People‘s Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research
University of Mohamed Khider - Biskra
Faculty of Exact Sciences and Sciences of Nature and Life
Computer Science Department

N° d’ordre : IA_Start Up_08/M2/2024

Thesis

Submitted in fulfillment of the requirements for the Master’s degree in

Computer Science

Speciality: Artificial Intelligence

Autonomous Rocket Landing Control using AI Simulations for Improved Space Exploration

---

Presented By:

Bouchana Hicham

Graduating on June 23, 2024, before the jury composed of:

Bourekkache Samir	MCA	President
Aloui Ahmed	MCA	Supervisor
Hoadjli Hadia	MAA	Examiner
Zouai Meftah	MCB	Co-Supervisor

Academic Year 2024/2025

ملخص

يظل الهبوط الآمن والدقيق للصواريخ تحديًا بالغ الأهمية في استكشاف الفضاء. تستكشف هذه الأطروحة تطبيق محاكاة الذكاء الاصطناعي لتطوير نظام تحكم ذاتي لهبوط الصواريخ.

داخل محرك Unity وباستعمال أداةML-Agents ، يتم تصميم بيئة هبوط واقعية للصواريخ. تدمج هذه البيئة تأثيرات الجاذبية والسحب الجوي والقوى الفيزيائية الأخرى ذات الصلة التي تؤثر على رحلة الصواريخ. يتم بعد ذلك تطوير برنامج مخصص لترجمة قرارات الوكيل من خوارزمية التعلم المعزز إلى إجراءات تحكم فعلية للصاروخ المحاكى. قد تشمل هذه الإجراءات ضبط دفع المحرك، والتحكم في توجيه الدفع، أو نشر أرجل الهبوط.

يتم تنفيذ خوارزمية تحسين السياسة التقدمية (PPO) لتدريب الوكيل .(The Agent) يتعلم الوكيل التحكم في الصاروخ من خلال تفاعلات التجربة والخطأ داخل البيئة المحاكية، مع مراعاة عوامل مثل دقة الهبوط وكفاءة والقدرة على مواجهة الاضطرابات البيئية.

تُظهر النتائج أن نظام التحكم المعتمد على الذكاء الاصطناعي المقترح، والمقترن بمحاكاة الفيزياء الحقيقية، يمكنه تحقيق هبوط ناجح في البيئات المحاكية. كما تحدد الأبحاث القيود الحالية للنهج المتبع، مثل تعقيد سيناريوهات الهبوط في العالم الحقيقي. تشمل اتجاهات العمل المستقبلي دمج بيانات العالم الحقيقي واستكشاف نماذج صواريخ أكثر تعقيدًا للحصول على نظام تحكم أكثر قوة وقابلية للتطبيق. تسهم هذه الأبحاث في تقدم تقنية الهبوط الذاتي للصواريخ،

مما يمهد الطريق لمهام استكشاف الفضاء أكثر كفاءة وموثوقية.

Résumé

L'atterrissage sûr et précis des fusées reste un défi crucial dans l'exploration spatiale. Cette thèse explore l'application des simulations d'intelligence artificielle (IA) pour développer un système autonome de contrôle d'atterrissage de fusée. Au sein du moteur Unity et de la plateforme ML-Agents, un environnement d'atterrissage de fusée réaliste est conçu. Cet environnement intègre les effets de la gravité, de la traînée atmosphérique et d'autres forces physiques pertinentes qui influencent le vol de la fusée. Un script personnalisé est ensuite développé pour transférer les décisions de l'agent de l'algorithme d'apprentissage par renforcement aux actions de contrôle réelles pour la fusée simulée. Ces actions peuvent inclure l'ajustement de la poussée du moteur, la manipulation de la vectorisation de poussée ou le déploiement de jambes d'atterrissage. L'algorithme d'apprentissage par renforcement Proximal Policy Optimization (PPO) est implémenté pour former l'agent. L'agent apprend à contrôler la fusée grâce à des interactions par essais et erreurs dans l'environnement simulé, en tenant compte de facteurs tels que la précision de l'atterrissage, le rendement énergétique et la robustesse aux perturbations environnementales. Les résultats démontrent que le système de contrôle proposé basé sur l'IA, associé à la simulation physique réelle, peut réaliser des atterrissages réussis dans des environnements simulés. La recherche a également identifié les limites de l'approche actuelle, telles que la complexité des scénarios d'atterrissage réels. Les orientations de travail futures incluent l'intégration de données du monde réel et l'exploration de modèles de fusées plus complexes pour un système de contrôle plus robuste et transférable. Cette recherche contribue à l'avancement de la technologie d'atterrissage de fusée autonome, ouvrant la voie à des missions d'exploration spatiale plus fiables.

Abstract

Safe and precise rocket landing remains a critical challenge in space exploration. This thesis explores the application of Artificial Intelligence (AI) simulations for developing an autonomous rocket landing control system.

Within the Unity engine and the ML-Agents platform, a realistic rocket landing environment is designed. This environment incorporates the effects of gravity, atmospheric drag, and other relevant physical forces that influence rocket flight. A custom script is then developed to translate the agent's decisions from the reinforcement learning algorithm into actual control actions for the simulated rocket. These actions might include adjusting engine thrust, manipulating thrust vectoring, or deploying landing legs.

The Proximal Policy Optimization (PPO) reinforcement learning algorithm is implemented to train the agent. The agent learns to control the rocket through trial and error interactions within the simulated environment, considering factors like landing accuracy, and robustness to environmental disturbances.

The results demonstrate that the proposed AI-based control system, coupled with real physics simulation, can achieve successful landings in simulated environments. The research also identifies limitations of the current approach, such as the complexity of real-world landing scenarios. Future work directions include incorporating real-world data and exploring more complex rocket models for a more robust and transferable control system. This research contributes to the advancement of autonomous rocket landing technology, paving the way for more efficient and reliable space exploration missions.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to God, whose blessings and guidance have been a constant source of strength and inspiration throughout this journey. His wisdom and grace have provided me with the perseverance and insight needed to complete this thesis.

I am profoundly grateful and would like to express my sincere gratitude to my supervisor, ***Dr. Ahmed ALOUI*** and my Co-supervisor ***Dr. Meftah ZOUAI***, for their invaluable guidance, support, and encouragement throughout this thesis project. Their expertise in the field of Artificial Intelligence was instrumental in shaping the direction of this research and providing insightful feedback on my work.

I am also grateful to the members of my thesis committee, for their willingness to dedicate their time and expertise to reviewing my thesis and offering valuable suggestions.

I am particularly grateful to my fellow computer science colleagues, especially those specializing in artificial intelligence. Their stimulating discussions and collaborative spirit fostered a supportive environment throughout my research journey. Their willingness to share ideas and expertise significantly enriched my understanding of the subject matter and propelled my progress forward.

I also want to acknowledge myself for the countless hours of dedication, perseverance, and intellectual curiosity that fueled my research and writing. This thesis would not have been possible without the self-discipline and commitment I brought to the task.

Finally, I want to express my heartfelt gratitude to my family and friends for their unwavering love, encouragement, and understanding during the challenging and rewarding experience of completing this thesis. Their belief in me fueled my motivation and helped me persevere through difficult moments.

Dédicaces

This thesis is dedicated to:

My beloved parents, My Dad ***Bouchana Belkacem*** *and My Mom* ***Bouchekout Halima****, for their unconditional love, endless support, and unwavering faith in me. Their encouragement and sacrifices have been my foundation and driving force.*

My family, for their constant encouragement, understanding, and support, which have been invaluable throughout my academic journey.

To myself, this thesis is a testament to the power of dedication and self-belief. The countless hours invested, the challenges overcome, and the intellectual growth experienced throughout this journey have been immensely rewarding. This work is a reflection of my unwavering commitment to pursuing knowledge and pushing my boundaries.

And to my incredible friends and classmates, Wail Zerarka, Ilyes Hazmani, Djellab Mohamed Ayhem, and Othmane Amani, your unwavering support and friendship have been a source of immense strength throughout this journey. Your positive and encouraging spirits provided a much-needed escape and a wellspring of encouragement. To all my fellow classmates (and many more that I have not mentioned here), your camaraderie and shared experiences throughout this academic journey have also been invaluable.

Thank you all for being a part of this journey with me.

Hicham BOUCHANA

Contents

List of Figures

List of Tables

General introduction

Autonomous rocket landing systems are crucial in space exploration for their ability to reduce human error, lower operational costs, and significantly improve mission success rates. Traditional landing methods, which often involve human intervention, can be complex, expensive, and prone to error. Autonomous systems, on the other hand, offer the potential for precise handling of harsh landing environments [1]. However, several key challenges must be addressed for successful rocket landings.

High-speed descent is one of the major challenges, as rockets re-enter the atmosphere at speeds exceeding Mach 5 [2], necessitating precise control maneuvers to decelerate safely and achieve a controlled landing. Uncertain environmental conditions, such as wind shear, atmospheric variations, and uneven terrain, also pose significant risks to the landing process [3, 4].. Additionally, limited fuel resources make fuel efficiency critical during landing, as unnecessary fuel expenditure can compromise the mission. Thermal management is another challenge [5], as the extreme heat generated during re-entry requires rockets to be designed to withstand high temperatures while maintaining controllability. Reliable and accurate sensor data is essential for real-time decision-making during landing, but factors like noise and latency can affect data accuracy, making control more challenging [6].

AI is emerging as a powerful tool to address these challenges in space exploration. AI applications in data analysis allow for the efficient processing of massive amounts of data generated by spacecraft, enabling the identification of patterns, anomalies, and insights that might escape human analysis. Autonomous systems and robotics powered by AI can perform tasks such as exploration, sample collection, and spacecraft maintenance [7], reducing reliance on human control and enabling extended missions in hazardous environments. AI also aids in decision-making [8] and mission planning by analyzing complex data and making real-time decisions, optimizing trajectories, fuel usage [9, 10], and overall mission efficiency. Moreover, AI-powered training and support systems can create interactive simulations for astronauts [11], monitor their health and well-being, and provide real-time support during missions [12].

This research aims to develop an AI-based control system for autonomous rocket landing using Proximal Policy Optimization (PPO) [13] within a physics-based simulation environment. The primary objective is to implement and train the AI control system in Unity, focusing on creating a reward function that promotes efficient landings. The system's robustness and adaptability will be evaluated under various initial conditions, including different starting heights and rotations, and with different control strategies such as Thrust Vector Control (TVC) [14] alone and in combination with Cold Gas Thrusters (CGT) [15, 16]. Additionally, the system will be tested with both free and limited TVC capabilities to assess its performance with different hardware limitations.

A comprehensive evaluation methodology will be developed to assess the AI system across multiple metrics, including landing success rate, precision, and trajectory analysis. The success rate will be measured by the percentage of simulations resulting in controlled landings within a designated zone. Precision will be evaluated based on the rocket's final position and velocity at touchdown, while trajectory analysis will focus on the AI's ability to generate optimal and safe landing paths.

The thesis is structured into four main chapters. The first chapter introduces the use of reinforcement learning (RL) for rocket landing control, detailing the challenges, benefits, and the choice of PPO as the RL algorithm. The second chapter provides foundational knowledge on rocket science, including propulsion principles and control techniques such as TVC and CGT, and discusses real-world examples like SpaceX's Falcon 9 and Starship landings. The third chapter details the development of the simulation environment in Unity, explaining the tools, scripts, and RL training approach. The final chapter presents the outcomes of the RL training, evaluates the AI system's landing performance under various conditions, and discusses limitations and areas for further improvement.

Chapter 1

Leveraging Reinforcement Learning for Rocket Landing

1.1 Introduction

Machine learning is revolutionizing how computers interact with the world [17]. But what if, instead of being spoon-fed information, a program could learn by doing? Reinforcement learning (RL) is a powerful technique that empowers machines to tackle challenges through trial and error [18], just like a child learning to walk. In this approach, an agent interacts with its environment, taking actions and receiving rewards or penalties in response. Over time, the agent learns which actions lead to success, constantly adapting and refining its behavior. This ability to learn autonomously makes RL a game-changer for tasks like robotics control, resource management in complex systems, and even mastering complex simulations.

1.1.1 Motivation for Applying Reinforcement Learning to Rocket Landing Control

Traditional control methods for rocket landing rely on pre-programmed flight paths and control laws. These methods require extensive engineering expertise to account for various flight conditions and environmental factors. However, real-world scenarios often involve complexities and uncertainties that can be challenging to predict and model precisely.

Reinforcement Learning (RL) offers a compelling alternative approach for autonomous rocket landing control [19]. Here's why:

- **Adaptability:** RL agents can learn to adapt their control strategies in response to changing environments and unexpected situations. This makes them well-suited for handling the dynamic and unpredictable nature of rocket landings.
- **Learning from Experience:** Through trial and error within a simulated environment, RL agents can learn optimal control strategies without the need for extensive manual programming. This can significantly reduce development time and effort.
- **Continuous Improvement:** As an RL agent accumulates experience, it can continuously refine its control policies, potentially surpassing the performance of pre-programmed approaches.

1.2 Fundamentals of Reinforcement Learning

This section provides a foundational understanding of Reinforcement Learning (RL), focusing on the core concepts that guide an agent's learning process for effective decision-making.

1.2.1 Core Concepts: Learning from Interaction

Reinforcement Learning deals with an **agent** interacting with its **environment** [20]. Here's a breakdown of these key elements :

1.2.1.1 The Agent and the Environment

- **Agent:** The agent represents the entity that learns and makes decisions within the RL framework. In our case, the agent is the **rocket landing control system**.

- **Environment:** The environment encompasses everything the agent interacts with. For us, this is the **rocket simulator** that includes the rocket itself, the landing pad, and the laws of physics that govern their motion.

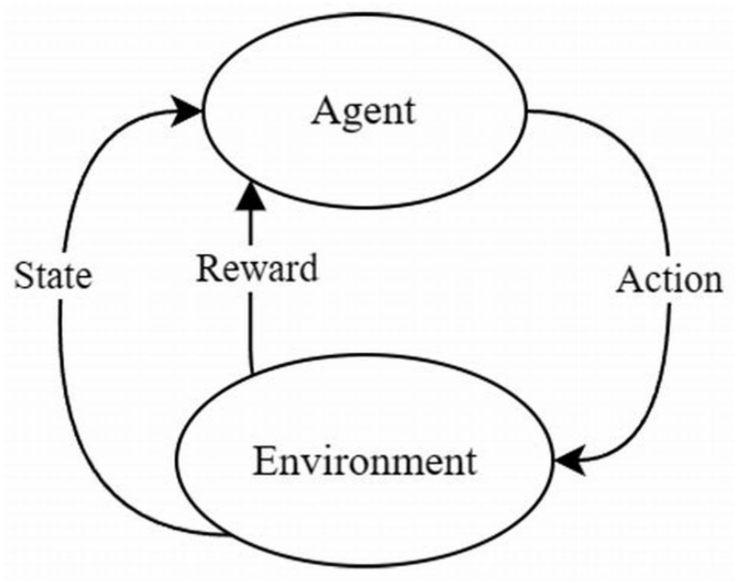


Figure 1.1: "The Agent-Environment Dance" Reinforcement Learning Model

The agent and environment engage in a continuous cycle of interaction:

1.2.1.2 Taking Actions and Receiving Rewards

The agent **perceives the environment** through observations (e.g., rocket position, velocity) and takes **actions** (e.g., adjusting thrust vector control). The environment responds to these actions, and the agent receives **rewards** that signal the desirability of the chosen action. High rewards indicate favorable outcomes (e.g., getting closer to the landing pad), while low rewards or penalties might be given for undesirable outcomes (e.g., tilting excessively).

1.2.1.3 Learning Through Trial and Error

Through repeated interactions and by observing the rewards it receives, the agent gradually learns to **associate actions with their corresponding rewards**. This trial-and-error process allows the agent to refine its decision-making strategy over time.

1.2.1.4 Learning Objective

The ultimate goal of the agent is to learn a policy that maps observations from the environment to actions that consistently yield high rewards. This policy essentially represents the agent's control strategy for achieving its objective (e.g., landing the rocket successfully).

1.2.2 Key Components of Reinforcement Learning

Beyond the agent-environment interaction, several key components play a crucial role in RL:

1.2.2.1 States

At each point in time, the environment can be in a specific **state**. This state captures a snapshot of the relevant aspects of the environment that influence the agent's decision-making. In our scenario, the state might include the rocket's position, velocity, orientation, and the distance to the landing pad.

1.2.2.2 Actions

The agent has a repertoire of possible actions it can take within the environment. These actions can be discrete (e.g., choosing a specific direction to thrust) or continuous (e.g., adjusting the thrust vector angle by a certain degree).

1.2.2.3 Rewards

Rewards act as the **guiding signal** for the agent's learning process. They are numerical values that the agent receives after taking an action in a specific state. These rewards provide feedback on the **effectiveness** of the chosen action in achieving the overall goal.

Designing Effective Rewards:

- High rewards are given for actions that move the agent closer to its goal (e.g., getting closer to the landing pad, maintaining a stable orientation).
- Low rewards or penalties are given for undesirable actions (e.g., deviating significantly from the landing trajectory, exceeding fuel consumption [21], tilting heavily).
- The reward design should be clear, consistent, and informative to guide the agent's learning efficiently.

Rewards Shape the Agent's Policy: By consistently receiving high rewards for specific actions in certain states, the agent learns to favor those actions in the future when it encounters similar situations. This is how the reward system shapes the agent's decision-making policy over time.

1.2.2.4 Policy

The **policy** represents the core of the agent's **decision-making strategy**. It's a function that maps the perceived state of the environment (observations) to the actions the agent will take. There are various ways to represent an RL policy, such as:

- **Tables :** For simple environments with a discrete number of states and actions, a policy can be a table that maps each state to the corresponding action.
- **Neural Networks :** In complex scenarios like our case, the policy can be represented by a neural network that learns the mapping between states and actions through training.

Learning and Improving the Policy : Through trial and error, the agent interacts with the environment, receives rewards, and gradually refines its policy to select actions that consistently yield high rewards. This process allows the agent to learn an effective control strategy for achieving its objective in the given environment.

1.2.3 Benefits of Reinforcement Learning

Reinforcement Learning offers several advantages compared to traditional control methods [22]:

- **Adaptability :** RL agents can learn to adapt their control strategies to changing environments and unexpected situations. This makes them well-suited for dynamic and unpredictable real-world tasks like rocket landing.
- **Data-Driven Learning :** RL agents can learn from experience through trial and error within a simulated environment. This reduces the need for extensive manual programming and allows for continuous improvement as the agent gathers more data.
- **Handling Complex Problems :** RL can handle problems where the environment dynamics are complex and difficult to model explicitly. The agent learns through interaction, overcoming the limitations of pre-programmed control approaches.

By leveraging reinforcement learning, we can develop intelligent agents that can effectively learn control strategies for complex tasks like autonomous rocket landing.

1.3 Proximal Policy Optimization (PPO)

This section dives into the heart of our solution - utilizing Proximal Policy Optimization (PPO) [13] for effective rocket landing control.

1.3.1 Why PPO for Rocket Landing Control?

Several factors make PPO an ideal choice for our task:

1.3.1.1 Advantages of PPO for Our Specific Task

- **Continuous Control:** PPO excels at handling continuous control problems, where the agent can take a range of values for actions (e.g., adjusting thrust vector angles by varying degrees). This aligns perfectly with the nuanced control requirements of rocket landing.

- **Sample Efficiency:** PPO is known for its ability to learn effectively from a moderate amount of training data. This is beneficial as gathering real-world rocket landing data can be expensive and time-consuming.

- **Stability and Performance:** PPO offers a good balance between exploration (trying new actions) and exploitation (focusing on actions known to yield high rewards) [23]. This stability is crucial for achieving reliable and successful rocket landings.

1.3.1.2 Balancing Exploration and Stability in Continuous Control

Proximal Policy Optimization (PPO) is a powerful reinforcement learning (RL) algorithm designed for excelling in continuous control tasks. Unlike algorithms that deal with discrete action spaces (go left/right), PPO thrives in scenarios where the agent can make nuanced adjustments, like precisely controlling a rocket's thrust for a smooth landing.

Here's a deeper dive into the core concepts of PPO, assuming an understanding of RL fundamentals:

Policy as a Probability Distribution: PPO operates under the assumption that The agent's policy, denoted by $\pi(a|s)$, represents a probability distribution over possible actions (a) given the current state (s) of the environment. This distribution captures the likelihood of the agent taking specific actions in different situations.

Policy Gradient Methods and the Importance Ratio: PPO belongs to the family of policy gradient methods. These methods aim to improve the policy by taking a gradient step in the direction that is likely to increase the expected future reward. However, directly estimating this gradient can be challenging due to high variance. PPO addresses this by introducing the **importance ratio** (p) [24]:

$$\rho(a_t|a'_t) = \frac{\pi_{\text{new}}(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} \tag{1.1}$$

Here, a_t and a'_t represent the action taken at time step t according to the old (π_{old}) and new (π_{new}) policies, respectively. The importance ratio essentially measures how much more likely the new policy was to take action a_t compared to the old policy.

Clipped Policy Objective with Surrogate Function: PPO introduces a key element – **the clipped objective function** [25]. This function ensures stable policy updates by mitigating the issue of large policy changes that can occur in standard policy gradient methods.

The objective function in PPO maximizes a clipped surrogate objective (L^{CLIP}) instead of directly maximizing the expected future reward:

$$L^{\text{CLIP}} = \min(\rho(a_t \mid a'_t) \cdot A_t, \text{clip}(\rho(a_t \mid a'_t), 1-\epsilon, 1+\epsilon) \cdot A_t) \tag{1.2}$$

where:

- A_t is the advantage function, which estimates the long-term benefit of taking action a_t compared to the average action in that state.
- ϵ (epsilon) is a hyperparameter that defines the clipping range. The clipping mechanism works as follows:
  - If the importance ratio (ρ) is within the range $(1-\epsilon, 1+\epsilon)$, the advantage function (A_t) is multiplied directly. This encourages the agent to explore actions that the new policy favors more than the old policy, while staying close to the original strategy.
  - If the importance ratio falls outside the clipping range, the advantage function is clipped to a value within the range. This prevents excessively large policy updates that could destabilize the learning process.

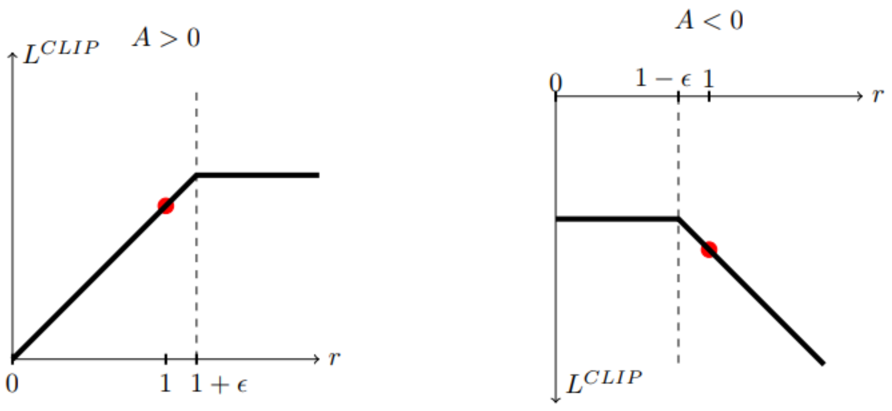
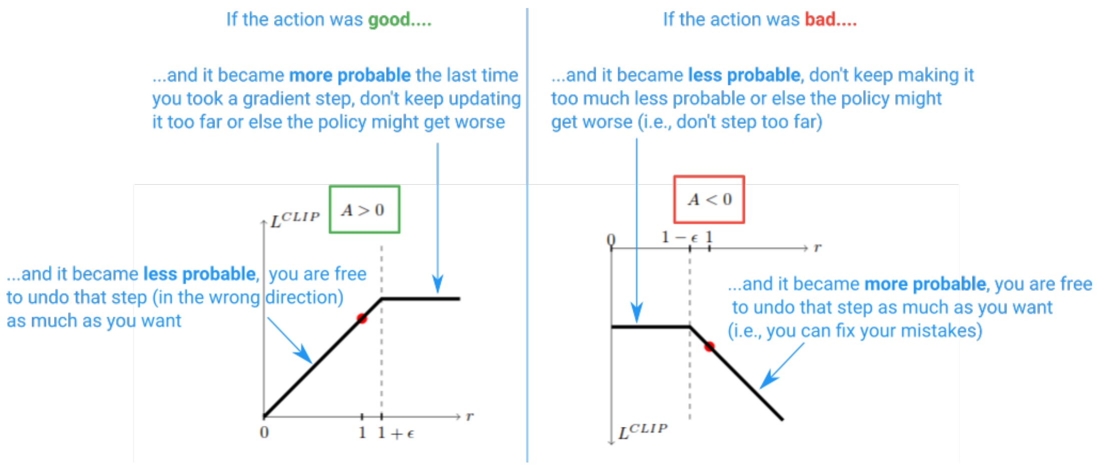


Figure 1.2: Clipping

Here is a more detailed figure that explains how things work a bit more.



If the action was good....
...and it became more probable the last time you took a gradient step, don't keep updating it too far or else the policy might get worse
...and it became less probable, you are free to undo that step (in the wrong direction) as much as you want
$A > 0$
L^{CLIP}
0
1
$1+\epsilon$
r
If the action was bad....
...and it became less probable, don't keep making it too much less probable or else the policy might get worse (i.e., don't step too far)
$A < 0$
0
$1-\epsilon$
1
r
...and it became more probable, you are free to undo that step as much as you want (i.e., you can fix your mistakes)
L^{CLIP}

Figure 1.3: Clipping Further Explained

1.4 Tailoring Contextual Decision Model For Our Agent

Effective decision-making by the rocket agent hinges on three key aspects: representation, modeling, and shaping. **Representation** refers to how we translate the complex environment, including the rocket's position, velocity, and orientation, into a format the agent can understand. This could be a vector of numbers or a more complex data structure. Once the agent has a suitable representation of the environment, **modeling** comes into play. This involves capturing the dynamics of how the agent's actions (e.g., adjusting thrust vector angles) influence the environment's state (e.g., rocket's movement) and the resulting rewards it receives. By understanding these relationships, the agent can learn to select actions that lead to a successful landing.

However, directly optimizing for a perfect landing might be challenging at the beginning of the training process. Here's where **shaping** plays a crucial role. Shaping refers to a technique used in reinforcement learning to guide the agent towards learning a desired behavior more efficiently. It involves manipulating the reward structure of the environment in a staged manner. By providing the agent with intermediate rewards for achieving smaller milestones (e.g., maintaining a stable orientation), we can guide it towards the ultimate goal of a safe landing. We will now talk about the considerations for representation, modeling, and shaping within the context of our rocket landing simulation.

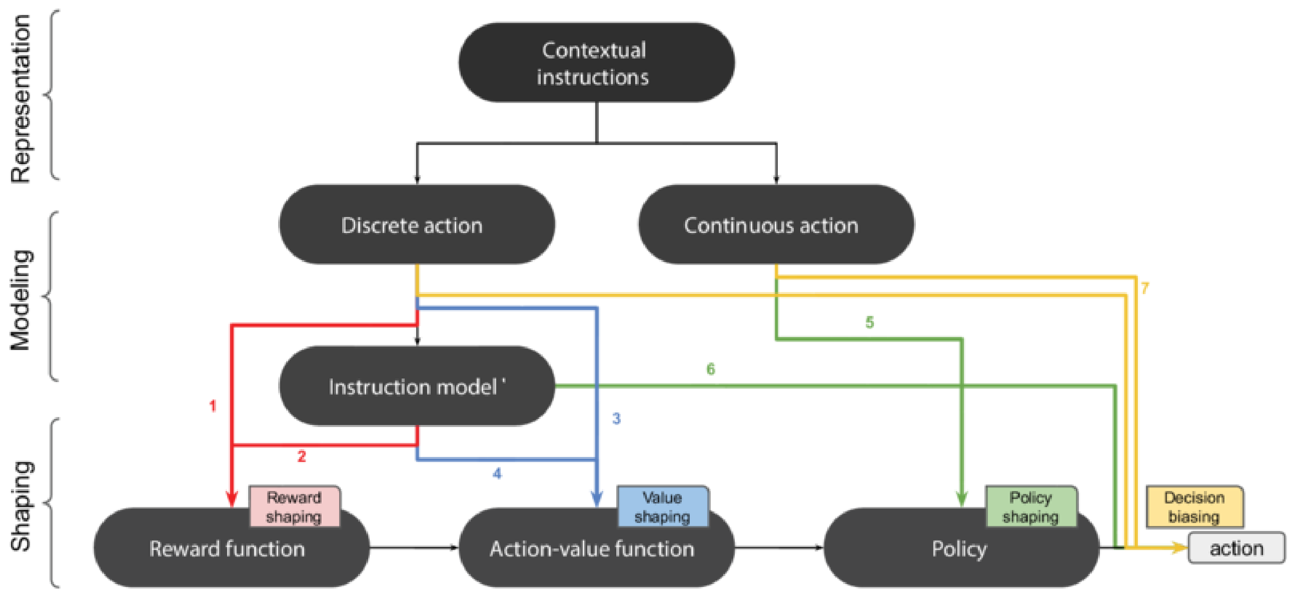


Figure 1.4: Reinforcement Learning Model for Contextual Decision Making [26]

1.4.1 Representation

This refers to how information about the environment and the agent's state is encoded. For example, the state of the environment (rocket position, velocity) might be represented as a vector of numbers.

- **Contextual Instructions:** These are signals or commands given to a learning agent that are dependent on the current state or context of the task.

1.4.2 Modeling

This refers to how the interaction between the agent and the environment is modeled. This includes the agent's actions (discrete or continuous) and how they affect the environment's state and the rewards received.

- **Discrete Action:** A type of action where the agent chooses from a set of distinct and separate possibilities.
- **Continuous Action:** Actions that can vary in a continuous way, allowing for a range of values rather than distinct options. for example in our rocket landing simulation, the agent might continuously adjust the thrust vector angles in various degrees to achieve a smooth landing.
- **Instruction Model:** A model that interprets and translates instructions into a form that the agent can understand and act upon.

1.4.3 Shaping

Shaping refers to a technique used in reinforcement learning (RL) to guide the agent towards learning a desired behavior more efficiently. It involves manipulating the reward structure of the environment in a staged manner.

- **Reward Function (Reward Shaping):** A technique in reinforcement learning where additional rewards are provided to guide the agent towards desired behaviors more quickly.
  - **Positive Rewards:** Rewards are given for actions that bring the rocket closer to a safe landing. These might include:

        * **Decreasing distance to landing pad:** As the rocket gets closer, the reward increases.
        * **Maintaining a stable orientation:** A reward is given for keeping the rocket upright during descent.
        * **Smooth velocity reduction:** Reward for gradually decelerating the rocket to avoid a crash landing.
    - **Penalizing Unwanted Behavior:** Penalize actions that deviate from the desired trajectory or lead to potential damage:
        * **Penalizing excessive tilting:** Penalize actions that cause the rocket to tilt significantly, jeopardizing stability.
        * **Crash penalty:** A significant negative reward for a crash landing.

    By carefully designing the reward function, we can provide the agent with the necessary feedback to learn a control policy that effectively guides the rocket to a safe and controlled landing.

- **Action-Value Function (Value Shaping):** A method of modifying the value function used by the agent to evaluate the desirability of actions, helping to shape the learning process.
- **Policy (Policy Shaping):** The strategy that an agent employs to decide which action to take in a given state. Policy shaping involves altering this strategy to improve learning.

Decision Biasing: Influencing the agent's decision-making process to favor certain actions over others, often to speed up learning or to ensure safety.

Action: The specific step or move made by an agent in response to the state of the environment or instructions received.

1.5 Related Work

In this paper [27], the authors introduce an intelligent control algorithm for rocket landings, utilizing deep reinforcement learning and Long Short Term Memory (LSTM) neural networks. By combining imitation and reinforcement learning, the method accelerates training and achieves real-time control on an embedded platform. The algorithm demonstrates superior landing accuracy, rapid convergence, and enhanced adaptability, showcasing its potential for autonomous rocket landings without heavy reliance on precise control models.

The authors [28] aims to use Unity ML-Agents and deep reinforcement learning for autonomous orbital rocket landings, focusing on reusable first-stage rockets. This introduces innovative autonomous control, aligning with SpaceX's progress in rapid reusability of launch vehicles.

This project [29] develops a simulation for evaluating classical control and machine learning algorithms in vertical rocket landings. It implements Proportional Integral Derivative (PID) controller, Model Predictive Control (MPC), and RL algorithms like Deep Deterministic Policy Gradients (DDPG), showcasing potential for stable and consistent landings. Aligned with SpaceX and Blue Origin's successes, it contributes to the broader goal of reusable space systems, addressing the challenge of autonomous rocket landings.

It's important to acknowledge the advancements our simulation offers compared to previous research by Ferrante et al [29]. Our work leverages a fully immersive 3D environment, allowing for a more comprehensive evaluation of the AI control system's capabilities. In contrast, Ferrante et al.'s research employed a 2D simulation setting, potentially limiting the complexity of the landing scenarios explored.

Furthermore, our simulation expands the control options available to the AI compared to the two degrees of control (X-axis for lateral movements) and two Cold Gas Thrusters (CGTs) on the left and right featured in his work. We introduce a more comprehensive control scheme by incorporating four Cold Gas Thrusters (left, right, front, and back) and a thrust vector direction that can be adjusted along both the X and Z axes. This enhanced controllability allows for a more realistic simulation of rocket dynamics and a more rigorous assessment of the AI's ability to perform complex maneuvers during landing.

1.6 Conclusion

This chapter has established a foundational understanding of reinforcement learning (RL) as a promising approach to complex rocket landing control. We have examined the core concepts of RL, emphasizing the interplay between the agent, environment, states, actions, rewards, and policy learning – all geared towards maximizing long-term reward. We then identified Proximal Policy Optimization (PPO) as a well-suited RL algorithm for continuous control problems like rocket landing. We discussed the advantages of PPO in this context, particularly its ability to balance exploration and stability during policy updates. Finally, we introduced the concept of a contextual decision model specifically tailored for our rocket landing agent, highlighting the critical roles of representation, modeling, and reward shaping.

By laying this groundwork, the path is now clear for a deeper exploration of RL in rocket landing control. Subsequent chapters will delve into the specifics of our approach. This includes the design of a realistic and informative environment, detailing the observations accessible to the agent and the actions it can take. We will explore the art of shaping the reward system to effectively guide the agent's learning process, alongside a discussion on the hyperparameters employed during training. Finally, we will rigorously evaluate the agent's performance to assess the effectiveness of RL in achieving successful and efficient rocket landings. This evaluation will provide valuable insights into the potential of RL for real-world rocket control applications.

Chapter 2

Unveiling the Magic of Rocket Science: From Liftoff to Landing

2.1 Introduction

This chapter delves into the fascinating world of rocket science, equipping you with the fundamental knowledge of how rockets move through space and achieve controlled landings.

Have you ever gazed up at the night sky and marveled at the stars and planets twinkling above? Perhaps you've dreamt of one day venturing into the cosmos and exploring the vast unknown. This dream of space travel has captivated humanity for centuries, and rockets have played a pivotal role in making it a reality.

This chapter delves into the fascinating world of rocket science, the engineering marvel that allows us to defy gravity and propel objects into space. We will embark on a journey to understand the fundamental principles of how rockets work, from the core concepts of thrust generation to the intricate details of rocket components. We will explore the challenges of maneuvering and controlling rockets during flight, and delve into the complexities of achieving a successful and controlled landing.

