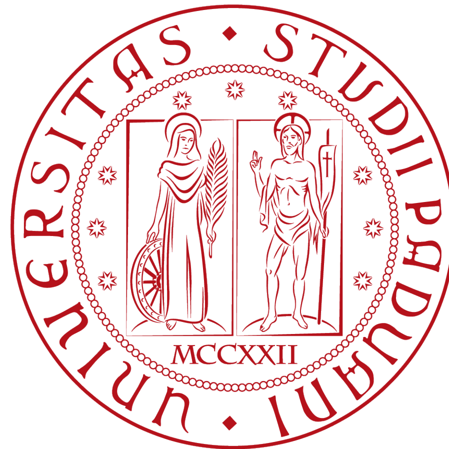
University of Padua

Department of Mathematics "Tullio Levi-Civita"

First Cycle Degree in Computer Science



A Reinforcement Learning Approach to Reforestation through Robots

Bachelor Thesis

Supervisor
Prof. Francesco Ranzato

Candidate
Francesco Ceccato
Identification Number: 2001639

Academic Year 2022-2023

Francesco Ceccato: *A Reinforcement Learning Approach to Reforestation through Robots,* Bachelor Thesis, © November 2023.

Abstract

Carbon mitigation effort is nowadays a pressing global challenge. While some advancements are achieved on mitigation techniques, the fact that deforestation hinders the capacity of the Earth to absorb the carbon dioxide generated by human endeavour needs to be addressed. A software simulation of a desert environment created through the Unity game engine is hereby documented. In this simulation, a grid of photovoltaic panels powers a device that produces water through dehumidification of the air. The water is consequently stored inside a reservoir and transferred to a robotic vehicle with the task of moving and watering trees that are planted within a given area. By means of Reinforcement Learning techniques, the robot will have to be trained to water the trees in the area, assuring their daily growth while minimizing the waste of water and assessing the local weather conditions in order to make decisions. This study does not concern the training itself and its results: nevertheless, some indications on how to perform the training are provided.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to Prof. Francesco Ranzato, as the thesis supervisor, for the invaluable help and guidance he provided during the drafting process.

I would like to thank Monica and Vincenzo and my aunt and uncle Giovanna (better known as "Giannina") and Renato for their unconditional support. Major thanks also to all my friends who accompanied me through this journey and in whose empathy I could find solace even in the toughest of times - in no particular order: Lara, Matteo, Giovanni D., Elisa, Agata, Michael, Marko, Giacomo, Alessandro, Giovanni T., Luca, Alessandro Cesare, Davide, Samir, Alberto, Sara (the list is not exhaustive, of course).

I would also like to offer a special word of thanks to Anna and Veronica as well as to my dear friends Chiara S. and Lorenzo S., whom I met not long before I enrolled in the University of Padua, and which I consider pillars of the community I belong to.

Padua, November 2023 Francesco Ceccato

Contents

1 Introduction and Summary 1
- 1.1 About the Contractor Company . . . 1
- 1.2 Document Structure . . . 1

2 General Discussion on the Approach 3

3 Characteristics of the Environment 5

4 Specifications of the Simulation 6
- 4.1 Geography of the Environment . . . 6
- 4.2 Tree Behavior and Growth . . . 6
- 4.3 Tree Health System . . . 8
- 4.4 Procedural Tree Generation . . . 8
- 4.5 Rover and Additional Scene Elements . . . 9
  - 4.5.1 Rover . . . 9
  - 4.5.2 Dehumidifier . . . 9
  - 4.5.3 Dispenser . . . 10
  - 4.5.4 Docking Station and Water Pump . . . 10
- 4.6 Weather Engine . . . 11
- 4.7 Gaussian Sampling . . . 12

5 Unity and C# Scripts Overview 13

6 Structure and Purpose of the Codebase 15
- 6.1 Overview . . . 15
- 6.2 UML Class Diagram . . . 15
- 6.3 Insights on the Modules . . . 17
  - 6.3.1 PV Energy and Weather Statistics Retrieval . . . 17
  - 6.3.2 Continuous Storage . . . 17
  - 6.3.3 Gaussian Sampling . . . 18
- 6.4 Design Patterns Employed . . . 18

7 Visual Rendering of the Scene 20

8 The Rover as a ML-Agent in Unity 24
- 8.1 Overview of Reinforcement Learning . . . 24
  - 8.1.1 Training and Inference . . . 25
- 8.2 The Unity Implementation: ML-Agents Toolkit . . . 25
  - 8.2.1 PyTorch . . . 25

8.2.2 Steps and Episodes 26
8.2.3 Requirements of an Agent Script 26
8.3 The Rover 28
8.3.1 Actions 29
8.3.2 Observations 29
8.3.3 Assumptions on the Rewards 31

9 Training Expectations **32**

10 Conclusions **34**
10.1 Final Remarks 34
10.2 Final Balance of the Internship 34
10.3 Expectations from the Internship: The Author's Perspective 35

Acronyms **36**

Glossary **37**

Bibliography **38**

List of Figures

4.1 A 3D render of the ESA ExoMy rover, upon which the rover in the simulation is based. . . . 10

5.1 The order of execution of the methods defined in the MonoBehavior class. Source: https://docs.unity3d.com/Manual/ExecutionOrder.html . 14

6.1 The UML class diagram for the software. . . . 16
6.2 The dependency between classes `AbsorptionArea` and `WeatherEngine`. 16

7.1 The rover, docking station (the yellow circular platform), dispenser (the light blue cube), and water pump (the dark blue parallelepiped). . . . 20
7.2 The Desert stage of the simulation. Only a few trees are present. . . . 21
7.3 The Mixed stage of the simulation. The amount of trees is noticeably higher than in the previous stage. . . . 21
7.4 The Forest stage of the simulation. The area is mostly covered in trees. 22
7.5 Screenshot of the game view, including the labels in the GUI. The area is viewed from the rover's perspective. In the corners a number of labels are visible: the current speed of the rover, the local time and number of elapsed days since the start, the amount of energy and water stored both in the rover and dispenser. . . . 22
7.6 A close-up view of a tree. The absorption area of the tree is delimited by a blue circle (though the area is implemented as a spherical collider). 23

List of Tables

4.1 This table shows the connection between stages and unique probability ranges. . . . 9

4.2 This table shows the parameters sent in a request to the PVGIS service. 11
4.3 This table shows the data to extract from the PVGIS API response. . 11
4.4 This table shows the parameters sent in the request to the Open-Meteo service. 12
4.5 This table shows the data to extract from the Open-Meteo response. . 12

8.1 The actions that the rover can perform as an Agent in the context of RL. 29
8.2 The "one-of-a-kind" observations collected during the training. 30
8.3 The observations collected during the training for each tree in the scene, to be stored in a variable-length vector. 31

Chapter 1

Introduction and Summary

Example of a reference to a term in the glossary:
Application Program Interface (API).

Example of an URL to a web resource:
`http://lightningmaps.org`.

Example of a citation:
citation [1]

1.1 About the Contractor Company

This study was conducted during a three-month internship at Piratech s.r.l.
Piratech is an IT company based in Padua, Italy. It offers a wide array of consulting services to businesses with the purpose of aiding digital transformation, including UX design discovery for digital products and delivery of full-fledged software solutions. It leverages an in-house team of developers, engineers and designers in order to create novel projects, and co-operates with academic institutions and founders to turn research projects into marketable goods.

1.2 Document Structure

Chapter 2 describes the foundations of theoretical approach for this study.

Chapter 3 covers the characteristics of the environment to simulate.

Chapter 4 articulates the formal specifications of the simulation.

Chapter 5 gives an overview of Unity and the C# programming language.

Chapter 6 provides insight into the codebase that makes up the simulation.

Chapter 7 showcases captioned images of parts of the simulation renders.

Chapter 8 describes Reinforcement Learning and the concept of ML-Agents.

Chapter 9 provides hints and expectations apropos of the training process.

Chapter 10 is devoted to the conclusions.

With respect to the text styling, the following typographic conventions have been adopted throughout the present document:

- Acronyms and shortened forms are defined in the glossary, which can be found at the end of this document.

Chapter 2

General Discussion on the Approach

In this chapter, the foundations and key concepts of the theoretical approach which motivated the pursuit of this research are explained.

This research arises from the desire to foster the growth and thriving of trees and shrubs in environments that are considered "hostile" due to the aridity (i.e. limited or low water content) of the soil, which is assumed to hinder the survivability of plants: a desert is a common example of such an environment.
Because salt water is not suitable for watering trees and shrubs (as the increase in salinity of the soil will hamper their growth [2]), whenever a source of fresh water located in proximity of the environment (such as a lake or river) is unavailable, one has to choose between desalination and dehumidification – two consolidated approaches - in order to produce fresh water. However, the provision of desalinated water can demand a considerable organizational and logistical effort if the distance between the area and sea is vast.
In equatorial deserts, the intense heat of the sunlight in the daytime can be exploited through a grid of photovoltaic (PV) panels which, when facing towards the sun, should provide energy to be subsequently stored in an appropriately sized battery, located not too far from the panel grid. The energy will be used to power one or more dehumidification systems: during nighttime, once the dew point (the temperature at which airborne water vapor will condense to form liquid water) is reached, the dehumidifiers will generate fresh water to be stored in a reservoir (plausibly placed alongside the battery inside the same physical machinery).
It is obvious that the demand for energy coming from the dehumidification systems imposes an adequate planning of the nominal power and efficiency requirements that have to be met by the PV panels grid. What is missing so far is a system to supply water regularly to each tree and shrub in the environment, once that they have been planted. The proposed solution is a self-driving rover on wheels, equipped with a battery and a water tank. The rover is assigned the task to move towards each tree and dump water on the ground as soon as its distance from the tree is adequately close, so that the roots of the tree will be able to absorb the water. The rover should detect when both battery charge and water levels are low and decide accordingly to move towards a fixed location where either the battery will be recharged, or the tank will be

reloaded. The energy needed to recharge the rover's battery shall be taken from the dispenser's battery: this means that the dispenser plays at least two roles, in terms of providing energy.
This process must be continuous and require no human intervention whatsoever, with the exception of scheduled maintenance operations. Local weather conditions in a given moment should also be taken into account by the rover's decision-making process. As described in the following chapters, a technique known as Reinforcement Learning will be employed to discover what the best course of action is in order to accomplish the aforementioned tasks.

