Simulation of an autonomous spacecraft

Author:
Mark Mócsy

Budapest, 2023, May

Table of Contents

Abstract

A Bachelor Thesis in Computer Science: Simulation of an autonomous spacecraft

Mark Mócsy

This bachelor thesis focuses on the possiblity of creating a simulated environment to train an artificial intelligence to pilot a spacecraft.

The environment was created with Unreal Engine as its baseline, then built from that a realistic physical model of the Moon was created, along with the Apollo Lunar Lander, then a simplistic interface for the machine learning.

The physical model includes proper gravitational model, where the gravitational pulls of other, large celestial bodies are also counted in, moreover the local gravitational inhomogenity also factored in during the calculations.

The model of the Apollo Lunar Lander was based on the original technical reports of the Apollo Program, ensuring that it is as close to the real one as it could be, while at places allowing for slight modifications in order to ensure that the AI will not do things I do not want it to do.

Then an artificial intelligence has started to train in this environment to prove the original statement of this thesis.

THIS PAGE IS INTENTIONALLY LEFT BLANK

1. Introduction

1.1 Idea

In recent years, the exploration of space has been accelerating with a renewed interest in missions to the Moon, Mars, and beyond. Private companies like SpaceX have been at the forefront of this renewed interest, developing new technologies to make space exploration more accessible and cost-effective. One of the key innovations of SpaceX is the ability to reuse rockets, which can drastically reduce the cost of space missions.

The Falcon 9 rocket, designed by SpaceX, is one such rocket that can be reused. In addition to its reusable design, the Falcon 9 has the remarkable ability to land its first stage by itself. This is achieved through the use of autonomous guidance and control systems, which allow the rocket to land safely on a platform in the ocean or on land.

Currently they are using a conventional control system to achieve this unmatched feast, which has many advantages, for example, easy do implement and you will know how it will react in different situations. However, this is one of the main issue with a conventional control system, since it has only been coded to handle normal situations or a few extreme ones, it cannot handle unexpected events properly. For example, if the landing gear wouldn’t open, or the fins won’t turn as much, it might not be able to land at all.

There is two main solution for this: adding extra cases to cover all possible points of errors, including both hardware and software issues, or to use artificial intelligence and teaching it to be able to handle unexpected situations, for example sudden propellant loss.

This thesis explore the second option, via creating a modular environment where one could train a neural network to pilot a spacecraft.

1.2 Description of the task

The task at hand involves creating a simulation environment that specifically focuses on the landing phase of spaceflight, enabling AI to learn how to autonomously and successfully land a spacecraft. Landing a spacecraft is a critical and complex maneuver, requiring precise control and navigation in order to achieve a safe and controlled descent onto a designated landing site.

When designing the simulation environment making it modular or semi-modular is important in order to ensure that in the future it can be used for any kind of spacecraft. It should incorporate realistic physics models and accurate representations of the spacecraft's dynamics, control systems, and propulsion mechanisms.

THIS PAGE IS INTENTIONALLY LEFT BLANK

2. User Documentation

This software was made for insider use, for our aerospace engineers and machine learning specialists, in order to help the design process of an autonomous spacecraft. The main user group is the latter, while the developers are the former group. While writing this document I tried to keep in mind that any of the two group can be a developer or a user with only knowledge about half of the environment used in this software.

2.1 System Requirements

	Minimum	Recommended
RAM	16GB	64GB
CPU	AMD Ryzen 5 5600X	AMD Ryzen 9 9900X
GPU	Nvidia GeForce RTX 2060	Nvidia GeForce RTX 4070 Ti
Disk Space	128GB	128GB

Operating System: Windows 10/11

Additional Software: Unreal Engine 5.1, Visual Studio 2022 and PyCharm Professional

2.2 Installation

If you do not have Unreal Engine installed, please follow these steps:

Download and run the Epic Games Launcher installer, then open the application

Log in, or register then choose the ‘Unreal Engine’ tab.

Afterwards choose the library and click on the little + icon next to the Engine Version texts

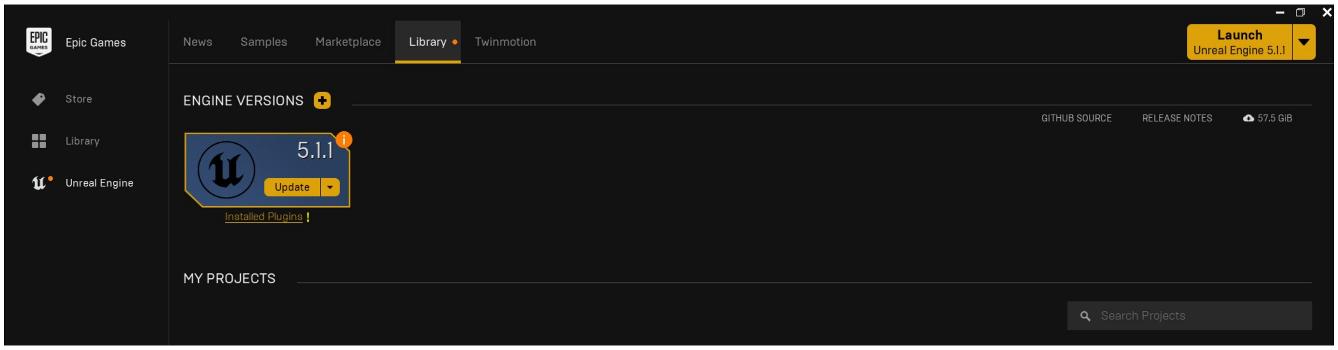


Figure 1: Installing Unreal Engine

Then just choose the correct version and click on install.

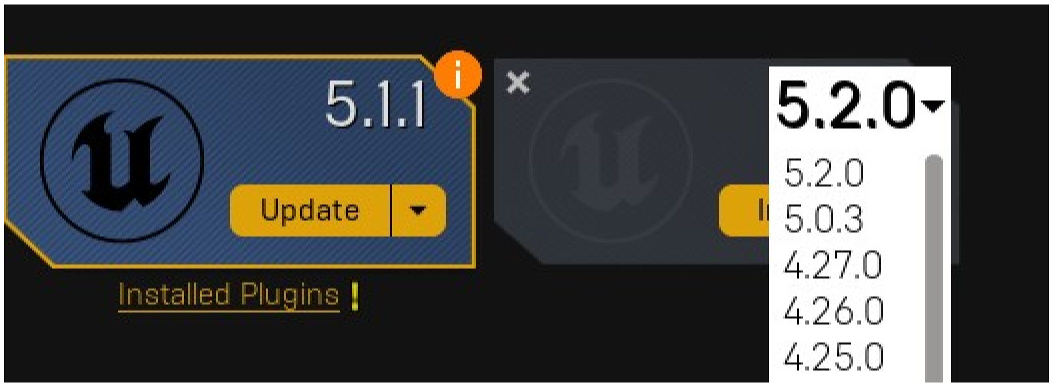


Figure 2: Version selection

Note: You can only see versions not installed on your pc, I have 5.1 installed, that's why I can't install it again. The list only contains the latest version of each minor release.

Ensure that these plugins are installed:

- Insta Deform Component: It can deform meshes, if those are defined correctly, used to create more realistic crashes.

- HLSL Material: This plugin helps creating and managing different materials and material textures

- Debug Function Library Lite: Easier debugging during run, gives the ability for extensive logging

- AsyncLevelLoad: Enables faster loading for the environment by creating parallel tasks

- Visual Studio Integration Tools: Information about the engine and better live coding/building experience

There might be a few others the Unreal Engine asks you to install, ones which were used during the development at a point, but currently not being used.

You can skip the first step if you plan to build the engine from the source code as well.

Then load the project in Visual Studio and run it with the default 'DebugGame Editor' settings.

If it fails, check if everything is installed in the correct place, sometimes the Engine is installed in Program Files\Epic Games\UE_5.1 instead of Program Files\UE_5.1. Alternatively you could rebuild the engine along with the project. Note: if the Unreal Engine tab at the Epic Games Launcher cannot identify the version of the engine the project was made in, you have to build the engine from the provided source code.

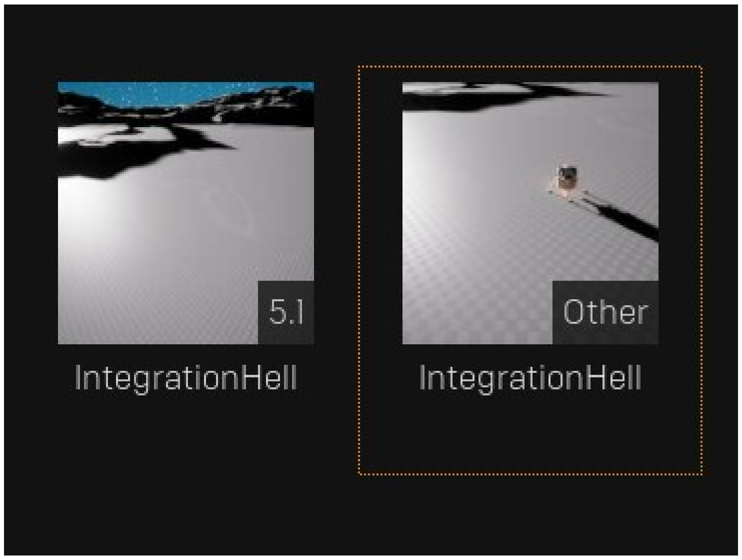


Figure 3: Same project, except that the engine used for the one on the right was compiled from the source code

Before you compile the source code, check if you can run it with the ‘Development Editor’ or with the ‘DebugGame Editor’ compile settings. (Do not use the other settings, as per those are not fully supported.)

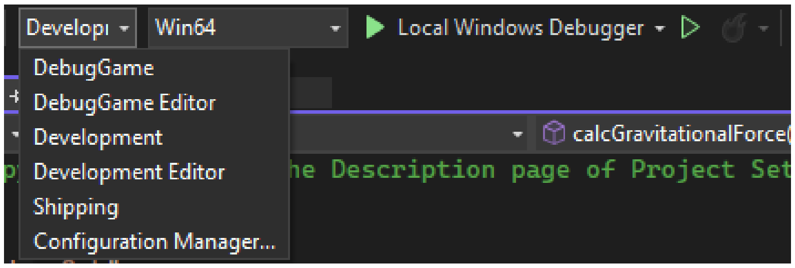
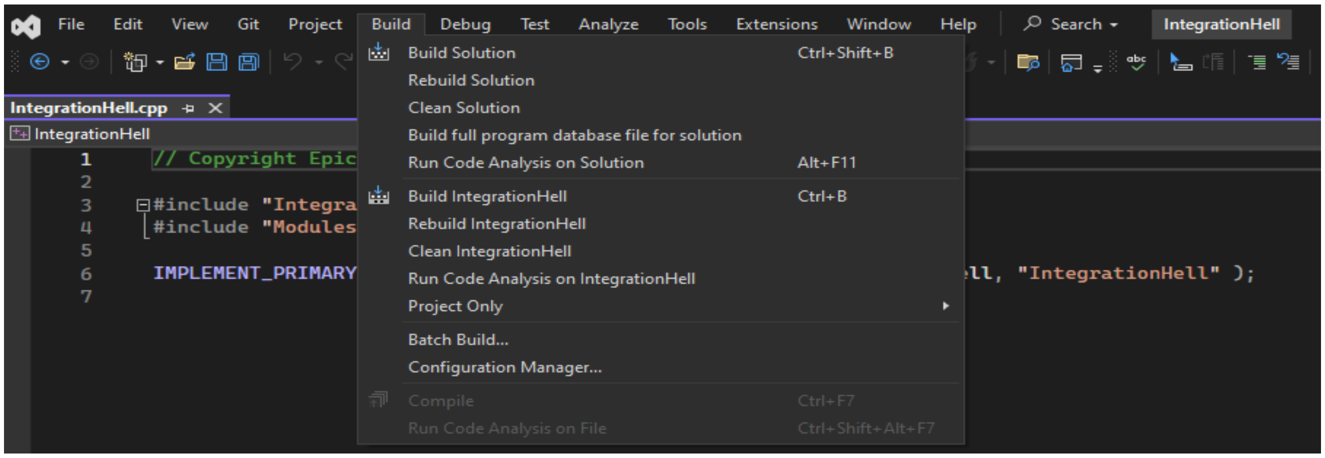


Figure 4: Possible options, only the ones with the 'Editor' are supported

In case you have to rebuild or build the engine, just open the build menu and select the ‘Rebuild Solution’ option, otherwise the ‘Rebuild <Project Name>’ should be sufficient.



```
// Copyright Epic

#include "Integra
#include "Modules

IMPLEMENT_PRIMARY                    ll, "IntegrationHell" );
```

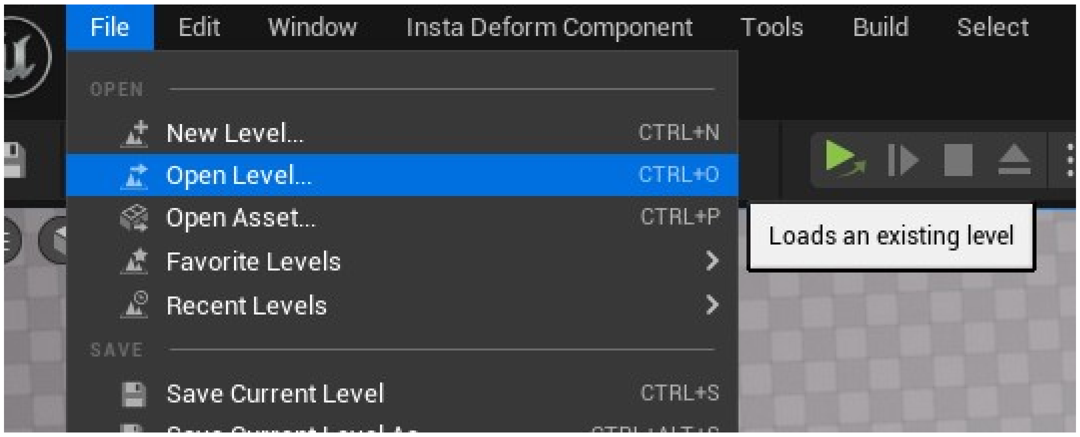
Figure 5: Build menu in Visual Studio

In case you plan to run the simulation and the AI on separate machines, ensure that the variable PythonServerIP is set to the correct address, otherwise the two modules will not be able to communicate.

