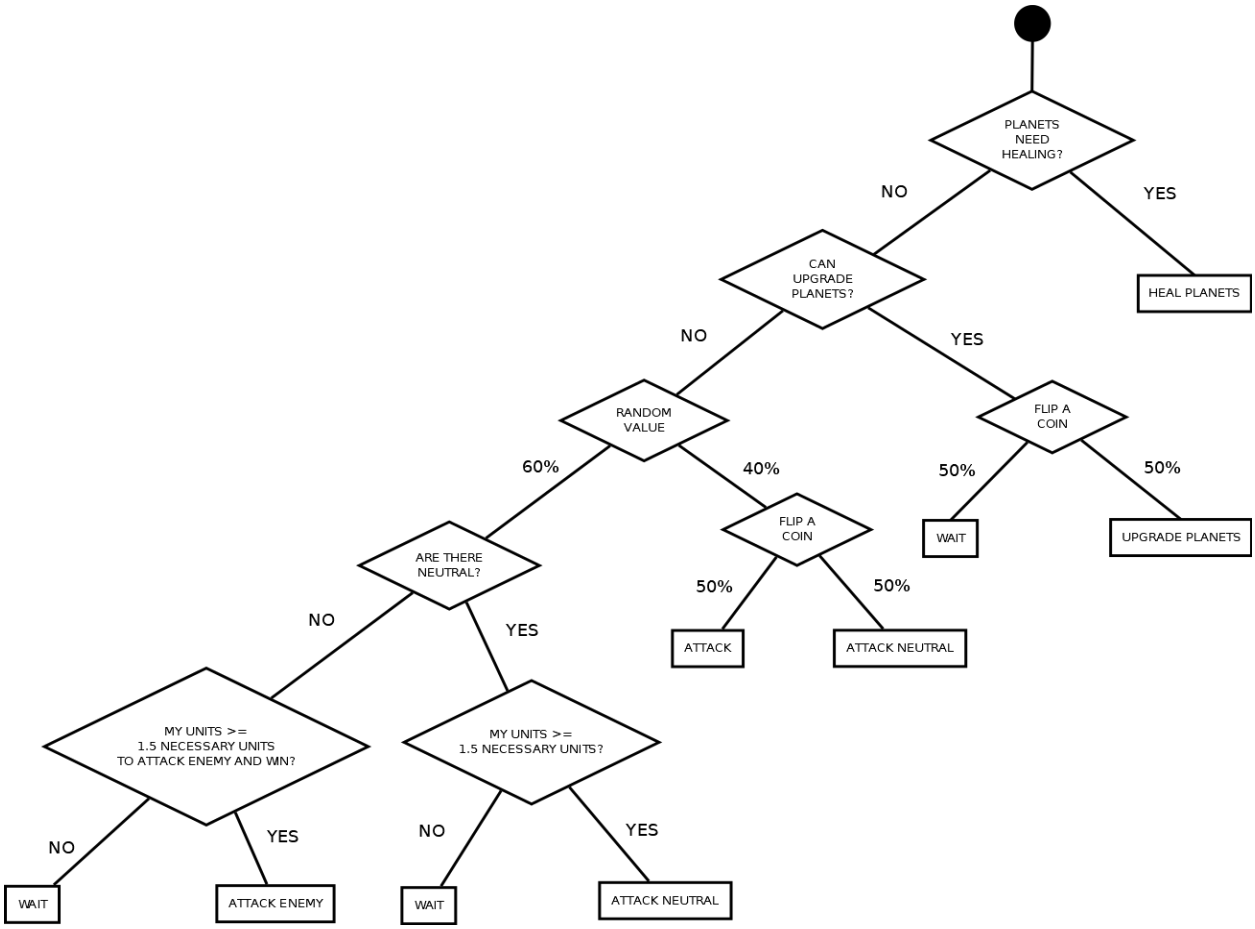


Figure 16: Classic AI decision tree.

The classic AI is the most complex of these three, although it is rather simple. It uses a behavior tree to decide which action is the most appropriate given the context of the game that is being played. To avoid making an AI too predictable, random elements have been introduced in the decision-making process, so it might perform strange and even illogical actions at any moment. Its decision tree can be found in Figure 16. The random values that can be appreciated are included to create a behavior more human-like. This will enhance the game play quality, by having an AI more enjoyable to fight and it will also will provide a rival less predictable for the machine learning techniques to practice. This is because the *classic* AI will be used to train and test the machine learning techniques that will be implemented.

Note that the classic AI does not use a tree in its implementation. The behaviour tree in Figure 16 is only a representation of a series of if-else structures used to take decisions.

The different types of AI will be implemented in separated classes that implement the AI interface (see Section 3.1). They will all be used the same way, except the montecarlo class (more information in Section 3.8); they will call the Decide method (Table 21) and return an action.

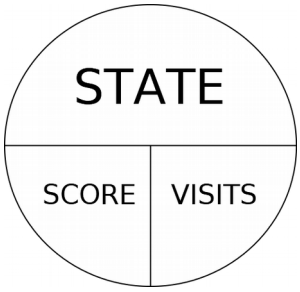


Figure 17: Diagram of a node

$$\frac{score}{visits} + \phi * \sqrt{\frac{\ln(parent\ visits)}{visits}}$$

Formula 3: Montecarlo best child formula

3.8 Montecarlo Tree Search

Montecarlo tree search (from now on, M.T.S.[ii]) is an AI technique that explores the space state of the game and chooses, based on the actual state of the game, the best possible action that can be done. The "best action" is the action with higher possibilities of being the first of a chain that ends with the player executing this AI type victory.

To do this, the states are represented as the nodes of a tree and the actions as the links that connect them. The algorithm then searches in this tree a chain of states that ends in victory and executes the next action in that chain to try to get to that victory state. An example can be seen in Figure 18 (tree) and Figure 17 (node). For this algorithm the training version of the game is used.

In our case, every node has 5 children (one per action) and the "distance" between states is the turns between AI ticks. Each connection represents an action, and its order is always the same.

However, because the space state of the game is so immensely huge, it can't be explored (or even stored) at a high enough speed for the algorithm to execute in a time period acceptable to be responsive in a videogame. Because of this, the algorithm uses heuristic techniques and rollouts to estimate the action with higher possibilities to lead to a victory.

This section provides only a basic introduction to the algorithm and its functioning. For more detailed information and implementation details, see Annex I.

Heuristics

A heuristic is a technique used to optimize algorithms by achieving good results immediately but without guaranteeing perfect or optimal ones. In other words, they are methods that "guide" algorithms to take decisions that *seem* to be the correct ones because they provide good results in the short term or because the fulfil other criteria but, in the long term, can lead to solutions that are not optimal (that is, there

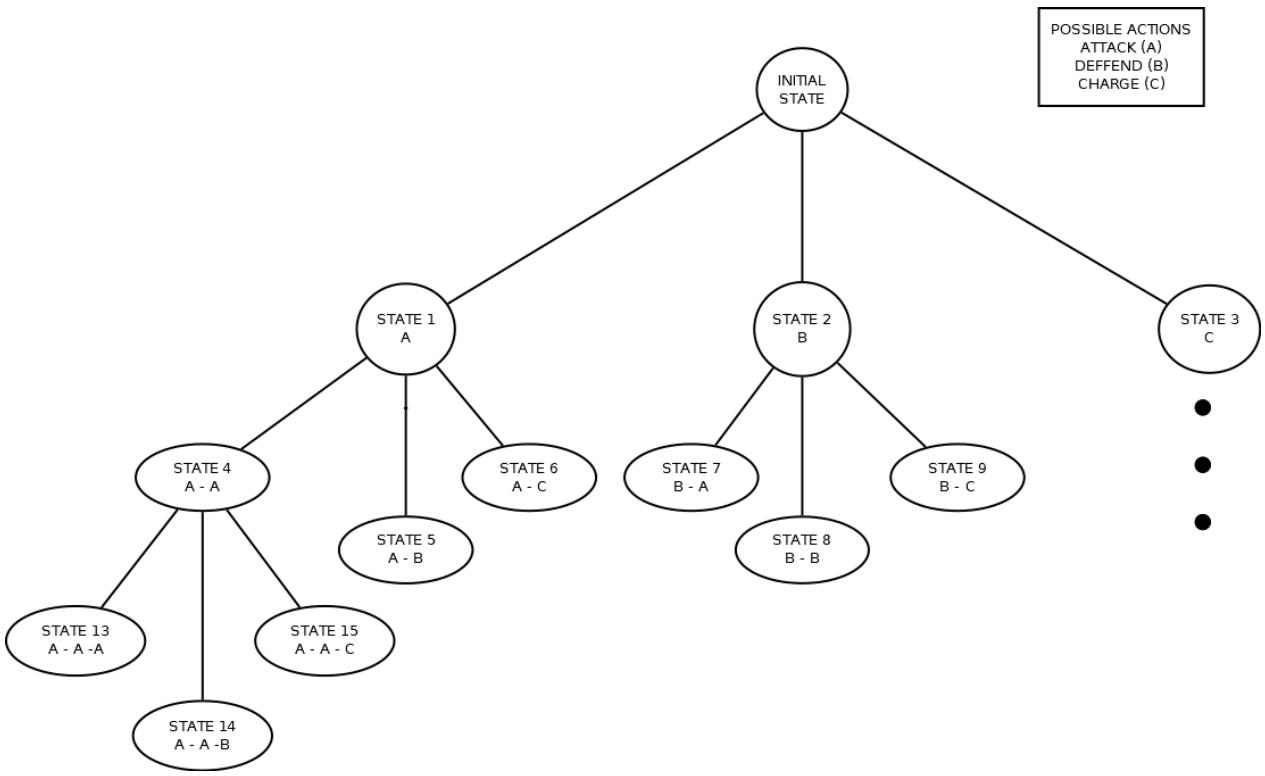


Figure 18: Example of a Montecarlo Tree structure

are better solutions than the one obtained). In exchange, heuristics accelerate algorithm and usually obtain answers quickly.

In M.T.S. heuristics are used to determine which states are explored before others. Based on the score and the number of visits, a value is computed for each child node of the node that is currently being explored (if it has children). The node with the higher score will be explored, and so on and so forth. Formula 3 is the formula being used for this purpose.

Again, for more information see Annex I – Heursitics.

Rollouts

A rollout is a simulation of the game. It might start from the very beginning or from any given state stored in a node and continues until the game ends.. The simulation is conducted by choosing random actions for each player and executing them.

Rollouts are executed using the training version of the game, as explained in Training Game Execution Cycle section.

Back-propagation

Once a rollout has finished, a score is obtained based on its results (for example, if the player using montecarlo wins a +10 score is obtained and if it loses, a -10 one is obtained). This score, however, needs to be extended through the tree to have an

influence in the exploration of new nodes. In other words, to decide whether to explore or not that node again, we need that score to influence the nodes that is necessary traverse to reach the simulated node again.

To do this, the score is backpropagated. Starting from the leaf node where the rollout started, its lower-level nodes (parents, grandparents, etc.) are traversed again until the root node is reached. While traversing these nodes, the obtained score is added to them, modifying he score they previously had and increasing their visit number.

Expansion

An expansion is when the children of a node are created. That is, the state of a node is taken as the origin point and simulated for the amount of turns between AI ticks. In this "mini-simulation", each player takes a random action except the one controlled by M.T.S.; which will execute the action corresponding to the children that is being created.

Notice that the state of the original node does not change at all. After each simulation it is copied to the new node and reset.

Algorithm

When the algorithm starts, the tree has only a root node with the current state of the game. The algorithm the expands this node by simulating every possible action and creates the first child nodes. After that, the exploration begins. For each node, besides the state, the number of visits and a score is stored (Figure 17). The algorithm then uses heuristics to choose a node to explore. When the node has been chosen, the algorithm can face three scenarios:

1. The node has children. In this case, another node is chosen from those children using heuristics again and explored.
2. The node is a leaf node (has no children) and has not been explored yet. In this case, a rollout is performed. After the rollout, the obtained score is back-propagated and the process starts again from the root node.
3. The node is a leaf node and has been explored. In this case, the node is expanded and a child is chosen using step 1.

An iteration is a full search - rollout - back-propagation cycle.

Because not all the state space can be explored, this algorithm is executed until a fixed number of iterations or a certain time has passed. Once this limit has been reached,

the execution stops and the best action is chosen comparing the score of the root node children, this time without using heuristics.

Results

The implementation of this technique has been successful. The main algorithm has been implemented and functions as expected. Even more, implementation decisions such as using an array for the information storage and thread usage resulted in a very good performance, leading to executions with 120.000 iterations in 2 seconds. However, the learning part of the algorithm has been severely affected by the game nature.

Because of the real-time gameplay in the normal version of the game, the human player actions can't be predicted in any way and can't be adapted to the tree structure. Human player can perform no actions, one action or several actions between AI ticks whether the AI can only perform one and thus, the tree with the fraction of the space state already explored is not valid between AI ticks.

This implies that instead of exploring a fraction of the space state, returning an action and waiting until the next turn to continue exploring where the last execution was stopped, the tree has to be discarded in its entirety and thus; a new one has to be created from scratch. Staring from the state of the game when the AI tick was triggered, of course.

However, due to good performance, the algorithm is still capable of offering a challenge to the player despite only having two seconds to learn from scratch.

The duration of this technique's implementation was more than expected. As a result of this, the second part of the project related to this technique, the difficulty adaptation, was suppressed to be able to achieve acceptable results in the Deep Reinforcement Learning part. This is also the reason behind not revising this technique for a better adaptation to the real-time nature of the game and leaving it as it is.

3.9 DEEP REINFORCEMENT LEARNING

Deep reinforcement learning is the newest and last technique that will be implemented in the project. It belongs to the machine learning domain, a research field that has revolutionized certain aspects of our life and promises to keep doing so in the following years.

