Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Computer Science
Multi-robot Systems

A realistic simulation of a UAV model in Unity for learning control policies in OpenAI Gym

Master's Thesis

Lev Kisselyov

Prague, January 2024

Study programme: Open Informatics
Branch of study: Software Engineering

Supervisor: Ing. Matěj Petrlík

Acknowledgments

I would like to express my gratitude to my supervisor Ing. Matěj Petrlík for his extensive guidance during the project and his quick and motivating reaction to numerous arising questions. Special gratitude goes to Bc. Tomáš Musil, my good friend, and author of the idea of thesis.

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name:	**Kisselyov Lev**	Personal ID number:	**483610**
Faculty / Institute:	**Faculty of Electrical Engineering**		
Department / Institute:	**Department of Computer Science**		
Study program:	**Open Informatics**		
Specialisation:	**Software Engineering**		

II. Master’s thesis details

Master’s thesis title in English:

A realistic simulation of a UAV model in Unity for learning control policies in OpenAI Gym

Master’s thesis title in Czech:

Realistická simulace modelu UAV v Unity pro učení řídicích strategií v OpenAI Gym

Guidelines:

This thesis aims to implement a realistic UAV model in the Unity game engine, connect it to ROS and to OpenAI Gym, and then develop and evaluate a reinforcement learning-based controller for the UAV that learns visual odometry-aware flight.

1. Get familiar with the Unity simulation environment and its use in robotics [1], the OpenAI Gym [2] framework, and the basics of the MRS UAV system [3].
2. Implement a realistic UAV in the Unity simulator, similar to the existing Gazebo MRS simulator for UAVs, and make the UAV controllable through an interface with OpenAI gym [2].
3. Formulate and design the reinforcement learning task and try several available reinforcement-learning algorithms [4] to learn a controller with which the UAV can fly without disrupting visual odometry (for example by avoiding quick motions that cause visual odometry to lose tracking [5]).
4. Implement a ROS-Unity interface and test control of the UAV with the trained model deployed as a node in ROS. Discuss the effects of the learned control policy on the quality of the visual odometry.

Bibliography / sources:

[1] Song, Yunlong, et al. "Flightmare: A flexible quadrotor simulator." Conference on Robot Learning. PMLR, 2021.
[2] Brockman, Greg, et al. "Openai gym." arXiv preprint arXiv:1606.01540 (2016).
[3] Baca, Tomas, et al. "The MRS UAV system: Pushing the frontiers of reproducible research, real-world deployment, and education with autonomous unmanned aerial vehicles." Journal of Intelligent & Robotic Systems 102.1 (2021): 26.
[4] Azar, Ahmad Taher, et al. "Drone deep reinforcement learning: A review." Electronics 10.9 (2021): 999.
[5] Wu, Xiangyu, et al. "Perception-aware receding horizon trajectory planning for multicopters with visual-inertial odometry." IEEE Access 10 (2022): 87911-87922.

Name and workplace of master’s thesis supervisor:

Ing. Matěj Petrlík   Multi-robot Systems FEE

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **14.08.2023**   Deadline for master's thesis submission: ___________

Assignment valid until: **16.02.2025**

Ing. Matěj Petrlík Supervisor’s signature	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the master’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master’s thesis, the author must state the names of consultants and include a list of references.

. Date of assignment receipt	Student’s signature

Declaration

I declare that presented work was developed independently, and that I have listed all sources of information used within, in accordance with the Methodical instructions for observing ethical principles in preparation of university theses.

Date ....

Abstract

UAV simulation is a complex field that is rapidly evolving, resulting in numerous available software options. The integration and cross-compatibility of the simulators pose a challenge for researchers in designing new robotic applications. Therefore, there is a constant need to expand the existing toolkit stack and integrate the new backward-compatible simulation environments that bring new perspectives to the research by utilizing their advantages. This work introduces Unity3D as a simulation tool in the UAV field, analyzing it from multiple perspectives. The project introduces a realistic simulation of the drone control system in combination with the Unity physics engine. Moreover, the references to the control system and other simulation data can be interchanged with the existing ROS tools through the ROS-TCP connector. Contributing to the multirotor RL research, we provide an examination of the native to Unity ML-Agents framework in terms of performance, convenience of the setup, and flexibility. The thesis provides a testing showcase of an RL model training aimed to teach the drone position control policy while taking into consideration visual odometry constraints. Consequently, the paper compares two state-of-the-art RL algorithms, PPO and SAC, in multiple categories. The final contribution of the project is a toolkit that can be utilized to provide a quick start for other developers in developing their Unity3D multirotor applications.

Keywords : Unmanned Aerial Vehicles, Unity3D, Multirotor Simulation, ML-Agents, Reinforcement Learning, Visual Odometry, ROS, Multirotor Control System

Abstrakt

Simulace bezpilotních multirotorů představuje komplexní oblast, která se rychle rozvíjí, což vede k mnoha dostupným softwarovým možnostem. Integrace a vzájemná kompatibilita simulátorů představují výzvu pro výzkumníky při návrhu nových robotických aplikací. Je proto stále zapotřebí rozšiřovat existující sadu nástrojů a integrovat nová zpětně kompatibilní simulační prostředí, která přinášejí nové perspektivy výzkumu využitím svých výhod.

Tato práce představuje Unity3D jako simulační nástroj v oblasti UAV a analyzuje ho z různých perspektiv. Projekt přináší realistickou simulaci řídicího systému dronu v kombinaci s simulací fyziky v Unity. Představený softwarový balíček umožňuje vzajemnou kominukaci simulačního prostředí s existujícími nástroji v ROSu prostřednictvím rozhraní ROS-TCP connector. Potenciálním příspěvkem do výzkumu posilovaného učení (RL) pro multirotory je poskytnutí hodnocení frameworku ML-Agents v Unity z hlediska kvality výsledných metod, možností nastavení parametrů modelu a flexibility.

Práce poskytuje ukázkové testování trénování modelu zaměřeného na učení se řízení polohy dronu s ohledem na omezení vizuální odometrie. V rámci předvedeného experimentu práce porovnává dva moderní RL algoritmy, PPO a SAC, v několika kategoriích.

Finálním přínosem projektu je sada nástrojů, která může být využita pro rychlý start vývojářů při vytváření svých aplikací pro multirotory v Unity3D.

Klíčová slova : Bezpilotní Prostředky, Automatické Řízení Multirotorů, Unity3D, ML-Agents, Posilované Učení, ROS

Contents

Chapter 1

Introduction

The need for realistic and versatile simulations has become paramount in the ever-evolving landscape of unmanned aerial vehicles (UAVs). Developing robust control systems and training environments is crucial as the world witnesses many drone applications, from surveillance and logistics to agriculture and entertainment. Simulations offer an indispensable tool in the development and testing of UAV systems. They provide a controlled environment for the development of algorithms, the exploration of new control strategies, and the training of pilots and autonomous systems. Real-world testing of UAVs can be costly, dangerous, and highly restricted in various regions. For example, developing approaches for subterranean search and rescue tasks requires extensive evaluation and expensive testing in vast environments [4] to verify the maturity of the solution before deployment as an assistive technology for first-responders. Moreover, to test the robustness of the approaches, they need to be tested in various environments such as man-made mining tunnels [22], unfinished or collapsed urban buildings [16], deep-spanning cave networks [17], or historical monuments containing objects of immense value [3]. Simulations, therefore, serve as a safe and cost-effective alternative.

This thesis embarks on a journey to create a highly realistic UAV simulation in Unity 3D, connect it to the Robot Operating System (ROS), and integrate it with a Reinforcement Learning (RL) training platform, specifically the Unity ML Agents framework. The primary objective is to design and assess a realistic physical simulation of a quadrotor model and corresponding control system for the UAV. Additionally, this research will explore the reinforcement learning approach to achieve a model that asserts stable flight control, preventing disruptions in the visual odometry of the drone. The final contribution of the thesis is a framework that will enable users to conduct simulations effortlessly and test various reinforcement learning tasks with a quick and configurable set of tools.

1.1 Related works

The area of research for modern robotics simulation environments is booming with substantial work. Depending on a problem's specification, several simulators that fit and utilize their benefits to optimize efficiency can be chosen. On the one hand, the need for perception-based control and photo-realistic visual-based training can be addressed using game engine simulators. On the other hand, a resource-consuming rendering engine is often combined with a more straightforward physics simulation, balancing the computation complexity. The following analysis underlines the advantages and drawbacks of multiple state-of-the-art simulations and describes how they address the scalability and continuous RL support.

AirSim is a heavily used simulator with a powerful rendering engine based on gaming Unreal Engine. The physics of the quadrotor is calculated by the so-called Fast Physics engine, a simplified engine with some shortcomings but efficient calculation. Nevertheless, the rendering allows working on perception-based RL tasks that rely on photo-realistic image rendering. *Figure 1.1* shows an example of a drone simulation in a suburban area performing image segmentation.

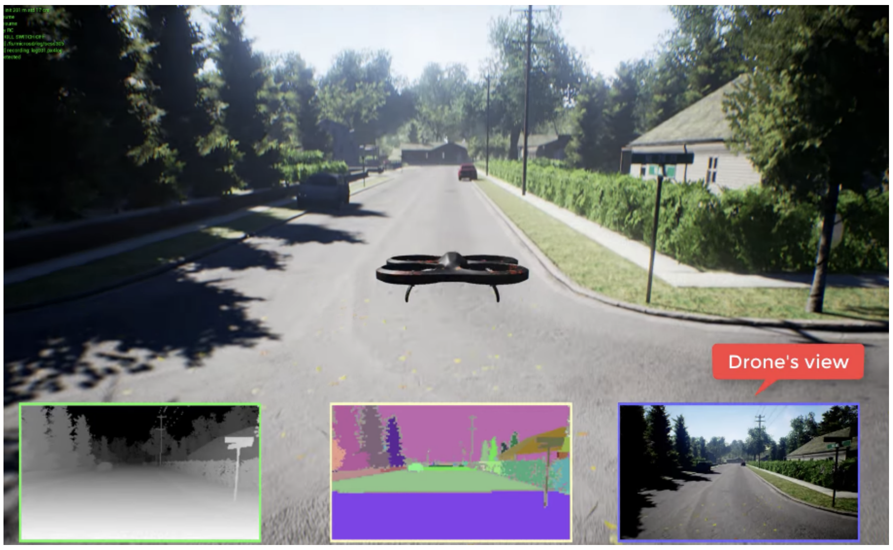


Figure 1.1: High fidelity rendering of a quadrotor in AirSim [34]

The simulator has been used as a platform for training RL tasks in extensive research [30]. The efficiency of the simulator was shown in [5], where J. Saunders et al. trained multiple agents in parallel and obtained significant speed-up in teaching quadrotor visual navigation. AirSim plugin for the Unreal Engine 4 game engine was used to design an RL-specialized OpenAI Gym wrapper, AirLearning [15], that offers advanced settings to help mimic real-life drone behavior. For instance, there is artificial downgrading of the computation efficiency to simulate on-board processor capability. Lastly, it is essential to note that Microsoft will no longer support the open-source package, as they are developing a new commercial package called *Project AirSim.*

The codependency of the rendering and the physics engine was addressed in the **Flight-Mare** simulator [18] by decoupling the rendering and physics engine. Figure 1.2 outlines the structure of Flightmare, specifically its modular rendering and dynamics modeling and connection to the Python backend for additional advanced applications.

Flightmare represents a fair balance of the simulation's efficiency by maintaining modularly separated solid photo-realistic visuals and a flexible physics engine. The decoupling is a design choice that allows the simulator to achieve noteworthy speed-up in RL. Moreover, for the physics simulation, depending on the application, Flightmare offers three different approaches to quadrotor dynamics:

- RotorS, a drone-oriented dynamics package providing more complex and precise but less efficient results [35];
- Lightweight parallel implementation of classical quadrotor dynamics optimized for high-speed sample gathering;
- Real-world dynamics presented in [26];

In its most efficient mode, the physical engine reaches 200,000 Hz simulation of a multi-core CPU. Users can use several camera sensor types, including RGB and depth cameras, as well

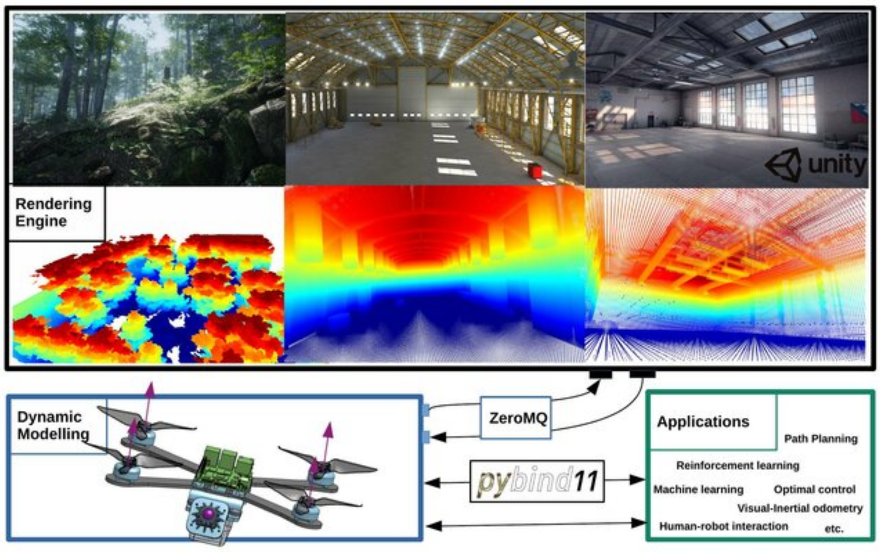


Figure 1.2: Overview of the Flightmare framework [18]

as specialized cameras for segmented views of the environment. However, the provided sensor suite does not support Light Detection and Ranging (LIDAR) sensors.

Due to its high scalability, gathering a large amount of data crucial for the reinforcement learning tasks became possible, consequently allowing the development of extreme tasks such as quadrotor racing [7].

One of the contributions of the thesis discussed in Section 1.2 is its compatibility with the MRS UAV system [13]. It is a comprehensive software stack providing essential tools for both real-life and simulation experiments. Currently, the UAV MRS system has three different types of simulation engines: Gazebo Classic, CoppeliaSim, and its internally implemented MRS Simulator.

Gazebo Classic is an open-source multi-purpose simulator allowing accurate physics simulation. The simulator has an extensive user base in the robotics field and is still being maintained. ROS natively supports Gazebo Classic. One of the benefits of the simulator is that the objects and models can be added to the environment dynamically during the simulation. As a part of the MRS UAV system, new terrains and maps created in Blender can be added. At the same time, Simulation In The Loop (SITL) can be sped up, which is beneficial for the data-demanding RL tasks. Gazebo can be combined with the OpenAI framework to implement a custom RL task; however, it requires implementing backend communication with the environment in case it resets, is held on pause, or is closed. There are multiple attempts to provide the Gazebo-ROS setup with a ready-to-use framework based on OpenAI algorithms allowing RL training [36], [25]. However, most of the toolkits do not have consistent maintenance, which means that a custom implementation is needed to keep up with the simulator updates. Nevertheless, recent publications suggest that Gazebo can still be used for RL training [1]. Gazebo implements plugin software design, allowing ROS-interconnected custom plugins to be created. The plugins are written in $C++$ and loaded into the simulation in SDF format at the beginning of the simulation. Amongst numerous use cases, plugins support quadrotor sensors and allow their adjustments and connection to ROS communication channels. In addition to the sensors, Gazebo provides plugins for manipulation with models inside the simulation environment, the environment itself, visuals, and GUI elements. Currently, Gazebo supports 4 physics engines: *ODE*, *Bullet*, *Simbody*, and *Dart*, with *ODE* as a default option.

Constructed using the C/C++ programming language, the Open Dynamics Engine (ODE) [45] represents an open-source framework featuring a dual-functional core comprising a robust simulation engine for rigid body dynamics and a collision detection engine. Characterized by its resilience and simplicity, ODE provides a stable foundation for simulating mechanical interactions, albeit with a performance profile that may be perceived as less efficient compared to more sophisticated counterparts, exemplified by the Bullet physics engine.

In contrast, Bullet, while sharing commonalities with ODE in supporting rigid bodies and streamlined collision detection, distinguishes itself by incorporating advanced functionalities such as multi-threading and native convex hull support. The latter enables the representation of convex hulls, a category of 3D shapes defined by discernable internal and external facets formed by an array of infinite planes.

Simbody is an open-source multibody dynamics software toolkit [42] implemented primarily in C++. It emphasizes flexibility and extensibility, providing an adaptable platform for complex biomechanical systems. As a consequence, it focuses on high-fidelity simulations of complex multibody systems, where the precision of the calculation has a higher priority than its efficiency.

DART [29] distinguishes itself in the realm of robotics simulation through its commitment to efficiency and real-time performance. Crafted also as a C++ library, DART emphasizes modularity, making it adjustable to the demands of articulated rigid body systems.

While Gazebo excels in simulating intricate robotic dynamics and interactions with multiple physics engines suitable for different tasks, it may not be ideal for applications that require a simulation environment with top-tier rendering quality. Gazebo emphasizes the precision of physics and the accurate representation of robotic systems more than delivering photo-realistic graphics.

CoppeliaSim is a versatile robotics simulator based on V-REP [41]. It supports various customization techniques: add-ons, plugins, and remote API clients. Multiple physical engines can process the physics calculation: Bullet physics, ODE, MuJoCo, Vortex Studio, and Newton Dynamics. A user can choose a suitable engine for a specific application depending on the precision and calculation speed requirements. Moreover, CoppeliaSim tries to utilize kinematics instead of only tracking the dynamics of the object to achieve the best performance. Figure 1.3 shows a position control application with a moving target reference.

Multiple attempts to bridge RL to CoppeliaSim have a limited capability and even less support for quadrotors training [20], [24].

1.2 Contributions

The limitation of all the simulators presented in *Section 1.1* is that they come with a set of corresponding pre-defined frameworks, meaning that the whole software stack should be built around them. For a modern, rapidly developing area of UAVs, the trait of flexibility plays an important role, as switching between different implementations poses a time-resource challenge. The benefit of the work presented in this thesis is its compatibility with the existing stack of the MRS (Multirobot Systems) team and its support for custom ROS infrastructure. It aims to establish a quick transfer between simulation environments.

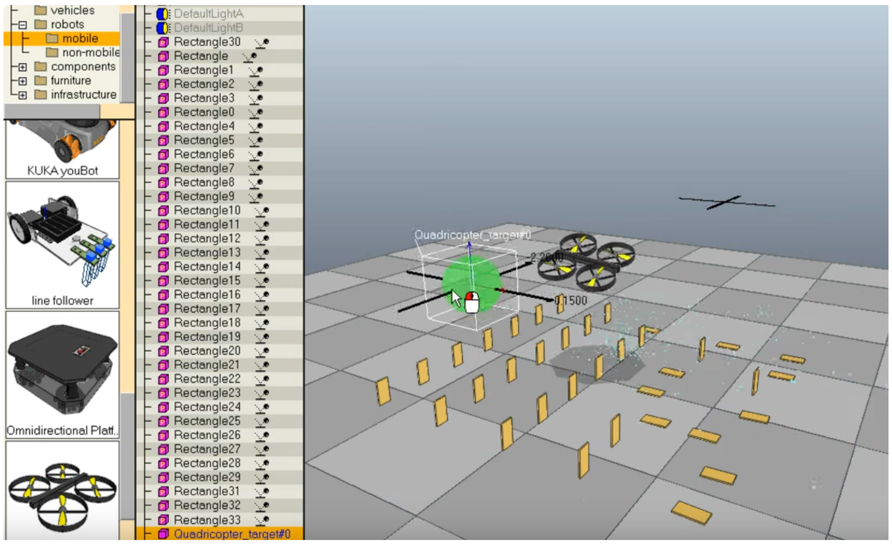


Figure 1.3: Position control simulation in CoppeliaSim [41]

Compared to existing Gazebo, CoppeliaSim, or MRS Simulator, simulations in Unity 3D can be beneficial for perception-based tasks. Unity provides multiple rendering options discussed in Section 2.1.1. Those options allow us to balance the computation speed and high-fidelity rendering. Potential usage of pre-defined Unity assets can lead to less development effort and, therefore, speed-up the development itself. Moreover, Unity provides a Unity Robotics Hub initiative [11] to support the robotics simulations. It is a toolkit bridging the Unity environment to standard robotics file formats and infrastructure. Sections 3.1 and 2.2.2 give more insights into their workflow.

Another helpful addition is a connection to the ML-Agents framework, an easy-to-setup RL toolkit allowing one to quickly develop a highly configurable training environment without the need for training code development. Despite having a configuration file interface for setting the model, ML-Agents provides an OpenAI Gym Wrapper and a low-level Python API. This combination of tools can be beneficial for testing minor RL problems without a long-term development commitment, as well as training complex networks for challenging tasks. ML-Agents now explicitly supports training cooperative behaviors. This means that one of the benefits of the framework would be its possible extension to learning collaborative swarm tasks with a shared reward function.

Moreover, while adding an entirely new simulation environment to the existing MRS stack, the presented work stays backward-compatible, allowing the hybrid splitting of different computations for the simulation between Unity scripts and ROS nodes. One such example could be the trajectory planning ROS node sending messages to the Unity script listener that sets the references for the multirotor control system.

The Unity-based physics simulation with a combination C# implementation of the drone control system makes physics simulation highly scalable. Depending on the hardware, we can achieve more than 20x time scaling factor even on a single machine.

The scripting features of Unity allow customization of the simulations, including setting up complex scenes and changing the scenes on the run, adding the weather conditions affecting the visibility of the scene, and other helpful rendering features resulting in powerful visualization capabilities. The environments can be dynamically changed; game objects can be created on the run. The changing conditions allow testing the performance of new custom algorithms for robustness.

Finally, the thesis provides a practical analysis of the presented tools and caveats encountered during the implementation of the framework.

1.3 Project Overview

The project's main deliverable is a comprehensive framework combining Unity 3D, ROS, and ML-Agents with an OpenAI Gym wrapper, creating a unified UAV simulation and control environment. This framework connects the following components:

- A set of tools for enabling a set up of simulation environment, the addition of the drones, including a walkthrough of the workflow. It explores the caveats in the compatibility of the simulation environment with the current MRS stack and suggests a solution for them.
- A MRS multirotor control system adapted to the Unity physics engine.
- ROS-Unity Connector, a bidirectional communication interface between ROS and Unity. The thesis includes testing the passing of the sensor data for visual odometry of the UAV and manual control deployed as a ROS node.
- ML-Agents integration in the form of the DroneAgent implementation that provides the configurable baseline for the training of quadrotors.
- Example of an RL training implementation testing the whole framework, including analyzing the performance of the available algorithms for a specific control learning task.

1.4 Mathematical notation

The mathematical notation used in the thesis is outlined below.