2.3 Usage

Using the software is pretty straightforward, you either launch it from the Unreal Engine 5.1 graphical interface, or run the code with one of the 'Editor' settings in Visual Studio. I suggest the latter, since that gives you more freedom during the process.

Once the editor is running, you could create a new level where you can set up the map to train the AI. Alternatively you can open the 'ExampleLevel' I have prepared.



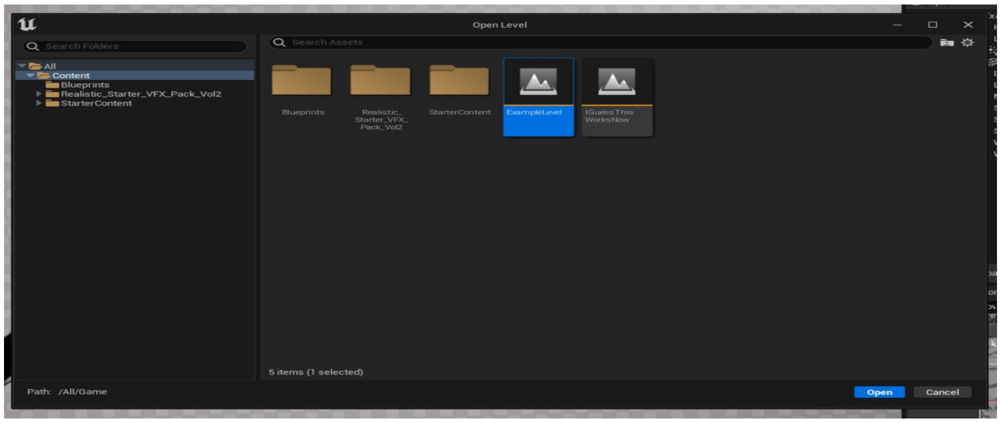


Figure 6: opening an existing level

If you made a new level, you have to manually add the spacecraft as an actor to the world.

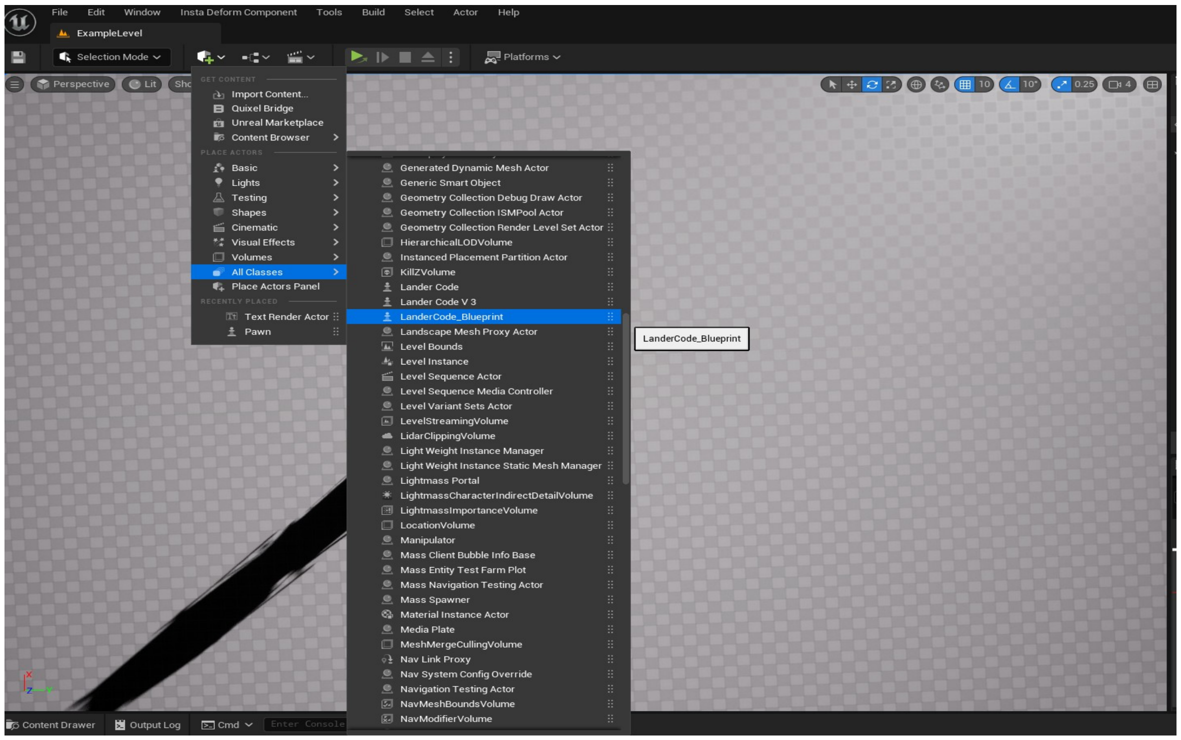
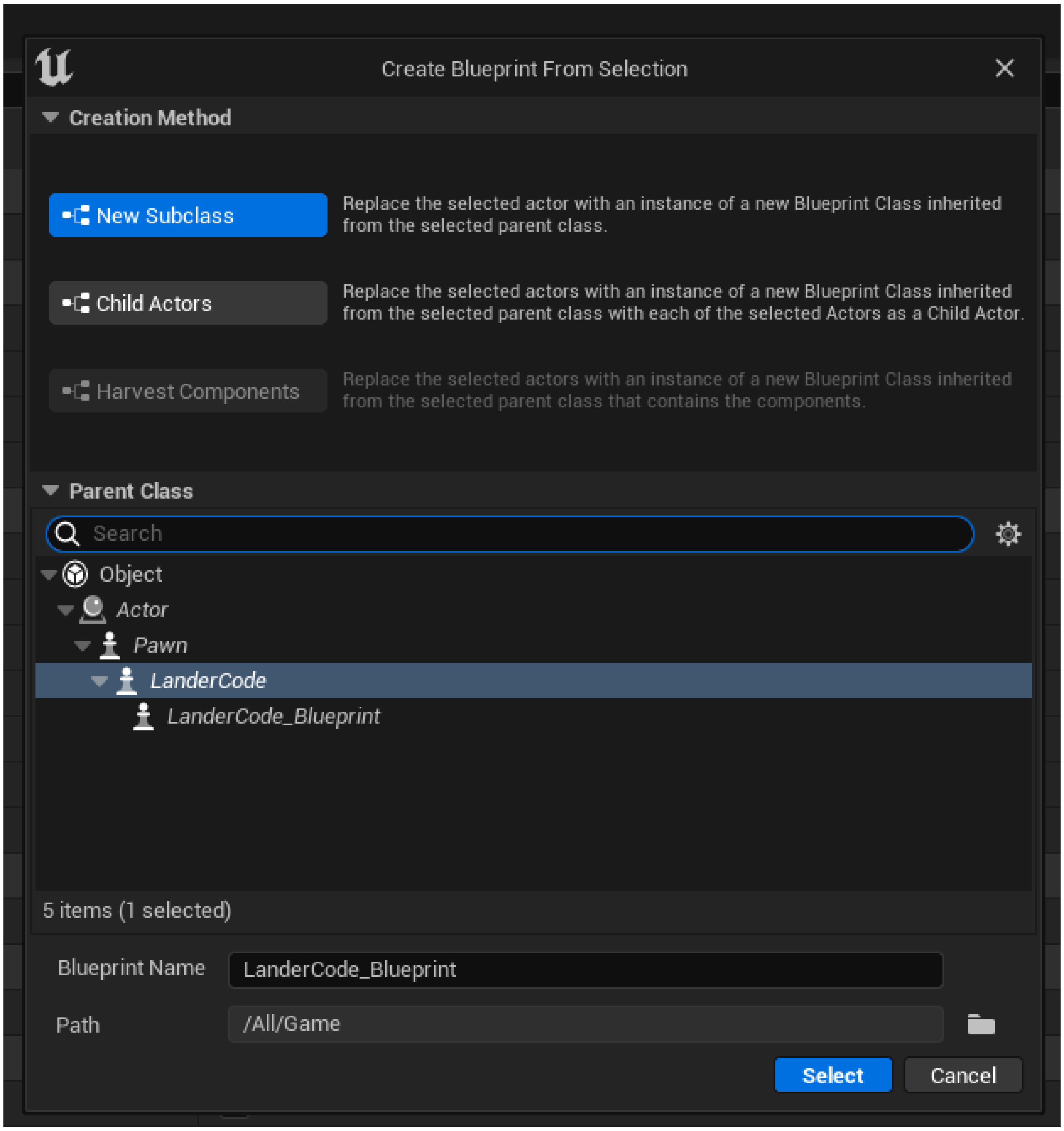


Figure 7: Inserting the blueprint version of the lander

There might be cases where you cannot see the blueprint version of the Lander, if that happens, do not panic, just place the Lander Code instead and convert it into a blueprint. It can be done via the blueprint icon on the details tab, while selecting the object.



Create Blueprint From Selection

Creation Method

Replace the selected actor with an instance of a new Blueprint Class inherited from the selected parent class.

Replace the selected actors with an instance of a new Blueprint Class inherited from the selected parent class with each of the selected Actors as a Child Actor.

Replace the selected actors with an instance of a new Blueprint Class inherited from the selected parent class that contains the components.

Parent Class

5 items (1 selected)

Blueprint Name: LanderCode_Blueprint

Path: /All/Game

Figure 8: Conversion to blueprint

Once the blueprint conversion is done, you can open the blueprint editor, where you can wire together and add the physical body to the code. If there's only a blueprint parent component there, first add a default scene component, then a wire mesh. You can do the latter via a simple drag and drop from the asset selector, there should be a prepared wire mesh model of the Apollo Lunar Lander in the content folder.

After this step, you can start to wire together a simple functionality. You'll need 2 events: BeginPlay and Tick. What you have to keep in mind during the process is that the base unit in the engine is centimeters, while the code uses the standard SI units, meters, so in order to ensure that the position of the lander in the world is the same as in the simulation, you have to multiply the position vectors by 100. The built in Kismet Math Library has a function for this. Alternatively you can downscale the mesh of the model by 100.

The setup phase is when you can edit the different weights used for the score and the reward function, you could do that in the code as well, but I suggest that you do that in the engine, since this way it's much faster to modify the values. These weight values should be positive numbers.