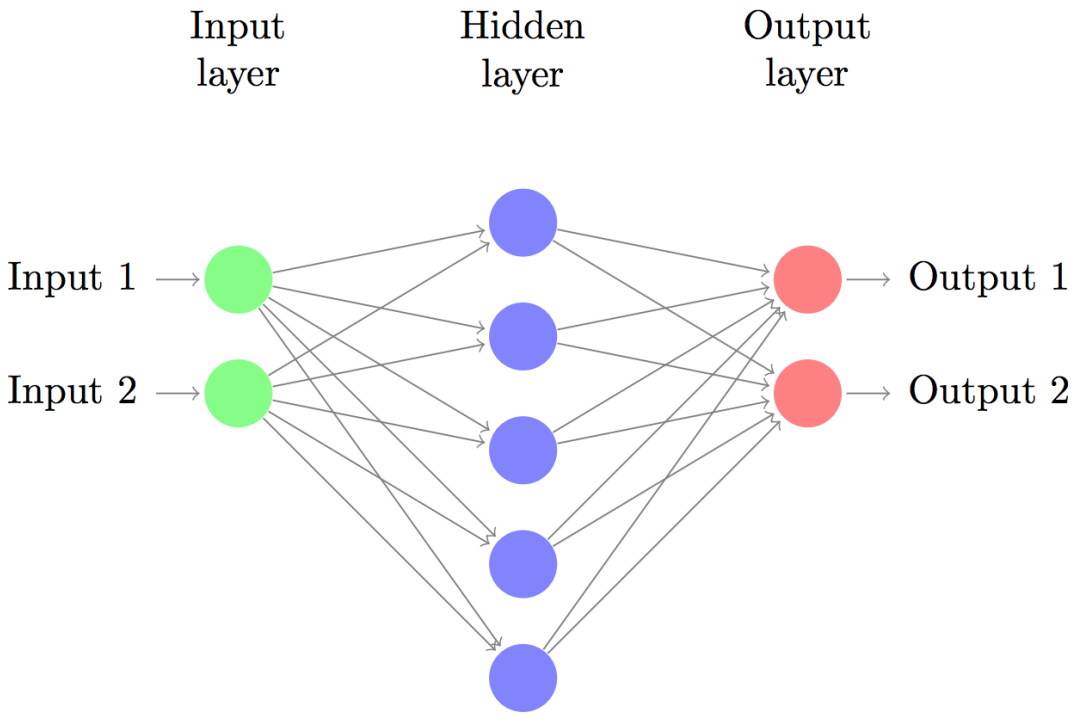


Figure 19: Example of a deep neural network

Again, this section will describe the technique and its implementation briefly. For a more detailed version, see Annex II.

Machine Learning

Machine learning refers to a group of techniques based on statistics that try to improve the performance of machines doing a certain task without explicitly programming them to do so. In other words, these techniques try to make computers and other devices *learn* how to do a task the most efficient and effective way possible.

The process starts with a computer or device trying to perform a task executing random actions and, obviously, failing miserably. However, with enough trial and error, the machine ends up performing the task remarkably well. In an ideal scenario, of course.

Models are used to achieve this. Models are mathematical representation of systems. In machine learning case, they represent the decision process the machine has to perform given certain output. When input values are provided to a model, it outputs another series of values that, if the model has been trained, represent the action that should take place next to perform whichever task the device is performing.

Machine learning techniques start with random models that are progressively modified. At the start, the output they produced is poor in quality (in terms of how the task is

performed) and slowly becomes better. This process called training, and it is necessary in all machine-learning techniques.

The devices, characters, etc. that perform the actions are called agents and the space (real or not) in with which they interact and the actions are performed is called environment.

Deep Reinforcement Learning

This group of techniques combine two of the most popular machine learning approaches: Deep Learning and Reinforcement Learning. The first one focuses in data representations. In other words, it studies ways of creatingand storing models in ways that allow for better learning. In this particular case, Deep Neural Networks are used.

A Deep Neural Network is a directed graph that can be traversed from one side (input side) to the other (output side). The graph is composed by layers of nodes. There are

three main types: input layer, hidden layer and output layer. An example can be seen in Figure 19.

Reinforcement learning uses the reward-punishment technique for learning. Based in the performance of the agent a positive or negative (punishment) reward is given. With enough time and rewards, the agent learns to maximize the reward and thus, to perform the required task correctly.

ML-Agents[ix]

ML-Agents is a plugin provided by Unity-Technologies[xxii] that allows the usage of some machine learning techniques in the Unity game engine. It simplifies this task by taking care of the implementation of the machine learning algorithms by himself and letting the developer focus on the environment and agent developing.

In this project, due to time and scope considerations, this plugin has been used to create an AI capable of playing the game *Auralux* learning to do so by its own. However, the plugin executes following its own cycle and thus, some adaptations are necessary to use it. This adaptations are manly the use of class heritage to implement functions that are later called by the plugin.

Setup

In our case, the environment is already developed. Levels of the game will be used to train and to let the AI show its acquired abilities. The main task is to obtain an appropriate agent, able to learn and to interact with the environment properly. As a base, a normal AI player of the game it is used. It can interact with the environment both acting upon it with its `Effector` and gather information about its state.

For the environment, both normal and training versions of the game will be used. The first one is used when the model has already been trained and it is ready to play against humans. The second is used to, as its own name hints, train the model.

Agent

Because the machine learning techniques are controlled and executed by the ml-agents plugin, agents only need to be able to interact with the environment and gather data. As it has been said, this has already been accomplished with a regular AI player, which can is able to interact using its `Effector` instance and can gather information from the scene.

However, the information needed is beyond the capabilities of a normal AI player for obtaining it. Because of this, the agent needs to be modified to be able to access the information directly from the game class which contains all the information in the environment. In addition, it has to implement some base classes provided by the plugin to be able to have some functions that belong to its (the plugin) execution cycle. This way the agent can send and receive information from the model.

The agent interacts mainly in three different phases with the plugin cycle:

- Sending information: When the plugin needs to gather information to feed the model and let this last one take decisions, this phase is executed. It is implemented with a function called `CollectObservations` that has to be implemented in the agent.
- Executing actions: After sending information, the model outputs values that the agent has to translate to actions and execute them.
- Reward: After executing actions, the agent has to gather information again and determine how good or bad it's situation is. This information is sent to the model in the form of positive and negative rewards.

Training

In training phases the plugin takes total control over the environment execution. It is executed at very high speed and saving as much as possible in the graphical aspect to provide faster and more efficient training.

Training is simply executing the environment (in our case, a game) over and over again until some criteria is met (for example, one million games have been completed).

Developers can program how it has to be reset every time one of this cycles ends and other minor things.

In our case, training sessions between 500.000 and 1.000.000 steps long were performed several times changing the output and the reward system of the model to try to get the best possible results. A plugin-adapted training version of the player was used.

For more information in this sessions and the changes between them, see Annex II – Training Sesions.

Normal Gameplay

Once the model is trained, it can be exported as a file and used inside the normal version of the game. The feeds output to it and it provides output which is provided to the agents by the plugin. The environment executes normally. A plugin-adapted version of the normal player is used.

Results

The usage of the plugin and the necessary adaptations were done in much less time than planned originally. This task was surprisingly easy and no major issues were found.

However, designing the agent and implementing it (by adapting the already programmed players) was harder. Not only that, the training phase was very deceiving. Not only the process is slow and hard to debug, but the obtained results were far from the best-case scenario.

To measure its success, the model-controlled player was confronted with random AI controlled players in a large number of games. The obtained models had only a slighter better performance than players executing random actions and even then, obtaining this models was a hard and long process. In fact, the best model obtained only 65% of victories in a two player game environment.

We believe that this might be due to a poor reward or/and input system that does not provide the model with adequate values.

Because of this, the difficulty adaptation part has been again suppressed again from the project to try to obtain better models.

4 TESTING AND RESULTS

4.1 Results

Although some already mentioned problems, the results of this project are, nonetheless, satisfactory. The main objective, the implementation and usage of machine learning techniques, has been reached and functions correctly. However, as it has been said, some parts of the project have been abandoned due to poor scheduling and lack of time.

The complete development of the project by weeks can be seen in Annex III.

- Research of methods of adapting machine learning techniques to video games. ✔
- Implementation of Montecalo Tree Search and its integration in the game flow. ✔
- Usage of Deep Learning as an AI in the game. ✔
- Difficulty adaptation. ✘

The last part, the difficulty adaptation of the implemented techniques has been completely suppressed from the project.

In terms of the main game, we consider its implementation a success and no major problems were encountered.

As form MTS, its implementation and execution, although more difficult than expected, were also successful. Not only that, we consider that our implementation is efficient and optimized. This is supported by the fact that, because the tree has to be started from scratch every AI tick it still presents a challenge to the player.

However, the time consumed in the implementation and testing part was more than expected and it could not be adapted to modify the difficulty in regards player performance.

Lastly, Deep Reinforcement Learning implementation was easier than expected. However, training and obtaining good results as not. The performance of the agent was 65% of victory in a 2 player scenario where one was controlled by deep learning and the other executed random actions.

4.2 Testing

To test performance and implementation success of M.T.S. and Deep Reinforcment Learning as well as simulations, the following procedures have been designed and followed:

- M.T.S.

  This technique has two main parts that need to be tested: the tree construction and the decision making process. Both of them were tested by executing the algorithm very few iterations (enough to create a big enough tree for testing) and by revising the creation process and the results by hand.

  In terms of performance, the test was performed by obtaining the greatest child number that had been visited during a more lengthy execution. The numbers obtained tend to differ greatly, which is normal due to the random nature of the simulations. However, the maximum child number tends to be between 80.000 and 120.000.

  Another factor that proves the correct execution of the algorithm is the fact that the number of iterations decreases when the root state is close to the game end.

- Deep Reinforcement Learning

  Because the algorithms and procedures are implemented in the ML-Agents plugin, the only thing that can be tested in our side is the model performance. To do this, the model is confronted against other AI players thousands of times and its performance is obtained as its ratio of victories.

- Simulations

  Simulations were tested by executing them step by step and analyzing their performance. All edge cases were simulated specifically and later revised to ensure correct executions.

4.3 Links

The game ready to be played, Unity project files and videos showing the most relevant AI combats can be found HERE.

5 CONCLUSION

This project has been a very long and sometimes frustrating journey. Parts of it have been abandoned and other parts have taken its time only to provide with improvable results. However, from its three main parts (leaving documentation out) the three have been successfully implemented which opens the door to future revisions that allow for the correction of the actual problems.

The first part, the game itself, was the easiest one. Due to the simplicity of the game and the growing experience of the author, it was the only part completed in time and with good results.

The second part, Montecarlo Tree Traversal, was the most complex in terms of implementation. The base algorithm although it might appear simple, hides lots of edge cases and unexpected complications. In top of that, implementing it efficiently provides with added complexity. Nonetheless, the results are even better than expected, even with the huge loss of potential due to the real-time nature of the game.

The third and last part, Deep Reinforcement Learning is the one were the results are most disappointing. The easier and faster implementation than expected is overshadowed by the poor results.

On top of that, the difficulty adaptation parts have been completely discarded from the project due to time issues. This has been caused due to poor time-estimations and the difficulties that have been found during development.

Despite all of the above, the overall experience has been gratifying. Overcoming unexpected problems and using an experimental plugin for experimental techniques has resulted in interesting (and sometimes stressful) challenges. Challenges that, at least most of them, have been overcome.

The author would also like to thank its professor and tutor, Raul Montoliu Colas, for introducing machine learning in his Advanced Interaction Techniques course, allowing him (the author) to discover this weird and sometimes magical world.

ANNEX I

MONTECARLO TREE SEARCH ALGORITHM AND ITS IMPLEMENTATION IN AURALUX

A1.1 What is Montecarlo Tree Search?

We are going to begin this part by explaining the basics of this algorithm and its uses. After this basic insight has been provided the implementation used in the game will be explained in detail, step by step, as well as its execution flow. However, before we can even start to explain the basics of the algorithm, there are some concepts that have to be understood by the reader:

- State: The values of the properties that configure a system at any given time, in other words, the information necessary to describe perfectly how is everything in the game. If we took a snapshot of the screen while playing the game we could see the number of units that each planet has, the position of the attacks (if any) and the units they carry, which player owns which planet, etc. That will be a state of the game, because we could "load" those values in the game and continue playing from that point on.
- State space: A collection of all the possible states of a system at any given time. In our case, imagine that we are playing the game and we wish to store every possible frame that can be displayed. That is, every that is different from any other frame because there is a different number of units in a planet, or because there is a planet that belongs to a different player or because there is an attack with a given number of units in a different position, etc.

The Montecarlo Tree Search (MTS from now on) is a heuristic search algorithm used in decision making. In other words, it is used to search something in a way that tends to lead to the desired result faster and that the result is used to make a decision. In greater detail:

- Search algorithm: If this is an algorithm which objective is to provide an agent with the best possible decision at any given moment, why is it defined as a search algorithm? What is it looking for?

  First of all, we have to understand that, even tough the game is represented with graphics trough a screen, the computer itself it's only able to process numbers. That implies that at any given moment, there is a group of numbers stored in memory that represent the state of the game. And when the graphics in the screen are updated, these numbers change accordingly. That means that for every state, for every frame, there is a group of numbers that configure the state of the game, a concept defined before as the *state space*.

  There are a lot of possible states in the game, represented by these numbers, but although they might be billions of them, its number it's not infinite. We can collect and store each and everyone of them, even if it uses lots of memory (again, this is the *state space*). And that has an interesting implication: inside

that collection can be found every possible combination of moves that can be executed in every possible situation and all their possible consequences.

Imagine that we freeze a game of *Lux Aura* between player A and B. The state of the game can be found inside the *state space*. But alongside this state are every possible state that can be reached by performing (or not performing actions) from this particular snapshot of the game.

Imagine now that in order for player A to win, a simple attack to the enemy is necessary. That means that to win, the state of the game will change to one where player A attacked, to another one where the attack was heading towards its objective and a final one where the attack reached the enemy and player A won.

The game has been in four different states until it was finished. But player A could have done any other action, leading to other states that might not end with him winning. So, if player A could find the chain of states that ends with victory at any given time, he will always win.

It is precisely that sequence of states what MTS seeks. It needs to know what sequence of actions to take in order to win at any given moment. And it is searched in the *state space* of the game, starting from the current state of the game.

- Tree: The states are organized in a tree structure. The root node is the initial state. Each node has as many children as there are possible actions that can be done. Using this structure, searching is quicker and more efficient. An example can be seen at Figure 20.