$\mathbf{x}$, $\boldsymbol{\alpha}$	vector, pseudo-vector, or tuple
$\hat{\mathbf{x}}$, $\hat{\boldsymbol{\omega}}$	unit vector or unit pseudo-vector
$\hat{\mathbf{e}}_1, \hat{\mathbf{e}}_2, \hat{\mathbf{e}}_3$	elements of the *standard basis*
$\hat{\mathbf{b}}_1, \hat{\mathbf{b}}_2, \hat{\mathbf{b}}_3$	elements of the right-handed body-fixed coordinate frame
$\mathbf{X}, \boldsymbol{\Omega}$	matrix
W	world frame
B	object's local frame
$\mathbf{T}_W^B$	translation matrix from local coordinate system to global coordinate system
$\mathbf{T}_B^W$	translation matrix from global coordinate system to local coordinate system
$\mathbf{I}$	identity matrix
$\mathbf{X}_{(a,b)}$	a specific matrix element, where a represents row and b column
$x = \mathbf{a}^\intercal \mathbf{b}$	inner product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x} = \mathbf{a} \times \mathbf{b}$	cross product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x} = \mathbf{a} \circ \mathbf{b}$	element-wise product of $\mathbf{a}$, $\mathbf{b} \in \mathbb{R}^3$
$\mathbf{x}_{(n)} = \mathbf{x}^\intercal \hat{\mathbf{e}}_n$	n^{th} vector element (row), $\mathbf{x}, \mathbf{e} \in \mathbb{R}^3$
x_d	x_d is *desired*, a reference
$\dot{x}, \ddot{x}, \dddot{x}, \ddddot{x}$	1^{st}, 2^{nd}, 3^{rd}, and 4^{th} time derivative of x
$x_{[n]}$	x at the sample n
$\mathbf{A}, \mathbf{B}, \mathbf{x}$	LTI system matrix, input matrix and input vector
$\mathbf{R}_{\mathrm{column}(i)}$	i-th column of matrix $\mathbf{R}$
SO(3)	3D special orthogonal group of rotations
SE(3)	*SO(3)* $\times \mathbb{R}^3$, special Euclidean group

Table 1.1: Mathematical notation, nomenclature, and notable symbols.

Chapter 2

Simulation Environment

This section describes Unity3D, the engine used for visual and physics simulations conducted in the thesis. It introduces the terminology used throughout the thesis and the technical features of the simulation. Later, it presents an approach for adding drones' [1] into the environment.

2.1 Unity3D Overview

Unity is a cross-platform game development engine encompassing a comprehensive suite of tools and functionalities for creating interactive digital experiences. Developed by Unity Technologies, the framework has gained prominence as a versatile and widely adopted solution in game design. Despite that, Unity facilitates the development of applications not only in the gaming industry but also extends its utility to areas such as robotics simulations and AI-based experiments. Integrating numerous packages available for installation from the Unity Assets Store makes the development of versatile robotics tasks more straightforward.

In this thesis, Unity3D is examined as a framework for both rendering and physics simulation. The engine is inherently modular, realized through a component-based architecture [39]. By leveraging this architecture, developers can construct intricate simulated landscapes by assembling modular components, each contributing to the dynamic and evolving nature of the simulation. Another potent aspect of Unity is scripting. The adaptability of Unity's native scripting language, C#, enhances the flexibility required for crafting complex simulation logic, behavior, or implementation of custom features (e.g., custom sensors).

2.1.1 Rendering Pipeline

As an engine for graphics rendering, the most recent version of Unity supports multiple graphics APIs, such as DirectX, Metal, OpenGL, and Vulkan [8]. The availability can vary from platform to platform.

Every rendering pipeline in Unity is realized by first removing the objects irrelevant to the current picture and objects that are not visible on the camera[2]. After that, the 3D image is projected into a 2D window and put into a pixel buffer, followed by rasterization. Lastly, Unity's rendering engine includes a broad suite of post-processing effects that can be applied to the final rendered image. These effects, such as depth of field, bloom, and ambient occlusion, add an additional layer of visual adjustments to the simulation.

Unity provides several pre-built options for its rendering pipeline, each providing different levels of customization, scalability, and, accordingly, the quality of the graphics:

[1]the method can be generalized to any robotic 3D models

[2]in Unity terminology, this is called frustum culling and occlusion culling

- *Built-In pipeline*, default option, which is general-purpose;
- *Universal Render Pipeline* (URP), scriptable pipeline, which allows scalability as well as the flexibility of the graphics implementation;
- *High Definition Render Pipeline* (HDRP), scriptable pipeline, used in case the photo-realism is essential.

Both URP and HDRP pipelines are inherently customizable. However, the same level of flexibility in implementing the earlier-mentioned rendering steps for the Built-In pipeline is a paid option. We currently use the default option for our framework, although scriptable render pipelines can be considered depending on the application. There is a possibility for custom shader creation. This empowers developers to tailor the visual aspects of materials and lighting to suit the specific requirements of a simulation. The ability to create custom shaders adds a layer of flexibility, enabling the fine-tuning of visual elements for diverse simulation scenarios. To optimize performance, Unity's rendering engine incorporates graphics processing unit (GPU) instancing, which allows for the efficient rendering of multiple instances of the same object, reducing the computational burden on the device. This optimization enables the speedup of simulations with complex scenes with numerous elements that must be rendered in real-time without sacrificing performance.

2.1.2 Physics Simulation

Unity employs NVIDIA's PhysX SDK as its core physics simulation engine. This SDK is utilized in several other game engines, including Unreal Engine (version 3 and higher), Gamebryo, Vision (version 6 and higher), HeroEngine, Instinct Engine, Panda3D, and BigWorld. PhysX within Unity handles the simulation of rigidbody dynamics, enabling developers to simulate the physical behavior of objects with mass and simulated properties. Unity's rigid bodies are instances of PhysX rigid bodies, and their interactions, such as gravity, forces, and collisions, are managed by the underlying PhysX engine. Specifically, PhysX provides the technology for Unity's collider components for accurate collision detection and corresponding response, defining shapes and volumes within the simulation. Unity supports various collider shapes, and PhysX ensures realistic interactions during runtime. When collisions occur, the engine calculates physical responses, including forces, friction, and restitution, providing visually plausible and physically realistic interactions. An early analysis of the physics engines [44] stated that PhysX had the most features and performed better in the integrator test, a precision examination of calculating a rigid body position due to applying multiple forces on it.

Being maintained by NVIDIA (as it acquired Ageia, the creators of PhysX), PhysX supports GPU acceleration, allowing specific physics calculations to be offloaded to the graphics processing unit. This feature, particularly utilizing CUDA-enabled GeForce GPUs, contributes to performance optimization, enhancing computational efficiency in simulations with a high number of physics calculations.

2.1.3 Unity UI

Unity Editor is a development environment provided by Unity Technologies. It serves as the primary interface through which developers, designers, and artists can build, design, and iterate on their Unity projects. Users can create objects, interact with scenes, and set the scripts' public variables from the editor. It also provides an interface to manage the project

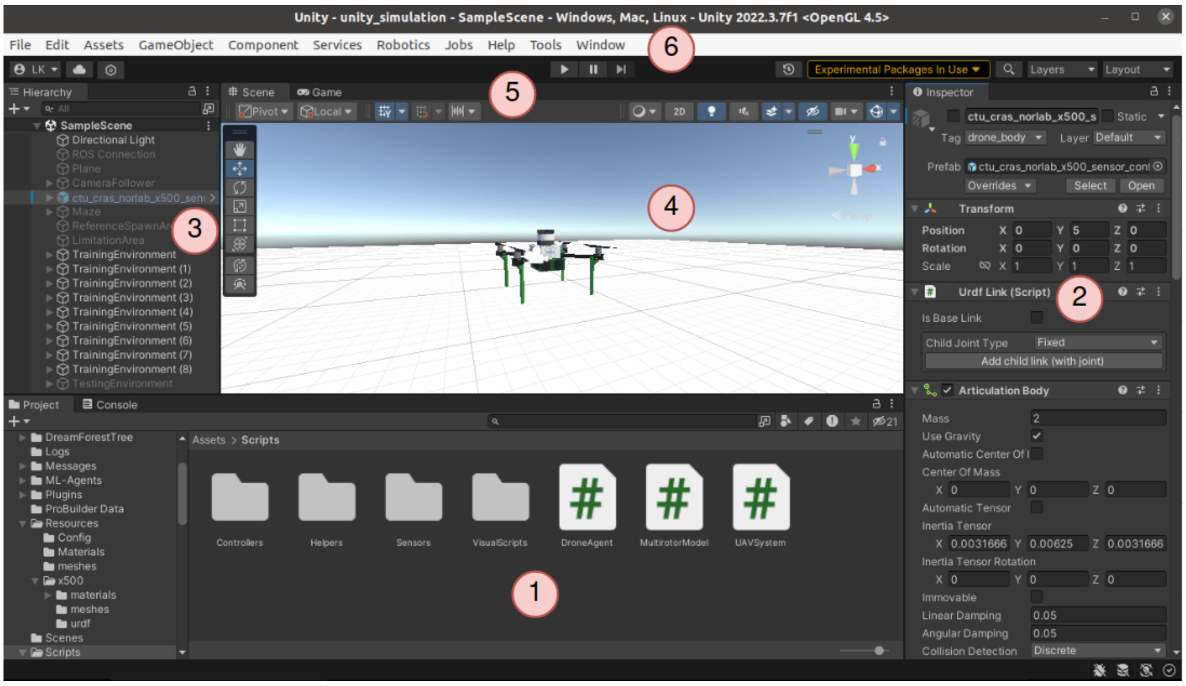

Unity - unity_simulation - SampleScene - Windows, Mac, Linux - Unity 2022.3.7f1 <OpenGL 4.5>
File Edit Assets GameObject Component Services Robotics Jobs Help Tools Window
Hierarchy
Scene
Game
Inspector
Project
Console
1
2
3
4
5
6

Figure 2.1: Unity Editor UI components

setup, graphics, and rendering settings. Part of Unity Editor is the package manager taking care of downloading the assets.

Figure 2.1 depicts the main UI components of the Unity Editor:

1. Asset Menu: C# scripts, materials, and shaders are located in this menu and can be directly assigned to objects in the scene.
2. Inspector: view for setting the values for variables, monitoring and adjusting the transformations, and adding new components.
3. Hierarchy window: represents the current hierarchy of game objects in the Unity scene.
4. Scene window: a visual representation of the current scene.
5. View bar: allows to switch between the Scene and Game views (the image rendered from cameras during the game) and setting the visual representation of some auxiliary elements (camera angle view, axes).
6. Toolbar: the menu containing project settings, exports, and imports, documentation reference, access to the package manager, as well as some custom asset dropdowns.

2.1.4 Unity Terminology

It is important to define the terminology used across the thesis that is specific to Unity3D and this framework:

- **Scene**: an asset that holds and organizes all the elements of a particular level, environment, or section of a game or simulation. It serves as a way to structure and manage the various components, such as game objects, lights, cameras, and more, that make up a specific part of the game world. A project can contain multiple scenes.
- **GameObject**: a fundamental entity that serves as a container for components, defining its properties, behaviors, and interactions within the game or simulation. The game

objects are organized in a hierarchy; a game object containing multiple other instances is called a parent, and the corresponding underlying game objects are children. A single GameObject can hold a RigidBody (or alternative) for physics simulation, collision meshes, and numerous user-defined scripts. In a script, the position and rotation of a GameObject can be accessed through the Transform class.

- **MonoBehaviour**: the base class for every script component of a GameObject. It should be implemented to achieve custom functionality during the life cycle by overriding predefined methods.
- **Transform**: a class that is automatically assigned to every GameObject. In addition to the position vector and rotation in the form of quaternions, it contains scaling parameters in each axis direction. To extract position and rotation in a script:

  ```
  Transform currentTransform = GetComponent<Transform>();
  Quaternion currentRotation = currentTransform.rotation;
  Vector3 currentPosition = currentTransform.position;
  ```

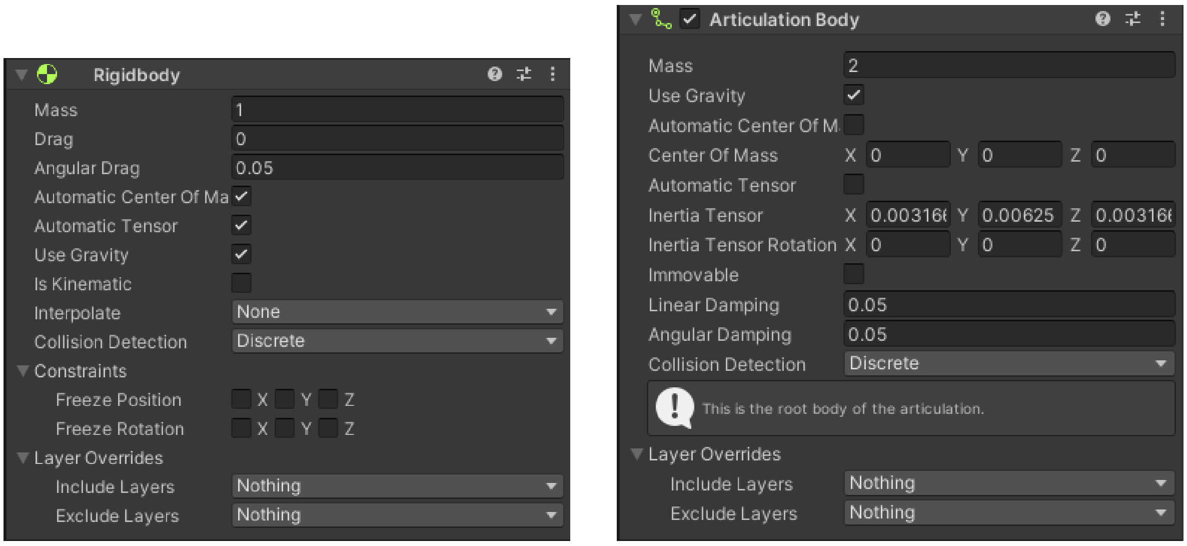
- **RigidBody**: the cornerstone component for physics simulation that represents the physics properties of an object in the simulation. The parameters that can be set up for a RigidBody are mass, drag, and angular drag. Gravity can be turned on and off for it. It can be configured in the Unity Editor and programmatically through a C# script. Similarly, the RigidBody class has methods that can be used to extract current linear and angular velocity (but not acceleration).
- **ArticulationBody**: an alternative to a RigidBody, but more common for robotics simulations. Provides simulation efficiency, for example, in the case of multi-joint manipulators or robots with complex structures. ArticulationBody has more parameters for setting the type of the joint and its characteristics. Here's the piece of code that is used to get a body's velocity and angular velocity.

  ```
  ArticulationBody articulationBody = GetComponent<ArticulationBody>();
  Vector3 angularVelocity = articulationBody.angularVelocity;
  Vector3 velocity = articulationBody.velocity;
  ```

- **Collider**: a base class for all collider types: BoxCollider, SphereCollider, CapsuleCollider, MeshCollider. It is used in the project to check the collisions and *triggers*. In *trigger* mode, the collider does not physically interact with the other rigid bodies. Instead, it sends events representing whether the collider detected some overlap with the other collider. MeshColliders, due to their complicated shape, are typically more expensive computation-wise than primitive colliders.
- **Update time**: period of calling the Update method of a standard MonoBehaviour Unity script.
- **FixedUpdate time**: similarly, time of calling the FixedUpdate of a standard MonoBehaviour Unity script. The difference between these times is discussed in detail in subsection 2.1.5.

In this thesis, a drone model instance in a simulation is represented by a GameObject with an ArticulationBody attached. The choice of ArticulationBody over RigidBody is given as it is the default physical component added by URDF converter[3].

[3]URDF convertor's benefit is that it can add a robotic manipulator as a whole and set the correct joint types.



Rigidbody

Mass	1
Drag	0
Angular Drag	0.05
Automatic Center Of Ma	✓
Automatic Tensor	✓
Use Gravity	✓
Is Kinematic	
Interpolate	None
Collision Detection	Discrete
Constraints	
Freeze Position	X Y Z
Freeze Rotation	X Y Z
Layer Overrides	
Include Layers	Nothing
Exclude Layers	Nothing

✓ Articulation Body

Mass	2
Use Gravity	✓
Automatic Center Of M	
Center Of Mass	X 0 Y 0 Z 0
Automatic Tensor	
Inertia Tensor	X 0.00316 Y 0.00625 Z 0.00316
Inertia Tensor Rotation	X 0 Y 0 Z 0
Immovable	
Linear Damping	0.05
Angular Damping	0.05
Collision Detection	Discrete
Layer Overrides	
Include Layers	Nothing
Exclude Layers	Nothing

This is the root body of the articulation.

(a) RigidBody in Unity Editor. (b) ArticulationBody in Unity Editor

Figure 2.2: Comparison of the available parameters for physical components

2.1.5 Simulation Technical Aspects

In this subsection, we present the technical aspects and important nuances encountered during the implementation of the project.

Left-hand coordinate system.

One of the things that is not fully compatible with the ROS notation is the internal left-handed coordinate system depicted in Figure 2.3. To overcome this issue, the ROS connector described in *Section 3.1* offers the methods to convert from Unity's coordinate system[4] to ROS system and back. The conversion is straightforward but cannot be overlooked.

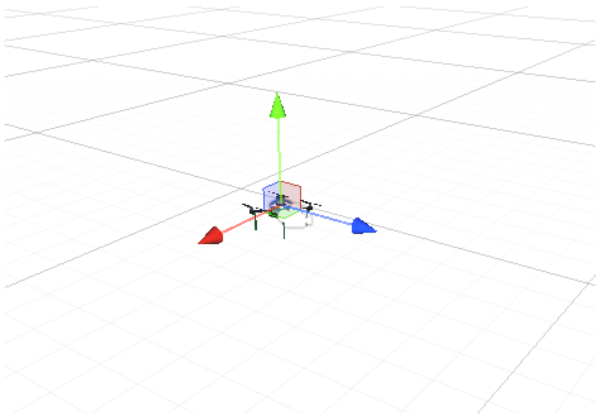


Figure 2.3: Illustration of the RUF Unity native coordinate system axes, where the red x-axis is directed to the right, the green y-axis is directed up, and the blue z-axis is directed forward.

[4]Left-handed system is called RUF (right-up-forward) and the standard to robotics right-handed system is called FLU (forward-left-up).

The conversion is different depending on the nature of the target values:

- Conversion of position and linear velocity vectors. Suppose that the position of the drone in FLU is represented by vector $\mathbf{r}_{FLU}$, or:

$$\mathbf{r}_{FLU} = (r_x, r_y, r_z), \tag{2.1}$$

  then the same position in the RUF coordinate system is:

$$\mathbf{r}_{RUF} = (-r_y, r_z, r_x). \tag{2.2}$$

- Conversion of angular velocity and torques. The angular velocity of an object is represented by vector $\boldsymbol{\omega}_{FLU}$.

$$\boldsymbol{\omega}_{FLU} = (\omega_x, \omega_y, \omega_z). \tag{2.3}$$

  To get the $\boldsymbol{\omega}_{RUF}$ we transform it:

$$\boldsymbol{\omega}_{RUF} = (\omega_y, -\omega_z, -\omega_x). \tag{2.4}$$

Note that to convert the vectors back, an inverse operation is performed.

The general suggestion when working with the framework presented in this thesis is to keep the state of the objects logically separated by methods and naming conventions to outline their corresponding coordinate system. The implementation of the drone's control system (see *Chapter 4*) has its internal coordinate system, and the coordinates are being transformed only on the inputs and outputs of the control blocks. The separation shown in Figure 2.4 prevents undefined coordinate states and undesired behavior.

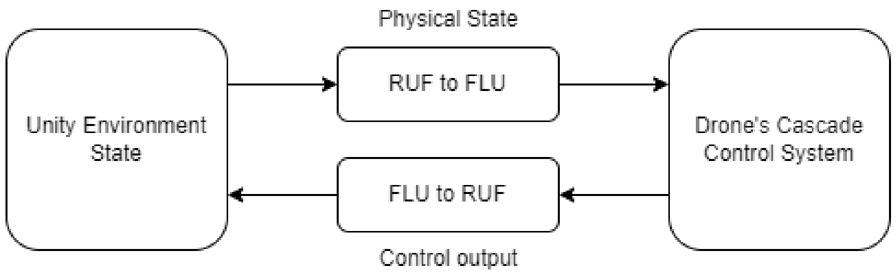


Figure 2.4: Example of conversion between Unity and ROS coordinate systems for a drone control system.

Simulation update rate.

Another critical simulation aspect is asynchronous *Update* and *FixedUpdate* rates. The Update method is invoked every frame update; therefore, it usually takes care of the animations and rendering. The rate is inconsistent and depends on the FPS (Frames per Second) the hardware platform achieves. On the contrary, FixedUpdate always has the identical period of invocation. This corresponds to the need for stable physics computation, which should be computed despite low FPS. Therefore, every physics-related calculation and assigning continuous rewards for RL training takes place there. The fixed time step can be set up in the project settings.

Coordinate systems transformation.

The change in the physical state of a rigid object, e.g., a drone's position, is simulated by applying corresponding forces and torques. They are defined by vectors of 3 values:

$$\mathbf{f} = (f_x, f_y, f_z), \tag{2.5}$$

where each of the elements of the vector represents the force applied along one of the global axes. However, the force is usually calculated in the coordinate system of an object itself. Conversion is applied by multiplication with a corresponding transformation matrix of $\mathbf{T}_W^B$:

$$\mathbf{f}_W = \mathbf{T}_W^B \cdot \mathbf{f}_B. \tag{2.6}$$

In Unity, *RigidObject* class has pre-defined methods for the application of force vectors in both global and local coordinate systems, *AddForce()* and *AddRelativeForce()* accordingly.

Similarly, to apply torques to the body in global and local coordinate systems, *AddTorque()* and *AddRelativeTorque()* can be utilized.

Another important conversion should happen in case the model of dynamics of a body in Unity uses the angular velocity getter method and expects the angular velocities in the local frame. Wrong conversion can cause the model's undesired behavior that is observed when its rotation along the y-axis (RUF) exceeds 180 degrees.

```
Vector3 globalCoordAngularVelocity = GetComponent<ArticulationBody>().angularVelocity;
Vector3 localCoordAngularVelocity =
    transform.InverseTransformDirection(globalCoordAngularVelocity);
```

Colliders

Collider components and their combination represent the shape of a body used for collision detection during the simulation. There are three main types of colliders supported in Unity:

- **Primitive Colliders**: These represent fundamental shapes such as Box, Circle, and Sphere.
- **Compound Colliders**: Formed by the union of various Primitive Colliders, these colliders provide more complex collision shapes.
- **Mesh Colliders**: This type utilizes a mesh similar in shape to the visual mesh to create highly accurate collision meshes.

However, it's important to note that Mesh Colliders have a limitation: they cannot collide with other Mesh Colliders. To address this limitation, a convex hull of the collision mesh must be used to enable collisions with other Mesh Colliders. In Unity, these collision meshes are restricted to 255 triangles. This limitation can result in sub-optimal performance during simulation, as the volume of these collision meshes often exceeds that of the visual meshes, leading to erratic behavior in the Articulation Body. To detect a collision with an object, special tags could be assigned to classify collision types. By interpreting the tags, the environment can react accordingly. In this thesis, the colliders are used for triggering the end of a training episode presented in Section 6.2.2.