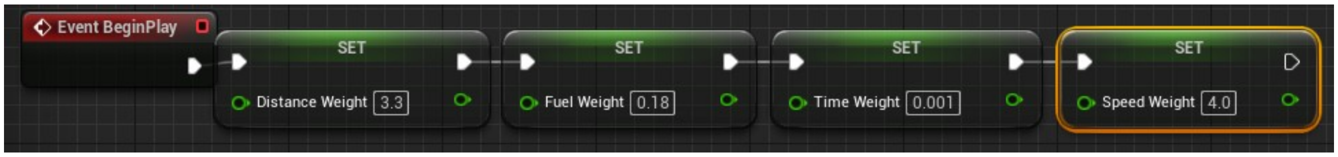


Figure 9: Example for weight setting sequence

You can skip this step if you are fine with using the default weight values:

Weight	Default value
distance	3.0
fuel	5.0
speed	9.0
time	0.01

If you do not use the Apollo Lunar Lander, as your spacecraft, then please jump to the Developer section, where I explain how to add different spacecraft to the simulation.

If you wish to train the neural network with noisy data, you can set the 'Noisy Sensor' variable to 'True' at this step.

With this, all variables are set, we can place the Lander at it's starting position, when the simulation starts. To achieve this, you should call the 'Restart Game' function, it's important, this function call is what ensures that the simulation and the neural network is synchronized at the start, since their loading time is different. Afterwards, use the built in 'Set World Location' function, give it the value of the 'Starting Coords' variable and check the 'Teleport' checkbox.

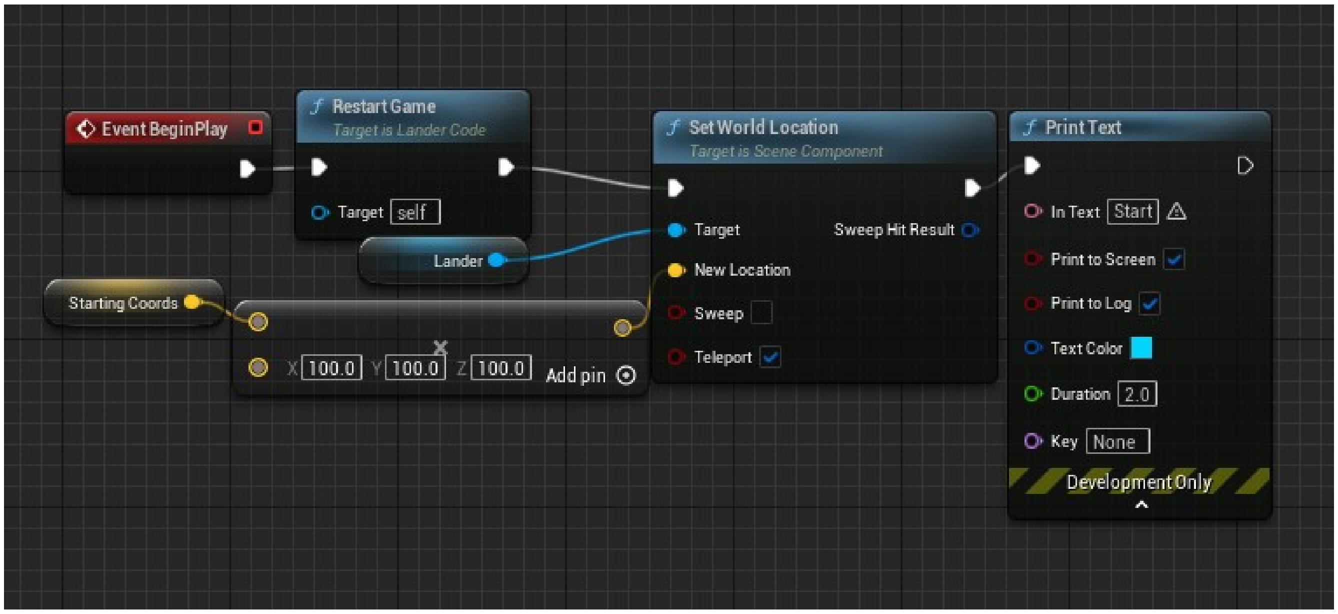


Figure 10: Example blueprint for the BeginPlay event with an additional text log output

Now, you can start to wire together the mid simulation parts. (It's done here and not in the code to ensure that the separate modules are working together properly, and to provide a visual aid about the process to you, the user.)

The next part is pretty straightforward: if the simulation is at the end of an etap, we restart the game, with the same setup as shown above, otherwise call the 'Lander Tick' function, feed in the delta time from the engine, move the component and check if it collided with the terrain and if it did, call the appropriate function, 'On Surface Hit'.

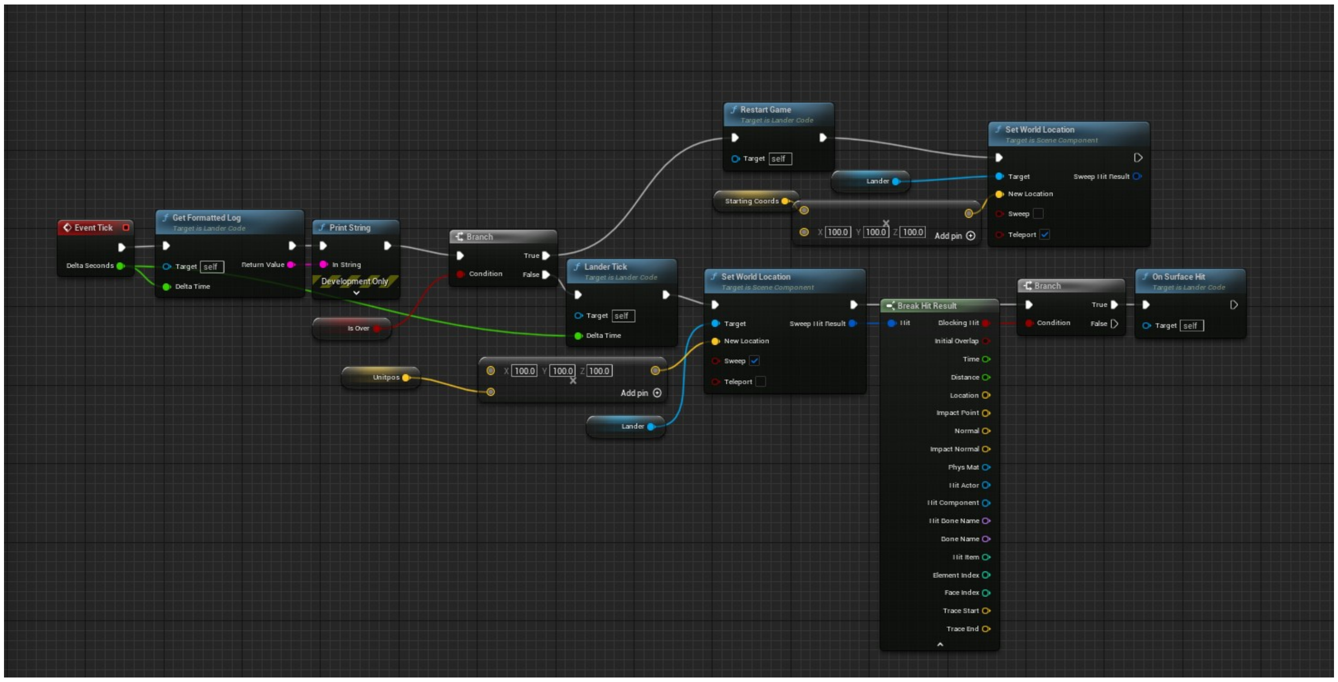


Figure 11: Example blueprint for the mid simulation tick with an additiona text log output

Now, the simulation is ready to be launched, but before you do that, you must set up and launch the neural network. If you check out the Python files, you can see two notebooks: Discrete and Continuous.

The first one should contain two examples – a Deep Q and a discrete A2C model. In order to properly train them, these will need a large action space – around 1440000000 different states, which makes it basically impossible to be used, unless you sacrifice usefulness via reducing the action space. I recommend that you do not use any discrete models, since the Lander will always operate in a continuous space, thus making the final version not that efficient as it could be.

The second one contains a Deep Deterministic Policy Gradient implementation, a slightly modified version of Antocapp's (Antonio Cappiello) Paperspace tutorial, to fit the environment.

The only parts you should touch here, outside the implementation of the neural network, is the host variable, that's needed to ensure the simulation and the neural network can communicate through the internet and the step function, especially the part where the returned action is denormalized, since the neural network usually returns with a vector of values between -1 or 0 and 1 while the simulation expects values for the engine between 0 and 100 and as for the gimballs between -6 and 6.

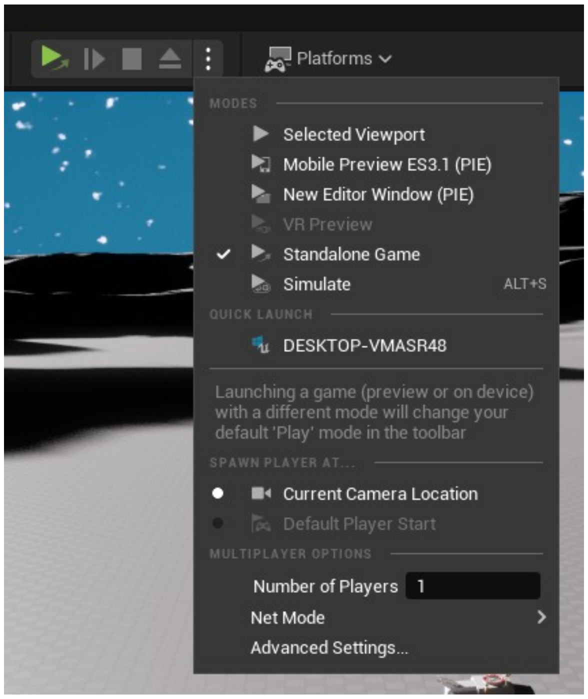
Once it's done, run the python code and then launch the simulation from the Unreal Editor and watch as the AI tries to land. (You could export the level to slightly increase the performance, but once you do it, you won't be able to modify a thing, so it is not advised.)

2.4 Notes

In some cases, after 10, 20, 50 or more etaps, the simulation or the training might stop, the most common error messages are ‘there are less than 68 bytes’ or ‘connection was forcible closed’. According to my investigation, these are the main causes of such:

- Memory corruption:
  - It might appear in the working memory of one of the parts, which might cause them to crash, or could affect the communication channel, where it might result in false data arrival flag or early end of message flag.
  - It’s not possible to avoid and totally random, the best option is to use Error Correcting memory, to reduce the chance of it happening. Running memory intensive processes during the training increase the chance of this issue to surface.
- Application forced shut down
  - In order to have a valid training process, you need both parts working, and so if one fails, the other might fail as well. The AI is more sensitive, the simulation will continue looping in most cases if the AI dies.
  - If you do not run the processes as an administrator and with high priority, the operating system might kill them to save system resources. Another cause for this issue might be your antivirus, since you’re running a user application which uses extremely high amount of system resources, thus producing a false positive report.
- System crash
  - Black screen or blue screen of death, the worst scenario, both apps and the operating system died.
  - Your hardware could not handle the load, either the PSU overdraw protection triggered, or the CPU/GPU overheat protection, causing your system to power down to save itself from permanent damage.

When you start the simulation, the best is to launch it as a Standalone Game in a new window.



✓ Standalone Game

● Current Camera Location

Number of Players 1

Launching a game (preview or on device) with a different mode will change your default 'Play' mode in the toolbar

During the development the focus was on the simulation, how to make it as realistic as possible while keeping the resource requirements as low as possible. In the Developer Documentation you can read the shortcuts taken in order to achieve this.

THIS PAGE IS INTENTIONALLY LEFT BLANK

3. Developer Documentation

3.1 Engine

The need for a game engine for a project like this is undeniable, since we plan to create a realistic environment, one which could be used later for machine vision learning as well. Creating any ray tracing algorithm is a hard to be done task for even larger companies, and would take up insane amount of time and research, so instead of that I decided to use a game engine to ease the development process.

Nowadays Unity and Unreal Engine are two of the most popular publicly available game engines. They have many similarities and differences, so let's compare them in some key areas:[1][2][3]

Ease of use:

Unity has a simpler user interface and is easier to learn for beginners. Its interface is clean and user-friendly, which makes it a great choice for developers who are new to game development or programming. Unity also has a wide range of tutorials, documentation, and a large community of developers that make it easier to learn.

Unreal Engine, on the other hand, has a steeper learning curve due to its more advanced features and interface. However, it provides a more powerful set of tools and advanced functionality, which makes it more appealing to experienced developers.

Graphics:

Both Unity and Unreal Engine offer powerful graphics capabilities. However, Unreal Engine is known for its advanced graphical capabilities, including features such as dynamic lighting, advanced particle systems, and real-time global illumination. It also has more advanced visual effects, which can help create more immersive and visually stunning games.

