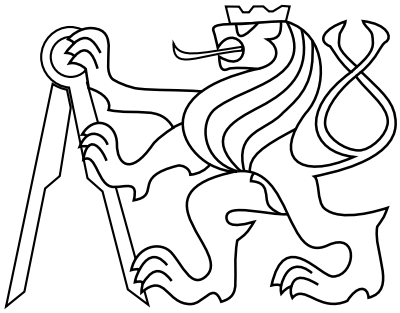
Czech Technical University in Prague

Faculty of Electrical Engineering
Department of Cybernetics
Multi-robot Systems



Drone Simulation Using Unreal Engine

Bachelor's Thesis

Jakub Jirkal

Prague, May 2023

Study programme: Open Informatics
Specialisation: Artificial Intelligence and Computer Science

Supervisor: Ing. Vojtěch Vonásek, Ph.D.

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jirkal Jakub** Personal ID number: **492212**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Drone Simulation Using Unreal Engine

Bachelor’s thesis title in Czech:

Využití nástroje Unreal Engine pro simulaci helikoptér

Guidelines:

1. Get familiar with Unreal Engine (version 5) [2] (UE5), study current topics in the robotic simulations [1,3].
2. Design a server-client system for simulation of drone flight in the virtual environment of UE5, where a server (virtual environment provided by UE5) provides data to the client (e.g. a user that wants to control the drone). The simulation should support managing multiple clients, should provide data from at least lidar (or similar) sensor and RGB (or RGB/D) camera. Design an efficient (fast, reliable) communication between the UE5 and the clients. The supposed usage of the simulation is mostly for providing realisic visualization in offline learning tools (e.g. for evoluationary robotics, reinforcement learning). The dynamics/kinematics of the drones is not important and can be assumed that it will be provided by an external library.
3. Implement the whole system (in c++), use state-of-the-art libraries for supporting functions (e.g. for communication). Design (model) virtual environments with indoor and outdoor scenarios.
4. Design and implement use-case for using the simulation in a reinforcement learning task [4]. Consider tasks that rely mostly on visual data (e.g. collision avoidance, flight to a position, formation following, etc.). Use a state-of-the-art library for reinforcement learning (e.g. using framework OpenAI).
5. (Optional) Investigate how to implement additional features: creating of drones/agents on the fly, reading mesh data (or bounding boxes) from UE5. Implement them if possible.
6. (Optional) Use dynamic model to simulate realistic flight of the drone. Use an external library for it (will be provided by the supervisor).

Bibliography / sources:

[1] J. Collins, S. Chand, A. Vanderkop and D. Howard, "A Review of Physics Simulators for Robotic Applications," in IEEE Access, vol. 9, pp. 51416-51431, 2021, doi: 10.1109/ACCESS.2021.3068769.
[2] Unreal engine 5, https://www.unrealengine.com/en-US/unreal-engine-5.
[3] Staranowicz, Aaron, and Gian Luca Mariottini. "A survey and comparison of commercial and open-source robotic simulator software." Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments. 2011.
[4] Arulkumaran, Kai, et al. "Deep reinforcement learning: A brief survey." IEEE Signal Processing Magazine 34.6 (2017): 26-38.

Name and workplace of bachelor's thesis supervisor:

Ing. Vojtěch Vonásek, Ph.D. Multi-robot Systems FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **24.01.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Vojtěch Vonásek, Ph.D.	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor'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 bachelor's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Acknowledgments

First and foremost, I would like to express my sincere gratitude to my supervisor Ing. Vojtěch Vonásek, Ph.D. for his guidance, feedback, assistance and valuable suggestions. I would also like to thank my family and my girlfriend for their support and patience throughout my studies.

Declaration

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

Prague, May 26, 2023 Jakub Jirkal

Abstract

Drone simulators differ in many ways and only a section of them supports realistic rendering. Differences between the simulated and the real world can be critical, as just a slight inconsistency could cause the drone model trained in the simulator to crash in the real world. The game industry nowadays produces games with realistic environments using different engines, among which Unreal Engine 5 stands out the most in terms of realism. The core concept of this thesis is to use the Unreal Engine 5 as a base for a photorealistic drone simulator.