2.1.6 Unity Assets

Unity has an extensive community that contributes and shares its work, which is realized through the Unity Assets Store. The assets include software packages, scenes, animations, GUI elements, and 3D models. Some of the assets are not free to encourage the creators. In the thesis, multiple assets were tested to create worlds for the UAVs to fly in. The forest environment depicted in Figure 2.5 is suitable for odometry-aware flights as it contains a large number of detectable features. The mine environment, such as the one displayed in Figure 2.7, is suitable for precise control and navigation tasks. The scene contains multiple pre-defined game objects that can be placed and adjusted. Lightning assets also play a significant role in simulating camera output in dark areas. For more specific applications, a user can choose to use numerous available assets, such as an industrial scene in Figure 2.6.

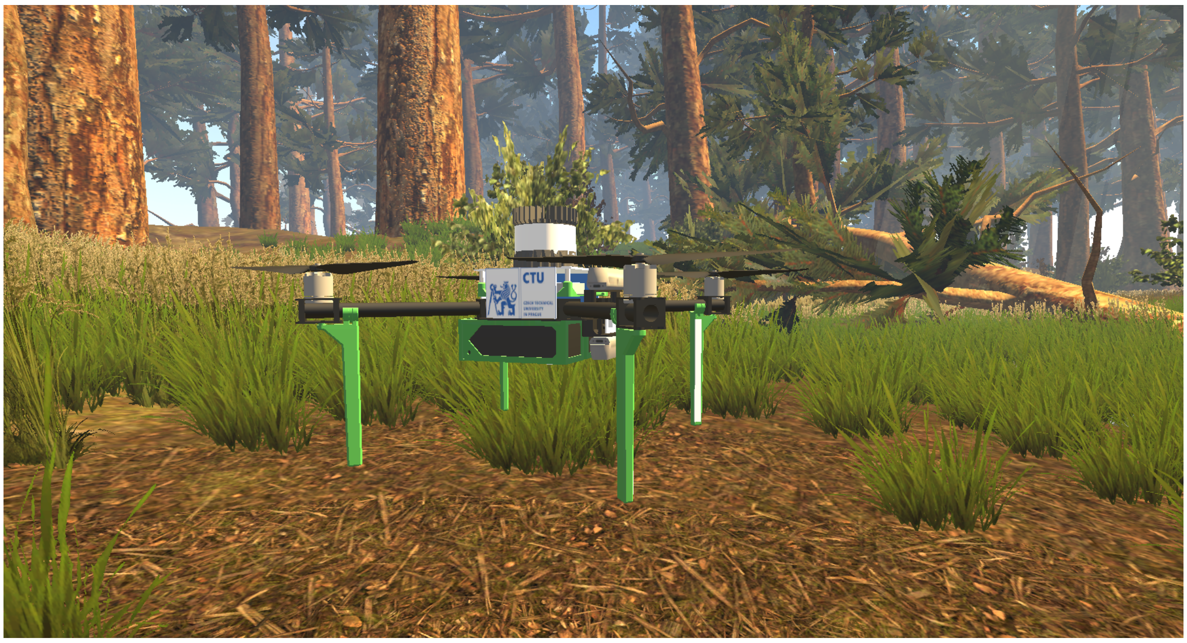


Figure 2.5: Forest testing environment with drone model using free asset [14].

Depending on the application, a user can choose to either load a pre-defined scene from the Asset store or build a simple scene using easy-to-use tools like ProBuilder [9]. It is a tool that allows the creation and design of polygon shapes that can be incorporated into terrain features.

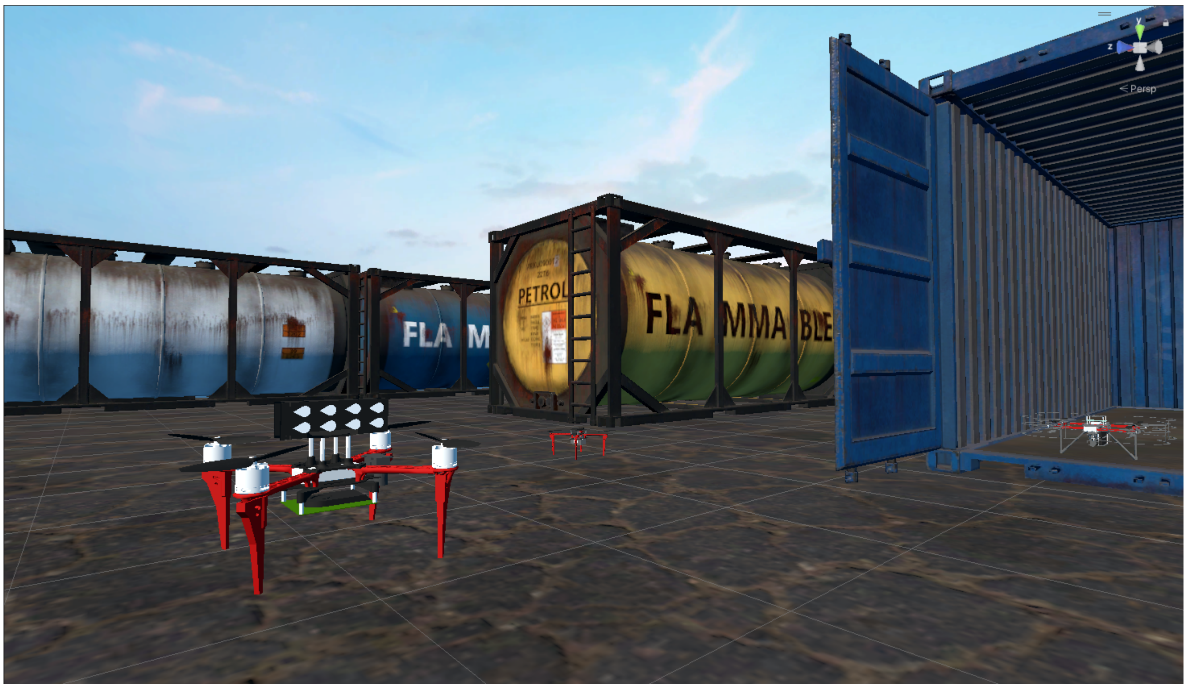


Figure 2.6: Drones placed in an industrial scene environment.

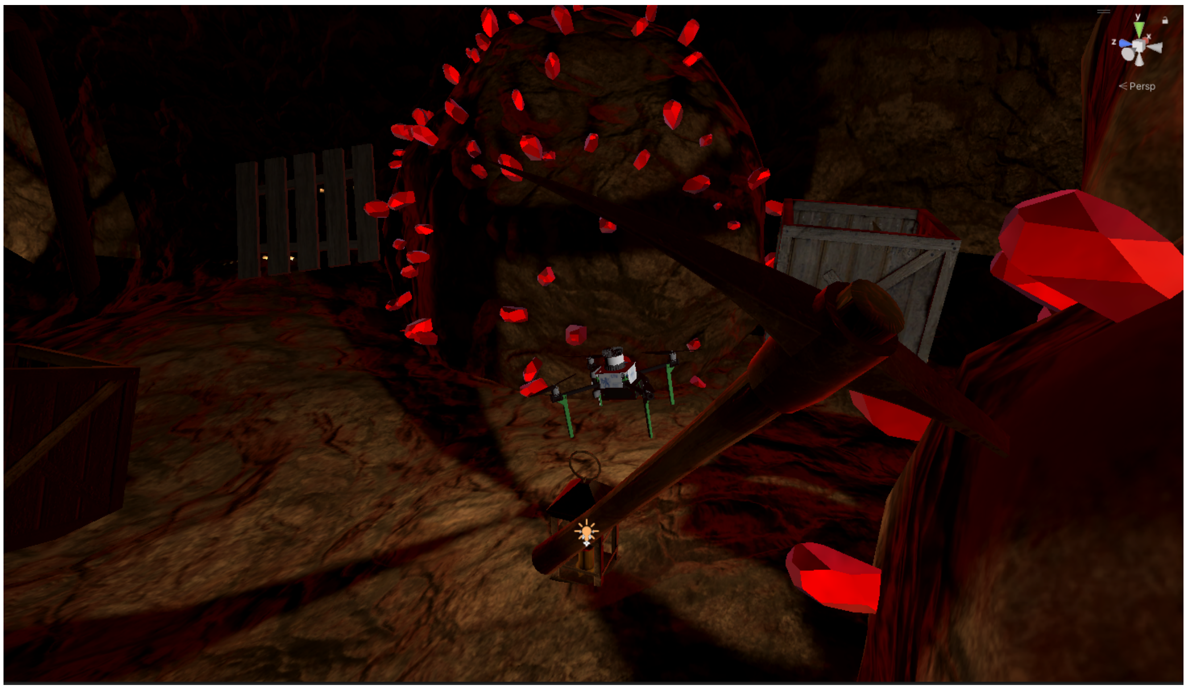


Figure 2.7: X500 drone flying in a mine environment.

2.2 Drone Models

This chapter discusses the import of robotic models, specifically quadrotors, into Unity 3D. There is no standard way to represent a robot model for a simulation in robotics, and the format is simulator-dependent. Usually, they are represented by a text file, but its structure is different depending on the simulation environment. Therefore, it is vital to establish the rules for the import of new models into the simulation world. The format for adding a new robot to the Unity scene is Unified Robotics Description Format (URDF), an XML specification used to represent a robot consisting of multiple parts. However, the thesis utilizes the models provided by the MRS UAV system [13] that should be first converted to URDF.

2.2.1 Model Files Conversion

MRS simulation in Gazebo tends to work with the drone models in Structured Data File (SDF) format. The file can define the characteristics of the robot, store the setup for the world environment, and manage the plugin insertion. To convert the SDF drone model into URDF, a development branch of the SDFormat library [23] was used. Note that the latest release of the library does not contain this functionality. The library's branch should be built from the source. Before the conversion, we have to adjust the SDF file by changing the prefixes of the paths inside the file from the prefix *model://* to *package://*. The following command can be utilized for that:

```
sed -i 's@model://@package://@g' model.sdf
```

All the prefixes inside the *model.sdf* were changed in place. To convert the file format from SDF to URDF, we use the terminal command:

```
ign sdf -u model.sdf
```

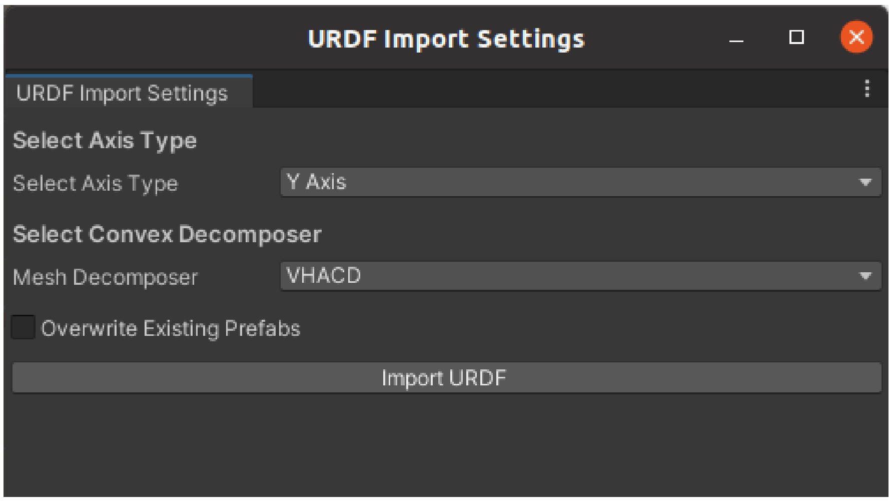
The *model.sdf* should be substituted with the correct path to the SDF model. The name of the newly produced file is *model.urdf*.

2.2.2 Model Import into Unity

The conversion is achieved; however, the files are not ready for import yet. The produced file contains the paths for uploading the meshes of the modular parts of the robot. In the case of MRS drone models, the paths are relative to the corresponding MRS simulation project setup. The files are placed into a Resources folder under Unity's Assets. The meshes are also moved to the Unity Assets folder, respecting the original file hierarchy. After this step, the URDF file is ready for import.

Unity Robotics Hub, mentioned in Section 1.2, provides an interface called *URDF Converter* that supports adding the model into a scene and translating the URDF-specific parameters to Unity syntax. It is distributed as an Asset via the Package manager (recommended installation from GitHub source) and adds a new option into the toolbar (see Subsection 2.1.3).

Upon the selection of the *Import Robot from URDF* option from the dropdown menu, the UI gives a couple of options defining the import mode as Figure 2.8 outlines.



URDF Import Settings

URDF Import Settings

Select Axis Type

Select Axis Type: Y Axis

Select Convex Decomposer

Mesh Decomposer: VHACD

☐ Overwrite Existing Prefabs

Import URDF

Figure 2.8: URDF Importer menu for import customization

The creation of the collider for the model is addressed in the convex hull parameter. The overly complex mesh collider issue discussed in subsection 2.1.5 is addressed in *URDF Converter* by implementing the Volumetric Hierarchical Approximate Convex Decomposition (VHACD) algorithm for convex hull[5] creation suitable for dynamic meshes introduced by Daniel Thul et al. [31]. The URDF file can also be configured to ignore collisions between specific links composing the robot. Figure 2.9 shows that the Unity Mesh Collider represents the visual shape with less precision, resulting in additional colliders, whereas the VHACD algorithm mimics the shape of the drone landing gear.

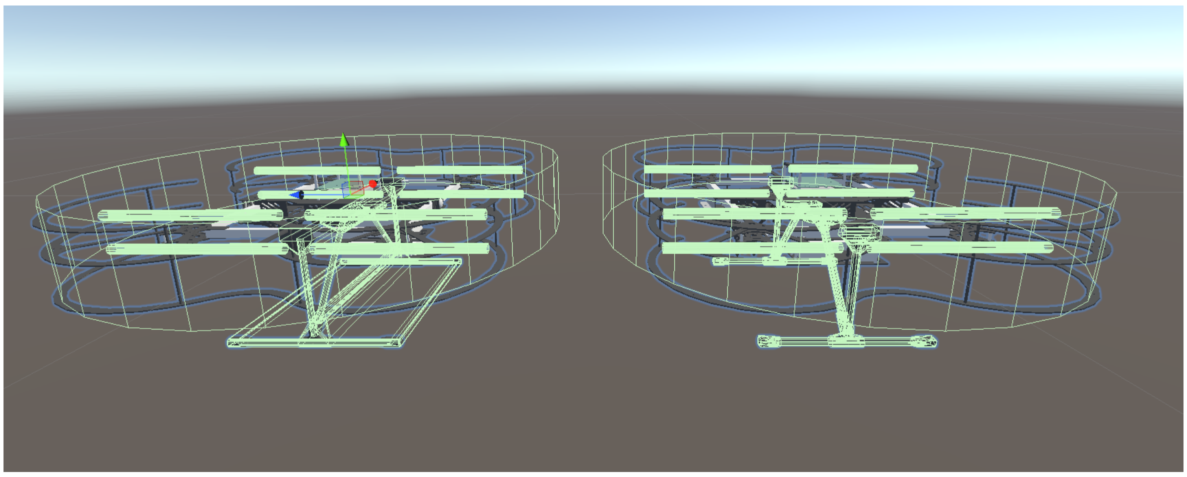


Figure 2.9: Comparison of Unity MeshCollider decomposition versus the VHACD for NAKI II model

The *URDF Converter* takes into consideration the materials and textures of the imported model and adds corresponding assets to the project. Drones are not the only robots supported by the tool. The Unity team initially tested the tool on the industrial robotic manipulators. Therefore, by default, it assigns an ArticulationBody, pictured in Figure 2.11, to

[5]A convex hull is the smallest convex shape that completely encloses a given set of points in space, ensuring that any line segment connecting two points within the shape lies entirely within the shape itself.

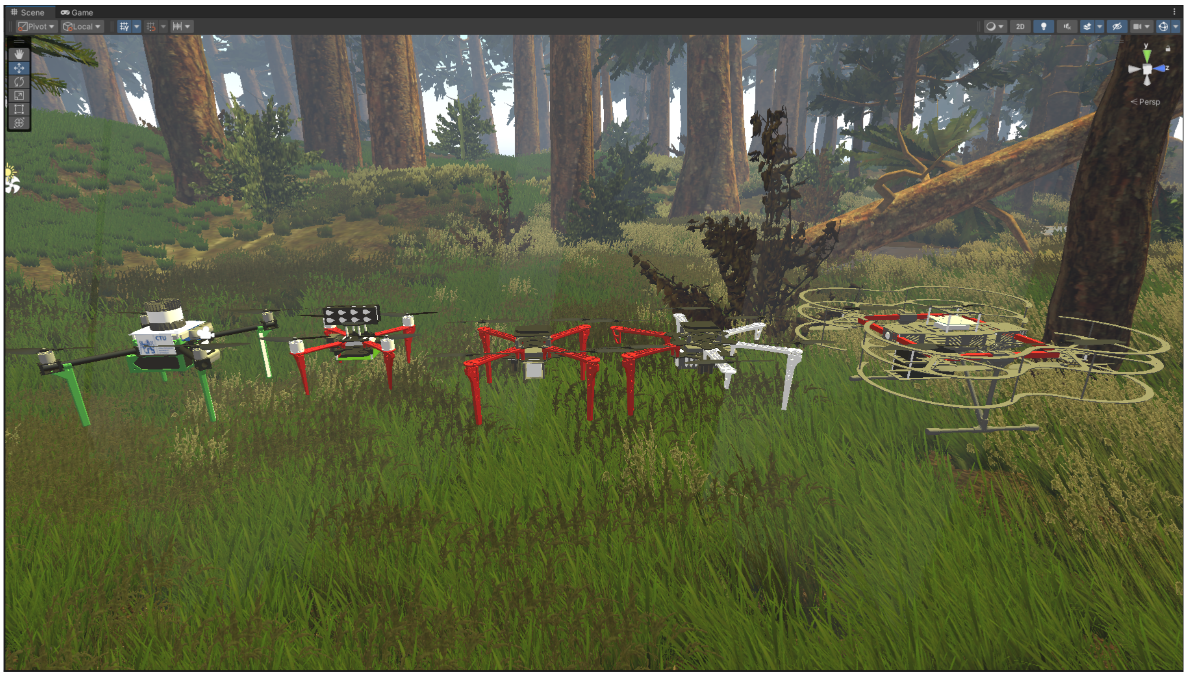
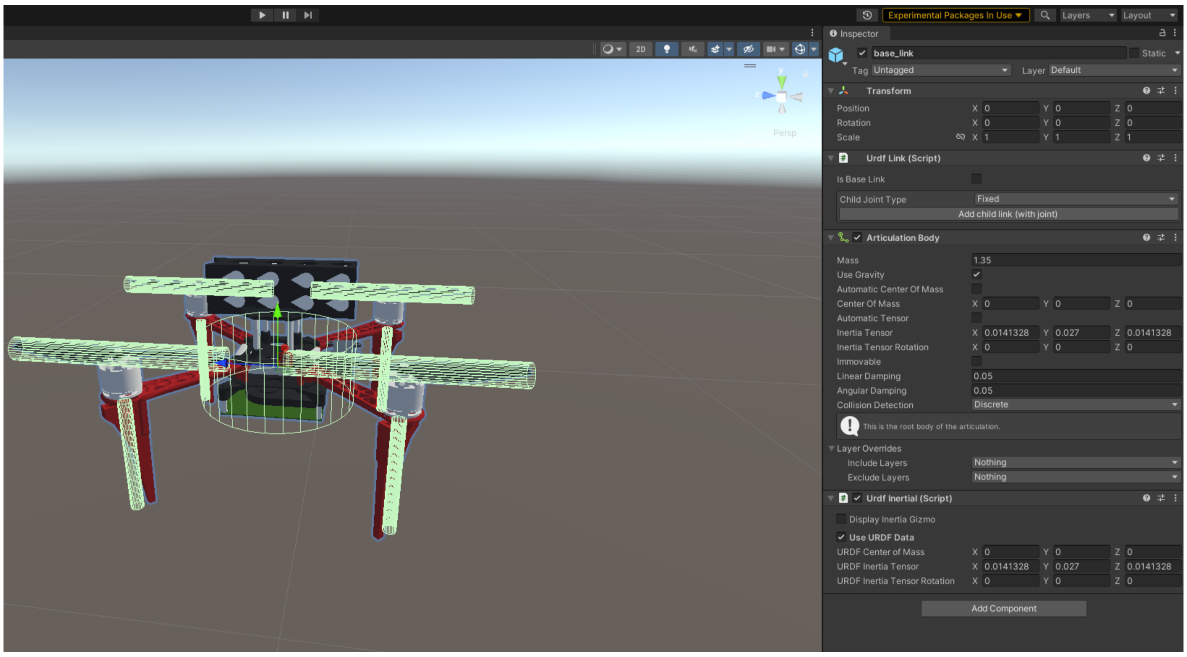


Figure 2.10: Drone models available in the framework, from left to right: **X500, F330, F450, F550** and **NAKI II**

simulate the physics of the model and fill the parameters defined in the URDF file. Note that in the case of a drone, the tool also creates the bodies for each rotor. Experiments with the multi-link drone model confirmed this causes a weird behavior in the simulation as the rotors start to rotate constantly. Therefore, in this project, we disable the ArticulationBodies for the rotors.



```
Inspector
✓ base_link                                   Static
Tag Untagged            Layer Default

Transform
Position                      X 0          Y 0       Z 0
Rotation                      X 0          Y 0       Z 0
Scale                         X 1          Y 1       Z 1

Urdf Link (Script)
Is Base Link                  ☐
Child Joint Type              Fixed
Add child link (with joint)

✓ Articulation Body
Mass                          1.35
Use Gravity                   ✓
Automatic Center Of Mass      ☐
Center Of Mass                X 0          Y 0       Z 0
Automatic Tensor              ☐
Inertia Tensor                X 0.0141328  Y 0.027   Z 0.0141328
Inertia Tensor Rotation       X 0          Y 0       Z 0
Immovable                     ☐
Linear Damping                0.05
Angular Damping               0.05
Collision Detection           Discrete
This is the root body of the articulation.
Layer Overrides
  Include Layers              Nothing
  Exclude Layers              Nothing

✓ Urdf Inertial (Script)
☐ Display Inertia Gizmo
✓ Use URDF Data
URDF Center of Mass           X 0          Y 0       Z 0
URDF Inertia Tensor           X 0.0141328  Y 0.027   Z 0.0141328
URDF Inertia Tensor Rotation  X 0          Y 0       Z 0

Add Component
```

Figure 2.11: Inspector view on the newly added drone model.

The tool also adds auxiliary components to the GameObject in a hierarchical manner. However, they are more specific to the robotic manipulators and thus are not used in the drone context.

Independently from the model obtained by the *URDF Importer*, to store the custom parameters for the drone models, such as the motor constants or articulation matrix used in the calculation of the control signal, the project keeps JSON configuration files in the Resources folder. The parameters are loaded dynamically through the C# script to avoid hardcoding. To prevent undesired behavior, these parameters should always be aligned with the URDF file before its import into the scene or directly through the Inspector tab in the Unity Editor after the import.

Chapter 3

Connecting ROS and Unity

This chapter describes the way the connection between Unity and ROS works and how to set up communication between ROS nodes and Unity Scripts. The chapter presents the reasoning for the choice of the connector and gives a short overview of the alternative ways of communication.