Unity, on the other hand, has more limited graphics capabilities and is better suited for simpler games or games with a more stylized art style. That said, Unity has been working to improve its graphics capabilities and has introduced new features such as the High-Definition Render Pipeline (HDRP) to enhance its graphics.

Performance:

Unreal Engine is optimized for high-performance gaming and can handle complex scenes with lots of characters and environmental elements. It uses a deferred rendering pipeline, which allows for real-time rendering of multiple light sources and advanced visual effects.

Unity, on the other hand, is designed to be more lightweight and is better suited for mobile and web-based games. It uses a forward rendering pipeline, which is less resource-intensive and allows for better performance on lower-end devices.

Scripting languages:

Unity uses C# as its primary scripting language. C# is a widely used programming language that is easy to learn and use, making it accessible to developers with varying levels of experience. Unity also supports other programming languages such as Boo and JavaScript, but C# is the most widely used.

Unreal Engine uses a visual scripting system called Blueprint, which allows developers to create game logic through a visual programming system. Blueprint is intuitive and easy to use, making it a great choice for beginners. Unreal Engine also supports C++, which can be used for more advanced programming tasks.

Licensing and cost:

Unity offers a free personal edition, as well as several paid versions with additional features. The paid versions range from $35/month to $125/month, depending on the features and support you need. Unity also charges a royalty fee of 5% on any commercial products created using the engine.

Unreal Engine, on the other hand, is free to use, but charges a royalty fee of 5% on any commercial products created using the engine. This means that you can use the engine for free, but if you make money from your game, you will need to pay a percentage of your earnings to Epic Games.

In conclusion, Unity and Unreal Engine are both powerful game engines with their own strengths and weaknesses. Unity is a user-friendly engine that is easy to learn and offers a solid set of features. It is well-suited for beginners and developers looking for a straightforward development process. On the other hand, Unreal Engine is renowned for its advanced graphics capabilities, including its superior ray tracing algorithm. It offers a more robust and performance-oriented engine, making it an excellent choice for developers who prioritize cutting-edge visuals and demanding projects.

It's worth noting that Unreal Engine has the advantage of being open source, which provides greater flexibility and customization options for developers who want to delve into the engine's source code. Additionally, if you are specifically interested in leveraging advanced ray tracing features, Unreal Engine's capabilities in this area may make it the more suitable choice for your project.

As an afterthought to this comparison, I have chosen Unreal Engine as the game engine for my project, and there are a few key reasons behind this decision.

First and foremost, Unreal Engine's open-source nature has been a major factor. The ability to access and modify the engine's source code provides an unparalleled level of control and customization. This allows me to tailor the engine to meet the specific needs of my project, adding unique features or optimizing performance in ways that align with my vision.

Additionally, Unreal Engine's advanced ray tracing algorithm has played a significant role in my decision. With its superior implementation of real-time ray tracing, Unreal Engine delivers stunning visual fidelity and lifelike lighting effects. This will greatly enhance the overall visual quality of my project and create a more immersive and realistic experience for players.

Moreover, Unreal Engine's extensive feature set and robust toolset offer a wide range of possibilities. From advanced particle systems and dynamic lighting to physics simulations and cinematic tools, the engine provides a comprehensive suite of functionalities that empower developers to bring their creative ideas to life.

Furthermore, the active and supportive community surrounding Unreal Engine has been another compelling aspect. The vibrant community provides a wealth of resources, tutorials, and forums, enabling developers to learn, collaborate, and seek assistance from fellow enthusiasts. This aspect of the Unreal Engine ecosystem ensures that I have access to valuable knowledge and support throughout my project's development.

And last but not least, the fact that Unreal Engine is scripting based engine. What it means is that it has a visual scripting, blueprint, for those who doesn’t know much about programming and for those who can code, a C++ based language, which fully supports all C++ features, except for the standard libraries, but there is a few which you could use if there is no function or type for your need. This made it possible to add support for codes written in other languages, since you are not directly coding, you are adding scripts or application programming interfaces to the engine, not classes or functions, which makes the whole process more modular and the final result will be less resource intensive.

In conclusion, the combination of Unreal Engine's open-source nature, superior ray tracing algorithm, extensive feature set, supportive community, and platform integration aligns perfectly with my project's goals. With Unreal Engine as my chosen game engine, I am confident in its ability to provide the necessary tools, flexibility, and visual fidelity to bring my creative vision to fruition.

A final note regargind Unreal Engine: It has a built in physics engine, which was designed for games and not for physical simulations, although through their scripting language it would be possible to turn it into a usable physics motor, but this isn’t as flexible as one written by me, nor could be as accurate. Although there is an async version of the physics motor, it does not have a stable release, and I would like to avoid any potential issues coming from an unstable version, therefore the safest route is to write a physics motor for this specific task.

3.2 Simulation

3.2.1 Environment

3.2.1.1 Gravity

When creating a gravitational model of a celestial object, one must check if they have to factor in additional celestial bodies, the inhomogeneity of celestial bodies and the rotation of the celestial body.

These depends on the length and the needed precision of the simulation, for example if the simulation is to determine the stability of an orbit, it has to be more precise and check for periodic changes in the gravitational field as well, while, in our case, a snapshot of the system is sufficient enough, since the landing won’t take more than a few moments compared to other time periods used in celestial mechanics.

Since most of the time the precision of the gravitational acceleration is only two decimals, and the precision of the mass of the celestial objects which could be found in different documents isn’t precise enough to let us calculate properly the third decimal, I have made the decision that I will only include something in the model if it affects the gravitational acceleration more than 0.001. Considering that there are three components, other than the mass of the object, and the 600 second long simulation time, we can easily calculate that the vertical speed component will be within 0.001*3*600 = 1.8m/s of the actual value. This might seem a lot, but considering the usual safety margins it is safe to assume that this much extra speed will not cause any issues.

The equation to calculate the gravitational acceleration:

$$a_{grav} = G\frac{M}{r^2}$$

Formula 1: Gravitational acceleration

From this we can calculate the ratio of the mass of the celestial object and the distance from our object, since we have determined that a=0.001 and G is the gravitational constant.

In the table below you can see the mass, the average distance and the gravitational acceleration caused by the given celestial body:

Celestial Object	Mass (10^{21} kg)	Distance (km)	$\frac{M}{r^2}$	a_{grav}
Sun	1989100000	149598023	88880176.63	5.93E-03
Mercury	330.11	91691000	39.26	2.62E-09
Venus	4867.5	41400000	2839.91	1.90E-07
Earth	5972.4	384399	40418890.54	2.70E-03
Mars	641.71	78340000	104.56	6.98E-09
Jupiter	1898187	62873000	480187.25	3.20E-05
Saturn	568317	1275000000	349.6	2.33E-08
Uranus	86318	2723950000	11.63	7.76E-10
Neptune	102413	1435140000	49.72	3.32E-09

Note: celestial objects, other than Earth, have nearly the same average distance from the Moon as from the Earth, this is why I used that distance to calculate it.

As you can see the gravitational acceleration of the Sun and the Earth is greater than $0.001 m/s^2$, therefore we must include them in our gravitational model. Although, since the distance between them and the Moon is significantly larger than the distance of the lander from the surface, we can consider the distance constant and ignore the inhomogeneity of these bodies, since from this distance these can be considered as point of mass.

When including them in the model, it is important to know their relative location to the landing site, since acceleration is a vector, thus the direction these bodies pull the lander has to be added.

The second part is the change of the gravitational field of a rotating body. At first glance a layman could say that it has a significant effect since there is a difference between the gravitational acceleration on Earth between different latitudes. But that effect comes from a different source, Newton's first law, and has little to do with the actual gravitational field. The change of the gravitational field by rotating a mass can be described by Einstein's gravitational theory, in 1918 H. Thirring mathematically showed that a rotating mass will slightly drag other bodies, although this force is not that significant. Nowadays it is known as the Lense-Thirring effect or metric and can be calculated via the following formula:

$$ds^2 = \left(1 - \frac{2GM}{rc^2}\right) c^2 dt^2 - \left(1 + \frac{2GM}{rc^2}\right) d\sigma^2 + 4 G_{\varepsilon_{ijk}} S^k \frac{x^i}{c^3 r^3} c\, dt\, dx^j$$

Formula 2: Levi-Civita metric

Where ds^2 is the metric, $d\sigma^2$ is the flat space element, ε is the Levi-Civita symbol and S is the angular momentum. From this, it is easy to determine that unless the mass of the rotating body or if the angular momentum, so the rotating speed is large enough, this effect cannot be measured. Considering that the rotating speed of the Moon is 2π/24 rad/days, and that it's mass is significantly smaller than the mass of the Sun, it is safe to assume that this metric is basically zero in our case.

The third component is the inhomogenity of the Moon. Let's say that the density function of the Moon is ρ(r,θ,φ), then the gravitational acceleration function at a given R+d distance from the center, where R is the radius of the celestial body:

$$a(\rho',\varphi')=\int_0^R\int_0^\pi\int_0^{2\pi} G\frac{\rho(r,\theta,\varphi)}{r^2+(R+d)^2-2r(R+d)(\sin(\theta)\sin(\theta')\cos(\varphi-\varphi')+\cos(\varphi)\cos(\varphi'))}dr\,d\theta\,d\varphi$$

Formula 3: Gravitational acceleration around a planet at a given height above mean sea level

The next step should be the calculation of the derivate function in order to find at which coordinates is the maximum and the minimum value is located. Doing it parameterized is difficult and would not answer the question which is how inhomogenious the celestial body could be, even if the values at the minimum and the maximum cannot differ more than by 0.002, since there will be still infinite number of possible solution to this equation. Alternatively I could use this equation with the data from the LGM2011 to model the Moon, but integration is an extremely costly operation, which I would like to avoid if possible, since hardware resources are extremely limited for me. Therefore, I looked at the values from the LGM2011, there is a few newer, more precise gravity field models, but the difference is in the 4th digit, thus not needed. From the model, I could determine that the gravitational acceleration around the Moon is between 1.6106m/s^2 and 1.6359m/s^2, which means a 0.0253m/s^2 difference.

It's easy to see that while we flight above an area, the gravitational acceleration will be constant, in the first 3 digits, so if I add a simple variable, which stay constant during an etap, and add it to the calculated acceleration, I could model the inhomogenious nature of the Moon, with basically no extra cost.

In the source code the following variables are used to describe the gravitational effect of the celestial body:

inhomConst:

A float type variable, it stores the local deviance of the gravitational acceleration from the average gravitational acceleration which we get from the mass and the radius of the celestial body.

Use it with the setinhomConst function, which generates the random local deviance value.

CelestialBodyMass:

A float type variable, stores the total mass of the celestial body on which the lander is trying to land.

CelestialBodyRadius:

A float type variable, stores the mean radius of the celestial body on which the lander is trying to land.

And the following functions are used to determine the gravitational acceleration:

calcTargetPlanetGravitationalForce:

Calculates the gravitational force between the lander and the celestial body at a given height. Since the coordinate system is set up in a way to always point towards the center of gravity, the first two components of the vector will be zero.

CalcGravitationalAcc:

Simple function to calculate the gravitational acceleration from a given center of mass at a distance.

CalcAccVectorCelestialBody:

This function is used to calculate the directional gravitational acceleration coming from a far away body. It uses a direction vector and the size of the acceleration vector.

CalcAccVectorCelestialBodyFromMass:

It works similarly as the function above, calculates the acceleration for each component from the distance vector and the mass of the celestial object.

3.2.1.2 Atmosphere

In order to properly model a planetary atmosphere, the following parameters have to be known:

- Composition of the atmosphere, what kind of gases can be found, potential polluting particles
- Mass and radius of the celestial object
- Distance from the star it’s orbiting around
- Speed of rotation
- Moons or other larger celestial objects around the planet

And so forth, this list is just the minimum amount of data needed in order to have a generic idea about the atmosphere of a planet. The geography has also significant effects, and less dense atmospheres behave slightly differently. So while it is possible to implement a parametric atmospheric model, which could be used for any planet, it is not wise to do so, due to the long list of parameters, and these models will always result in complex calculation, which I would like to avoid whenever it is possible in order to optimize the performance of the simulation.

Therefore based on NASA’s Earth Global Atmospheric Model[4] and the NRLMSISE Standard Atmospheric Model[5] I have implemented a simplistic atmospheric model of the Earth. This code can be used as a baseline for similar models for other planets, as long as it’s not too different from Earth.

The implementation does not include random wind gusts or the geographical differences. Most of the cases it’s not an issue, since most flights happen above the clouds, and at that altitude most geographical effects will not be that significant. In it’s current state it can give an estimated state of the atmosphere at a give coordinate. It is planned to add additional factors, such as weather, to this model.