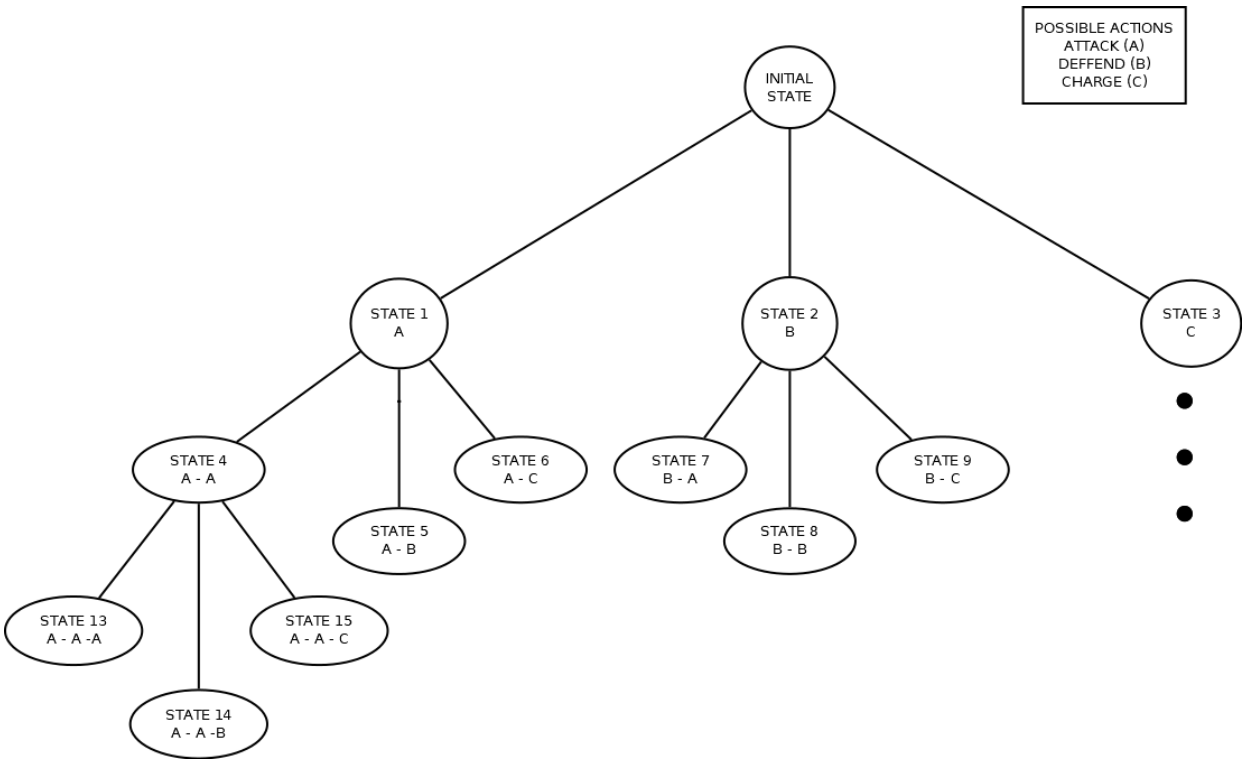


Figure 20: Example of a Montecarlo Tree structure

- Heuristic: The state space is huge. If there was only one planet the space state would have approximately 37 million possible states. That means that a normal search to find the desired sequence of actions is out of the question. The algorithm needs *something* to help him find that sequence faster.

  That is where heuristics come in handy. They are a kind of measure that allows an algorithm to know how "well" is it doing in its task and to point out if the decisions it has made *appear* to have brought it closer to the desired result or not, so it can then continue or stop and redesign its strategy. They are not perfect and can fail (that is, they can result in a non optimal strategy) because are based in estimations, but in situations like this its usage it's necessary.

- Decision making: Once the algorithm ends, its results describe the odds of winning after executing certain actions. This information is used to take decisions.

MTS execution can be briefly explained (in the case of game applications) as follows: starting from the current state of the game, the algorithm looks through the *space state* for the sequence (or sequences) of actions that end with the victory of the player that is using it and returns the odds every action to be the start of a sequence that leads to victory.

It has been applied in video games such as *Total War: Rome II*[xxiii] and *AlphaGo*[xxiv]. This last case, the most recent and famous one, uses MTS in combination with *deep*

learning[xxv] to play the game Go[xxiv], a table game for which a computer program able to defeat humans was impossible to program until 2015, the year *AlphaGo* beat the first professional human player.

A1.2 States

We have previously defined what is a state and a *state space*. However we have not defined yet what information is necessary to describe a state in our game. A state o a *Lux Aura* game can be divided in three sub-states:

1. States of the entities

   Information that describes the state of a single entity at any given moment (31)

2. States of players

   Information that describes the state of a player at any given moment. The state of a player contains the states of the planets that controls (32)

3. State of attacks

   The information that describes the state of an attack at any given moment (33).

This states are represented internally with the classes TeventEntity, Tplayer and TattackInfo respectively and are stored inside an instance of the class TGame. The *state space*, again, would be a group made by all the possible states; that is, all the possible combinations of the presented values.

To obtain these states, the previously explained in Snapshots section snapshot methods are used. When invoked, they gather the necessary information, create an instance of the training class that is required and initialize it with the gathered information. The result is the state of the object the moment the method was called.

Table 31: State of entities

INFORMATION	VALUE RANGE
Owner	-1 – 5
Level	1 – 3
Experience	0 – 150
Experience for next level	0 – 150
Health	0 – 100
Units	0 – 9999
Current contestant	0 – 5

Table 32: State of players

INFORMATION	VALUE RANGE
Planets	List of entities that can be at any state

Table 33: State of attacks

INFORMATION	VALUE RANGE
Remaining turns	0 - 100
Player that fired it	0 - 5
Units	0 – 9
Id of the objective	0 - 10

$$\frac{score}{visits} + \phi * \sqrt{\frac{\ln(parent\ visits)}{visits}}$$

Formula 4: Montecarlo best child formula

A1.3 Heuristics

As explained before, a heuristic is a measure of some kind used to tell if a process *seems* to be going in the correct direction to obtain the results it desires. It is used in many programs and search algorithms to optimize it's execution time by giving priority to those paths or decisions that seem to get the algorithm closer to its objective. Its usage has a counterpart, and that is that the algorithm might not always find the best solution.

In our case, the heuristic is used to explore further the decision sequences that have a higher rate of success (that is, those which simulations ended in victory more times). It translates to having a value per node that shows how "good" it is and to decide which one should be explored. The value comes from Formula 4.

When we need to choose between the possible actions, represented by child nodes, this formula will calculate a value for each possible option. Once it has been done, the higher value obtained is chosen and the node that obtained said score is explored.

We can distinguish two parts in the formula:

- Exploitation term: This term gives weight to keep exploring nodes that already have been explored based on their results. In other words, it's the term that pushes the algorithm to only take routes that have been taken before and have had good results.

$$\frac{score}{visits}$$

Formula 5: Exploitation term

- Exploration term: This term gives weight to those nodes that have not been visited or have been visited less times. In other words, it pushes the algorithm to prefer nodes that have been little explored against those that are well known and have given good results.

$$\sqrt{\frac{\ln(parent\ visits)}{visits}}$$

Formula 6: Exploration term

The algorithm has to find a balance between these two values that allows it to explore enough that the best sequences are found but, at the same time, exploit the good results obtained to ensure that they are, indeed, good.

To do so, the constant value "ϕ" is used. A higher value will focus the algorithm more on exploration and a lower one will make the algorithm exploit more the good results already obtained. In our case, different values were tried and we believe that the best performance for our case can be found between values 1.8 and 2.2.

Another thing to consider is the fact that the formula will return an infinite value for non-explored nodes. This is because the visits of a non-visited node are 0. When the division of the second term is computed, an infinite value will be returned. This implies that the algorithm will always prefer nodes that have not been explored yet instead of already visited ones, whichever might be its score.

A1.4 M.T.S. Algorithm

Now that the necessary terms have been explained and that we gave a general overview about what does this technique exactly does and how are defined the states through will it will search for the best possible sequence of actions; it's time to explain its algorithm. A diagram of this process can be found at Figure 22.

First of all, although the algorithm searches among every possible state of the game, the complete *state space* is not stored anywhere. Storing it would require lots of memory and, because of its size, searching anything on it would require high resources usage. Moreover, having every possible state is not necessary, because ate very given moment, only a fraction of it is accessible. For example, having a level with five planets discards immediately every state that had more than 5 planets in it.

Because of this, the algorithm starts by storing the actual state of the game. That is, it takes a snapshot of the whole game and stores it as its initial point. Starting there, the search begins. But before we can start explaining the actual process, we need to state the information that is contained in the nodes that will populate the tree.

A node contains three values:

- A state of the game
- Number of times it was visited
- Score

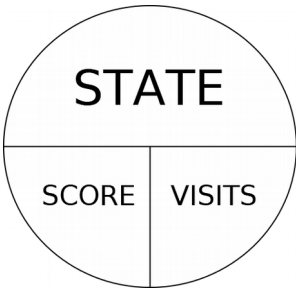


Figure 21: Diagram of a node

Having these, the process can perform all the actions required. The algorithm follows these steps:

1. The state of the game is obtained using snapshots and it is placed at the root node. The algorithm will start from here, that is, the root node will be the current node.

2. Main loop of the algorithm:

   2.1. If the current node has children:

      2.1.1. The best child is chosen:

         - For this, Formula 4 is used.

2.1.2. Once a child has been chosen, it becomes the current node and the algorithm returns to step 2.1.

2.2. If the node has no children:

2.2.1. If the node had not been visited before a rollout is performed.

- A rollout consists of a simulation in which the starting state is the one found in the node that is being explored and continues until the game ends.

2.2.1.1. Once the simulation ends, depending on its result, a score is obtained (for example, if the game has been won the score will be 10 and if it was lost it would be -10).

2.2.1.2. The score is back-propagated.

- This is, the score is added to all the nodes that were visited starting from the current node and until the root is reached.
    - CAREFUL: Only the obtained score is added to each node, not the total score of its child.

2.2.1.3. We start from the root again at step 2.1

2.2.2. If the node had been visited before, it is expanded

- For every possible action, a new node is created. That implies creating a new node, simulating a step in the simulation performing the corresponding action and setting the node as a child of the current node.

2.2.3. Once the node ha been expanded, the algorithm returns to step 2.1 with the node that was expanded as the current node.

- To save time and resources, a rollout can be performed in the first child of the node, because that is the one that will be chosen by Formula 4.

3. This process continues until it finishes or it is stopped (see next sub-section).

4. Once it is done, the best action (the child node of the root that has the best score) is returned as the best action that the algorithm found.

Start and finish

Given the huge size of the *state space*, we can't expect the algorithm to search trough all of it in a reasonable amount of time except in the cases where the game is close to being finished. Because of this, the algorithm has to stop based on factors like time or number of iterations to obtain a decision in a period of time short enough to be acceptable in the flow of a video game.

Since in our case a clock has been already implemented, we will use time limitations. When the execution of the algorithm is started, a new timer will be set to trigger an event after a certain amount of time (in our case, 2 seconds were enough). The main loop of the algorithm (step 2) will be executed until the timer triggers and event. When this happens, the algorithm will move to step 4, and the exploration will be stopped.

This implies that, since not all possible states will be explored, in order to obtain a good solution the algorithm needs to have enough time to travel through the biggest possible part of said space. In other words, it has to run as fast and efficient as possible to search through the highest possible number of states until it runs out of time and stops. This has been accomplished using the techniques described in Implementation Notes section..

A1.5 M.T.S. Trace

To help further in M.T.S. understanding, we provide a trace of a simplified case first steps execution. In this scenario, there are only two possible actions that can be performed at any given moment. There are only two possible outcomes with +20 and +10 score values (victory and defeat).

The trace can be found in Figure 23, Figure 24 and Figure 25.