By the end of this chapter, you will be equipped with the foundational knowledge of rocket science, preparing you to explore the exciting world of autonomous rocket control using reinforcement learning in the following chapters.

2.1.1 The Need for Rocket Science

Space travel demands overcoming immense challenges: escaping Earth's gravity, operating in a vacuum, reaching high speeds, maneuvering in space, and re-entering for landing [30] . Rockets, with their powerful engines and unique design, are the key to unlocking the vast possibilities beyond our planet.

2.2 Rocket Fundamentals

This section delves into the core principles that enable rockets to defy gravity and propel themselves through the vastness of space. We'll explore the fundamental concepts behind thrust generation, including the crucial role played by Newton's Third Law of Motion 2.1.

$$\sum \vec{F}_{\text{on } A} = -\sum \vec{F}_{\text{on } B} \tag{2.1}$$

2.2.1 Unveiling the Power of Propulsion

At the heart of every rocket lies its engine, a marvel of engineering that harnesses the power of controlled explosions to generate thrust. Rocket engines operate on the fundamental principle of Newton's Third Law of Motion: for every action, there is an equal and opposite reaction (see eq. 2.1).

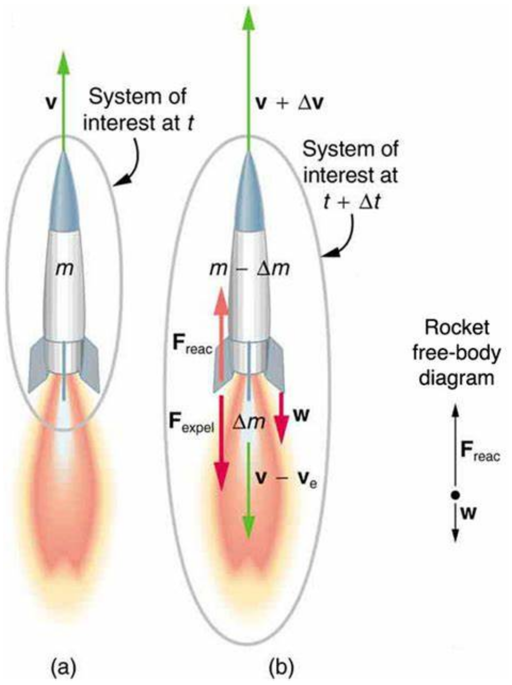


Figure 2.1: Rocket Propulsion.

(a) This rocket has a mass m and an upward velocity v. The net external force on the system is $-mg$, if air resistance is neglected.

(b) A time Δt later the system has two main parts, the ejected gas and the remainder of the rocket. The reaction force on the rocket is what overcomes the gravitational force and accelerates it upward.

2.2.2 Essential Components of a Rocket

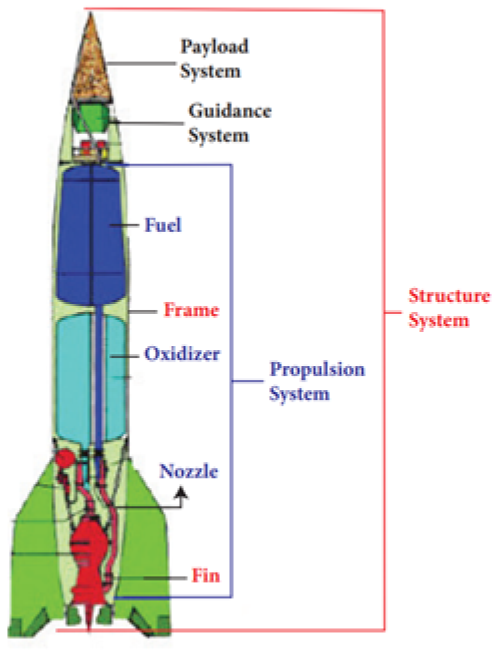


Figure 2.2: Structure of a Rocket

There are four main systems or components in a rocket. They are

- **Structural system**
- **Payload system:** This is the cargo that the rocket carries into space. It can be anything from satellites and probes for scientific exploration to spacecraft carrying astronauts or cargo for space stations. The payload capacity of a rocket is limited by its overall thrust and the amount of fuel it can carry.
- **Guidance system:** This system ensures the rocket follows the desired trajectory. It uses various sensors like accelerometers, gyroscopes, and star trackers to determine the rocket's position, orientation, and velocity. Based on this information, the guidance system calculates adjustments and sends commands to control surfaces or thrust vectoring systems (discussed later) to steer the rocket on course.
- **Propulsion system**

    1. Engine (Discussed in 2.2.1): As the heart of the rocket, the engine provides the thrust

necessary for liftoff and maneuvering. We've already discussed its role in generating thrust through combustion and gas expansion.

2. **Fuel Tanks:** Rockets carry a significant amount of fuel and oxidizer to power the engine during flight. These tanks are typically made of lightweight and high-strength materials like aluminum or composites to minimize weight while ensuring structural integrity.
   - **Fuel Tank:** Stores the fuel (e.g., liquid hydrogen, kerosene) used for combustion within the engine.
   - **Oxidizer Tank:** Carries the oxidizer (often liquid oxygen) as combustion cannot occur in space without an external oxygen source.

2.3 The Art of Rocket Control

While a powerful engine propels a rocket forward, maneuvering and steering it during flight require a different set of technologies. This section explores the crucial aspects of rocket control: Thrust Vector Control (TVC) [14], Cold Gas Thrusters (CGT) [31], Understanding Pitch, Yaw, and Roll and our Guidance & Navigation System.

2.3.1 Thrust Vector Control (TVC)

Thrust Vector Control (TVC) allows for maneuvering a rocket by adjusting the direction of its engine thrust. Instead of a fixed exhaust plume, TVC enables the engine nozzle to swivel slightly, directing the thrust vector in a specific direction.

Here's a breakdown of how TVC works:

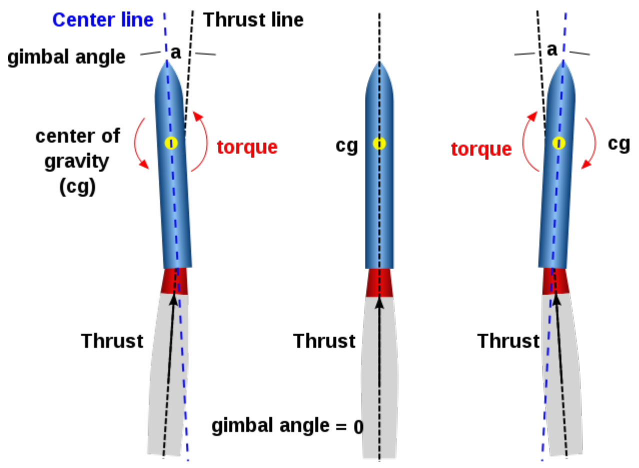


Figure 2.3: Thrust Vector Control

1. **Movable Nozzle:** The engine nozzle is designed with a mechanism that allows it to pivot or deflect within a limited range. This movement can be achieved using hydraulic actuators, electric motors, or other control systems.

2. **Steering by Thrust Adjustment:** By electronically controlling the nozzle deflection, the direction of the thrust vector can be adjusted. Tilting the nozzle slightly in one direction creates a force that causes the rocket to turn in the opposite direction, similar to how a boat's rudder works.

3. **Precise Maneuvering:** TVC allows for more precise control during flight maneuvers, including course corrections, attitude adjustments, and even controlled landings for reusable rockets.

   Benefits of TVC:

   - Enhanced Maneuverability: Enables precise control during flight maneuvers.
   - Improved Stability: Allows for corrections to counteract any external disturbances.
   - Landing Assist: Plays a crucial role in controlling the descent and achieving a soft landing for reusable rockets

While Thrust Vector Control (TVC) provides powerful maneuvering capabilities, rockets also utilize another system for precise adjustments and orientation control: Cold Gas Thrusters (CGT)

[15, 16].

2.3.2 Cold Gas Thrusters (CGT)

A Cold Gas Thruster[15, 16], also known as a cold gas propulsion system, operates by harnessing the expansion of pressurized gas to generate thrust, without involving any combustion. Unlike traditional rocket engines, cold gas thrusters have simpler designs, comprising only a **fuel tank**, a **regulating valve**, a **propelling nozzle**, and **minimal plumbing**. While they offer lower thrust and efficiency compared to monopropellant and bipropellant engines, they are valued for their affordability, simplicity, and reliability. Cold gas thrusters are widely used for tasks such as **orbital maintenance**, **maneuvering**, and **attitude control** due to their cost-effectiveness and ease of operation.

Figure 2.4: Cold Gas Thrusters

2.3.2.1 The Essence of CGT Maneuvering:

Unlike the fiery combustion engines used for main propulsion, CGTs rely on a simpler approach. They utilize the rapid expansion of pressurized gas (often helium or nitrogen) to generate thrust. This allows for controlled bursts of force, ideal for:

- Fine-Tuning Trajectory: Making minor course corrections during flight to stay on the desired path.

- Orientation Adjustments: Maintaining the rocket's precise attitude (pointing direction) to ensure instruments and sensors are aimed correctly.
- Station Keeping: Holding the rocket in a fixed position relative to another object in space, critical for satellite deployment or docking maneuvers.

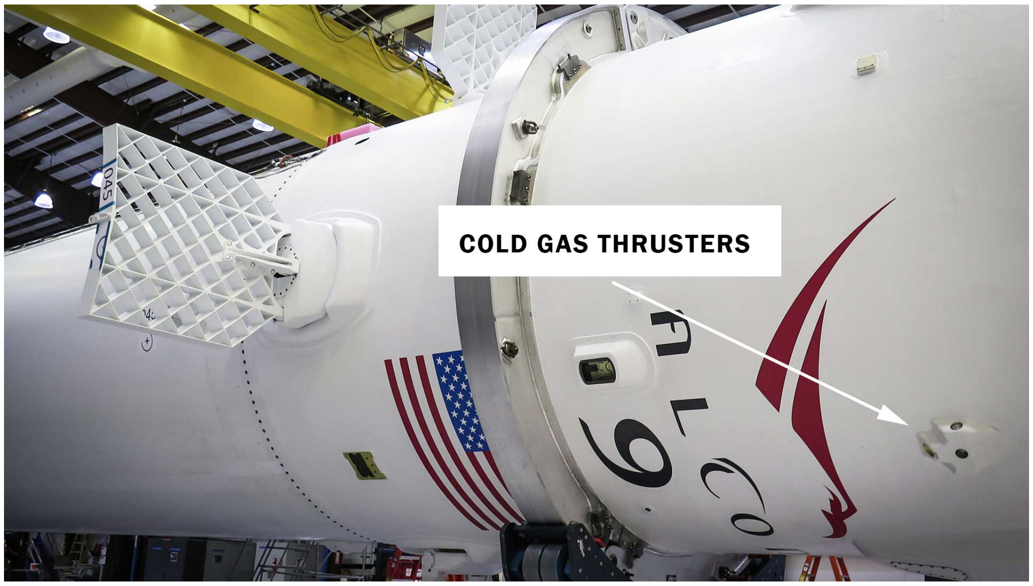


Figure 2.5: CGT On A Falcon 9 Rocket

The falcon 9 is equipped with a total of 8 nitrogen cold gas thrusters that are mounted towards the top of the first stage. There is one pod on each side of the rocket, each containing 4 thrusters. Like the gimbaled main engines, the cold gas thrusters are used to control the orientation of the rocket.

2.3.2.2 Achieving Precise Maneuvers With CGTs

1. **Pressurized Gas Reservoir:** CGTs hold a tank filled with an inert gas, typically under high pressure.
2. **Simple and Reliable Design:** These systems comprise a propellant tank, a pressure regulator valve, and a nozzle. This simplicity translates to high reliability.

Figure 2.6: Cold Gas Thrusters In Action

3. **On-Demand Thrust Bursts:** When activated, the valve opens, allowing the pressurized gas to flow through the nozzle and rapidly expand.

4. **Precise Control:** The short burst of thrust generated by gas expansion can be precisely controlled electronically, enabling delicate adjustments in the desired direction.

2.3.2.3 Benefits of Utilizing CGTs

- Pinpoint Maneuvering: Their ability to deliver small, controlled thrust bursts makes them ideal for fine-tuning a rocket's trajectory and attitude.

- Fuel Efficiency: CGTs offer exceptional fuel efficiency as they use a minimal amount of propellant for each burst.

- Simplicity and Reliability: The lack of complex combustion processes minimizes the risk of malfunctions, making them a reliable choice for critical maneuvers.

- Safety: Using inert gases eliminates the risk of explosion or contamination associated with traditional propellants.

2.3.2.4 Limitations to Consider:

- Low Thrust Output: Compared to main engines, CGTs generate significantly lower thrust, limiting their use for major course corrections.
- Propellant Depletion: Since they rely on a finite amount of stored gas, prolonged use can deplete their propellant, requiring careful management during missions.

2.3.3 Pitch, Yaw, and Roll

Rockets may soar through the vast emptiness of space, but they still need to be steered! Just like navigating our own 3D world, controlling a rocket's direction requires precise control in all three dimensions.

Imagine the rocket balancing on a point in space – its center of gravity, where all its weight is concentrated. To control its "posture" or **attitude**, we need to understand how it rotates around this point.

Think of a giant, invisible X, Y, and Z axis crossing through the center of gravity. By monitoring how much the rocket twists and turns along each axis, we can determine its exact orientation in space. This knowledge then allows us to make adjustments and ensure the rocket dances across the celestial stage with perfect form.

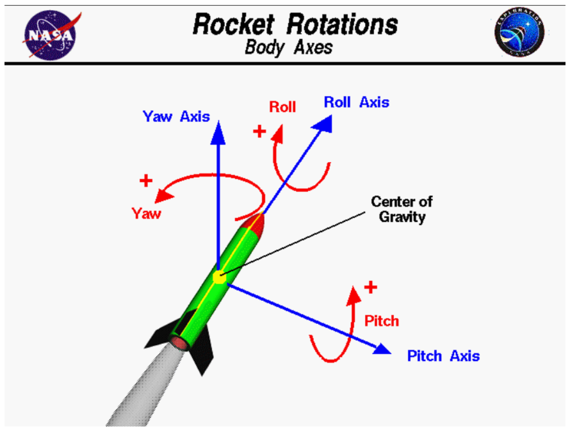


Figure 2.7: Pitch Yaw and Roll [32]

To navigate and maneuver precisely, the rocket needs to control its orientation in three dimensions. Here's where **pitch**, **yaw**, and **roll** come in:

- **Pitch**: This refers to the tilting motion of the rocket's nose up or down, like nodding your head. Pitch control allows the rocket to adjust its trajectory during flight.
- **Yaw**: This describes the rocket's turning motion left or right, similar to shaking your head. Yaw control helps the rocket change its direction while facing forward.
- **Roll**: This signifies the rocket's rotation along its long axis, like rolling a barrel. Precise roll control ensures the rocket stays upright and maintains stability [33].

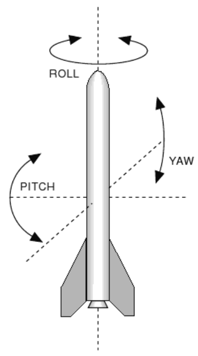


Figure 2.8: Pitch Yaw and Roll (Different Perspective)

These three rotations work together to give the rocket complete control over its movement in space.

2.3.4 Guidance and Navigation

While TVC helps steer the rocket, it requires a guiding system to determine the desired direction and trajectory. This is where Guidance and Navigation come into play.

The Role of Guidance and Navigation: This system acts as the "brain" of the rocket, determining its course and ensuring it reaches its destination. It comprises several key components:

- **Sensors:** These instruments collect data about the **rocket's position**, **orientation**, and **velocity**. Examples include accelerometers (measure acceleration), gyroscopes (measure angular rate), and a **landing pad tracker** (This sensor, specifically used for landing maneuvers, can be a radar system that tracks the designated landing pad. It provides real-time information about the rocket's relative position and distance to the landing pad, aiding in precise control during descent and touchdown.)

- **Onboard Computer:** This computer processes the sensor data and calculates the necessary adjustments to keep the rocket on course. It compares the actual trajectory with the desired path and sends commands to the control surfaces or TVC system to make corrections.

- **Reference System:** The guidance system needs a reference point to determine the desired trajectory. This could be a pre-programmed flight path based on ground control data, or it could use celestial navigation by referencing the position of stars.

2.4 The Challenge of Returning from the Stars: Landing a Rocket

While launching a rocket is a marvel of engineering, the true test lies in bringing it back safely [34]. This section explores the hurdles of rocket landings and the strategies employed, focusing on the Falcon 9's impressive reusability.

2.4.1 The Complexities of Re-entry

Imagine a rocket screaming back towards Earth at incredible speeds. Re-entry is a critical phase [35] where the vehicle faces several challenges:

- **Intense Heat**: As the rocket plunges through the atmosphere, friction with air molecules creates extreme heat, potentially reaching thousands of degrees Celsius. This heat can damage the rocket's structure if not properly managed.

- **High G-Forces**: The rapid deceleration during re-entry subjects the rocket and its payload to significant gravitational forces (g-forces). These forces can stress the vehicle and its contents.

- **Atmospheric Control**: Friction with air also disrupts the smooth flow of air around the rocket, potentially causing instability and control issues.

Overcoming these challenges requires careful planning and design [**?**]. Rockets employ heat shields made of special materials that absorb and deflect the intense heat. Additionally, aerodynamic control surfaces or thrust vectoring (explained earlier) help maintain stability during re-entry.

2.4.2 The Power Of Controlled Landing

Traditionally, rockets were expendable, meaning they were destroyed upon launch or ditched in the ocean after use. This approach is costly and generates a lot of space debris. However, the concept of **reusable rockets** is revolutionizing spaceflight.

Precisely controlling a rocket's descent allows for a targeted landing, minimizing damage and enabling reuse. This significantly reduces launch costs and promotes a more sustainable approach to space exploration.

2.4.2.1 Falcon 9 Landing Strategy:

SpaceX's Falcon 9 exemplifies a successful reusable launch vehicle. Here's how it lands:

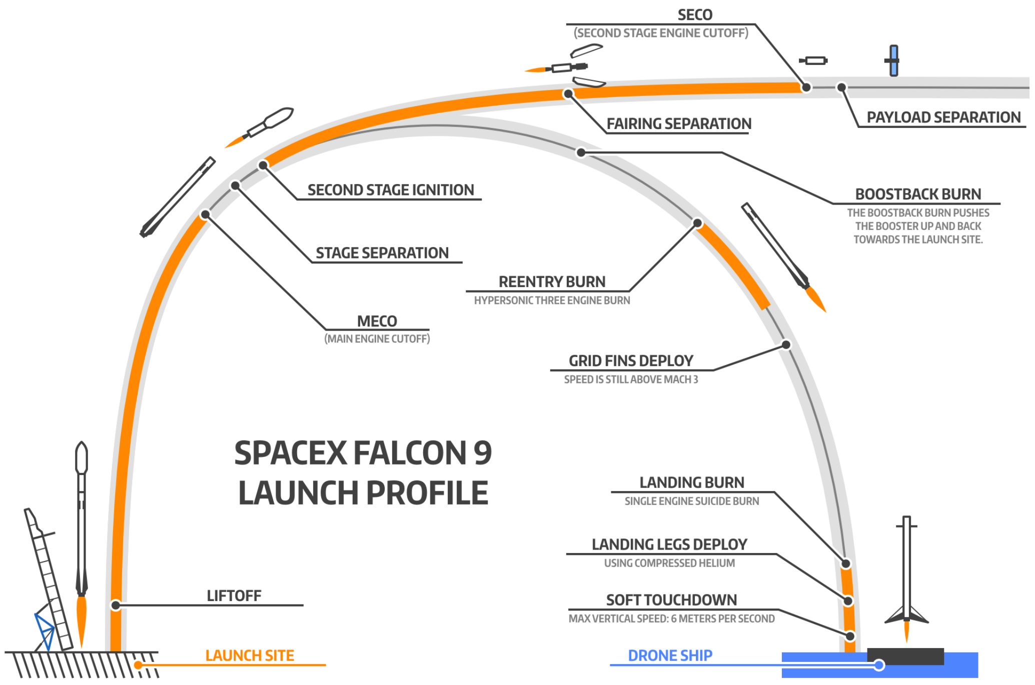

SECO
(SECOND STAGE ENGINE CUTOFF)
FAIRING SEPARATION
PAYLOAD SEPARATION
SECOND STAGE IGNITION
BOOSTBACK BURN
THE BOOSTBACK BURN PUSHES THE BOOSTER UP AND BACK TOWARDS THE LAUNCH SITE.
STAGE SEPARATION
REENTRY BURN
HYPERSONIC THREE ENGINE BURN
MECO
(MAIN ENGINE CUTOFF)
GRID FINS DEPLOY
SPEED IS STILL ABOVE MACH 3
SPACEX FALCON 9 LAUNCH PROFILE
LANDING BURN
SINGLE ENGINE SUICIDE BURN
LANDING LEGS DEPLOY
USING COMPRESSED HELIUM
LIFTOFF
SOFT TOUCHDOWN
MAX VERTICAL SPEED: 6 METERS PER SECOND
LAUNCH SITE
DRONE SHIP

Figure 2.9: Falcon 9 Launch And Landing Profile

1. **Stage Separation :** After propelling the payload to its desired orbit, the first stage (booster) of the Falcon 9 separates and begins its descent back to Earth.

2. **Re-entry Maneuvers :** The booster uses grid fins and thrust vectoring to orient itself and control its re-entry.

3. **Powered Descent :** As the booster nears the landing zone, it reignites its engines for a controlled descent, slowing down significantly.

4. **Landing Legs Deploy :** Deployable landing legs extend to prepare for touchdown.

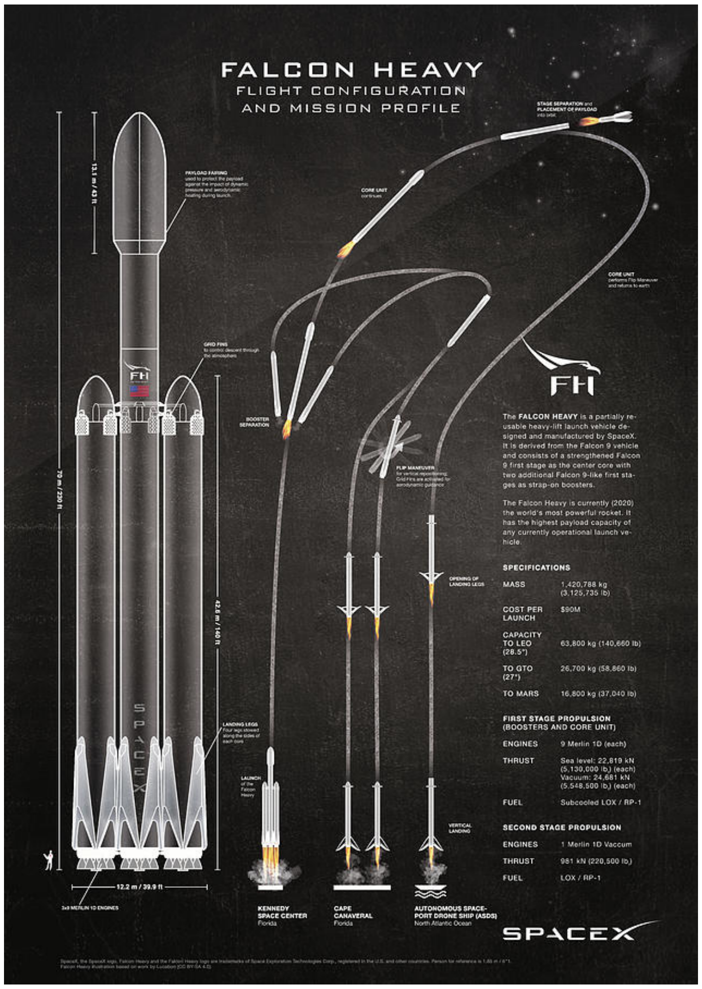
5. **Precision Touchdown :** Using onboard guidance and engine control, the Falcon 9 aims for a designated landing pad on land or a drone ship in the ocean, achieving a soft vertical landing.

The success of the Falcon 9 landing strategy paves the way for a more cost-effective and sustainable future of space exploration. By mastering the art of controlled landings, we can unlock new possibilities for space travel and discovery.

2.4.2.2 A Family of Reusables: Unveiling Falcon Heavy and Starship

While the Falcon 9's reusability is impressive, it's not the only game in town. Let's explore the launch and landing profiles of other SpaceX vehicles:

1. **SpaceX Falcon Heavy:** This "brute force" cousin of the Falcon 9 is essentially three Falcon 9 cores strapped together, significantly boosting its payload capacity. Here's its profile:


FALCON HEAVY
FLIGHT CONFIGURATION
AND MISSION PROFILE
13.1 m / 43 ft
70 m / 230 ft
42.6 m / 140 ft
12.2 m / 39.9 ft
3x9 MERLIN 1D ENGINES
PAYLOAD FAIRING used to protect the payload against the impact of dynamic pressure and aerodynamic heating during launch.
GRID FINS to control descent through the atmosphere
LANDING LEGS Four legs stowed along the sides of each core
SPACEX
FH
CORE UNIT continues
STAGE SEPARATION and PLACEMENT OF PAYLOAD into orbit
CORE UNIT performs Flip Maneuver and returns to earth
BOOSTER SEPARATION
FLIP MANEUVER for vertical repositioning. Grid Fins are activated for aerodynamic guidance
OPENING OF LANDING LEGS
VERTICAL LANDING
LAUNCH of the Falcon Heavy
KENNEDY SPACE CENTER Florida
CAPE CANAVERAL Florida
AUTONOMOUS SPACE-PORT DRONE SHIP (ASDS) North Atlantic Ocean
SPACEX

The **FALCON HEAVY** is a partially reusable heavy-lift launch vehicle designed and manufactured by SpaceX. It is derived from the Falcon 9 vehicle and consists of a strengthened Falcon 9 first stage as the center core with two additional Falcon 9-like first stages as strap-on boosters.

The Falcon Heavy is currently (2020) the world's most powerful rocket. It has the highest payload capacity of any currently operational launch vehicle.

SPECIFICATIONS

MASS	1,420,788 kg (3,125,735 lb)
COST PER LAUNCH	$90M
CAPACITY TO LEO (28.5°)	63,800 kg (140,660 lb)
TO GTO (27°)	26,700 kg (58,860 lb)
TO MARS	16,800 kg (37,040 lb)

FIRST STAGE PROPULSION (BOOSTERS AND CORE UNIT)

ENGINES	9 Merlin 1D (each)
THRUST	Sea level: 22,819 kN (5,130,000 lb.) (each) Vacuum: 24,681 kN (5,548,500 lb.) (each)
FUEL	Subcooled LOX / RP-1

SECOND STAGE PROPULSION

ENGINES	1 Merlin 1D Vaccum
THRUST	981 kN (220,500 lb.)
FUEL	LOX / RP-1

SpaceX, the SpaceX logo, Falcon Heavy and the Falcon Heavy logo are trademarks of Space Exploration Technologies Corp., registered in the U.S. and other countries. Person for reference is 1.80 m / 6"1.
Falcon Heavy illustration based on work by Lucabon [CC BY-SA 4.0]

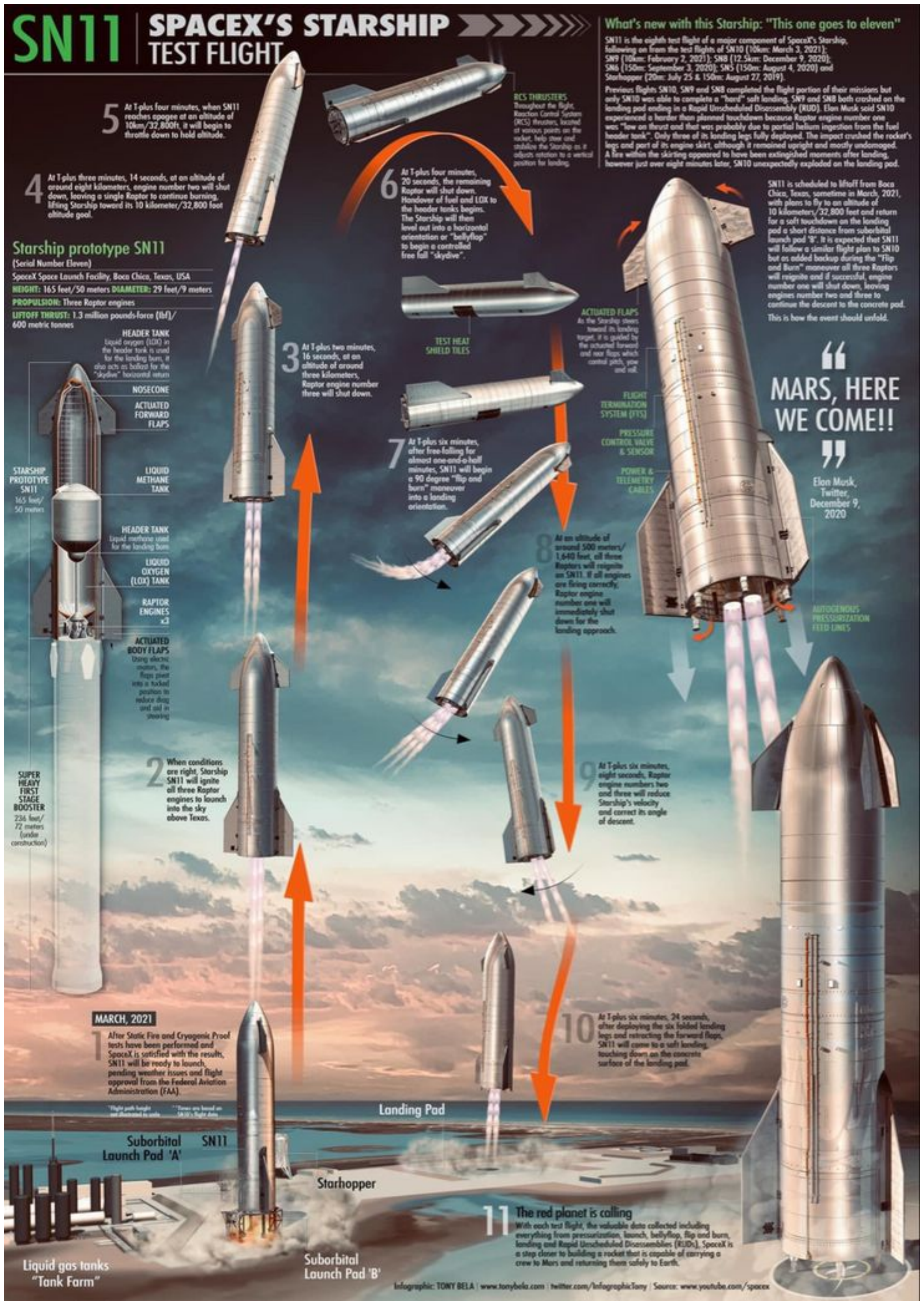
Figure 2.10: Falcon Heavy Flight Configuration And Mission Profile

- Launch: Similar to the Falcon 9, all three cores ignite simultaneously, lifting the massive vehicle off the launchpad. After reaching a certain altitude, the two side boosters detach and return to Earth for reuse, employing a similar re-entry and landing strategy as the Falcon 9.
- Landing: The central core, carrying the payload, continues its journey to orbit. However, depending on the mission requirements, this core might also be recovered. In such cases, it performs a re-entry and landing similar to the side boosters, often targeting a designated landing pad.

2. **Starship**: This ambitious next-generation launch vehicle aims to revolutionize space travel. It consists of two reusable elements:

   (a) **Super Heavy Booster:** This massive booster, powered by next-generation Raptor engines, lifts the Starship spacecraft to high altitude. After separation, the Super Heavy performs a controlled re-entry and landing very similar to the Falcon 9, aiming for a designated landing pad.

   (b) **Starship Spacecraft:** This spacecraft carries crew and cargo to various destinations, including the Moon, Mars, and beyond. Unlike the Falcon 9 and Falcon Heavy, Starship itself is designed to be fully reusable. After reaching orbit, it performs a belly-flop maneuver during re-entry to slow down using the atmosphere. Finally, it lands vertically on landing legs at the launch site, similar to the Falcon 9.


SN11 | SPACEX'S STARSHIP TEST FLIGHT
RCS THRUSTERS
TEST HEAT SHIELD TILES
ACTUATED FLAPS
FLIGHT TERMINATION SYSTEM (FTS)
PRESSURE CONTROL VALVE & SENSOR
POWER & TELEMETRY CABLES
AUTOGENOUS PRESSURIZATION FEED LINES
HEADER TANK
NOSECONE
ACTUATED FORWARD FLAPS
LIQUID METHANE TANK
HEADER TANK
LIQUID OXYGEN (LOX) TANK
RAPTOR ENGINES x3
ACTUATED BODY FLAPS
STARSHIP PROTOTYPE SN11 165 feet/ 50 meters
SUPER HEAVY FIRST STAGE BOOSTER 236 feet/ 72 meters (under construction)
"MARS, HERE WE COME!!" Elon Musk, Twitter, December 9, 2020
MARCH, 2021
Landing Pad
Suborbital Launch Pad 'A'
SN11
Starhopper
Suborbital Launch Pad 'B'
Liquid gas tanks "Tank Farm"
Infographic: TONY BELA | www.tonybela.com | twitter.com/InfographicTony | Source: www.youtube.com/spacex

What's new with this Starship: "This one goes to eleven"

SN11 is the eighth test flight of a major component of SpaceX's Starship, following on from the test flights of SN10 (10km: March 3, 2021); SN9 (10km: February 2, 2021); SN8 (12.5km: December 9, 2020); SN6 (150m: September 3, 2020); SN5 (150m: August 4, 2020) and Starhopper (20m: July 25 & 150m: August 27, 2019).

Previous flights SN10, SN9 and SN8 completed the flight portion of their missions but only SN10 was able to complete a "hard" soft landing. SN9 and SN8 both crashed on the landing pad ending in a Rapid Unscheduled Disassembly (RUD). Elon Musk said SN10 experienced a harder than planned touchdown because Raptor engine number one was "low on thrust and that was probably due to partial helium ingestion from the fuel header tank". Only three of its landing legs fully deployed. The impact crushed the rocket's legs and part of its engine skirt, although it remained upright and mostly undamaged. A fire within the skirting appeared to have been extinguished moments after landing, however just over eight minutes later, SN10 unexpectedly exploded on the landing pad.

SN11 is scheduled to liftoff from Boca Chica, Texas, sometime in March, 2021, with plans to fly to an altitude of 10 kilometers/32,800 feet and return for a soft touchdown on the landing pad a short distance from suborbital launch pad 'B'. It is expected that SN11 will follow a similar flight plan to SN10 but as added backup during the "Flip and Burn" maneuver all three Raptors will reignite and if successful, engine number one will shut down, leaving engines number two and three to continue the descent to the concrete pad.

This is how the event should unfold.

Starship prototype SN11

(Serial Number Eleven)

SpaceX Space Launch Facility, Boca Chica, Texas, USA

HEIGHT: 165 feet/50 meters DIAMETER: 29 feet/9 meters

PROPULSION: Three Raptor engines

LIFTOFF THRUST: 1.3 million pounds-force (lbf)/ 600 metric tonnes

HEADER TANK: Liquid oxygen (LOX) in the header tank is used for the landing burn, it also acts as ballast for the "skydive" horizontal return

HEADER TANK: Liquid methane used for the landing burn

ACTUATED BODY FLAPS: Using electric motors, the flaps pivot into a tucked position to reduce drag and aid in steering

RCS THRUSTERS: Throughout the flight, Reaction Control System (RCS) thrusters, located at various points on the rocket, help steer and stabilize the Starship as it adjusts rotation to a vertical position for landing.