Chapter 3

Characteristics of the Environment

In this chapter, a number of characteristics that should be possessed by an environment that is meant to be simulated through this software are articulated.

The point of the "digital twin" which is the object of this study, far from being a mere "game", is to inquire on the feasibility of the approach explained in Chapter 2 when applied to real geographic locations on Earth. Countries that possess the characteristics used as reference point - dry, hot, desertic terrain covering a wide surface - are those which surround the Persian Gulf and the Gulf of Oman: Oman, Qatar, the United Arab Emirates, Bahrain, and others.
In these areas relative humidity of air steadily increases in nighttime as the temperature goes down: this allows for a large-scale production of water through dehumidification, a technique which cannot be considered novel as it was already proposed by Khalil [3](in the specific context of United Arab Emirates), Fathieh and others. [4] The very high temperature reached in those places during the day, even up to 100° F (38° C), makes for very harsh working conditions that a human could hardly endure for prolonged amounts of time while moving around and pouring water on the ground (given the additional burden of the water that must be carried) without severe health effects. In game, only a small-scale scene can be actually explored. At the time of writing the contractor Piratech s.r.l. intends to purchase a small lot in the municipality of Nardò, Italy, in the region of Puglia, to test the system and gather additional data which may prove useful. Nardò is one of the Italian municipalities with the scarcest yearly precipitation (slightly over 500 mm of rain per year on average) and a climate that roughly approximates that of countries with lower latitudes, at least if compared with other municipalities of Italy.
Some could argue that in the long term, desalination of salt water is a more efficient approach in terms of energy consumption: while this may be proven true, the infrastructure and maintenance cost for the required system is such that financial expenses could jeopardise the attempts to put such a system in order, so the dehumidification approach was ultimately chosen.

Chapter 4

Specifications of the Simulation

In this chapter, the formal specifications of the simulation (including mathematical formulas) are articulated.

4.1 Geography of the Environment

This simulation takes place in a desertic square lot, with the square side being 100 m wide: the total surface is thus equal to 100^2 m^2 = 10.000 m^2. The lot is bounded by invisible walls, which make it impossible for any non-stationary entity to leave the area. The altitude of the terrain ranges between 0 and 20 m. Because sand dunes are assumed not to be significantly steep, in no point of the terrain is the slope expected to exceed 30°. A variable amount of trees and saplings are spread over the square surface. In this simulation, the tree species is not taken into consideration: this happens because the specialization of the trees' behavior according to their species is outside the scope of this work and assumed to be developed during future advancements. The displacement of the trees is described in Section 4.4.

4.2 Tree Behavior and Growth

As mentioned in Chapter 2, in order for the rover to behave as intended trees need to be planted beforehand.
Signed integer variables are used to store the values of parameters that only accept natural numbers, while single precision (32-bit) floating point variables are used otherwise.
A tree is characterized by a number of parameters (*the SI units of measurement for non-dimensionless numbers are enclosed between square brackets from here onwards, in all the sections that ensue*):

- **Age**: An integer number which represents the age of a tree, or rather, the amount of days elapsed since it has been planted.
- **Initial Need** [l]: The amount of water a tree needs to be provided during its first day of life (from day 0 to day 1) in order to survive. In the beginning of the new day, a calculation is performed to determine whether the demand was met or not – the same applies to each day of the tree's lifespan.

- **Final Growth Age**: The integer number of days that must pass in order for the tree to reach its final water demand (**Final Need**).
- **Final Need** $[l]$: The amount of water a tree needs to be provided on a daily basis upon reaching its Final Growth Age, for all the following days.
- **Daily Need Function**: A piecewise-defined function which has as its first sub-function a line (polynomial of degree 1) in the form $y = mx + q$, that gives the amount of water $[l]$ the tree needs to be provided during the day x of its lifespan in order to survive.
  This is a very simple function that will eventually be replaced with a more complex one in future advancements. The angular coefficient m is given by the ratio $\frac{FN-IN}{FGA}$, where FN is the Final Need, IN is the Initial Need and FGA is the Final Growth Age, while the intercept q is equal to IN. The function is
  $$N(x) = \begin{cases} \frac{FN-IN}{FGA}x + IN & \text{if } x \leq FGA \\ FN & \text{otherwise.} \end{cases} \tag{4.1}$$
- **Daily Bound**: A function used to determine both the **daily lower bound and upper bound** $[l]$ of an interval centered in the value x returned by the Daily Need Function. If the provision of water in a given day falls within the interval, then there will be no Health Points loss. (See the definition below.)
  An example of such function is
  $$B(x) = \left(1 \pm \frac{30}{100}\right) x \tag{4.2}$$
- **Health Points**: A numeric integer value that indicates the health of the tree. If the health points amount falls below 0 as a consequence of a daily update, then the plant is dead and cannot be revived. A tree has at most 100 Health Points (which is also the initial amount). The technique used to compute the Health Points loss or gain is described in Section 4.3.

In the simulation, the tree is endowed with a circular area at the base of the trunk called **Absorption Area**. The Absorption Area is meant to simulate, in a very simple fashion, the area where the roots can be found beneath the ground. This means that all water that is poured on the ground within the perimeter of said area will be absorbed by the roots and supplied to the whole tree.
The Absorption Area has the following characteristics:

- **Initial Radius** $[m]$: The initial value of the radius of the Absorption Area during the first day of the tree's lifespan.
- **Maximum Size Age**: The integer number of days that must pass in order for the area to reach its **Maximum Radius**. It is supposed to match the tree's **Final Growth Age**.
- **Maximum Radius** $[m]$: The maximum radius that can be reached by the Absorption Area.

- **Area Size Function**: A piecewise-defined function which has as its first sub-function a line (polynomial of degree 1) in the form $y = mx + q$, that gives the radius of the Absorption Area during the day x of its lifespan.
  Similarly to the Daily Need Function, the angular coefficient m is given by the ratio $\frac{FR-IR}{MSA}$, where FR is the Final Radius, IR is the Initial Radius and MSA is the Maximum Size Age, while the intercept q is equal to IR. The function is

$$A(x) = \begin{cases} \frac{FR-IR}{MSA}x + IR & \text{if } x \leq MSA \\ FR & \text{otherwise.} \end{cases} \tag{4.3}$$

4.3 Tree Health System

A tree has an initial amount of Health Points (100) when it is first planted. If a tree was provided too much, or not enough water over the day, points will be lost as the next day starts. However, they can be regained in the following days, provided that the amount of received water (which is subject to a daily change) is correct. The daily variation in the number of Health Points is subject to a law as well.

Let P_l and P_g be the maximum amount of Health Points that can be lost or regained at the beginning of each day (The value of the two variables is not supposed to vary). Let t_i be the "target" water quantity and $B_-(t_i)$, $B_+(t_i)$ be the lower bound and upper bound calculated in Section 4.2 for day i. Then, if x is the amount of water provided during the day i of the tree's lifespan, the amount of points (which can be negative) to be added to the score at the beginning of day $i + 1$ is

$$S_i(x) = \frac{-P_g|x - t_i|}{t_i - B_-(t_i)} + P_g \tag{4.4}$$

but the obtained value is clamped in the interval $[-P_l, P_g]$, meaning that if it is smaller than $-P_l$ it will be set to $-P_l$. (The absolute maximum value of the function is equal to P_g by construction.)

Intuitively, when the provided value is perfectly equal to t_i the numerator is zero, and $S_i(x) = P_g$: the gain is maximum. However, a discrepancy from the target will cause the numerator to be a negative, non-zero number.
The roots of the function are $B_-(t_i)$ and $B_+(t_i)$: there is an obvious vertical symmetry with respect to t_i. In other words, Health Points are gained so long as the amount of provided water is not above or below the two bounds, otherwise they are lost.

4.4 Procedural Tree Generation

The generation of the trees in the 100×100m lot abides by the following rules:

1. The terrain is first divided into a 14x14 square grid. The side of each square is thus $\frac{100}{14} \approx 7.14$ m wide.
2. The user can choose one among three different Stages: **Desert**, **Mixed** and **Forest**. A unique range of probabilities (numbers within the $[0, 1]$ real interval) is linked to each stage with no overlaps, as shown in the table.
   Depending on the current Stage, a number is randomly extracted by sampling

Stage	Range
Desert	$[0, 0.05)$
Mixed	$[0.05, 0.4)$
Forest	$[0.4, 1]$

Table 4.1: This table shows the connection between stages and unique probability ranges.

from the continuous uniform distribution on the interval $[a, b)$, where a and b are the boundaries of the associated interval.

3. The number obtained in the previous step is the probability that a tree will spawn in a cell of the grid. If a tree is to successfully spawn within a cell, then its position is calculated as follows:
A random radius r in the interval $[0, 2]$ and a random angle α in the interval $[-\pi, \pi]$ are extracted by sampling from continuous uniform distributions. The polar coordinates of a point are thusly obtained: the point represents the offset of the tree from the center of the cell. Therefore, if (x_{C_i}, y_{C_i}) is the center of the cell i, the final position of the tree in Cartesian coordinates is

$$(x_{C_i} + r \cdot cos(\alpha), y_{C_i} + r \cdot sin(\alpha)) \tag{4.5}$$

4.5 Rover and Additional Scene Elements

4.5.1 Rover

The rover is a 0.45×0.35×0.35m six-wheeled, self-driving vehicle, based on the ExoMy 3D-printed rover developed by the European Space Agency [5] (More details can be found at https://esa-prl.github.io/ExoMy/).
It is equipped with a water tank and a battery, which acts as a main source of energy and allows motion. Of the three wheel axles, only the front one is a steering axle: the maximum wheel rotation angle is equal to 45° in both directions. The weight is ≈ 2.5 kg (when the water tank is empty).
The rover is characterized by four main parameters.