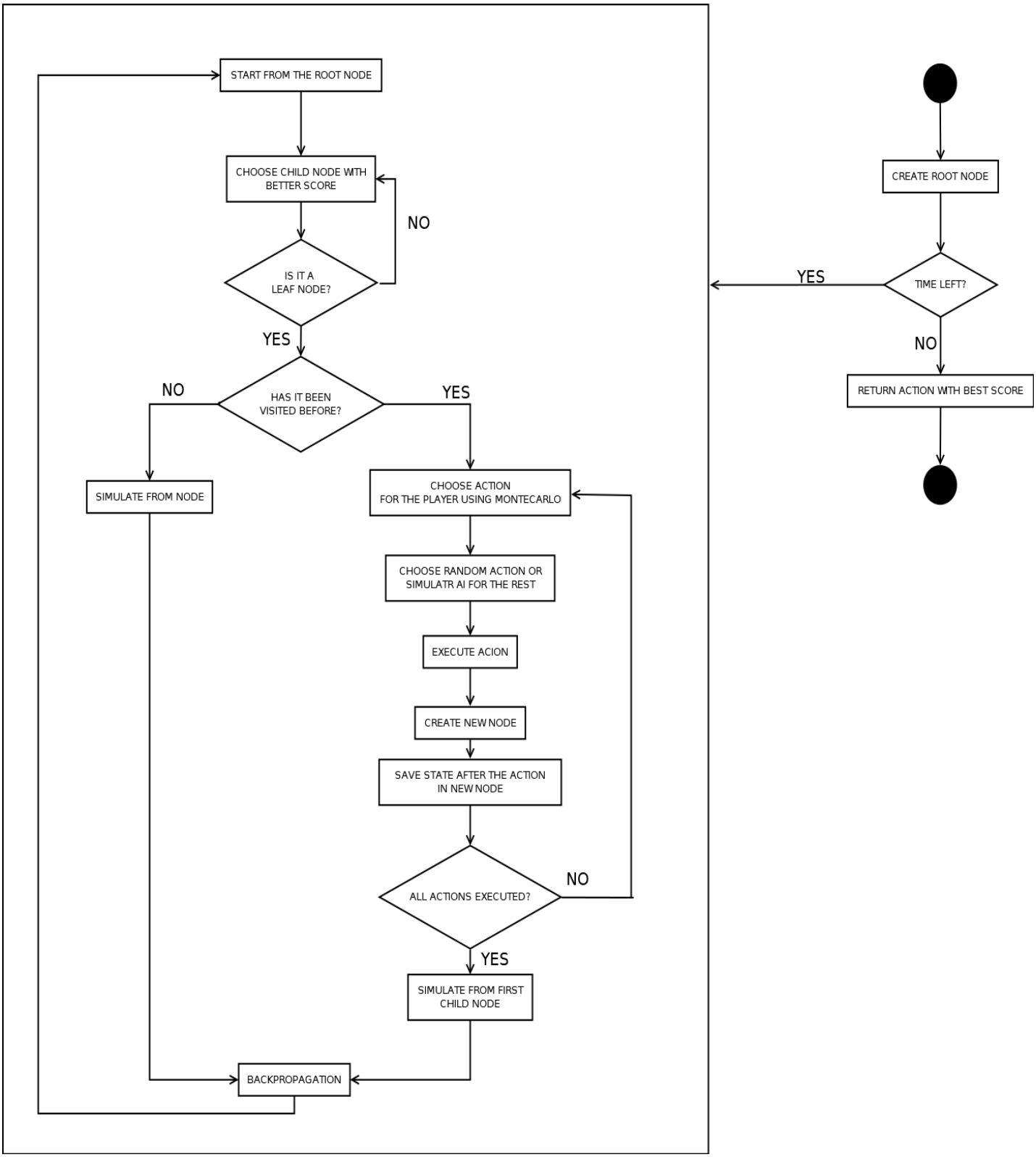


Figure 22: MTS execution flow

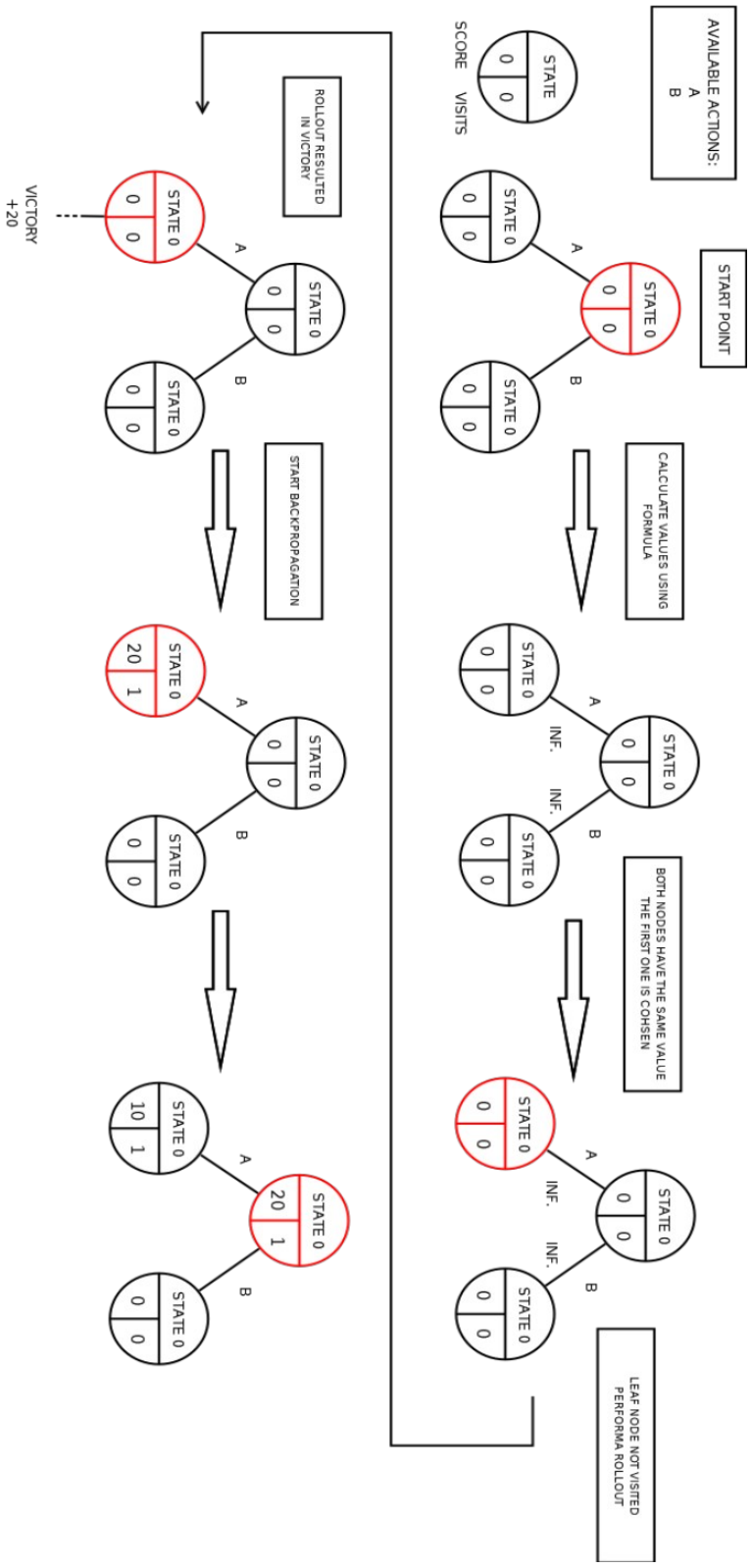


Figure 23: First part of the M.T.S. execution trace

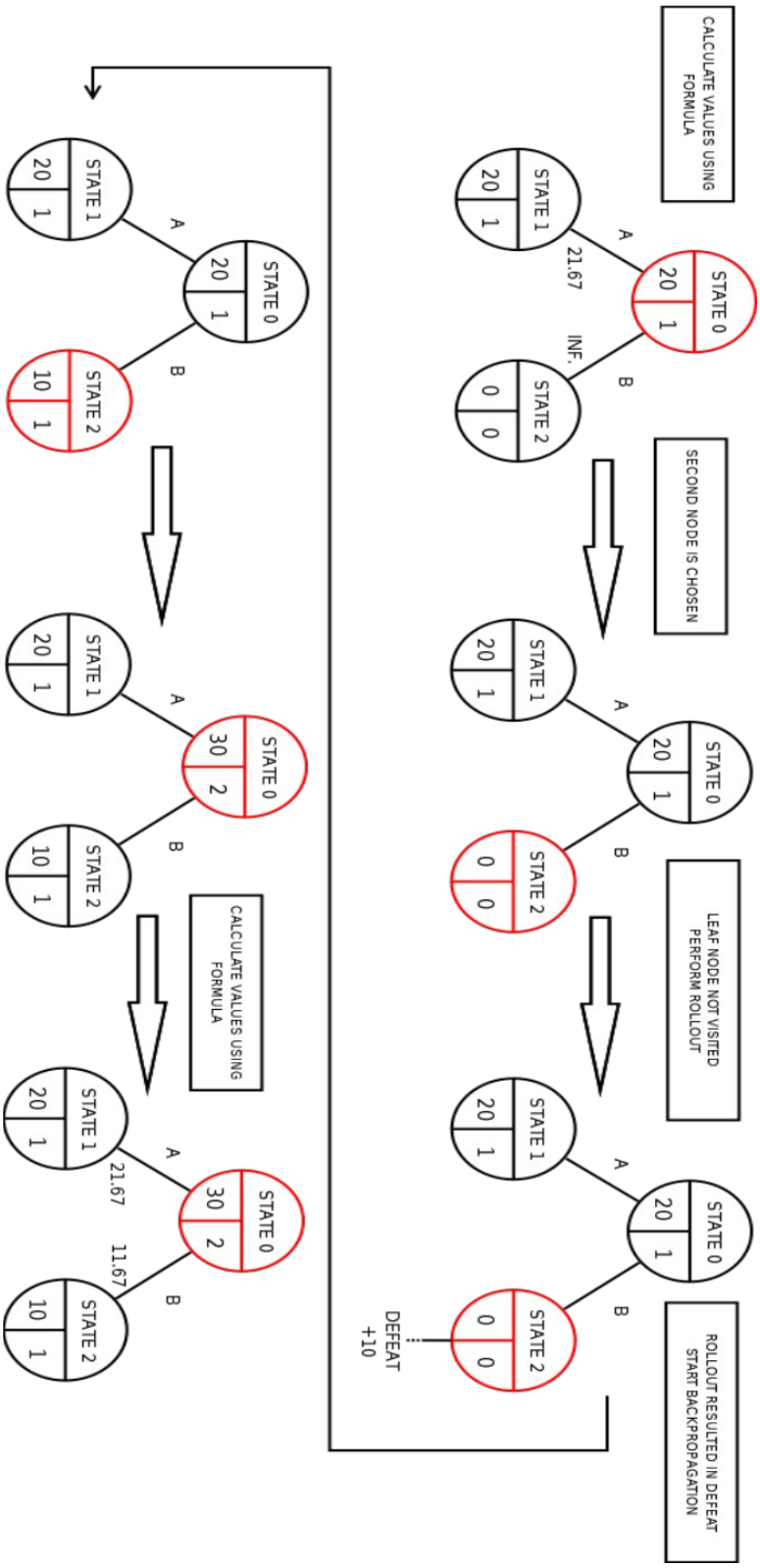


Figure 24: Second part of the M.T.S. execution trace

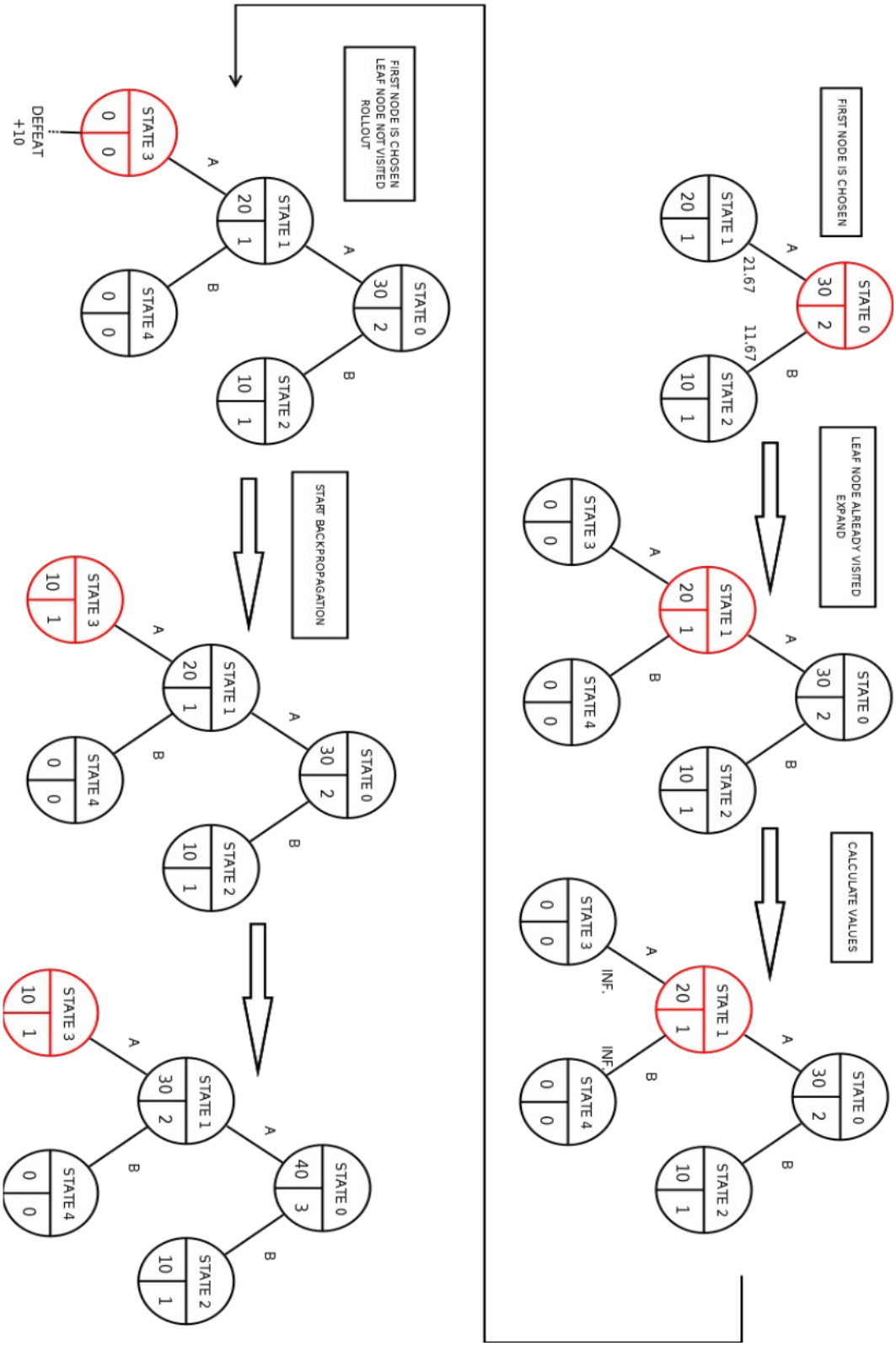


Figure 25: Third and last part of the M.T.S. execution trace

$$parent = \frac{child\ position - 1}{5}$$

Formula 7: Formula for finding parent

$$child = parent\ position * 5 + child\ numbe$$

Formula 8: Formula for finding children position

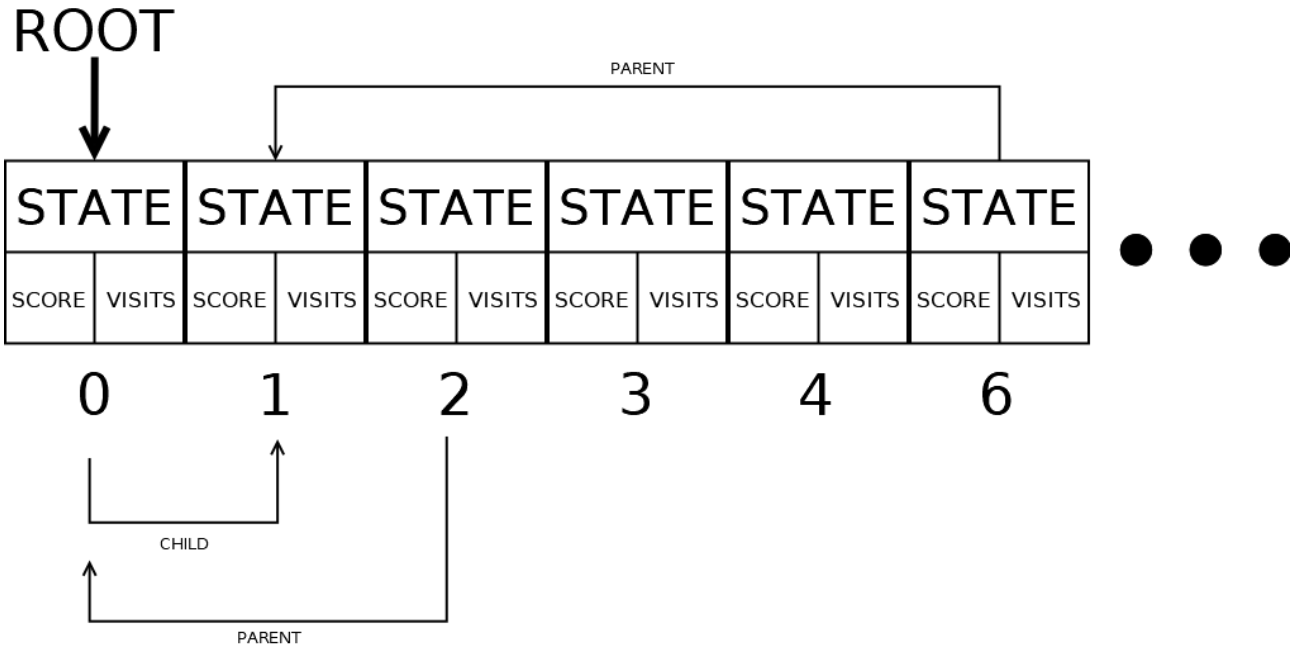


Figure 26: Diagram of the array structure

A1.6 Implementation Notes

In this section we will expose how this algorithm has been implemented to obtain an acceptable level of performance and use the available resources in the best possible way. However, there might be other ways of implementing it that might be better for other scenarios and/or applications, so we let the reader decide to follow or not these notes in its own implementation. The two techniques described optimize the two main aspects of the algorithm: search and simulation, respectively.