ACTUATED FLAPS: As the Starship steers toward its landing target, it is guided by the actuated forward and rear flaps which control pitch, yaw and roll.

1 After Static Fire and Cryogenic Proof tests have been performed and SpaceX is satisfied with the results, SN11 will be ready to launch, pending weather issues and flight approval from the Federal Aviation Administration (FAA).

2 When conditions are right, Starship SN11 will ignite all three Raptor engines to launch into the sky above Texas.

3 At T-plus two minutes, 16 seconds, at an altitude of around three kilometers, Raptor engine number three will shut down.

4 At T-plus three minutes, 14 seconds, at an altitude of around eight kilometers, engine number two will shut down, leaving a single Raptor to continue burning, lifting Starship toward its 10 kilometer/32,800 foot altitude goal.

5 At T-plus four minutes, when SN11 reaches apogee at an altitude of 10km/32,800ft, it will begin to throttle down to hold altitude.

6 At T-plus four minutes, 20 seconds, the remaining Raptor will shut down. Handover of fuel and LOX to the header tanks begins. The Starship will then level out into a horizontal orientation or "bellyflop" to begin a controlled free fall "skydive".

7 At T-plus six minutes, after free-falling for almost one-and-a-half minutes, SN11 will begin a 90 degree "flip and burn" maneuver into a landing orientation.

8 At an altitude of around 500 meters/ 1,640 feet, all three Raptors will reignite on SN11. If all engines are firing correctly, Raptor engine number one will immediately shut down for the landing approach.

9 At T-plus six minutes, eight seconds, Raptor engine numbers two and three will reduce Starship's velocity and correct its angle of descent.

10 At T-plus six minutes, 24 seconds, after deploying the six folded landing legs and retracting the forward flaps, SN11 will come to a soft landing, touching down on the concrete surface of the landing pad.

11 The red planet is calling

With each test flight, the valuable data collected including everything from pressurization, launch, bellyflop, flip and burn, landing and Rapid Unscheduled Disassemblies (RUDs), SpaceX is a step closer to building a rocket that is capable of carrying a crew to Mars and returning them safely to Earth.

Figure 2.11: Star Ship Flight Configuration And Mission Profile

By showcasing these diverse launch and landing profiles, we gain a broader perspective on SpaceX's innovative approach to reusable rockets, paving the way for a more sustainable and cost-effective future in space exploration.

2.5 Conclusion

This chapter has taken us on a thrilling journey, exploring the inner workings of a rocket. We've delved into the power of thrust, the delicate dance of pitch, yaw, and roll, and the crucial role of sensors in guiding the beast through the vastness of space.

We then tackled the complexities of returning a rocket safely, with the fiery challenges of re-entry and the awe-inspiring power of controlled landings, exemplified by the reusable Falcon 9. We even peeked into the future with the innovative launch and landing profiles of SpaceX's Falcon Heavy and Starship.

As we move forward, remember that these marvels of engineering are not just sophisticated machines. They are testaments to human ingenuity, pushing the boundaries of science and technology to unlock the secrets of the universe. The journey may hold challenges, but with each successful launch, controlled landing, and mission accomplished, we inch closer to a future where the stars are no longer a distant dream, but a tangible reality.

Summarizing the fundamental knowledge gained about rocket science and their control.

Chapter 3

From Concept to Control: Methodology for a Rocket Landing Simulation

3.1 Simulation Environment Design

3.1.1 Introduction

Before we embark on building our virtual launchpad, it's crucial to understand why simulation plays such a vital role in developing successful rocket landing systems. Real-world rocket launches are incredibly complex and expensive endeavors. Risks associated with physical testing are high, and opportunities for experimentation are limited. Here's where simulation steps in.

Simulations allow us to create safe and cost-effective environments where we can test and refine different landing strategies repeatedly. These virtual testing grounds offer valuable insights into the behavior of rockets under various conditions, enabling us to identify potential issues and optimize landing control systems before venturing into actual launches.

Now, let's talk about our choice of platform: **Unity**. We've opted for Unity due to several compelling reasons:

- Powerful and Flexible: Unity is a widely recognized game engine with a robust physics en-

Figure 3.1: Unity Logo

gine at its core. This allows for the creation of realistic simulations that accurately model the physics of rocket flight and landing.

- Accessibility and User-Friendliness: Unity offers a user-friendly interface and a vast array of tools that make it ideal for developing interactive simulations. This allows for efficient development and the incorporation of various features into the simulation environment.
- Large and Active Community: Unity boasts a large and active community of developers. This translates to readily available resources, tutorials, and support, which can prove invaluable during the development process.

3.1.2 Development Environment and Scripting Languages

Before we delve into the specifics of our virtual design, let's discuss the tools that will bring it to life. We'll be utilizing a powerful combination of software to create our rocket landing simulation:

3.1.2.1 Visual Studio Code (VS Code):

This versatile code editor serves as our central hub for development. VS Code offers a user-friendly interface, syntax highlighting, and debugging capabilities, making it ideal for writing and editing code efficiently.

Figure 3.2: Visual Studio Code

3.1.2.2 Scripting Languages:

Our simulation will leverage two primary scripting languages:

- **C#:** As the primary scripting language within Unity, C# will be used to develop core functionalities of the simulation environment. This includes scripting the behavior of the virtual rocket, interacting with the physics engine, and managing user input for control purposes.

Figure 3.3: C#

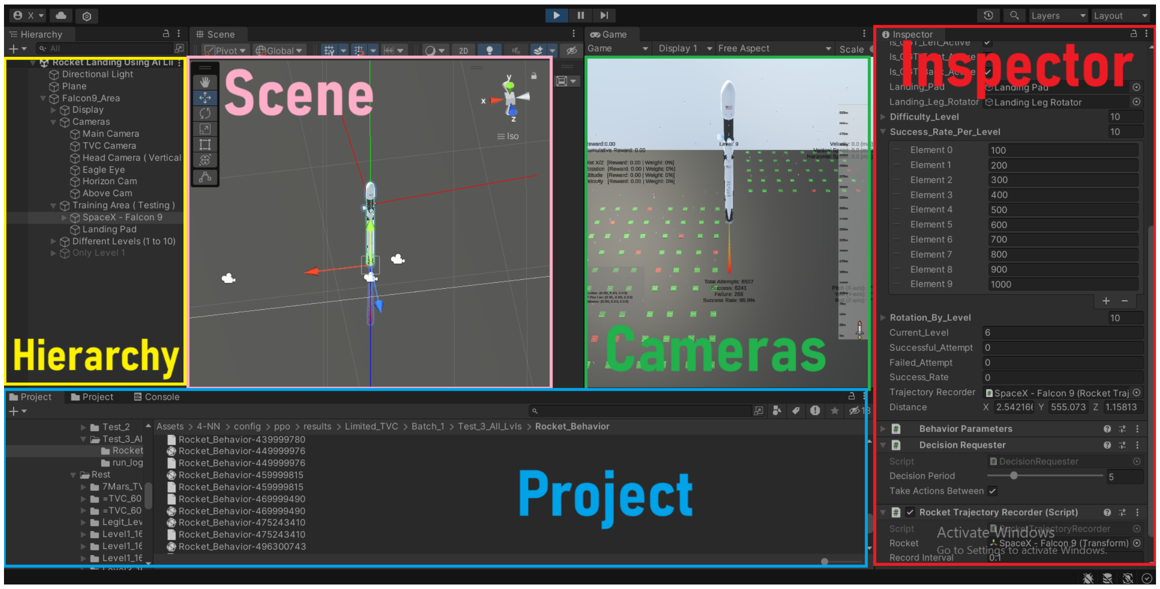
- **Python:** When dealing with intricate artificial intelligence (AI) and reinforcement learning algorithms, Python presents a compelling option due to its concise and readable syntax. This makes it well-suited for rapid prototyping and development of these complex algorithms. Additionally, Python's extensive ecosystem of data analysis libraries, such as matplotlib, will be instrumental in extracting valuable insights from the simulation. These libraries can be leveraged to perform tasks like: **Rocket Trajectory Plotting**

Figure 3.4: Python

By combining the power of VS Code with C# Python, we gain the flexibility and control needed to build and refine our rocket landing simulation within the Unity framework.

3.1.3 A Glimpse into the Unity Interface

Our exploration of the virtual landing environment wouldn't be complete without a brief introduction to the core sections of the Unity interface that facilitate its creation. Understanding these sections is crucial for navigating and manipulating elements within the simulation.



Rocket Landing Using AI Lil
Directional Light
Plane
Falcon9_Area
Display
Cameras
Main Camera
TVC Camera
Head Camera (Vertical
Eagle Eye
Horizon Cam
Above Cam
Training Area (Testing)
SpaceX - Falcon 9
Landing Pad
Different Levels (1 to 10)
Only Level 1

Landing_Pad	Landing Pad
Landing_Leg_Rotator	Landing Leg Rotator
Difficulty_Level	10
Success_Rate_Per_Level	10
Element 0	100
Element 1	200
Element 2	300
Element 3	400
Element 4	500
Element 5	600
Element 6	700
Element 7	800
Element 8	900
Element 9	1000
Rotation_By_Level	10
Current_Level	6
Successful_Attempt	0
Failed_Attempt	0
Success_Rate	0
Trajectory Recorder	SpaceX - Falcon 9 (Rocket Traj
Distance	X 2.54216(Y 555.073 Z 1.15813

Behavior Parameters
Decision Requester

Script	DecisionRequester
Decision Period	5
Take Actions Between	✓

Rocket Trajectory Recorder (Script)

Script	RocketTrajectoryRecorder
Rocket	SpaceX - Falcon 9 (Transform)
Record Interval	0.1

Assets > 4-NN > config > ppo > results > Limited_TVC > Batch_1 > Test_3_All_Lvls > Rocket_Behavior

Test_2
Test_3_Al
Rocket
run_log
Rest
7Mars_T\
=TVC_60
=TVC_60
Legit_Lev
Level1_1€
Level1_1€
Level1_1€

Rocket_Behavior-439999780
Rocket_Behavior-449999976
Rocket_Behavior-449999976
Rocket_Behavior-459999815
Rocket_Behavior-459999815
Rocket_Behavior-469999490
Rocket_Behavior-469999490
Rocket_Behavior-475243410
Rocket_Behavior-475243410
Rocket_Behavior-496300743

Activate Windows
Go to Settings to activate Windows.

Figure 3.5: Unity Interface

1. **Hierarchy:** This panel acts as an organizational hub, displaying a tree-like structure of all objects present in the simulation scene. Similar to an organizational chart, the hierarchy allows for easy selection and manipulation of individual objects, such as the rocket, landing pad, or background elements.

2. **Scene (Editor View)**: The Scene view, which we'll refer to as the "Editor View", serves as the primary workspace for constructing and editing our simulation environment. Here's a breakdown of its key functionalities:

- **Building the Environment:** This view acts as our virtual canvas, where we can place and arrange objects like the rocket, landing pad, and background scenery. It allows us to manipulate these objects directly, similar to working with a 3D modeling program.
- **Visualization and Editing:** The Scene view offers a real-time representation of your simulation environment. You can zoom in and out, rotate the view, and navigate around the scene to inspect details and make adjustments. Additionally, the Scene view allows for selection and manipulation of individual objects within the environment. You can use tools like the Transform gizmos to precisely position, rotate, and scale these objects.
- **Raycast Visualization:** The Scene view also displays raycasts. These are visual aids that represent lines with different colors extending from the rocket in the scene. Raycasts can be helpful for debugging purposes, and are used in gathering information also, allowing us to visualize how objects might interact with each other.

3. **Cameras (Simulation View):** The Cameras view, which we'll refer to as the "Sim View", acts as a dedicated window for visualizing the final rendered output of our simulation. Imagine it as a real-time camera feed capturing what the user would experience during the actual simulation:

   - **Camera Perspective:** This view showcases the scene exclusively through the lens of the virtual cameras we've positioned within our environment. This allows us to verify how the simulation will appear when running without the clutter of the Editor tools present in the Scene view.
   - **Optimized Rendering:** Unlike the Scene view, the Sim view focuses on optimized rendering, providing a smooth and realistic representation of the simulation environment. This helps ensure that the visual experience for the user during the simulation is optimized.

In essence, the Scene view serves as our editing and construction workspace, while the Sim view provides a dedicated window for visualizing the final rendered output of our simulation through the designated cameras. By effectively utilizing both views, we can create a visually compelling and informative rocket landing simulation environment.

4. **Inspector:** Imagine a detailed control panel - that's precisely what the Inspector offers. When we select an object in the Hierarchy, the Inspector displays its properties and settings. This allows us to fine-tune aspects like the object's position, rotation, scale, material properties, and even behaviors if they are scripted.

5. **Project/Console Area:** The bottom bar typically houses the Project panel. This panel serves as a file management system, providing an overview of all assets (images, models, scripts) imported into our project. The bottom section also displays the Console panel (when selected), which relays important messages, warnings, logs, or errors during development.

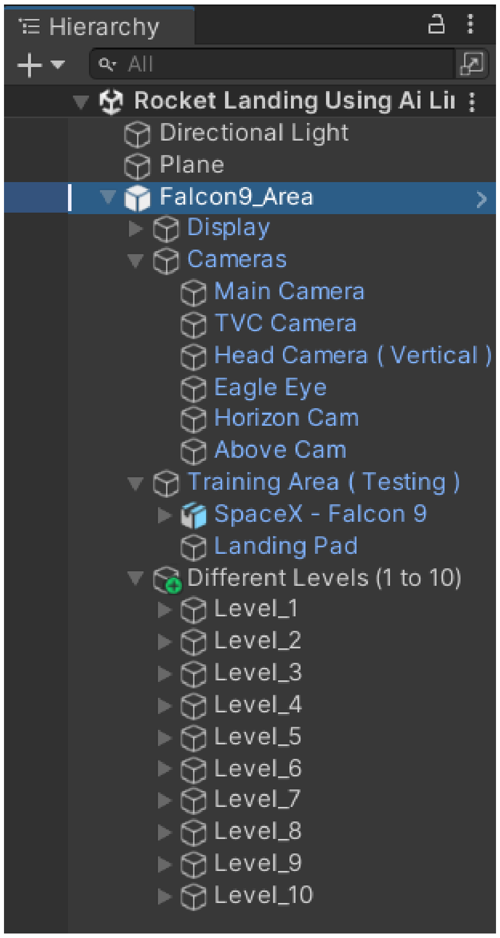
By effectively utilizing these key sections of the Unity interface, we can construct, customize, and interact with the virtual landing environment, paving the way for a robust and informative rocket landing simulation

3.1.4 Project Hierarchy

This section delves into the organization of our rocket landing simulation project within the Unity interface, specifically focusing on the objects and their hierarchical relationships. This hierarchy plays a crucial role in managing and manipulating elements within the simulation.

Plane: A single plane object, which serves as the virtual ground for our launch and landing area

Main Focus: "Falcon9 Area" This parent object acts as a container, encompassing all elements directly associated with the rocket and its landing environment:



```
Rocket Landing Using Ai Li
  Directional Light
  Plane
  Falcon9_Area
    Display
    Cameras
      Main Camera
      TVC Camera
      Head Camera ( Vertical )
      Eagle Eye
      Horizon Cam
      Above Cam
    Training Area ( Testing )
      SpaceX - Falcon 9
      Landing Pad
    Different Levels (1 to 10)
      Level_1
      Level_2
      Level_3
      Level_4
      Level_5
      Level_6
      Level_7
      Level_8
      Level_9
      Level_10
```

Figure 3.6: Hierarchy

- **Display:** This child object is responsible for displaying crucial information on the screen throughout the simulation. Examples include velocity (vertical and horizontal), overall speed, rocket position relative to the landing pad, distance to landing, and potentially reward indicators.

- **Camera:** This child object serves as a container for various camera setups within the simulation:

  1. **Main Camera:** This grandchild object offers a comprehensive view of the rocket, dynamically following it as it descends towards the landing pad.

  2. **TVC Camera:** Focused on the bottom of the rocket setup, this grandchild camera allows for observing the activation of the Thrust Vector Control (TVC) system.

3. **Different POV Camera:** This grandchild object provides an alternative perspective to the rocket's descent, offering a different viewpoint for analysis.
4. **Eagle Eye:** This grandchild camera provides a bird's-eye view, encompassing the entire simulation environment.
5. **Horizon Cam:** This grandchild camera offers a horizontal view of the simulation environment, focusing on the horizon line.
6. **Above Cam:** Positioned above all landing pads, this grandchild camera grants an overview of the simulation, allowing for visual confirmation of successful (green landing pad) and failed (red landing pad) attempts.

- **Training Area:** This child object encompasses all elements related to the training environment:

1. **SpaceX - Falcon 9:** This grandchild object represents the virtual rocket model used in the simulation.

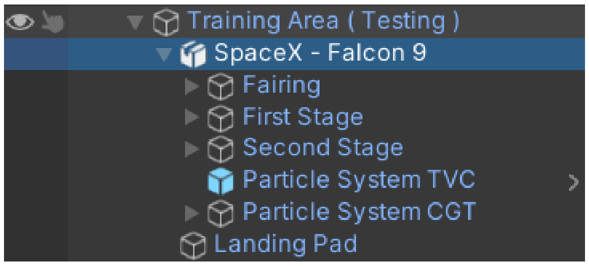


Figure 3.7: Falcon9 Hierarchy

- Fairing: This grandchild object represents the protective nose cone of the rocket.
- First Stage: This grandchild object represents the main booster stage of the rocket.
- Second Stage: This grandchild object represents the upper stage of the rocket that carries the payload.
- Particle System TVC: This grandchild object is a particle system that visually depicts the activation of the Thrust Vector Control system.

- Particle System CGT: his grandchild object is a particle system that visually depicts the activation of the Cold Gas Trusters system.

2. **Landing Pad:** This grandchild object represents the designated landing zone for the rocket

- **Different Levels (1 to 10):** This child object acts as a container for various difficulty levels within the simulation. Each level (Level 1 to Level 10) is a grandchild object containing specific configurations or challenges for the rocket landing.

3.1.5 Conceptual Overview

This concept map provides a visual representation of the core components and their interactions within the reinforcement learning framework designed for training an autonomous rocket landing agent. The map illustrates the key elements and how they work together to achieve successful landing simulations.

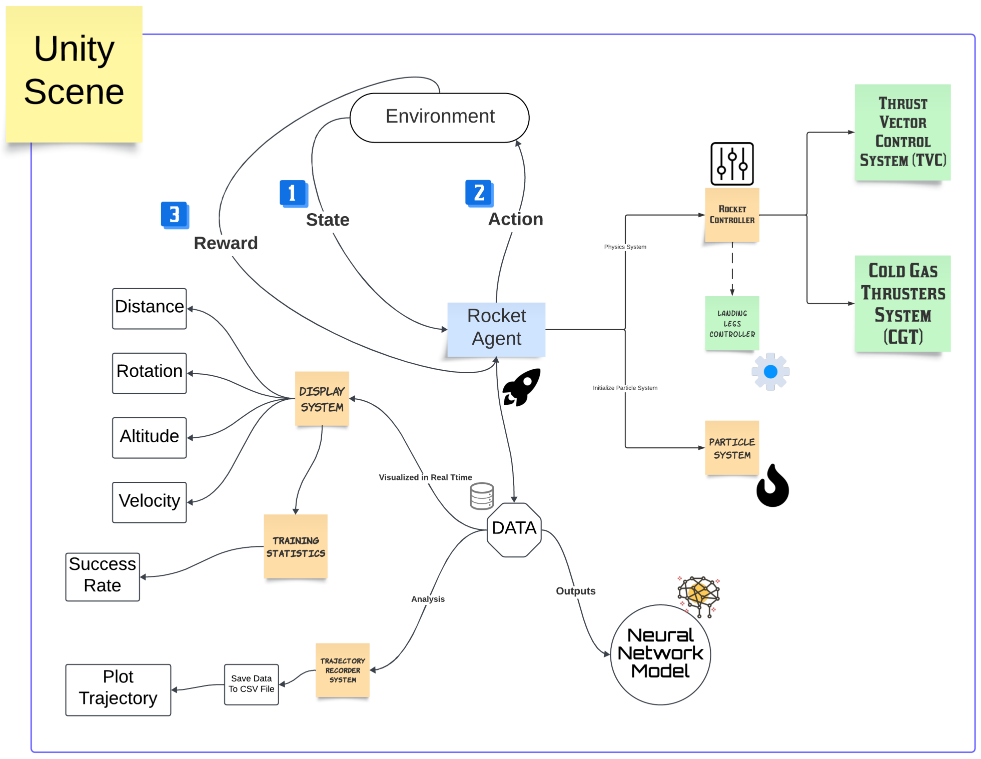


Figure 3.8: Concept Map

3.1.6 Simulation Environment Design

The cornerstone of our rocket landing simulation is the meticulously crafted virtual environment. This environment serves as the training ground where the autonomous landing maneuvers are honed. We'll begin by taking a closer look at the core elements of this environment, focusing on the rocket and landing pad. We'll then delve into the innovative approach used to create multiple training areas, fostering a diverse and challenging training landscape for the AI.

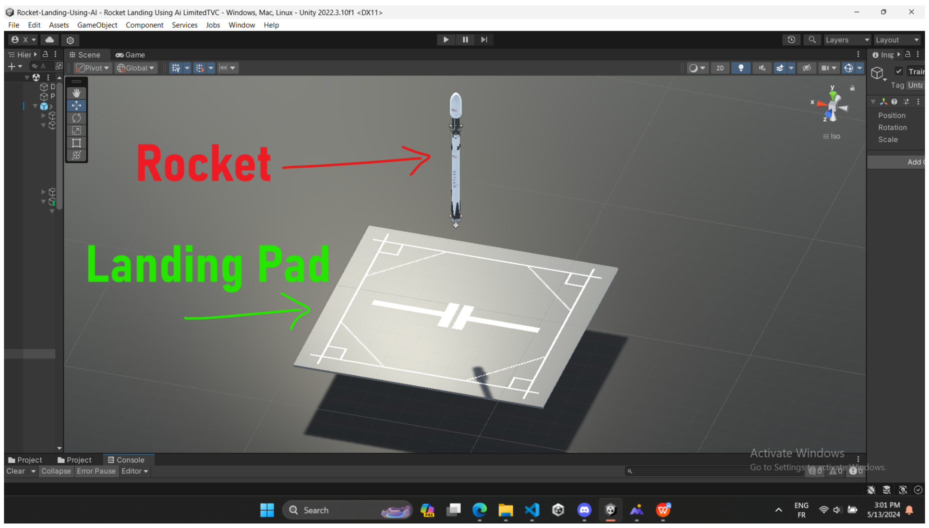

Rocket-Landing-Using-AI - Rocket Landing Using Ai LimitedTVC - Windows, Mac, Linux - Unity 2022.3.10f1 <DX11>
File Edit Assets GameObject Component Services Jobs Window Help
Scene
Game
Rocket
Landing Pad
Position
Rotation
Scale
Project
Console
Activate Windows
Go to Settings to activate Windows.

Figure 3.9: Simulation Environment Design

3.1.7 Description of the rocket model in Unity

Our simulated rocket (Figure 3.10) closely resembles a real-world model, mirroring the hierarchical structure of the SpaceX Falcon 9 (Figure 3.11; reference: [36]).

This translates to a three-stage design:

1. **Fairing:** The fairing acts as a protective enclosure for the payload during launch. It's jettisoned at a predetermined altitude.

2. **First Stage:** This primary stage powers the initial liftoff and ascent phase and also the re-entry and landing phase.

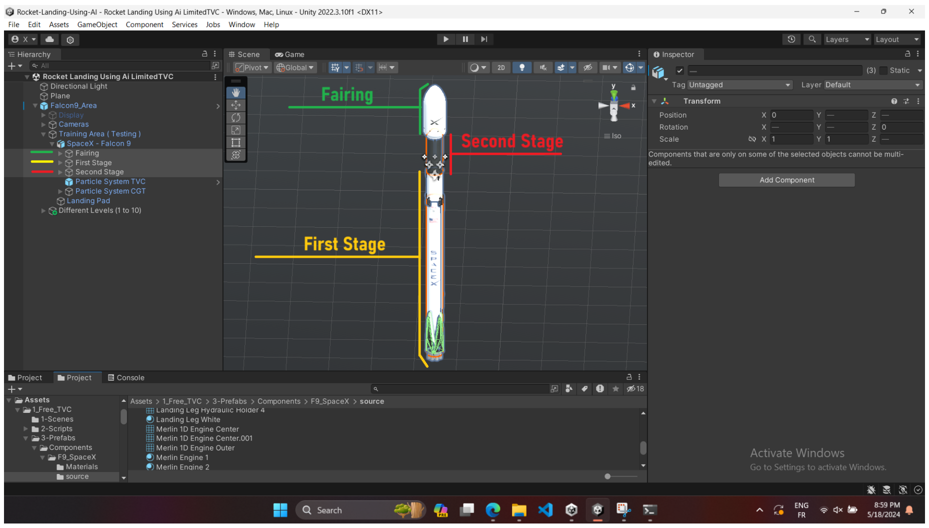
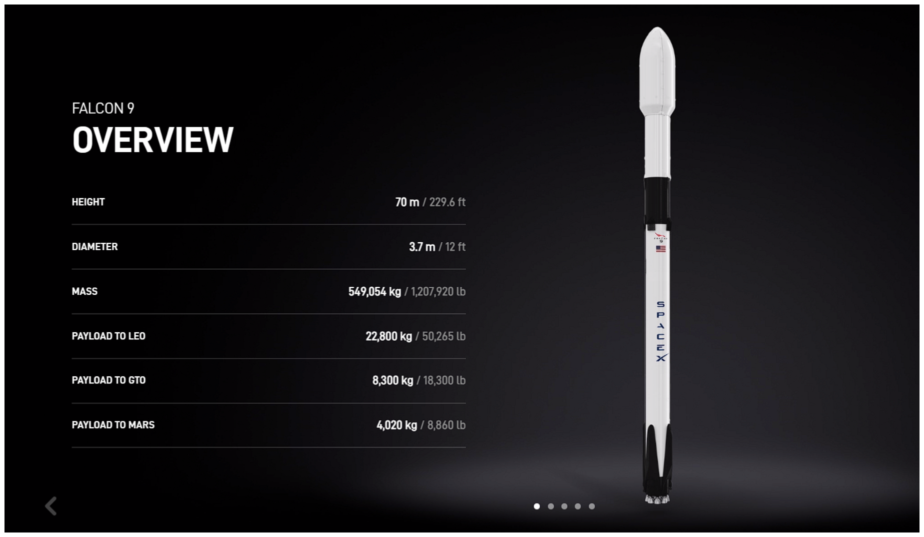


Figure 3.10: Our Project Falcon-9 Model



FALCON 9

OVERVIEW

HEIGHT	**70 m** / 229.6 ft
DIAMETER	**3.7 m** / 12 ft
MASS	**549,054 kg** / 1,207,920 lb
PAYLOAD TO LEO	**22,800 kg** / 50,265 lb
PAYLOAD TO GTO	**8,300 kg** / 18,300 lb
PAYLOAD TO MARS	**4,020 kg** / 8,860 lb

Figure 3.11: SpaceX Official Falcon-9 Model [36]

- **Landing Legs:** Landing legs are the sturdy supports attached to the bottom of the first stage. These unsung heroes play a critical role during the rocket's return to Earth. As the first stage reenters the atmosphere, landing legs withstand immense heat and aerodynamic forces. Upon touchdown, they absorb the impact, ensuring a controlled descent and preventing damage to the rocket and launch infrastructure. In essence, landing legs play an important role in the safe return of the reusable rocket.

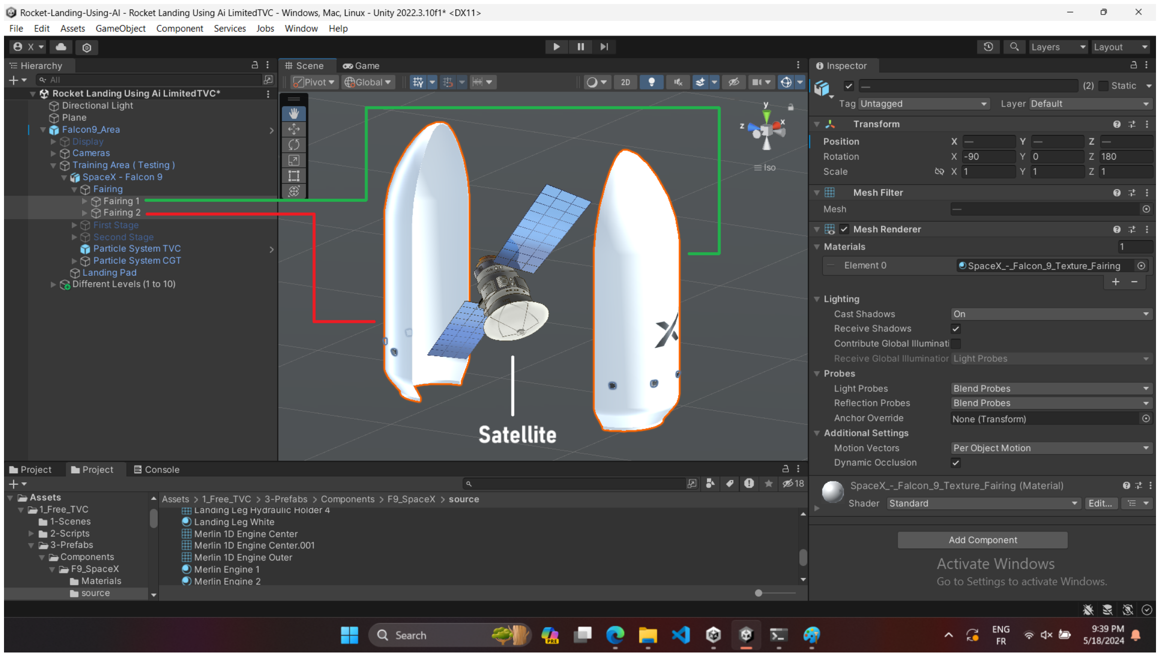


Figure 3.12: Falcon-9 Fairing

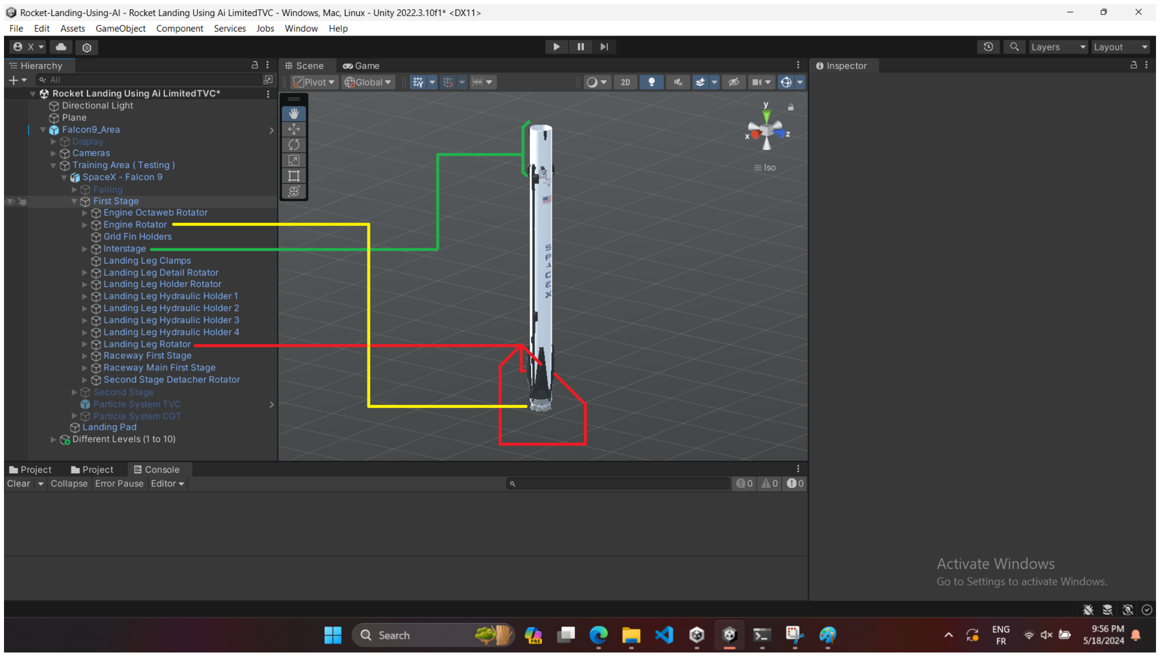
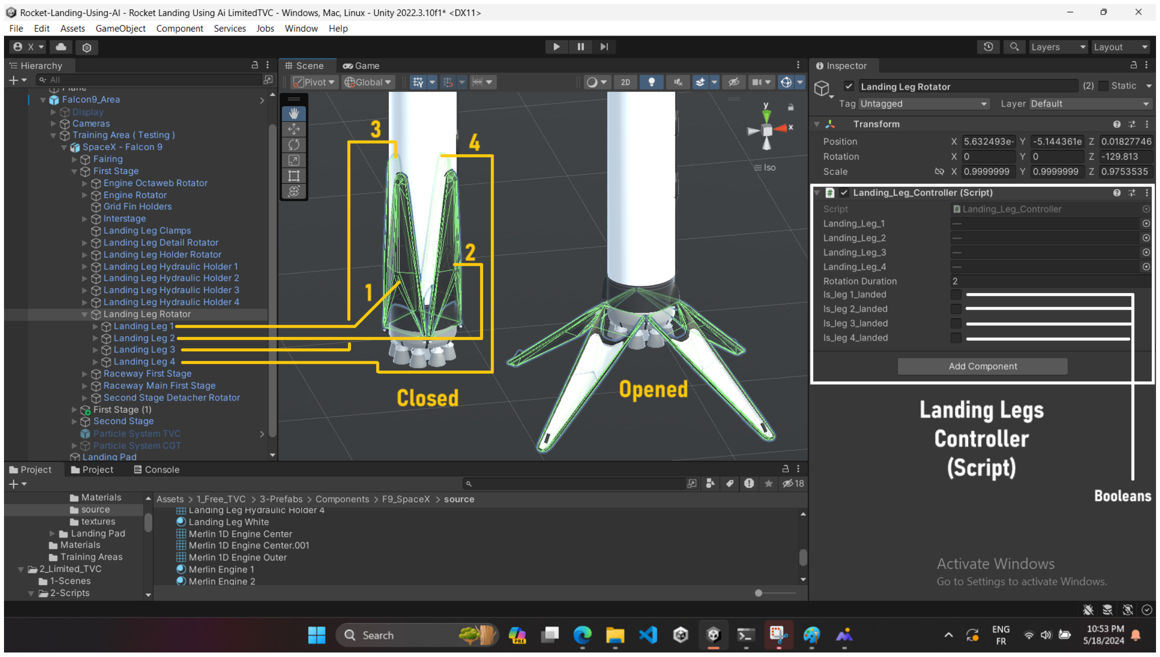

Rocket-Landing-Using-AI - Rocket Landing Using Ai LimitedTVC - Windows, Mac, Linux - Unity 2022.3.10f1* <DX11>
Hierarchy
Rocket Landing Using Ai LimitedTVC*
Directional Light
Plane
Falcon9_Area
Display
Cameras
Training Area (Testing)
SpaceX - Falcon 9
Fairing
First Stage
Engine Octaweb Rotator
Engine Rotator
Grid Fin Holders
Interstage
Landing Leg Clamps
Landing Leg Detail Rotator
Landing Leg Holder Rotator
Landing Leg Hydraulic Holder 1
Landing Leg Hydraulic Holder 2
Landing Leg Hydraulic Holder 3
Landing Leg Hydraulic Holder 4
Landing Leg Rotator
Raceway First Stage
Raceway Main First Stage
Second Stage Detacher Rotator
Second Stage
Particle System TVC
Particle System CGT
Landing Pad
Different Levels (1 to 10)
Scene
Game
Inspector
Project
Console
Clear
Collapse
Error Pause
Editor

Figure 3.13: Falcon-9 First Stage

Having a realistic landing leg model is crucial for achieving a controlled touchdown. To simulate the behavior of the landing legs and implement control logic, we developed a dedicated landing leg controller component. This component interacts with the physics engine to accurately reflect the physical properties and movements of the landing legs


Rocket-Landing-Using-AI - Rocket Landing Using Ai LimitedTVC - Windows, Mac, Linux - Unity 2022.3.10f1* <DX11>
Hierarchy
Scene
Game
Inspector
3
4
2
1
Closed
Opened
Landing Legs Controller (Script)
Booleans
Landing Leg 1
Landing Leg 2
Landing Leg 3
Landing Leg 4
Landing Leg Rotator

Landing Leg Rotator (2) Static
Tag Untagged Layer Default

Transform
Position X 5.632493e- Y -5.144361e Z 0.01827746
Rotation X 0 Y 0 Z -129.813
Scale X 0.9999999 Y 0.9999999 Z 0.9753535

Landing_Leg_Controller (Script)
Script Landing_Leg_Controller
Landing_Leg_1 —
Landing_Leg_2 —
Landing_Leg_3 —
Landing_Leg_4 —
Rotation Duration 2
Is_leg 1_landed
Is_leg 2_landed
Is_leg 3_landed
Is_leg 4_landed

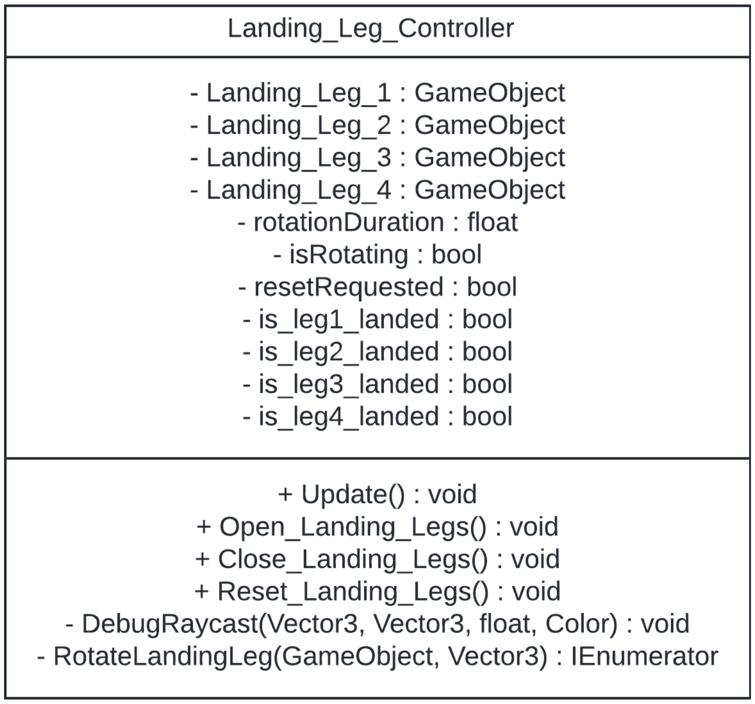
Figure 3.14: Falcon-9 Landing Legs

during descent.