Currently, there are multiple available options for communication between ROS and Unity. The most widely used are ROS#, ROS-TCP-Connector, and ROS.NET. The extensive overview and benchmarks were recently presented by J. Allspaw, G. LeMasurier, and H. Yanco [2]. The performance evaluation showed that the more recent ROS TCP Connector can transfer small to medium-sized messages with less overhead than other implementations in most scenarios. The reason is that sending messages requires a serialization step, which is implemented in ROS TCP Connector more efficiently than its alternatives. However, the potential bottleneck of the connector is its dependency on routing a single routing node. The performance of this design choice is visible in the instances of larger size, where ROS.NET outperforms ROS TCP Connection because it connects to each actively communicating ROS node directly. For the framework presented in this thesis, the ROS TCP Connection implementation was chosen due to its continuous ongoing maintenance by the Unity team as part of their Unity Robotics Hub project. However, we have to note that multiple publications ([10], [2]) stress that the TCP connector is still the main bottleneck of the whole ROS-Unity simulation pipeline.

3.1 ROS TCP Connector

The cornerstone of connection between a Unity scene and ROS infrastructure is a so-called ROS TCP connector [32]. The package can be installed from a Git repository or the Unity package manager. The TCP Endpoint runs as a ROS node on a local URL specified in a launch file (default is 127.0.0.1) and manages the transfer of the data packages. As mentioned earlier, each message undergoes a serialization step before the actual sending. The package supports serialization of multiple types of standard ROS messages; here are several examples[1]:

- Standard messages: floats, integers, byte arrays;
- Geometry messages: vectors, rotation and translation matrices, quaternions;
- Sensor messages: images, IMU data, battery state;
- Machine Learning (ML) messages: facilitating object recognition;
- Message format: allowing the sending of octomaps.

For more complex applications, the connector must handle the transfer of custom message types. Therefore, as a part of the thesis, the feature of creating a custom message type was tested in Section 3.2.

[1]More message types can be found in ROS TCP Connector GitHub page [32]

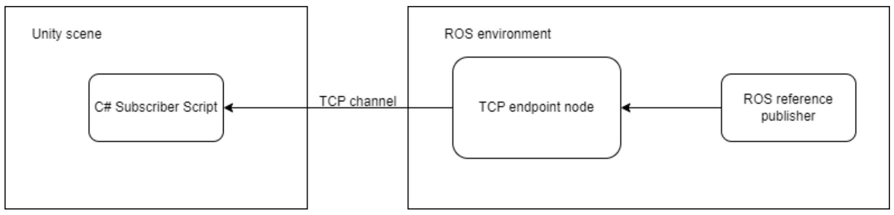


Figure 3.1: Connecting publisher node to ROS using TCP connector

3.2 Creating a subscriber in Unity

This section describes the process of adding a new subscriber to a ROS topic inside Unity. The process is presented by creating a custom message for manual drone control by sending references from the PS4 controller through the ROS publisher. The pipeline was set up in this way to provide a communication gateway between the reference definition on the ROS side and the control system simulation on the Unity side.

The PS4 controller publishing node is implemented in Python. The message is defined in the following way in a file with *.msg* extension. It consists of a vector representing the current reference interpreted on the Unity side according to the corresponding control input modality chosen in advance.

To ensure the correct message parsing, a C# message class representing the custom type is generated using a message-generation tool, which is a part of the ROS-TCP-Connector. The same message file used for the ROS publisher node definition should be imported as input for the importer.

The final step for the connection is to add a callback for the communication through a topic:

```
ROSConnection.GetOrCreateInstance()
    .Subscribe<VelocityRefMsg>("controller_command", SetVelocityRosCallback);
```

Here, the *VelocityRefMsg* is the name of the message being transferred through the topic *controller_command* and *SetVelocityRosCallback* is the name of a callback interpreting the values passed in the message. The callback sets the values for the internal variables.

On the ROS side, the required topics and subscription to the joystick controller are initialized by a *tmuxinator* configuration file and corresponding bash script.

It is crucial to note that this code should be part of a Unity script, specifically the main agent script. The PS4 Controller ROS publisher mode presented here is an example and can be exchanged for trajectory planning nodes publishing consecutive references.

3.3 Sensoric data publishers in Unity

In this section, we establish a critical use case involving the integration of a publisher on the Unity side. This communication setup entails the drone publishing data from its Inertial Measurement Unit (IMU) sensor and RGB camera image into specific ROS channels. Subsequently, this data is utilized for estimating the drone's odometry by applying the OpenVINS software, as discussed in Section 3.4.

The IMU sensor is critical to a drone's navigation and control system. The sensor provides information about the drone's orientation, acceleration, and angular velocity. The IMU sensor is emulated through a C# script, which is a part of the Unity Robotics Hub toolkit [11], designed to support robotics navigation applications. Besides data collection, the script facilitates necessary conversion steps between coordinate system notations. Furthermore, it allows for the introduction of Gaussian noise[2] or bias error. Another script, *IMUROSPublisher*, handles the publication of IMU data by collecting it during each *FixedUpdate* and transmitting it to ROS in a pre-defined *ImuMsg* format, as explained in Section 3.1. The ROS connection is established as follows:

```
ROSConnection.GetOrCreateInstance()
    .RegisterPublisher<ImuMsg>(topic_name);
```

It's worth noting that *topic_name* serves as a variable for the topic name within the script, and it can be set up via the Inspector tab in the Unity Editor.

Similar to the IMU sensor, the RGB camera image is extracted using the *RGBCamera* script and periodically published by the *RGBCameraROSPublisher* into specified topics. However, working with the camera image simulated in Unity involves several setup steps within the Editor. By default, the picture's resolution extracted from the simulation changes based on the current window size inside the Editor. To address this, we created a fixed-resolution camera with a hardcoded resolution of 800×600 pixels. The camera setup is saved as a preset, and the published image is compressed to enhance communication efficiency.

3.4 OpenVINS

OpenVINS (Visual-Inertial Navigation System) is an open-source software package for visual-inertial odometry and SLAM (Simultaneous Localization and Mapping). It is used in robotics and computer vision applications, particularly for estimating a vehicle's or camera's pose (position and orientation) in real-time. A fork of the original OpenVINS is included in the stack of the MRS UAV system. The functioning of OpenVINS hinges on the availability of IMU data and RGB images, which jointly contribute to establishing the drone's visual odometry. As defined in Section 3.3, dedicated publishers provide the necessary data. However, a crucial step lies in mapping this data correctly to serve as input for OpenVINS. The steps to establish a connection between the Unity environment and the OpenVINS odometry processing ROS node are:

- Assign the sensor data publishing scripts to the drone GameObject.
- Set up the required image quality of the RGB publisher and image publishing rate.
- Calibrate the position and rotation of the camera relative to the IMU coordinate system within the Unity simulation. This involves considering coordinate system conversions. Similarly, calibrate the position of the IMU sensor relative to the body frame of the UAV. On the OpenVINS side, the resulting translation and rotation parameters for the camera and IMU are incorporated into matrices specified in the configuration files.
- Set up the OpenVINS launch file and configuration files to subscribe to the topics with IMU data and decompressed RGB images. This step should be aligned with the sensor publishing scripts.

[2]Gaussian noise is a type of signal interference characterized by a probability density function aligned with the normal distribution, commonly known as the bell curve.

- Initialize the ROS-TCP Connector communication channel and run the simulation in Unity Editor.
- Execute a ROS node for republishing, which takes a compressed image and outputs the raw image in its original size:

```
rosrun image_transport republish compressed
    in:=<raw_image_topic> out:=<decompressed_image_topic>
```

- (Optional) The feature detection can be visualized using *RViz*[3]
- In case the feature extraction is struggling with stable feature detection, fine-tune parameters in the OpenVINS configuration files that impact odometry, including but not limited to FPS, noise parameters, etc. Often, the issue lies in incorrect camera position calibration.

All required topics and terminal commands are put together into a *tmuxinator* configuration and are run through a bash script.

[3]RViz serves as a 3D visualization software tool designed for robots, sensors, and algorithms, providing the capability to visualize the robot's perception of its environment, whether in a real-world scenario or within a simulated environment.

Chapter 4

UAV Control System

This chapter explains the part of the framework that implements the control system based on the MRS UAV system [13]. The control system has a hierarchical structure and can accept references on different levels.

Supported reference input modes are:

Mode	**Definition**	**Size (elem.)**
PositionCmd	*position* vector $\mathbf{r}_{ref}$, *heading angle* $\eta_{\mathbf{ref}}$	4
VelocityHdgCmd	*velocity* vector $\dot{\mathbf{r}}_{ref}$, *heading angle* $\eta_{\mathbf{ref}}$	4
VelocityHdgRateCmd	*velocity* vector $\dot{\mathbf{r}}_{ref}$, *heading rate* $\dot{\eta}_{\mathbf{ref}}$	4
AccelerationHdgCmd	*acceleration* vector $\ddot{\mathbf{r}}_{ref}$, *heading angle* $\eta_{\mathbf{ref}}$	4
AccelerationHdgRateCmd	*acceleration* vector $\ddot{\mathbf{r}}_{ref}$, *heading rate* $\dot{\eta}_{\mathbf{ref}}$	4
TiltHdgRateCmd	tilts vector $\mathbf{t}$, *heading rate* $\dot{\eta}_{\mathbf{ref}}$, *throttle* F_{tref}	5
AttitudeCmd	orientation matrix $\mathbf{R}_d$, *throttle* F_{tref}	10
AttitudeRateCmd	*roll*, *pitch*, *yaw* rates, *throttle* F_{tref}	4
ControlGroupCmd	*roll*, *pitch*, *yaw* torques, *throttle* F_{tref}	4
ActuatorCmd	*motor rpm* vector $\boldsymbol{\omega}$	$NMotors$

Table 4.1: The hardware characteristics device used for conducting the experiments presented in the thesis. The number of the motors depends on the type of the UAV.

The workflow is that the references located higher in the hierarchy are step-by-step converted to the lower hierarchy references until they reach the *ActuatorCmd*. Each level is processed by a corresponding controller; therefore, in the thesis, the controllers presented are *position*, *velocity*, *acceleration*, *attitude*, *attitude rates* controllers and the lowest level, which is called a *mixer*. For the purpose of compatibility, we provide the interface accepting the *ActuatorCmd* mode. However, the physics simulation of the control system's output is realized by applying the corresponding forces and torques to the ArticulationBody. More details about the application of the output are discussed in Section 4.2

A *reference dispatcher* addresses the versatility of reference modes. It adjusts the control to accept references on a specified level.

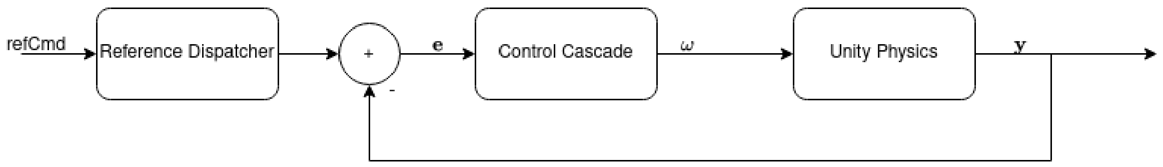


Figure 4.1: Abstract model of the reference passing to the control cascade.

Note that the diagram 4.1 depicts abstract reference *refCmd*, error $\mathbf{e}$ and output $\mathbf{y}$;

however, the output of the control cascade is always the motor's angular velocity represented in rotations per minute (RPM).

4.1 Cascade Control System

This section describes the internal view inside the control cascade on each level. It outlines the step-by-step propagation of the reference. For the purpose of demonstration of the calculation, we assume a quadrotor with four motors.

4.1.1 PID controller

Multiple controllers are using PID controllers internally. Therefore, the project implementation provides an abstract PID class that incorporates the proportional, derivative, and integral feedback, saturation limits, and anti-windup filters. To calculate the derivative, the controller keeps track of the error from the previous update and performs the difference between it and the current error. A cumulative sum of the error represents integral feedback. The PID block accepts the update period as an input argument, making it adaptable to increased simulation speed. The controller works with a scalar error; therefore, we initialize an instance of this class for each element of the error vector. Lastly, the constants for the PID controller can be initialized with default values, but usually, the values are passed during the initialization of the class.

4.1.2 Position Controller

Position control takes reference position $\mathbf{r}_{ref}$, current position $\mathbf{r}$ and calculates a simple error:

$$\mathbf{r}_{err} = \mathbf{r}_{ref} - \mathbf{r}. \tag{4.1}$$

Each element of the vector $\mathbf{r}_{err}$ is passed through the corresponding PID controller, and the output control signal is passed as a *VelocityHdgCmd* type reference to the following velocity controller.

4.1.3 Velocity Controller

Velocity controllers can work in two modes:

- **Input**: *VelocityHdgRate*, **output**: *AccelerationHdgRate*;
- **Input**: *VelocityHdg*, **output**: *AccelerationHdg*;

In both cases, the controller calculates a similar error as in the case of the position controller:

$$\dot{\mathbf{r}}_{err} = \dot{\mathbf{r}}_{ref} - \dot{\mathbf{r}}, \tag{4.2}$$

and passes the reference through the PID controller. However, depending on the provided input reference, the controller outputs either the heading angle or the rate of its change.

4.1.4 Acceleration Controller

The calculation of the control signal of the acceleration controller varies depending on what type of input signal is passed.

AccelerationHdgRate mode

First, we calculate the desired force vector, i.e., the force the drone's acceleration controller aims to achieve. It is computed as the sum of the reference acceleration vector and a gravity vector $\mathbf{g}$ scaled by the drone's mass m:

$$\mathbf{f}_d = (\ddot{\mathbf{r}} + \begin{bmatrix} 0 \\ 0 \\ \mathrm{g} \end{bmatrix}) \cdot \mathrm{m}. \tag{4.3}$$

The desired force is then normalized:

$$\mathbf{f_{dnorm}} = \frac{\mathbf{f_d}}{\|\mathbf{f_d}\|_2}. \tag{4.4}$$

The output control signal is represented as an instance of the *TiltHdgRate* class. The tilt vector provides information about what inclination the drone should apply to achieve the desired force direction. AccelerationHdgRate doesn't change the input heading rate and passes it directly into the *TiltHdgRate* instance. The last part is a calculation of the normalized throttle. The designed method first calculates the thrust force by taking the dot product of the desired force vector $\mathbf{f}_d$ and the drone's upward direction, which is represented by the third column of the rotation matrix $\mathbf{R}$ in the drone's state.

$$F_t = \mathbf{f}_d \cdot \mathbf{R}_{\text{column}(3)}. \tag{4.5}$$

The thrust force F_t is then normalized and mapped to a normalized throttle value.

$$throttle = \frac{\sqrt{\frac{F_t}{K_f \cdot NMotors}} - \omega_{max}}{\omega_{max} - \omega_{min}}, \tag{4.6}$$

where

- K_f is a linear coefficient of the motor
- ω_{max}, ω_{min} are the maximum and minimum rotations per minute the motor can achieve
- $NMotors$ represents the number of motors the drone has

Finally, the tilt vector, heading rate, and throttle are passed to the attitude controller.

AccelerationHdg mode

In this mode, the goal is to construct a rotation matrix $\mathbf{R}_{ref} \in SE(3)$ and throttle from the *AccelerationHdg* to provide an *Attitude* reference. The calculation follows the same steps presented in equations (4.3) and (4.4) to obtain the desired force vector. The next step is to calculate the reference direction in the XY plane based on the desired heading η:

$$\mathbf{bx}_d = \begin{bmatrix} \cos(\eta) \\ \sin(\eta) \\ 0 \end{bmatrix}. \tag{4.7}$$

An oblique projector is constructed to project vectors onto the plane orthogonal to the desired force vector $\mathbf{f}_{dnorm}$. This is achieved through the following matrix operations.

$$\mathbf{P}_{zcomp} = \mathbf{I} - \mathbf{f}_{dnorm}\mathbf{f}_{dnorm}^T. \tag{4.8}$$

The equation (4.8) obtains a projection matrix that projects vectors onto the orthogonal subspace (complementary) to the direction of the $\mathbf{f}_{dnorm}$ vector. We calculate matrices $\mathbf{A}$ and $\mathbf{B}$:

$$\mathbf{A} = \begin{bmatrix}\mathbf{P}_{zcomp1} & \mathbf{P}_{zcomp2}\end{bmatrix}, \tag{4.9}$$

$$\mathbf{B} = \begin{bmatrix}\hat{\mathbf{e}}_1 & \hat{\mathbf{e}}_2\end{bmatrix}, \tag{4.10}$$

and then we multiply the $\mathbf{B}^T$ with $\mathbf{A}$ and obtain the pseudoinverse matrix:

$$\mathbf{L}^+ = ((\mathbf{B}^T\mathbf{A})^T\mathbf{A})^{-1}(\mathbf{B}^T\mathbf{A})^T. \tag{4.11}$$

The oblique projector is the following:

$$\mathbf{P}_{oblique} = \mathbf{A}\mathbf{L}^+\mathbf{B}^T. \tag{4.12}$$

Finally, we use the projector to construct the orientation matrix R_{ref}:

$$\mathbf{R}_{ref} = \begin{bmatrix}\frac{\mathbf{P}_{oblique}\mathbf{bx}_d}{\|\mathbf{P}_{oblique}\mathbf{bx}_d\|_2} & \mathbf{f}_{dnorm} \times \mathbf{P}_{oblique}\mathbf{bx}_d & \mathbf{f}_{dnorm}\end{bmatrix}. \tag{4.13}$$

The throttle is calculated according to the equations (4.5) and (4.6). The orientation matrix $\mathbf{R}_{ref}$ and *throttle* are sent as *Attitude* reference to the attitude controller.

4.1.5 Attitude Controller

Attitude controller accepts *Attitude* and *TiltHdgRate* input references. The workflow for each reference is described in the following subsections.

Attitude mode

In this mode, we calculate the orientation error $\mathbf{R}_{err}$ between the reference $\mathbf{R}_{ref}$ and the orientation $\mathbf{R}$ of the current state:

$$\mathbf{R}_{err} = \frac{1}{2}(\mathbf{R}_{ref}^T\mathbf{R} - \mathbf{R}_{ref}\mathbf{R}^T). \tag{4.14}$$

This approach to error calculation was introduced in [43] and has a constraint for relatively minor deviations between the reference and current orientations. From the error matrix, only relevant values are extracted into error vector $\mathbf{r}_{err}$:

$$\mathbf{r}_{err} = \begin{bmatrix}\frac{1}{2}(\mathbf{R}_{err(1,2)} - \mathbf{R}_{err(2,1)}) \\ \frac{1}{2}(\mathbf{R}_{err(2,0)} - \mathbf{R}_{err(0,2)}) \\ \frac{1}{2}(\mathbf{R}_{err(0,1)} - \mathbf{R}_{err(1,0)})\end{bmatrix}. \tag{4.15}$$

Consequently, each value of the vector $\mathbf{r}_{err}$ is passed through the PID controller, and the control signal $\boldsymbol{\omega}_d$ is passed into an *AttitudeRate* reference in combination with the throttle coming directly from the *Attitude* reference.

TiltHdgRate mode

We start by forming the reference orientation matrix from the tilts vector $\mathbf{t}$:

$$\mathbf{t}_{norm} = \frac{\mathbf{t}}{\|\mathbf{t}\|_2}. \tag{4.16}$$

Then, we consequently calculate each column of the $\mathbf{R}_d$:

$$\mathbf{R}_{dcolumn(3)} = \mathbf{t}_{norm}, \tag{4.17}$$

$$\mathbf{R}_{\mathbf{d}column(2)} = \frac{\mathbf{R}_{\mathbf{d}column(3)} \times \mathbf{R}_{column(0)}}{\|\mathbf{R}_{\mathbf{d}column(3)} \times \mathbf{R}_{column(0)}\|_2}, \tag{4.18}$$

$$\mathbf{R}_{\mathbf{d}column(1)} = \frac{\mathbf{R}_{\mathbf{d}column(2)} \times \mathbf{R}_{\mathbf{d}column(3)}}{\|\mathbf{R}_{\mathbf{d}column(2)} \times \mathbf{R}_{\mathbf{d}column(3)}\|_2}. \tag{4.19}$$

After this step, the calculation follows the same approach to calculate the $\mathbf{R}_{err}$ by applying the equation (4.14), vectorizing the error as in equation (4.15) and passing it to the PID controller resulting in desired angular rates $\boldsymbol{\omega}_d$.

However, according to [13], the produced $\boldsymbol{\omega}_d$, in this case, falls into a phenomenon called parasitic heading rate that has to be compensated. To estimate it, we calculate the heading rate $\dot{\boldsymbol{\eta}}$ starting with forming the angular velocity tensor:

$$\boldsymbol{\omega}_{tensor} = \begin{bmatrix} 0 & -\boldsymbol{\omega}_3 & \boldsymbol{\omega}_2 \\ \boldsymbol{\omega}_3 & 0 & -\boldsymbol{\omega}_1 \\ -\boldsymbol{\omega}_2 & \boldsymbol{\omega}_1 & 0 \end{bmatrix}. \tag{4.20}$$

Then, we calculate the rotation matrix derivative by multiplying the current rotation $\mathbf{R}$ with the $\boldsymbol{\omega}_d$:

$$\dot{\mathbf{R}} = \mathbf{R} \cdot \boldsymbol{\omega}_d, \tag{4.21}$$

$$\dot{\boldsymbol{\eta}} = \frac{-\mathbf{R}_{(1,0)}}{\mathbf{R}^2_{(0,0)} + \mathbf{R}^2_{(1,0)}} \dot{\mathbf{R}}_{(0,0)} + \frac{\mathbf{R}_{(0,0)}}{\mathbf{R}^2_{(0,0)} + \mathbf{R}^2_{(1,0)}} \dot{\mathbf{R}}_{(1,0)}, \tag{4.22}$$

where the coefficients of $\dot{\mathbf{R}}_{(0,0)}$ and $\dot{\mathbf{R}}_{(1,0)}$ represent the *atan2* partial derivatives. The whole equation (4.22) represents the total differential. The method responsible for the calculation of the heading rate from intrinsic angular rates also ensures proper handling of potential numerical issues when the denominator is close to zero. Note that this computation is only valid for the small tilts that are typically less than 10°. The parasitic heading is then converted to yaw rate correction. To achieve that we perform multiple steps outlined below, starting with defining the heading vector $\boldsymbol{\eta}$ and orbital velocity $\boldsymbol{\nu}_{orbital}$:

$$\boldsymbol{\eta} = \begin{bmatrix} \mathbf{R}_{(0,0)} & \mathbf{R}_{(1,0)} & 0 \end{bmatrix}^T, \tag{4.23}$$

$$\boldsymbol{\nu}_{orbital} = \begin{bmatrix} 0 & 0 & \dot{\boldsymbol{\eta}} \end{bmatrix}^T \times \boldsymbol{\eta}. \tag{4.24}$$

The projector to the heading orbital velocity vector subspace is then:

$$\mathbf{P} = (\frac{\hat{\mathbf{e}}_3 \times \boldsymbol{\eta}}{\|\hat{\mathbf{e}}_3 \times \boldsymbol{\eta}\|_2})^T \frac{\hat{\mathbf{e}}_3 \times \boldsymbol{\eta}}{\|\hat{\mathbf{e}}_3 \times \boldsymbol{\eta}\|_2}. \tag{4.25}$$

The body yaw orbital velocity vector is projected onto the subspace spanned by the heading orbital velocity vector.

$$\mathbf{R}_{\mathbf{p}column(2)} = \mathbf{P} \cdot \mathbf{R}_{column(2)}. \tag{4.26}$$

The direction of the yaw rate is determined based on the dot product between the heading orbital velocity vector and the projected vector. The yaw rate magnitude is calculated by dividing the norm of the orbital velocity vector by the norm of the projected vector.

$$direction = sign(\boldsymbol{\nu}_{orbital} \cdot \mathbf{R}_{\mathbf{p}column(2)}). \tag{4.27}$$

$$\boldsymbol{\omega}_{3corr} = direction * (\frac{\|\boldsymbol{\nu}_{orbital}\|_2}{\|\mathbf{R}_{\mathbf{p}column(2)}\|_2}). \tag{4.28}$$

If the yaw rate $\boldsymbol{\omega}_{3corr}$ is finite, it is added as a correction to control signal from PID forming the final $\boldsymbol{\omega}_3$:

$$\boldsymbol{\omega}_3 = \boldsymbol{\omega}_{3pid} + \boldsymbol{\omega}_{3corr}. \tag{4.29}$$

The output control signal from the attitude controller working in *TiltHdgRate* input reference mode is again the *AttitudeRate* reference containing the angular velocities $\boldsymbol{\omega}$ and *throttle.*

4.1.6 Rate Controller

The allowed input the controller accepts is the *AttitudeRate* reference. The workflow is straightforward. First, the angular velocity error $\boldsymbol{\omega}_{err}$ is computed based on the difference between the reference $\boldsymbol{\omega}_{ref}$ and current state velocity $\boldsymbol{\omega}$:

$$\boldsymbol{\omega}_{err} = \boldsymbol{\omega}_{ref} - \boldsymbol{\omega}. \tag{4.30}$$

Each element is then passed to the corresponding PID controllers, and the *ControlGroupCmd* output reference is formed by passing the outputs of the PIDs combined with the unchanged *throttle* from the input *AttitudeRate* reference.