- **Power** $[W]$: The nominal power of the rover. It is assumed to be around **100** W.
- **Maximum Battery Charge** $[C]$: The maximum capacity of the battery. Ideally, the battery should allow the rover to stay powered on for several hours in a row.
- **Maximum Water Capacity** $[L]$: The maximum amount of water that can be contained in the water tank. It is assumed to be around **4** L.
- **Water Dumping Rate** $\left[\frac{L}{s}\right]$: The rate at which water that is contained in the tank is dumped on the ground. It is assumed to be around **0.05** $\frac{L}{s}$.

4.5.2 Dehumidifier

The Dehumidifier is a system to extract water from air. A condensate dehumidifer uses a refrigeration cycle to collect water known as condensate: as greywater, it contains fewer pathogens than other types of wastewater, and is generally safe to use

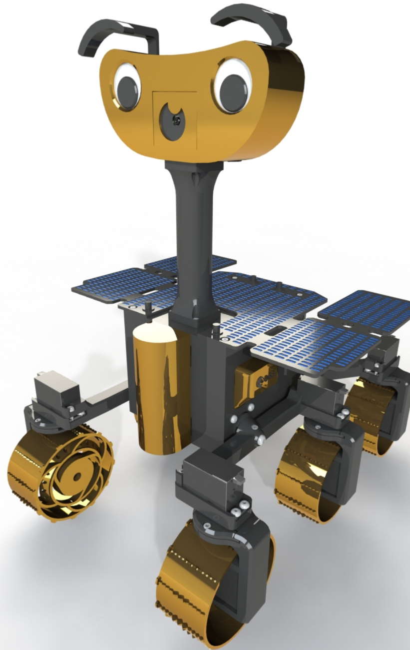


Figure 4.1: A 3D render of the ESA ExoMy rover, upon which the rover in the simulation is based.

for landscape or crop irrigation purposes. In the simulation, the related graphic asset is omitted.

4.5.3 Dispenser

The Dispenser is a machinery that hosts both a water reservoir to contain the water produced on a daily basis by the dehumidifier, and a battery to store the energy produced by the PV grid system. In the game, it is a cube whose size is not expected to match the size of the real dispenser.

4.5.4 Docking Station and Water Pump

In order to transfer energy and water from the Dispenser to the Rover, two components are needed.

A **Docking Station** is a circular platform where the battery of the Rover is recharged, as the Rover stands still on top of it. The energy is transferred from the dispenser's battery to the rover's. It plays a major role in the simulation, along with the Water Pump, because the Rover must head towards the Station as soon as a sufficiently low battery level is detected. If the Rover fails to reach the Station in time and its energy supply is depleted, it will no longer be able to move, and will have to be manually relocated and recharged through human intervention.

In a similar fashion, a **Water Pump** is a device that is used to refill the Rover's water

tank, by taking it from the water reservoir in the Dispenser.
In the simulation, it is sufficient that the Rover is nearby the Water Pump for the refilling process to begin, while refilling obviously implies additional mechanical processes in reality.

The Docking Station and Water Pump have respectively a fixed **Recharging Rate** $[W]$ and **Refilling Rate** $\left[\frac{L}{s}\right]$.

4.6 Weather Engine

Note: *In the simulation, the graphic asset for the photovoltaic panels grid is omitted.*
While some degree of arbitrariness could be tolerated for the values of some of the parameters hitherto described, there are physical quantities whose value should be closely aligned with real statistical measures for the sake of accuracy. Hence the need to devise a system to extract and refine data obtained from public Representational State Transfer (REST) API calls to services which make such measures available to the general public. For each parameter, a table of the inputs (values of the parameters for the HTTP GET request to the specific endpoint) and outputs (values that must be returned by the API call in a JavaScript Object Notation (JSON) interchange file) is provided. Note that the output values in the table do not necessarily correspond to what is directly contained in the returned JSON file: instead, they are frequently obtained via local extraction and manipulation of the contents of the file.

- **Solar Energy Generation**

  Source: Photovoltaic Geographical Information System (PVGIS) by European Commission
  Base Endpoint URL: `https://re.jrc.ec.europa.eu/api/v5_2/PVcalc`

Name	Type	Notes
Latitude	floating point	The real-life latitude of the PV grid location, in Decimal Degrees (DD).
Longitude	floating point	The real-life longitude of the PV grid location, in Decimal Degrees (DD).
Power $[kW]$	floating point	The nominal power of the PV system.
Loss	floating point	Sum of the power losses of the system (percentage of the total energy delivered to the powerline).

Table 4.2: This table shows the parameters sent in a request to the PVGIS service.

Name	Type	Notes
Daily Energy Production $[J]$	floating point	The energy produced during daytime in a single day, assuming optimal inclination and orientation angles of the PV grid.

Table 4.3: This table shows the data to extract from the PVGIS API response.

- **Precipitation Statistics**

  Source: Open-Meteo
  Base Endpoint URL: https://archive-api.open-meteo.com/v1/archive

Name	Type	Notes
Latitude	floating point	The real-life latitude of the area, in Decimal Degrees (DD).
Longitude	floating point	The real-life longitude of the area, in Decimal Degrees (DD).

Table 4.4: This table shows the parameters sent in the request to the Open-Meteo service.

Name	Type	Notes
Cumulative Precipitation $[mm]$	floating point	The cumulative amount of precipitation in one year.
Rainy Days	integer	The number of days in one year with cumulative precipitation exceeding a 2.5 mm threshold.

Table 4.5: This table shows the data to extract from the Open-Meteo response.

The parameters listed above are part of the weather engine of the simulation. They are stored in dedicated configuration files, which are hosted in secondary memory and read by the software at startup.

4.7 Gaussian Sampling

A margin of difference and uncertainty must be left to the initial daily values of the available dispenser water and energy retrieved from the API for increased flexibility. However, the values shouldn't diverge from the expected ones to a great extent. For this reason, the initial values for water and energy are respectively sampled from the random variables

$$X_1 \sim \mathcal{N}(\mu_1, \sigma_1^2) \ , \ X_2 \sim \mathcal{N}(\mu_2, \sigma_2^2) \tag{4.6}$$

where μ_1 and μ_2 are the values retrieved by the API for available daily water and energy at the beginning of the simulation, and the two standard deviations are some aptly chosen values. A reference to the implementation of the random sampling technique shall be included in Chapter 6. A new implementation was needed because Unity only exposes a library function for random sampling from a real-valued linear distribution by default.

At this point, the reader may have noticed that the actual numeric values of most parameters listed in the sections above are left blank. This is because the assignment of their values requires prior knowledge about a tree's behavior which is outside the scope of expertise of a computer scientist. With some research or additional support from a professional in the field of Forest Science, one could easily gather real values at their discretion and fit them in the simulation.

Chapter 5

Unity and C# Scripts Overview

In this chapter, a brief overview of Unity - the software used to build the simulation - and of the C# programming language is provided to the reader.

Unity is a cross-platform game engine developed by Unity Technologies, released in June 2005. According to Wikipedia, it "[...] is considered easy to use for beginner developers, and is popular for indie game development. The engine can be used to create three-dimensional (3D) and two-dimensional (2D) games, as well as interactive simulations and other experiences." [6]
The engine has been selected because of its relatively steep learning curve, and the integrated support for reinforcement learning via the ML Agents library: more information about the topic can be found in Chapter 5.
In Unity, entities are known as Game Objects, and their behavior can be controlled through scripts. A Script is a file written in C#: a general-purpose high-level programming language which, in a similar way to older languages such as C++ and Java, supports classes, inheritance, polymorphism and other concepts pertaining to object-oriented programming (although it is not the only supported programming paradigm).
Like every other C# document, a Script has a .cs file extension. A Script contains the definition of a derived class (which usually has the same name as the file) that inherits from a base class called `MonoBehavior`: that is, a class whose instances shall all be executed in a single thread (as per the Unity specifications). A Script can be attached to a Game Object: the behavior of the object is controlled by overriding a number of event functions (which are methods exposed by the base class) in each Script. The widespread use of scripts that inherit from the `MonoBehavior` class averts typical dangers that may be faced in concurrent programming, such as race condition and deadlock. [7]
At runtime, event functions are invoked in several different moments. In Figure 1, the predetermined order of execution for the event functions defined in the `MonoBehavior` class is shown.
The `Update` function is called once per frame, whereas the `FixedUpdate` function is invoked with a fixed time frequency: the `FixedUpdate` invocations are finely synchronized with the calculation steps done by the Unity physics engine (employed e.g. to determine if a collision between bodies has happened, or calculate the velocity and position of bodies whose behavior is described by equations of motion.)
Many of these functions do not require overriding.