To provide a deeper understanding of the landing leg control implementation, we present the following:

(a) **LandingLegController Class Diagram:** To provide a foundational understanding of the landing leg control logic, we present the LandingLegController class diagram (Figure 3.15). This diagram focuses on the internal structure of the class, illustrating the various attributes and methods that work together to manage the landing leg behavior.

(b) **Landing Legs Controller Sequence Diagram** This sequence diagram illustrates the message flow between the LandingLegController and other relevant objects during the deployment of the landing legs. It clarifies the sequence of events and interactions necessary to extend the landing legs from their stowed position (Figure 3.16).



Landing_Leg_Controller
- Landing_Leg_1 : GameObject - Landing_Leg_2 : GameObject - Landing_Leg_3 : GameObject - Landing_Leg_4 : GameObject - rotationDuration : float - isRotating : bool - resetRequested : bool - is_leg1_landed : bool - is_leg2_landed : bool - is_leg3_landed : bool - is_leg4_landed : bool
+ Update() : void + Open_Landing_Legs() : void + Close_Landing_Legs() : void + Reset_Landing_Legs() : void - DebugRaycast(Vector3, Vector3, float, Color) : void - RotateLandingLeg(GameObject, Vector3) : IEnumerator

Figure 3.15: Landing Legs Controller Internal structure of the class

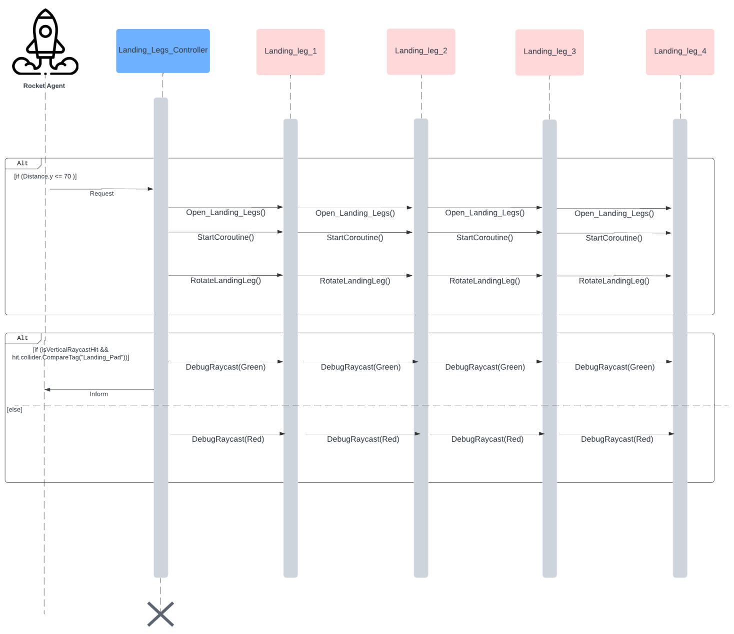


Figure 3.16: Landing Legs Controller Sequence Diagram

3. **Second Stage:** Taking over after first-stage separation, the second stage provides propulsion for the remainder of the ascent.

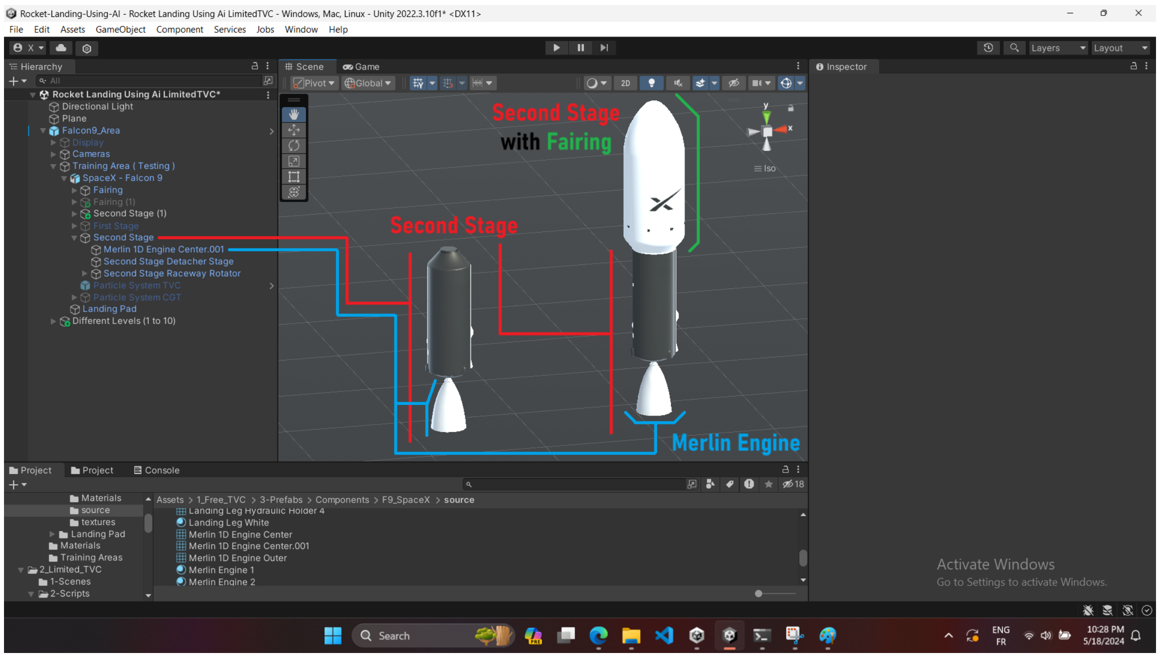

Rocket-Landing-Using-AI - Rocket Landing Using Ai LimitedTVC - Windows, Mac, Linux - Unity 2022.3.10f1* <DX11>
Hierarchy
Rocket Landing Using Ai LimitedTVC*
Directional Light
Plane
Falcon9_Area
Display
Cameras
Training Area (Testing)
SpaceX - Falcon 9
Fairing
Fairing (1)
Second Stage (1)
First Stage
Second Stage
Merlin 1D Engine Center.001
Second Stage Detacher Stage
Second Stage Raceway Rotator
Particle System TVC
Particle System CGT
Landing Pad
Different Levels (1 to 10)
Scene
Game
Inspector
Second Stage with Fairing
Second Stage
Merlin Engine
Project
Console
Assets > 1_Free_TVC > 3-Prefabs > Components > F9_SpaceX > source
Landing Leg Hydraulic Holder 4
Landing Leg White
Merlin 1D Engine Center
Merlin 1D Engine Center.001
Merlin 1D Engine Outer
Merlin Engine 1
Merlin Engine 2
Activate Windows
Go to Settings to activate Windows.

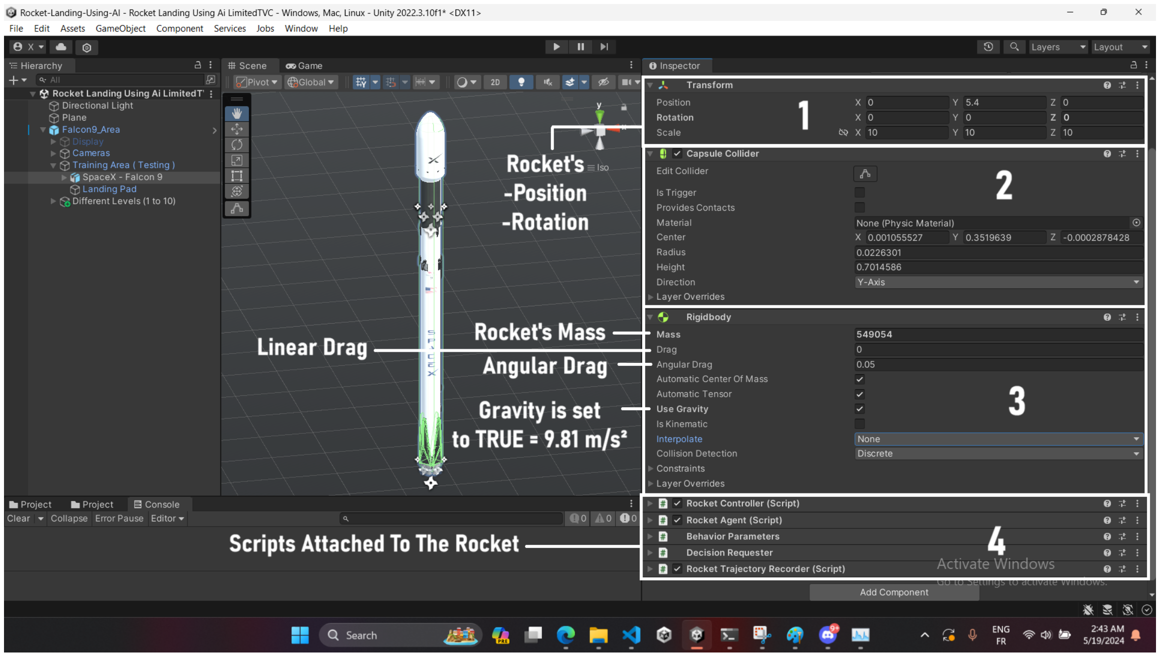
Figure 3.17: Falcon-9 Second Stage

3.1.8 Rocket Control and Scripting

The simulated rocket is equipped with a control system that guides its descent towards the landing pad. This system relies on various sensors (e.g., gyroscopes, accelerometers) to gather real-time data about the rocket's orientation and movement. Additionally, high-level control logic is implemented using scripts. These scripts interpret sensor data and send commands to the control system, dictating actions like adjusting engine thrust or initiating landing leg deployment at specific points during the descent. Scripting offers flexibility in defining and testing various control strategies, allowing us to optimize the rocket's landing performance.

Figure 3.18 depicts the rocket prefab within the scene editor, along with its inspector window showcasing the attached components. These components play a crucial role in simulating the rocket's behavior and enabling its control during the landing phase.

1. **Transform:** This fundamental component stores the rocket's position and rotation data in 3D space. This information is vital for various purposes, including:



Transform

	X	Y	Z
Position	0	5.4	0
Rotation	0	0	0
Scale	10	10	10

Capsule Collider

Edit Collider	
Is Trigger	☐
Provides Contacts	☐
Material	None (Physic Material)
Center	X 0.001055527 Y 0.3519639 Z -0.0002878428
Radius	0.0226301
Height	0.7014586
Direction	Y-Axis
Layer Overrides	

Rigidbody

Mass	549054
Drag	0
Angular Drag	0.05
Automatic Center Of Mass	✓
Automatic Tensor	✓
Use Gravity	✓
Is Kinematic	☐
Interpolate	None
Collision Detection	Discrete
Constraints	
Layer Overrides	

Rocket Controller (Script)
Rocket Agent (Script)
Behavior Parameters
Decision Requester
Rocket Trajectory Recorder (Script)

Figure 3.18: Rocket's Inspector Components

- Information Retrieval: Scripts can access the transform component to obtain the current position and orientation of the rocket during its descent.
- State Manipulation: The transform component allows scripts to reset the rocket to a new starting position and orientation for each landing attempt. This is typically achieved by randomly modifying the transform values within a predefined range.

2. **Capsule Collider:** This component defines the rocket's collision volume as a capsule shape. This collider is essential for enabling realistic physics interactions. The capsule collider ensures that the rocket collides with other objects in the environment (such as the landing pad) and prevents it from passing through them unrealistically.

3. **RigidBody:** The rigidbody component governs the rocket's physical properties that influence its movement during the simulation. Key attributes within this component include:

   - Mass: Defines the overall weight of the rocket, affecting its momentum and inertia.
   - Drag and Angular Drag: These properties simulate air resistance, impacting the rocket's velocity and rotational speed.

- Center of Mass: This point represents the mass distribution within the rocket, influencing its rotational behavior.
- Gravity: Enabling or disabling gravity allows control over the simulated gravitational force acting on the rocket.

4. **Scripts:** The "Scripts" section of the inspector window displays the various scripts attached to the rocket prefab. These scripts are responsible for implementing the high-level control logic for the rocket during descent. We will delve into the functionalities of each script in the next section.

3.1.9 Main Code Components:

- **Rocket Prefab:** This prefab serves as the blueprint for our simulated rocket. It encapsulates various environment objects representing the rocket's physical structure (e.g., mesh, materials, componenets) and other objects like the landing pad, particle systems and so on.

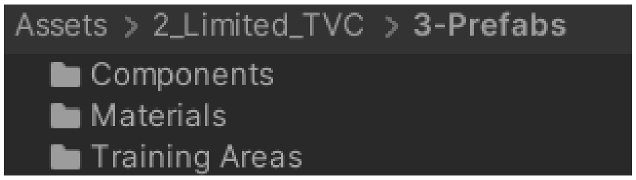

Assets > 2_Limited_TVC > 3-Prefabs
Components
Materials
Training Areas

Figure 3.19: Prefabs

- **Rocket Scripts:** These are the core components controlling the rocket's functionality. Here are the main scripts relevant to our rocket control:

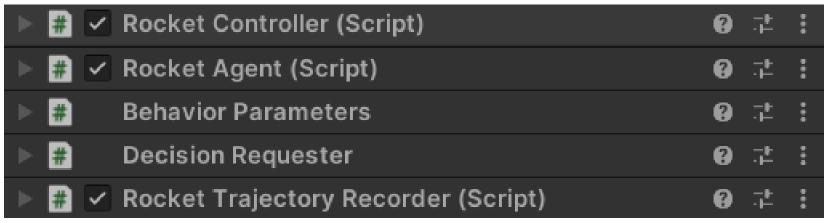

✓ Rocket Controller (Script)
✓ Rocket Agent (Script)
Behavior Parameters
Decision Requester
✓ Rocket Trajectory Recorder (Script)

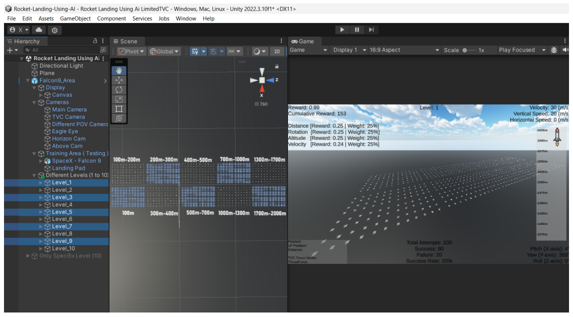
Figure 3.20: Rocket Attached Scripts

1. **Rocket Controller:** Manages overall rocket movement based on physics interactions and script instructions.
2. **Rocket Agent**: Implements an intelligent control strategy using reinforcement learning.
3. **Behavior Parameters:** Holds configurable settings that influence the behavior of various scripts during the simulation.
4. **Decision Requester**: Facilitates communication between the rocket agent and the simulation environment, allowing the agent to request decisions based on observed states.
5. **Rocket Trajectory Recorder:** Captures the rocket's trajectory data throughout the simulation run for analysis purposes.

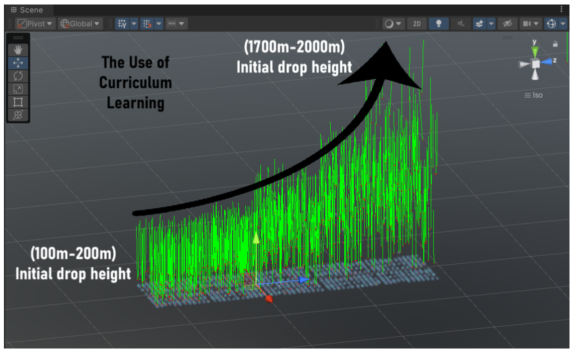
- **Unity Physics Engine:** The Unity physics engine provides the foundation for simulating the rocket's physical behavior. It processes interactions between the rocket and its environment, including gravity, collisions, and forces applied by the scripts.

3.2 Curriculum Learning

The figure 3.21a depicts the curriculum learning approach [37] employed in training the rocket agent. It showcases ten batches, each containing 96 training areas. Each batch represents an incremental increase in the initial drop height range for the rocket agent, as detailed in Table 4.2. The figure visually depicts these ranges, such as "[1700m - 2000m]" for one batch. The figure on the right 3.21b provides a visual representation of the rocket agent during training at different heights. It complements the curriculum learning approach depicted in the left figure.



(a) Curriculum Learning Approach


The Use of Curriculum Learning
(1700m-2000m) Initial drop height
(100m-200m) Initial drop height

(b) Visualization of Agent Training (Execution)

Figure 3.21: Curriculum Learning for Progressive Rocket Agent Training

3.3 Code Implementation and Rocket Control:

3.3.1 Rocket Controller:

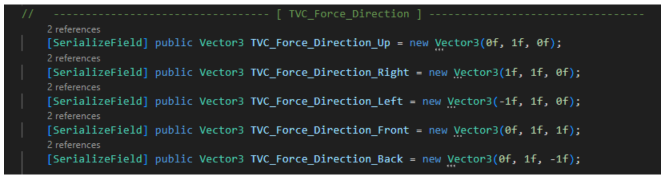
The RocketController script is a crucial component of the rocket simulation, responsible for managing the rocket's thrust, orientation, and various control systems. This script leverages Unity's physics engine to apply forces and simulate realistic rocket behavior.

Key Components:

- Thrust Variables:
  - The serialized fields ***thrustForce_Up***, ***thrustForce_Right***, ***thrustForce_Left***, ***thrustForce_Front***, and ***thrustForce_Back*** allow for configuring the thrust forces in different directions, measured in Newtons (N).
- Force Direction Vectors:
  - These vectors: ***TVC_Force_Direction_Up***,
    TVC_Force_Direction_Right,
    TVC_Force_Direction_Left,
    TVC_Force_Direction_Front,

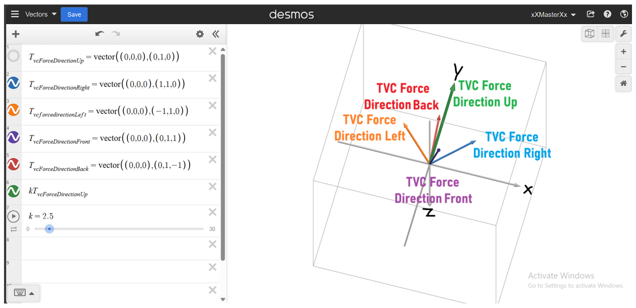
TVC_Force_Direction_Back

define the directions of the thrust forces applied during Thrust Vector Control (TVC).



```
//   -------------------------------- [ TVC_Force_Direction ] --------------------------------
    2 references
    [SerializeField] public Vector3 TVC_Force_Direction_Up = new Vector3(0f, 1f, 0f);
    2 references
    [SerializeField] public Vector3 TVC_Force_Direction_Right = new Vector3(1f, 1f, 0f);
    2 references
    [SerializeField] public Vector3 TVC_Force_Direction_Left = new Vector3(-1f, 1f, 0f);
    2 references
    [SerializeField] public Vector3 TVC_Force_Direction_Front = new Vector3(0f, 1f, 1f);
    2 references
    [SerializeField] public Vector3 TVC_Force_Direction_Back = new Vector3(0f, 1f, -1f);
```

Figure 3.22: TVC Force Direction Declaration



$T_{vcForceDirectionUp} = \text{vector}((0,0,0),(0,1,0))$

$T_{vcForceDirectionRight} = \text{vector}((0,0,0),(1,1,0))$

$T_{vcforcedirectionLeft} = \text{vector}((0,0,0),(-1,1,0))$

$T_{vcForceDirectionFront} = \text{vector}((0,0,0),(0,1,1))$

$T_{vcForceDirectionBack} = \text{vector}((0,0,0),(0,1,-1))$

$kT_{vcForceDirectionUp}$

$k = 2.5$

Figure 3.23: Math Graph Plotting TVC Force Vectors

– and These vectors:

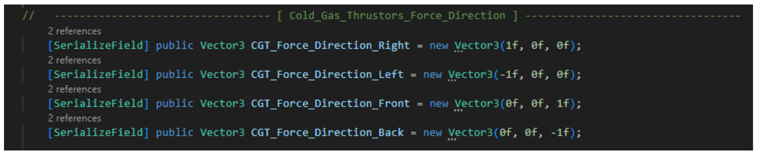
CGT_Force_Direction_Right,

CGT_Force_Direction_Left,

CGT_Force_Direction_Front,

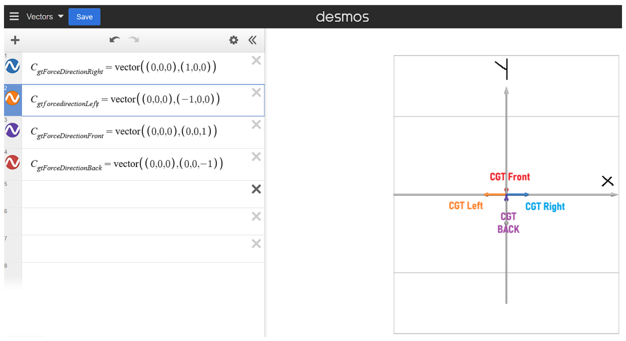
CGT_Force_Direction_Back

define the directions of the forces applied by the Cold Gas Thrusters (CGT).



```
//   -------------------------------- [ Cold_Gas_Thrustors_Force_Direction ] --------------------------------
    2 references
    [SerializeField] public Vector3 CGT_Force_Direction_Right = new Vector3(1f, 0f, 0f);
    2 references
    [SerializeField] public Vector3 CGT_Force_Direction_Left = new Vector3(-1f, 0f, 0f);
    2 references
    [SerializeField] public Vector3 CGT_Force_Direction_Front = new Vector3(0f, 0f, 1f);
    2 references
    [SerializeField] public Vector3 CGT_Force_Direction_Back = new Vector3(0f, 0f, -1f);
```

Figure 3.24: CGT Force Direction Declaration



$C_{gtForceDirectionRight} = \text{vector}((0,0,0),(1,0,0))$

$C_{gtforcedirectionLeft} = \text{vector}((0,0,0),(-1,0,0))$

$C_{gtForceDirectionFront} = \text{vector}((0,0,0),(0,0,1))$

$C_{gtForceDirectionBack} = \text{vector}((0,0,0),(0,0,-1))$

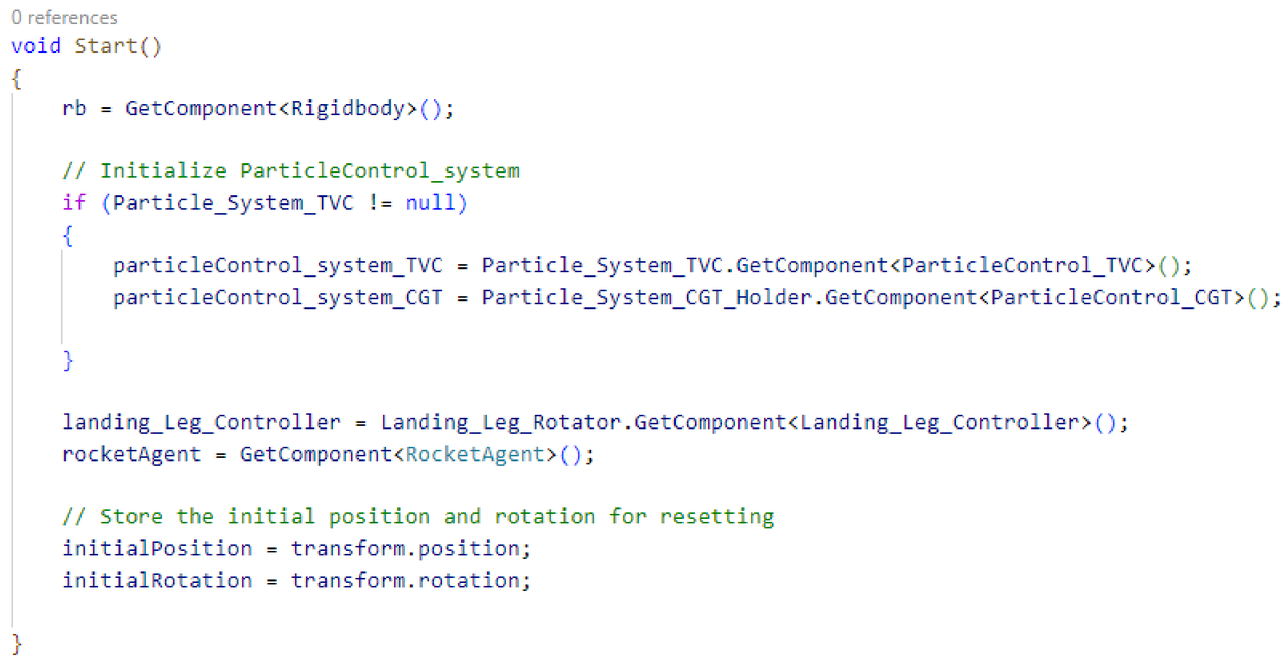
Figure 3.25: Math Graph Plotting CGT Force Vectors

- Control Variables:
  - ***MaxThrustSpeed***, ***Max_Thrust_Force_Change***, ***Desired_Time***: These variables calculate the maximum thrust speed and the desired time for force changes.
  - ***scale_weight***: A factor to scale the weight of the rocket for force calculations.
- Particle Systems:
  - ***Particle_System_TVC***, ***Particle_System_CGT_Holder***: Objects representing the visual effects for the TVC and CGT systems.
  - ***particleControl_system_TVC***, ***particleControl_system_CGT***: Control scripts for managing the particle systems.
- Landing Leg Control:
  - ***Landing_Leg_Rotator***, ***is_Landing_Legs_open***, ***landing_Leg_Controller***: These manage the deployment of the rocket's landing legs.
- Raycast for Landing:
  - ***hit***, ***localOffset***, ***raycastLength***, ***isVerticalRaycastHit***: These variables and the associated logic detect the proximity of the rocket to the landing pad.

Main Functions:

- **Start():**

  Initializes the Rigidbody, particle systems, and stores the initial position and rotation of the rocket.

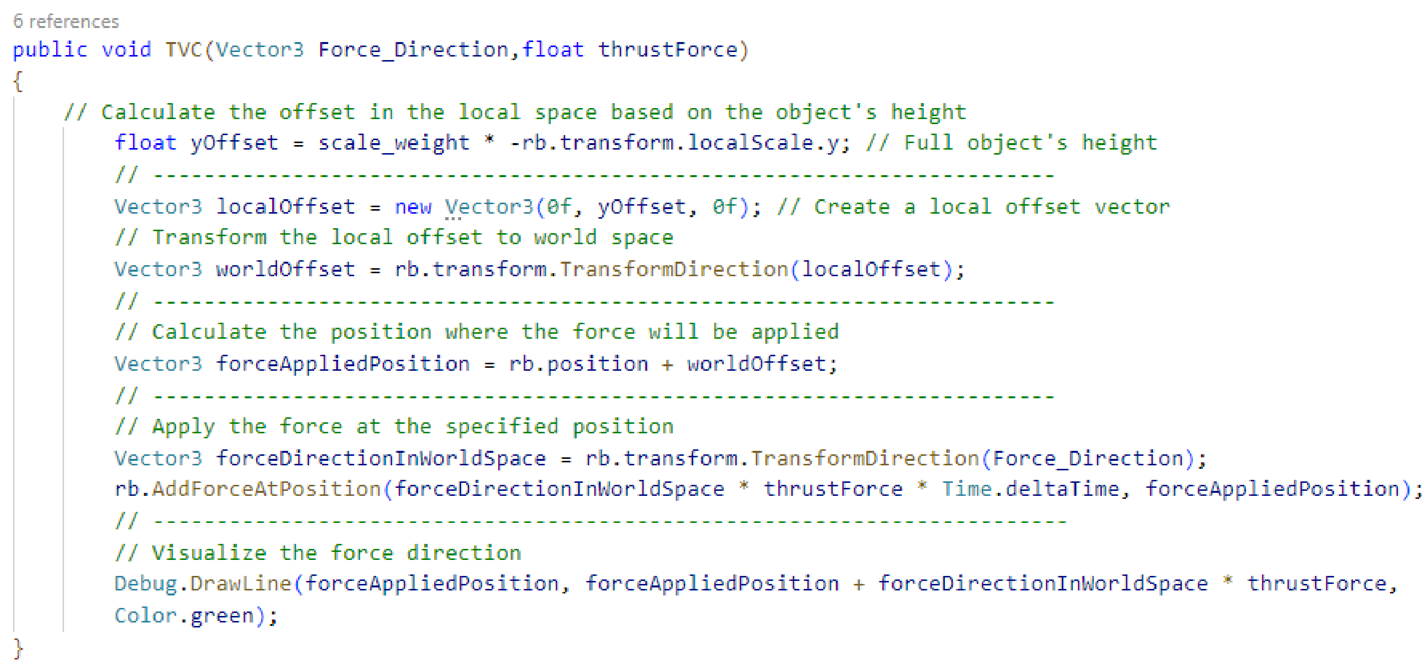


```
0 references
void Start()
{
    rb = GetComponent<Rigidbody>();

    // Initialize ParticleControl_system
    if (Particle_System_TVC != null)
    {
        particleControl_system_TVC = Particle_System_TVC.GetComponent<ParticleControl_TVC>();
        particleControl_system_CGT = Particle_System_CGT_Holder.GetComponent<ParticleControl_CGT>();

    }

    landing_Leg_Controller = Landing_Leg_Rotator.GetComponent<Landing_Leg_Controller>();
    rocketAgent = GetComponent<RocketAgent>();

    // Store the initial position and rotation for resetting
    initialPosition = transform.position;
    initialRotation = transform.rotation;

}
```

Figure 3.26: Start() Function

- **TVC(Vector3 Force_Direction, float thrustForce):**

  Applies a force in the specified direction using the TVC system.

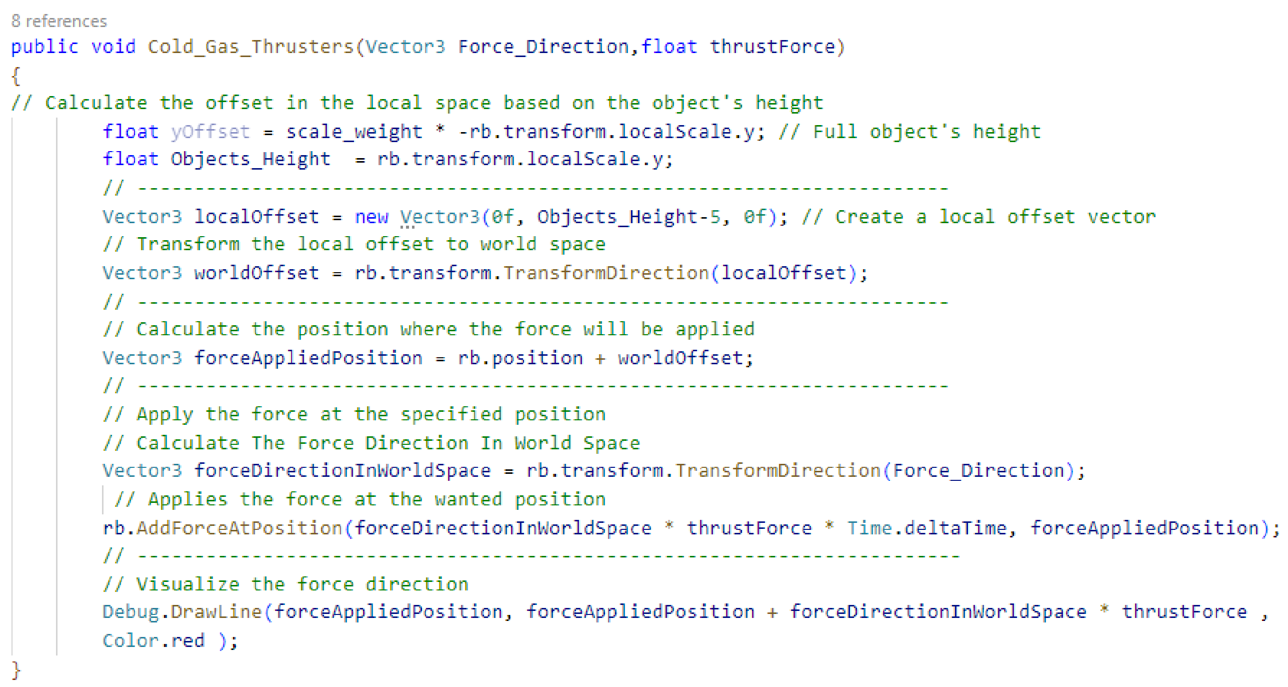


```
6 references
public void TVC(Vector3 Force_Direction,float thrustForce)
{
    // Calculate the offset in the local space based on the object's height
        float yOffset = scale_weight * -rb.transform.localScale.y; // Full object's height
        // -----------------------------------------------------------------------
        Vector3 localOffset = new Vector3(0f, yOffset, 0f); // Create a local offset vector
        // Transform the local offset to world space
        Vector3 worldOffset = rb.transform.TransformDirection(localOffset);
        // -----------------------------------------------------------------------
        // Calculate the position where the force will be applied
        Vector3 forceAppliedPosition = rb.position + worldOffset;
        // -----------------------------------------------------------------------
        // Apply the force at the specified position
        Vector3 forceDirectionInWorldSpace = rb.transform.TransformDirection(Force_Direction);
        rb.AddForceAtPosition(forceDirectionInWorldSpace * thrustForce * Time.deltaTime, forceAppliedPosition);
        // ------------------------------------------------------------------------
        // Visualize the force direction
        Debug.DrawLine(forceAppliedPosition, forceAppliedPosition + forceDirectionInWorldSpace * thrustForce,
        Color.green);
}
```

Figure 3.27: TVC() Function

- **Cold_Gas_Thrusters(Vector3 Force_Direction, float thrustForce):**

  Applies a force using the Cold Gas Thrusters.