4.1.7 Mixer

In the mixer controller, the final motor output is calculated. It takes the input in the form of a *ControlGroupCmd* reference and multiplies it by the inversion of the allocation matrix $\boldsymbol{\Gamma}^{-1}$. The values are normalized in the range between 0 and 1.

4.2 Control Signal Output

To apply the control output (*motors rpm* vector), we conduct the force-torque allocation:

$$\begin{bmatrix} F_t \\ \tau_1 \\ \tau_2 \\ \tau_3 \end{bmatrix} = \mathbf{\Gamma} \begin{bmatrix} F_1 \\ F_2 \\ F_3 \\ F_4 \end{bmatrix}, \tag{4.31}$$

where F_1, F_2, F_3 and F_4 are the forces produced by each of the rotors. The result thrust force F_t is applied along the $\hat{\mathbf{b}}_3$ vector converted to the left-handed coordinate system using the *AddRelativeForce()* method of the ArticulationBody. The torques vector $\boldsymbol{\tau}$ is similarly applied (after the conversion) by executing the *AddRelativeTorque()* function. The allocation matrix $\mathbf{\Gamma}$ is pre-computed during the initialization of the control system:

$$\mathbf{\Gamma} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -\frac{d}{\sqrt{2}} & \frac{d}{\sqrt{2}} & \frac{d}{\sqrt{2}} & -\frac{d}{\sqrt{2}} \\ -\frac{d}{\sqrt{2}} & \frac{d}{\sqrt{2}} & -\frac{d}{\sqrt{2}} & \frac{d}{\sqrt{2}} \\ -c_{tf} & -c_{tf} & c_{tf} & c_{tf} \end{bmatrix}. \tag{4.32}$$

The inversion $\mathbf{\Gamma}^{-1}$ exists when then the linear force constant $c_{tf} \neq 0$ and $d \neq 0$. Here, c_{tf} represents a propeller's drag that changes according to the aerodynamic characteristics of a multirotor's propeller, and d is the length of the drone's arms.

Note that we can avoid this step and directly apply the ControlGroupCmd reference to the drone's ArticulationBody. Still, we include the mixer controller for the purpose of completeness of the cascade control system.

Figure 4.2 depicts the whole cascade system and its references. It shows the consequent processing of the references from the higher-level controllers to the low-level *ControlCmd*. The control system, however, can directly accept and process any of the outlined reference commands. For example, for the standard drone manual control with a joystick, the implementation allows control of the drone in the attitude mode.

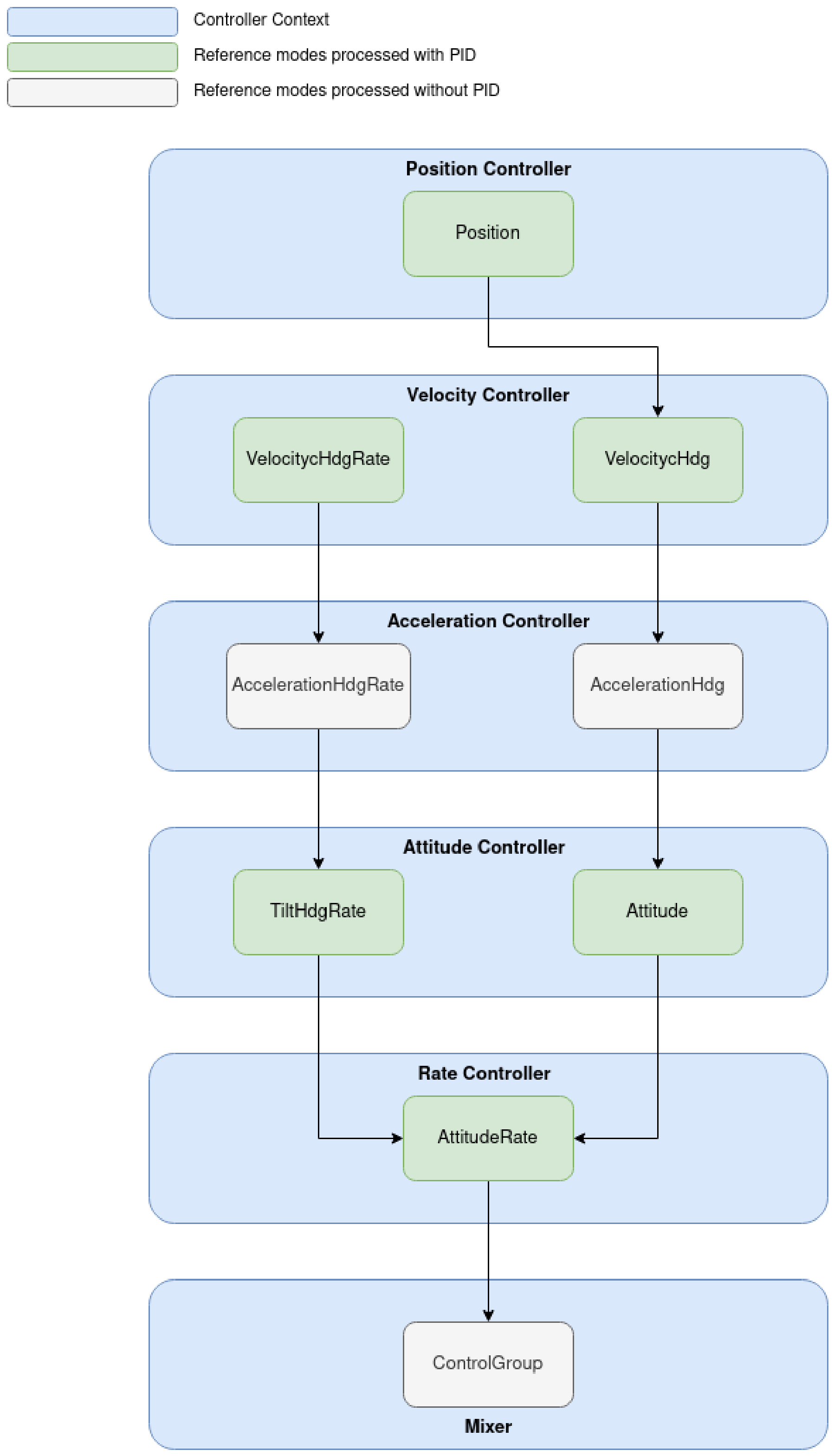


Figure 4.2: The whole cascade control system hierarchy.

Chapter 5

ML-Agents Framework

The simulation framework should be able to provide state-of-the-art tools for training UAV agents to test the robustness of new algorithms and intricate behaviors. RL is a complex domain that has encountered extensive academic attention, resulting in a vast theoretical base. For the simulation presented in the thesis, we aim for the simplest way of integrating and adjusting the hyperparameters for modern algorithms, therefore mitigating the complexity of the setup. The most suitable option for the Unity environment is the ML-Agents toolkit, as it provides a balance of customization and configuration complexity. It offers the algorithms that proved to be the most successful according to multiple studies, including the off-policy PPO algorithm and on-policy SAC. Finally, we present the practical implementation of the drone agent that can be used for various RL tasks.

5.1 ML-Agents Introduction

In the field of artificial intelligence and machine learning, the progression of ML-Agents from an open-source package to a widely utilized academic tool has been noteworthy. Originating as a project by Unity Technologies in 2017, ML-Agents initially targeted game development within the Unity framework. Early publications [28] showed the framework's potential in combination with complex physics simulation and various sensory inputs provided by the Unity simulation environment. Over time, its application expanded beyond gaming, gaining traction in academic research and serving as a platform for benchmarking and evaluating RL algorithms' performance for various tasks [6].

ML-Agents' primary objective is to empower developers to create agents capable of learning and decision-making based on their interactions with the environment. Using the reinforcement learning (RL) approach, these agents receive observations from the environment and feedback in the form of rewards or penalties for their actions, iteratively refining their decision-making processes.

The choice of ML-Agents for this thesis is explained by its several characteristics. The framework is native to the Unity environment, meaning it is already incorporated into the engine, adequately tested, and actively maintained. It provides a communication channel between the Unity scene and Python training scripts, mitigating the time complexity of its custom implementation.

Versatility is a crucial attribute contributing to ML-Agents' acceptance as a tool for academic research. Benefitting from the Unity native C# scripting, the developers can adjust the framework to various scenarios. The usage of pre-defined assets can significantly speed up the development process.

ML-Agents framework is open-source, which enables developers and researchers to understand the code base and provide their feedback, therefore contributing to the enhancement

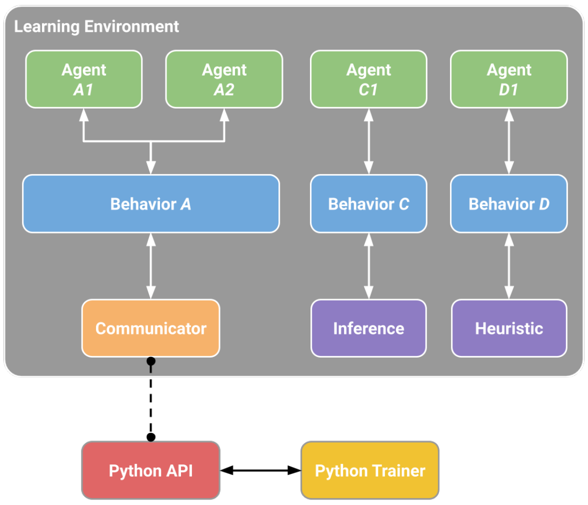


Figure 5.1: Overview diagram of the ML-Agents toolkit provided by ML-Agents documentation. [12]

of the software. This collaborative effort resulted in continual refinements, addressing specific academic requirements and challenges. ML-Agents evolved into a dynamic, community-driven tool shaped by the collective intelligence of its user base.

5.2 ML-Agents Building Blocks

On the Unity side, ML-Agents provides **ML-Agents Unity SDK**, allowing developers to set up any Unity scene as a learning environment. It requires identifying the agents and specifying the RL task to establish and define behavior. Behavior is a component containing the global parameters of an agent and acting as a function. It takes the input rewards and observations and outputs the corresponding action. The logic behind the decision-making process defines the type of behavior: learning, heuristic, or inference. *Learning* type means that the behavior is currently in the active training stage. After training, the model is saved and can be attached to a behavior. This time, the actions will be generated by an already trained network without adjusting its internal weights. This behavior type is called *inference*. The last type is called *heuristic* and practically means that the decisions are hard-coded without considering the observations and reward; therefore, the actions have no impact on the model, and no training is happening. Heuristic mode is used in case of user input to test the game or simulation environment, for example, by using a keyboard or controller. Creating a behavior requires implementing an Agent class and overriding pre-defined methods. This way, we can provide observations and rewards in a custom manner suitable for the RL task. Before training, global behavior parameters have to be set up in line with the Agent script. They include the number of observations, actions, or even their types (discrete or continuous). The length of an observation array should always be set up correctly for behavior as, inherently, it is represented by a float array with no additional explicit meaning.

A training scene in Unity can have multiple behaviors, as emphasized in Figure 5.1, as well as multiple agents sharing the same behavior. Each behavior has a unique name specified in Unity Editor under the Inspector tab. The name is crucial, as it serves as a key for connection to the backend. Developers should adjust the name in other parts of the toolkit depending on what behavior they want to train.

It is important to note that the agents sharing the same behavior do not share the same observations or actions. However, their experience contributes to the update of the whole network. This allows training a model on multiple agents with one learning type behavior, providing a significant speed-up. Other agents with different behavior types can coexist in the same learning scene. For example, we can simultaneously allow an agent with an inference behavior type to take action using the pre-trained model and control another agent using heuristic manual input.

The learning environment on the Unity side uses an external communicator to send the simulation results to the **Python Low-Level API**, making it decoupled from the Unity SDK. The Python code for the backend is distributed as a package called *mlagents_envs*. The API allows interaction with the Unity simulation flow from a Python script. Moreover, the package allows the use of custom machine-learning algorithms that are not included as standard options.

The final part of the standard pipeline is the Python package responsible for the training pipeline, **Python Trainer**. It provides a command named *mlagents-learn* that initiates the training configurable by the optional arguments.

Additionally, as part of the *mlagents-envs*, ML-Agents offers a wrapper for the OpenAI Gym. The wrapper currently supports single-agent tasks. However, it can be helpful as it allows for quick testing of other algorithms than PPO and SAC. This functionality is mentioned in Chapter 5.3.

It's important to mention that ML-Agents offers an additional wrapper for **PettingZoo** in the context of multi-agent collaborative environments. This Pythonic interface is designed to represent general multi-agent reinforcement learning (MARL) problems, although its functionalities are not explored within the scope of this thesis.

Monitoring the training process and displaying the results in the plot for more straightforward interpretation and comparison is facilitated using *TensorBoard*. It is a web-based visualization tool provided by TensorFlow, a popular open-source machine-learning library installed as part of the *mlagents* Python package. It serves as a tool for visualizing various aspects of the training process. The tool is used to provide analysis of the learning conducted in Sections 6.3, 6.4.

5.3 Open AI Gym

OpenAI Gym is a toolkit that allows the design, development, and comparison of various RL algorithms. It offers a standardized and modular interface for interacting with diverse RL environments. The toolkit also includes a range of pre-built environments covering classic control tasks, robotics, and Atari games. OpenAI Gym serves as a benchmarking platform, enabling fair comparisons between different RL algorithms. The toolkit is open-source, widely adopted, and integrated with popular RL libraries, making it a valuable resource for researchers and developers in the RL domain. However, the development of the Gym was moved to a new platform called Gymnasium in 2021. At the moment, ML-Agents provides a wrapper only for the OpenAI Gym, which makes it a possible source of dependency issues and problems with the lack of maintenance.

The wrapper is a part of the *mlgents_envs* Python package. It connects the Gym environment to the simulation in Unity and allows interactions with it. Combined with the OpenAI baselines [21], it can be used to test other RL algorithms in Unity that are not included in the standard ML-Agents set.

However, it has several limitations. Among the most crucial is the lack of support for providing stacked vectors of observations that are used to keep track of the agent's past states. Moreover, it only supports training single agent behaviors, excluding collaborative tasks. Some Gym features are not implemented, e.g., environment registration using *gym.make()* method and rendering of the next step of the environment using *env.render()*.

Due to the limitations of the wrapper and the dependency conflicts of the required Python packages (the legacy Gym code base requires outdated package versions that are deprecated), it was decided to focus on the native to Unity ML-Agents environment and actively maintained algorithms provided as a part of the standard ML-Agents Python backend.

5.4 Reinforcement Learning Algorithms

ML-Agents offers two differently designed RL algorithms: Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC). Both algorithms are prepared to be used out of the box. It means a user doesn't have to interact with the underlying backend training code in Python. The point of interaction with the setup is the YAML file with the definition of the network itself, its hyperparameters, and settings for the training process.

5.4.1 PPO algorithm

Proximal Policy Optimization (PPO) is a reinforcement learning algorithm introduced by John Schulman [33] and is currently the default option the ML-Agents offers. PPO has two primary loss functions: the policy loss and the value function loss. Each serves a distinct purpose in training the agent. The algorithm is associated with the on-policy optimization methods, and its contribution is the following formulation of the policy loss:

$$L^{CLIP}(\theta) = \mathbb{E}_t[min(r_t(\theta))\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t], \tag{5.1}$$

where:

- θ represents the policy.
- $\mathbb{E}$ operator means the empirical average over a finite batch of samples.
- $r_t(\theta)$ is a ratio between probabilities of the new and old stochastic policies.
- $\hat{A}_t$ is an estimator of the advantage function at a time stamp t.
- ϵ is a hyperparameter of the network.

Its difference from the previous on-policy algorithms as Trust Region Policy Optimization (TRPO) [40] is that it clips the r_t, i.e., the probability ratio. Consequently, it means that the algorithm penalizes such changes of a policy θ that move the ratio further away from 1. Practically, it means that it prevents excessively large updates of the policy. The impact of this change on the training is better performance, sample complexity, and ease of tuning while staying stable as previous trust-region methods. The training process, however, involves a combination of two losses, with the second being the *value function loss*.

$$L^{VF}(\theta) = \frac{1}{2}\mathbb{E}_t[(V_\theta(s_t) - V_{target}(s_t))^2], \tag{5.2}$$

where:

- $V_\theta(s_t)$ is the predicted value function for the state s_t.
- $V_{target}(s_t)$ is the target value function for the state s_t.

The target value function is a combination of the immediate reward r_t and the estimate of the value function at the next state $V_\theta(s_{t+1})$ discounted by the factor γ:

$$V_{target}(s_t) = r_t + \gamma V_\theta(s_{t+1}). \tag{5.3}$$

This loss represents the average difference between the learning algorithm's expectation of a state's value and the empirically observed value of that state. It is associated with training the value function, which estimates the expected cumulative reward from a given state. The value function is crucial for assessing the advantage of chosen actions in the policy update. A well-trained value function helps determine the quality of actions, contributing to PPO's overall performance stability.

The algorithm has shown stable performance in training agents for robotics and gaming applications [19]. It provides a balance between the performance of the trained models and the sample efficiency.

5.4.2 SAC algorithm

Soft Actor-Critic (SAC) is another policy optimization reinforcement learning algorithm developed by UC Berkeley and Google [27]. It differs from PPO's approach to data sampling as it is an off-policy algorithm. The original publication reveals that it is more sample-efficient than on-policy methods due to not losing the samples while performing the policy updates. However, the benefit of efficiency in sampling brings the complexity of setup compared to PPO. At the same time, SAC outperforms other off-policy algorithms such as Deep Deterministic Policy Gradient (DDPG) [38] and Twin Delayed Deep Deterministic Policy Gradient (TD3PG) in terms of robustness in hyperparameter tuning. Additionally, the unique feature of SAC compared to DDPG and TD3PG is the optimization of the policy entropy within the objective function. It naturally encourages exploration, preventing the network from executing the same actions.

SAC is categorized as a Q-learning algorithm. However, it employs several features. SAC concurrently learns two Q-functions Q_{ϕ_1}, Q_{ϕ_2} and a policy π_θ. The corresponding losses are:

- **Critic Loss** (or Value Function Loss) is designed to minimize the difference between the predicted state-action value and the current value incorporating the clipped double-Q trick. SAC takes the minimum Q-value between the two Q approximators. The loss is computed over a batch of experiences sampled from the replay buffer and can be formulated into the following equation:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim D}[\frac{1}{2}(Q_{\phi_i}(s,a) - (r + \gamma(\min_{j=1,2} Q_{\phi_{targ,j}}(s',a') - \alpha \log_{\pi_\theta(a'|s')})))^2], \tag{5.4}$$

  where:
  - s' is the next state.
  - a' is the action chosen by the policy.
  - $Q_{\phi_i}(s,a)$ is the current state-action value.
  - $Q_{\phi_{targ}}$ is another Q-function representing the running average of itself called the *target* [37].
  - γ is a discount factor.

  - $\alpha \log_{\pi_\theta(a'|s'))}$ represents the entropy regularization term of the policy π_θ, where α is the hyperparameter.
  - $\mathbb{E}_{(s,a,r,s')\sim D}$ is the expectation taken over a batch of experiences sampled from the replay buffer D. This batch-based approach ensures the optimization is performed on a representative subset of the agent's experiences.

The targets are updated as follows:

$$\phi_{targ,j} = (1-\tau)\phi_{targ,j} + \tau\phi_j. \tag{5.5}$$

The τ hyperparameter can be set up through the configuration file in the ML-Agents framework; however, most of the time, its value is 0.005. Critic Loss aims to minimize the squared difference between the predicted and target state-action values. This encourages the critic to accurately estimate the expected cumulative reward, providing a reliable signal for the policy update in the SAC algorithm.

- **Actor Loss** (or Policy Loss) aims to adjust the policy parameters to maximize the expected cumulative reward. It is designed to encourage actions that lead to higher returns while accounting for the entropy of the policy distribution. The equation of the loss is as follows:

$$L^{\pi}(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\alpha \log_{\pi_\theta(a|s)))} - \min_{j=1,2} Q_{\phi_j}(s,a)\right], \tag{5.6}$$

where:
  - $Q_{\phi_j}(s,a)$ is the value function term. It represents the estimated value of the minimal of the Q-function approximators. The goal is to subtract this value from the entropy, effectively encouraging actions that have higher expected returns according to the critic.
  - $\mathbb{E}_{s\sim D}$ is, similarly as in equation (5.4), the expectation taken over a batch of states sampled from the replay buffer D.

The ML-Agents framework helps to mitigate the complexity of the SAC algorithm setup due to the abstraction layer between the training Python backend and the Unity environment.

5.5 Drone Agent

Section 5.1 introduced a way to set up a simulation scene for training by creating various behaviors. This section presents a practical example of such behavior for the drone. The behavior is defined by overriding methods of an Agent class, resulting in *DroneAgent.cs* script. The script is attached to a drone placed in a scene to ensure the behavior is applied. After that, the framework has the actual GameObject that represents an agent.