We introduced and developed a simulator based on the Unreal Engine 5 with features such as a LiDAR sensor, an RGB camera, a position/tilt sensor and multiple drone support. The simulator is divided into multiple TCP/IP servers and allows for fast and reliable communication with the clients. We benchmarked the simulator on various scenarios and compared its performance in different environments and drone counts. We trained two drone models in reinforcement learning tasks and confirmed the ability of the simulator to serve machine learning purposes.

Keywords Drone simulation, Unreal Engine 5, Realistic rendering, Reinforcement learning

Abstrakt

Simulátory dronů se rozlišují v mnoha aspektech a pouze několik z nich podporuje realistické renderování. Rozdíly mezi simulovaným a reálným světem mohou mít kritické následky, jelikož drobná nesrovnalost mezi nimi může zapříčinit havárii dronu v reálném světě s modelem natrénovaným v simulátoru. Herní průmysl produkuje hry s realistickými prostředími za pomoci různých enginů, mezi kterými z pohledu realističnosti nejvíce vyčnívá Unreal Engine 5. Základním konceptem této práce je použití Unreal Engine 5 jako základu pro fotorealistický simulátor dronů.

Představili a vyvinuli jsme simulátor založený na Unreal Engine 5. Simulátor obsahuje LiDAR sensor, RGB kameru, poziční/rotační sensor a podporu několika dronů. Simulátor je rozdělen do několika TCP/IP serverů a podporuje rychlou a spolehlivou komunikaci s klienty. Změřili jsme výkon simulátoru na několika rozlišných scénářích a výsledky porovnali na základě rozdílných prostředí a počtu dronů. Natrénovali jsme dva modely dronů v úlohách zpětnovazebného učení a potvrdili jsme schopnost simulátoru sloužit pro účely strojového učení.

Klíčová slova Simulace dronů, Unreal Engine 5, Realistické renderování, Zpětnovazebné učení

Abbreviations

API Application Programming Interface

FOV Field of View

FPS Frames Per Second

GPS Global Positioning System

GUI Graphic User Interface

IMU Inertial Measurement Unit

LiDAR Light Detection and Ranging

MRS Multi-robot Systems

RGB Red Green Blue

RGBD Red Green Blue-Depth

UAV Unmanned Aerial Vehicle

Contents

List of Figures

List of Tables

Chapter 1

Introduction

It is not always possible to operate drones in the real world. For drone model evaluation and learning purposes, it is better to use a simulation rather than the real world for numerous reasons. First, drone model failures are possible and it is better to catch them in a simulated environment, as it is cheaper and safer. Second, there can be multiple drones operated at the same time in the same environment, as simulators either allow for parallel computing or can be run side-by-side. Third, the simulators can run faster than the real world, which allows for faster evaluation and learning. Fourth, in contrast to the real world, the simulated environment allows for easy map and scenario change in a matter of seconds. The major disadvantage of simulation is the possible inconsistencies between the simulated environment and the real world, being it either non-realistic collision handling, physics simulation or rendering.

The inconsistencies between the real world and the simulated environment can be critical. For instance, imprecise collision detection in the simulator can cause the drone to crash in the real world. Likewise, realistic rendering is crucial for learning a drone model based on camera data, as just a slight inconsistency could make the learned model malfunction in the real world. Reasons described above lead to an emerging trend for realistic simulators, which can find their place in machine learning tasks.

Nowadays, the game industry produces games with immersive and realistic environments. These environments are processed and rendered in a game engine, which is the base of the game. The game engines differ in many ways, however, the Unreal Engine 5 [15] stands out in the realistic rendering field. The Unreal Engine 5 features state-of-the-art rendering capabilities, physics simulation, precise collision handling and a marketplace [18] with a rich selection of community digital assets (objects, textures, etc.) and maps. A capture of the marketplace can be seen in Fig. 1.1. The creation of the environment in the Unreal Engine is a simple task with the addition of the marketplace and allows the user to focus more on the actual drone training rather than creating the environment and the assets. The main idea of this thesis is to leverage the realistic rendering capabilities the Unreal Engine 5 offers and use the engine as a base for the drone simulator.