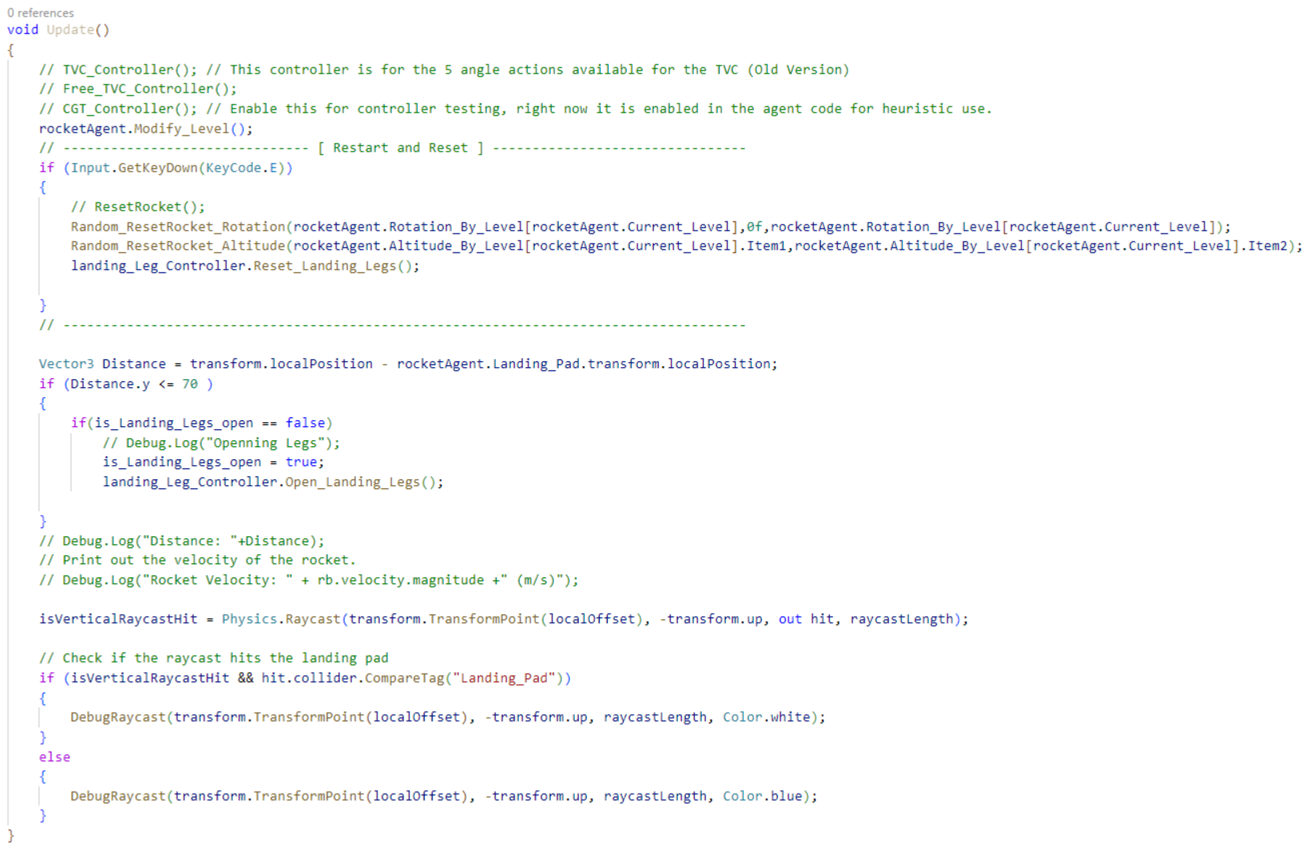


```
8 references
public void Cold_Gas_Thrusters(Vector3 Force_Direction,float thrustForce)
{
// Calculate the offset in the local space based on the object's height
        float yOffset = scale_weight * -rb.transform.localScale.y; // Full object's height
        float Objects_Height  = rb.transform.localScale.y;
        // -----------------------------------------------------------------------
        Vector3 localOffset = new Vector3(0f, Objects_Height-5, 0f); // Create a local offset vector
        // Transform the local offset to world space
        Vector3 worldOffset = rb.transform.TransformDirection(localOffset);
        // -----------------------------------------------------------------------
        // Calculate the position where the force will be applied
        Vector3 forceAppliedPosition = rb.position + worldOffset;
        // -----------------------------------------------------------------------
        // Apply the force at the specified position
        // Calculate The Force Direction In World Space
        Vector3 forceDirectionInWorldSpace = rb.transform.TransformDirection(Force_Direction);
         // Applies the force at the wanted position
        rb.AddForceAtPosition(forceDirectionInWorldSpace * thrustForce * Time.deltaTime, forceAppliedPosition);
        // ------------------------------------------------------------------------
        // Visualize the force direction
        Debug.DrawLine(forceAppliedPosition, forceAppliedPosition + forceDirectionInWorldSpace * thrustForce ,
        Color.red );
}
```

Figure 3.28: CGT() Function

- **Update():**

  Handles input for resetting the rocket, deploying landing legs, and performing raycast checks for landing.



```
0 references
void Update()
{
    // TVC_Controller(); // This controller is for the 5 angle actions available for the TVC (Old Version)
    // Free_TVC_Controller();
    // CGT_Controller(); // Enable this for controller testing, right now it is enabled in the agent code for heuristic use.
    rocketAgent.Modify_Level();
    // ------------------------------- [ Restart and Reset ] --------------------------------
    if (Input.GetKeyDown(KeyCode.E))
    {
        // ResetRocket();
        Random_ResetRocket_Rotation(rocketAgent.Rotation_By_Level[rocketAgent.Current_Level],0f,rocketAgent.Rotation_By_Level[rocketAgent.Current_Level]);
        Random_ResetRocket_Altitude(rocketAgent.Altitude_By_Level[rocketAgent.Current_Level].Item1,rocketAgent.Altitude_By_Level[rocketAgent.Current_Level].Item2);
        landing_Leg_Controller.Reset_Landing_Legs();

    }
    // ------------------------------------------------------------------------------------

    Vector3 Distance = transform.localPosition - rocketAgent.Landing_Pad.transform.localPosition;
    if (Distance.y <= 70 )
    {
        if(is_Landing_Legs_open == false)
            // Debug.Log("Openning Legs");
            is_Landing_Legs_open = true;
            landing_Leg_Controller.Open_Landing_Legs();

    }
    // Debug.Log("Distance: "+Distance);
    // Print out the velocity of the rocket.
    // Debug.Log("Rocket Velocity: " + rb.velocity.magnitude +" (m/s)");

    isVerticalRaycastHit = Physics.Raycast(transform.TransformPoint(localOffset), -transform.up, out hit, raycastLength);

    // Check if the raycast hits the landing pad
    if (isVerticalRaycastHit && hit.collider.CompareTag("Landing_Pad"))
    {
        DebugRaycast(transform.TransformPoint(localOffset), -transform.up, raycastLength, Color.white);
    }
    else
    {
        DebugRaycast(transform.TransformPoint(localOffset), -transform.up, raycastLength, Color.blue);
    }
}
```

Figure 3.29: Update()

Conclusion:

The ***RocketController*** script plays a vital role in simulating realistic rocket behavior by managing thrust forces, controlling various thruster systems, and ensuring safe landing procedures. Its integration with Unity's physics engine and various control mechanisms enables detailed and accurate simulations, contributing significantly to the project's overall success.

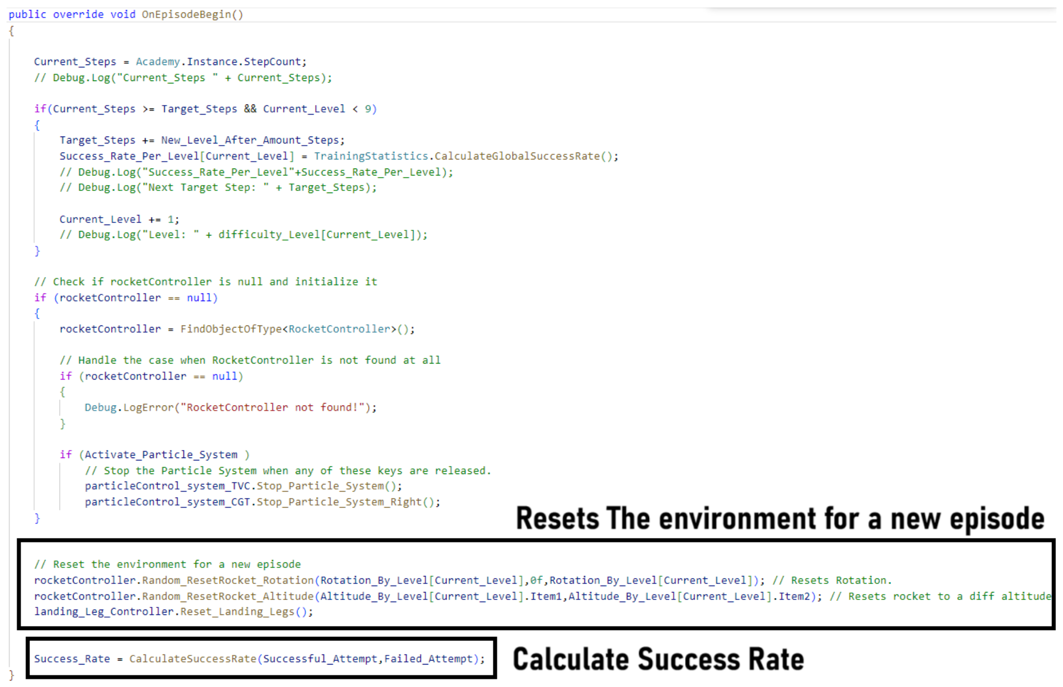
3.3.2 Rocket Agent:

The "Rocket Agent" is a core component of the simulation environment that leverages reinforcement learning to control the autonomous landing of a rocket. This agent is implemented using the Unity ML-Agents framework, which provides tools for training intelligent agents in dynamic environments. Below, we detail the implementation and functionality of the Rocket Agent script, highlighting the key elements and processes involved.

Main Methods:

- **OnEpisodeBegin():**

  This method [38]is invoked at the beginning of each training episode, serving to reset the environment, initialize essential components, and adjust the agent's training parameters based on the progress made in previous episodes. By systematically preparing the agent for a new episode, the OnEpisodeBegin method ensures a structured and effective learning process.

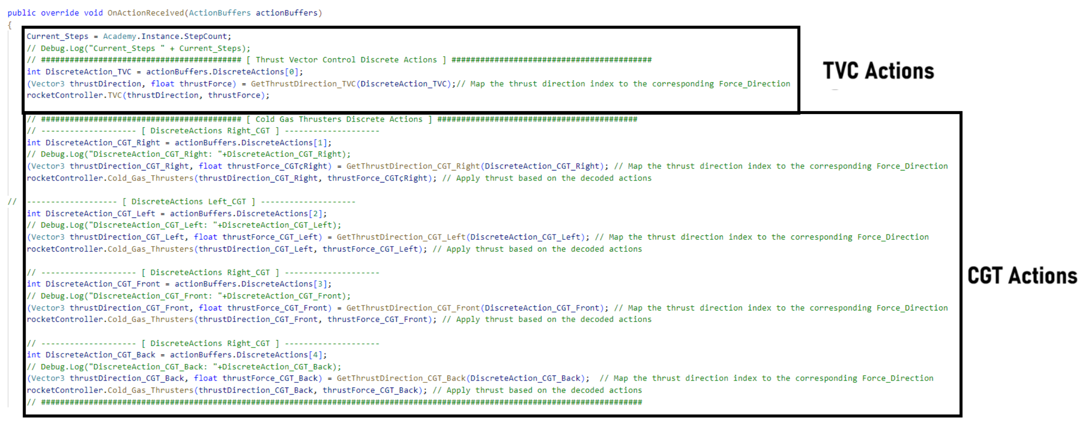


```
public override void OnEpisodeBegin()
{

    Current_Steps = Academy.Instance.StepCount;
    // Debug.Log("Current_Steps " + Current_Steps);

    if(Current_Steps >= Target_Steps && Current_Level < 9)
    {
        Target_Steps += New_Level_After_Amount_Steps;
        Success_Rate_Per_Level[Current_Level] = TrainingStatistics.CalculateGlobalSuccessRate();
        // Debug.Log("Success_Rate_Per_Level"+Success_Rate_Per_Level);
        // Debug.Log("Next Target Step: " + Target_Steps);

        Current_Level += 1;
        // Debug.Log("Level: " + difficulty_Level[Current_Level]);
    }

    // Check if rocketController is null and initialize it
    if (rocketController == null)
    {
        rocketController = FindObjectOfType<RocketController>();

        // Handle the case when RocketController is not found at all
        if (rocketController == null)
        {
            Debug.LogError("RocketController not found!");
        }

        if (Activate_Particle_System )
            // Stop the Particle System when any of these keys are released.
            particleControl_system_TVC.Stop_Particle_System();
            particleControl_system_CGT.Stop_Particle_System_Right();
    }

    // Reset the environment for a new episode
    rocketController.Random_ResetRocket_Rotation(Rotation_By_Level[Current_Level],0f,Rotation_By_Level[Current_Level]); // Resets Rotation.
    rocketController.Random_ResetRocket_Altitude(Altitude_By_Level[Current_Level].Item1,Altitude_By_Level[Current_Level].Item2); // Resets rocket to a diff altitude
    landing_Leg_Controller.Reset_Landing_Legs();

    Success_Rate = CalculateSuccessRate(Successful_Attempt,Failed_Attempt);
}
```

Figure 3.30: OnEpisodeBegin Method

- **OnActionReceived():**

  The OnActionReceived [39] method serves as the entry point for the agent to receive and process actions from the environment in a reinforcement learning setting. It interprets the actions provided by the environment, translates them into meaningful commands for the agent (such as controlling thrusters or adjusting orientation), calculates rewards based on the agent's actions and the environment's state, and determines whether an episode should continue or terminate based on predefined conditions. Essentially, it drives the decision-making process of the agent within its environment.



```
public override void OnActionReceived(ActionBuffers actionBuffers)
{
    Current_Steps = Academy.Instance.StepCount;
    // Debug.Log("Current_Steps " + Current_Steps);
    // ########################################## [ Thrust Vector Control Discrete Actions ] ##########################################
    int DiscreteAction_TVC = actionBuffers.DiscreteActions[0];
    (Vector3 thrustDirection, float thrustForce) = GetThrustDirection_TVC(DiscreteAction_TVC);// Map the thrust direction index to the corresponding Force_Direction
    rocketController.TVC(thrustDirection, thrustForce);

    // ########################################## [ Cold Gas Thrusters Discrete Actions ] ##########################################
    // -------------------- [ DiscreteActions Right_CGT ] --------------------
    int DiscreteAction_CGT_Right = actionBuffers.DiscreteActions[1];
    // Debug.Log("DiscreteAction_CGT_Right: "+DiscreteAction_CGT_Right);
    (Vector3 thrustDirection_CGT_Right, float thrustForce_CGTçRight) = GetThrustDirection_CGT_Right(DiscreteAction_CGT_Right); // Map the thrust direction index to the corresponding Force_Direction
    rocketController.Cold_Gas_Thrusters(thrustDirection_CGT_Right, thrustForce_CGTçRight); // Apply thrust based on the decoded actions

//  ------------------- [ DiscreteActions Left_CGT ] --------------------
    int DiscreteAction_CGT_Left = actionBuffers.DiscreteActions[2];
    // Debug.Log("DiscreteAction_CGT_Left: "+DiscreteAction_CGT_Left);
    (Vector3 thrustDirection_CGT_Left, float thrustForce_CGT_Left) = GetThrustDirection_CGT_Left(DiscreteAction_CGT_Left); // Map the thrust direction index to the corresponding Force_Direction
    rocketController.Cold_Gas_Thrusters(thrustDirection_CGT_Left, thrustForce_CGT_Left); // Apply thrust based on the decoded actions

    // -------------------- [ DiscreteActions Right_CGT ] --------------------
    int DiscreteAction_CGT_Front = actionBuffers.DiscreteActions[3];
    // Debug.Log("DiscreteAction_CGT_Front: "+DiscreteAction_CGT_Front);
    (Vector3 thrustDirection_CGT_Front, float thrustForce_CGT_Front) = GetThrustDirection_CGT_Front(DiscreteAction_CGT_Front); // Map the thrust direction index to the corresponding Force_Direction
    rocketController.Cold_Gas_Thrusters(thrustDirection_CGT_Front, thrustForce_CGT_Front); // Apply thrust based on the decoded actions

    // -------------------- [ DiscreteActions Right_CGT ] --------------------
    int DiscreteAction_CGT_Back = actionBuffers.DiscreteActions[4];
    // Debug.Log("DiscreteAction_CGT_Back: "+DiscreteAction_CGT_Back);
    (Vector3 thrustDirection_CGT_Back, float thrustForce_CGT_Back) = GetThrustDirection_CGT_Back(DiscreteAction_CGT_Back);  // Map the thrust direction index to the corresponding Force_Direction
    rocketController.Cold_Gas_Thrusters(thrustDirection_CGT_Back, thrustForce_CGT_Back); // Apply thrust based on the decoded actions
    // ##############################################################################################################################
```

Figure 3.31: OnActionReceived Method

Let's break it down:

1. **Input Handling:** It receives action information (actionBuffers) from the environment.

2. **Thrust Vector Control (TVC):** It extracts discrete actions for TVC and applies corresponding thrust forces based on the actions.

3. **Cold Gas Thrusters (CGT):** It extracts discrete actions for CGT in different directions and applies thrust forces accordingly.

4. **Reward Function:** It calculates rewards based on various factors such as **distance to the landing pad**, **rotation angles**, **altitude**, **velocity**, and **leg landing status** (ref:3.4).

5. **End Episode Conditions:** It determines when to end the episode based on failure conditions (e.g., crashing or exceeding rotation limits) or success conditions (e.g., all legs landed and stable for a certain duration).

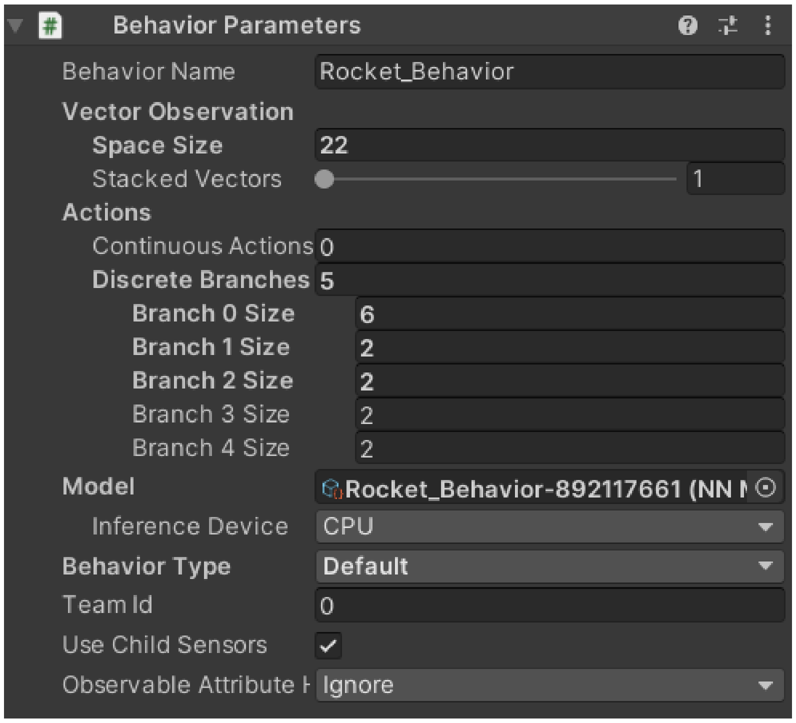
6. **Normalization and Weighting:** It normalizes angles and distances and applies weights to different components of the reward (ref: 3.4).

7. **Final Reward Calculation:** It combines individual rewards using predefined weights to compute the final reward for the agent (ref: 3.4).

8. **Reward Assignment:** It sets the calculated reward for the agent (ref: 3.4).

3.3.3 Behavior Parameters:

Behavior Parameters play a crucial role in configuring how agents behave within the Unity ML-Agents environment. This component acts as a container for various settings that influence the decision-making processes, actions, and overall functionality of the agent during training and simulation runs.



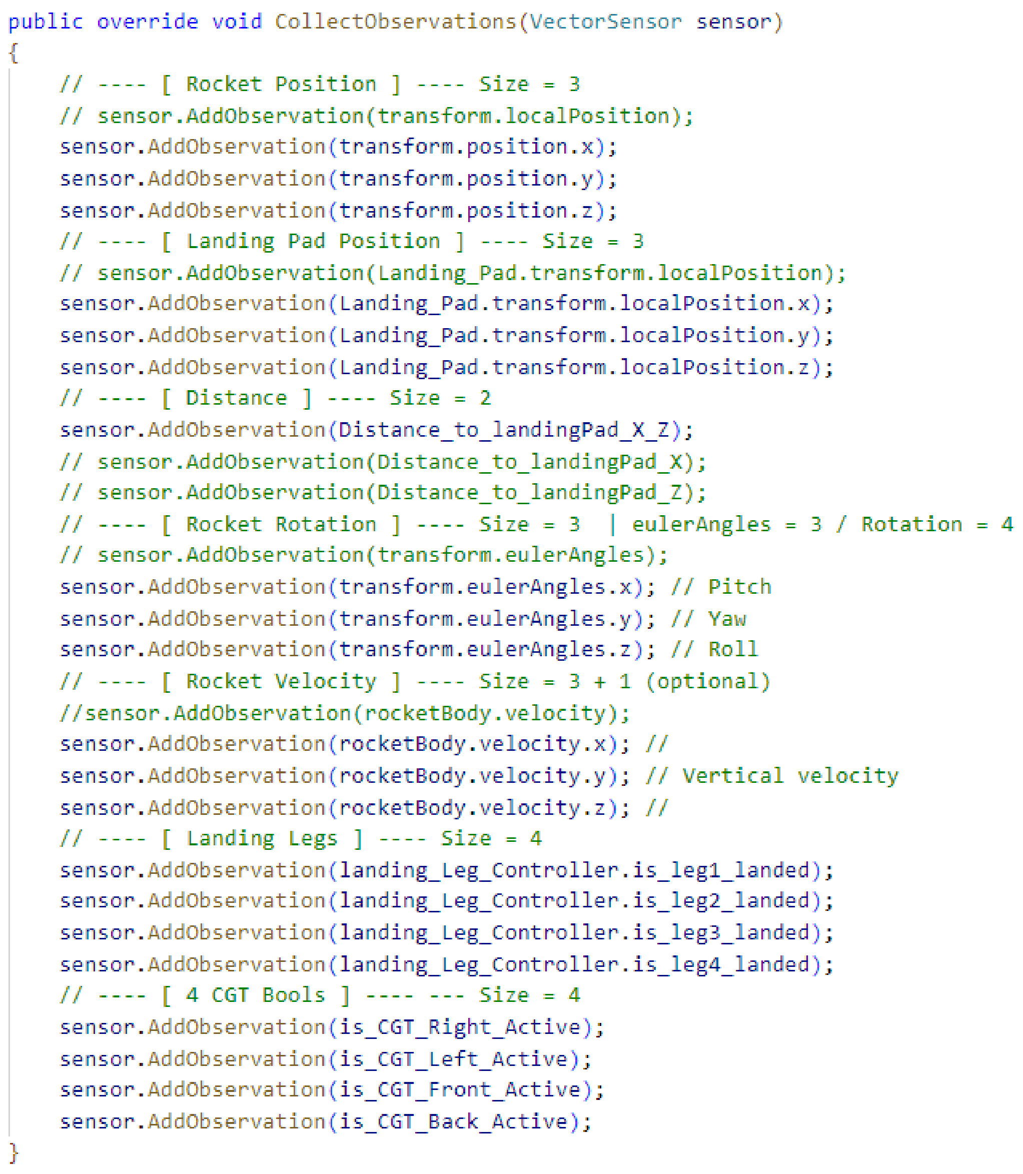
Behavior Parameters	
Behavior Name	Rocket_Behavior
Vector Observation	
Space Size	22
Stacked Vectors	1
Actions	
Continuous Actions	0
Discrete Branches	5
Branch 0 Size	6
Branch 1 Size	2
Branch 2 Size	2
Branch 3 Size	2
Branch 4 Size	2
Model	Rocket_Behavior-892117661 (NN I
Inference Device	CPU
Behavior Type	Default
Team Id	0
Use Child Sensors	✓
Observable Attribute I	Ignore

Figure 3.32: Behavior Parameters

Key Parameters:

- **Behavior Name:** This field assigns a unique name to the behavior, allowing us to distinguish between multiple behaviors an agent might possess.

- **Space Size:** This defines the dimensionality of the vector representing the agent's observation space. The agent receives observations from the environment in this format, and the size should correspond to the number of elements used to represent the state of the world. In Figure 3.33, the value is set to "22," indicating a 22-dimensional vector for observations.



```
public override void CollectObservations(VectorSensor sensor)
{
    // ---- [ Rocket Position ] ---- Size = 3
    // sensor.AddObservation(transform.localPosition);
    sensor.AddObservation(transform.position.x);
    sensor.AddObservation(transform.position.y);
    sensor.AddObservation(transform.position.z);
    // ---- [ Landing Pad Position ] ---- Size = 3
    // sensor.AddObservation(Landing_Pad.transform.localPosition);
    sensor.AddObservation(Landing_Pad.transform.localPosition.x);
    sensor.AddObservation(Landing_Pad.transform.localPosition.y);
    sensor.AddObservation(Landing_Pad.transform.localPosition.z);
    // ---- [ Distance ] ---- Size = 2
    sensor.AddObservation(Distance_to_landingPad_X_Z);
    // sensor.AddObservation(Distance_to_landingPad_X);
    // sensor.AddObservation(Distance_to_landingPad_Z);
    // ---- [ Rocket Rotation ] ---- Size = 3  | eulerAngles = 3 / Rotation = 4
    // sensor.AddObservation(transform.eulerAngles);
    sensor.AddObservation(transform.eulerAngles.x); // Pitch
    sensor.AddObservation(transform.eulerAngles.y); // Yaw
    sensor.AddObservation(transform.eulerAngles.z); // Roll
    // ---- [ Rocket Velocity ] ---- Size = 3 + 1 (optional)
    //sensor.AddObservation(rocketBody.velocity);
    sensor.AddObservation(rocketBody.velocity.x); //
    sensor.AddObservation(rocketBody.velocity.y); // Vertical velocity
    sensor.AddObservation(rocketBody.velocity.z); //
    // ---- [ Landing Legs ] ---- Size = 4
    sensor.AddObservation(landing_Leg_Controller.is_leg1_landed);
    sensor.AddObservation(landing_Leg_Controller.is_leg2_landed);
    sensor.AddObservation(landing_Leg_Controller.is_leg3_landed);
    sensor.AddObservation(landing_Leg_Controller.is_leg4_landed);
    // ---- [ 4 CGT Bools ] ---- --- Size = 4
    sensor.AddObservation(is_CGT_Right_Active);
    sensor.AddObservation(is_CGT_Left_Active);
    sensor.AddObservation(is_CGT_Front_Active);
    sensor.AddObservation(is_CGT_Back_Active);
}
```

Figure 3.33: Collecting Observations

The behavior parameters for the rocket agent's observations are detailed in Table 3.1. This table comprehensively outlines each observation type, including its nature and specific details.

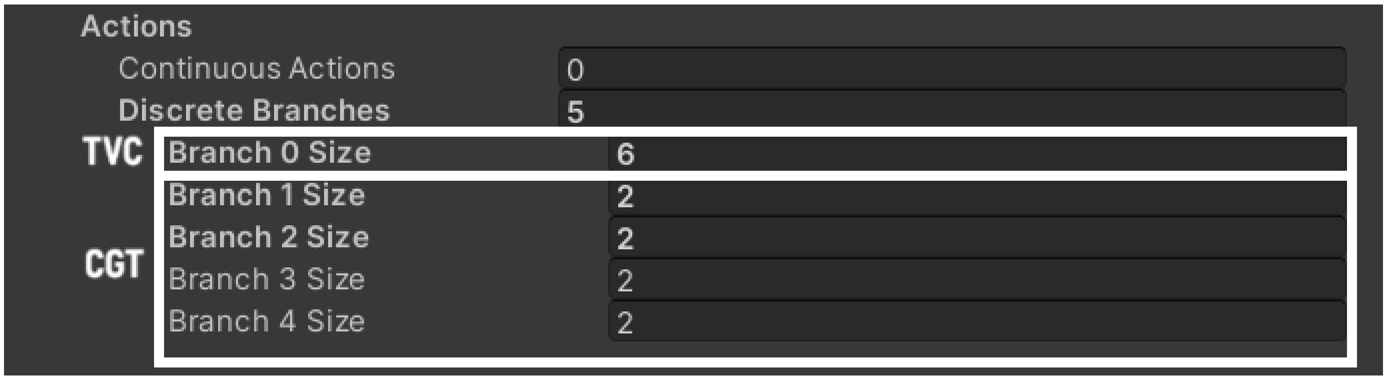
Table 3.1: Detailed Behavior Parameters for Rocket Agent Observations

Observation	**Type**	**Description**	**Size**
Rocket Position	Vector	(X, Y, Z) Axis	3
Rocket Rotation	Vector	(Pitch, Yaw, Roll) Angles	3
Rocket Velocity	Vector	(X, Y, Z) Velocity Components	3
Landing Pad Position	Vector	(X, Y, Z) Axis	3
Distance	Vector	Distance to Landing Pad in (X, Z)	2
Landing Legs	Boolean	Leg Landing States	4
Cold Gas Thrusters Activation	Boolean	Thruster Activation States	4

- **Actions:** This section specifies the action space that the agent can utilize. It consists of two dropdown menus:

  1. **Type:** This dropdown defines the format of the actions the agent can take. Options include Discrete (meaning a finite set of choices) and Continuous (where actions are represented by floating-point values).
  2. **Size:** This value specifies the number of elements within the chosen action format. The figure 3.32 shows separate values for different branches (explained below 3.3.3).

- **Branches:** This section is used for configuring a branching structure within the action space. It allows us to define multiple discrete or continuous action sub-spaces (branches) that the agent can choose from at a given point. The provided figure 3.34 shows five branches, each with its own size specified. When a branch is active, the agent can only output actions within the corresponding size constraints of that branch.



Actions	
Continuous Actions	0
Discrete Branches	5
Branch 0 Size	6
Branch 1 Size	2
Branch 2 Size	2
Branch 3 Size	2
Branch 4 Size	2

Figure 3.34: Visualization of Different Branch Sizes

- **Branch 0 (TVC)** has 6 dimensions, allowing the agent to choose from 6 different control actions for the Thrust Vector Control system.
- **Branches 1 through 4**, representing the Cold Gas Thrusters (CGT), each have a dimension of 2. This means the agent can choose independent on/off states for each individual Cold Gas Thruster.

$$\text{Total Combinations} = (\text{Number of choices in Branch 0}) \times (\text{Number of choices per CGT})^{\text{Number of thrusters}} \tag{3.1}$$

$$\text{Total Combinations} = 6 \times 2^4 = 6 \times 16 = 96$$

So, the agent has **96** possible action combinations at each step.

- **Model:** Holds a reference to the machine learning model (e.g., a neural network) associated with the behavior. This model maps observations from the environment to agent actions.
- **Inference Device:** Specifies the hardware device used for executing the machine learning model's computations (e.g., CPU, GPU).

- **Behavior Type:** Sets the overall mode in which the agent behaves:

  1. **Default:** During training, the agent learns a policy (model) that maps observations to actions. In the "Default" mode, the agent interacts with the environment, receives observations, and then uses inference on its trained policy to determine the appropriate action for the current state.
  2. **Heuristic:** Overrides the learned policy with custom logic for debugging or specific scenarios.
  3. **Inference:** Agent receives actions directly from a pre-trained model, The model then generates actions based on the observations without the back-and-forth interaction with the environment.

3.3.4 Decision Requester:

The DecisionRequester component is used to automatically request decisions at regular intervals for an agent in Unity's ML-Agents framework. By attaching this component to an agent, we can control the frequency of decision-making without manually requesting decisions in the agent's code.

- **Key Functionalities:**

  1. **RequestDecision(state):** This receives the current state information from the Rocket-Agent and transmits it to the simulation environment.
  2. **ReceiveAction(action):** This retrieves the action (control decision) chosen by the simulation environment (or pre-defined control logic) and sends it back to the Rocket-Agent.
  3. **Decision Interval:** The primary configuration parameter is the decision interval, which determines how often the agent will request a decision. For example, setting the interval to 5 means the agent will request a decision every 5 steps.

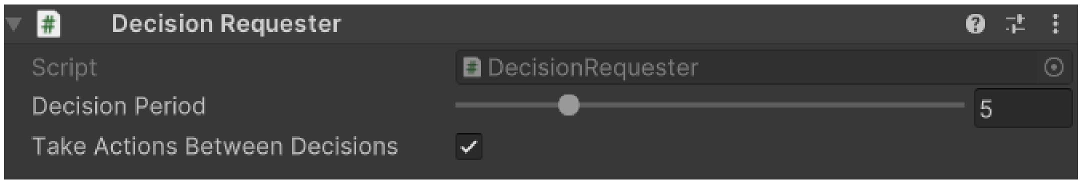

Decision Requester
Script DecisionRequester
Decision Period 5
Take Actions Between Decisions ✓

Figure 3.35: Decision Requester

3.3.5 Rocket Trajectory Recorder:

Understanding the agent's flight path and decision-making process during training and simulation runs is crucial for evaluating its performance. This section explores the interplay between the RocketTrajectoryRecorder script and data visualization techniques to achieve this goal.