5.5.1 Agent Class

Inside the custom definition of the Agent class, provided by the ML-Agents SDK, we define the routines needed for the training workflow depicted in Figure 5.2. To implement the application of incoming actions, the *OnActionReceived()* method is overridden. This method is called at every decision requesting point of the simulation and translates the actions into a reference for a controller on a specific level[1] based on pre-defined reference mode. To achieve

[1]Currently, the supported options are references for position, velocity, acceleration, attitude, and attitude rate controllers

that, the reference is sent to the UAV system, and we perform one step of the control calculation discussed in Section 4. A decision is every new action produced by the network. At

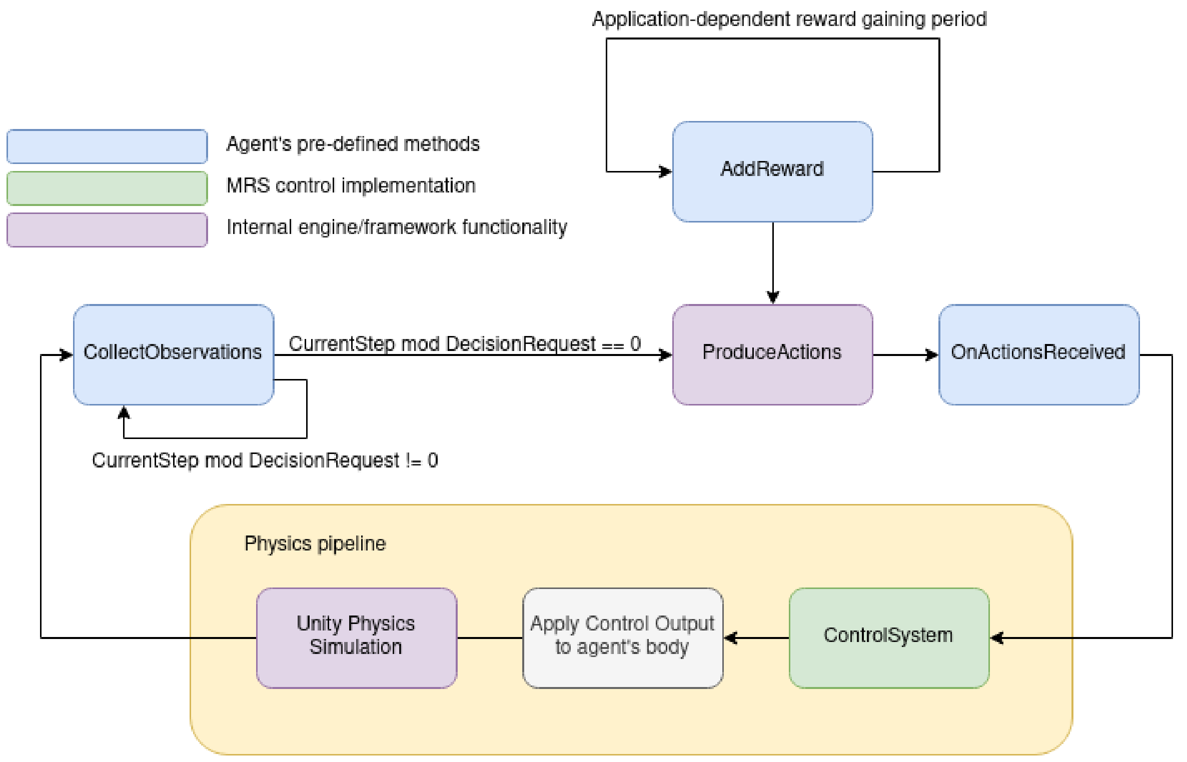

Application-dependent reward gaining period
Agent's pre-defined methods
MRS control implementation
Internal engine/framework functionality
AddReward
CollectObservations
CurrentStep mod DecisionRequest == 0
ProduceActions
OnActionsReceived
CurrentStep mod DecisionRequest != 0
Physics pipeline
Unity Physics Simulation
Apply Control Output to agent's body
ControlSystem

Figure 5.2: DroneBehavior workflow cycle while training a model on agent detail level

the same time, each time step of the fixed update of the environment, *CollectObservations()* method is called. Inside this method, we pass the model state (or additional data) to the agent to provide relevant task information to the agent. The observation is technically represented by an array of float values (each provided observation is stacked into one array). The length of the array should be set up in the Unity Editor similarly to the actions array mentioned earlier.

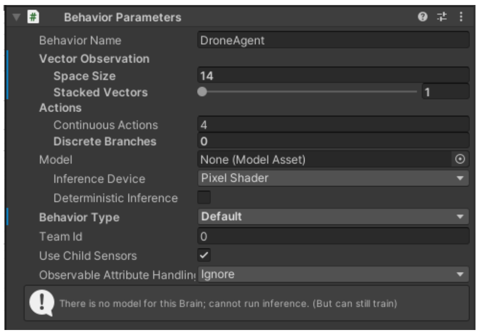
Assigning a reward to the agent is realized by passing a reward value to the *AddReward()* method. Because the implementation of reward is specific to a particular RL task, it can be called in any method of an agent class. However, for the task presented in this thesis, the reward is assigned every fixed update. Section 6.2 discusses more details about the concrete reward.

Among practical simulation-related pre-defined methods of the Agent class, there are *Initialize()* and *OnEpisodeBegin()*. *Initialize()* is called upon once at the start of the simulation upon enabling the agent. It is used to provide references to other GameObjects or optionally connect to ROS. On the contrary, the *OnEpisodeBegin()* method is called each time the agent starts a new learning episode to establish the same starting conditions of the experiment. The environment and the state of the drone are reset.

DroneAgent is designed in a way that it is a wrapper of the UAV quadrotor model that is, at the same time, an ML-Agents behavior that can be trained for various tasks. However, training a control task is impossible without assuring the correctness of implementing all consequential parts of the control pipeline. For that purpose, the agent's *Heuristics()* method is designed to take the references from the ROS node. We utilize the approach presented in Section 3.2 to achieve that. Generally, *Heuristics()* method can take any form of manual input suitable for the task.

5.5.2 Inspector Tab Settings

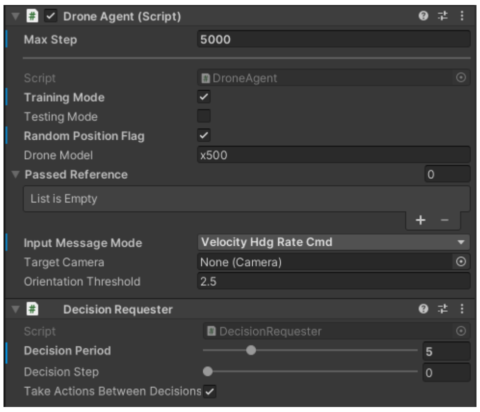
Once the Agent class is implemented, the parameters on the Unity side should also be adjusted. The script should be assigned to the agent GameObject, which automatically adds the components depicted in Figure 5.3.



Behavior Parameters	
Behavior Name	DroneAgent
Vector Observation	
Space Size	14
Stacked Vectors	1
Actions	
Continuous Actions	4
Discrete Branches	0
Model	None (Model Asset)
Inference Device	Pixel Shader
Deterministic Inference	
Behavior Type	Default
Team Id	0
Use Child Sensors	✓
Observable Attribute Handlin	Ignore

There is no model for this Brain; cannot run inference. (But can still train)

(a) Behavior Parameters Menu



Drone Agent (Script)	
Max Step	5000
Script	DroneAgent
Training Mode	✓
Testing Mode	
Random Position Flag	✓
Drone Model	x500
Passed Reference	0
List is Empty	
Input Message Mode	Velocity Hdg Rate Cmd
Target Camera	None (Camera)
Orientation Threshold	2.5

Decision Requester	
Script	DecisionRequester
Decision Period	5
Decision Step	0
Take Actions Between Decisions	✓

(b) DroneAgent class parameters and Decision Requester

Figure 5.3: DroneAgent GameObject's set up in the Inspector tab

BehaviorParameters menu provides the interface for setting up the name of the training behavior, length of the actions and observations arrays, and current behavior type. The *StackedVectors* option allows to save multiple observations from the past, keeping the agent aware of the previous states. This comes with the increased training complexity. Action space can be continuous or discrete. DroneAgent's action space is continuous, therefore, the *DiscreteBranches* parameter is set to zero, while the number of *ContinuousActions* is four, representing the length of the provided reference vector. For the model parameters, the options are either to use a pre-trained model or leave the model as *None* to start training the agent. Behavior type has one of the possible values discussed in Section 5.1: Inference, Default, or Heuristics. The default type is used when training. Additionally, there is a *TeamId* parameter for grouping the agents into teams with shared reward and *UseChildSensors* flag to look for the sensors in the lower hierarchy (children) GameObjects. Those parameters are not relevant for the DroneAgent.

One of the crucial Unity components of an agent instance needed for the training of the model is a component called *DecisionRequester* shown in Figure 5.3b that is provided by the ML-Agents Unity package. Its purpose is to trigger the extraction of observations from an agent with a pre-defined regularity. The extraction period can be set by changing the *DecisionPeriod* parameter, e.g., with the current settings, it will trigger the observations every 5 *FixedUpdate* steps. The exact step within the *DecisionPeriod* when the observation is taken is addressed by the *DecisionStep* parameter. The same action is repeated when the *TakeActionsBetweenDecisions* flag is activated. Otherwise, the agent is waiting for the decision point. Without the *DecisionRequester*, the decision could only be triggered manually using

RequestDecision() method inside an Agent class. For the Drone Agent, the decisions should be taken during every *FixedUpdate* to prevent delays in control system signals.

The reference mode is set as DroneAgent's parameter in Figure 5.3b under the DroneAgent script view. The provided implementation allows switching between the training and testing modes without changing the underlying code. The drone model has to be specified to read the additional parameters from JSON files. A user can choose what type of input reference messages the drone will expect during the simulation.

Chapter 6

Training

This chapter illustrates the experiments with the RL training conducted using the ML-Agents framework combined with the drone's control system discussed in Chapter 4. More task-specific details of the drone agent's implementation and the Unity Scene setup that were not addressed in Section 5.5 are introduced here. Consequently, the Sections 6.3 and 6.4 outline training specifics with particular algorithms[1].

6.1 Hardware/Software Setup

The following table contains the characteristics of the laptop used for the experiments.

Hardware Device	**Characteristics**
Laptop	Lenovo Legion 5 Pro 16ACH6H
RAM	16 GB, DDR4-3200
CPU	AMD Ryzen 7 5800H / 3.2 GHz
GPU	NVIDIA® GeForce RTX™ 3060 Laptop GPU, 6GB GDDR6
WLAN	Wi-Fi® 6, 802.11ax 2x2 Wi-Fi
Bluetooth	Bluetooth 5.1, M.2 card

Table 6.1: The hardware characteristics device used for conducting the experiments presented in the thesis.

Similarly, the software setup during the project is the follows:

Software	**Version**
Unity Editor	2022.3.7f1
Rider IDE	2023.3
Python	3.10.12
ML-Agents SDK	3.0.0-exp.1 2023-10-09
mlagents (python package)	1.0.0
mlagents-envs (python package)	1.0.0
ROS TCP Connector	0.7.0-preview
URDF Importer	0.5.2-preview

Table 6.2: The software packages and corresponding versions.

[1]A video of the training process can be found on https://mrs.felk.cvut.cz/kisselyov2024thesis

6.2 RL Task definition

This experiment aimed to teach the drone to reach a specific position without creating excessively aggressive motion that would cause the drone to lose visual odometry in the possible real scenario. The model has to achieve the goal by controlling the references on the velocity control level in every *FixedUpdate* step. The task should be designed in a generalizable way, i.e., the drone should be able to accept any position state and not rely on the assumption of getting the same every time.

6.2.1 Unity Scene Setup

The scene setup is crucial as it has to allow multiple DroneAgents to train simultaneously without interrupting each other. Therefore, several training environments were created for nine agents, forming a training grid displayed in Figure 6.1. With this scene, it is possible to train all agent instances in parallel, as every agent contributes to updating the same behavior. The number of agents was estimated empirically according to the hardware capabilities. Each training platform is identical and consists of the following parts: a simple horizontal plane, an internal cube for representing the visual range of target position spawn, and an external cube to restrict the allowed flying space. The size of the reference spawning area is 80 × 80 × 19 meters (leaving 1 meter above the ground to prevent collision with the ground), and the corresponding size of the flying space is 190 × 190 × 95 meters. The plane and the external borders are equipped with colliders that send callbacks when the drone's mesh overlaps with them. This grid can be created quicker by utilizing the Unity *prefabs*. It allows to store one

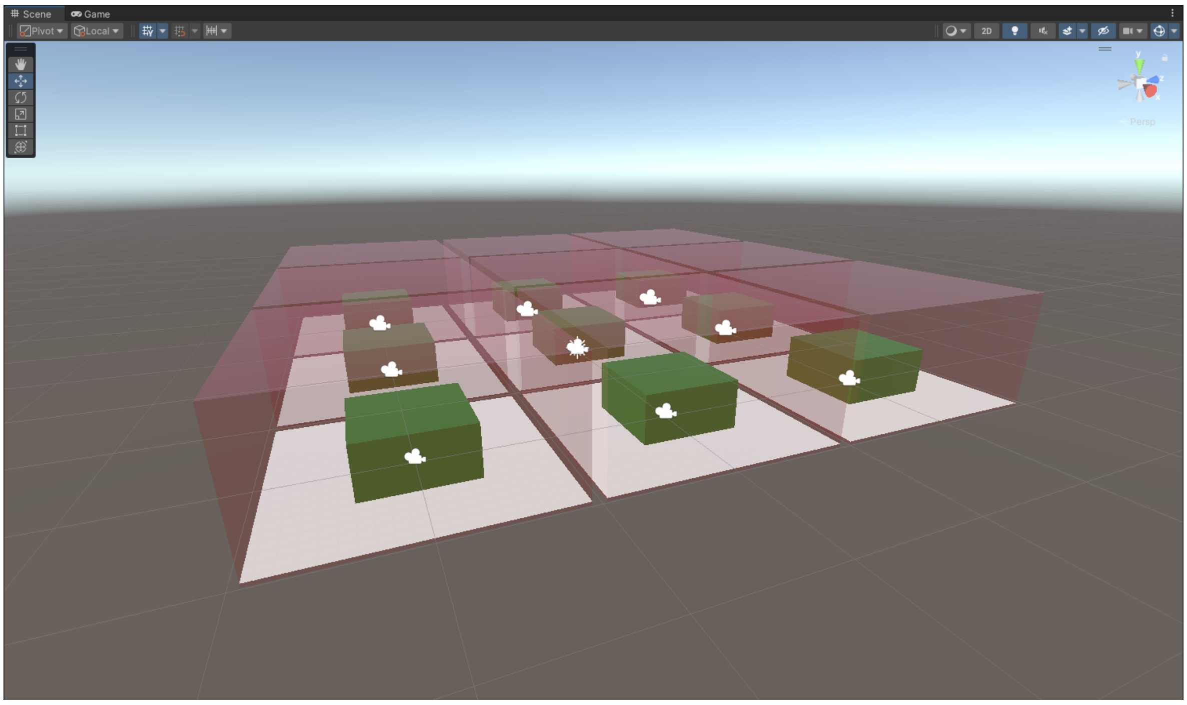


Figure 6.1: Training scene setup with multiple agents.

environment with all nested GameObjects and quickly transfer it into any scene within the project.

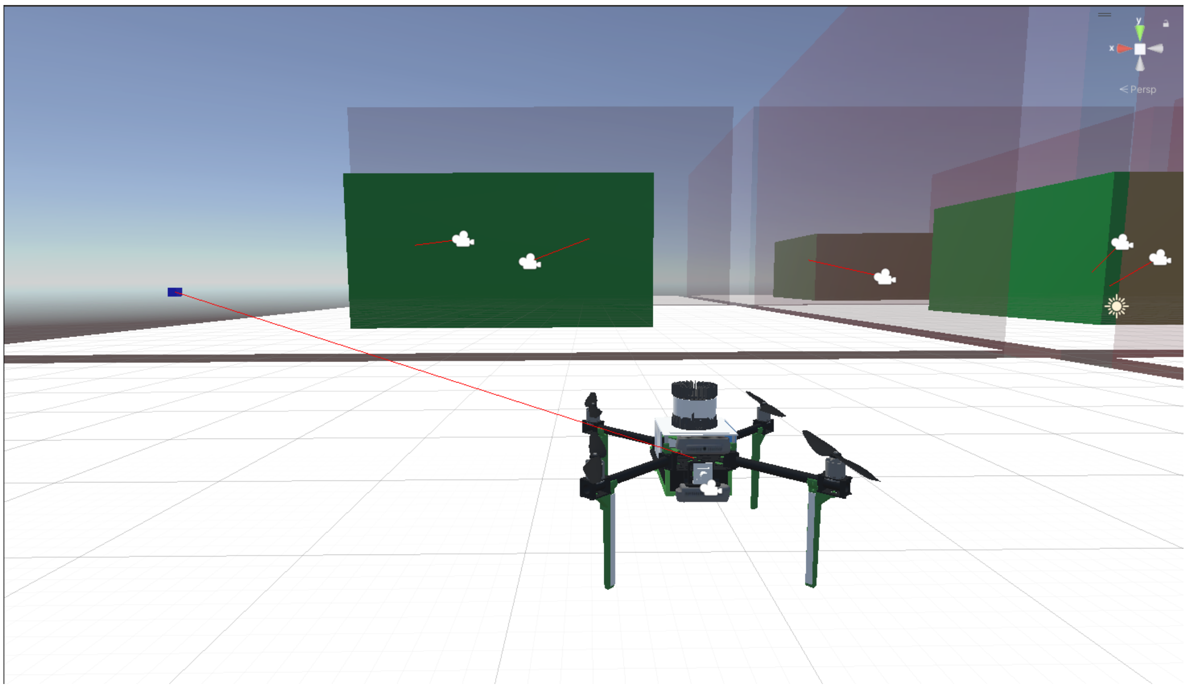


Figure 6.2: Image of a drone with the target goal position during a training episode.

To help visually represent the training process, allowing a quick heuristic assessment of the learning state, we provide a set of auxiliary routines as shown in Figure 6.2. Upon the simulation, the target is represented by a dynamically spawned GameObject that has no colliders and serves only a visual purpose. Every *Update* of the simulation, the script draws a line from the drone to the target position that helps to establish the movement direction of the drone. During the simulation, to speed up the sample gathering, the timescale is increased to 20 (20 *FixedUpdates* during the original fixed step).

6.2.2 Episode Formulation

The learning is split into episodes. During an episode, the agent performs actions and collects a cumulative reward. The episodes aim to provide unified initial conditions for the DroneAgent and limit the number of actions it can take. The state of the agent is reset every time the *OnEpisodeBegin()* method is executed. The position of the drone and target is set to different random positions inside the reference spawning area. The duration of an episode is specified through the configuration file discussed in Section 6.2.4. However, the episode can end before the maximal simulation step upon reaching the target position or triggering the colliders of the plane or outside borders with corresponding static rewards.

6.2.3 Rewards and Observations

Following each drone action, the environment provides feedback in the form of rewards and concurrently supplies observations to facilitate the agent's understanding of the evolving dynamics. The observations must provide enough information for the agent to achieve the target position. Here's the outline of the observations used during the training:

- Normalized linear velocity vector (3 values).

- The magnitude of the linear velocity vector normalized by the maximal allowed speed of the drone (1 value).
- Normalized vector from the current position to the target (3 values).
- Normalized angular velocity vector (3 values).
- Normalized quaternion representing the rotational transformation q_{rel} required to achieve the target rotation q_{target} from the current rotation q_{curr}.(4 values):

$$q_{rel} = q_{target} q_{current}^{-1}. \tag{6.1}$$

- Normalized angle between two quaternions q_{curr} and q_{target} (1 value):

$$\theta = cos^{-1}(2\langle q_{curr}, q_{target}\rangle^2 - 1), \tag{6.2}$$

  where $\langle q_{curr}, q_{target}\rangle$ is the dot product of quaternion vectors. This metric represents the magnitude of rotation needed to transition from one orientation to another, computed in Unity using the built-in method *Quaternion.Angle(*q_{curr}, q_{target}*)*.

The aggregate number of observations, represented by the length of the concatenated array, stands at 15. Normalizing the observations is a crucial step, as the model cannot learn properly from non-normalized inputs. To enhance the model's generalizability and mitigate the risk of over-fitting to a particular target, the observations are intentionally designed to be relative to the final target. This relative design fosters adaptability. A key facet of the observation process involves their periodic extraction at each *FixedUpdate*, ensuring a consistent and well-timed data input.

The presented *reward function* underwent multiple tuning iterations. In alignment with the dynamic nature of the drone's control actions, the reward is assigned at every *FixedUpdate*, allowing instantaneous responsiveness to changes in the model's behavior. This frequent feedback loop contributes to the efficiency of the training process.

Comprising various components, the *reward function* is intricately structured to capture and reinforce desired behaviors in the learning process:

- Distance reward, which is calculated as the Euclidean norm of the vector between the current position and the target. The reward is gradually scaled up when the drone approaches the target.
- Velocity direction reward, which is calculated as the dot product of the current linear velocity vector and expected direction of movement, i.e., again, the vector from between the current position and the target.
- High acceleration penalty, a penalty for sending references that significantly differ from the current velocity.
- *(Optional)* Angular velocity stability reward, although the reasoning for this reward is the same, it directly rewards the model for the stable changes of orientation, which experimentally proved effective.
- Final positive reward for reaching the target with threshold tolerance for both position and orientation.
- Final negative reward for triggered collision with the border area or the horizontal plane.

Each part of the reward is normalized to stabilize the output of the whole reward function. To address different scales, every modular part has a corresponding coefficient for fine-tuning. The list of the coefficients is included in Appendix 8.1 in Table 1.

Notably, the most substantial relative coefficients are assigned to the position and velocity rewards, given their pivotal role in propelling the agent towards the target. The penalty for high acceleration is delicately calibrated to 10% of the position reward, maintaining a nuanced balance. Similarly, the penalty associated with angular velocity stability is scaled at 50% relative to the position reward, establishing a proportionate adjustment. To ensure the drone's stabilization at the destination, the final positive reward is intentionally set at a high value, compellingly encouraging the agent to converge successfully. Conversely, a negative equivalent is attributed to the drone in response to triggering a collision with the borders, thereby reinforcing the consequence of undesired actions.

During the exploration of alternative reward functions, various modifications to the distance reward were examined. One approach involved dynamically adjusting the distance reward based on the drone's proximity to the target compared to the previous state gauged at each *FixedUpdate*. Specifically, if the drone exhibited a trajectory bringing it closer to the target, the distance reward was augmented by a factor ranging between $10 - 20\%$. Conversely, if the drone's distance increased compared to the previous update, the distance reward was diminished by the same coefficient — this reward adaptation aimed to mitigate issues related to excessive motion. Another explored variation focused on assigning the distance reward exclusively for reductions in the distance between the target and the drone. However, this specific reward configuration proved more challenging to scale within the broader reward function.

Throughout the training process, a recurring challenge surfaced in the form of target overshooting, leading to undesirable oscillations. In response, a preventive measure was implemented by fine-tuning the velocity reward under circumstances where the drone approaches proximity to the target. More precisely, when the drone reaches a distance of 1 meter, the velocity reward undergoes a multiplicative adjustment determined by the deviation of the drone's current velocity from the target velocity. The target velocity is intentionally set at a low value, progressively decreasing (from 0.1 down to 0 m/s) as the drone approaches the target. This deliberate configuration compels the drone to decelerate gradually as it nears the goal, mitigating the tendency for overshooting and encouraging the agent to smoother converge to the target.