Several simulators with realistic rendering capabilities exist in the field of aerial robotics and will be discussed in Chapter 2. This thesis is assigned by the Multi-robot Systems (MRS) Group [23]. The group assumes the increasing usage of reinforcement learning in drone learning tasks based on camera data and plans to use the simulator proposed in this thesis in their research and as a base for students writing their thesis under the MRS Group. The group does not want to be dependent on external simulators, as their support is not guaranteed and the community is smaller compared to the Unreal Engine. Other advantages of having a custom simulator are bigger control over the simulator and the possibility to implement custom functionalities. The disadvantage of this approach is the need for people, who will be extending and maintaining the simulator.

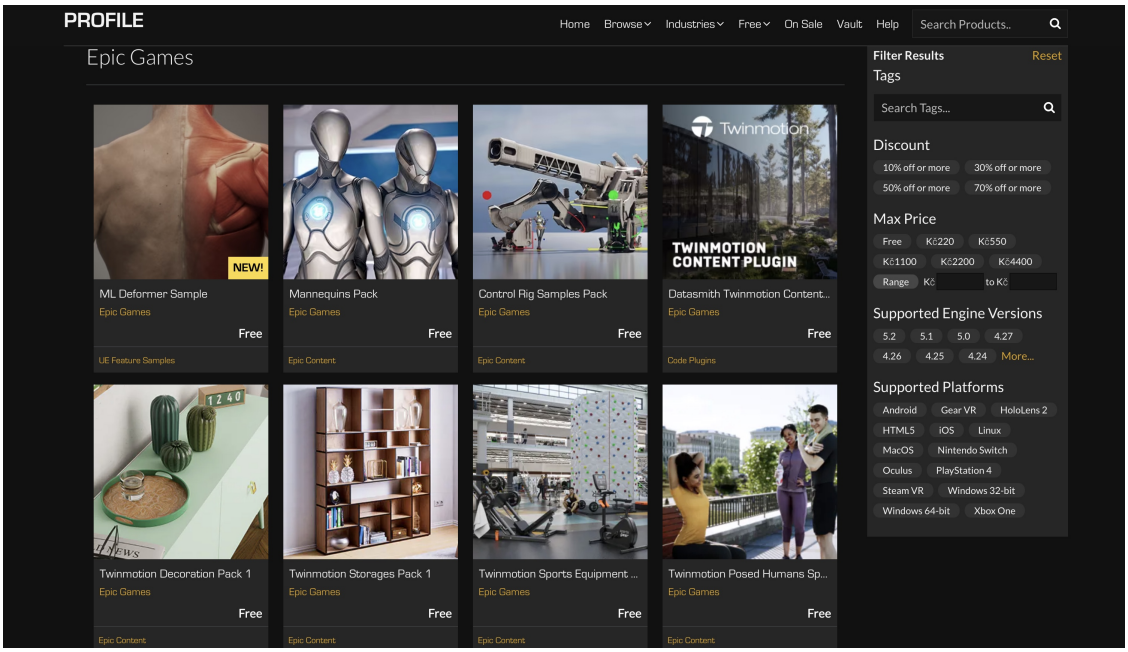


Figure 1.1: A sample of Unreal Engine Marketplace. Captured from [11].