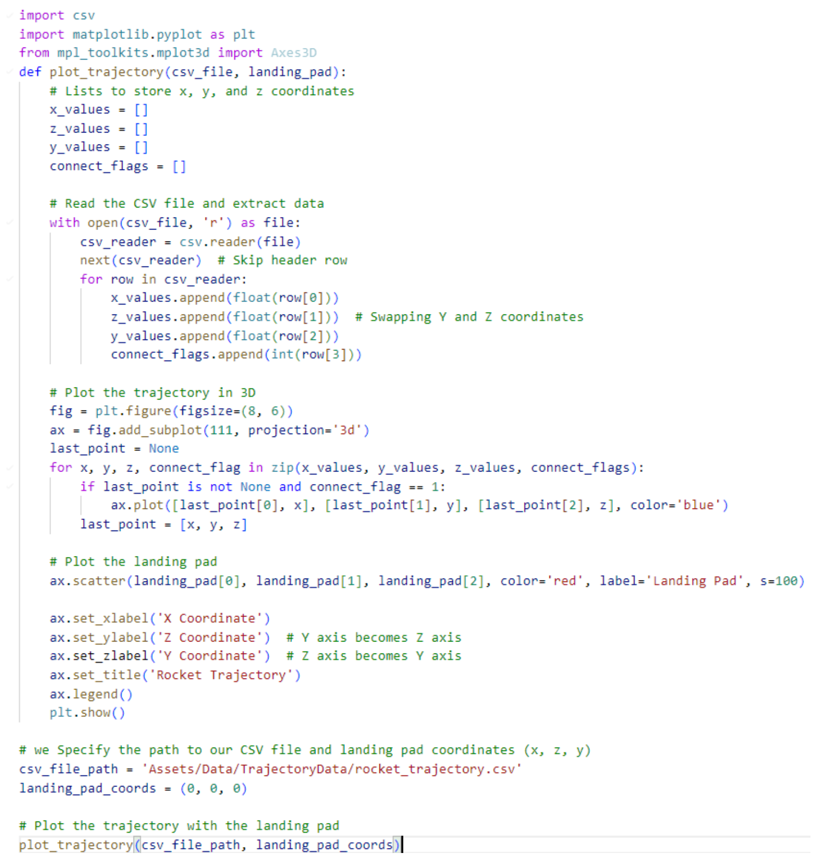
1. **Trajectory Recording:** The RocketTrajectoryRecorder script acts as a monitoring tool, capturing the agent's trajectory data at predefined intervals. This data typically includes the rocket's position (X, Y, and Z coordinates) at each recorded instance. Additionally, it incorporate flags indicating distinct flight phases. Essentially, the script builds a time-series record of the agent's movements throughout the simulation run.

Algorithme 1 : Rocket Trajectory Recording Algorithm

```
Initialize an empty list trajectoryPoints
Initialize an empty list connectFlags
Set timer ← 0
Define recordInterval as the time interval for recording positions

while simulation is running do
  Increment timer by $\Delta t$
  if timer ≥ recordInterval then
    timer ← 0
    RecordRocketPosition()
  end if
end while

Procedure RecordRocketPosition()
if trajectoryPoints is not empty then

  if IsNewPhase(currentPos, previousPos) then
          Add 0 to connectFlags

  end if

end if
   Add current rocket position to trajectoryPoints
   Add 1 to connectFlags
End Procedure

Function IsNewPhase(currentPos, previousPos)
return (currentPos.y - previousPos.y > 50.0)
End Function

Procedure SaveTrajectoryData(filePath)
for each point in trajectoryPoints do
  Write (x, y, z, connectFlags[i]) to CSV file
end for
End Procedure
```

2. **Data Export and Visualization:** The recorded trajectory data is stored in a format suitable for further analysis, such as a CSV (Comma-Separated Values) file. This file format allows for easy import into various data analysis and visualization tools. Tools like Python's matplotlib library, as demonstrated in the provided code snippet, can be leveraged to create

informative visualizations of the flight path.



```python
import csv
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
def plot_trajectory(csv_file, landing_pad):
    # Lists to store x, y, and z coordinates
    x_values = []
    z_values = []
    y_values = []
    connect_flags = []

    # Read the CSV file and extract data
    with open(csv_file, 'r') as file:
        csv_reader = csv.reader(file)
        next(csv_reader)  # Skip header row
        for row in csv_reader:
            x_values.append(float(row[0]))
            z_values.append(float(row[1]))  # Swapping Y and Z coordinates
            y_values.append(float(row[2]))
            connect_flags.append(int(row[3]))

    # Plot the trajectory in 3D
    fig = plt.figure(figsize=(8, 6))
    ax = fig.add_subplot(111, projection='3d')
    last_point = None
    for x, y, z, connect_flag in zip(x_values, y_values, z_values, connect_flags):
        if last_point is not None and connect_flag == 1:
            ax.plot([last_point[0], x], [last_point[1], y], [last_point[2], z], color='blue')
        last_point = [x, y, z]

    # Plot the landing pad
    ax.scatter(landing_pad[0], landing_pad[1], landing_pad[2], color='red', label='Landing Pad', s=100)

    ax.set_xlabel('X Coordinate')
    ax.set_ylabel('Z Coordinate')  # Y axis becomes Z axis
    ax.set_zlabel('Y Coordinate')  # Z axis becomes Y axis
    ax.set_title('Rocket Trajectory')
    ax.legend()
    plt.show()

# we Specify the path to our CSV file and landing pad coordinates (x, z, y)
csv_file_path = 'Assets/Data/TrajectoryData/rocket_trajectory.csv'
landing_pad_coords = (0, 0, 0)

# Plot the trajectory with the landing pad
plot_trajectory(csv_file_path, landing_pad_coords)
```

Figure 3.36: Trajectory Visualization Script

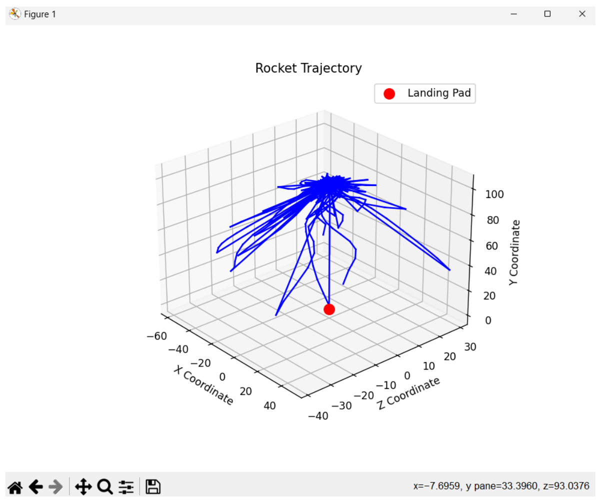
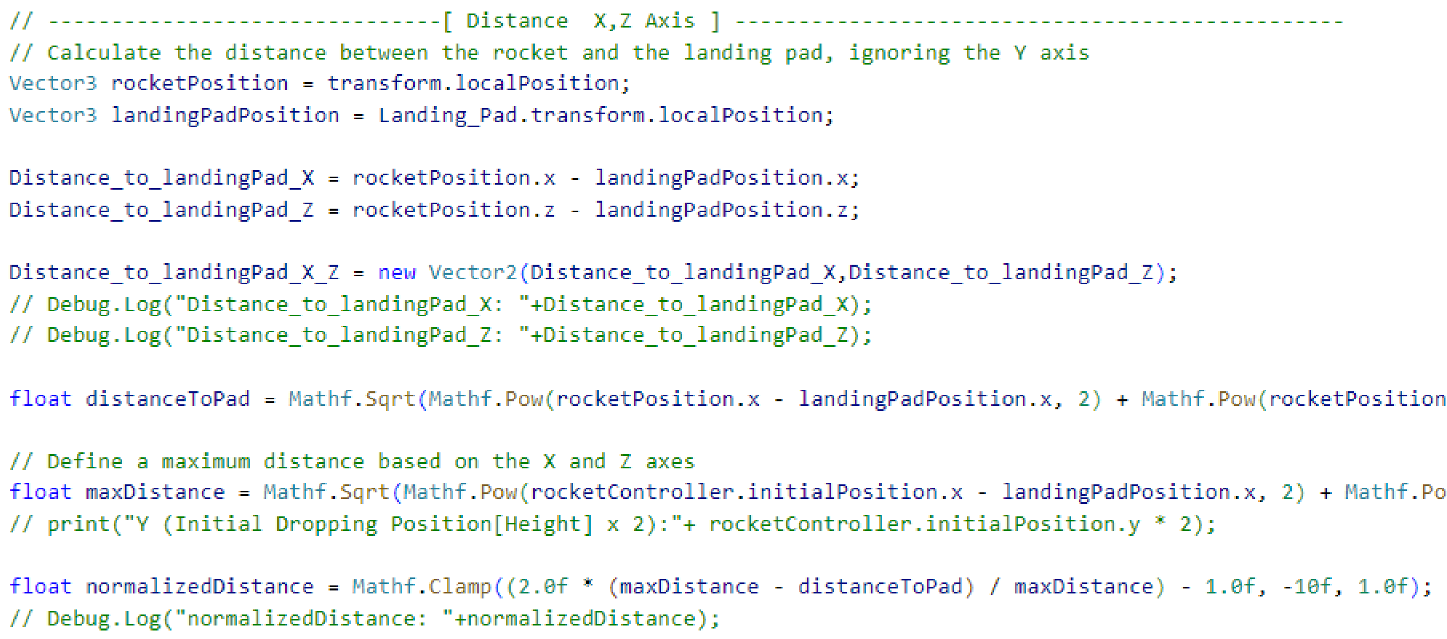


Figure 3.37: Rocket Trajectory Plot

3.4 Reward Shaping

3.4.1 Distance

The distance reward shaping mechanism plays a pivotal role in guiding the rocket towards the landing pad by incentivizing it to minimize the distance to the target location. Here's a detailed breakdown of the distance reward shaping process:



```
// -------------------------------[ Distance  X,Z Axis ] ------------------------------------------------
// Calculate the distance between the rocket and the landing pad, ignoring the Y axis
Vector3 rocketPosition = transform.localPosition;
Vector3 landingPadPosition = Landing_Pad.transform.localPosition;

Distance_to_landingPad_X = rocketPosition.x - landingPadPosition.x;
Distance_to_landingPad_Z = rocketPosition.z - landingPadPosition.z;

Distance_to_landingPad_X_Z = new Vector2(Distance_to_landingPad_X,Distance_to_landingPad_Z);
// Debug.Log("Distance_to_landingPad_X: "+Distance_to_landingPad_X);
// Debug.Log("Distance_to_landingPad_Z: "+Distance_to_landingPad_Z);

float distanceToPad = Mathf.Sqrt(Mathf.Pow(rocketPosition.x - landingPadPosition.x, 2) + Mathf.Pow(rocketPosition

// Define a maximum distance based on the X and Z axes
float maxDistance = Mathf.Sqrt(Mathf.Pow(rocketController.initialPosition.x - landingPadPosition.x, 2) + Mathf.Po
// print("Y (Initial Dropping Position[Height] x 2):"+ rocketController.initialPosition.y * 2);

float normalizedDistance = Mathf.Clamp((2.0f * (maxDistance - distanceToPad) / maxDistance) - 1.0f, -10f, 1.0f);
// Debug.Log("normalizedDistance: "+normalizedDistance);
```

Figure 3.38: Distance Reward Section

The distance between the rocket and the landing pad is computed in the X and Z axes while ignoring the Y axis. This calculation enables the agent to focus solely on horizontal positioning relative to the target.

$$\begin{aligned} \text{rocketPosition} &= \text{transform.localPosition} \\ \text{landingPadPosition} &= \text{Landing\_Pad.transform.localPosition} \\ \text{Distance\_to\_landingPad\_X} &= \text{rocketPosition.x} - \text{landingPadPosition.x} \\ \text{Distance\_to\_landingPad\_Z} &= \text{rocketPosition.z} - \text{landingPadPosition.z} \end{aligned}$$

The Euclidean distance between the rocket and the landing pad is then calculated using the Pythagorean theorem:

$$\text{distanceToPad} = \sqrt{(\text{rocketPosition.x} - \text{landingPadPosition.x})^2 + (\text{rocketPosition.z} - \text{landingPadPosition.z})^2} \tag{3.2}$$

The reward for distance is determined based on how close the rocket is to the landing pad relative to a predefined maximum distance. This distance reward is normalized to ensure it falls within a specified range, typically between -1 and 1, but in our case, we extend the range to [-10, 1] to heavily penalize any large deviation. The formula used for normalizing the distance reward is as follows:

$$\text{normalizedDistance} = \text{Clamp}\left(\frac{2 \times (\text{maxDistance} - \text{distanceToPad})}{\text{maxDistance}} - 1.0, -10, 1.0\right) \tag{3.3}$$

The resulting normalized distance reward guides the rocket's behavior towards the landing pad, with penalties for being far away and rewards for getting closer.

3.4.2 Rotation

The rotation component is crucial in ensuring the rocket maintains a stable orientation during its descent and landing. This component considers the **pitch**, **yaw**, and **roll** angles of the rocket and normalizes these angles to calculate a rotation reward. The detailed steps are as follows:

- **Extracting Rotation Angles:** The rocket's current rotation angles are extracted from its transform component. These angles represent the pitch, yaw, and roll of the rocket.

$$\text{pitchAngle} = \text{transform.eulerAngles.x} \tag{3.4}$$

$$\text{yawAngle} = \text{transform.eulerAngles.y} \tag{3.5}$$

$$\text{rollAngle} = \text{transform.eulerAngles.z} \tag{3.6}$$

- **Defining Maximum Allowed Angles:** The maximum permissible angles for pitch, yaw, and roll are defined to prevent excessive rotation, which could destabilize the rocket.

$$\text{maxPitch} = 45.0^\circ \tag{3.7}$$

$$\text{maxYaw} = 45.0^\circ \tag{3.8}$$

$$\text{maxRoll} = 45.0^\circ \tag{3.9}$$

- **Normalizing Angles:** The rotation angles are normalized to the range $[-180^\circ, 180^\circ)$ to handle the wrap-around effect that occurs at 360°.
- **Calculating Normalized Rotation:** Each angle is normalized based on its maximum allowed value, yielding a value between 0 and 1. This normalization ensures that smaller deviations from the ideal orientation are rewarded higher than larger deviations.

$$\text{normalizedPitch} = \text{clamp}\left(1.0 - \left|\frac{\text{pitchAngle}}{\text{maxPitch}}\right|, 0.0, 1.0\right) \tag{3.10}$$

$$\text{normalizedYaw} = \text{clamp}\left(1.0 - \left|\frac{\text{yawAngle}}{\text{maxYaw}}\right|, 0.0, 1.0\right) \tag{3.11}$$

$$\text{normalizedRoll} = \text{clamp}\left(1.0 - \left|\frac{\text{rollAngle}}{\text{maxRoll}}\right|, 0.0, 1.0\right) \tag{3.12}$$

- **Weighting Rotation Components:** Each normalized rotation angle is assigned a weight to balance their contributions to the final rotation reward. Here, pitch and roll are given higher weights compared to yaw.

$$\text{pitchWeight} = 0.45 \quad \text{(45\% of the total value)} \tag{3.13}$$

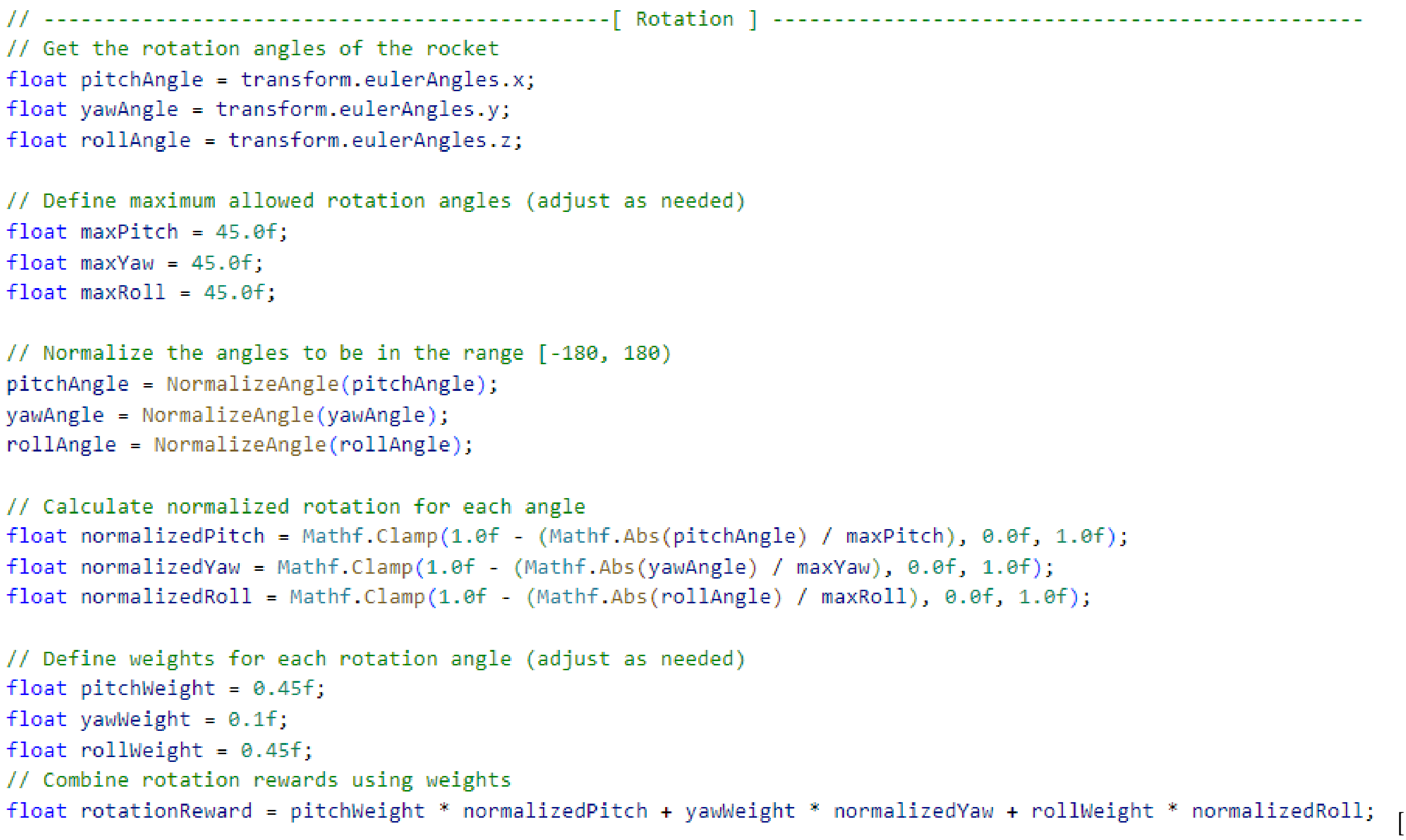
$$\text{yawWeight} = 0.1 \quad \text{(10\% of the total value)} \tag{3.14}$$

$$\text{rollWeight} = 0.45 \quad \text{(45\% of the total value)} \tag{3.15}$$

- **Combining Rotation Rewards:** The final rotation reward is a weighted sum of the normalized rotation components.

$$\begin{aligned} \text{rotationReward} = {} & (\text{pitchWeight} \cdot \text{normalizedPitch}) \\ & + (\text{yawWeight} \cdot \text{normalizedYaw}) \\ & + (\text{rollWeight} \cdot \text{normalizedRoll}) \end{aligned} \tag{3.16}$$

This detailed consideration of the rocket's rotation helps ensure that the rocket maintains stability and orientation during the descent and landing phases, contributing significantly to the overall reward shaping mechanism.

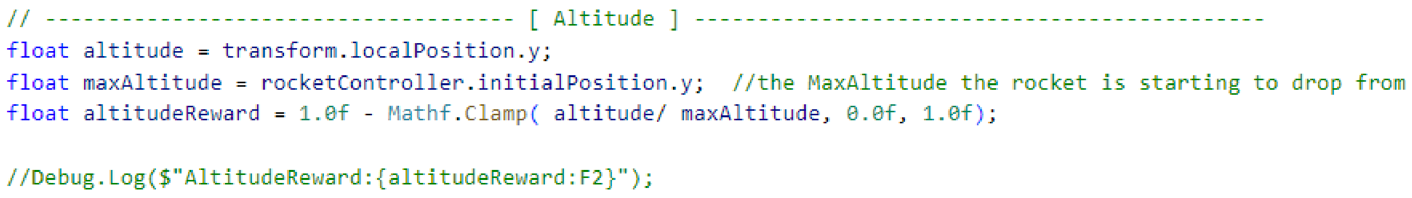


```
// ---------------------------------------------[ Rotation ] ----------------------------------------------
// Get the rotation angles of the rocket
float pitchAngle = transform.eulerAngles.x;
float yawAngle = transform.eulerAngles.y;
float rollAngle = transform.eulerAngles.z;

// Define maximum allowed rotation angles (adjust as needed)
float maxPitch = 45.0f;
float maxYaw = 45.0f;
float maxRoll = 45.0f;

// Normalize the angles to be in the range [-180, 180)
pitchAngle = NormalizeAngle(pitchAngle);
yawAngle = NormalizeAngle(yawAngle);
rollAngle = NormalizeAngle(rollAngle);

// Calculate normalized rotation for each angle
float normalizedPitch = Mathf.Clamp(1.0f - (Mathf.Abs(pitchAngle) / maxPitch), 0.0f, 1.0f);
float normalizedYaw = Mathf.Clamp(1.0f - (Mathf.Abs(yawAngle) / maxYaw), 0.0f, 1.0f);
float normalizedRoll = Mathf.Clamp(1.0f - (Mathf.Abs(rollAngle) / maxRoll), 0.0f, 1.0f);

// Define weights for each rotation angle (adjust as needed)
float pitchWeight = 0.45f;
float yawWeight = 0.1f;
float rollWeight = 0.45f;
// Combine rotation rewards using weights
float rotationReward = pitchWeight * normalizedPitch + yawWeight * normalizedYaw + rollWeight * normalizedRoll;  [
```

Figure 3.39: Rotation Reward Section

3.4.3 Altitude

The altitude of the rocket is measured relative to its initial drop height. The reward mechanism is designed to encourage the rocket to minimize its altitude in a controlled manner.



```
// ------------------------------------ [ Altitude ] --------------------------------------------
float altitude = transform.localPosition.y;
float maxAltitude = rocketController.initialPosition.y;  //the MaxAltitude the rocket is starting to drop from
float altitudeReward = 1.0f - Mathf.Clamp( altitude/ maxAltitude, 0.0f, 1.0f);

//Debug.Log($"AltitudeReward:{altitudeReward:F2}");
```

Figure 3.40: Altitude Reward Section

The altitude is determined by the y-coordinate of the rocket's local position:

$$\text{altitude} = \text{transform.localPosition.y} \tag{3.17}$$

The maximum altitude is defined as the initial height from which the rocket begins its descent:

$$\text{maxAltitude} = \text{rocketController.initialPosition.y} \tag{3.18}$$

The reward for altitude is computed by normalizing the current altitude relative to the maximum altitude. This normalization ensures that the altitude reward ranges from 0 to 1, where 1 represents the lowest possible altitude (i.e., at the landing pad level), and 0 represents the maximum initial altitude. The formula used to calculate the altitude reward is as follows:

$$\text{altitudeReward} = 1.0 - \text{Clamp}\left(\frac{\text{altitude}}{\text{maxAltitude}}, 0.0, 1.0\right) \tag{3.19}$$

In this context, the Clamp function ensures that the ratio $\frac{\text{altitude}}{\text{maxAltitude}}$ remains within the interval $[0.0, 1.0]$, preventing values outside this range that might result from measurement noise or other factors.

The resulting altitude reward is then combined with other rewards (such as distance, rotation, and velocity) to form the overall reward signal guiding the rocket's descent.

3.4.4 Velocity

The reward for velocity is determined by comparing the rocket's vertical descent speed to a target descent speed. This comparison is used to calculate a reward that encourages the rocket to maintain an optimal descent velocity. The reward is normalized to ensure it falls within a specified range, typically between -1 and 1, but in our case, we extend the range to [-10, 1] to heavily penalize any large deviation. The formula used for normalizing the velocity reward is as follows:

$$\text{velocityDifference} = \left|\text{verticalDescentSpeed} - \text{targetDescentSpeed}\right| \tag{3.20}$$

$$\text{normalizedDifference} = \text{Clamp}\left(1.0 - \frac{\text{velocityDifference}}{\text{maxDifference}}, -10.0, 1.0\right) \tag{3.21}$$

$$\text{velocityReward} = \text{normalizedDifference} \tag{3.22}$$

Here's a detailed explanation:

1. **Vertical Descent Speed Calculation**: The vertical descent speed is obtained from the negative of the rocket's y-axis velocity, since positive y is upward. This is given by:

$$\text{verticalDescentSpeed} = -\text{rocketBody.velocity.y} \tag{3.23}$$

2. **Target and Maximum Descent Speed**: The target descent speed is set to a desired value, for example, 10.0 m/s. The maximum allowable descent speed is set to another value, for example, 15.0 m/s. These values can be adjusted based on specific requirements:

$$\text{targetDescentSpeed} = 10.0 m/s \quad \text{and} \quad \text{MaximumDescentSpeed} = 15.0 m/s$$

3. **Velocity Difference**: The absolute difference between the vertical descent speed and the target descent speed is calculated to measure how far the current speed deviates from the target:

$$\text{velocityDifference} = \left|\text{verticalDescentSpeed} - \text{targetDescentSpeed}\right| \tag{3.24}$$

4. **Normalization of the Difference**: The velocity difference is then normalized to fall within the range of [0, 1] based on a maximum difference, which can be adjusted as needed. This normalization ensures that as the rocket's descent speed gets closer to the target, the reward increases. The range is extended to [-10, 1] to penalize large deviations more heavily:

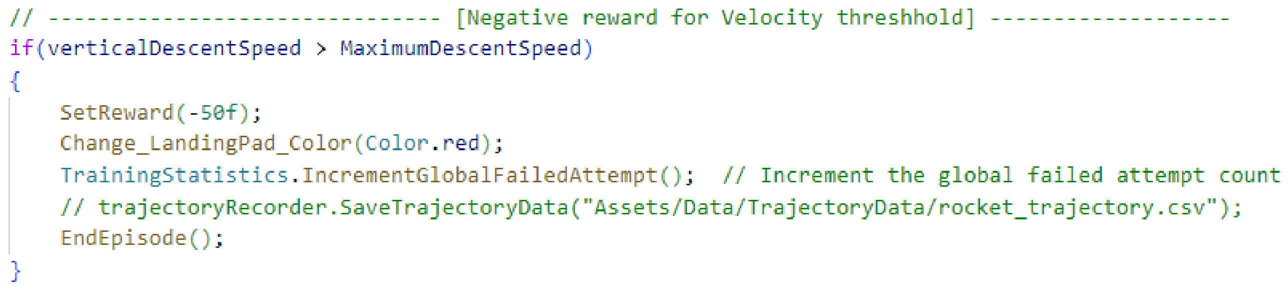
$$\text{normalizedDifference} = \text{Clamp}\left(1.0 - \frac{\text{velocityDifference}}{\text{maxDifference}}, -10.0, 1.0\right) \tag{3.25}$$

5. **Velocity Reward Calculation**: Finally, the normalized difference is used to calculate the

velocity reward. The reward increases as the velocity gets closer to the target:

$$\text{velocityReward} = \text{normalizedDifference} \tag{3.26}$$

This reward shaping encourages the rocket to maintain a descent speed close to the target, ensuring a controlled and safe landing.



```
// ------------------------------ [Negative reward for Velocity threshhold] -------------------
if(verticalDescentSpeed > MaximumDescentSpeed)
{
    SetReward(-50f);
    Change_LandingPad_Color(Color.red);
    TrainingStatistics.IncrementGlobalFailedAttempt();  // Increment the global failed attempt count
    // trajectoryRecorder.SaveTrajectoryData("Assets/Data/TrajectoryData/rocket_trajectory.csv");
    EndEpisode();
}
```

Figure 3.41: Velocity Termination Negative reward

In addition to rewarding proximity to the target descent speed, a significant penalty is imposed if the vertical descent speed exceeds the maximum allowable descent speed. If this threshold is breached, a substantial negative reward is issued, the landing pad color is changed to red, and the episode ends:

Mathematically, the condition for applying the negative reward is:

$$\text{if } v_y > v_{\max}, \quad \text{then apply a reward of } -50.0$$

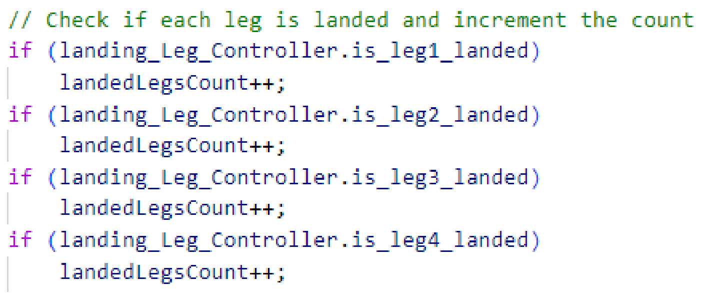
3.4.5 Landing Legs Status

The landing leg reward system is designed to incentivize the rocket to land stably by rewarding the number of landing legs that make contact with the ground. Each leg's landing status is checked, and an exponential reward is calculated based on the count of landed legs. Additionally, if all legs are landed and the rocket remains stable for a specified duration, a substantial reward is given, and the episode ends successfully.

First, the number of landed legs is determined:

$$\text{landedLegsCount} = 0 \tag{3.27}$$

The landing status of each leg is checked and the count is incremented accordingly:



```
// Check if each leg is landed and increment the count
if (landing_Leg_Controller.is_leg1_landed)
    landedLegsCount++;
if (landing_Leg_Controller.is_leg2_landed)
    landedLegsCount++;
if (landing_Leg_Controller.is_leg3_landed)
    landedLegsCount++;
if (landing_Leg_Controller.is_leg4_landed)
    landedLegsCount++;
```

Figure 3.42: Landing legs count

An exponential reward is calculated based on the number of landed legs:

$$\text{legLandingReward} = 2^{\text{landedLegsCount}} \quad \text{(i.e., 1, 4, 8, 16)} \tag{3.28}$$

If at least one leg is landed, this reward is applied:

```
// Increment the reward for successfully landed legs
if (landedLegsCount != 0) {
    SetReward(legLandingReward);
}
```

To ensure a successful landing, all legs must be landed and remain stable for a specified duration. This is checked using a timer:

```
// Check if all legs are landed
allLegsLanded = landing_Leg_Controller.is_leg1_landed &&
                landing_Leg_Controller.is_leg2_landed &&
                landing_Leg_Controller.is_leg3_landed &&
                landing_Leg_Controller.is_leg4_landed;
```

```
if (allLegsLanded) {
    // Increment the timer when all legs are landed
    landingTimer += Time.deltaTime;
    // Check if the landing duration threshold is reached (e.g., 15 seconds)
    if (landingTimer >= landingDurationThreshold) {
        // End the episode and provide a positive reward
        SetReward(successfulLandingReward[Current_Level]);
        Change_LandingPad_Color(Color.green);
        TrainingStatistics.IncrementGlobalSuccessfulAttempt();  // Increment the global
        EndEpisode();
    }
} else {
    // Reset the timer if not all legs are landed
    landingTimer = 0f;
}
```

In summary, the reward system for landing legs is as follows:

1. Calculate the number of landed legs.
2. Apply an exponential reward based on the number of landed legs.
3. If all legs are landed and the rocket remains stable for a threshold duration, provide a significant reward and end the episode successfully.

This system ensures that the rocket is rewarded for stable landings, promoting safe and controlled descent.

3.4.6 Final Reward

The overall reward for the rocket's performance is derived by combining four distinct components: **distance**, **rotation**, **altitude**, and **velocity** rewards. Each component contributes equally to the final reward, ensuring a balanced evaluation of the rocket's landing process.

First, each component is assigned an equal weight:

$$\text{distanceWeight\_reward} = 0.25 \tag{3.29}$$

$$\text{rotationWeight\_reward} = 0.25 \tag{3.30}$$

$$\text{altitudeWeight\_reward} = 0.25 \tag{3.31}$$

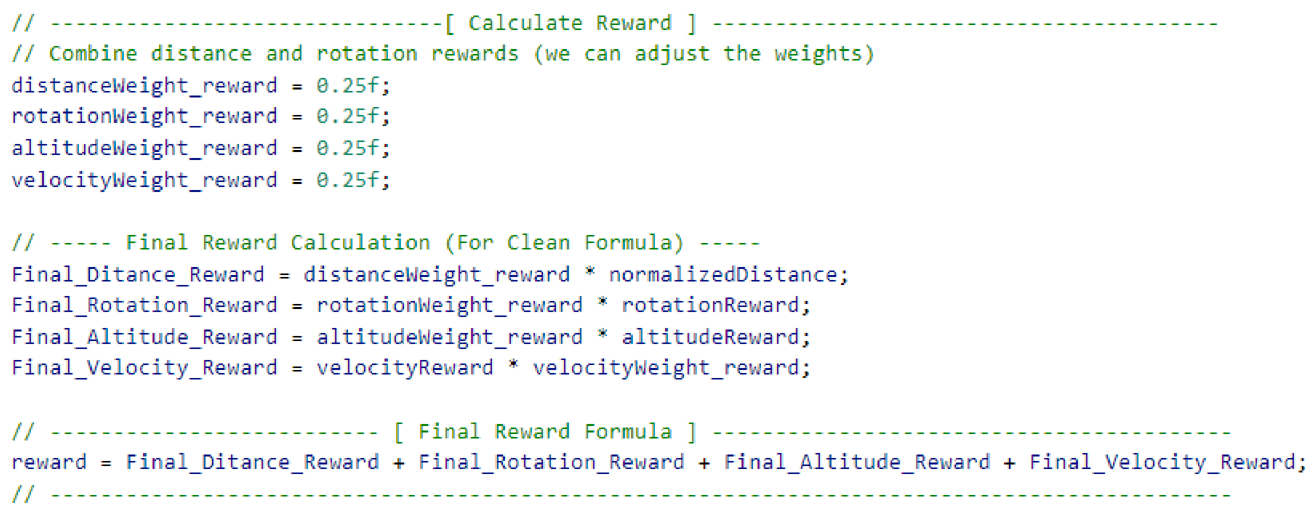
$$\text{velocityWeight\_reward} = 0.25 \tag{3.32}$$

The weighted rewards for each component are calculated as follows:

$$\begin{aligned}
\text{Final\_Distance\_Reward} &= \text{distanceWeight\_reward} \times \text{normalizedDistance} \\
\text{Final\_Rotation\_Reward} &= \text{rotationWeight\_reward} \times \text{rotationReward} \\
\text{Final\_Altitude\_Reward} &= \text{altitudeWeight\_reward} \times \text{altitudeReward} \\
\text{Final\_Velocity\_Reward} &= \text{velocityWeight\_reward} \times \text{velocityReward}
\end{aligned}$$

The overall reward is then obtained by summing these weighted rewards:

$$\textbf{reward} = \textit{Final\_Distance\_Reward} + \textit{Final\_Rotation\_Reward} + \textit{Final\_Altitude\_Reward} + \textit{Final\_Velocity\_Reward} \tag{3.33}$$



```
// ------------------------------[ Calculate Reward ] --------------------------------------
// Combine distance and rotation rewards (we can adjust the weights)
distanceWeight_reward = 0.25f;
rotationWeight_reward = 0.25f;
altitudeWeight_reward = 0.25f;
velocityWeight_reward = 0.25f;

// ----- Final Reward Calculation (For Clean Formula) -----
Final_Ditance_Reward = distanceWeight_reward * normalizedDistance;
Final_Rotation_Reward = rotationWeight_reward * rotationReward;
Final_Altitude_Reward = altitudeWeight_reward * altitudeReward;
Final_Velocity_Reward = velocityReward * velocityWeight_reward;

// ------------------------- [ Final Reward Formula ] ----------------------------------------
reward = Final_Ditance_Reward + Final_Rotation_Reward + Final_Altitude_Reward + Final_Velocity_Reward;
// ---------------------------------------------------------------------------------------------
```

Figure 3.43: Final Reward Section

This comprehensive reward structure ensures that the rocket is evaluated based on its proximity to the landing pad (**distance reward**), its orientation (**rotation reward**), its altitude (**altitude reward**), and its descent velocity (**velocity reward**). By weighting each component equally, the reward system promotes a balanced approach to optimizing all critical aspects of the rocket's landing process.