Array implementation

Because we want to obtain the best performance possible, the tree structure used by the algorithm has been implemented using an array. This results in the exploitation of the available cache[16] memory, because the information will be stored continuously in memory, and thus, in faster accesses. This will make the search part (the one in which the algorithm travels from node to node) faster.

Using this structure implies that we no longer need to store memory addresses for the children and parent nodes, only the position of the node in the array is necessary to obtain them. To find the children or the parent of a certain node using this number, formulas Formula 7 and Formula 8 are used.

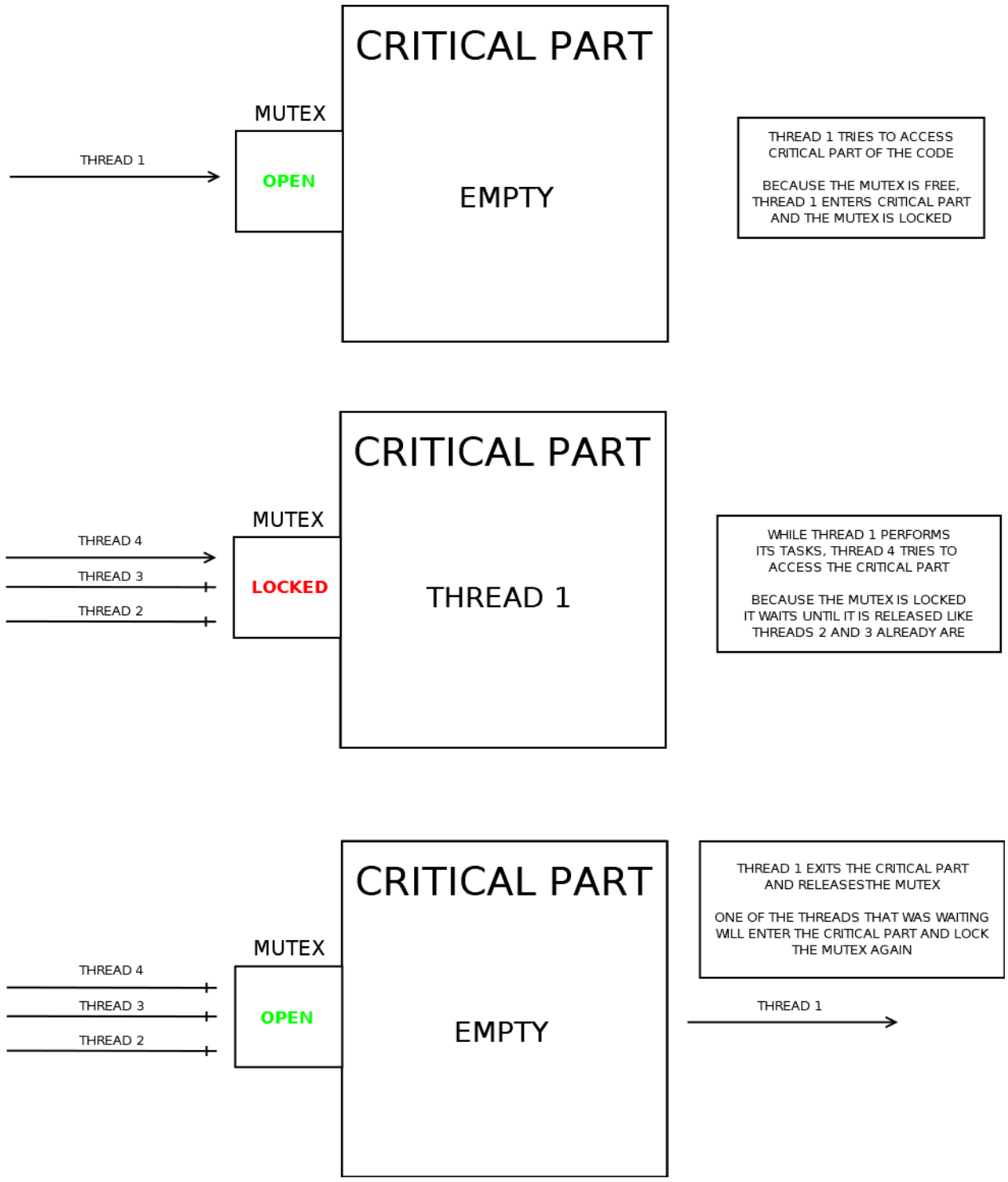


Figure 27: Example of mutex - dependent execution

Parent and child position refer to its position in the array, while child number refers to which one of the exisitng children we try to access. The first two have a value range that goes from 0 to the length of the array and the second one from 0 to 4. This is because each node has as many children as possible actions and, as it has been previously explained, in our case there are only 5 possible actions. A diagram of the array can be found in Figure 26.

Threading

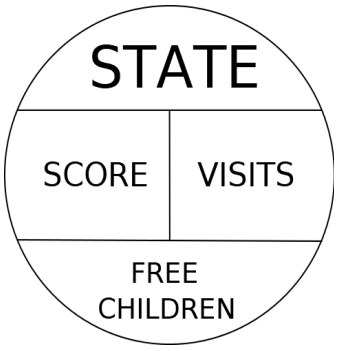


Figure 28: Node structure prepared for threading

As already exposed, we need to explore the maximum amount of the *state space* possible until the algorithm runs out of time and its execution stops. The most expensive part of this process is the "rollout", or the simulation of a game until it ends. This part could result in only having to simulate a few steps or having to simulate a whole game from start to finish. The length of the game will also vary, because of the randomness of the decision-making involved. In other words, the simulation itself can't be optimized any further than the optimizations already done when creating the training version.

The only than that can be done, then, it's to execute simultaneously as many simulations as possible and using the available resources as much as possible while there is still time to do so. To accomplish this, threading will be used.

Threading[xxvi] is the execution of simultaneous flows of execution from a process (a thread), sharing the information this "parent" process has (variables) among other things.

To extract the most performance of this techniques, there are some minor modifications that need to be done in the structure of a node; but before we enter in detail, the reader has to keep in mind two things:

- Simulations are independent

  A simulation is completely independent from every other one in the tree. They start from a state stored in a node and, for every node, only one simulation is performed. That implies that, as long as simulations start from different nodes, they can be performed at the same time.

- Modifications in the tree are not

  Searching a new node to explore or back-propagating a score are tasks that can't be done simultaneously, because it might result in race conditions and the obtaining of strange and unpredictable results. This is because unlike the states from which the simulations start, the tree data is common and thus, every modification is performed over the same data set. Thus, thread synchronization[xxvii] is needed.

In other words, there can be as many simulations as needed being computed at the same time, but when the tree has to be traversed to find the next node to simulate or to back-propagate a score, it can only be done by one thread at a time. With this said, we can start to design the structure and function of the treads that will be used.

Because the algorithm relies on a main loop that executes the same logic over and over, we can simply use a thread for every iteration that has to be done. That is, a thread will be in charge of looking for a leaf node to simulate, execute its simulation and back-propagating the obtained score. As said before, the first and last tasks are not independent. That means that when a thread needs to back-propagate the score or to search a new node, it will have to check if there is another thread accessing the tree (no matter if performing the same or a different task) and, if there is, wait until it ends.

This has been achieved using locks or mutex[xxviii], synchronization mechanism that allow to block the execution of threads when critical parts of the code are reached. The reader can imagine them as a shop with only one fitting room. Lots of people can be choosing clothes at the same time, but when they want to see how they look with their new outfits, they need to wait until the fitting room is empty. The door of that room will be a mutex, because it would only allow to access the critical part (the fitting room) when nobody else was inside.

In our case, the tree has its own mutex. When a thread wants to modify its information, the mutex is closed. It will be released when the thread finishes whatever it needs to do. But until this (the release) happens, other thread that want to access the tree will check the mutex, see that it is locked and wait until it is released to continue. A diagram that explains this can be found in Figure 27.

Using threads introduces a new problem that we hadn't before: now we need to keep track of which nodes are being simulated and which are free. To do this efficiently, we have to change a little our node structure (Figure 28) adding the number of free children of the node.

By keeping track of the number of children a certain node has that are not being used by other threads to simulate, we can efficiently perform searches. Having this number will allow us to avoid visiting nodes where no children are available. The part of the algorithm regarding the search of a new node to explore with the necessary changes goes as follows:

1. Start from the root node
2. If it has free children, continue, if not, wait and repeat this step.
3. Select the best child, which will be the current node.
4. Choose the best child node.
5. Subtract one to the free children number.

6. The chosen child becomes the current node.
7. If it is not a leaf node, return to step 4; otherwise mark as occupied and continue with the algorithm.

Notice that if the leaf node is expanded, it will start with only 4 free children, because one of the new nodes will be immediately chosen for a *rollout*.

Now, the second part of the algorithm that has been changed, the one related to back-propagation of the score:

1. Start from a leaf node.
2. Add the obtained score to the current node parent's score.
3. Add one to the free children number in the parent.
4. If the parent is not the root node, return to step 1 with the parent as the new current node.

An example trace using this strategy can be found in Figure 29.

Advantages

Now that this changes have been explained, we will discuss why they are worth to be applied despite the increase on complexity of the code and is susceptibility to new and more difficult to detect errors.

Starting with the usage of an array to store the tree, this will allow for smaller access time periods when a search is being performed. Because the whole tree is stored continuously in memory and it is accessed so frequently (ideally there is always a thread that has ended its simulation and needs to perform operations using the tree date) the cache[xxix] is highly exploited.

Cache is a small but very quick memory used to retain information that is very likely to be used soon by the processor to reduce access time to it. The main principles that determine its usage (spatial[xxx] and temporal locality) determine that data that has been used recently and data closer to the later one will probably be used in the future. Since the tree data is constantly being used and traversed, and using an array implies that it is stored in a continuous block, it will stay in cache for more time and that will reduce access time to the information.

In the other hand, because the bottle neck is the access to this information, using threads grants that there will always be simulations running while other threads wait

for their turn to use the tree. Because simulating is more expensive than searching for a node, there should be a balance between the nodes that are waiting to access the tree data and the ones simulating, because various threads will have accessed the tree by the time a simulation is done.

Recycling

One last thing to keep in mind is that creating new threads has a cost. If we really want to obtain the best performance possible, we need to recycle threads to avoid paying the cost of creating them over and over again.

Given that the threads do always the same (search for a node, simulate and back-propagate) and that the results of their operations are stored in the tree data, we can simply make the threads repeat their functions once they are done. The idea is to create a reasonable amount of threads for the system capabilities and generate a pool[xvii] of threads ready to be used.

Threads have to be created as needed, that is, they will created whenever the tree *mutex*[xxviii] is free and stop once the maximum thread number has been reached. From that point on, the threads continue executing until a variable that has to be accessible and read-only for all of them tells them to stop and exit.

In our implementation, before a thread tries to access the tree data (either for searching a new node or for back-propagation) this variable is checked. If it has the exit value, the thread stops, if not, it continues.

A1.7 Using M.T.S.

Before we explained that, given the huge size of the *state space*, the algorithm is not able to explore all the states that can be reached in normal situations. This implies that, in order to get the best solution possible, the algorithm has to run the longest amount of time or the highest number of iterations possible. Since iterations can have different duration, we followed a time-based strategy to use M.T.S. in *LuxAura.*

In opposition to the other AI types, M.T.S. does not retrieve an action immediately. Its execution is started and, at the same time, a timer does the same thing. When the timer triggers an event (that is, when the amount of time we decided has passed), the main loop stops and an Action is returned for its execution. All threads are then suppressed. That implies that the events that occur in the game during the time that the M.T.S. needs to take a decision won't be taken into account in the decision-making until the next M.T.S. iteration.