From these two models I could conclude that the atmosphere can be modeled via a simplistic linear interpolation between specific altitudes, or via set values, so with a lookup table, for which I got the data from the NRLMSISE model.

If you wish to work with this, there are 4 functions and 7 variables which are used by the model:

equatorialRadius, polarRadius

float type variables, stores the polar/eqatorial radius of a planet, do not change it while the simulation is running

atm_temperature, atm_density, atm_pressure, atm_kinematicViscosity, atm_speedOfSound:

floatin point variables, these store the different parameters of the atmosphere at a given point

getPlanetEllipsoidRadius:

this function gives the radius of the planet at a given inclination

getGeopotentialAltitude:

this function adjusts the height to the local sea level, based on the actual radius

getAthmosphericData:

this function will calculate the atmospheric data, with the help of the set data points with the getISAData function, to which it passes the preset data which came from the mentioned above models.

GetISAData:

this function calculates the approximated value of the atmospheric data

3.2.2 Spacecraft

When modeling a spacecraft the most important part is the engine. When flying in space the rest could even be ignored, since with advanced gyroscopes or mechanically operated RCS stabilizers, which are not a modern technology, even the Apollo modules had some form of automated rotation control. Of course, there are cases when the spacecraft has to chance its orientation, but landing on a planetary surface is not one of them, except if there is a need for a longer flight and there is an atmosphere, so for example in the case of the Falcon 9 rocket, it is safe to assume that the autostabilizers are on and properly working, therefore simplifying the model, saving resources. Implementation of torque is in the development roadmap.

It is safe to assume that the engine used by the spacecraft is a liquid fuel chemical engine, because of the following reasons:

- Physical engines while having a high ISP, lack thrust. The exception is the cold gas thruster, but that still cannot provide sufficient thrust and has a low TWR. These are mainly used for course correction or in deep space where the gravitational pull of the celestial objects are negligible.

- While solid fuel or hybrid chemical engines are extremely powerful, they cannot shut down or throttle properly which is a must when it comes to maneuvering. These are mainly used for military purposes and as a booster during launch attempts.

Liquid fuel engines are the most complex and have the most variation, which would make the modeling process hard, but there is a few way to group them, which helps a lot. The first group is the monopropellant or the fixed fuel-oxidizer ratio engine. In this case the only value needed is the pressure at the injector and the fuel to determine the thrust of the engine. In most cases the throttle – thrust function will be a linear line, with a few breaking points, based on the valve or the injector, therefore if the thrust value at sea level or in vacuum is known for these points, the thrust can be calculated with a simple linear interpolation.

The situation with variable ratio engines is not this simple. In this case the different fuels and oxidizers could produce totally different thrust values, based on the mixing ratio and the pressure they arrive to the injector and then different injectors will produce different combustion rate in the engine, although since the injector is fix in an engine, it can be ignored, but it means that different engines with same fuel and same mixture ration could produce different thrust values.

Calculating the thrust for bipropellant engines is a two step process: First calculating the state of the chamber, then from that with some basic gas mechanics the thrust value can be determined.

The second step is the easy part, since there are parametric formulas for that, these could be found in the ‘Bible’ of rocket engineering, Rocket Propulsion Elements (George P. Sutton, Oscar Biblarz):[6]

$$v_e=\sqrt{\frac{2\gamma}{\gamma-1}\frac{RT_c}{M}(1-(\frac{p_e}{p_c})^{\frac{\gamma-1}{\gamma}})}$$

Formula 4: Average velocity of the escaping gas

The equation above gives the average velocity of the gas exiting the engine, where:

T_c is the chamber temperature in Kelvin

R is the universal gas constant

M is the average molar mass of the gas in kg/kmol

p_c is the chamber pressure in Pascal

p_e is the pressure at the nozzle exit plane in Pascal

γ is the isentropic expansion factor

$$F = \dot{m} v_e + (p_e - p_a) A_e$$

Formula 5: Thrust calculation

The equation above gives the thrust of an engine where:

ṁ is the mass flow rate in kg/s

p_a is the ambient pressure in Pa

A_e is the cross sectional area of the nozzle exhaust in m^2

$$\dot{m} = \frac{A p_c}{\sqrt{T_c}} \sqrt{\frac{\gamma}{R}} M \left(1 + \frac{\gamma - 1}{2} M^2\right)^{\frac{-\gamma+1}{2(\gamma-1)}}$$

Formula 6: Mass flow rate

The equation above gives the mass flow rate for the engine, where

A is the area of the nozzle throat in m^2

Most parameters are given from the design of the engine, what could differ is the chamber pressure and temperature and the average molar mass of the exiting gas. The pressure at the nozzle throat can be calculated from the chamber diameter, the throat diameter and the inner pressure.

With these equations one could calculate the thrust of a given engine, but determining the value of the parameters is not a trivial task, and requires deep understanding of fluid mechanics and thermodynamics, not speaking about the chemistry of the fuel, since it’s an imperfect burn even with perfect ratios, the result has to be calculated based on particle dynamics, making the simulation unnecessarily complex.

It is much better if the engine with different throttle and mixture ratios is tested or simulated with specialized software which results in different data points, and if the relation between these values are known, or can be approximated with polynomial functions, it could be used in the simulation environment.

Alternatively the sea level thrust values can be measured and from that, based on that data, and even simpler model can be used for thrust calculations.

I my case, I plan to model the Lunar Lander, which had the Descent Propulsion System, or DPS, as its main engine. Unfortunately not many of the original documents are available today, lot of the data was lost during the years, but I was able to locate 3 technical documents in NASA's archive, Apollo Spacecraft Liquid Primary Propulsion Systems which contained the summary of the propulsion systems planned to be used for the Moon missions, Apollo Experience Report – Descent Propulsion System, a detailed document about the properties of the DPS, Liquid Rocket Engine Nozzles, which is mainly about the effects of different rocket engine nozzles, which did not contain direct information about the Lunar Lander, but useful equations which I could use to validate my calculations based on the data from the static fire tests.

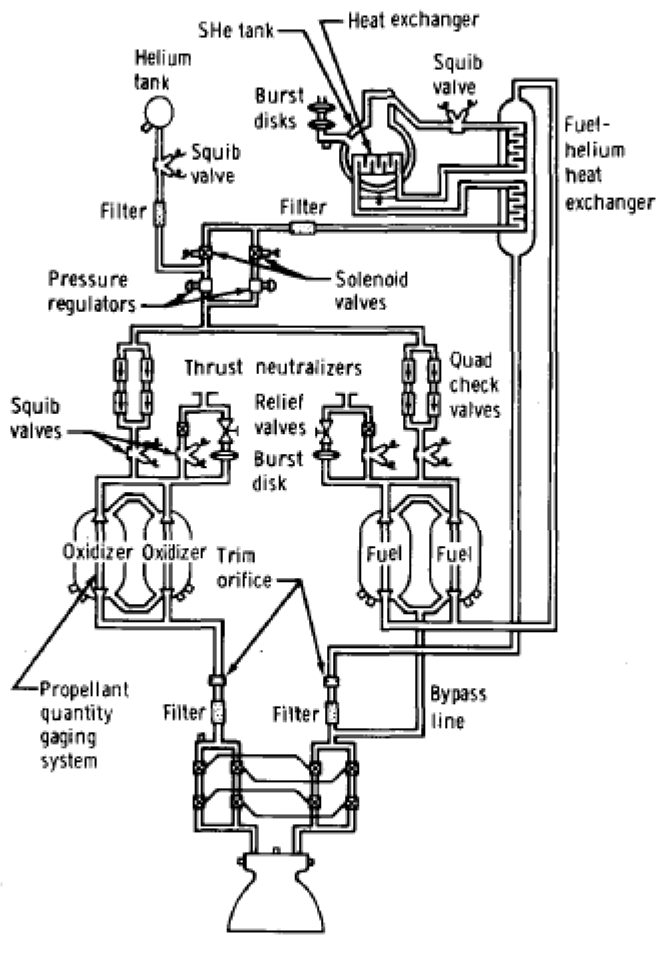


Figure 12: Schematic of the DPS (Source: [7]*)*

The DPS had a simplistic design, the engineers of the Apollo program could not afford any potential error during the landing, since there was no way to fix anything during the mission. This lead to the decision of using a pressure fed cycle and a fixed ratio of the oxidizer and the propellant, which means it can be simulated as a monopropellant engine. (Note: While theoretically it was possible to change the ratio, it was prohibited for the astronauts, since it could have lead to issues mid flight.)

The DPS used a pintle injector which allowed the engine to be throttled between 100 and 10 percent of it’s total thrust, below 10 percent throttle, there was not enough material in the engine chamber to keep the combustion alive, therefore going below 10 percent meant that the engine was shut down. There was only one guaranteed res e flights it was proven that it can handle up to 4 restarts, h f possible.

While the thrust values are k ısured thrust value, while for the simulation it was import or if there is a few breaking points. The fact that the 60 a except for full thrust, meant that the thrust-throttle functi

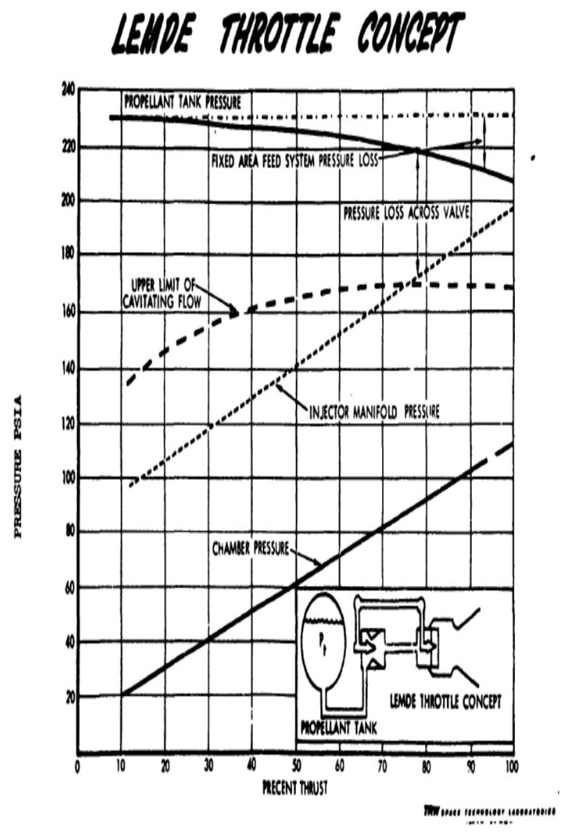


Figure 13: Throttle - Pressure diagram (Source: [8]*)*

There isn't many references left to be used for the simulation, although in one of them could be found the diagram above which describes the different pressure values of the engine, before and after the injector and inside the engine.

From this diagram and from two other documents related to the propulsion systems used during the Apollo program[7][8][9], I could determine that there was a pressure drop at around 95, 75 and 65 percent thrust, so the non operating zone in the manual included an extra 5 percent, in case due to potential manufacturing errors, which might have an effect on these breaking points. In the code I used 95 and 65 as the breaking points, and in order to discourage the AI from using a throttle value between 65 and 95, I deliberately made it less efficient in this throttle range.

The rest of the model is straightforward, except for the minimum change in the thrust vector control and in the throttling. These values came from past experience when working with similar systems, same for the accuracy of the PID system used for setting the desired values for these.

In the code the following functions and variables are used to describe the lander and it's behavior in the simulation environment:

min_throttle_change: the minimum value needed to change the throttle

throttle_pid_accuracy: how accurately can the lander set the given throttle value

min_gimball_change: the minimum movement of the gimbal

gimball_pid_accuracy: how accurately can the lander set the gimball angle

monopropellant: the current amount of fuel

gimball_pos_x, gimball_pos_y: current TVC values

engine_throttle, fuel_valve, oxidizer_valve: current state of the engine's throttling

battery_charge: charge of the batteries

target_throttle, target_ox_valve, target_fuel_valve, target_gimball_pos_x,target_gimball_pos_y:

target values for the PID system

number_of_restarts: how many times the engine can be restarted

drymass: the mass of the lander without the fuel

FuelDecrease: calculates the amount of fuel used up in the last DeltaTime seconds, based on the throttle

adjustComponents: models the PID, moves the components with a set speed.

ThrustToApply: calculates the thrust vector of the lander

3.2.3 Spacecraft-Environment integration

The following methods used to ensure that the spacecraft behaves properly in the environment, which includes both the game engine and the physics model.

CalcSpeedGainFromHeight: simple function, calculates the velocity the lander will hit the ground from a given height and starting velocity.