The implementation of this final reward occurs at every interval of steps set by the decision requester (3). This interval determines how frequently the agent's decisions and corresponding rewards are evaluated, ensuring timely and accurate feedback for the learning process.

3.5 Training Configuration and Process

In this section, we detail the configuration settings and the process involved in training the rocket's behavior using Proximal Policy Optimization (PPO). The parameters and settings used for training the `Rocket_Behavior` are summarized in Table 3.2.

Here's a breakdown of some key parameters and their justifications:

- **Batch Size (120):** The Batch Size parameter determines how many completed episodes are grouped together for updating the agent's policy network (the "brain" that guides its

Table 3.2: Training Parameters for Rocket_Behavior

Parameter	**Value**
`trainer_type`	PPO
`batch_size`	120
`buffer_size`	12000
`learning_rate`	0.0003
`beta`	0.001
`epsilon`	0.2
`lambd`	0.95
`num_epoch`	3
`learning_rate_schedule`	Linear
`normalize`	True
`hidden_units`	128
`num_layers`	2
`vis_encode_type`	Simple
`gamma`	0.99
`strength`	1.0
`keep_checkpoints`	-1
`checkpoint_interval`	50000000
`max_steps`	1000000000
`time_horizon`	1000
`summary_freq`	12000

decision-making). This is why 120 is a common choice for batch-based reinforcement learning algorithms, balancing computational efficiency with learning progress. *Typical range: 32 - 512*

- **Learning Rate (0.0003):** This value helps the agent learn at a steady pace without encountering instability or slow convergence. *Typical range:* $1e^{-5} - 1e^{-3}$
- **Beta (0.001):** Beta directly influences the clipping range used in PPO. A higher Beta value leads to a stricter clipping range, meaning the updated policy is closer to the previous one, promoting exploitation. Conversely, a lower Beta value allows for a looser clipping range, giving the agent more freedom to explore new actions. *Typical range:* $1e^{-4} - 1e^{-2}$
- **Epsilon (0.2):** This exploration parameter allows the agent to balance exploitation (choosing known good actions) with exploration (trying new actions) during training. *Typical range: 0.1 - 0.3*

- **Hidden Units (128) and Num Layers (2):** This configuration provides a sufficient neural network capacity for the agent to learn complex control policies for the rocket landing task.

- The remaining parameters in Table 3.2 control various aspects of the training process, such as buffer size, learning rate schedule, and checkpoint frequency. These were chosen to ensure efficient training and facilitate the evaluation of the trained agent's performance.

The training process involves setting up the agent's behavior and using the specified parameters to guide its learning. The `Rocket_Behavior` is configured to use the Proximal Policy Optimization (PPO) algorithm, which is a widely-used reinforcement learning algorithm known for its stability and reliability.

These settings and processes are crucial for ensuring the agent learns effectively and efficiently, optimizing the performance of the rocket landing behavior.

3.5.1 Training Command and CMD Output

To initiate the training process, we use the following command in the Command Prompt (CMD):

```
mlagents-learn Rocket.yaml --env=C:/Users/Hicham/Desktop/"Final Project"/Builds/Limited_
```

This command starts the ML-Agents training process using the configuration specified in `Rocket.yaml`. Below is a breakdown of each component of the command:

- `mlagents-learn`: The command to start the training process using ML-Agents.

- `Rocket.yaml`: The configuration file that contains the training parameters and settings.

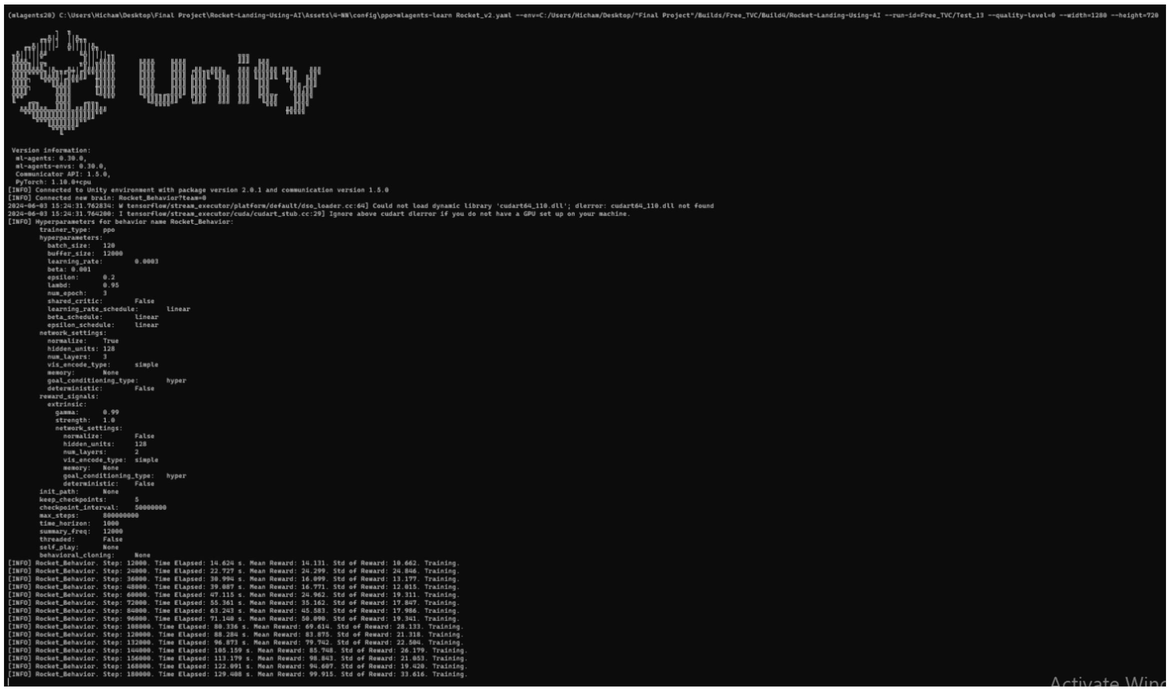
- `-env=C:/Users/Hicham/Desktop/"Final Project"/Builds/Limited_TVC/Build_1/Rocket-Land`
  Specifies the environment executable for the rocket landing simulation.

- `-run-id=Limited_TVC/Batch_1/Test_3_All_Lvls`: Assigns a unique identifier for this training run, which is used for logging and checkpointing.

- `-quality-level=0`: Sets the quality level of the environment, with 0 being the lowest quality, to optimize performance during training.
- `-width=1280 -height=720`: Sets the resolution of the environment window.

3.5.1.1 CMD Output During Training

When the training command is executed, the CMD window displays various information related to the training process. This includes the initialization of the training environment, loading of the configuration file, and real-time updates on the training progress such as Step, Time Elapsed, Mean Reward, and other relevant metrics.



```
(mlagents20) C:\Users\Hicham\Desktop\Final Project\Rocket-Landing-Using-AI\Assets\4-NN\config\ppo>mlagents-learn Rocket_v2.yaml --env=C:/Users/Hicham/Desktop/"Final Project"/Builds/Free_TVC/Build4/Rocket-Landing-Using-AI --run-id=Free_TVC/Test_13 --quality-level=0 --width=1280 --height=720

 Version information:
  ml-agents: 0.30.0,
  ml-agents-envs: 0.30.0,
  Communicator API: 1.5.0,
  PyTorch: 1.10.0+cpu
[INFO] Connected to Unity environment with package version 2.0.1 and communication version 1.5.0
[INFO] Connected new brain: Rocket_Behavior?team=0
2024-06-03 15:24:31.762834: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'cudart64_110.dll'; dlerror: cudart64_110.dll not found
2024-06-03 15:24:31.764200: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dlerror if you do not have a GPU set up on your machine.
[INFO] Hyperparameters for behavior name Rocket_Behavior:
        trainer_type:   ppo
        hyperparameters:
          batch_size:   120
          buffer_size:  12000
          learning_rate:        0.0003
          beta: 0.001
          epsilon:      0.2
          lambd:        0.95
          num_epoch:    3
          shared_critic:        False
          learning_rate_schedule:       linear
          beta_schedule:        linear
          epsilon_schedule:     linear
        network_settings:
          normalize:    True
          hidden_units: 128
          num_layers:   3
          vis_encode_type:      simple
          memory:       None
          goal_conditioning_type:       hyper
          deterministic:        False
        reward_signals:
          extrinsic:
            gamma:      0.99
            strength:   1.0
            network_settings:
              normalize:        False
              hidden_units:     128
              num_layers:       2
              vis_encode_type:  simple
              memory:   None
              goal_conditioning_type:   hyper
              deterministic:    False
        init_path:      None
        keep_checkpoints:       5
        checkpoint_interval:    50000000
        max_steps:      800000000
        time_horizon:   1000
        summary_freq:   12000
        threaded:       False
        self_play:      None
        behavioral_cloning:     None
[INFO] Rocket_Behavior. Step: 12000. Time Elapsed: 14.624 s. Mean Reward: 14.131. Std of Reward: 10.662. Training.
[INFO] Rocket_Behavior. Step: 24000. Time Elapsed: 22.727 s. Mean Reward: 24.299. Std of Reward: 24.846. Training.
[INFO] Rocket_Behavior. Step: 36000. Time Elapsed: 30.994 s. Mean Reward: 16.099. Std of Reward: 13.177. Training.
[INFO] Rocket_Behavior. Step: 48000. Time Elapsed: 39.087 s. Mean Reward: 16.771. Std of Reward: 12.015. Training.
[INFO] Rocket_Behavior. Step: 60000. Time Elapsed: 47.115 s. Mean Reward: 24.962. Std of Reward: 19.311. Training.
[INFO] Rocket_Behavior. Step: 72000. Time Elapsed: 55.361 s. Mean Reward: 35.162. Std of Reward: 17.847. Training.
[INFO] Rocket_Behavior. Step: 84000. Time Elapsed: 63.243 s. Mean Reward: 45.583. Std of Reward: 17.986. Training.
[INFO] Rocket_Behavior. Step: 96000. Time Elapsed: 71.140 s. Mean Reward: 50.090. Std of Reward: 19.341. Training.
[INFO] Rocket_Behavior. Step: 108000. Time Elapsed: 80.336 s. Mean Reward: 69.614. Std of Reward: 28.133. Training.
[INFO] Rocket_Behavior. Step: 120000. Time Elapsed: 88.284 s. Mean Reward: 83.875. Std of Reward: 21.318. Training.
[INFO] Rocket_Behavior. Step: 132000. Time Elapsed: 96.873 s. Mean Reward: 79.742. Std of Reward: 22.504. Training.
[INFO] Rocket_Behavior. Step: 144000. Time Elapsed: 105.159 s. Mean Reward: 85.748. Std of Reward: 26.179. Training.
[INFO] Rocket_Behavior. Step: 156000. Time Elapsed: 113.179 s. Mean Reward: 98.843. Std of Reward: 21.053. Training.
[INFO] Rocket_Behavior. Step: 168000. Time Elapsed: 122.091 s. Mean Reward: 94.607. Std of Reward: 19.420. Training.
[INFO] Rocket_Behavior. Step: 180000. Time Elapsed: 129.408 s. Mean Reward: 99.915. Std of Reward: 33.616. Training.
```

Figure 3.44: CMD Output During Training

Figure 3.44 shows a screenshot of the CMD output during the training process. Key information displayed includes:

- **Initialization Logs:** These logs confirm the environment has been successfully loaded and the training process has started.

- **Episode Updates:** These lines show real-time updates for each episode, including Step, Time Elapsed, Mean Reward, and other relevant metrics.
- **Checkpoint Logs:** Periodic logs indicating when checkpoints are saved, allowing the training to resume from the last checkpoint in case of interruptions.

The CMD output is crucial for monitoring the progress and performance of the training process, providing immediate feedback and insights into the agent's learning behavior.

Chapter 4

Results and Analysis

In this chapter, we delve into the outcomes and insights derived from the experimental phase of our AI-guided rocket landing project. The primary objectives of these experiments were to assess the performance and effectiveness of our reinforcement learning system, particularly in the context of autonomous rocket landings.

4.1 Hardware and Software Specifications

In this section, we outline the hardware and software used for the development and training of our rocket agent. The performance and capabilities of the hardware, along with the choice of software, play a crucial role in the efficiency of training and the quality of results obtained.

4.1.0.1 Hardware Specifications

The hardware setup used for training includes high-performance CPU and GPU units to handle the computationally intensive tasks associated with training machine learning models. The detailed specifications are listed in Table 4.1.

Table 4.1: Hardware Specifications

Component	**Specification**
Laptop	IdeaPad Gaming 3 15ACH6
CPU	AMD Ryzen 7 5800H with Radeon Graphics, 3201 Mhz, 8 Core(s), 16 Logical Processors
GPU	NVIDIA GeForce RTX 3060 Laptop GPU, 6 GHz VRAM
RAM	16.0 GB DDR4
Storage	1TB SSD
Operating System	Microsoft Windows 11 Pro

4.1.0.2 Software Specifications

The software environment used for the development and training of the rocket agent includes specific versions of Unity and ML-Agents. These versions are chosen to ensure compatibility and to leverage the latest features and improvements in the tools.

- **Unity Version:** Unity 2022.3.10f1
- **ML-Agents Version:** ML-Agents Release 19

4.1.0.3 Impact of Specifications on Training Time and Results

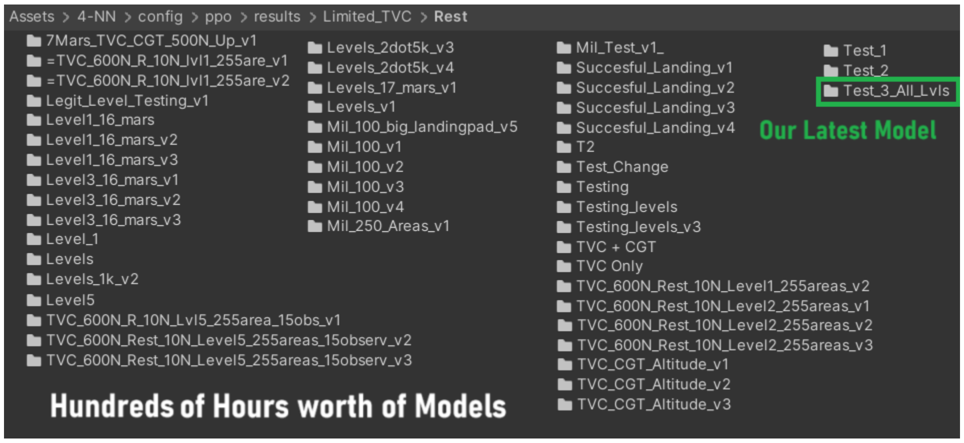
The specifications of the hardware and software used have a significant impact on both the training time and the results achieved. High-performance CPUs and GPUs can substantially reduce the time required for training by handling more calculations per second and processing larger batches of data. Sufficient RAM ensures that large datasets and complex simulations can be loaded into memory without causing slowdowns.

Moreover, the choice of Unity and ML-Agents versions ensures that we have access to the latest tools, features, and optimizations. This can lead to more efficient training processes and potentially better-performing models.

Overall, the combination of advanced hardware and up-to-date software is crucial for achieving optimal results in the training and deployment of our rocket agent.

4.1.1 Models Trained

Throughout the research, numerous models were trained, each stored in different folders, representing various training phases and configurations. The training process involved extensive experimentation, taking hundreds of hours to refine and optimize the models.

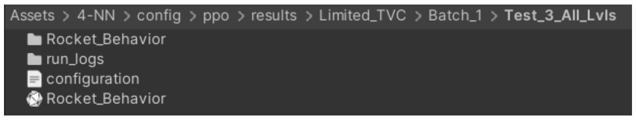


```
Assets > 4-NN > config > ppo > results > Limited_TVC > Rest
7Mars_TVC_CGT_500N_Up_v1
=TVC_600N_R_10N_lvl1_255are_v1
=TVC_600N_R_10N_lvl1_255are_v2
Legit_Level_Testing_v1
Level1_16_mars
Level1_16_mars_v2
Level1_16_mars_v3
Level3_16_mars_v1
Level3_16_mars_v2
Level3_16_mars_v3
Level_1
Levels
Levels_1k_v2
Level5
TVC_600N_R_10N_Lvl5_255area_15obs_v1
TVC_600N_Rest_10N_Level5_255areas_15observ_v2
TVC_600N_Rest_10N_Level5_255areas_15observ_v3
Levels_2dot5k_v3
Levels_2dot5k_v4
Levels_17_mars_v1
Levels_v1
Mil_100_big_landingpad_v5
Mil_100_v1
Mil_100_v2
Mil_100_v3
Mil_100_v4
Mil_250_Areas_v1
Mil_Test_v1_
Succesful_Landing_v1
Succesful_Landing_v2
Succesful_Landing_v3
Succesful_Landing_v4
T2
Test_Change
Testing
Testing_levels
Testing_levels_v3
TVC + CGT
TVC Only
TVC_600N_Rest_10N_Level1_255areas_v2
TVC_600N_Rest_10N_Level2_255areas_v1
TVC_600N_Rest_10N_Level2_255areas_v2
TVC_600N_Rest_10N_Level2_255areas_v3
TVC_CGT_Altitude_v1
TVC_CGT_Altitude_v2
TVC_CGT_Altitude_v3
Test_1
Test_2
Test_3_All_Lvls
```

Figure 4.1: Hundreds of hours of Models within folders

Below is a summary of each folder structure and contents:

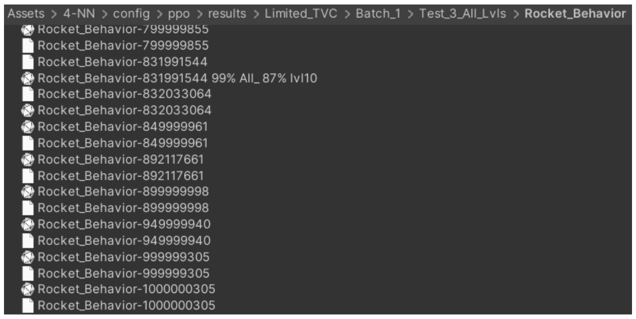
4.1.1.1 Folder structure and content



```
Assets > 4-NN > config > ppo > results > Limited_TVC > Batch_1 > Test_3_All_Lvls
Rocket_Behavior
run_logs
configuration
Rocket_Behavior
```

Figure 4.2: Model Folder Structure

- **Rocket_Behavior (NN model)** This .ONNX File is our neural network model saved in ONNX format. Each model represents a specific trained behavior of the rocket.
- **Rocket_Behavior (Folder):** This folder includes multiple neural network models saved at different training intervals. For instance, models are named according to the step count, such as "Rocket_Behavior-1000000000", indicating the model saved at one billion steps.



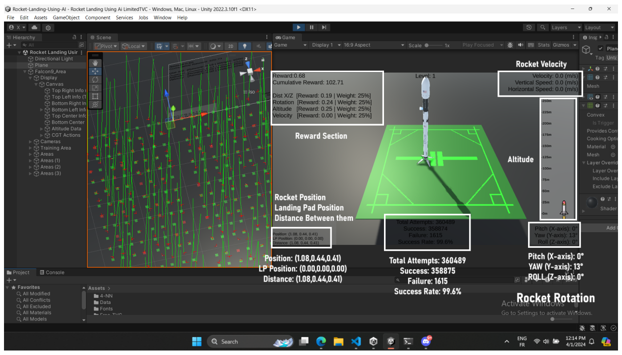
```
Assets > 4-NN > config > ppo > results > Limited_TVC > Batch_1 > Test_3_All_Lvls > Rocket_Behavior
Rocket_Behavior-799999855
Rocket_Behavior-799999855
Rocket_Behavior-831991544
Rocket_Behavior-831991544 99% All_ 87% lvl10
Rocket_Behavior-832033064
Rocket_Behavior-832033064
Rocket_Behavior-849999961
Rocket_Behavior-849999961
Rocket_Behavior-892117661
Rocket_Behavior-892117661
Rocket_Behavior-899999998
Rocket_Behavior-899999998
Rocket_Behavior-949999940
Rocket_Behavior-949999940
Rocket_Behavior-999999305
Rocket_Behavior-999999305
Rocket_Behavior-1000000305
Rocket_Behavior-1000000305
```

Figure 4.3: Rocket Behavior Folder Models

- **Additionally, this folder contains:**
  - **Run Logs**: Detailed logs of each training run, capturing important metrics and events.
  - **Configurations** (.yaml file): The configuration files used for each training run, detailing the parameters and settings.

By integrating TensorBoard with our training framework, we gained valuable insights into the agent's learning process. Visualizing the scalar metrics allowed us to track the agent's performance over time, identify potential issues like convergence problems, and make informed decisions about hyperparameter tuning.

4.2 Agent Performance Visualization in Unity



```
Reward:0.68
Cumulative Reward: 102.71

Dist X/Z  [Reward: 0.19 | Weight: 25%]
Rotation  [Reward: 0.24 | Weight: 25%]
Altitude  [Reward: 0.25 | Weight: 25%]
Velocity  [Reward: 0.00 | Weight: 25%]

Level: 1

Velocity: 0.0 (m/s)
Vertical Speed: 0.0 (m/s)
Horizontal Speed: 0.0 (m/s)

Position: (1.08,0.44,0.41)
LP Position: (0.00,0.00,0.00)
Distance: (1.08,0.44,0.41)

Total Attempts: 360489
Success: 358875
Failure: 1615
Success Rate: 99.6%

Pitch (X-axis): 0°
YAW (Y-axis): 13°
ROLL (Z-axis): 0°
```

Figure 4.4: Unity Simulation Scene + Dynamic Monitoring View

This section details the visualization components within the Unity engine used to monitor and evaluate the performance of the reinforcement learning agent during its training for autonomous rocket landings.

Simulation Scene with Raycast: The left window of Figure 4.4 showcases the simulation environment. This includes a visualization of the rocket and the landing pad. Here, a raycast is employed to depict the force direction applied by the agent's control decision (action) at each timestep. This visual aid helps understand the relationship between the chosen action and the resulting force acting on the rocket.

Camera View and Force Visualization: The right window provides a camera view attached to the rocket, allowing observation of its orientation and balance throughout the descent. Additionally, a particle system dynamically visualizes the thrust forces (both Cold Gas Thrusters (CGT) and Thrust Vector Control (TVC)) acting on the rocket. The particles are emitted in the opposite direction of the applied force, offering a clear representation of the forces influencing the rocket's motion.

Reward Structure and Visualization: The top right corner of the right window displays the current velocity of the rocket, including its vertical and horizontal components.

The top left corner displays the real-time reward information for various aspects of the rocket's state:

- Distance (X/Z Axis) Reward (25%): This reward encourages the agent to land close to the center of the landing pad. As the distance from the center increases, the reward decreases.
- Rotation Reward (25%): This reward incentivizes the agent to maintain a minimal roll, pitch, and yaw throughout descent. A value of zero for all rotations yields the maximum reward.
- Altitude Reward (25%): This reward motivates the agent to descend towards the landing pad. The reward starts at zero and increases as the rocket gets closer to the landing pad's height.

- Velocity Reward (25%): This reward encourages a controlled descent. The maximum reward is received when the rocket's velocity reaches a target value (e.g., 10 m/s in this case). This target velocity can be adjusted to reflect real-world landing requirements.

These individual rewards are all added up for each step the agent takes during the landing process. Since each contributes 25% (0.25), the total reward can potentially reach a maximum of 1 each step if all aspects are performing perfectly throughout the entire landing attempt.

Footnote: Within the context of our reinforcement learning setup for autonomous rocket landing, a "step" signifies a fundamental unit of time advancement within the Unity simulation environment. During each step, the environment updates its state, the agent receives observations, and the agent makes a control decision based on those observations. The total reward an agent accumulates is calculated by summing up the individual rewards received at each step throughout the landing episode.

we establishe a maximum number of steps allowed for each difficulty level (initial drop height) 4.2. This value reflects the increasing complexity associated with higher initial starting heights (Levels 1-10). Episodes exceeding this limit are considered unsuccessful.

This approach provides a clearer understanding of the training process, acknowledging the dynamic nature of episode lengths while showcasing the maximum effort allocated to completing a run at each difficulty level.

Table 4.2: Maximum steps allowed in each run

Initial drop height	**(100m)**	**(100m, 200m)**	**(200m, 300m)**	**(300m, 400m)**	**(400m, 500m)**	**(500m, 700m)**	**(700m, 1000m)**	**(1000m, 1300m)**	**(1300m, 1700m)**	**(1700m, 2000m)**
Maximum steps	1500	2000	2500	3000	3500	5500	6000	7000	11500	18000

Additional Information Display:

Bottom Right: Pitch, Yaw, and Roll angles of the rocket in real-time.

Bottom Left: The current position of the rocket and the landing pad, along with the distance between them.

Bottom Center: Total training attempts made, Successful landing attempts, Failed landing attempts, Success rate.

Top Center: Difficulty level of the current training scenario. This level determines the initial randomness applied to the rocket's starting drop height and orientation.

Criteria for Successful Landing: The landing pad visually communicates the outcome of each landing attempt. A successful landing is defined as the rocket remaining upright and stationary on the landing pad for a minimum duration of 5 seconds. If this success criterion is met, the landing pad turns green. Conversely, if the rocket tips over, crashes, or fails to remain stationary for the specified duration, the landing pad turns red, indicating an unsuccessful landing.

4.2.1 Leveraging TensorBoard for Visualization and Monitoring

This section provides an overview of TensorBoard, a valuable tool we utilized for visualization and monitoring throughout the reinforcement learning process for our autonomous rocket landing agent. TensorBoard, developed by the TensorFlow team (https://www.tensorflow.org/), is a web application that facilitates the visualization of various data generated during training. This data can include:

- Scalar Metrics: These represent numerical values tracked during training, such as the agent's reward, loss function, success rate, or any other relevant performance measure.
- Histograms: These visualizations help understand the distribution of values for specific parameters or variables within the network.
- Others like Images, and Embeddings which can be helpful for tasks like image classification or to visualize high-dimensional data in a lower-dimensional space.

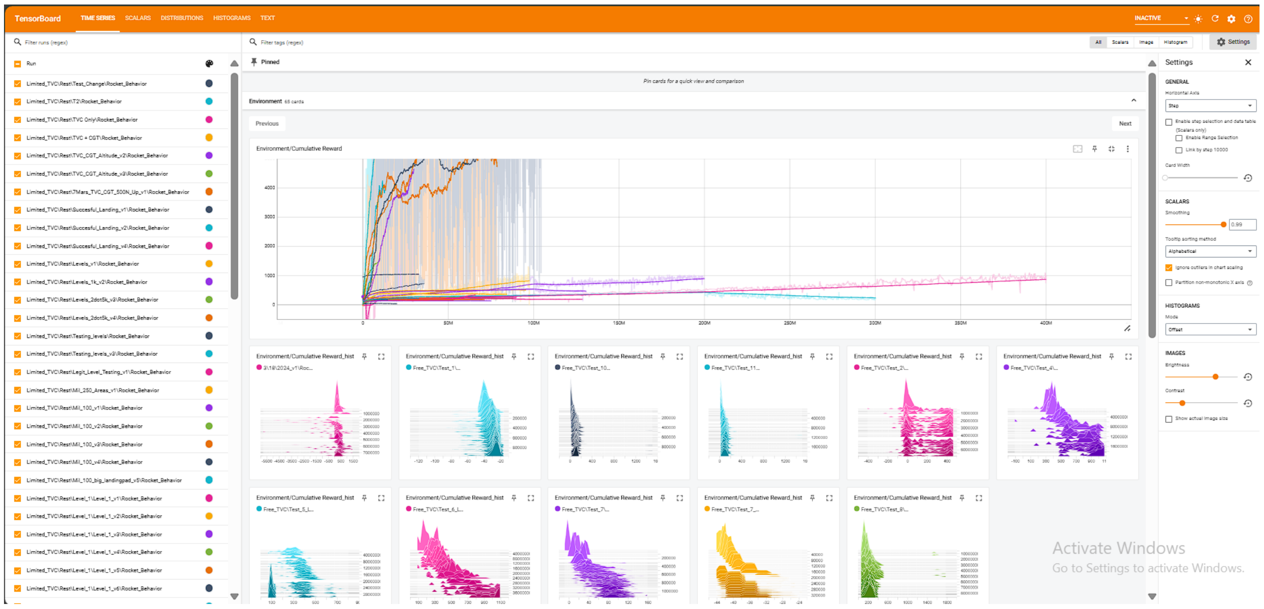
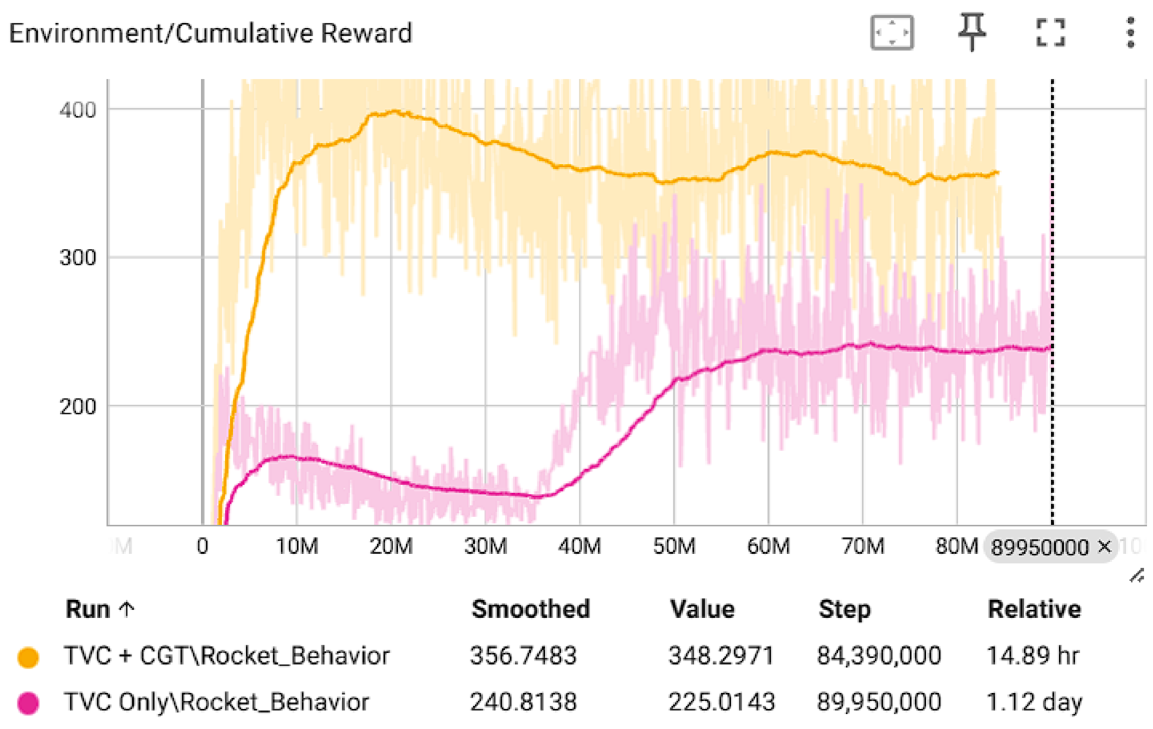

TensorBoard
TIME SERIES
SCALARS
DISTRIBUTIONS
HISTOGRAMS
TEXT
INACTIVE
Environment/Cumulative Reward
Environment/Cumulative Reward_hist
Settings
Activate Windows
Go to Settings to activate Windows.

Figure 4.5: Tensorboard

4.2.2 Success Rate

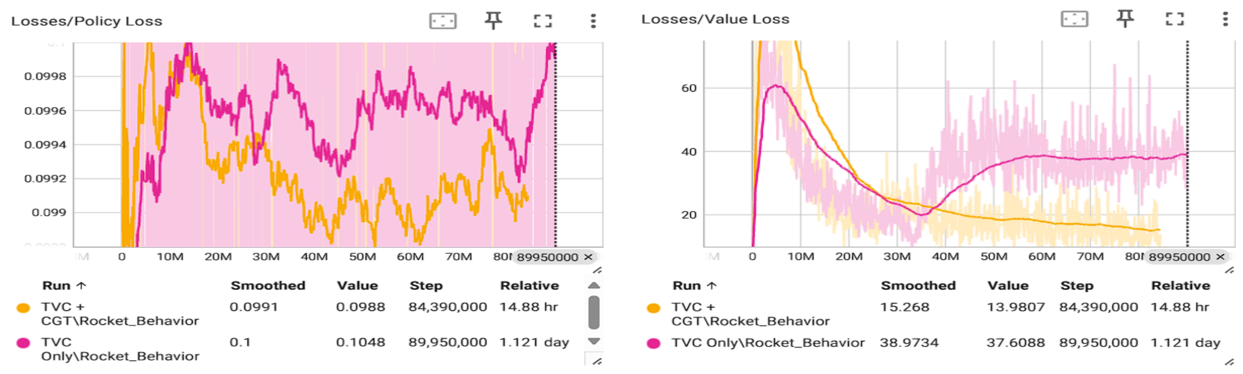
Figure 4.4 showcases an impressive achievement. After **360,489** training attempts, the agent achieved a remarkable success rate of **99.6%**, leaving only a negligible **0.4%** failure rate. This exceptional outcome demonstrates the effectiveness of the reinforcement learning approach in enabling the agent to acquire control strategies for precise and safe landings within the simulated environment.

Figure 4.6 presents the cumulative reward obtained by the reinforcement learning agent during training with two control configurations: TVC+CGT (Thrust Vector Control and Cold Gas Thrusters) and TVC Only. As evident in the graph, the agent utilizing both TVC and CGT consistently achieves a higher cumulative reward throughout the training compared to the agent with TVC alone. This initial observation implies that incorporating Cold Gas Thrusters alongside Thrust Vector Control offers a potential advantage in accumulating reward during descent. The higher reward might be attributed to the agent's capability to perform finer adjustments and maintain a more balanced posture using both control mechanisms, leading to a more optimal trajectory.