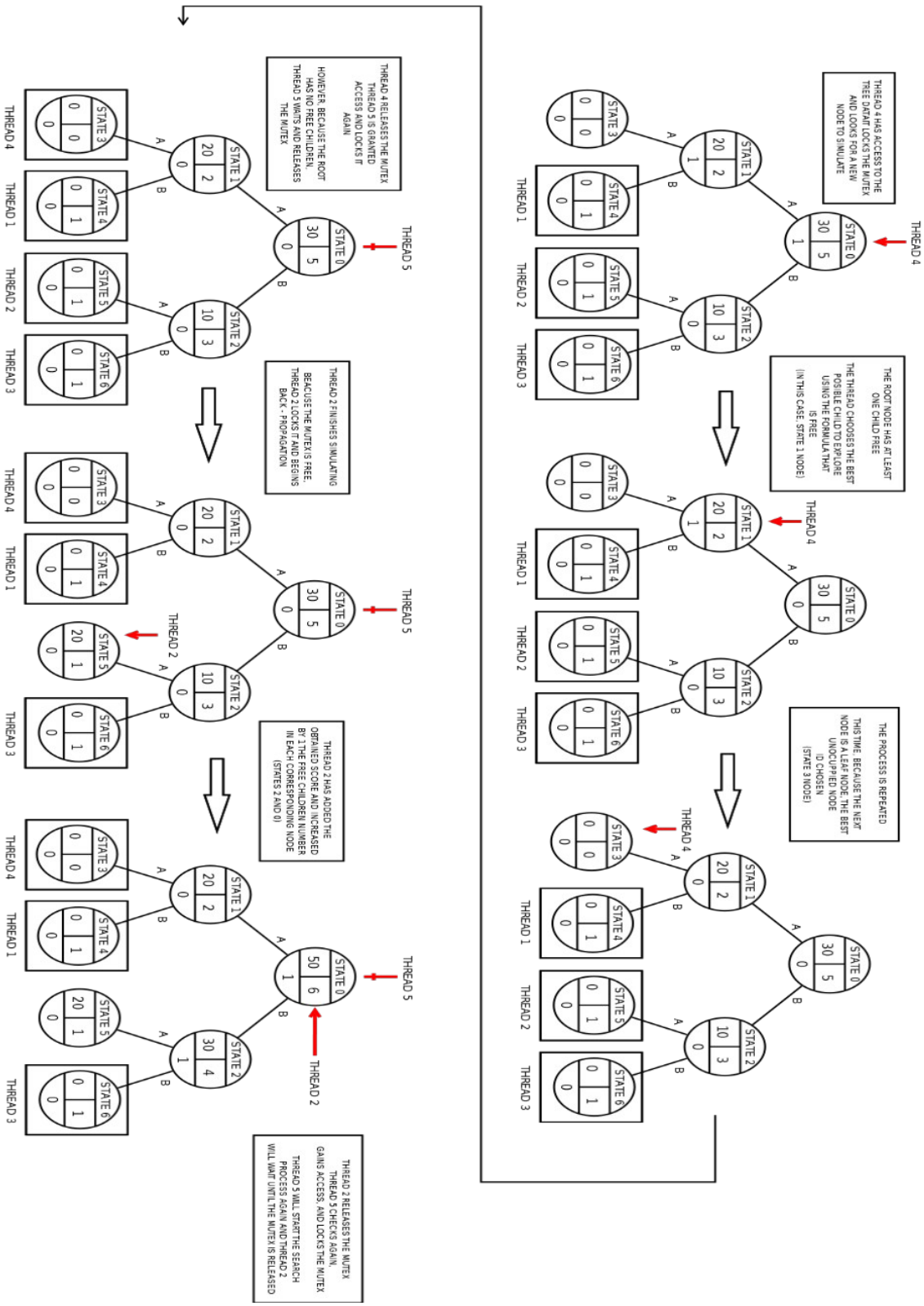


Figure 29: Execution of M.T.S. with threading trace

ANNEX II

DEEP REINFORCEMENT LEARNING AND ITS IMPLEMENTATION IN AURALUX USING ML-AGENTS PLUGIN FOR UNITY

A2.1 State of the Art

In this section the general state and concepts of machine learning will be explained. Once a general overview has been given, the techniques hat have been used in this project will be explained in a more detailed manner.

Machine Learning

The concept of machine learning refers to a group of techniques based on statistics designed to give machines the capacity to "learn", that is, to allow them to progressively improve their performance on a specific task based on data. In other words, techniques that gather and process great amounts of information in order to find relations in that data that allow for a performance improvement.

A basic example would be a software that allows a machine to learn to recognize apples in pictures. First, the machine would not be capable of recognizing apples, it would need to be trained. In order to train, thousands of images with apples would have to be provided to the machine. The software would then process this images and look for patterns in the data that allow it to successfully recognize apples. After thousands of images, the machine would be capable of performing its task with a high success rate.

The example used describes a specific machine learning technique, there are other methods that follow different approaches. However, all follow the same basic steps: training iterations (gathering and processing of information) that result in a progressive improvement in performance on a specified task.

This learned "capability" of performing a task is expressed in a model. A mathematical representation that, when given input proceeding from the environment outputs values that represent the next action to be taken.

This techniques have existed for a long time. The term itself was coined in 1959 by Arthur Samuel[xxxi]. It surged from the research of AI and had a promising early development. However, because it was based in statistics, a rift appeared between machine learning and AI, which followed a more logical knowledge-based approach. This caused a separation of fields that lasted until now at days. In the 1990 decade, this techniques appeared as a separate field and began to flourish, a trend that continues in the present.

Today, machine learning Is responsible for most of the advances in technology. From cars that drive themselves[xxxii] to personal assistants[xxxiii], machine learning is gaining more and more importance in our everyday life. And It seems that its importance will continue increasing for a long time.

Deep reinforcement learning is the combination of two of the most recognizable areas of machine learning: deep learning and reinforcement learning, which will be explained in greater detail below.

But first of all, there are some concepts that need to be clarified:

- Environment: Includes everything that influences in the learning process, that is, all the elements that the entity that is learning has to process and take into account to be able to improve.
- Agent: The entity that is learning. Notice that it doesn't has to be a machine itself. It can be a program or an element of a program (that is our case) that uses this techniques to "learn" to perform correctly a task.
- Actions: What the agent can do in the environment. Including the lack of action.
- Model: Models are mathematical representation of systems. When input values are provided to a model, it outputs another series of values that, if the model has been trained, represent the action that should take place next to perform whichever task the device is performing.

Reinforcement learning

This area of machine learning is heavily inspired by behaviorist psychology[xxxiv]. It is based in the reward-punishment principle. The agents performs actions that will be rewarded or punished. Based on those rewards, it is capable of identifying which actions to do based on the environment state to get the highest reward possible.

For this to be possible, the role of the interpreter is needed. This role is in charge of providing with a positive or negative reward to the agent based on the results of its actions. It's like a judge that decides whether or not the agent is performing correctly and rewards it accordingly.

Because the agent has to decide based on the environment state, a set of rules is needed to determine what will be seen by him and how. The agent perceives the environment using this rules and decides what to do, hoping to reach an objective state that will provide a high reward.

Deep Learning

Deep learning is part of a broader family of machine learning methods based on learning data representations. They generate models of data that give an appropriate output when given an input of some sort with a certain value. The most popular and the method that will be used in this project is deep neural networks.

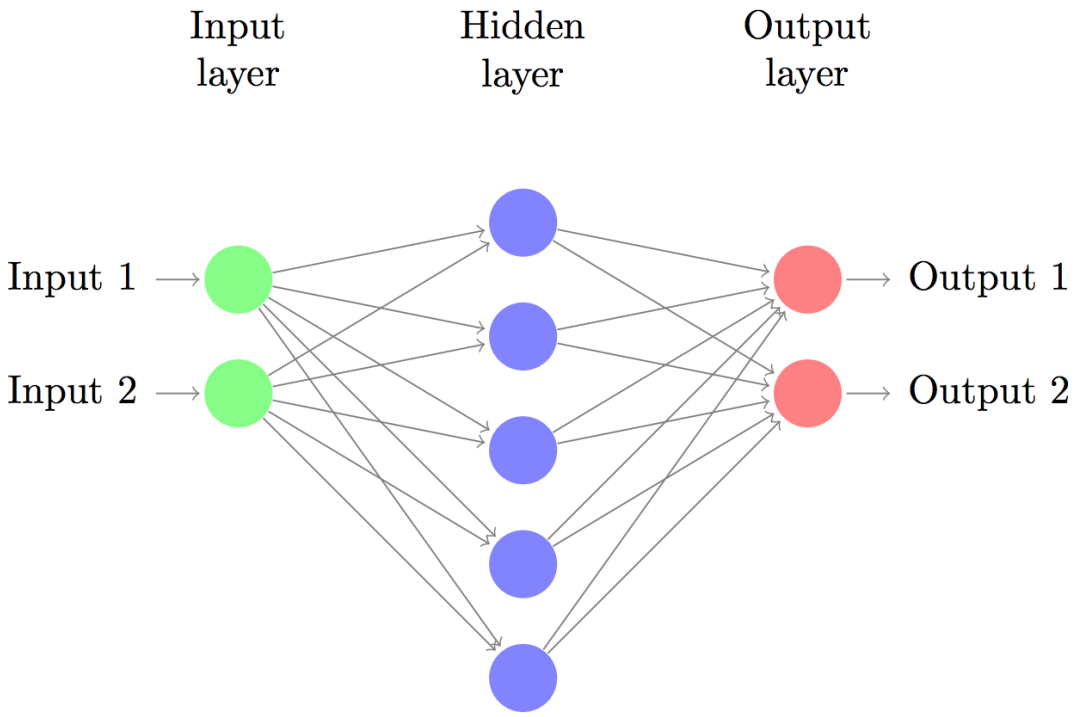


Figure 30: Example of a deep neural network

Deep neural networks are composed of a series of nodes linked together. Three parts can be found: input layer, hidden layer(s) and output layers. In the input layer the values that are needed to take a decision are used. This values are transformed in the hidden layers and then the desired values are appear in the output layer. The values are transformed using numbers that are obtained though training sessions.

Each node contains a series of values (weights) used to modify the input and provide a processed output. The original value enters through the input layer and travels all the way to the output layer. When the values traverse the different nodes, they are transformed until the final values reach the last layer, the input layer. These values are then used to make a decision.

Deep Reinforcement Learning

Being a combination of the previous two groups of techniques, deep reinforcement learning uses neural networks as support for the model that is created and trained and the reward-punishment system of reinforcement learning to adapt and improve the model.

Training

Another characteristic of machine learning techniques is that they require extensive cycles of training to be able to learn. Because agents have to learn, they require periods in which they try to resolve the problems they will have to face later in controlled environments. This training periods can be extremely long and include thousands or millions of examples, depending on the complexity of the task.

Is during this training periods where the agents adjust their models to be able to perform correctly in these tasks. In our first case, reinforcement learning, the agent learns to maximize the reward obtained when performing actions in an environment and in the second one, the *DNN* adjust their internal nodes to provide with a correct output when an input is provided.

Training is the most important aspect of machine learning, because it determines the situations in which an agent will respond correctly when provided with data. If an agent is trained in an environment and that environment changes after training it, it will most likely not be able to respond correctly to any given situation. If the agent receives different or inaccurate input, its response won't be good to perform the required tasks either.

Besides, training an agent to make it behave correctly in different environments and situations is extremely difficult and complicated. The more controlled and specific the environment is, the more likely is for the agent to learn and adapt efficiently, but the agent will have less flexibility in terms of changes in the environment.

A2.2 ML- Agents

For this project, we have used the *ml-agents*[xxii] plugin provided by Unity Technologies[vi] under the Apache License 2.0[xxxv]. This plugin allows the usage of machine learning techniques in the Unity engine. It simplifies the implementation of this techniques by allowing the user to skip the implementation and programming of the data structures and algorithms and enables him to focus only in designing the environment and the training for the model. In this and later sections the basics of the plugin functioning will be explained as well as how has been done the implementation for this project.

Ml-agents (from now on, *MLA*) works synchronously with the engine during environment executions and as a controller during training. This is done to maximize performance by adapting the unity flow to the necessities of the plugin. The training part is executed externally using *Python.*

Moreover, it could be said that the plugin itself it is only a bridge to communicate unity with the python environment that executes Tensorflow[viii], the framework in which the model is created and trained.

The functioning of the plugin can be summed up in a 3-step cycle:

1. Observation: Information of the environment is collected and sent to the neural network.
2. Action: The neural network returns an action, which is executed.
3. Reward: Based on the changes in the environment, a reward is sent to the neural network.

Repeating this cycle thousands, even millions, of times the model improves its performance in the execution of the task that it is being trained to do. While training, the model is not in Unity, it is in the *python* environment. Necessary variables and the programs in charge of creating and updating the model are in that environment too. Unity is only used as support for the simulated environment with which the agent will interact.

However, executing the environment normally wouldn't be enough, because the training could take even weeks. The python environment takes total control of an executable with the environment and executes it in very low resolutions and a hundred times more quickly than normal. This allows for fast and efficient training.

Once the training is done, the model is exported as a file that can be used by the plugin to execute it inside Unity. That means that the python environment that controlled the engine is no longer necessary and the game can be played normally. By using the provided file, the neural network can be questioned for actions at any given time during the game execution.

A2.3 Academy, Brain and Agents

These are the three components required in a unity scene for the plugin to be able to function. In this section, the basics of its functioning and, in greater detail, its implementation in the project will be explained.

Academy

As its name suggests, the academy is in charge of setting the environment correctly. It is in charge of controlling the time scale of the engine (the speed at which the time inside the scene will advance), the resolution of the environment, the steps that will be executed until it is reset, etc. It also controls the other plugin-related objects in the scene.

It is also used to change the environment and to reset it. How the environment is reset and how should it change as time passes are functions defined in this object. However,

due to singularities in the project and decisions taken during development, these functions have not been used.

For our project this component was simply added. No changes were made.

Brain

Brains are the components in charge of communicating with the model. They send information and receive the outputs. It needs an academy in the scene to be able to do this, and the brain it has to be a child object of it. Brains have different modes, but here only the two used in the project will be treated.

- External: In this mode, the model is not in the unity environment, but in the external python one. During training phases, they share the information of the scene with the python environment and receive output from it.
- Internal: In this mode, the model is inside the unity scene and attached to the brain. This time, the brain itself feed the model with information and obtains the output.

Because brains are in charge of collecting and sending information to the model, the amount of information they should expect needs to be set. In our case we configured it with the following values:

- Continuous observation space type: Because data like ratios will be provided to the model, the space type has to be continuous, that is, the range of values it can get is not limited.
- Observation space size of 4: The number of values that will be provided to the model.
- Discrete action space type: The number of possible actions Is limited, so a continuous space type is not needed.
- Action space size of 5: There are only five possible actions.

Agent

Represents the entity inside the game that will "learn". It provides a series of functions that allow interaction with the brain. These functions, however, are executed externally, from the academy; which means that they can't be called when needed. Agents also need a brain in the scene. Various agents can use the same or different brains.