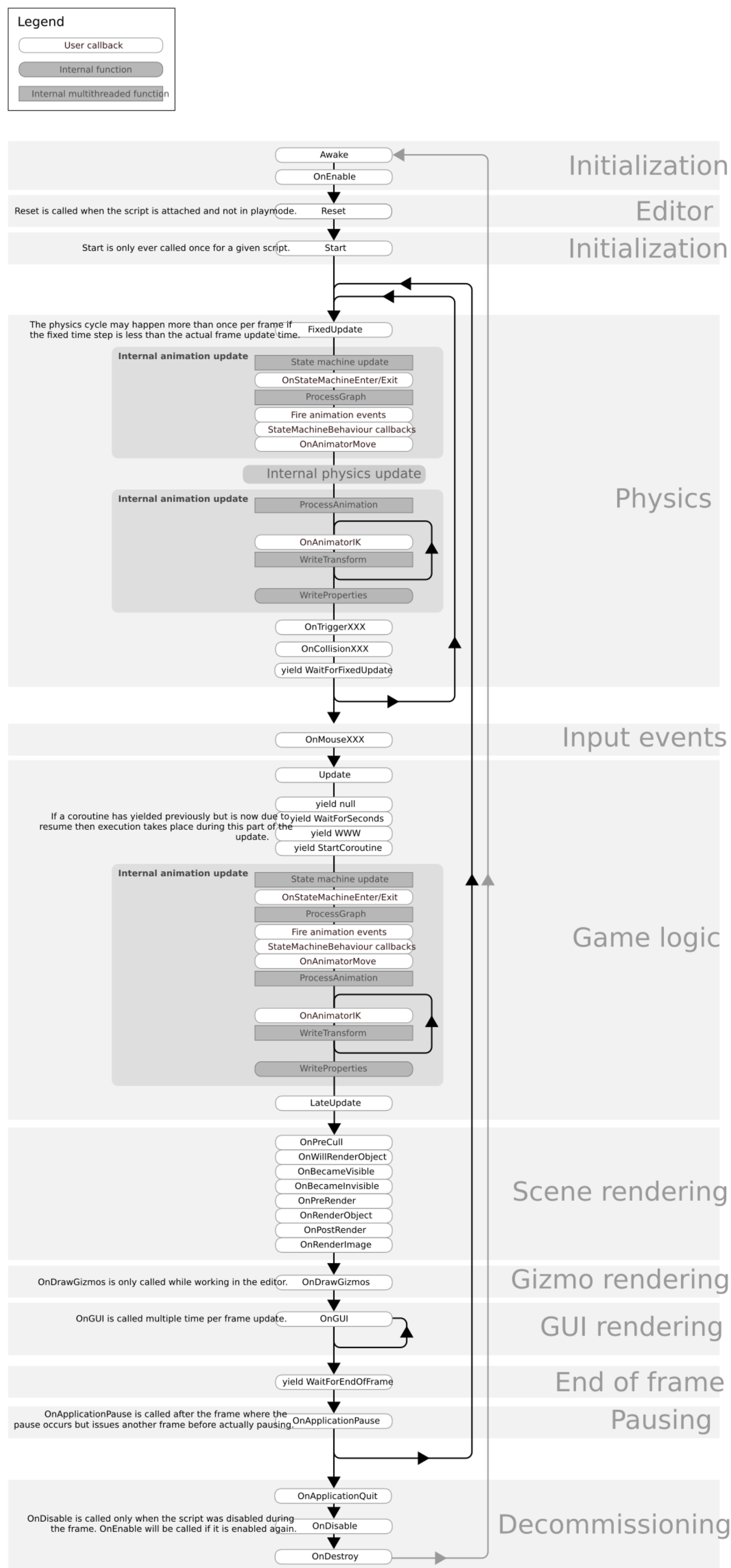


Figure 5.1: The order of execution of the methods defined in the MonoBehavior class. Source: https://docs.unity3d.com/Manual/ExecutionOrder.html

Chapter 6

Structure and Purpose of the Codebase

In this chapter, insight into the codebase which makes up the simulation is provided. Here, a UML class diagram and an a description of the role fulfilled by each module can be found.

6.1 Overview

In Unity, one can notice the absence of a clear-cut distinction between components that belong to the business logic (i.e. which have the sole purpose of storing the state of the application) and presentation logic (i.e. which are only needed to display data retrieved from the business logic to the user). Or rather, while it is possible to enforce such a decoupling in one's own codebase organization, the role of several components provided in the Unity library is mixed. As an example, changing the value of the fields of a `Rigidbody` class instance (a component that makes the Game Object it is attached to act as a rigid body, in the sense of classical physics) may produce effects that are immediately visible on screen, even though, technically speaking, it may be seen as a modification of the underlying state of the application. Rather than adopting a specific major software design pattern (such as the Model-View-Controller pattern, or one of its derivatives) it was decided to build self-contained components, or rather, scripts that contain fields that represent both the underlying state and presentational embellishments. This is because the added complexity and explosion in the amount of unique class definitions that would have resulted was deemed unnecessary, as well as because of time constraints.

6.2 UML Class Diagram

Here the Unified Modeling Language (UML) class diagram for the codebase is shown. However, with the exception of the Agent class (visible in the bottom right corner), "ready-made" classes which are provided in the Unity DLLs are omitted from the representation: it only shows those classes that have been modeled especially for the simulation. (Fields and most methods are also omitted.)

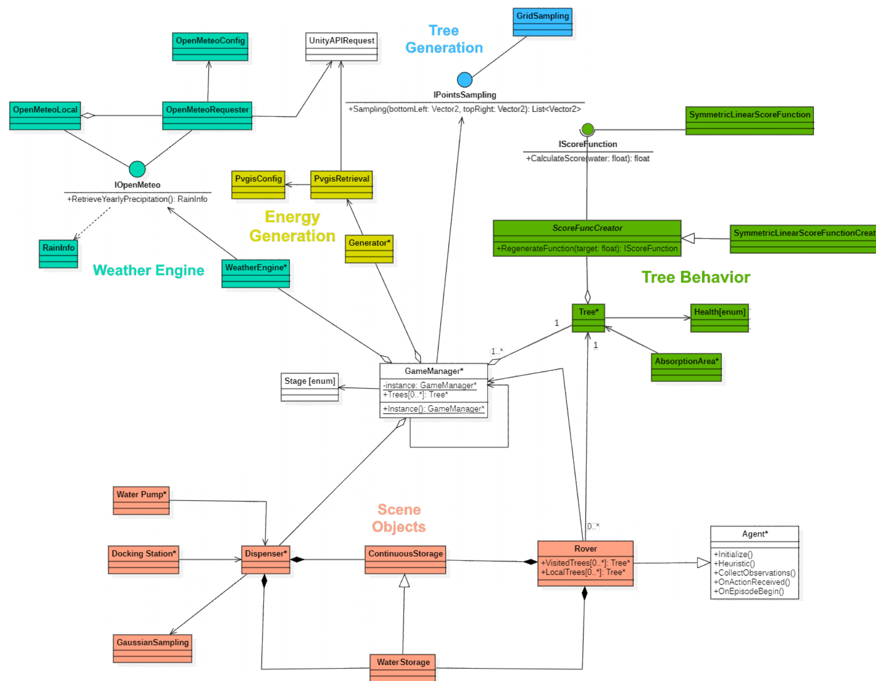


Figure 6.1: The UML class diagram for the software.

Furthermore, an additional dependency between the classes `AbsorptionArea` and `WeatherEngine` can be found. It is displayed in a separate image (lest the diagram in the previous image become cluttered):

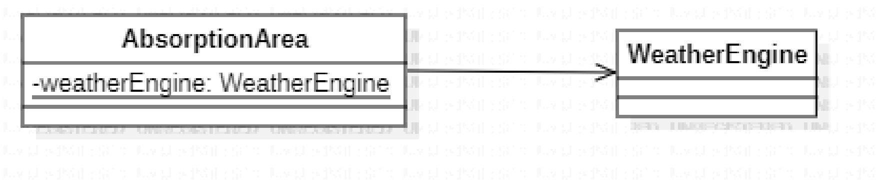


Figure 6.2: The dependency between classes `AbsorptionArea` and `WeatherEngine`.

Colors are used to group classes into clusters, according to the role they play in the simulation.
When appended to a class name, the asterisk * means that said class inherits directly from `MonoBehavior` (a class whose purpose is described in Chapter 5). The [enum] mark means that the class is actually an enumerated type (i.e. a a data type consisting

of a finite set of named values, which are then translated by the compiler - every occurrence of the name in the code is replaced with a number).
Unsurprisingly (given the nature of the component-based architecture), the `GameManager` class (which is a singleton) contains references to most of the core elements of the scene and handles its regeneration.

6.3 Insights on the Modules

6.3.1 PV Energy and Weather Statistics Retrieval

It is unwise to issue a request to obtain a JSON file containing the information described in Chapter 4 to the website that exposes the API endpoints every time the application is started, or, worse still, in multiple occasions during the same run. This is because a continuous reliance on a connection to the Internet is something most certainly unwanted: in addition, a limit on the daily number of API calls which are handled by the service may be imposed to users who cannot afford a costly subscription.
Therefore, a system to download the information only once (and then read it locally whenever it is requested again) has been devised. For the connection to the Open-Meteo API, the steps are the following:

1. The client (`WeatherEngine`) calls the `RetrieveYearlyPrecipitation` function of an `OpenMeteoLocal` object, which must return a `RainInfo` object.
2. A check is performed to verify if a text file (with a .txt file extension) with the information had already been downloaded in a location with an unchanging path (for instance, in the Windows file system it can be "C:\ReforestationSim\openmeteo.txt").
3. If no file was found, the `RetrieveYearlyPrecipitation` function of the `OpenMeteoRequester` is called by the `OpenMeteoLocal` object:
   - Firstly, an `OpenMeteoConfig` object is constructed: the construction involves opening and reading a configuration file (for instance, in the Windows file system it can be "C:\ReforestationSim\openmeteo.txt") which contains the values of the input parameters for the request.
   - Then, the values are passed to a freshly created `UnityAPIRequest` object (which acts as a wrapper for the HTTP GET request). A request is so created and sent: if it is successful, the data received by the server is processed and wrapped in a `RainInfo` object returned to the caller (which in turn will return it to the first caller in the chain, which is `WeatherEngine`).
4. The information retrieved in the previous step is saved in a file with the same path as the one in step 2.