6.2.4 Training Configuration File

The configuration file is the interface for interaction with the model parameters and the RL algorithms. RL is sensitive to the tuning of the hyperparameters, so the settings should be addressed carefully and tested during the training. Therefore, multiple combinations were tested and tracked during the training for further analysis. We list the values used during the training in Appendix 8.1. The following are the observations of the impact of the tuning of relevant parameters:

- **trainer_type**: specifies which RL to use, PPO or SAC.
- **keep_checkpoints**: the number of checkpoints that ml-agents keeps. It allows us to keep track of the previous states of the network in case it is needed to return while removing unnecessary backups.
- **checkpoint_interval**: the number of experiences collected between each save of a checkpoint, i.e., the period of checkpoints.
- **max_steps**: the number of total experiences for one whole training. The simulation stops upon reaching the maximum amount. Moreover, the optional changes of some parameters such as *learning_rate* are relative to this value.

- **time_horizon** defines the number of steps collected before saving them to the experience buffer. When the number is reached, the value estimation is used to predict the final potential reward considering the current state. Due to frequent reward assignments (every *FixedUpdate*), a smaller number was used because a consequent reward tends to result in a better estimate.
- **threaded**: allows asynchronous simulation steps at the same time as the model updates. Setting this parameter to true resulted in a marginal speed-up of the computation in the case of the SAC algorithm due to its on-policy nature.
- **network_settings** is a group of parameters defining the network structure:
  - **hidden_units**: represents the number of nodes in each fully connected neural network layer. The relation between the observations and actions is considered not complex during the position control training. Therefore, the number is low.
  - **num_layers**: represents the number of layers in the network. Similarly to the **hidden_units** parameter, setting this parameter too low reduces the size of the network, i.e., its complexity. The default value was left.
  - **vis_encode_type**: the parameter is relevant to the tasks with visual observations and, therefore, not utilized in current training.
  - **normalize**: this parameter allows providing non-normalized observation values as the normalization is happening on the Python side using the moving average. The parameter is set to *false* as the observations are normalized before the input into the model inside the C# script. This assures stable and consistent normalized values.
- general **hyperparameters** is the group of parameters that are shared for both presented algorithms:
  - **batch_size**: the number of decisions realized before each gradient descent. Because the presented task is continuous, the batch size is set higher than default. That is explained by the fact that continuous problems require more samples to learn because the state space is more extensive. At the same time, the parameter was set differently for PPO and SAC due to the specifics of the algorithms.
  - **buffer_size**: the number of experiences collected before the training script updates the model. The model learns from the accumulated rewards and observations. Should have the size of multiple *batch_size*. Bigger *buffer_size* results in more stable updates. Corresponding to the *batch_size* was set to different values for PPO and SAC.
  - **learning_rate**: learning rate used when performing the gradient descent. Experimentally, it was identified that high *learning_rate* (~ 0.003) results in slower growth of the cumulative reward and, therefore, slower learning.
  - **learning_rate_schedule**: provides two options, *linear* or *constant*. By default, the linear decline option is set for the PPO algorithm to ensure its stable convergence and kept constant for SAC to establish the natural convergence of its Q-function.
- **reward_signals** the parameters affecting the incorporation of the reward signal into the training:
  - **gamma**: determines the discount factor for future rewards originating from the environment. This factor reflects how far into the future the agent should consider potential rewards. A bigger value is appropriate when the agent needs to act in the present to prepare for rewards in the distant future. Set to 0.99, considerably a high value.
  - **strength**: reward multiplier. Left as default because the *C#* script internally addresses the reward's scaling.

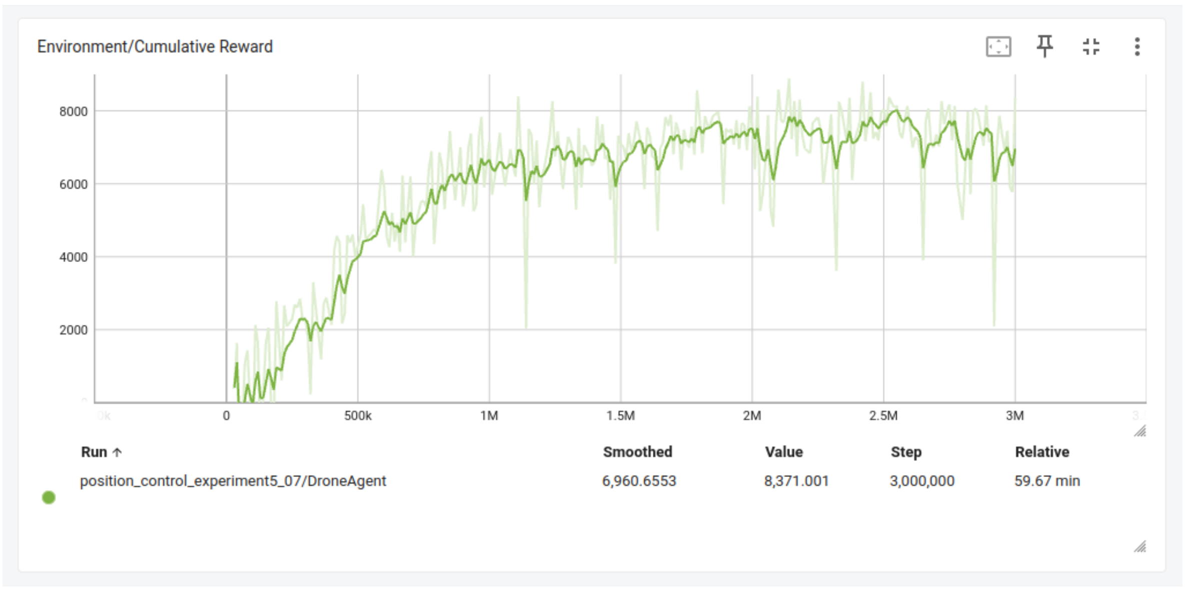
6.3 PPO training process

The entire configuration file used for training is included in the Appendix 8.1. The whole grid of 9 drones was utilized for the training, creating more samples in a shorter time. The *max_step* parameter is set accordingly to provide enough samples for the model to reach the maximum possible reward from the training. Setting the *buffer_size* to a relatively higher number, 20480, which is ten times higher than the *batch_size*, is a requirement of the algorithm connected to its relatively lower sample efficiency.

Among the PPO-specific hyperparameters that we can adjust in the ML-Agents training configuration file, there are:

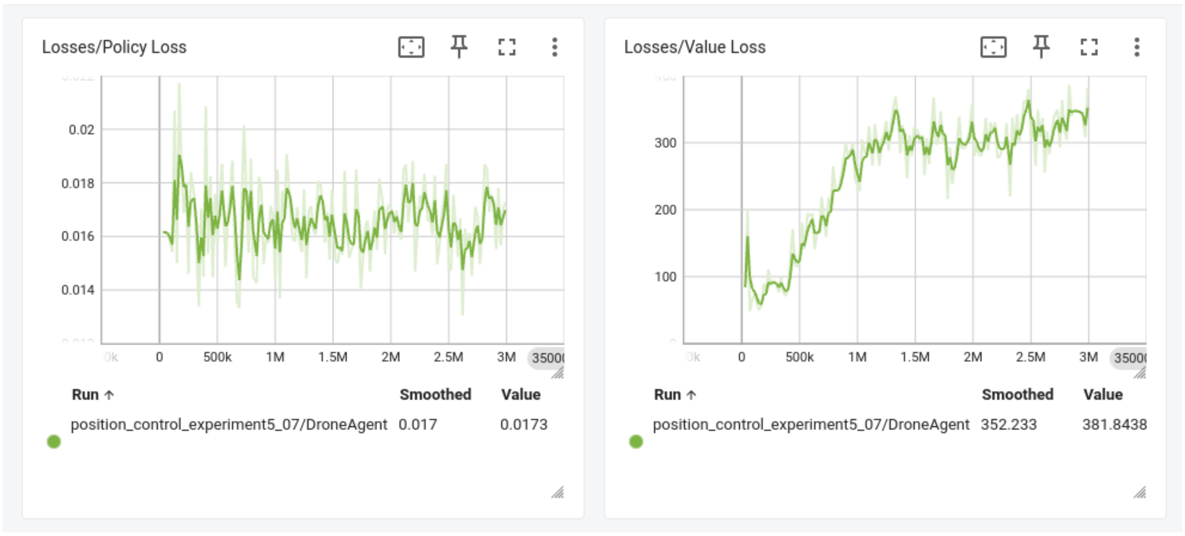
- **beta**: is an entropy parameter. Adjusting the beta hyperparameter influences the level of exploration undertaken by the agent. A higher value encourages more exploration. In the context of the current task, a relatively elevated value was chosen to foster extensive exploration.
- **beta_schedule**: change of the beta parameter over time, which determines the evolution of the entropy parameter over time and offers two options: *linear* or *constant*. Opting for the *linear* schedule facilitates a gradual reduction of the entropy parameter, reaching 0 by the end of the training. This linear progression ensures a smooth transition from exploratory to exploitative behavior.
- **epsilon**: corresponds to the ϵ from equation (5.1). Regulating it means finding the balance between the training speed and the updates' stability. Taking into consideration the quick positive reward gain observed during the training, stability was given the preference; therefore, the parameter's value is low.
- **epsilon_schedule**: change of epsilon over time, *linear* or *constant*. The decline of the epsilon parameter was set to be linear to provide more stability by the end of the training.
- **num_epoch**: the hyperparameter denotes the number of times the experience buffer is employed for gradient descent optimization. Setting this parameter to 7 (default is 3) speeds up the training as is possible due to large *batch_size*, which balances the stability of the updates.

The observed changes of the cumulative reward plot are depicted in Figure 6.3. The plot is smoothed to remove the outliers. We also provide the graphs (Figure 6.4) of the loss functions for analysis of the training process.



Run ↑	Smoothed	Value	Step	Relative
● position_control_experiment5_07/DroneAgent	6,960.6553	8,371.001	3,000,000	59.67 min

Figure 6.3: Reward plot during training with PPO algorithm.



Run ↑	Smoothed	Value
● position_control_experiment5_07/DroneAgent	0.017	0.0173

Run ↑	Smoothed	Value
● position_control_experiment5_07/DroneAgent	352.233	381.8438

Figure 6.4: Loss functions observed during the training with PPO.

6.4 SAC training process

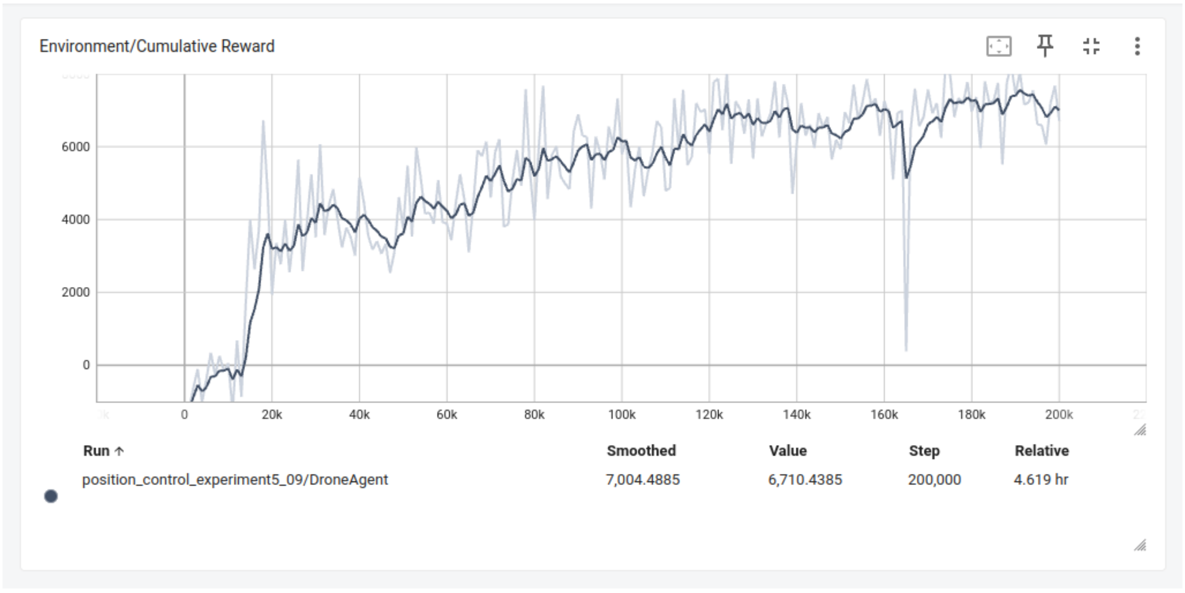
Due to high sample efficiency, we don't need to speed up the generation of experience samples for training with the SAC algorithm. Therefore, the training grid was reduced to one drone. Moreover, when multiple drones are training in parallel, they tend to quickly collect a batch of experiences before being able to explore the action space. To fill several episodes into a single batch, its size should be significantly increased, which would result in computational complexity the testing hardware cannot provide. The *buffer_size* is increased mainly due to the off-policy nature of SAC, which relies on a diverse set of experiences for effective learning.

In the training configuration file, specifically for SAC, we can set the following parameters:

- **buffer_init_steps**: specifies the number of steps the algorithm initially uses to fill the buffer. This parameter is set to 1000 to prefill a couple of episodes before the start of the training.
- **init_entcoef**: the entropy coefficient set at the beginning of training. During training, the coefficient is adjusted to the predefined target entropy defined in [27]. However, this parameter allows controlling the exploration at the beginning of the training. This coefficient is set to a relatively high value to provide early exploratory behavior.

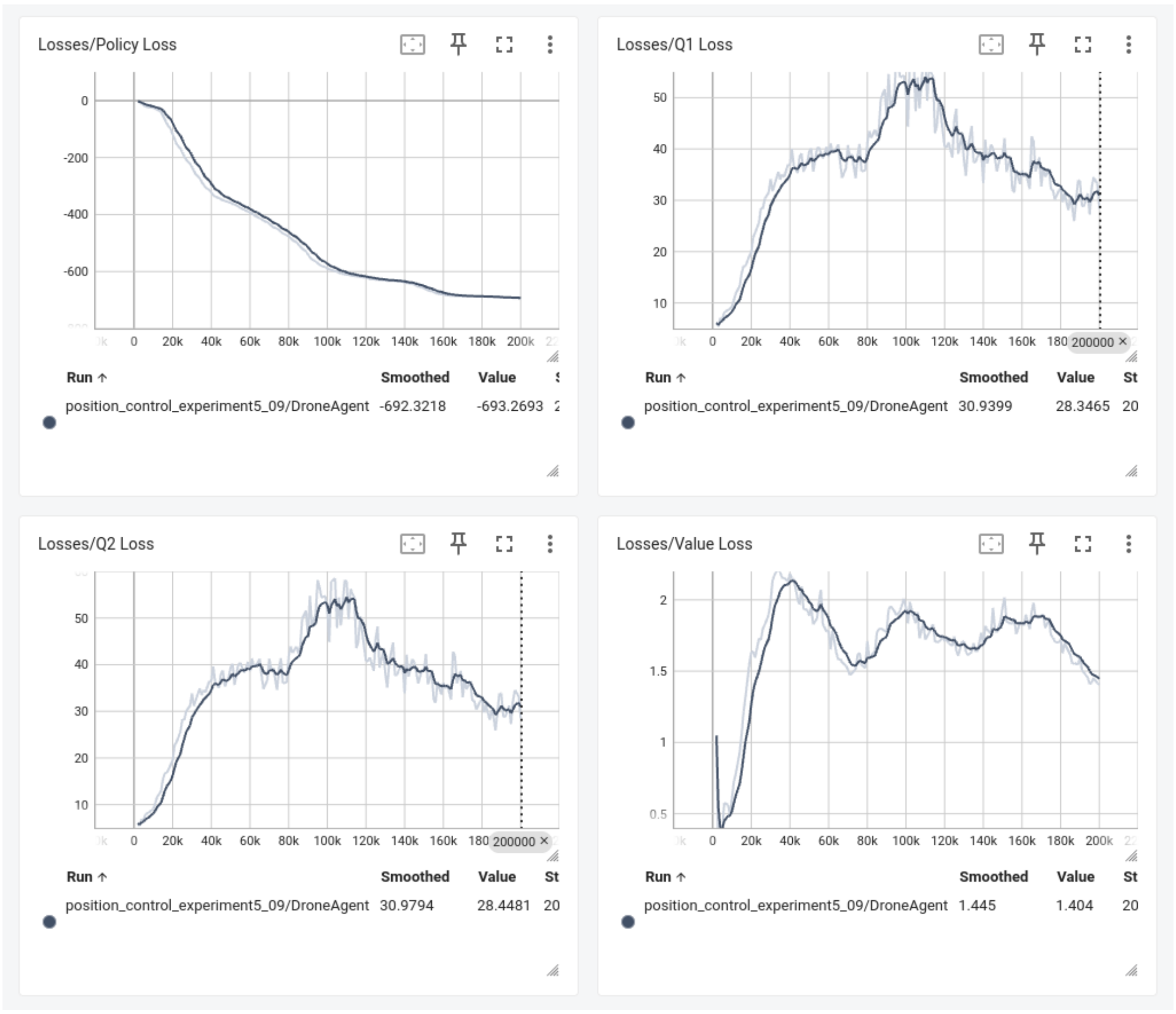
The training workflow is periodic and follows the same pattern: the agent executes actions and collects experiences until the *batch_size* is reached. Then, the whole environment is paused while the training script performs an update of the model.

The behavior during training is depicted in the cumulative reward and the loss plots in Figure 6.5 and Figure 6.6.



Run ↑	Smoothed	Value	Step	Relative
● position_control_experiment5_09/DroneAgent	7,004.4885	6,710.4385	200,000	4.619 hr

Figure 6.5: Reward during the training with SAC algorithm



Run ↑	Smoothed	Value	!
● position_control_experiment5_09/DroneAgent	-692.3218	-693.2693	2

Run ↑	Smoothed	Value	St
● position_control_experiment5_09/DroneAgent	30.9399	28.3465	20

Run ↑	Smoothed	Value	St
● position_control_experiment5_09/DroneAgent	30.9794	28.4481	20

Run ↑	Smoothed	Value	St
● position_control_experiment5_09/DroneAgent	1.445	1.404	20

Figure 6.6: Loss functions observed during the training with SAC algorithm

Chapter 7

Discussion and Models Evaluation

Following the experimentation detailed in the previous chapter, this section undertakes an analysis of the observations, discusses outcomes, and evaluates the performance of the trained models.

7.1 Training Evaluation

During the training, we observed multiple characteristics of the algorithms, which make them suitable for different applications, depending on their requirements. Several critical aspects are considered during the training evaluation:

- **Sample Efficiency**.
  As anticipated, the PPO algorithm required more experiences to attain a comparable to SAC reward level. In Figure 6.3, we observe rapid reward growth over the first million samples, followed by a gradual convergence, reaching its maximum at 2.5 million samples. In contrast, SAC achieved a similar reward level within 200 thousand samples. The substantial early growth in SAC corresponds to a rapid acquisition of the basic desired behavior facilitated by a large update based on the successful action sequences.
- **Real time training length**.
  Despite SAC's superior sample efficiency, training the model with the SAC algorithm took significantly longer than PPO. This discrepancy is attributed to the extensive processing of collected samples during SAC's model update. However, this bottleneck might be the attribute of the underlying hardware platform, not the algorithm itself. Conversely, PPO discards samples progressively during training, making its training speed more contingent on sample acquisition rather than model update. We can claim that with the current setup, we can achieve comparably similar performance quality in a shorter real-world time using the PPO algorithm. Moreover, tuning hyperparameters for SAC requires more time due to the slow training speed.
- **Update Stability**.
  Under the presented hyperparameter setup, PPO demonstrated more stable reward growth without sudden and abrupt changes. The clipping of the probability ratio, preventing large policy updates (as discussed in Section 5.4.1), contributes to this stability. However, SAC's update stability can be fine-tuned by adjusting the size of the experience buffer. Another possible explanation for the more rapid updates during the training of the SAC model is the usage of a single agent. In contrast, PPO was trained while exploiting nine drones in parallel[1].

[1]Note that for SAC the number of agents training in parallel would not significantly affect the real-world time spent on the training because the bottleneck was the model update based on gathered samples.

- **Loss Analysis**.
  In PPO, the evolution of the *value loss* function (see Figure 6.4) over time is another indicator of successful training. This correlates to the quality of the model's estimate of the value of each state. The plot is growing as long as the agent is learning. The *policy loss*, in turn, corresponds to the changes in the policy, and it has oscillatory behavior. The vital characteristic here is the magnitude of changes, which should be less than 1. In our case, the oscillations are small in magnitude, with the difference between the maximum and minimum value being ~ 0.009.
  The loss plots depicted in Figure 6.6 offer valuable insights into the evolving dynamics of the SAC algorithm during training. The decreasing trend observed in the policy loss function indicates the algorithm's progressive learning in generating policies that yield higher rewards. At the same time, the diminishing Q1 and Q2 loss functions signify the algorithm's improving capability to predict expected rewards accurately. The value loss function also decreases over time. Still, it fluctuates more than the other loss functions, corresponding to the algorithm learning to predict the expected reward for a given state. Notably, the policy loss function is decreasing at a faster rate than the Q1 and Q2 loss functions. This suggests that the algorithm prioritizes learning the policy over learning the Q-functions.

7.2 Algorithms' Performance Evaluation

Both algorithms successfully trained the model to reach the desired position and orientation. Nevertheless, certain aspects of the desired behavior were more challenging to achieve. Subsequent convergence is crucial as the agent learns to eliminate excessive motions and quick accelerations. Notably, mitigating undesired oscillations (discussed in Section 6.2.3) is one of the most intricate aspects of the behavior, sensitive to tuning. The additional reward for the lower angular velocity helped to improve the stability of the drone's angular movements, removing the undesired rotations and helping the agent slowly achieve the final rotation. Experiments revealed that capturing nuanced motion near the goal is more challenging for PPO, requiring multiple reproductions of successful sequences. In contrast, SAC efficiently extracts complex behavior without excessive reproduction. At the same time, despite both algorithms achieving the training target, SAC demonstrated smoother motions overall (see Figure 7.2), exhibiting fewer undesired deviations from the direction to the goal than PPO (depicted in Figure 7.1). Based on multiple observations, the model trained by the PPO algorithm tends to approach the target by a parabolic trajectory.

The adjustments of the reward function for the close distances to the target successfully provided the agent with enough information to reduce the overshoot. In the case of SAC, the model learned to avoid them entirely.

Due to the relative definitions of the reward, the algorithms can be generalized to any position in the world frame within the trained range, which makes both models adaptable to different environments.