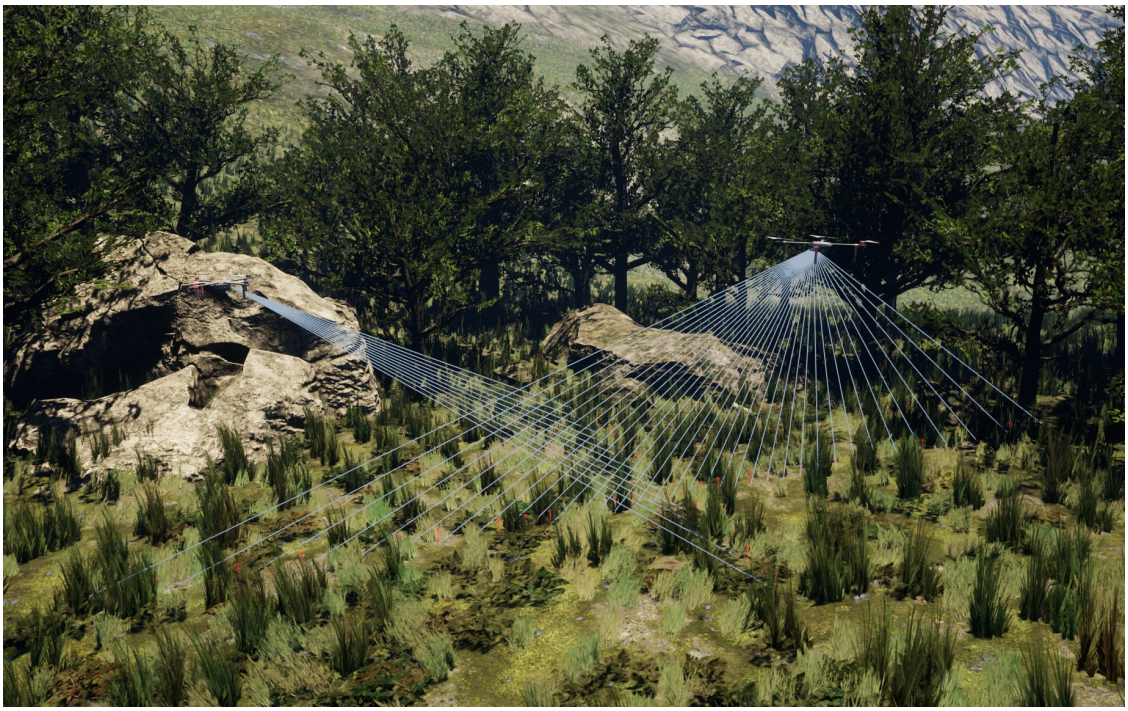


Figure 1.2: A sample of an environment and drones with LiDAR beams captured from the proposed simulator.

1.1 Goals

The goal of this thesis is to research and develop a simulator focused on realistic visualization based on Unreal Engine 5. The research includes an analysis of drone simulator components, the simulator efficiency and the possible capabilities. The simulator will expose a server to which clients can connect and operate the drones in a fast and reliable way. Collision detection will be handled appropriately and the drone will be comprised of a real-world drone model. Features such as a Light Detection and Ranging (LiDAR) sensor, an Red Green Blue (RGB) camera and a localization sensor will be present. The simulator will be able to create and remove drones on the fly. The drone dynamics/kinematics model is assumed to be provided by an external library and will not be implemented. A sample captured from the proposed simulator can be seen in Fig. 1.2.

Another goal of this thesis is to develop a client for the simulator in C++. The client will be able to operate the drone and the state of the simulator over the network. On top of the client, there will be an OpenAI Gym environment [24] implemented, which allows for easy reinforcement learning development.

The simulator will be benchmarked on various scenarios. Two reinforcement learning tasks will be prepared, learned and validated. The first task will rely on LiDAR data with a direction vector to a target and the drone will be learned to fly through an obstacle playground to the target. On the second task, the drone will be learned to avoid obstacles and fly to the target as well, with the difference that RGB camera data will be provided and the drone must find the target using the provided camera data.

Chapter 2

Related work

Several simulators exist in the field of aerial robotics. This chapter focuses on an overview of simulators, that are suitable for training and evaluating drone models. As the goal of this thesis is to research and develop a simulator focused on realistic visualization, we will mainly be discussing the simulators, that support such capabilities.

In 2011, there was a survey paper of both open-source and commercial robotic simulators created [46]. The paper displays signs of being antiquated, as some of the information is outdated, for instance, according to [3], Webots is open-source, despite authors claiming it to be commercial. Due to this fact, we will discuss eligible simulators from this paper, but will base the information about them on recent sources.