A similar technique, if not somewhat simpler, is also employed for the extraction of energy statistics from the PVGIS API, although no interface is used.

6.3.2 Continuous Storage

The `ContinuousStorage` class is used to model the battery as well as the water tank and reservoir (via the `WaterStorage` derivate class). It can be extended to represent any sort of container that contains a real-valued quantity of something and can be loaded and discharged, regardless of the unit of measurement.

Some of the public methods purposefully violate the Command-Query separation principle (according to which a method should either produce a side effect or return a value, but not both): in particular, the method used to take an amount of the substance from the container both subtracts the amount and returns the subtracted value (which can be less than the requested value if there wasn't enough substance left), while the method used to add an amount of the substance only adds up to the maximum capacity, returning the remainder. This makes for a versatile system to guarantee that there are no losses while transferring the substance from a container to a different one, regardless of the transfer rate.

6.3.3 Gaussian Sampling

The implementation of the sampling from a normal (Gaussian) distribution in the `GaussianSampling` class is based on the Marsaglia polar method [8], which in its original formulation allows to sample from $\mathcal{N}(0, 1)$. By multiplying the obtained value by the standard deviation σ and adding μ we obtain a new value, as if it were sampled from $\mathcal{N}(\mu, \sigma)$.
However, the value must not be unreasonably small or large. Trying to "clamp" systematically the value in an interval through addition or subtraction could result in a lot of values "amassing" in the infimum and supremum of the set: this would induce a bias towards the two bounds being picked by the function.
Therefore, the algorithm performs a number of iterations until the sampled value falls within the interval: the number of iterations is finite, in order to prevent a potentially infinite (though very unlikely) loop from occurring.

6.4 Design Patterns Employed

Factory Method: Allows to create objects without specifying the exact class that will be instantiated: it is needed for the generation of `IScoreFunction` objects. Even though for now there is only one concrete type, we want the Score Function to be easily substituted with a more complex one in the future, which better models the thriving or decay of a tree: that is why an interface was adopted.
The `Tree` class does not have to be aware of the concrete type of the function: since the function has to be regenerated on a daily basis - given that the target amount of water also changes - a dedicated, specific creator class is used to handle the regeneration.

Proxy: Lets one provide a substitute or placeholder for another object. When the `WeatherEngine` requests the Open-Meteo data, it doesn't have to know that they could have been already downloaded in the past. `OpenMeteoLocal` acts as a "layer" that attempts to retrieve the data locally and in doing so prevents unneeded API calls, unbeknownst to the client.

Strategy: Enables selecting an algorithm at runtime. Used with `GridSampling` in the context of the generation of the 2D points where the trees should be located in the map: allows one to easily switch with a different algorithm, such as the Poisson disk sampling [9].

Singleton: Lets one ensure that a class has only one instance, while providing a global access point to this instance. It is used for the `GameManager` class, as it wouldn't

make sense to have multiple instances of the one element which has full control over the simulation loop.

Chapter 7

Visual Rendering of the Scene

This chapter is devoted to showcasing a variety of captioned images, showing the game renders.

Almost all the images that follow are screenshots of the “Scene View” in the Unity GUI (graphic user interface), besides the last one, which shows the “Game View”.
The core difference between Scene View and Game View is that in the former the user is allowed to inspect and click on any Game Object, drag the camera, zoom in/out as they please and perform many more actions, while the latter is basically tantamount to what the user would see while playing the actual videogame, outside of Unity. As one can see, there is a camera attached to the Rover: further considerations on the rover being endowed with an image sensor can be found in Chapter 8, Section 3.2.
All the images are files with a PNG (Portable Networks Graphic) lossless format, having a 32-bit depth (meaning that 32 bits are used to represent one pixel in the image).

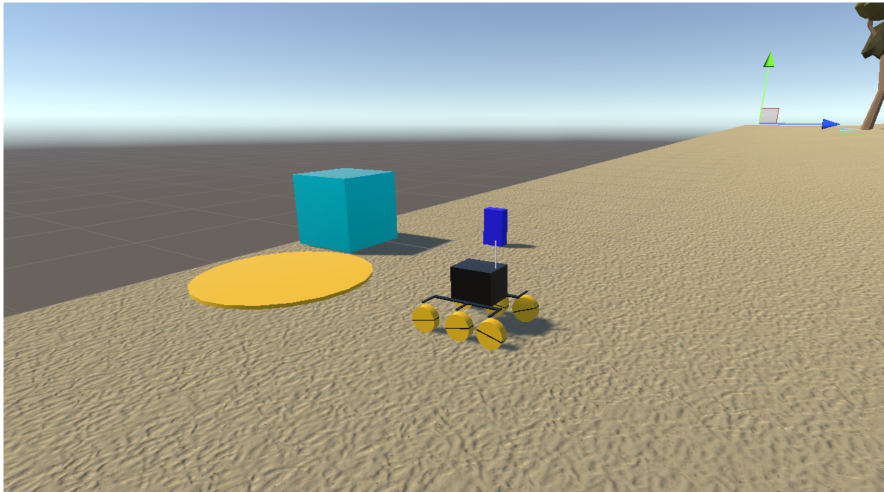


Figure 7.1: The rover, docking station (the yellow circular platform), dispenser (the light blue cube), and water pump (the dark blue parallelepiped).

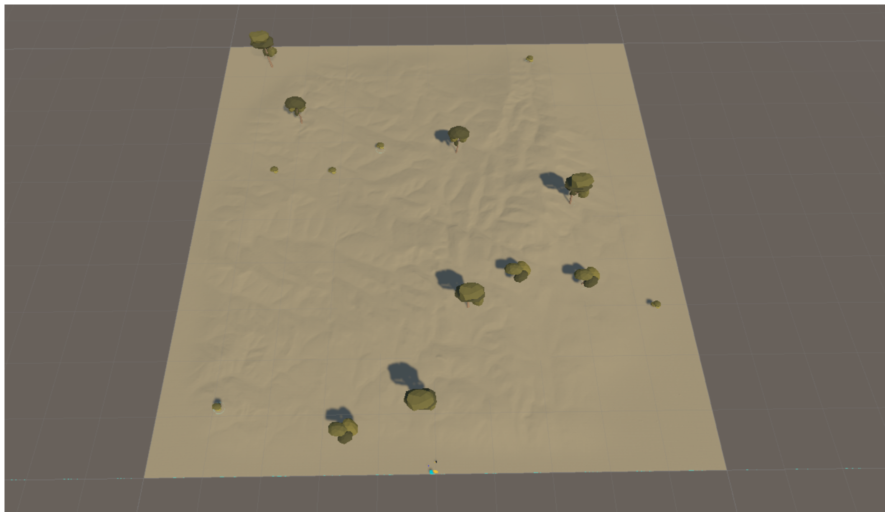


Figure 7.2: The Desert stage of the simulation. Only a few trees are present.

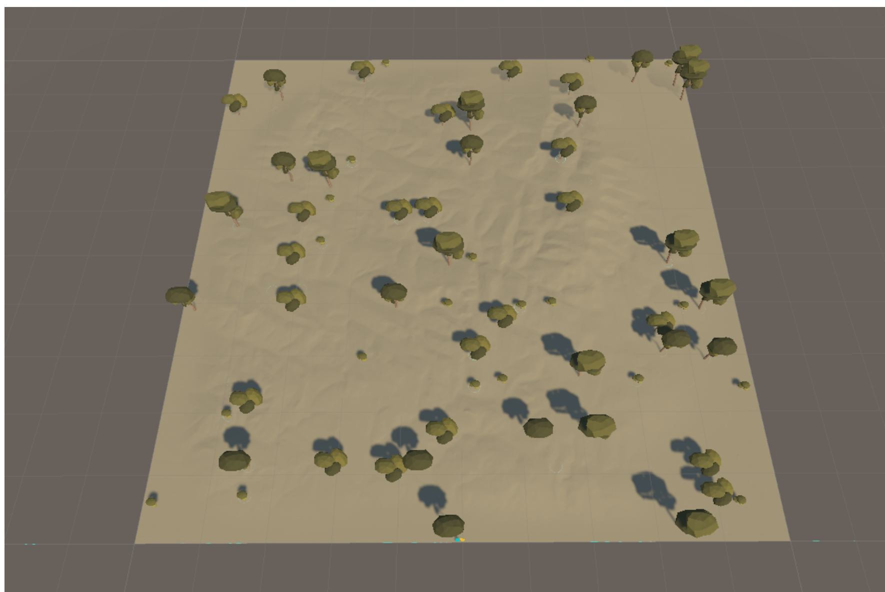


Figure 7.3: The Mixed stage of the simulation. The amount of trees is noticeably higher than in the previous stage.

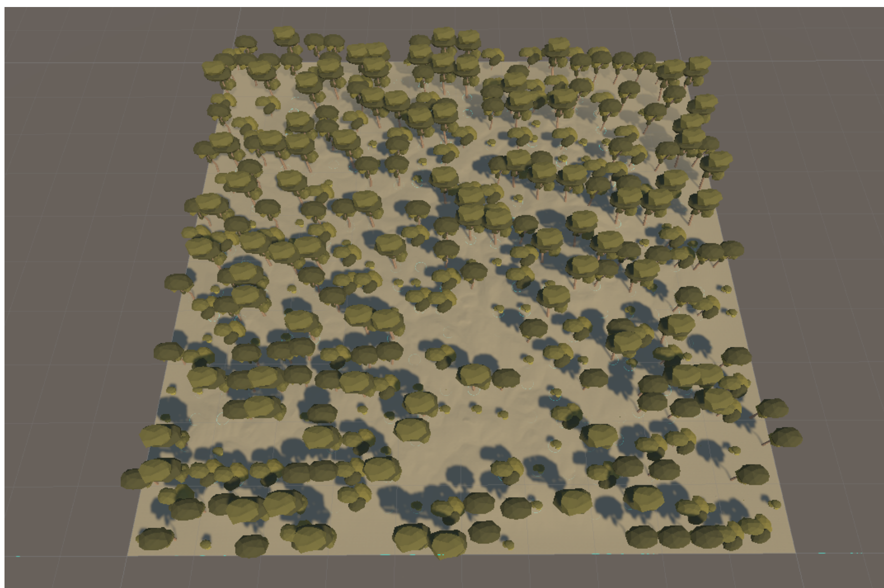


Figure 7.4: The Forest stage of the simulation. The area is mostly covered in trees.