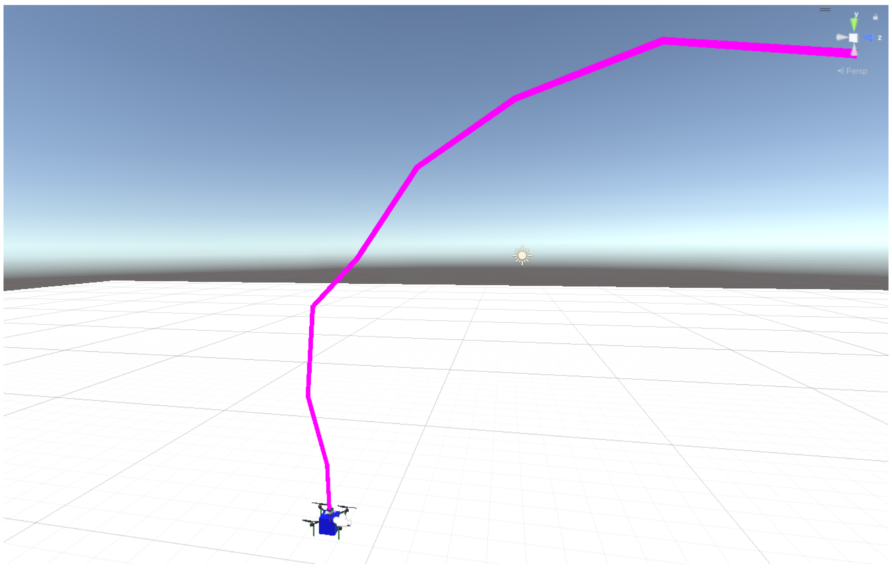


Figure 7.1: Trajectory of the trained PPO model for DroneAgent behavior

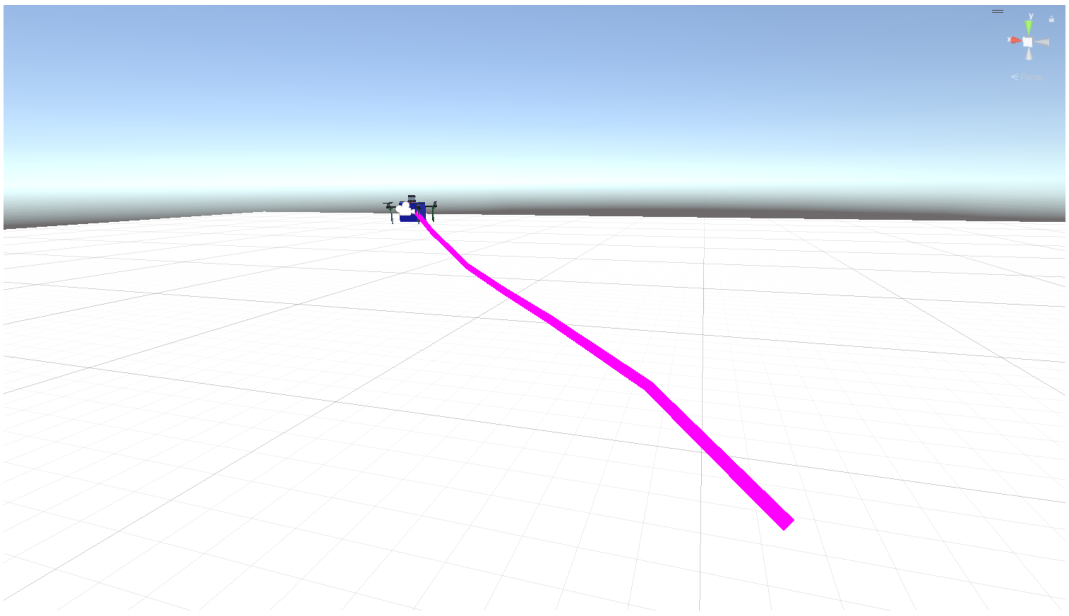
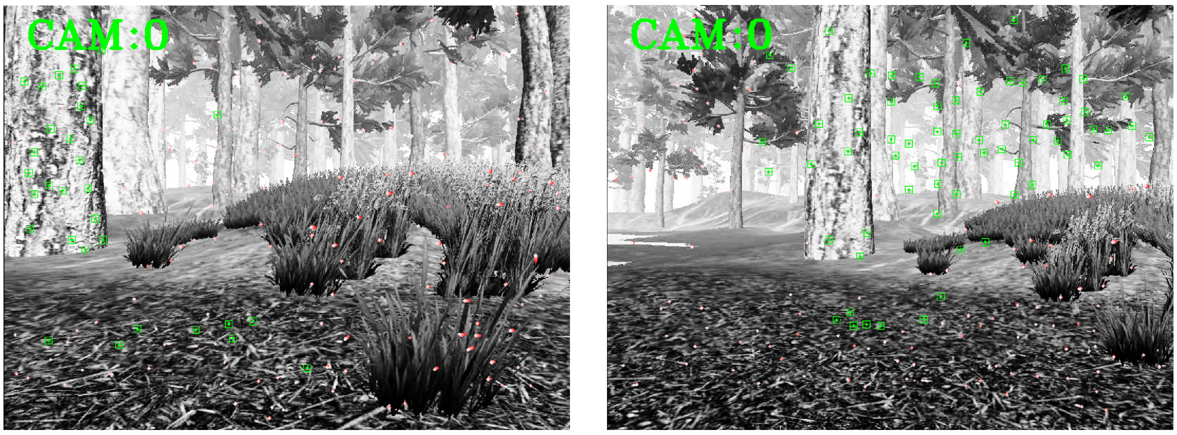


Figure 7.2: Trajectory of the trained SAC model for DroneAgent behavior

7.2.1 Flying with OpenVINS

In this section, we provide an analysis of the ability of the algorithms not to lose tracked features used to estimate the UAV odometry while flying toward the target. The observations were captured in the forest environment (Figure 2.5) and industrial side (Figure 2.6) as different scenes provide different graphics settings, shaders, and lighting.

In our experimental setup, we deployed agents in a forest environment, specifying target positions in the Inspector for both PPO and SAC pre-trained models. The forest environment, with dynamic elements like wind and constant movement, poses challenges for feature detection. Figure 7.3a illustrates that during rotational movements, the agent potentially loses

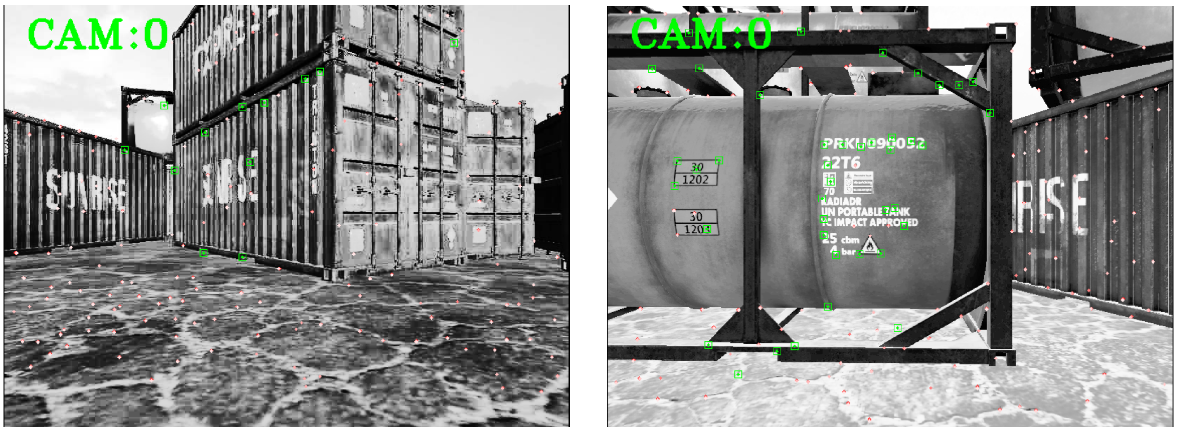


(a) Features tracking stabilized after rotation. (b) Features tracking during a slow direct flight.

Figure 7.3: Flying with OpenVINS in the forest scene

unstable features and instead primarily focuses on static environment objects, such as trees and ground textures. Direct flights (Figure 7.3b) can be problematic, especially if the model hasn't properly learned slow acceleration behavior.

Both algorithms aimed to preserve the odometry during the flight by avoiding quick accelerations and keeping the low average velocity; however, they showed different stability. Due to the quality of the trajectory, the PPO model has a higher tendency to create excessive motions than the model trained with SAC. That often results in losing some of the tracked features. Moreover, the smooth angular rotations adapted from training in the SAC model helped in keeping the awareness of its current position. The models performed better in proximity to the target, benefiting from reward adjustments for close distances as discussed in Section 6.2.3.



(a) Feature tracking with low resolution input. (b) Feature tracking with high resolution input.

Figure 7.4: Flying with OpenVINS in an industrial docking bay scene.

Experiments with the quality of the image showed that feature tracking benefits from having a higher resolution and higher quality picture, directly affecting the status of visual odometry[2]. The quality of the image was regulated through a compression parameter for the JPG encoding. The tolerance for rotational movements can be pushed higher by providing better inputs and enabling a higher FPS rate. During the flight depicted in Figure 7.4a, the agent had occasional issues with feature detection. At the same time, the problems were not reproduced in the case of the flight with a better quality camera input shown in Figure 7.4b.

7.2.2 Evaluation Summary.

For the same reward function and observations, SAC outperformed the PPO algorithm in the precision of the control task. It achieved a better quality flying trajectory and smoother motion without accidental high angular velocity changes resulting in more stable visual odometry tracking. However, large updates of the policy performed by SAC mean that it is more dependent on encountering the correct sequence of action. Multiple training sessions outlined that SAC can show a consistently bad training curve before it finds the high-reward sequence. PPO utilized more agents simultaneously, therefore the model had more experiences to perform gradual and stable updates. At the same time, using PPO for the tuning of the reward function proved effective, as it takes significantly less time to train the model on the hardware platform used during the experiments.

[2]For image quality experiments, we used the SAC model.

Chapter 8

Conclusion

In this thesis, we introduced a Unity-based simulation environment tailored for drone applications, integrating multiple software frameworks into a unified toolkit. Section 1.1 compared existing simulators, examining their physics, rendering engines, and reinforcement learning (RL) capabilities, establishing the motivation for adapting a novel simulation environment. Chapter 2 provides an introduction to Unity-specific simulation aspects and terminology, emphasizing their relevance in the unmanned aerial vehicle (UAV) domain. The discussion outlines potential challenges and proposed solutions for transitioning from other drone simulation environments, notably the MRS UAV system.

Our toolkit facilitates the swift setup of Unity scenes, whether custom-created or imported as assets (Section 2.1.6), for comprehensive testing of UAV-related scenarios. Additionally, the incorporation of new drone models through URDF format files is detailed in Section 2.2.

Chapter 3 showcases a bidirectional connection between Unity and the Robot Operating System (ROS), enabling the adaptation of existing ROS stacks to the new simulation environment. A critical use case involving the transmission of sensor data from the drone to ROS, specifically the Inertial Measurement Unit (IMU) and RGB camera for visual odometry calculation (Section 3.4), is highlighted.

The drone control system, implemented in C# and integrated with the Unity physics engine, enables realistic simulations of various control tasks. Chapter 4 elucidates the theoretical foundations behind the implementation, showcasing its adaptability to references from Unity scripts or ROS nodes.

A notable contribution of this thesis is the examination of the Unity ML-Agents framework in Chapter 5. The framework's natural integration into the Unity workflow sets it apart from alternatives, leading to the implementation of the DroneAgent behavior. This behavior acts as a wrapper for the physical simulation of the drone, designed explicitly for RL training. ML-Agents also contains a wrapper for training in the (now deprecated) OpenAI Gym, which was in the end not used for the actual training for the reasons explained in Section 5.3.

The practical application of all modular parts of the thesis into a position control RL task provides a showcase of the toolkit and tests the implementation at the same time. Training an RL model is sensitive and involves multiple iterations of hyperparameter and reward signal tuning. An extensive analysis of the available parameters and options for setting up the environment is provided in Chapter 6. The observations of the training process are discussed in Chapter 7 to understand the advantages and disadvantages of PPO and SAC algorithms. The agents' performance is assessed in an unseen environment, incorporating visual odometry calculations using OpenVINS as a final evaluation test. The evaluation of the trained model concluded that while training a PPO model can be fairly easy, the higher quality and precision of the policy is achieved by SAC due to its thorough optimization of Q-functions.

8.1 Future Improvements, Project Evolution

The framework introduced in this thesis has accomplished its predetermined objectives, providing a new drone simulation environment. At the same time, its deliberate inherent modular design and adaptability leave a potential for extensive expansion, presenting multiple areas for future developments and enhancements. Among multiple areas that could be explored:

- The simulator could be included as another MRS simulation option. That would enable more developers to contribute to the research using the toolkit.
- For area-specific simulations, the toolkit can potentially include a set of predefined custom-designed assets suitable for quick RL tests. That would make the testing of research theories simpler and even more convenient.
- RL learning can be expanded and tested on visual-based tasks to utilize Unity's diverse rendering features to a higher degree. That would provide an additional benchmark for the speed of the ROS-Unity communication.
- For more stable odometry-aware flights, we can explore further fine-tuning of the rewards and hyperparameters that would help increase the quality of the flying trajectories and even more efficiently remove excessive movements.
- The toolkit can also be expanded to not only drone-specific simulations but also other types of robots or UAVs. The explored setup of the training would be similar.

References

[1] A. T. Azar, M. Z. Sardar, S. Ahmed, A. E. Hassanien, and N. A. Kamal, “Autonomous robot navigation and exploration using deep reinforcement learning with gazebo and ros,” in *Proceedings of the 9th International Conference on Advanced Intelligent Systems and Informatics 2023*, Cham: Springer Nature Switzerland, 2023, pp. 287–299.

[2] A. Jordan, L. Gregory, and Y. Holly, “Comparing performance between different implementations of ros for unity,” 2023.

[3] P. Petracek, V. Kratky, T. Baca, M. Petrlik, and M. Saska, “New era in cultural heritage preservation: Cooperative aerial autonomy for fast digitalization of difficult-to-access interiors of historical monuments,” *IEEE Robotics and Automation Magazine*, pp. 2–19, 2023.

[4] M. Petrlík, P. Petráček, V. Krátký, *et al.*, “Uavs beneath the surface: Cooperative autonomy for subterranean search and rescue in darpa subt,” *Field Robotics*, vol. 3, pp. 1–68, Jan. 2023.

[5] J. Saunders, S. Saeedi, and W. Lil, “Parallel reinforcement learning simulation for visual quadrotor navigation,” in *2023 IEEE International Conference on Robotics and Automation (ICRA)*, 2023, pp. 1357–1363.

[6] Y. Savid, R. Mahmoudi, R. Maskeliunas, and R. Damaševičius, “Simulated autonomous driving using reinforcement learning: A comparative study on unity’s ml-agents framework,” *Information*, vol. 14, p. 290, May 2023.

[7] Y. Song, A. Romero, M. Müller, V. Koltun, and D. Scaramuzza, “Reaching the limit in autonomous racing: Optimal control versus reinforcement learning,” *Science Robotics*, vol. 8, no. 82, Sep. 2023.

[8] “Unity user manual 2022.3 (lts).” (2023), [Online]. Available: `https://docs.unity3d.com/Manual/UnityManual.html` (visited on 12/21/2023).

[9] “Worldbuilding in the unity editor.” (2023), [Online]. Available: `https://unity.com/features/probuilder`.

[10] J. Platt and K. Ricks, “Comparative analysis of ros-unity3d and ros-gazebo for mobile ground robot simulation,” *Journal of Intelligent Robotic Systems*, vol. 106, no. 4, 2022.

[11] “Unity robotics hub.” (2022), [Online]. Available: `https://github.com/Unity-Technologies/Unity-Robotics-Hub/tree/main`.

[12] “Unity technologies.” (2022), [Online]. Available: `https://unity-technologies.github.io/ml-agents/`.

[13] T. Baca, M. Petrlik, M. Vrba, *et al.*, “The MRS UAV System: Pushing the Frontiers of Reproducible Research, Real-world Deployment, and Education with Autonomous Unmanned Aerial Vehicles,” *Journal of Intelligent & Robotic Systems*, vol. 102, no. 26, pp. 1–28, 1 May 2021.

[14] “Dream forest tree.” (2021), [Online]. Available: `https://assetstore.unity.com/packages/3d/vegetation/trees/dream-forest-tree-105297`.

[15] S. Krishnan, B. Boroujerdian, W. Fu, A. Faust, and V. Janapa Reddi, “Air learning: A deep reinforcement learning gym for autonomous aerial robot visual navigation,” *Machine Learning*, vol. 110, pp. 1–40, Sep. 2021.

[16] V. Krátký, P. Petráček, T. Báča, and M. Saska, “An autonomous unmanned aerial vehicle system for fast exploration of large complex indoor environments,” *Journal of Field Robotics*, vol. 38, May 2021.

[17] P. Petráček, V. Krátký, M. Petrlík, T. Báča, R. Kratochvil, and M. Saska, “Large-scale exploration of cave environments by unmanned aerial vehicles,” *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, Jul. 2021.

[18] Y. Song, S. Naji, E. Kaufmann, A. Loquercio, and D. Scaramuzza, "Flightmare: A flexible quadrotor simulator," in *Proceedings of the 2020 Conference on Robot Learning*, 2021, pp. 1147–1157.

[19] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. M. Bayen, and Y. Wu, "The surprising effectiveness of ppo in cooperative multi-agent games," in *Neural Information Processing Systems*, 2021.

[20] "Coppeliasim reinforcement learning." (2020), [Online]. Available: `https://github.com/moliqingwa/coppeliasim_deeprl` (visited on 12/26/2023).

[21] "Openai baselines." (2020), [Online]. Available: `https://github.com/openai/baselines`.

[22] M. Petrlík, T. Báča, D. Hert, M. Vrba, T. Krajník, and M. Saska, "A robust uav system for operations in a constrained environment," *IEEE Robotics and Automation Letters*, vol. PP, pp. 1–1, Feb. 2020.

[23] "Sdformat." (2020), [Online]. Available: `http://sdformat.org/`.

[24] S. James, M. Freese, and A. J. Davison, "Pyrep: Bringing v-rep to deep robot learning," *ArXiv*, vol. abs/1906.11176, 2019.

[25] N. G. Lopez, Y. L. E. Nuin, E. B. Moral, *et al.*, "Gym-gazebo2, a toolkit for reinforcement learning using ros 2 and gazebo," *ArXiv*, vol. abs/1903.06278, 2019.

[26] M. Faessler, A. Franchi, and D. Scaramuzza, "Differential flatness of quadrotor dynamics subject to rotor drag for accurate tracking of high-speed trajectories," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, 620–626, Apr. 2018.

[27] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning*, J. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 1861–1870.

[28] A. Juliani, V. Berges, E. Vckay, *et al.*, "Unity: A general platform for intelligent agents," *CoRR*, vol. abs/1809.02627, 2018. arXiv: `1809.02627`.

[29] J. Lee, M. Grey, S. Ha, *et al.*, "Dart: Dynamic animation and robotics toolkit," *The Journal of Open Source Software*, vol. 3, p. 500, Feb. 2018.

[30] J. Luo, S. Green, P. Feghali, G. Legrady, and Ç. K. Koç, "Reinforcement learning and trustworthy autonomy," in *Cyber-Physical Systems Security*, Ç. K. Koç, Ed. Cham: Springer International Publishing, 2018, pp. 191–217.

[31] D. Thul, L. Ladický, S. Jeong, and M. Pollefeys, "Approximate convex decomposition and transfer for animated meshes," *ACM Transactions on Graphics*, vol. 37, pp. 1–10, Dec. 2018.

[32] "Ros tcp connector." (2017), [Online]. Available: `https://github.com/Unity-Technologies/ROS-TCP-Connector` (visited on 12/10/2023).

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms.," *CoRR*, vol. abs/1707.06347, 2017.

[34] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Airsim: High-fidelity visual and physical simulation for autonomous vehicles," in *International Symposium on Field and Service Robotics*, 2017.

[35] F. Furrer, M. Burri, M. Achtelik, and R. Siegwart, "Rotors – a modular gazebo mav simulator framework," *Studies in Computational Intelligence*, vol. 625, pp. 595–625, Jan. 2016.

[36] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the openai gym for robotics: A toolkit for reinforcement learning using ros and gazebo," *ArXiv*, vol. abs/1608.05742, 2016.

[37] H. V. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *AAAI Conference on Artificial Intelligence*, 2015.

[38] T. P. Lillicrap, J. J. Hunt, A. Pritzel, *et al.*, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[39] F. Messaoudi, G. Simon, and A. Ksentini, “Dissecting games engines: The case of unity3d,” in *2015 International Workshop on Network and Systems Support for Games (NetGames)*, IEEE, 2015, pp. 1–6.

[40] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, “Trust region policy optimization,” in *Proceedings of the 32nd International Conference on Machine Learning*, F. Bach and D. Blei, Eds., ser. Proceedings of Machine Learning Research, vol. 37, Lille, France: PMLR, 2015, pp. 1889–1897.

[41] E. Rohmer, S. Singh, and M. Freese, “V-rep: A versatile and scalable robot simulation framework,” *Proceedings of the ... IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1321–1326, Nov. 2013.

[42] M. A. Sherman, A. Seth, and S. L. Delp, “Simbody: Multibody dynamics for biomedical research,” *Procedia IUTAM*, vol. 2, pp. 241–261, 2011, IUTAM Symposium on Human Body Dynamics.

[43] T. Lee, M. Leok, and N. H. McClamroch, “Geometric tracking control of a quadrotor uav on se(3),” in *49th IEEE Conference on Decision and Control (CDC)*, 2010, pp. 5420–5425.

[44] A. Boeing and T. Braunl, “Evaluation of real-time physics simulation systems,” in *Conference on Computer Graphics and Interactive Techniques in Australasia and Southeast Asia*, Dec. 2007, pp. 281–288.

[45] “Open dynamics engine.” (2004), [Online]. Available: `https://www.ode.org/`.

Appendix A

The reward coefficients used in training for relative scaling of its modular parts:

Reward Name	**Reward Coefficient**	**Assignment Period**
Distance Reward	1	Every *FixedUpdate*
Velocity Reward	0.5	Every *FixedUpdate*
High acceleration penalty	-0.01	Every *FixedUpdate*
Angular velocity stability	0.005	Every *FixedUpdate*
Final positive reward	100	Maximum once per episode
Final negative reward	-100	Maximum once per episode

Table 1: Relative reward scaling coefficients.

The full version of the configuration file used for the training of the RL task with the PPO algorithm described in Section 6.3:

Parameter	**Value**
trainer_type	ppo
keep_checkpoints	5
checkpoint_interval	500000
max_steps	3000000
time_horizon	64
summary_freq	10000
threaded	true
batch_size	2048
buffer_size	20480
learning_rate	0.0005
beta	0.001
beta_schedule	linear
epsilon	0.2
epsilon_schedule	linear
lambd	0.95
num_epoch	7
learning_rate_schedule	linear
normalize	false
hidden_units	128
num_layers	2
goal_conditioning_type	hyper
deterministic	false
gamma	0.99
strength	1.0

Table 2: PPO algorithm training configuration file.

The full version of the configuration file used for the training of the RL task with the SAC algorithm described in Section 6.3:

Parameter	**Value**
trainer_type	sac
keep_checkpoints	5
checkpoint_interval	10000
max_steps	100000
time_horizon	64
summary_freq	1000
threaded	true
batch_size	500
buffer_size	500000
learning_rate	0.001
buffer_init_steps	0.001
save_replay_buffer	false
tau	0.005
learning_rate_schedule	constant
normalize	false
hidden_units	128
num_layers	2
goal_conditioning_type	hyper
deterministic	false
gamma	0.99
strength	1.0

Table 3: SAC algorithm training configuration file.