In [39], authors dedicated a chapter specifically for aerial robotics. They provided a table of feature comparisons of aerial robotics simulators which shows that the differences between such simulators are noticeable. The authors state that AirSim and Flightmare support realistic rendering and thus will be discussed more in-depth.

There are a few features that the simulator must have to be able to simulate drones. First of all, it must work in 3D space. Some simulators work only in a 2D space, but the third dimension is essential for providing a realistic environment. Secondly, fast collision detection has to be supported. This is crucial, as crashing a drone in the real world can be an expensive mistake. At last, some sensors implementation must be present — usually a combination of a camera, LiDAR, Inertial Measurement Unit (IMU) and Global Positioning System (GPS). Without the sensors, the drone is "blind" and has no way of navigating in the space.

ROS (Robot Operating System) is an open-source framework, featuring a set of tools and libraries for developing robotics software [47]. It aims to provide a common interface for building and testing various robots, regardless of their hardware and software. Out of the box, the ROS is not capable of simulating drones, but many of the simulators listed below implement ROS interface and allow for using a unified communication protocol.

Gazebo is a popular open-source multi robot simulator [39, 49]. The rendering engine is based on OpenGL, which is an Application Programming Interface (API) to graphics hardware, enabling 2D and 3D graphics rendering [48]. As authors of [39] state, Gazebo does not provide realistic rendering. Drone dynamics is handled by the Open Dynamics Engine [30]. Sensor-wise, the simulator implements an Red Green Blue-Depth (RGBD) camera, an IMU, a LiDAR and a GPS. Gazebo is commonly used in combination with ROS and is an industry standard.

Webots is an open-source 3D mobile robots simulation software [3, 50] developed by Cyberbotics Ltd. It features a set of predefined robots, including DJI Mavic 2 PRO quadcopter [2]. Webots offers a LiDAR, IMU, GPS and camera sensor, but it does not offer a realistic environment rendering (see Fig. 2.1) [39].

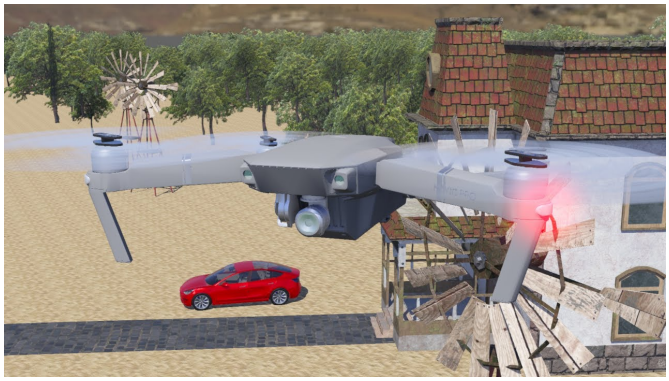


Figure 2.1: A sample of a drone and an environment in Webots. Taken from a thumbnail of a video [36], referenced at [2].

jMAVSim is an open-source multirotor simulator written in Java [1], focusing mainly on the simulation of drone dynamics. It supports a MAVLink protocol, which is a lightweight messaging protocol for communicating with drones [22]. The rendering of the simulator is based on Java 3D library [26] and does not provide realistic rendering, as can be seen in Fig. 2.2. jMAVSim implements an IMU and a GPS and does not offer a LiDAR and a camera in contrast to other simulators. As per [1], the project is now discontinued.