LanderAcceleration: calculates the current acceleration vector for the lander

calcVelocity: calculates the current velocity of the lander from the acceleration and the previous velocity value

calcPosition: moves the lander in the simulation space based on the current velocity and the DeltaTime seconds

onSurfaceHit: this function determines whether the lander survived the landing or not. Currently the maximum velocity is 7m/s, the reason behind this value is simple: there is game, Kerbal Space Program, where the components disintegrate if they hit the surface with 7m/s. A more realistic value would be ~15m/s, but since the goal is to ensure that the AI lands the spacecraft as smoothly as possible, having a lower threshold is better.

3.3 Machine Learning

When it comes to the creation of an autopilot for a spacecraft, there are many different options. One of these options is machine learning, especially reinforcement learning, the following reasons made me decide that for this project it will be optimal.[10][11][12]

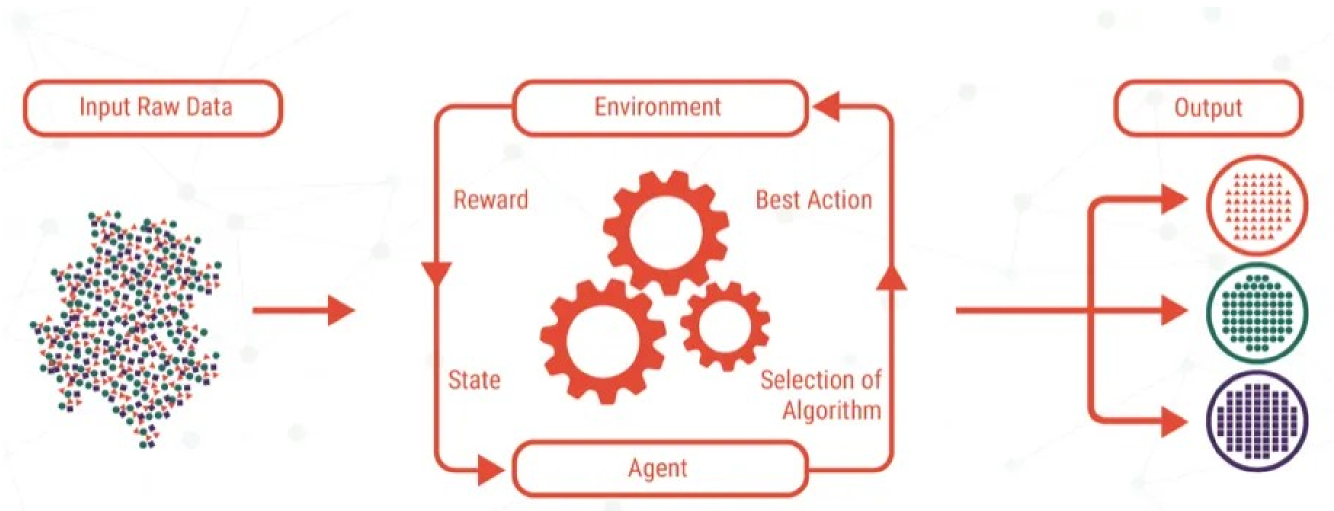


Figure 14: Reinforcement Learning (Source: [10])

Complex and Dynamic Environment: Piloting and landing a spacecraft involve operating within a complex and dynamic environment, with numerous variables and changing conditions. Reinforcement learning is well-suited for such tasks as it allows the AI agent to learn and adapt its behavior based on feedback received from the environment. By interacting with the environment, the AI can explore different actions and learn to make optimal decisions in real-time.

Lack of Explicit Instruction: In many cases, explicit instructions or rules for piloting and landing a spacecraft may be difficult to define or may not be available. Reinforcement learning enables the AI to learn through trial and error, without the need for explicit programming or predefined rules. The AI learns by receiving feedback in the form of rewards or penalties based on its actions, allowing it to develop its own strategies and policies.

Generalization and Transfer Learning: Reinforcement learning algorithms have the ability to generalize knowledge and transfer it to new, unseen situations. Once an AI has learned how to pilot and land a spacecraft through reinforcement learning, it can potentially apply that knowledge to different spacecraft or similar tasks with minimal additional training. This ability to generalize and transfer learning is particularly valuable when dealing with spacecraft, as there may be variations in designs or mission requirements.

Adaptability to Uncertainty: Space missions often involve uncertainties and unpredictable situations. Reinforcement learning allows the AI to adapt and make decisions in the presence of uncertainty. By continuously interacting with the environment and receiving feedback, the AI can adjust its behavior and learn to handle unforeseen circumstances, such as sensor failures or changing weather conditions.

Iterative Improvement: Reinforcement learning facilitates iterative improvement over time. The AI agent can continuously learn from its experiences and update its policies to enhance performance. As more data is collected and the AI gains more experience, it can refine its decision-making process and achieve better outcomes. This iterative improvement is crucial for the safety and reliability required in spacecraft piloting and landing.

However, it's important to note that training an AI requires extensive simulation environments, accurate models, and careful consideration of safety measures. Additionally, reinforcement learning alone may not be sufficient, and it may be necessary to combine it with other techniques or incorporate human oversight to ensure the safety of the mission.

At first I implemented a simple Deep Q network, a widely used reinforcement learning algorithm with a discrete action space, and then a discrete Actor Critic algorithm, so I could compare the two. However, this was not working, since the environment used a continuous action space, which meant I had to discretize the action space without restricting the AI in any way. This meant that the final action space had (12/0.01)*(12/0.01)*(100/0.1)=1,440,000,000 different states. With one state being stored in a single byte, this means one needs 1,44 GB of RAM for the result vector. In order to have a neuron layer which can produce such result, one of its dimension must fit this, and with multiple neurons in a single layer, it would eat up terabytes of RAM in seconds, rendering most of the computers unusable. Therefore a discrete model is not viable for this application.

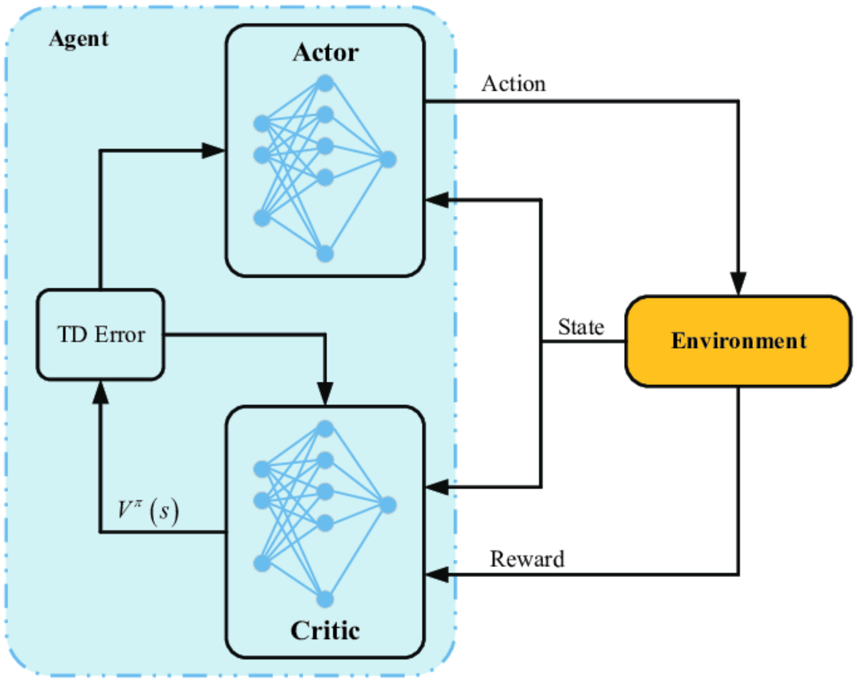
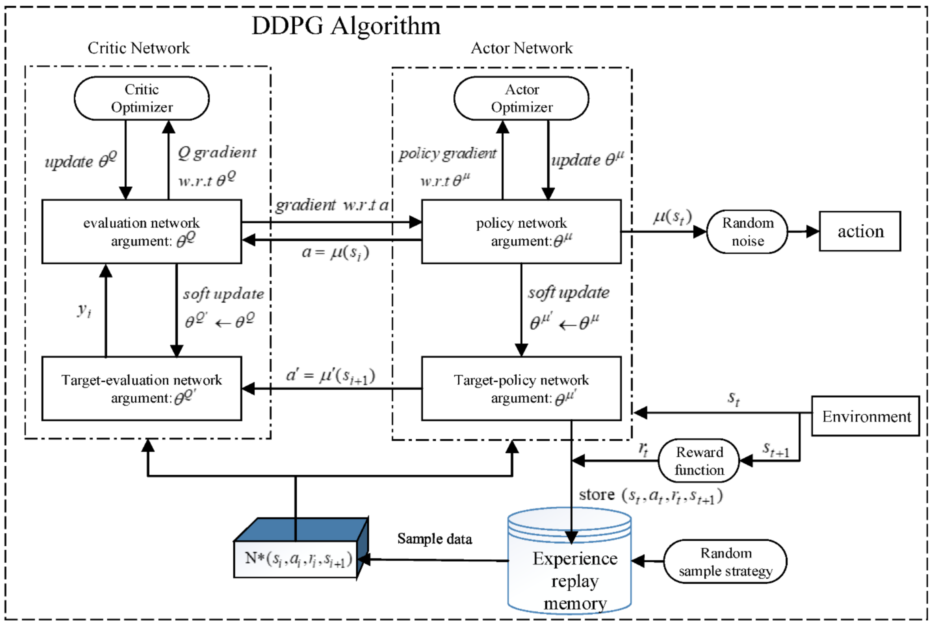


Figure 15: Actor Critic Algorithm (Source: [11])

With a little modification the Actor Critic algorithm could have been used as a continuous model, however based on my research it was not the optimal solution, but one of it’s variant, DDPG.


DDPG Algorithm
Critic Network
Actor Network
Critic Optimizer
Actor Optimizer
update θ^Q
Q gradient w.r.t θ^Q
policy gradient w.r.t θ^μ
update θ^μ
gradient w.r.t a
evaluation network argument: θ^Q
$a = \mu(s_i)$
policy network argument: θ^μ
$\mu(s_t)$
Random noise
action
y_i
soft update $\theta^{Q'} \leftarrow \theta^Q$
soft update $\theta^{\mu'} \leftarrow \theta^\mu$
Target-evaluation network argument: $\theta^{Q'}$
$a' = \mu'(s_{i+1})$
Target-policy network argument: $\theta^{\mu'}$
s_t
Environment
r_t
Reward function
s_{t+1}
store (s_t, a_t, r_t, s_{t+1})
Sample data
$N*(s_i, a_i, r_i, s_{i+1})$
Experience replay memory
Random sample strategy
Figure 16: DDPG Algorithm (Source: [12])

The DDPG (Deep Deterministic Policy Gradient)is an algorithm used in reinforcement learning that combines deep neural networks with the policy gradient methods to tackle continuous action spaces. It is particularly well-suited for problems with continuous control, such as robotics or spacecraft piloting. DDPG operates by simultaneously learning a value function and a policy. The value function estimates the expected cumulative reward for a given state-action pair, while the policy function determines the best action to take in a given state. The key idea behind DDPG is to leverage the advantages of deep neural networks to approximate the value and policy functions, enabling the algorithm to handle high-dimensional state and action spaces. By employing an actor-critic approach, where the actor network represents the policy and the critic network represents the value function, DDPG learns to maximize the expected cumulative reward by updating the actor's policy based on the critic's evaluation. Additionally, to address the issue of exploration in continuous action spaces, DDPG introduces an exploration strategy based on adding noise to the actor's chosen actions. Through iterative learning and adjustment of the neural network parameters, DDPG aims to find an optimal policy that maximizes the expected reward in continuous control tasks.

DDPG and A2C are reinforcement learning algorithms with distinct strengths. DDPG is particularly advantageous in scenarios with continuous action spaces, where it uses a deterministic policy to output continuous actions. It addresses exploration in such spaces by adding noise to the actor's chosen actions during training. DDPG also benefits from experience replay, allowing it to learn from diverse past experiences and improve sample efficiency. Being an off-policy algorithm, DDPG can learn from older policy data, further enhancing its efficiency. On the other hand, A2C converges faster and is simpler to implement, making it suitable for a range of problems, especially those with discrete action spaces. Ultimately, the choice between DDPG and A2C depends on the specific problem, action space characteristics, and trade-offs between sample efficiency, convergence speed, and implementation complexity.

In my case the convergence rate is not as important as having a memory, since in the case of a vehicle, it’s past actions are as important as it’s current state, so the AI can learn paths, not just actions.