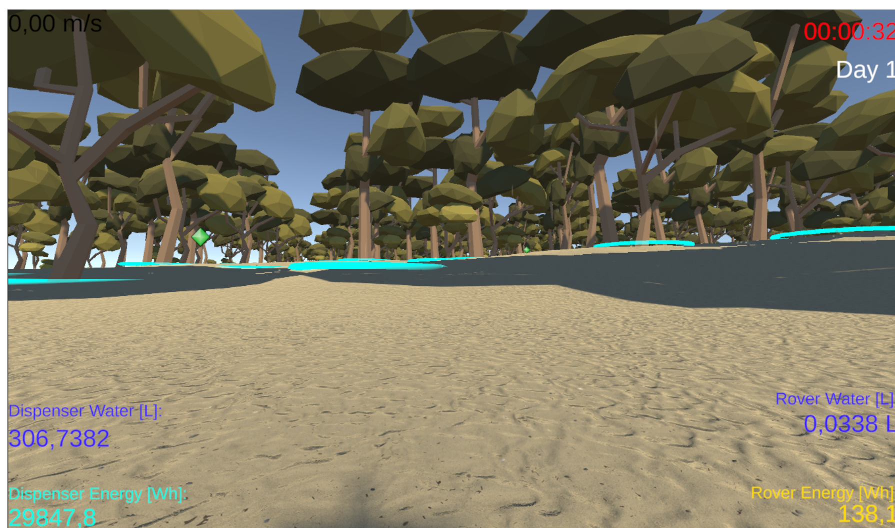


Figure 7.5: Screenshot of the game view, including the labels in the GUI. The area is viewed from the rover's perspective. In the corners a number of labels are visible: the current speed of the rover, the local time and number of elapsed days since the start, the amount of energy and water stored both in the rover and dispenser.

Figure 7.6: A close-up view of a tree. The absorption area of the tree is delimited by a blue circle (though the area is implemented as a spherical collider).

Chapter 8

The Rover as a ML-Agent in Unity

In this chapter, an overview of the concept of Reinforcement Learning as a whole, its implementation in the Unity ML-Agents Toolkit and the configuration of the Agent for the Rover is provided.

8.1 Overview of Reinforcement Learning

Machine Learning is a field of computer science and a branch of AI (Artificial Intelligence) which, broadly speaking, concerns with the development of software agents that are able to improve their effectiveness automatically with experience, thanks to the usage of underlying mathematical optimization methods. Reinforcement Learning (RL) is one of the three main categories of Machine Learning techniques (along with Supervised and Unsupervised Learning). Reinforcement Learning

> "[...] is the type of learning guided by a specific objective. An agent learns by interacting with an unknown environment, typically in a try-and-error way. This is the most common way of learning for a child, who does something and observes what it happens. The agent receives feedback in terms of a reward (or punishment) from the environment; then, it uses this feedback to train itself and collect experience and knowledge about the environment. Reinforcement Learning problems are related to learning which is the best action to perform, situation-by-situation, in order to maximize the aggregated reward. RL agent has to learn a policy (i.e. a complete mapping between situations and actions) by trying actions out without any domain expert has told it, as in many other forms of machine learning." [10]

Without delving into the mathematical foundations, it can be said that Reinforcement Learning revolves around four main concepts:

- **Actions**: the "degrees of freedom" of the entity that needs to be trained;
- **Rewards**: feedback received from the environment as a consequence of events occurring;
- **Observations**: pieces of information about the state of the environment, which are analyzed to infer what had happened when a reward was gained, in order to be able to reproduce the behavior;
- **Policy**: the strategy that suggests the actions to take, progressively upgraded by the Reinforcement Learning algorithm upon receiving Observations. Essentially, it is a map from Observations to Actions.

All of the elements above must provided to the training algorithm in the form of numeric values - a requirement which demands the concoction of a suitable mathematical model of the environment.
The goal is to maximize the cumulative reward obtained by the agent in a given timeframe, which in Unity corresponds to an Episode.

8.1.1 Training and Inference

While the details may be different, every branch of Machine Learning involves a Training Phase and an Inference Phase.

Training Phase

It involves building a model from sketch using the provided data, a process which can potentially last several hours or days, depending on the required specialization level. In the Unity Reinforcement Learning, the training phase learns the optimal policy through guided trials.

Inference Phase

It involves applying the model built during the Training Phase to new, previously unseen data. The agent observes and takes actions "in the wild" using its learned policy.

8.2 The Unity Implementation: ML-Agents Toolkit

The **Unity Machine Learning Agents Toolkit** (ML-Agents) is an open-source project that enables games and simulations to serve as environments for training intelligent agents. It provides several implementations for Reinforcement Learning, Imitation Learning, Neuroevolution and other methods to train intelligent agents for 2D and 3D games, and is based on the PyTorch library.

8.2.1 PyTorch

PyTorch is an optimized free and open-source (FOSS) framework for Machine Learning. It is a port of the now superseded Torch library, originally written in C but now rewritten in the Python programming language. In PyTorch, computations are performed using data-flow graphs: the output of the learning process is an Open Neural Network Exchange-compliant model, having the .onnx file extension, that can be associated

with an Agent. The learning process described here makes use of a pre-implemented algorithm made available by PyTorch (implementing a new one is an immensely complex operation, which requires advanced knowledge of the foundations of Machine Learning itself).

8.2.2 Steps and Episodes

In the Unity jargon, a Decision is the specification produced by a Policy for a single action to be carried out given an observation. A Step, on the other hand, corresponds to an atomic change of the engine that happens between two or more Agent decisions. The training process can be divided into Episodes: after the completion of a given number of Steps, modifications to the scene can take place. Such number can be set (or read) through the MaxStep field of the Agent class.
The rationale behind splitting the training into episodes is the following: if one sets the max step value to a reasonable estimate of the time it should take to complete a task, agents that haven't succeeded in that time frame will start a new training episode rather than continue to fail to obtain rewards.
An Episode should be lengthy enough to guarantee that some rewards will be obtained before the end but, at the same time, its length (in terms of number of Steps) should be limited. The end of an Episode can be forced externally: for example, after a fixed or variable amount of seconds have passed since the termination of the previous one.

8.2.3 Requirements of an Agent Script

Besides those of an ordinary Script, an Agent has specific characteristics, including properties and methods that need to be overridden by the programmer.

Parent Class

A ML Agent script needs to inherit from the Agent class, provided by the Unity ML Agents library, and not from MonoBehaviour.

Override Method: Initialize

The lines of code normally executed within the body of the Start method (see Figure 5.1) must be moved inside the body of the method named Initialize.

Override Method: Heuristic

The Heuristic method has the following signature:

```
public override void Heuristic(in ActionBuffers actionsOut)
```

The object identified by the formal parameter actionsOut has two public fields named ContinuousActions and DiscreteActions. The in keyword signifies that the parameter is passed "by value" and not "by reference": it is the equivalent of the C++ syntax ActionBuffers& actionsOut. It is needed in this context because ActionBuffers is a struct, and not a class: in C#, class-type objects are passed "by reference" to functions by default.
The type of ContinuousActions is ActionSegment<float>, whereas the type of DiscreteActions is ActionSegment<int>. Here, the generic programming technique is used: the angle

brackets, as in C++, enclose the type parameter of the class. The `ActionSegment<T>` class has a member called `Array` of type `T[]`, where `T` is the type parameter: `Array` inherits the type parameter of the object that contains it.
It follows that we are going to have two distinct arrays of type `float[]` and `int[]`. Each cell of the array must be assigned a numeric value that depends on the current state of an input mechanism: for instance, if the left arrow key is fully pressed in a given moment the value will be -1, or +1 if the right arrow key is pressed instead. The brain will take control of the inputs and send the signals itself, as if it were controlling the mouse or keyboard: however, a mapping between inputs and actions to be performed must be created beforehand.

The difference between discrete and continuous actions lies in how discrete actions have a fixed number of input values (thus the integer-valued array), whiile continuous actions are linked to inputs that can assume all the real values within a certain range. Continuous actions may be needed to process movement along a spatial axis, while discrete actions can be used to order that an "atomic" action be performed.

Override Method: OnActionReceived

The OnActionReceived method has the following signature:

```
public override void OnActionReceived(ActionBuffers actions)
```

When the `OnActionReceived` method is invoked by Unity, the same parameter as in the `Heuristic` function is passed. This method is invoked regardless of the inference or training mode being active. The instructions needed to perform the actual in-game operations "requested" in the `Heuristic` method should be executed in this method. In the example of the previous section, the lines of code that cause an object to move left or right along the Z axis should be located in `OnActionReceived`.
This is done by extracting the "pure" input values stored in the arrays. Note that the order of the elements in the array is important: the correct index of each input value in the array must be fixed and known.