In our implementations, this are the functions and attributes needed. Keep in mind that the `Agent` class extends the `AI` interface, so the interaction with other classes is the same that with other AI types. This attributes are also the same for both training and

Table 34: Agent attributes

NAME	TYPE	VALUE RANGE	DESCRIPTION
`CurrentGame`	TGame/Game	-	Reference to the instance of the game that is being played. It is used to access information in the observation phase.
`playerID`	int	-	ID of the player that uses this AI instance. It is used to access information in the observation phase.
`effector`	Teffector / Effector	-	Instance of the `Effector/TEffector` that will execute the actions commanded by the AI.

Table 35: Agent methods

NAME	ARGUMENTS	RETURN	DESCRIPTION
`Decide`	-	-	The method of the `AI` interface. Because the model output depends on MLA cycle, it simply returns `None`.
`InitializaeAgent`	-	-	One of the functions of the MLA cycle. It executes once, at the beginning of the scene, and it is used to perform the necessary tasks to initialize an agent. It is used to initialize the current game reference mentioned before. Because the game will only play one time in normal mode or reset itself continuously in training mode, the reference does not need to be updated and thus, it is reasonable to use this function.
`CollectObservations`	-	-	This function is executed once per observation-action-reward cycle. It is used to send information to the model that will be used by it to return an action. More details about the information that is provided in NOMBRE SECCION.
`AgentAction`	`Float[], string`	-	This function is executed once per observation-action-reward cycle. It receives an action that the model has chosen as the best and it is the place to set the reward of the agent for the current cycle.

normal versions. The only change is their type, that corresponds to the version it is being played.
Again, functions that belong to the observation-action-reward cycle are executed externally, so the flow of the game has to adapt to them. The agent cycle is explained in greater detail in the section Annex II – Agent Cycle. An agent cycle is also called step.

A2.4 Machine Learning Parameters -PPO

Machine learning process has also some parameters that can be adjusted to obtain efficient and well-trained models. The parameters presented in this section belong to the PPO (Proximal Policy Optimization)[xxxvi] algorithm, the suggested by MLA and the one that adapts better to our scenario.

Without entering in much detail, PPO is an algorithm designed to back-propagate the results of stacks of actions in order to train the model reducing the importance of stack size and without a great increase in complexity. It is a ploicy-based algorithm, which means that it gathers data to build a state-action map and then uses the map to update the model values.

The parameters that can be modified using this algorithm and the values we consider the best in our case are the following:

- Gamma: Balance between immediate (lower values) and future rewards (higher values). Value: 0.99.
- Lambda: Balance between the reliance on the estimated value of the agent rewards and the actual value received in the environment. Value: 0.95.
- Buffer size: Number of steps to do before back-propagation. Value: 20480.
- Batch Size: Number of steps used in a back-propagation iteration (gradient descent). Value: 4096.
- Number of epochs: Number of passes through the buffer during gradient descents. Value: 4.
- Learning rate: Strength with which the model is updated during back-propagation. Value: 3.0e-4
- Time horizon: Number of steps to collect before being added to the buffer. If the episode (in our case, the game) finishes before reaching the number, an estimate is used until the number is fulfilled. Value: 64.
- Max steps: Amount of steps per session. Value: Between 5e5 and 1e6.
- Beta: Entropy regularization. Higher values imply more random decisions and exploration of the action space. Value: 1e-2.
- Epsilon: Acceptable threshold of divergence between old and new policies (groups of steps). Value: 0.2.
- Number of layers: Hidden layers present in the neural network (see State of the art). Value: 2.
- Hidden units: Nodes per hidden layer (see State of the art). Value: 128.

Keep in mind the provided values might have varied slightly during training session, but the provided ones provided the best results.

A2.5 Agent Cycle

In this section the functioning of the agent and how the game needs to be adapted to the MLA plugin is explained. First, we wish to remember that MLA functions follow its own cycle and thus, can't be called when needed. This makes changes in the execution flow necessary, specially in training mode. Both, this mode and normal mode will be covered separately.

Normal Mode

In the normal mode the only major change needed is that the Agent has to execute the action that the model outputs by itself, in the `AgentAction` function. That is why a reference to the player `Effector` is needed. During normal games, the agent uses the "Decision on demand" setting. This allows for decisions to be taken when needed, instead of every "x" number of MLA cycles.

This does not mean that the MLA functions are called directly, it means that the cycle is started when needed, but functions are still called when the plugin is ready to do so.

This cycle is:

1. The game is being played in normal mode.
2. An AI tick is issued.
3. Decide method of the agent is called.
   3.1. A decision to is requested to the MLA plugin.
   3.2. `Actions.None` is retuned, and thus, no action is executed.
4. The game continues.
5. As part of the MLA cycle, `CollectObservations` is called.
   5.1. The method provides the model with the necessary information.
6. The game continues
7. When the MLA cycle has finished, `AgentAction` is called with the output of the model.
   7.1. The provided action is executed.
8. The game continues until the next AI tick, when step 2 will happen again.

A diagram of this cycle can be found in Figure 31.

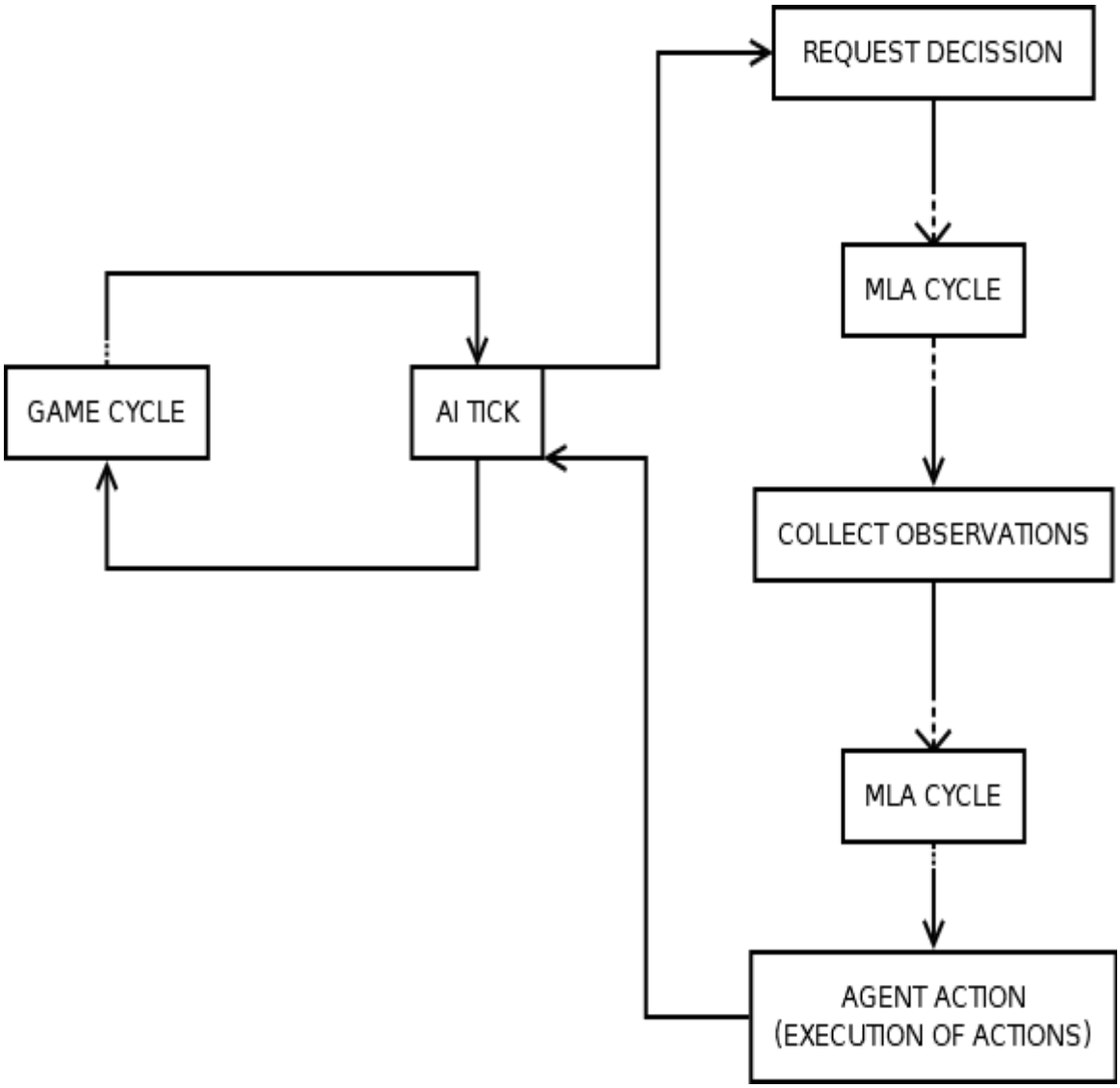


Figure 31: Diagram of the agent cycle during a normal game.

Training Mode

In training mode, the plugin takes full control of the game execution flow. This grants maximum performance and faster training. However, that implies that the training cycle has to be adapted to use the plugin flow as "engine", that is, to advance following the plugin cycle.

To do so, the only real change is that the simulations have to be advanced when a decision has been retrieved by the model. Instead of doing it continuously, when the `AgentAction` function is called every player executes an action and the simulation is advanced until the following AI tick. The reward is computed after, based on the differences in the state of the game before executing the actions and after advancing the simulation.

Step by step:

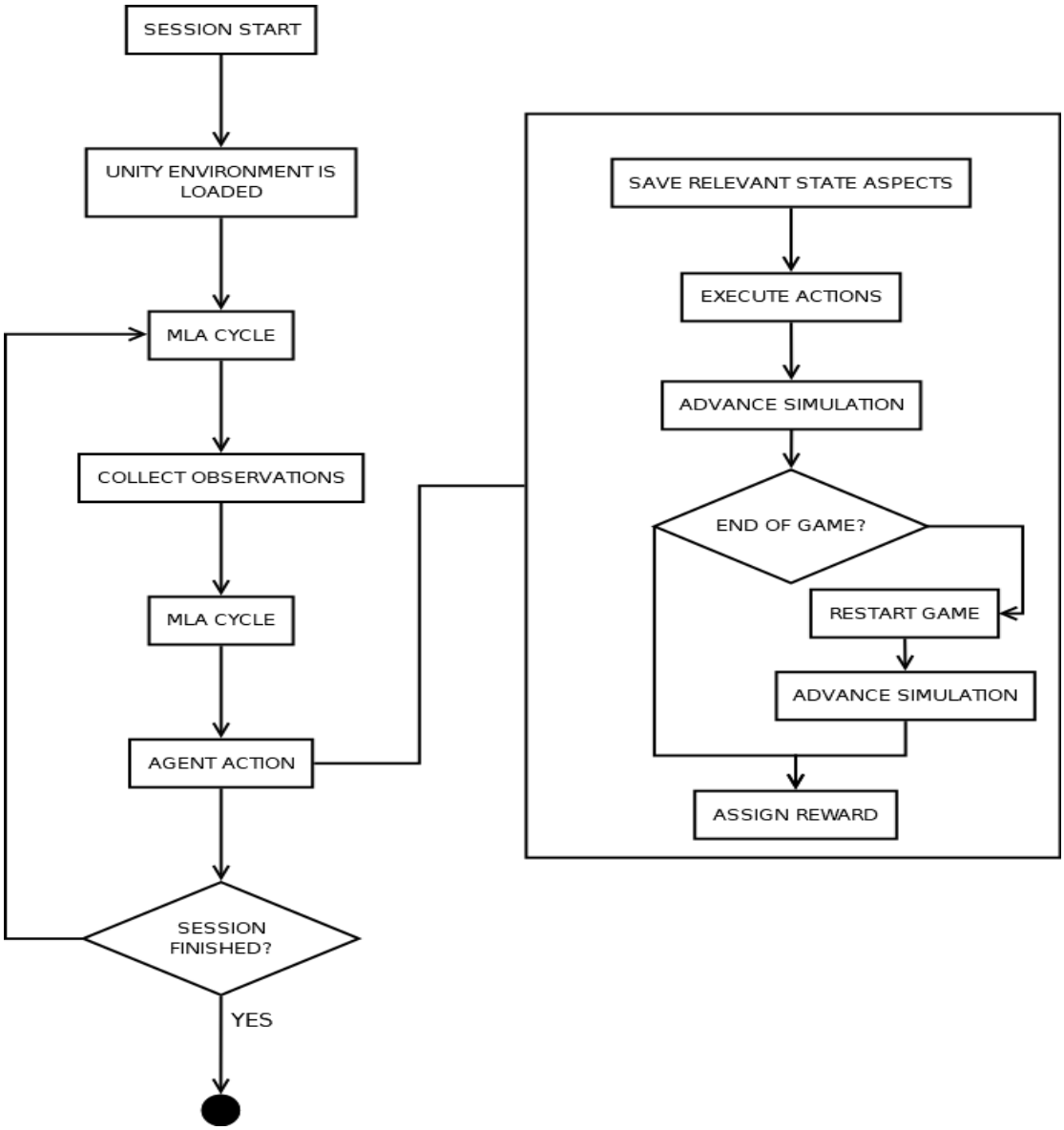


Figure 32: MLA training game cycle.

1. Training is initialized (plugin unity-independent execution)
2. MLA starts Unity environment training version is initialized.
3. The first turns until the first AI tick are simulated.
4. MLA cycle starts.
5. As a part of the MLA cycle, `CollectObservations` is called.
6. MLA cycle continues.
7. `AgentAction` is called.
   7.1. Relevant variables for reward computation are stored (see Rewards).

7.2. Every player executes an action (the player with the machine learning AI executes the action provided by the plugin).

7.3. The simulation is advanced until the next AI tick.

7.4. If some player won:

7.4.1. The game is restarted.

7.4.2. The first turns until the first AI ticks are simulated.

7.5. Based on the changes between the state of the game in point 7.1 and its current state, reward is computed and passed to the model (see Rewards).

8. If the training is not finished, return to step 4.

Training will finish when the amount of steps provided is reached, This value is ste before starting the session in the plugin files.

A2.6 Model Input

To give proper output and learn, the model need to perceive the world somehow. This is were input enters the equation. It will determine what the agent is able (or not) to learn. If the provided input is too much or not appropriate, noise will be generated and the model might not learn properly. Not enough input and the model won't be able to learn.