With the help of Antocapp’s (Antonio Cappiello) Paperspace tutorial[13], which I mainly followed, I was manage to create a working Neural Network. Due to my lack of experience in this field and the time I had to research, I mainly modified the code, not written it from zero.

During the manual testing phase of the software, I have manged to make a few decent screenshots of the AI learning to land, the screenshots are not from the same runs, they were made at different times.

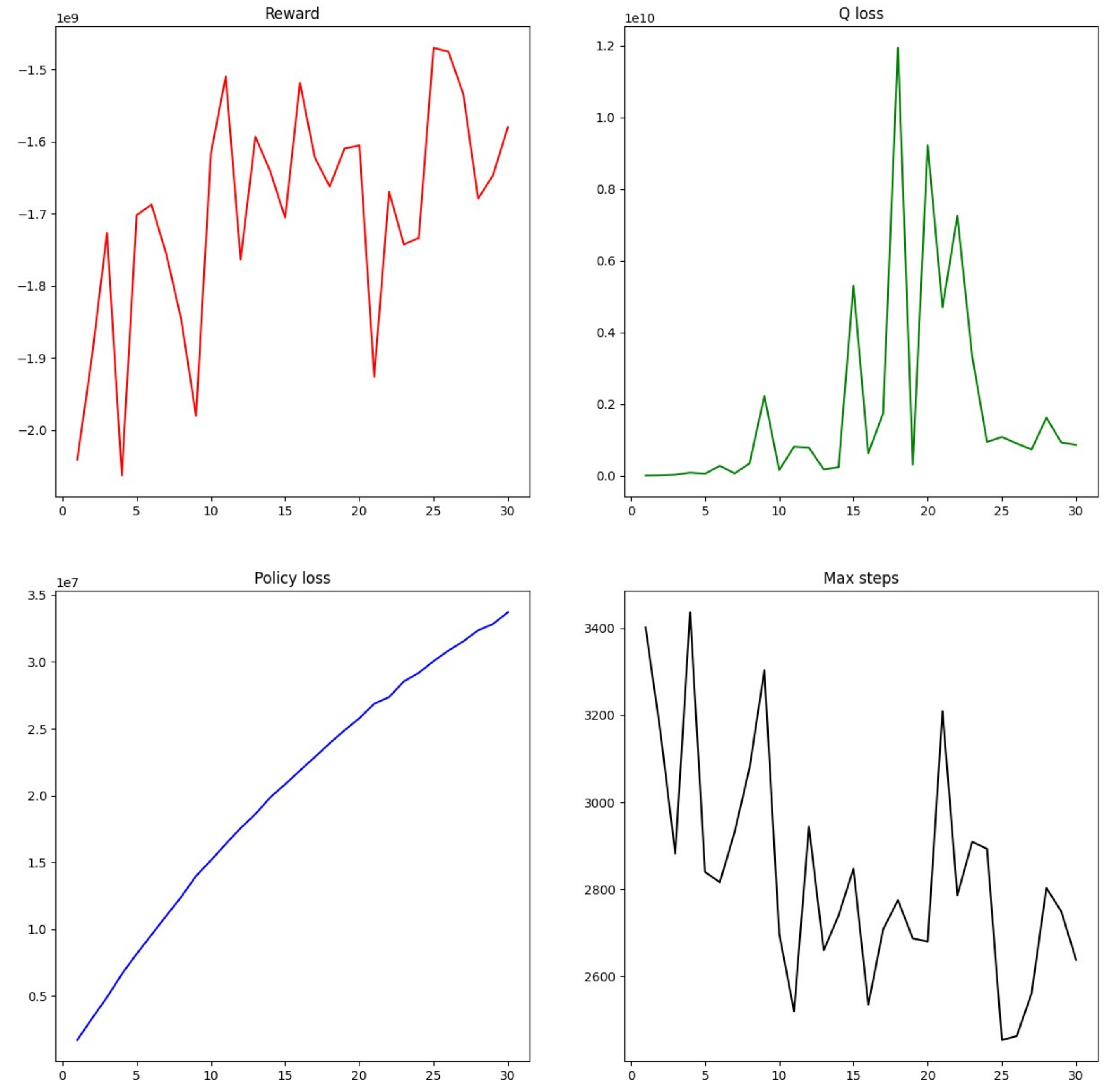


Figure 17: Different figures printed out to follow the changes in the neural network

I consider this a lucky run, since if you look at the rewards chart, you can see that it already started to converge towards positive numbers, while the amount of steps, which are the amount of actions the AI took, significantly decreased.

I could talk about the Q loss and the Policy loss charts, however only 30 runs are not enough to be able to say anything, although with more runs the Policy loss should be a less steep curve and the Q loss function will start to flatten. Here is an average run:

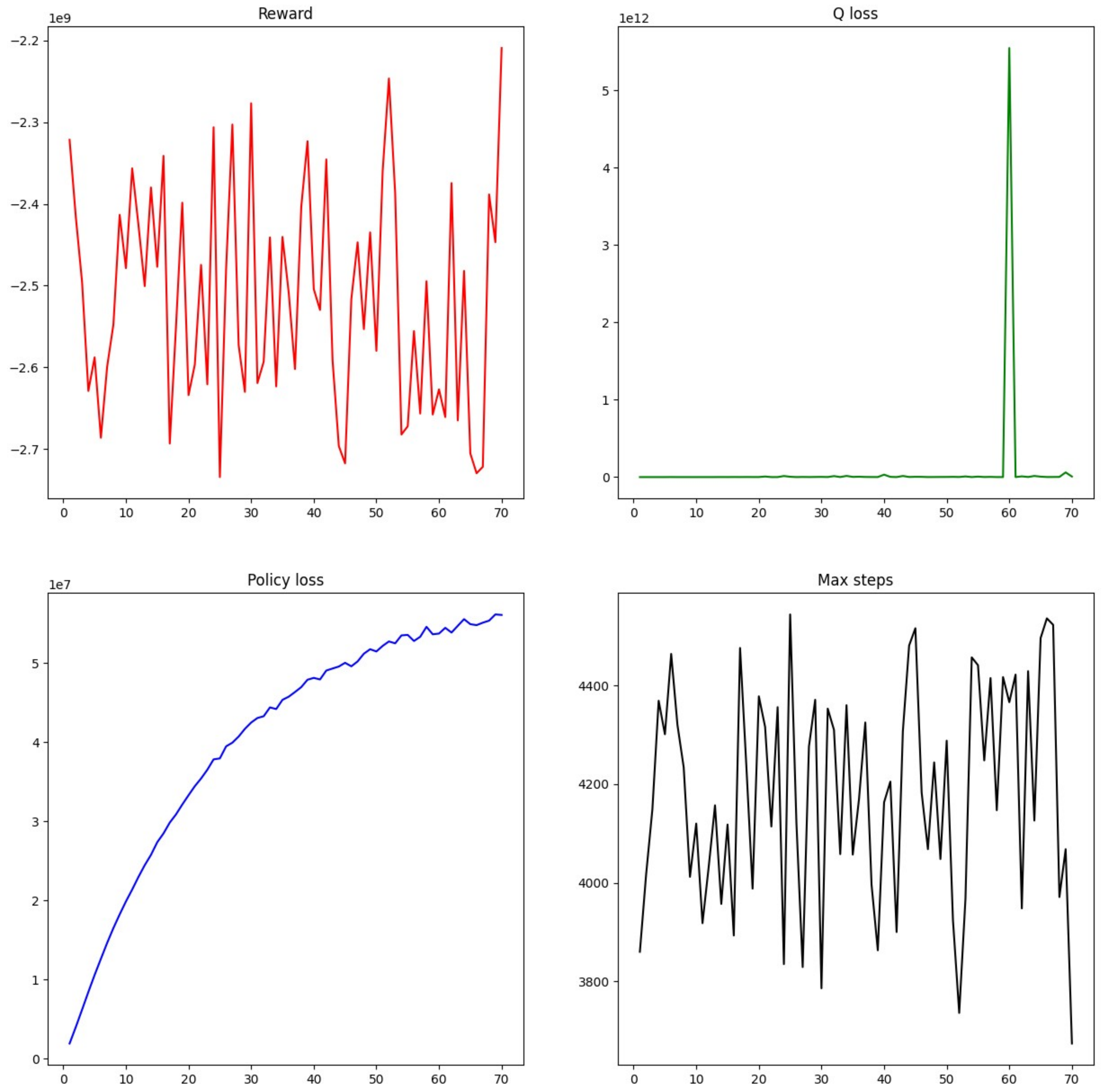
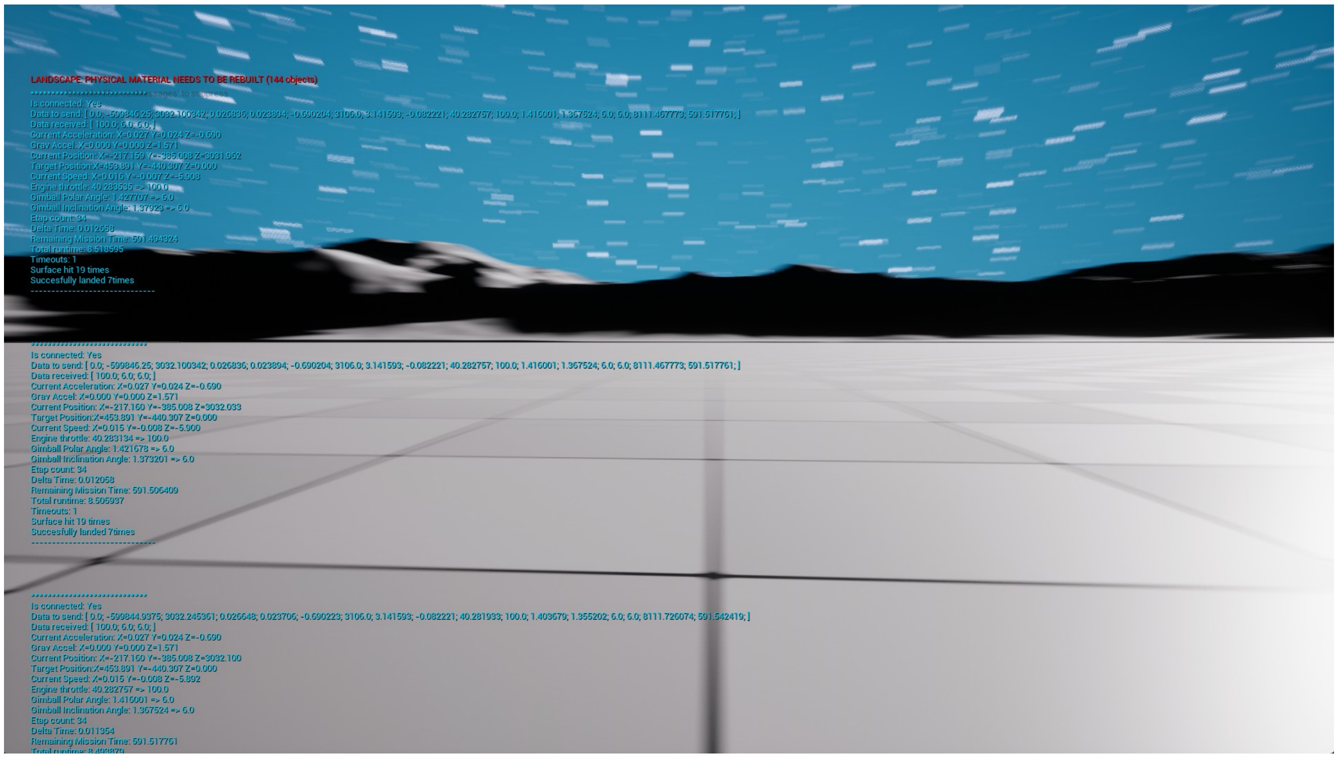


Figure 18: Graphs from the warming up period

Note that there is basically no convergence so far, this is due to one of the setting, so during the first 100 etaps are so called warmup rounds without proper training. This value could be less, but based on my experimenting, it is not wise to train without at least 50 warmup etaps. Afterwards the convergence could be seen, if I could manage to reach at least 500 runs, that would appear on the charts.