Override Method: CollectObservations

The `CollectObservations` method has the following signature:

```
public override void CollectObservations(VectorSensor sensor)
```

Observations are numeric values. One of the overloads in the `AddObservation` group of methods of the sensor object must be called: howsoever, this will result in the `AddFloatObs(float)` being invoked. All values should preferably be in the [0,1] or [-1,1] range in order for the algorithm to converge at a higher speed.
As described earlier, the training algorithm needs observations to get the picture of the inputs that led to a higher reward being obtained than in the previous episode, and, in doing so, try to replicate them or improve their effectiveness.

Override Method: OnEpisodeBegin

The `OnEpisodeBegin` method has the following signature:

```
public override void OnEpisodeBegin()
```

Every operation which must be necessarily performed in the beginning of each Episode must be executed in this handler. The method will be regularly invoked upon the completion of a fixed number of steps, or manually (by calling the `EndEpisode` method).

Once the methods described earlier have been overridden and the training process has started, the agent simulation loop is orchestrated as follows:

1. Calls the `OnEpisodeBegin` function for each Agent in the scene.
2. Gathers information about the scene. This is done by calling the `CollectObservations` function for each Agent in the scene, as well as updating their sensor and collecting the resulting observations.
3. Uses each Agent's Policy to decide on the Agent's next action.
4. Calls the `OnActionReceived` function for each Agent in the scene, passing in the action chosen by the Agent's Policy.
5. Calls the Agent's `OnEpisodeBegin` function if the Agent has reached its Max Step count or has otherwise marked itself as `EndEpisode`.

8.3 The Rover

As seen in the previous section, the `Rover` class does not have `MonoBehaviour` as its immediate parent class: instead, it is `Agent` (which in turn is derived from `MonoBehaviour` as well). This allows the invocation of the specific methods listed in the previous section, needed for the algorithm to understand the actions and observations involved.

What follows is a group of tables where specific actions, observations and rewards (the three core elements of RL) are stated.

8.3.1 Actions

Name	Type	Values Range	Description
Forward Motion	Continuous	$[-1, 1]$	The value is 1 if it the maximum forward speed is desired, 0 if the rover is standing still, -1 for maximum backwards speed.
Steering Angle	Continuous	$[-1, 1]$	The value is 1 for a 45° right steering angle, 0 if the wheels are aligned with the vehicle's body, -1 for a 45° left steering angle.
Brake	Discrete	$\{0, 1\}$	The value is 0 if the brake is not activated in a given moment, 1 otherwise.
Water Dumping	Discrete	$\{0, 1\}$	The value is 0 if in a given moment the water is not being dumped on the ground (with the dumping rate mentioned in Chapter 4), 1 otherwise.

Table 8.1: The actions that the rover can perform as an Agent in the context of RL.

8.3.2 Observations

Needless to say, the physical rover is not an "all-knowing" entity, unlike the agent in the simulation. Observations should be decided sensibly, because they must pertain to quantities that can be physically measured through sensors and devices that can be mounted on a real vehicle and must be relatively affordable, budget-wise. For instance, the rover can be equipped with a Global Positioning System (GPS) sensor to understand its geographic location and the distance from the other elements; a camera, in order to know if a tree is in sight; a rain sensor, to understand whether it's raining or not in a given moment. All of the aforementioned elements can be powered by the same battery used to allow motion, or by an auxiliary one.

Note that the following observations are not guaranteed to yield a satisfactory model. They have been chosen speculatively and might be sub-optimal; this is because the devising of suitable RL observations and rewards requires a lengthy, careful "trial and error" approach which has not been attempted at the time of writing, owing to time constraints. Still, they are the product of a preliminary evaluation of the possible outcomes, no matter how naive.

All vectors are defined in $\mathbb{R}^3$ and contain x, y, z coordinates used to express distances in the 3D space. Note that the vectors have been normalized (though not in an "ordinary" fashion: each component has been brought in the $[0, 1]$ range by multiplying the vector by the scalar $\frac{1}{100}$ since the area is 100 m wide, and a distance across one axis can be of at most 100 m).
The observations are collected on every Fixed Update, which by default takes place every 0.02 seconds.

Name	Type	Description
Distance between Rover and Docking Station	vector	Obtained either by subtracting v_2 to v_1, or the other way around: so long as the position of the rover location vector and the other vector in the subtraction is consistent across all observations.
Distance between Rover and Water Pump	vector	
Residual Rover Battery Percentage	floating point	
Residual Rover Water Tank Percentage	floating point	
Residual Dispenser Battery Percentage	floating point	
Residual Water Reservoir Percentage	floating point	

Table 8.2: The "one-of-a-kind" observations collected during the training.

The observations in the second table are collected for each tree in the environment and stored in a variable length buffer sensor. The `BufferSensor` component makes use of an attention module [11], which can be useful in situations in which the Agent must pay attention to a varying number of entities. Attention mechanisms enable solving problems that require comparative reasoning between entities in a scene: this is because comparisons should be performed to determine which tree the Rover should move towards next.

Notes on the Orientation Index

The Orientation Index is a scalar product (or dot product) between the normalized vector that represents the distance $v_1 - v_2$ between the rover and a target (from now on v_d) and the normalized vector representing the Z (or "forward") axis of the rover in world space (from now on v_f).
The quantity

$$v_d \cdot v_f \tag{8.1}$$

is equal to 1 if the rover **facing towards** the target, 0 if the rover is **rotated by 90°**, and -1 if it is **facing in the opposite direction**.
This observation is needed in order to "aid" the rover in heading towards a target in the most efficient way (i.e. by moving forwards in a straight line, rather than zigzag or even backwards). A “coupled” reward will be needed to continuously dissuade the index from becoming smaller than 1 (See the subsection below).

Name	Type	Notes
Distance between Rover and Tree	vector	
Tree was Visited	boolean flag (0/1)	There exists a dynamic-length array where references to trees that were visited (meaning that the rover entered the Absorption Area) during the ongoing episode is kept: the references are cleared at the beginning of a new episode.
Is Raining	boolean flag (0/1)	Rain makes the provision of water to a tree unnecessary, if not even harmful. Measurement of the precipitation intensity (instead of assuming some fixed rate) could result in an improvement.
Orientation Index	floating point	It is explained above.

Table 8.3: The observations collected during the training for each tree in the scene, to be stored in a variable-length vector.

8.3.3 Assumptions on the Rewards

As the agent becomes increasingly skilled at doing certain things, new rewards should be added to mirror the developer's intent to teach a new behavior. A specific behavior should be coupled to an insulated "batch" of rewards and penalties, which is not initially active and can be activated later on when needed, during "breaks" in a training run (PyTorch allows the user to save multiple "checkpoints" that are created when a certain number of steps is reached, and resume the training from one of those checkpoints).
It is worth mentioning that while the reward system can be altered during pauses in a single training run, the observations must necessarily stay the same throughout the entire run. Sometimes it may be useful to "bind" together rewards and observations: in the case of the Orientation Index (see the previous subsection) the value can be observed while, at the same time, an appropriate reward or penalty can be given (for example by modeling a smooth reward function $f(x)$ with a global maximum for $x = 1$ and a global minimum for $x = -1$, where the argument x is the Orientation Index itself). Of course, this is not always possible (or recommended).
In the following chapter, which is devoted to the training, we will indeed return to the subject of rewards.

Chapter 9

Training Expectations

In this chapter, "non-prescriptive" hints, suggestions as well as expectations on the training process introduced in the previous chapter are provided to the reader.

Training a Reinforcement Learning agent, depending on the sophistication of the reasoning that the agent must be rendered capable of, often proves to be far from an easy task. While some training sessions have been attempted, the hardships faced while trying to induce the rover to maintain an acceptable cruise speed when moving towards a target (even in the presence of proximity-based rewards) only hint at how time-consuming the full learning process might be if the rewards and observations system is sub-optimal.
One of the unavoidable risks of RL is that the agent learns a behavior which is slightly or thoroughly different from the desired one - this doesn't happen because of a flawed learning algorithm, but due to poor choices in the nature of rewards (the agent finds a way to "cheat" and maximize the reward function in a way that the programmer was not aware of) or overlooking of constraints the agent is subject to in real life.

The goal of the training in its entirety is to minimize the cumulative loss of Health Points of the trees (something which obviously requires maintaining an ever-growing supply of water, and consequently of energy) over time as well as the expenditure of water and energy in a time frame that could possibly last months, or even years: this while preventing depletion of the rover's battery or the rover itself getting stuck. Ideally, the rover should successfully develop an ability to approach all the trees in the area, one by one, and water them regularly (potentially refraining from doing so in the presence of rain). Since the battery does not necessarily have to be fully recharged once the station is reached, the rover should concoct a "schedule" and plan ahead to recharge itself and refill the tank on a regular basis. However, the rover should also be careful not to try to reach a point too far from the docking station when the battery charge is soon to run out. All this should aim at keeping the Health Points of the trees constantly high: a major penalty should be carried in the event of a tree dying owing to water deficiency protracted over several days.

The number of rainy days in a year was extracted from Open-Meteo because we wanted the probability of rain at the beginning of an Episode to correspond to that very number, divided by 365 (the number of days in one year): however, a slightly

different approach can be tried. The point is that, while taking rain into account, there should be no "overfitting" either: the chance of rain should reflect that of the geographical location that is meant to be simulated.

One of the parameters of the Open-Meteo API call allows the user to choose the reference month in the specified location: this functionality should be exploited because the rover should be able to water an area in lieu of human workers for a period that spans multiple months, or even one or more full years, similiar to how a Mars rover mission can last two or more Earth years.