As explained before, input is provided in the `CollectObservations` function, from the MLA cycle. Across training sessions, the state parameters provided to the model have changed (see Training Sessions & Models) to look for the combination that allowed for better learning. Here we expose, one by one, the factor that have been used at least once.

It has to be kept in mind that these factors have to be level-independent. We want to train models able to play in very different scenarios, so we can't evaluate factors that depend on a specific level (like the total number of planets) to feed the model.

- Unit generation ratio: The amount of units generated by all the planets belonging to the player with the machine learning AI per turn.
- Enemy unit generation ratio: Same as the last one, but with the combined ratios of the enemies (mean) or with the ratio of the strongest one.
- Neutral planets: Number of planets that don't belong to any player.
- Incoming attacks: If there is at least one attack headed towards a planet conquered by the player with the machine learning AI.

- Number of planets: Number of planets that belong to the machine learning controlled player.
- Number of enemy players: Number of planets that belong to other players.
- Can Heal: If there is at least one planet without full health.
- Can level up: If there is at least one planet below its maximum level.
- Units number: Total number of units of the machine learning player.

A2.7 Rewards

A reward is a number that tells how the agent is doing in the game that it is playing. They can be positive if the agent is doing the correct things at the correct time or negative if not. Rewards are given by the designers, that is, it is programmed when the rewards have to be issued (based on the game state) and its amount.

This is perhaps the most important part of the training because it determines what will the agent learn through training. For example, let's suppose that a trainer wants to teach a dog to jump over barriers. To do so, it uses a ball that throws over the barrier. The dog learns to go after the ball and jump the obstacle with enough tries. However, the dog has not learned to jump obstacles, it has learned to go after a ball. If the dog is in front of an obstacle but the ball is thrown in the opposite direction, it will always go after the ball, ignoring its real objective, the obstacle.

In our case the same happens. Machine learning techniques find models that give them the highest score possible, but they don't really "know" what is happening. A model simply outputs a series of numbers that, based on previous experiences, translate to a high reward (either immediate or not) given certain input, without taking into account what is being displayed in the screen.

That means that if rewards are not correctly given, models might learn how to obtain the highest rewards but not how to, in our particular case, play the game. This can also happen on the contrary. If the reward is given correctly and the model needs to "learn" to play the game, it might find design exploits that allow it to win always, suppressing the fun factor.

Reward Factors

The following are the factors that have been used at some point to compute the final reward for the model. Please keep in mind that in the different training sessions explained in Training Sessions & Models the number and importance of the factors that were used varies between them.

- Unavailable Actions: If the model outputs an action that can't be executed, like healing when there are no wounded planets that belong to the player, a negative reward is issued.
- Victory/Defeat: The most important (in terms of weight) reward. If the player with the machine-learning AI wins, this reward is issued positively. If not, it is issued negatively.
- Difference in generation ratio: If the generation ratio has increased or not comparing it to the one obtained in `CollectObservations`. If it has increased, a positive reward is given. If it has decreased, a negative one is issued. This ratio can be modified because:
    - A planet has been conquered
    - A planet has been upgraded

  Depending on which of the two has happened, the reward varies (see Training Sessions & Models)
- Difference with enemy generation ratio: If the enemy ratio is higher or equal, a negative reward is issued. If not, a positive or no reward is given.
- Wounded planets penalization: If there is at least one planet without full health, a negative reward is issued.

A2.8 Training Sessions and Models

Because training a model is a rather empirical process in which trial and error is required, different combinations of the previously mentioned factors have been used across several sessions. Each session generates a model that is tested to measure its quality.

The training scene can be seen in Figure 33. There are three simultaneous players controlled by the random AI, so the normal percentages of victory for each is 33%. The quality of a model is measured by how much that percentage increases. In some sessions, one of the enemies uses the dumb AI (which does nothing), so the normal percent is 50%.

These are the training sessions and the combination of input and reward factors that have been used. We will refer to the player controlled by the model as "model".

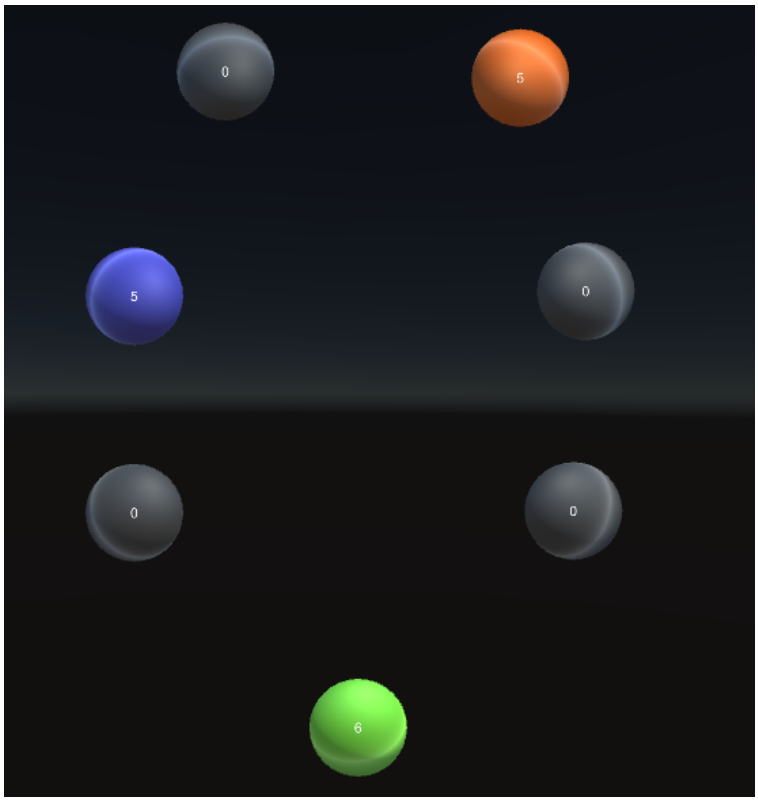


Figure 33: Training session environment

1. All the factors are used, for rewards and input. Victory percentages fell below the 20%. Model only started to "play" after another player unit generation ratio increased.
2. Ratio difference reward was changed to punish equal ratios. Conquering/losing planets factor was removed from reward computing to reduce noise. Model started playing before but, again, the victory ratio was still near random values. Main problem now was the continuous usage of unavailable actions.
3. Penalty for unavailable actions was increased. The problem, however, continued.
4. Penalty was increased again. The problem continued.
5. Game flow was modified to stop simulating until a valid action was suggested, increasing the penalty for this type of incorrect behaviour. One enemy was switched to the dumb AI.
6. Unavailable actions were reduced drastically, but Model was still not playing good enough. Again, the ratio of victories was only slightly superior to normal random values. Main problem now was the attacks to other players in early phases of the game.
7. Training time was increased with no appreciable results.
8. Changes in flow made in session 5 were undone. All rewards were reduced except victory/defeat. Conquering and losing planets computed for the reward again. This session obtained the worst model, with a victory ratio of below 18%.

9. Conquering / losing planets reward suppressed again. No major changes.
10. Training was simplified removing all reward and input factors except unit generation ratio (only Model one), conquering planets and victory. Good results, slightly above random.
11. Ratio changes reward updated to give more weight to upgrades. Slight improvement.
12. Input is changed to only neutral planets, can level up and generation ratio. Reward for upgrading was increased. Again, slightly better.

ANNEX III

PROJECT DEVELOPMENT TABLE

Table 36: Project development process per weeks

WEEK	TASK ID	TASKS	HOURS	OBSERVATIONS	DIFFICULTIES	TOTAL HOURS
29/01 - 04/02	**DOCU**	TFG Start (redaction of technical proposal)	5	Decision of TFG theme.	-	5
05/02 - 11/02	**G01**	Lux Aura creation - Event System -Basic control and combat system - Graphics	20	Creation of the base game.	-	25
	DOCU	Design Document Redaction	3	-	-	28
12/02 - 18/02	**G01**	Lux Aura AI and polishing	20	To be able to work with threading, switching to Unity 2018 is needed	Threading in Unity 2018 is not quite ready, switching back to 2017.	48
	DOCU	Design Document Redaction	3	-	-	51
19/02 - 25/02	**MT01 / G02**	Montecarlo Tree Search documentation Training game creation	7	This version will be used to simulate games in Montecarlo and to train the neural network	-	58
	DOCU	Design Document Redaction	3		-	61
26/02 - 04/03	**G02**	Training game	16	Creation of the training version of	-	77

		creation & debug		the game and initial debugging		
5/03-11/03	**MT02**	Montecarlo tree	20	Creation of the classes necessary to support the Montecarlo execution	-	97
12/03 - 18/03	**MT02**	Montecarlo testing	21	Inclusion of Montecarlo Tree search in the game and testing	-	118
12/03 - 25/03	**MT02**	Montecarlo testing	7	Testing and bug fixing	-	125
	DOCU	Documenting game and training verison	14		-	139
26/03-01/04	**DL01**	Research of neural networks in Unity	21	Searching information and setting a basic environment	Had to use older version of the environment to make it work	160
02/04 - 08/04	**DL02**	Create basic project using ml-agents in Unity	15		-	175
		update ml-agents to last version	7		-	182
09/04 - 15/04	**DL02**	Adapting game and training structure to use machine learning	21	This also includes investigation on the different ways of using ml-agents	-	203
16/04 - 22/04	**DL02**	””	21	Creation of training strategy and reward assignation	-	224

23/04 - 29/04	**DL02**	Adapting the agent to play a normal game	5	-	-	229
	DL02	Training the agent	16	-	Having some problems, it doesn't quite learn	245
30/04 - 06/04	**DL02**	””	12	-	””	257
	DOCU	Writing the main document	9	-	-	266
7-04 - 13/04	**DOCU**	""	21	-	-	287
14/04 - 20/04	**DOCU**	""	21	Memory ready for correction	-	308

REFERENCES

i Wikipedia – Machine Learning https://en.wikipedia.org/wiki/Artificial_neural_network [Last visited 10/04 2018]

ii Int8 - Montecarlo Tree Search. https://int8.io/monte-carlo-tree-search-beginners-guide/ [Last visited 01/06/2018]

iii Deep Mind – Deep Reinforcement Learning. https://deepmind.com/blog/deep-reinforcement-learning/ [Last Visited 19/10/2018]

iv Walktrough of the game Ratchet & Clank. Youtube - https://www.youtube.com/watch?v=bso_HL0lhqQ [Last visited 01/06/2018]

v Google Play – Auralux https://play.google.com/store/apps/details?id=com.wardrumstudios.auralux [Last visited 11/05/2018]

vi Unity Engine web page. https://unity3d.com/es/. [Last visited 10/03/2018]

vii *Tensorboard* framework web page. https://www.tensorflow.org/programmers_guide/summaries_and_tensorboard [Last visited 05/04/2018]

viii Tensorflow framework web page. https://www.tensorflow.org/ [Last visited 05/04/2018]

ix Ml-agents Unity plugin. https://github.com/Unity-Technologies/ml-agents. [Las access 17/05/2018]

x Python web page. [Last visited 20/05/2018]

xi Unity MonoBehaviour. https://docs.unity3d.com/ScriptReference/MonoBehaviour.html [Last visited 11/05/2018]

xii Unity execution order documentation. https://docs.unity3d.com/Manual/ExecutionOrder.html [Last visited 5/03/2018]

xiii Polymorphism in C# (.NET C# programming guide). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism [Last visited 11/05/2018]

xiv .NET Framework Timer class documentation (NET API) from Microsoft. https://msdn.microsoft.com/en-us/library/system.timers.timer(v=vs.110).aspx [Last visited 28/03/2018]

xv Structures in C# (.NET C# programming guide). https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/structs

xvi GameObjects Unity documentation. https://docs.unity3d.com/ScriptReference/GameObject.html [Last visited 11/05/2018]

xvii Wikipedia – Pool Pattern. https://en.wikipedia.org/wiki/Object_pool_pattern [Last visited 11/05/2018]

xviii Wikipedia – Cohesion concept. https://en.wikipedia.org/wiki/Cohesion_(computer_science)

xix Unity Manual – Coroutines. https://docs.unity3d.com/Manual/Coroutines.html [Last visited 11/05/2018]

xx Interfaces in C# (.NET C# language references). https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/interface

xxi Enumerators in C# (.NET C# programming guide). https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/enum [Last visited 11/05/2018]

xxii Unity Technologies web page. https://unity3d.com. [Last access 10/05/2018]

xxiii Total War II webpage. https://www.totalwar.com/total_war_rome_ii_emperor_edition [Last visited 11/05/2018]

xxiv Google Deepmind - AlphaGo project. https://deepmind.com/research/alphago/ [Last visited 11/05/2018]

xxv Wikipedia – Deep Learning https://en.wikipedia.org/wiki/Deep_learning [Last visited 11/05/2018]

REFERENCES

xxviThreading in C# (.NET C# programming guide). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/ [Last visited 11/05/2018]

xxviiThread synchronization in C# (.NET C# guide). https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/threading/thread-synchronization [Last visited 11/05/2018]

xxviiiMutex in C# (.NET guide). https://docs.microsoft.com/en-us/dotnet/standard/threading/mutexes [Last visited 11/05/2018]

xxix Wikipedia – Cache. https://en.wikipedia.org/wiki/Cache_(computing) [Last visited 11/05/2018]

xxx Wikipedia – Spatial & temporal locality. https://en.wikipedia.org/wiki/Locality_of_reference#Types_of_locality [Last visited [11/05/2018]

xxxiWikipedia - Arthur Samuel. https://en.wikipedia.org/wiki/Arthur_Samuel [Last access 10/05/2018]

xxxii Tesla autopilot. http://fortune.com/2015/10/16/how-tesla-autopilot-learns/ [Last access 10/05/2018]

xxxiii Wikipedia – Amazon Echo (virtual assistant). https://en.wikipedia.org/wiki/Amazon_Echo [Last access 10/05/2018]

xxxiv Wikipedia – Reinforcement learning.https://en.wikipedia.org/wiki/Reinforcement_learning [Last access 10/05/2018]

xxxvApache License. https://www.apache.org/licenses/ [Last access 10/05/2018]

xxxvi PPO in OpenAI (https://blog.openai.com/openai-baselines-ppo/) [Last Access 17/05/2018]