While running the training, there is an output on the screen, showing different statistics or debug data regarging the current etap and all of the runs:



```
LANDSCAPE: PHYSICAL MATERIAL NEEDS TO BE REBUILT (144 objects)
******************************ssages' to suppress
Is connected: Yes
Data to send: [ 0.0; -599846.25; 3032.100342; 0.026836; 0.023894; -0.690204; 3106.0; 3.141593; -0.082221; 40.282757; 100.0; 1.416001; 1.367524; 6.0; 6.0; 8111.467773; 591.517761; ]
Data received: [ 100.0; 6.0; 6.0; ]
Current Acceleration: X=0.027 Y=0.024 Z=-0.690
Grav Accel: X=0.000 Y=0.000 Z=1.571
Current Position: X=-217.159 Y=-385.008 Z=3031.962
Target Position:X=453.891 Y=-440.307 Z=0.000
Current Speed: X=0.016 Y=-0.007 Z=-5.908
Engine throttle: 40.283535 => 100.0
Gimball Polar Angle: 1.427707 => 6.0
Gimball Inclination Angle: 1.37923 => 6.0
Etap count: 34
Delta Time: 0.012658
Remaining Mission Time: 591.494324
Total runtime: 8.518595
Timeouts: 1
Surface hit 19 times
Succesfully landed 7times
------------------------------

******************************
Is connected: Yes
Data to send: [ 0.0; -599846.25; 3032.100342; 0.026836; 0.023894; -0.690204; 3106.0; 3.141593; -0.082221; 40.282757; 100.0; 1.416001; 1.367524; 6.0; 6.0; 8111.467773; 591.517761; ]
Data received: [ 100.0; 6.0; 6.0; ]
Current Acceleration: X=0.027 Y=0.024 Z=-0.690
Grav Accel: X=0.000 Y=0.000 Z=1.571
Current Position: X=-217.160 Y=-385.008 Z=3032.033
Target Position:X=453.891 Y=-440.307 Z=0.000
Current Speed: X=0.015 Y=-0.008 Z=-5.900
Engine throttle: 40.283134 => 100.0
Gimball Polar Angle: 1.421678 => 6.0
Gimball Inclination Angle: 1.373201 => 6.0
Etap count: 34
Delta Time: 0.012058
Remaining Mission Time: 591.506409
Total runtime: 8.505937
Timeouts: 1
Surface hit 19 times
Succesfully landed 7times
------------------------------

******************************
Is connected: Yes
Data to send: [ 0.0; -599844.9375; 3032.245361; 0.026648; 0.023706; -0.690223; 3106.0; 3.141593; -0.082221; 40.281933; 100.0; 1.403679; 1.355202; 6.0; 6.0; 8111.726074; 591.542419; ]
Data received: [ 100.0; 6.0; 6.0; ]
Current Acceleration: X=0.027 Y=0.024 Z=-0.690
Grav Accel: X=0.000 Y=0.000 Z=1.571
Current Position: X=-217.160 Y=-385.008 Z=3032.100
Target Position:X=453.891 Y=-440.307 Z=0.000
Current Speed: X=0.015 Y=-0.008 Z=-5.892
Engine throttle: 40.282757 => 100.0
Gimball Polar Angle: 1.416001 => 6.0
Gimball Inclination Angle: 1.367524 => 6.0
Etap count: 34
Delta Time: 0.011354
Remaining Mission Time: 591.517761
Total runtime: 8.493879
```

Figure 19: Ingame debug informations

Currently the software does not count when the Lander reaches the upper limit, 5500 meters, but it counts all the times it ran out of time, when it hit the surface and the times when the velocity at that moment was slower than the given threshold.

3.3 Simulation – Neural Network Integration

On paper Unreal Engine has built in support for machine learning, however this is still in beta, and could have potential bugs, so I have decided not to risk it and instead separate the environment and the neural network. This way they can be run on different computers, debugged separately.

To ensure that the two modules are communicating, I have implemented a simple socket based communication, main reason behind this was that it lacked complex functions and is one of the most lightweight protocols, and when every little resource counts, I will take the most resource optimal solution.

The LanderEnvironment in the python code handles the integration on the neural network's side, with decoding the current state of the environment, providing example actions and modifying the actions gotten from the neural network, since it returns with floating points between -1 and 1 while the environment requires values between 0 and 100 for the throttle and values between -6 and 6 for the TVC.

While it would be possible to calculate the reward from the gotten states, it's better if those are calculated from the actual values, which are in the environment.

I have created two separate functions to calculate the reward for the actions, these are currently set to punish for mistakes. The getScore function determines the reward for a single step, this is used during the simulation. The getReward function is used when an etap has ended, it calculates a score based on the final state.

3.4 Additional Functions

In order to make the environment function properly, these additional functions were implemented, not including the ones required by the Unreal Engine:

missionTimeOver: in order to avoid infinitely long missions, a maximum timeframe, 600 seconds, was set and this function handles the end of the etap, in case the timer runs out

addGaussianNoise: this function helps to add a noise to the sensors, default deviance is 0.01

roundtoFloatPrecision: this function was meant to help to simulate the accuracy of the different sensors

roundToGivenDecimal: this function will round a floating point to the given decimal value, default decimal is 4

cartesianToPolar, **polarToCartesian**: coordinate conversion between the two mainly used coordinate system

boolToString, **floatArrayToString**: simple functions to help the readability of the logs

getFormattedLog: returns with an easy to read string which could be written to a file or just sent to the standard output.

3.5 Development Roadmap

While this software could be used to train an AI to flight or land the Apollo Lunar Lander, which was the original goal, it has many potential in it. In the future the following features are planned to be add:

- Bipropellant engines: Nowadays the most commonly used engines are bipropellant engines where the mixing ratio can be set

- Additional engine types: In order to properly simulate a satellite or an ICBM, non liquid fuel engines must be supported.

- Staging: Currently only single stage spacecrafts are supported, while all of the currently used launch vehicles are multistage rockets.

- Proper Atmospheric model, with drag: Current atmospheric model is a simple statistical model, without winds or other weather effect, and the code should be able to calculate the deceleration and the heating from the air resistance

- Launch to orbit: In order to have a fully autonomous spacecraft, it must be able to learn how to leave a planet in a controlled way, since just accelerating will result in getting away from the planet, while starting to orbiting around one is more difficult. This part contains a proper interface for machine learning.

- Simulation of an orbiting spacecraft: Determining if a spacecraft is orbiting around a body is not that difficult, simulating it going around a planet while other objects are affecting it is another topic, which is a necessary next step.

- Course correction: In case of unstable orbits, the spacecraft must be able to change its course, and this part contains an interface for machine learning.

- Additional space objects: During the last 30 years humanity has created an insane amount of space objects (satellites, debris, etc.) around the Earth. Collision with any of this will have a devastating effect, therefore the spacecraft must be able to identify and avoid any of these. A few of these should be different space stations or space vehicles.

- Docking: This should enable fully automated restock missions for any space station, or could help with docking situations similar to the Lunar Module and Command Module connection during the Apollo missions.

- Interplanetary spaceflight: So far there was only one celestial body and an infinite sky. In order to simulate colonization of the Solar System, adding this feature is a must. Moreover this would enable a full Moon mission training or simulation.

- Spherical harmonics based calculations for inhomogeneous celestial bodies: While for landing or launching the current gravitational model is perfect, for precise simulation of any orbiting space object, a model where the inhomogenity of the celestial body is not ignored is necessary, since these imperfections cause perturbations in the orbit, which effects its stability.

- Image recognition based landing site targeting or positioning: The current simulation uses a ‘radio signal’ based landing site targeting, which means that the spacecraft continuously gets a radio signal from the landing site, which is not available in the real world, except a few cases. To prepare the AI for such, additional cameras and imagery is needed.

- Additional terrain and material types: Currently there is only rock as the material, while there should be at least a dozen more, and with that different terrains can be generated, to increase the variability of the environment, thus making the AI more versatile.

3.6 Notes

THIS PAGE IS INTENTIONALLY LEFT BLANK

4. Conclusion

“I choose to land on the Moon. I choose to land on the Moon... I choose to land on the Moon in this bachelor thesis and do the other things, not because they are easy, but because they are hard; because that goal will serve to organize and measure the best of my energies and skills, because that challenge is one that I am willing to accept, one I am unwilling to postpone, and one I intend to win, and the others, too.”[1] [14]

The development process of the simulation environment involved implementation of different realistic physical models, each were optimized with different goals in mind, few were more resource effective, few were more realistic. It took a long research and calculations to find a model which were somewhere in the middle – realistic enough for a simulation while not using up all of the available resources. This lead to some cut corners, but only after the calculations showed that these will not compromise the final result. The second part of the development was the AI, where choosing the perfect algorithm seemed to be the hardest part, then setting the proper weights for each value during the teaching phase, in order to ensure that the AI will learn how to land at one point. Although, due to the long training time requirement and close deadlines, I did not manage to produce a proper landing, the fact that it is possible for the AI to learn to land the Apollo Lunar Lander was proven during the tests.

While this thesis demonstrates promising results, there are avenues for further exploration and improvement. Future work should focus on expanding the simulation environment to incorporate additional lunar lander models and mission scenarios, enabling comprehensive training for a variety of spacecraft. Additionally, fine-tuning the AI training algorithms and exploring hybrid approaches that combine reinforcement learning with other techniques could enhance the AI agent's adaptability to varying lunar surface conditions. Furthermore, conducting user studies to evaluate the effectiveness of the simulation environment as an educational tool would be valuable for its implementation in training programs.

In conclusion, this bachelor thesis successfully developed a simulation environment to teach an AI to land with the Apollo Lunar Lander on the Moon. Through a robust methodology and rigorous experimentation, I managed to demonstrate that it is possible to teach an AI to land with a space vehicle. This work contributes to the fields of AI, lunar landing simulations, and space exploration, with the potential to revolutionize future lunar missions and astronaut training programs. The development of advanced AI-driven landing systems holds immense promise for enhancing the safety, precision, and efficiency of space exploration endeavors. As we continue to unlock the capabilities of AI and simulations, we pave the way for new frontiers in space technology and human exploration.

1 Reference to John F. Kennedy’s speech, We are going to the Moon, which started the space race to the Moon.

THIS PAGE IS INTENTIONALLY LEFT BLANK

5. References

[1] https://hackr.io/blog/unity-vs-unreal-engine#:~:text=With%20Unity%2C%20game%20developers%20can,fidelity%20and%20photorealistic%203D%20graphics – 2023/05/20
[2] https://blog.udemy.com/unity-vs-unreal-which-game-engine-is-best-for-you/ - 2023/05/20
[3] https://gamedevacademy.org/unity-vs-unreal/ - 2023/05/20
[4] F.W. Leslie; C.G. Justus: The NASA Marshall Space Flight Center Earth Global Reference Atmospheric Model—2010 Version, NASA/TM—2011–216467, Marshall Space Flight Center, Alabama, United States: National Aeronautics and Space Administration, 2011
[5] J. M. Picone, A. E. Hedin, D. P. Drob, A. C. Aikin: NRLMSISE-00 empirical model of the atmosphere: Statistical comparisons and scientific issues, Journal of Geophysical Research (Space Physics), Volume 107, Issue A12, 2002
[6] George P. Sutton, Oscar Biblarz: Rocket Propulsion Elements, Wiley, 2001,[751],ISBN978047132642
[7] Jr. W. R. Hammock, E. C. Curie, A. E. Fisher: Apollo Experience Report: Descent propulsion system, NASA-TN-D-7143, Lyndon B. Johnson Space Center Houston, Texas, United States: National Aeronautics and Space Administration, 1973
[8] W. Bartlett, Z. D. Kirkland, R. W. Polifka, J. C. Smithson, G. L. Spencer: Apollo spacecraft liquid primary propulsion systems, NASA-TM-X-65048, Lyndon B. Johnson Space Center Houston, Texas, United States: National Aeronautics and Space Administration, 1966
[9] Unknown Author: Liquid rocket engine nozzles, NASA-SP-8120, 1976
[10] https://medium.com/analytics-vidhya/what-is-machine-learning-446d570cadab - 2023/05/22
[11] Hoang Thi Huong Giang, Tran Nhut Khai Hoan, Pham Duy Thanh, Insoo Koo: Hybrid NOMA/OMA-based Dynamic Power Allocation Scheme Using Deep Reinforcement Learning in 5G Network, Applied Sciences, Volume 10, Issue 12, 2020
[12] Baosu Guo, Yu-Xiu Zhang, Weiquing Zheng, Du Yanhua: An Autonomous Path Planning Model for Unmanned Ships Based on Deep Reinforcement Learning, Sensors, Volume 20, Issue 2, 2020
[13] https://blog.paperspace.com/physics-control-tasks-with-deep-reinforcement-learning/ - 2023/05/22
[14] https://www.jfklibrary.org/learn/about-jfk/historic-speeches/address-at-rice-university-on-the-nations-space-effort - 2023/05/22