The position and number of trees in the area can can vary significantly. It was chosen to resort to the RL approach because treating this problem in terms of a multivariate mathematical optimization problem (which can be solved by a deterministic algorithm) may prove very difficult, given the inherent complexity and the sheer number of variables that come into play. As seen in the Specifications, a multiple-stage system (each "stage" representing a different stadium in the foreseen evolution of the forest) has been devised. The training should ideally be performed in an incremental fashion: a certain amount of episodes set in the first stage, with very few trees, followed by more episodes set in the second stage, and so on.
This is because the earliest part of the training must necessarily focus on the movement technique before anything else - much before any capability for "high-level" reasoning regarding the number of watered trees, health percentage and the likes is pursued. During this phase, which will involve trying to reach a fixed target in the shortest amount of time and learning to retain the same pace, the area should not be flooded with trees that would potentially act as obstacles - while circumvention of obstacles can (and ought to) be pursued.

Rewards should be given with a parameterized frequency that can be tuned (coherently with the "trial and error" approach that epitomizes RL), while observations can be normally recorded during Fixed Updates.
The training can be performed on a local machine as well as on cloud infrastructure. It is possible to set up an Elastic Compute Cloud (EC2) instance on Amazon Web Services (AWS) for training ML-Agents environments, or a Container or Virtual Machine on Microsoft Azure (further indications can be found at https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Training-on-Amazon-Web-Service.md and https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/Training-on-Microsoft-Azure.md respectively). In order to avoid a potential waste of time, it is of vital importance to mention that a single "fruitful" training run may start exhibiting signs of a somewhat successful learning after no less than a several hundred thousand steps, if not over a million: in the worst case, several runs are expected to be attempted just to see their results completely discarded and start again with a new one. This, along with the training being classified as a "heavy load" task in terms of CPU and GPU usage, is why one may be led to consider training remotely, benefiting from a reduced demand for user interaction and supervision in doing so.

Chapter 10

Conclusions

10.1 Final Remarks

The “digital twin” that was created during an internship at Piratech s.r.l., no matter how inaccurate in its current state, will hopefully be used as a work bench: in fact, the author believes that its strength lies in its modularity. Thanks to the adoption of code-level interfaces, many modules can be easily replaced over time with ones that reflect real-life behaviors with increasing accuracy. Moreover, the RL approach is not the only one that is guaranteed to work: different kinds of ML-based approaches (such as Imitation Learning) can be also tried while relying on the same infrastructure and codebase, although this may bring about the need for some degree of refactoring.
No real pretensions to strongly mirror reality was had in the first place. The very limited training (certainly not in terms of time, but in terms of achieved goals) that was performed, which was mainly devoted to teach movement from one point to another in the area, was partly underwhelming but encouraging at the same time. The author believes in the existence of large room for improvement, which can be attained through tweaks and corrections in the reward or observation system.

10.2 Final Balance of the Internship

The following table contains estimated duration for each major activity which would have taken place during the internship at Piratech s.r.l., with a total of 300 hours.

<table>
<tr><th>Estimated Duration (hours)</th><th colspan="2">Description of the activity</th></tr>
<tr><td>20</td><td colspan="2">Learning Unity and the C# language</td></tr>
<tr><td>20</td><td colspan="2">Learning the basics of ML-Agents in Unity</td></tr>
<tr><td rowspan="10">200</td><td colspan="2">Designing the virtual world</td></tr>
<tr><td>20</td><td>Random generation of terrains</td></tr>
<tr><td>20</td><td>Implementing the atmospheric engine</td></tr>
<tr><td>20</td><td>Implementing the ecologic engine</td></tr>
<tr><td>20</td><td>Obtaining stylised 3D models for the rover and trees</td></tr>
<tr><td>20</td><td>Displacement of the game assets in Unity</td></tr>
<tr><td>40</td><td>Creation of the C# scripts</td></tr>
<tr><td>20</td><td>Devising the game loop</td></tr>
<tr><td>20</td><td>Determining the ML goals according to the stage</td></tr>
<tr><td>20</td><td>Testing the ML-Agents</td></tr>
<tr><td>40</td><td colspan="2">Creation of a pipeline to interpret the results of the training</td></tr>
<tr><td>40</td><td colspan="2">Simulation of the reforestation process</td></tr>
</table>

While a majority of the tasks have been successfully carried out, testing the Agent proved a very long task, to the extent that the 80 hours that ideally should have been devoted to the last two activities in the table (creating a pipeline to interpret the results, then conducting a full-fledged simulation of the reforestation process) were entirely spent training the rover. On the other hand, prior knowledge of the C# language was already possessed by the developer (and author of the present document): this rendered the coding process substantially easier. However, it must be remarked that training the Agent was not the primary goal of the internship. No insurmountable difficulties were faced while designing the simulation and writing code, with one small exception: rather than use Blender to create fresh new 3D models, it was decided to resort on pre-built, downloadable assets from the Unity Asset Store website (`assetstore.unity.com`) for the trees and rain. This because systematically exporting meshes from Blender to Unity while preserving the original textures of the models was deemed too lengthy a process.

10.3 Expectations from the Internship: The Author's Perspective

This internship gave me a strong chance to test my understanding of the principles of Object-Oriented Programming (OOP) in a real-life scenario, in contrast to the work I did in the context of academic assignments so far. The realization that I overcame the difficulties in the coding part without ever needing to seek help from the Chief Executive Officer (CEO) or other more experienced co-workers certainly boosted my confidence. It may be argued that a slightly more consistent oversight of the work from the CEO - who had to deal with numerous other tasks in the meantime - could have translated into a more successful end result: yet, from a different perspective, this made me reflect on the importance of autonomy as a skill in the working field.
As for the "tech stack" of the project, we could say I didn't really have any specific initial expectations about what I would have learned: in the end I am glad to have gained some valuable knowledge of the inner workings of Unity, along with the fundamentals of Reinforcement Learning, which is a field I had no prior exposure to.

Acronyms

API Application Program Interface. 1, 37

JSON JavaScript Object Notation. 11, 37

REST REpresentational State Transfer. 11, 37

RL Reinforcement Learning. 24, 37

UML Unified Modeling Language. 15, 37

Glossary

API An application programming interface (API) is a set of rules that enables a software program to transmit data to another software program. APIs enable developers to avoid redundant work; instead of building and rebuilding application functions that already exist, developers can incorporate existing ones into their new applications by formatting requests as the API requires. A message directed at an API is known as an "API call". API calls have to be formatted in accordance with the API's requirements in order to work. The API's requirements are called its "schema". The schema also describes the types of responses that are provided to each request. 36

JSON JSON (JavaScript Object Notation) is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays (or other serializable values). It is a common data format with diverse uses in electronic data interchange, including that of web applications with servers. 36

REST REST (Representational state transfer) is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web. REST defines a set of constraints for how the architecture of a distributed, Internet-scale hypermedia system, such as the Web, should behave. 36

RL Reinforcement learning (RL) is an interdisciplinary area of machine learning and optimal control concerned with how an intelligent agent ought to take actions in a dynamic environment in order to maximize the cumulative reward. 36

UML The unified modeling language (UML) is a general-purpose visual modeling language that is intended to provide a standard way to visualize the design of a system. UML provides a standard notation for many types of diagrams which can be roughly divided into three main groups: behavior diagrams, interaction diagrams, and structure diagrams. The specification for UML was first introduced in 1997 by computer scientists J. Rumbaugh, I. Jacobson and G. Booch. 36

Bibliography

[1] Daniel T. Jones James P. Womack. *Lean Thinking, Second Edition*. Simon & Schuster, Inc., 2010 (cit. on p. 1).

[2] Leon Bernstein. "Crop Growth and Salinity". In: *Drainage for Agriculture*. John Wiley & Sons, Ltd, 1974. Chap. 3, pp. 39–54. ISBN: 9780891182115 (cit. on p. 3).

[3] Adel Khalil. "Dehumidification of atmospheric air as a potential source of fresh water in the UAE". In: *Desalination* 93.1 (1993). Proceedings of Desal '92 Arabian Gulf Regional Water Desalination Symposium, pp. 587–596. ISSN: 0011-9164. DOI: https://doi.org/10.1016/0011-9164(93)80133-8. URL: https://www.sciencedirect.com/science/article/pii/0011916493801338 (cit. on p. 5).

[4] Farhad Fathieh et al. "Practical water production from desert air". In: *Science Advances* 4.6 (2018), eaat3198. DOI: 10.1126/sciadv.aat3198. eprint: https://www.science.org/doi/pdf/10.1126/sciadv.aat3198. URL: https://www.science.org/doi/abs/10.1126/sciadv.aat3198 (cit. on p. 5).

[5] Miro Voellmy and Maximilian Ehrhardt. "ExoMy: A Low Cost 3D Printed Rover". In: Oct. 2020 (cit. on p. 9).

[6] *Unity*. URL: https://en.wikipedia.org/wiki/Unity_(game_engine) (cit. on p. 13).

[7] *Tips for Creating Thread-Safe Code (Avoiding Race Conditions)*. URL: https://objectcomputing.com/resources/publications/sett/april-2000-tips-for-creating-thread-safe-code-avoiding-race-conditions (cit. on p. 13).

[8] G. Marsaglia and T. A. Bray. "A Convenient Method for Generating Normal Variables". In: *SIAM Review* 6.3 (1964), pp. 260–264. ISSN: 00361445. URL: http://www.jstor.org/stable/2027592 (visited on 11/13/2023) (cit. on p. 18).

[9] Robert Bridson. "Fast Poisson Disk Sampling in Arbitrary Dimensions". In: *ACM SIGGRAPH 2007 Sketches*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, 22–es. ISBN: 9781450347266. DOI: 10.1145/1278780.1278807. URL: https://doi.org/10.1145/1278780.1278807 (cit. on p. 18).

[10] Muddasar Naeem, Syed Rizvi, and Antonio Coronato. "A Gentle Introduction to Reinforcement Learning and its Application in Different Fields". In: *IEEE Access*

8 (Jan. 2020), pp. 209320–209344. DOI: 10.1109/ACCESS.2020.3038605 (cit. on p. 24).

[11] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL] (cit. on p. 30).