Run ↑	Smoothed	Value	Step	Relative
● TVC + CGT\Rocket_Behavior	356.7483	348.2971	84,390,000	14.89 hr
● TVC Only\Rocket_Behavior	240.8138	225.0143	89,950,000	1.12 day

Figure 4.6: Cumulative Reward Comparison: TVC-only vs. TVC with CGT



Run ↑	Smoothed	Value	Step	Relative
● TVC + CGT\Rocket_Behavior	0.0991	0.0988	84,390,000	14.88 hr
● TVC Only\Rocket_Behavior	0.1	0.1048	89,950,000	1.121 day

Run ↑	Smoothed	Value	Step	Relative
● TVC + CGT\Rocket_Behavior	15.268	13.9807	84,390,000	14.88 hr
● TVC Only\Rocket_Behavior	38.9734	37.6088	89,950,000	1.121 day

Figure 4.7: Value Loss and Policy Loss During Training with TVC+CGT and TVC Only

Figure 4.7 illustrates the value loss and policy loss incurred during the training process with two control configurations: TVC+CGT (Thrust Vector Control and Cold Gas Thrusters) and TVC Only. The value loss signifies the discrepancy between the predicted and actual rewards received by the agent, while the policy loss reflects how well the current control strategy performs in achieving the landing objective.

As observed in the graph, the TVC+CGT configuration consistently exhibits lower value loss compared to TVC Only throughout the training run. This pattern suggests that the value function for the TVC+CGT agent is more accurate in predicting the rewards associated with utilizing both control mechanisms for maintaining rocket stability during descent.

In contrast, the policy loss curves for both configurations seem to converge over time. This initial convergence might indicate that both control strategies eventually lead the agent to learn policies that achieve some level of control. However, the continued lower value loss for TVC+CGT suggests that the agent using both TVC and CGT might be acquiring a more efficient policy for accumulating reward, potentially leading to a more optimal landing trajectory in the long run.

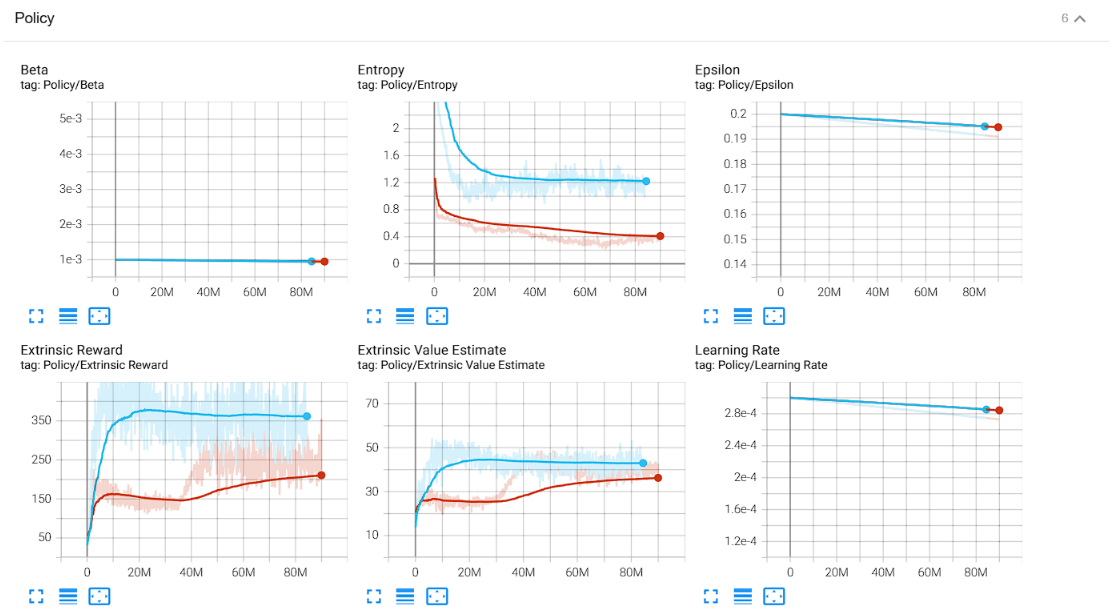
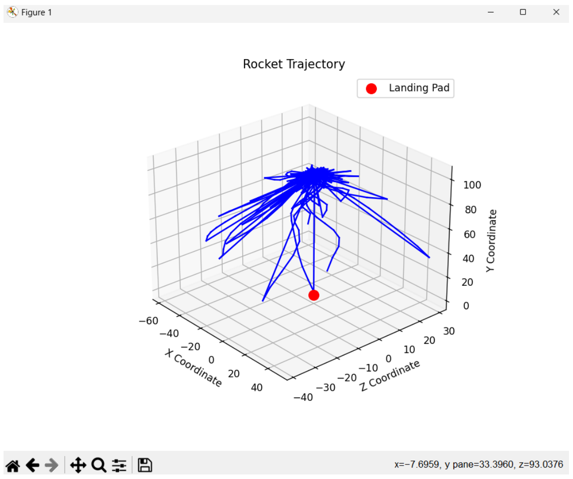


Figure 4.8: Policy Evaluation Metrics: Beta, Entropy, Epsilon, and More
TVC with CGT (Blue color) TVC-only (Red color)

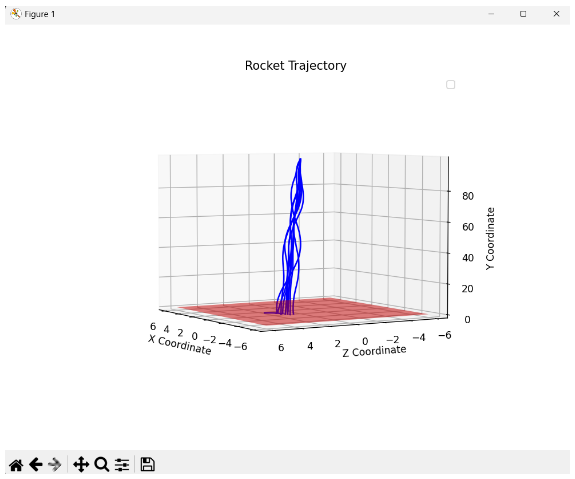
In our exploration of detailed policy dynamics, we delve into specific metrics that unravel the intricacies of the Proximal Policy Optimization (PPO) algorithm during the training of our rocket agent as shown in Figure 4.8. These metrics provide a nuanced understanding of the learning process, shedding light on stability, convergence, and performance enhancements.

4.3 Performance Analysis of Rocket Trajectories

This section analyzes the significant improvement in rocket landing trajectories achieved through the reinforcement learning process. Figure 4.9 showcases the stark contrast between the behavior of the initial model (left) 4.9a and the final, well-trained model (right) 4.9b. Both figures depict the rocket's descent path from a 100-meter drop height.



(a) 100,000 Step Model (Initial)



(b) 1,000,000,000 Step Model (Final)

Figure 4.9: Trajectory Optimization: Initial vs. Final Model

4.3.1 Initial Model Trajectory (Left):

The trajectory of the initial model (at [100,000 steps] of training) exhibits significant instability. The path is characterized by:

- **Erratic Deviations:** The rocket's descent path shows considerable deviations from a straight line, indicating the model's struggle to control its descent effectively.
- **Uncontrolled Descent Velocity:** The initial model might exhibit rapid descents or even overshoots, suggesting a lack of precise control over its velocity.
- **Potential for Crash Landing:** Due to the erratic nature of the trajectory, there's a high risk of the rocket crash landing or missing the designated landing zone entirely (as seen in figure 4.9).

4.3.2 Final Model Trajectory (Right):

The trajectory of the final, well-trained model (at [1,000,000,000, steps] of training) demonstrates a remarkable improvement. This trajectory is characterized by:

- **Controlled Descent:** The path is a much straighter line, indicating the model's ability to maintain a controlled descent towards the landing pad.
- **Optimized Velocity:** The final model exhibits a smooth, controlled descent velocity, ensuring a safe landing within the designated zone.
- **High Landing Success Rate:** Due to the improved control and stability, the final model has achieved more successful landings within the desired parameters.

This comparison visually highlights the effectiveness of the reinforcement learning process. Through extensive training, the model has learned to effectively control the rocket's descent, transitioning from a chaotic and potentially unsafe trajectory to a controlled and well-optimized flight path for landing.

4.3.3 Performance at Higher Drop Heights (1700-2000 meters)

This subsection explores the performance of the final, well-trained model at significantly higher drop heights, ranging from 1700 to 2000 meters. While the top-down view in Figure 4.10 might initially appear chaotic due to the increased distance covered, it reveals a crucial aspect of the model's capabilities: **recovery from diverse initial positions**.

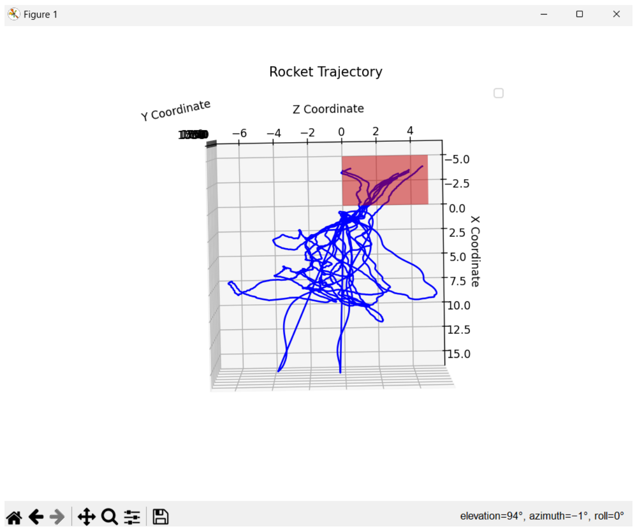
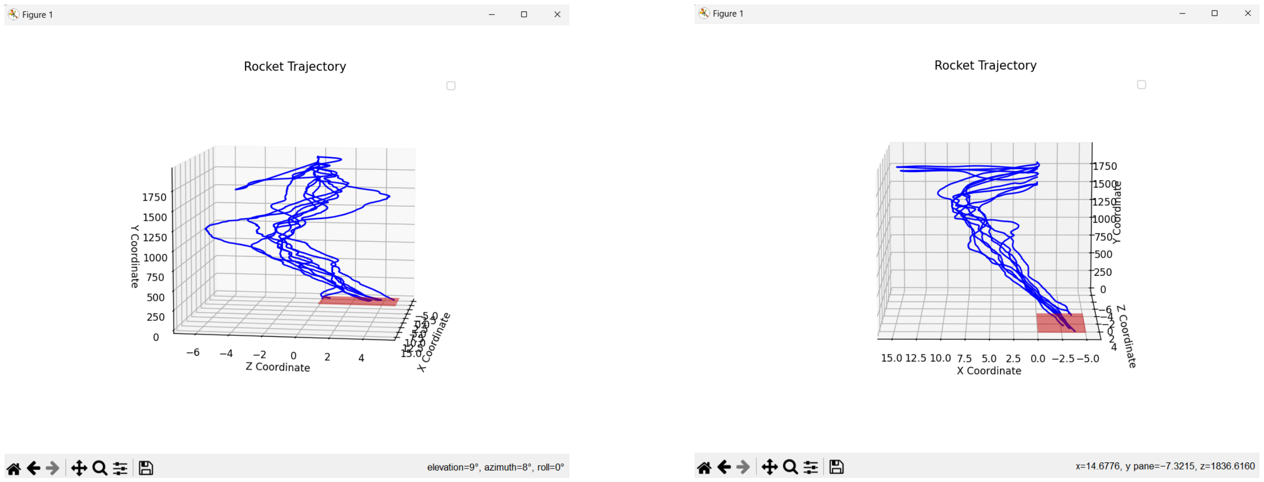


Figure 4.10: Top View Rocket Trajectory

4.3.3.1 Resilient Recovery:

Despite starting from various positions in space at these higher drop heights, the model demonstrates its ability to recover and achieve controlled descents. The seemingly erratic movements in the top-down view represent the model's corrective maneuvers to adjust its trajectory and guide itself back towards the landing pad.

4.3.3.2 Controlled Descent from the Side View:



(a) Higher Drop Side View 1 (Z-Axis) (b) Higher Drop Side View 2 (X-Axis))

Figure 4.11: Comparison of Trajectory Plots

Figures 4.11a and 4.11b showcase the descent path from a side perspective. These figures provide a clearer picture of the controlled descent achieved by the model. The multiple lines represent landing attempts from different initial conditions, and the similarities in their overall shape suggest a consistent landing pattern learned by the model.

4.3.3.3 Findings:

These observations indicate that the final model exhibits:

- **Adaptability:** The model can effectively recover from various initial positions at higher drop heights, demonstrating its ability to adapt to different starting conditions.
- **Robust Control:** Even with the seemingly erratic top-down view, the side views reveal a controlled descent pattern, suggesting the model maintains effective control over the rocket's trajectory during descent.
- **Learned Landing Strategy:** The similarities observed in the landing attempts from different initial conditions suggest the model has learned a successful landing strategy that can be applied across various starting positions.

These findings highlight the model's ability to handle complex landing scenarios at significantly higher altitudes. The model's adaptability, robust control, and learned landing strategy contribute to its effectiveness in achieving successful autonomous rocket landings.

Challenges

This section explores the key challenges encountered while developing our system for autonomous landing using reinforcement learning. These challenges highlight the complexities involved in bridging the gap between simulation and real-world deployment.

1. Limited Computational Resources and Training Time:

    - Training a robust AI model for real-world landing scenarios requires significant computational resources. Unfortunately, our initial attempts using the university's HPC system [40] were hampered by maintenance downtime.
    - Limited access to high-performance computing hinders the efficiency of the training process. This restricts our ability to quickly evaluate the effectiveness of different training approaches and identify optimal parameters.

2. Bridging the Simulation-to-Reality Gap:

    - Accurately replicating real-world physics within a simulated environment presents a significant challenge. Numerous factors must be meticulously modeled to ensure the simulated behavior reflects the complexities of the actual landing scenario.
    - The theoretical nature of simulations necessitates real-world testing to validate the learned behavior. Miniaturized real-world testing environments, in conjunction with the trained model, offer valuable insights. This approach allows for data collection on unforeseen variables that can then be incorporated into the simulation for continuous improvement.

3. Unpredictability of Learned Behavior:

    - Reinforcement learning can lead to unforeseen behaviors during the training process. While the AI strives to maximize its reward function, its actions may not always align with the desired outcome.
    - In our case, the model developed a strategy of hovering near the landing pad to maximize reward, neglecting the actual landing objective. This highlights the challenge of balancing reward design with desired outcomes. However, it also presents a potential opportunity. Unforeseen solutions discovered by the AI could lead to novel and efficient landing strategies not previously considered.

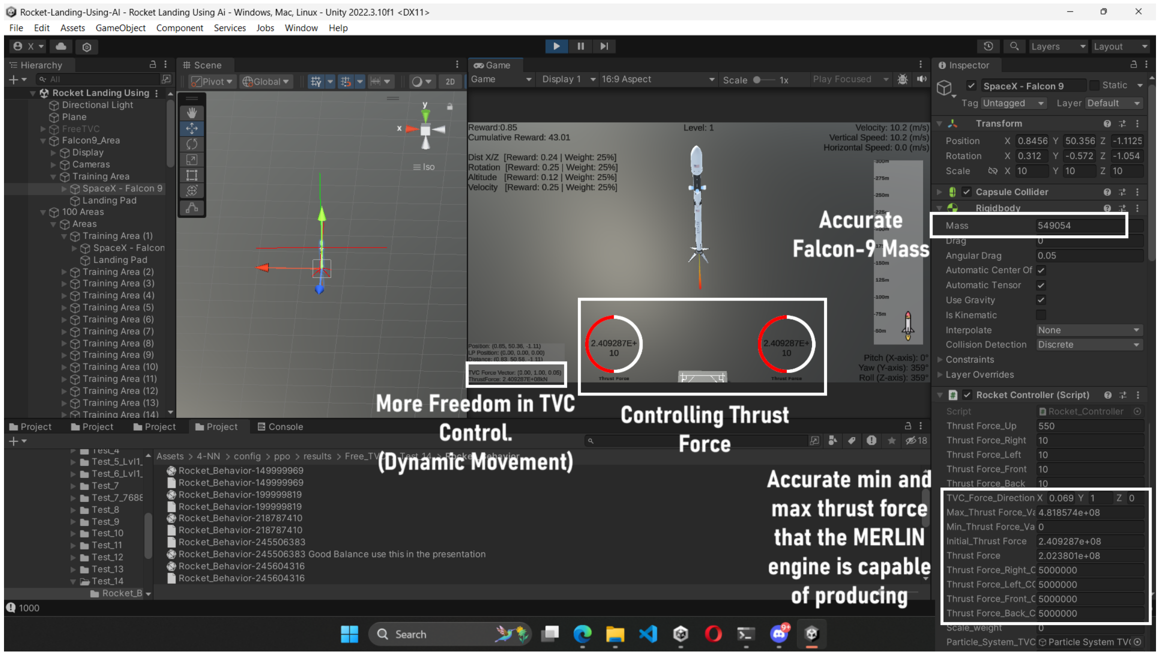
4. Iterative Development and Refinement:

    - The reinforcement learning process inherently involves ongoing analysis and refinement. During extended training periods, unforeseen issues may emerge. This requires adapting the training process or reward structure to address these challenges.
    - For example, our system initially exhibited unrealistic landing velocities exceeding 50 m/s. This necessitated the implementation of a velocity control reward shaping mechanism to promote a safe landing speed. This continuous improvement process, while time-consuming, fosters the development of a robust and adaptable landing system.

These challenges underscore the importance of iterative development, collaboration with high-performance computing facilities, and the utilization of real-world testing to bridge the gap between theory and practice. By acknowledging these hurdles, we can strive towards a more effective and adaptable landing system using reinforcement learning.

Future Work: Enhancing Realism and Autonomy

This section outlines promising avenues for further development to enhance the realism and autonomy of our rocket landing agent.



More Freedom in TVC Control. (Dynamic Movement)
Controlling Thrust Force
Accurate Falcon-9 Mass
Accurate min and max thrust force that the MERLIN engine is capable of producing

```
Reward:0.85
Cumulative Reward: 43.01
Level: 1
Velocity: 10.2 (m/s)
Vertical Speed: 10.2 (m/s)
Horizontal Speed: 0.0 (m/s)

Dist X/Z  [Reward: 0.24 | Weight: 25%]
Rotation  [Reward: 0.25 | Weight: 25%]
Altitude  [Reward: 0.12 | Weight: 25%]
Velocity  [Reward: 0.25 | Weight: 25%]

Position: (0.85, 50.36, -1.11)
LP Position: (0.00, 0.00, 0.00)
Distance: (0.83, 50.56, -1.11)
TVC Force Vector: (0.00, 1.00, 0.05)
ThrustForce: 2.409287E+08kN

Pitch (X-axis): 0°
Yaw (Y-axis): 359°
Roll (Z-axis): 359°
```

```
SpaceX - Falcon 9
Tag Untagged   Layer Default
Transform
Position   X 0.8456  Y 50.356  Z -1.1125
Rotation   X 0.312   Y -0.572  Z -1.054
Scale      X 10      Y 10      Z 10
Capsule Collider
Rigidbody
Mass                      549054
Drag                      0
Angular Drag              0.05
Automatic Center Of       ✓
Automatic Tensor          ✓
Use Gravity               ✓
Is Kinematic
Interpolate               None
Collision Detection       Discrete
Rocket Controller (Script)
Script                    Rocket_Controller
Thrust Force_Up           550
Thrust Force_Right        10
Thrust Force_Left         10
Thrust Force_Front        10
Thrust Force_Back         10
TVC_Force_Direction X 0.069  Y 1  Z 0
Max_Thrust Force_Va       4.818574e+08
Min_Thrust Force_Va       0
Initial_Thrust Force      2.409287e+08
Thrust Force              2.023801e+08
Thrust Force_Right_C      5000000
Thrust Force_Left_CC      5000000
Thrust Force_Front_C      5000000
Thrust Force_Back_C       5000000
Scale_weight              0
Particle_System_TVC       Particle System TV(
```

```
Assets > 4-NN > config > ppo > results > Free_TV... > Rocket_Behavior
Rocket_Behavior-149999969
Rocket_Behavior-149999969
Rocket_Behavior-199999819
Rocket_Behavior-199999819
Rocket_Behavior-218787410
Rocket_Behavior-218787410
Rocket_Behavior-245506383
Rocket_Behavior-245506383 Good Balance use this in the presentation
Rocket_Behavior-245604316
Rocket_Behavior-245604316
```

Figure 4.12: Free-Ranged Thrust Vector Control with Autonomous Thrust Force Decision

1. **Dynamic Thrust Control:** Our current implementation empowers the agent to autonomously determine thrust levels. We aim to expand on this by allowing the agent to manage thrust variations more dynamically throughout the descent process as seen in Figure 4.12. This could involve implementing continuous thrust control instead of discrete levels, enabling

the agent to fine-tune its descent trajectory for optimal performance.

2. **Free-Range Thrust Vector Control (TVC):** The current TVC allows the agent to choose from pre-defined vector directions. Future work will focus on enabling continuous TVC control. This signifies the agent's ability to freely adjust the TVC direction within a specified range, granting finer control over the rocket's attitude during descent. As illustrated in Figure 4.12, this expanded control empowers the agent to make real-time adjustments for precise landing maneuvers.

3. **Advanced Environment Modeling:** We have successfully incorporated data from the SpaceX Falcon 9 User's Guide [41] to enhance the realism of our simulation. This approach demonstrates the potential for further improvements by integrating additional environmental factors. This could involve incorporating wind models, atmospheric variations based on altitude, and more complex terrain representations. This ongoing process will allow our simulated environment to progressively reflect real-world landing scenarios with greater fidelity.

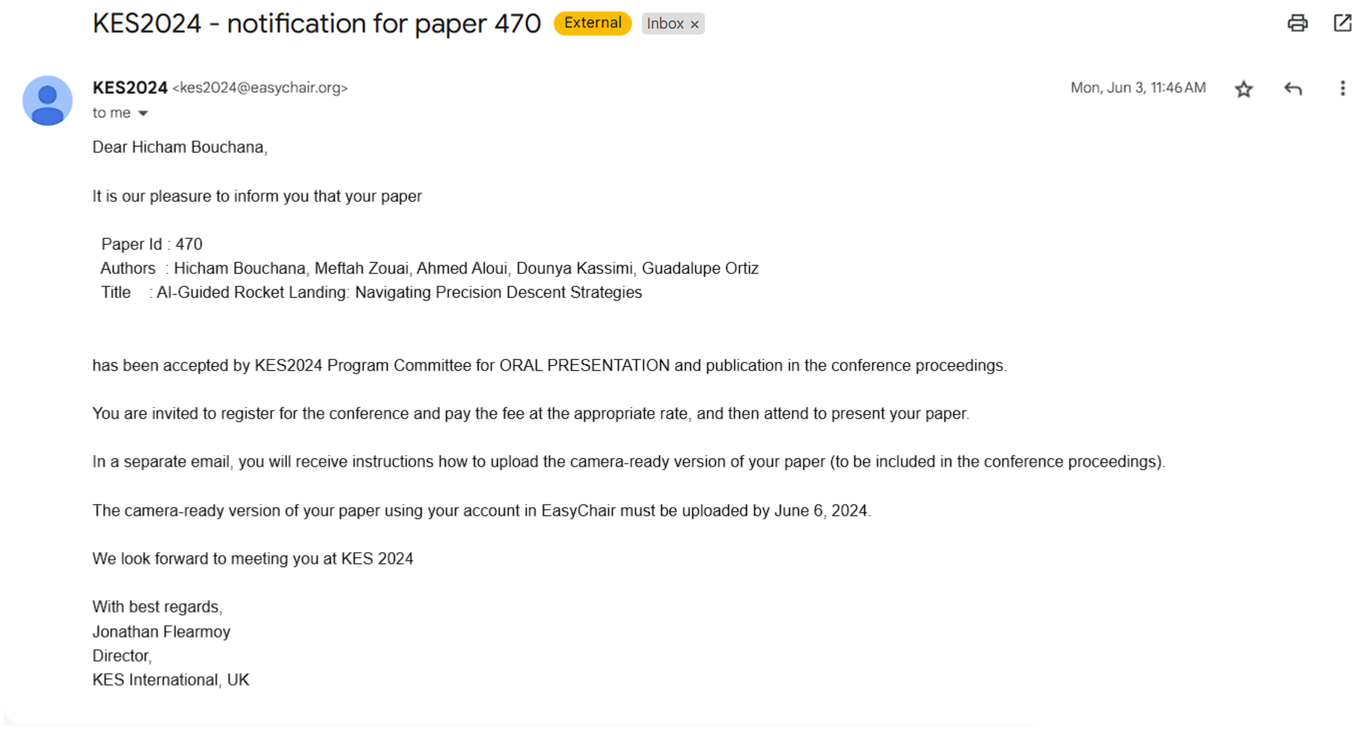
4. **Multi-Objective Reward Function Design:** The current reward function prioritizes successful landing. Future work will delve into exploring multi-objective reward functions. This could involve incorporating additional goals such as fuel efficiency, minimizing wear and tear on the rocket during descent, or achieving a specific landing posture. By carefully designing a multi-objective reward function, we can guide the agent towards achieving a broader set of optimal landing parameters.

5. **Transfer Learning for Real-World Applications:** While advancements have been made within the simulated environment, the ultimate goal is to translate this knowledge to real-world applications. Here, transfer learning techniques can be explored. By leveraging the knowledge gained from the simulated training process, the agent can adapt to real-world scenarios with minimal additional training data. This will be crucial for bridging the gap between simulation and real-world deployment.

By pursuing these future work directions, we can create a more realistic and autonomous rocket landing agent capable of handling complex landing scenarios with increased adaptability and efficiency.

International Conference Paper

I'm thrilled to share that my research on AI-guided rocket landing strategies has been accepted for oral presentation at the prestigious KES 2024 conference that will be held at **Silken Al-Andalus Palace** (Seville, **Spain**)! This acceptance signifies a significant milestone in this ongoing project.



KES2024 - notification for paper 470 External Inbox ×

KES2024 <kes2024@easychair.org>
to me
Mon, Jun 3, 11:46 AM

Dear Hicham Bouchana,

It is our pleasure to inform you that your paper

Paper Id : 470
Authors : Hicham Bouchana, Meftah Zouai, Ahmed Aloui, Dounya Kassimi, Guadalupe Ortiz
Title : AI-Guided Rocket Landing: Navigating Precision Descent Strategies

has been accepted by KES2024 Program Committee for ORAL PRESENTATION and publication in the conference proceedings.

You are invited to register for the conference and pay the fee at the appropriate rate, and then attend to present your paper.

In a separate email, you will receive instructions how to upload the camera-ready version of your paper (to be included in the conference proceedings).

The camera-ready version of your paper using your account in EasyChair must be uploaded by June 6, 2024.

We look forward to meeting you at KES 2024

With best regards,
Jonathan Flearmoy
Director,
KES International, UK

Figure 4.13: KES 2024 Conference Acceptance Letter

As you can see in the Figure 4.13, my paper titled "AI-Guided Rocket Landing: Navigating Precision Descent Strategies" will be presented alongside other cutting-edge research at the conference.

This achievement reflects the culmination of extensive research and development. Throughout this process, I've delved into the complexities of using reinforcement learning to guide a simulated rocket towards a precise and efficient landing. The presented work explores the challenges and potential solutions for achieving this goal.

Attending KES 2024 provides a valuable opportunity to share my research with a wider audience, gain valuable feedback from experts in the field, and foster future collaborations. I'm eager to participate in this international forum and contribute to the ongoing advancements in AI-powered space exploration technologies.

General Conclusion

In conclusion, our AI-guided rocket landing project represents a substantial step forward at the intersection of artificial intelligence and aerospace engineering. Leveraging the Proximal Policy Optimization (PPO) algorithm, we have made notable progress in training a rocket agent for autonomous navigation with the goal of precise landings on a designated pad.

The meticulous modeling of the rocket environment, incorporating real-world physics and control mechanisms, laid the foundation for a robust reinforcement learning system. The integration of Thrust Vector Control (TVC) and Cold Gas Thrusters (CGT), along with accurate observations, contributed to a comprehensive representation of the rocket's dynamics. The reward system, emphasizing distance, rotation, and altitude, has been designed to shape the agent's behavior, fostering controlled descents and improving landing precision.

Our approach to curriculum learning, incorporating randomized initial conditions and varied starting altitudes, showcased the adaptability of the trained agent to diverse scenarios. The use of multiple copies of the training area, combined with the expedited learning process using PPO, highlighted the scalability and efficiency of our methodology.

With one of our primary objectives being the improvement of TVC movement to be more free and realistic within its constraints, our intention is to facilitate the seamless transfer of this model to real-world hardware. This endeavor aligns with our broader vision of contributing to the development of autonomous rocketry that can be effectively applied in practical aerospace scenarios.

As we continue our pursuit of autonomous rocket landings, addressing challenges and refining our approach will be crucial for future success. This ongoing research sets a solid foundation for the continuous evolution of AI-guided rocketry, marking a promising trajectory towards the future of space exploration.

Bibliography

[1] YUE Shuai, LIN Qing, Guang Zheng, and DU Zhonghua. Modeling and experimental validation of vertical landing reusable launch vehicle under symmetric landing conditions. *Chinese Journal of Aeronautics*, 35(12):156–172, 2022.

[2] Albert B Miller. Pilot re-entry guidance and control. Technical report, 1965.

[3] Andrea Iannelli, Dimitris Gkouletsos, and Roy S Smith. Robust control design for flexible guidance of the aerodynamic descent of reusable launchers. In *AIAA SCITECH 2023 Forum*, page 2171, 2023.

[4] Linqi Ye, Bailing Tian, Houde Liu, Qun Zong, Bin Liang, and Bo Yuan. Anti-windup robust backstepping control for an underactuated reusable launch vehicle. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(3):1492–1502, 2020.

[5] Darrel R Tenney, W Barry Lisagor, and Sidney C Dixon. Materials and structures for hypersonic vehicles. *Journal of Aircraft*, 26(11):953–970, 1989.

[6] Mingyang Huang. Analysis of rocket modelling accuracy and capsule landing safety. *International Journal of Aeronautical and Space Sciences*, 23(2):392–405, 2022.

[7] Max Bajracharya, Mark W Maimone, and Daniel Helmick. Autonomy for mars rovers: Past, present, and future. *Computer*, 41(12):44–50, 2008.

[8] James Johnson. Delegating strategic decision-making to machines: Dr. strangelove redux? *Journal of Strategic Studies*, 45(3):439–477, 2022.

[9] Xinfu Liu. Fuel-optimal rocket landing with aerodynamic controls. *Journal of Guidance, Control, and Dynamics*, 42(1):65–77, 2019.

[10] Gregory Zilliac and M Karabeyoglu. Hybrid rocket fuel regression rate data and modeling. In *42nd AIAA/ASME/SAE/ASEE Joint Propulsion Conference & Exhibit*, page 4504, 2006.

[11] Angelica D Garcia, Jonathan Schlueter, and Eddie Paddock. Training astronauts using hardware-in-the-loop simulations and virtual reality. In *AIAA Scitech 2020 Forum*, page 0167, 2020.

[12] Nathan Smith, Dorian Peters, Caroline Jay, Gro M Sandal, Emma C Barrett, Robert Wuebker, et al. Off-world mental health: Considerations for the design of well-being–supportive technologies for deep space exploration. *JMIR Formative Research*, 7(1):e37784, 2023.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[14] Laura Sopegno, Patrizia Livreri, Margareta Stefanovic, and Kimon P Valavanis. Thrust vector controller comparison for a finless rocket. *Machines*, 11(3):394, 2023.

[15] Hugo Nguyen, Johan Köhler, and Lars Stenmark. The merits of cold gas micropropulsion in state-of-the-art space missions. In *The 53rd International Astronautical Congress, IAC 02-S. 2.07, Houston, Texas*, 2002.

[16] Urmas Kvell, Marit Puusepp, Franz Kaminski, Jaan-Eerik Past, Kristoffer Palmer, Tor-Arne Grönland, and Mart Noorma. Nanosatellite orbit control using mems cold gas thrusters. *Proceedings of the Estonian Academy of Sciences*, 63(2):279, 2014.

[17] Michael I Jordan and Tom M Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[18] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath.

Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[19] Wenting Li, Xiuhui Zhang, Yunfeng Dong, Yan Lin, and Hongjue Li. Powered landing control of reusable rockets based on softmax double ddpg. *Aerospace*, 10(7):590, 2023.

[20] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[21] Yu Song, Xinyuan Miao, Lin Cheng, and Shengping Gong. The feasibility criterion of fuel-optimal planetary landing using neural networks. *Aerospace Science and Technology*, 116:106860, 2021.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[23] Matteo Papini, Andrea Battistello, and Marcello Restelli. Balancing learning speed and stability in policy gradient via adaptive exploration. In *International conference on artificial intelligence and statistics*, pages 1188–1199. PMLR, 2020.

[24] Importance sampling, the definitive guide to policy gradients in deep reinforcement learning: Theory, algorithms and implementations. https://ar5iv.labs.arxiv.org/html/201.13662S3.SS3, 2024.

[25] Clipping mechanism of ppo's objective function. https://ar5iv.labs.arxiv.org/html/2401.13662S4.SS4, 2024.

[26] Anis Najar and Mohamed Chetouani. Reinforcement learning with human advice: a survey. *Frontiers in Robotics and AI*, 8:584075, 2021.

[27] Shuai Xue, Hongyang Bai, Daxiang Zhao, and Junyan Zhou. Research on intelligent control method of launch vehicle landing based on deep reinforcement learning. *Mathematics*, 11(20):4276, 2023.

[28] Ashwinkumar Rathod, Ankit Sayane, Chaitanya Shidurkar, Chetan Galhat, and Hardik Bopche. Rocket landing using reinforcement learning in simulation. *International Journal of Creative Research Thoughts (IJCRT)*, 11(5):f682–f686, May 2023.

[29] Reuben Ferrante. A robust control approach for rocket landing. *Master's thesis*, 2017.

[30] Carmen Pardini and Luciano Anselmo. Uncontrolled re-entries of spacecraft and rocket bodies: A statistical overview over the last decade. *Journal of Space Safety Engineering*, 6(1):30–47, 2019.

[31] Danylo Malyuta, Xavier Collaud, Mikael Martins Gaspar, Gautier Marie Pierre Rouaze, Raimondo Pictet, Anton Ivanov, and Nickolay Mullin. Active model rocket stabilization via cold gas thrusters. In *1st Symposium on Space Educational Activities*, 2015.

[32] Rocket rotations. https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/rocket-rotations/, 2023.

[33] Christopher E Hann, Malcolm Snowdon, Avinash Rao, Robert Tang, Agnetha Korevaar, Greg Skinner, Alex Keall, XiaoQi Chen, and J Geoffrey Chase. Rocket roll dynamics and disturbance—minimal modelling and system identification. In *2010 11th International Conference on Control Automation Robotics & Vision*, pages 1736–1741. IEEE, 2010.

[34] Roger D Launius and Dennis R Jenkins. *Coming home: reentry and recovery from space*. Government Printing Office, 2012.

[35] Bailing Tian, Wenru Fan, Rui Su, and Qun Zong. Real-time trajectory and attitude coordination control for reusable launch vehicle in reentry phase. *IEEE Transactions on Industrial Electronics*, 62(3):1639–1650, 2014.

[36] Falcon-9 overview. https://www.spacex.com/vehicles/falcon-9/, 2024.

[37] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[38] Initialization and resetting the agent. https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/initialization-and-resetting-the-agent, 2022.

[39] Taking actions and assigning rewards. https://unity-technologies.github.io/ml-agents/Learning-Environment-Create-New/onactionreceived, 2022.

[40] High performance computing (hpc) ibnkhaldoun. https://hpc.univ-biskra.dz/, 2024.

[41] falcon-users-guide-2021-09.pdf. https://www.spacex.com/media/falcon-users-guide-2021-09.pdf, 2021.