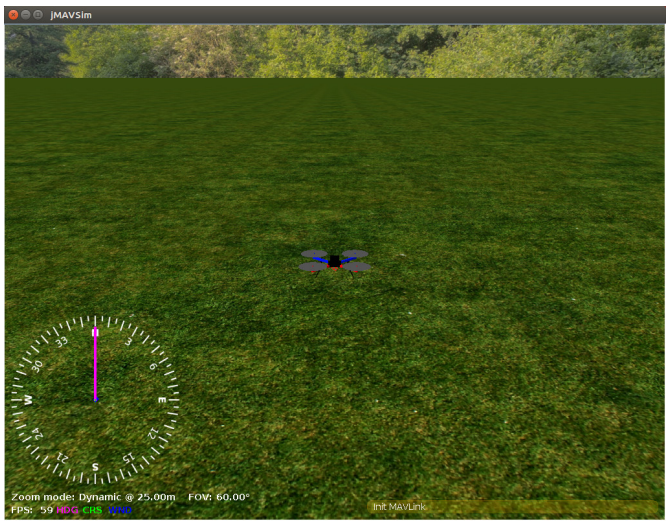


Figure 2.2: A sample of a drone and an environment in jMAVSim. Taken from [28].

MATLAB with Simulink establishes a general-purpose simulation environment with a set of toolboxes and algorithms [31]. The UAV Toolbox includes a framework with Unmanned Aerial Vehicle (UAV) models and environment based on Unreal Engine [32], which provides realistic rendering. A sample of the environment can be seen in Fig. 2.3. The framework features a predefined scene, a quadrotor model, an RGBD camera and a LiDAR sensor. The simulator is tightly incorporated into the Matlab ecosystem and does not provide a universal API.

Figure 2.3: A sample of a drone and an environment in MATLAB with Simulink. Taken from [33].

AirSim is an open-source simulator [43] developed by Microsoft. The simulator is based on Unreal Engine 4 (which is now a legacy version) and provides realistic rendering and collision detection. It features a GPS, an RGBD camera, a LiDAR and an IMU sensors and provides a drone model with dynamics out of the box. A snapshot from AirSim with various cameras and their modes can be seen in Fig. 2.4. As authors of [39] state, AirSim is resource-intensive and requires powerful hardware.

Figure 2.4: A snapshot from AirSim. Taken from [43].

Flightmare is an open-source flexible modular quadrotor simulator [41]. Its central principle is the decoupling of a rendering and physics engine, which brings the benefit of letting the end user to decide, whether he needs both of the engines running. The system overview and decoupling schema of Flightmare can be seen in Fig. 2.5. As the authors show, turning off the rendering engine increases performance — they achieved 230 Hz with a rendering simulation in comparison with 200,000 Hz with a physics simulation. The rendering engine is based on Unity [34], which is a popular 3D game engine. Flightmare offers three variants of quadrotor dynamics, the implementation of an RGBD camera, IMU sensor and an API for extracting point clouds of the environment. In contrast to other simulators, it does not offer a LiDAR [39].

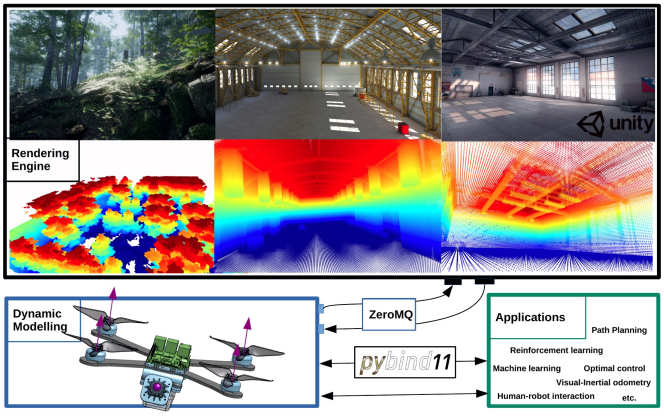


Figure 2.5: A system overview of Flightmare. Taken from [41].

FlightGoggles is an open-source simulator based on Unity [42]. Its primary concept is the utilization of photogrammetry — an approach for generating 3D objects and environments from photos [51]. Using this method, real world environments can be scanned, processed into a simulation environment and used for simulation purposes. A snapshot of such an environment can be seen in Fig. 2.6. Similarly to Flightmare, it features a modular architecture, which allows for running rendering and collision detection separately. FlightGoggles includes an RGBD camera, an IMU and a LiDAR.

Figure 2.6: A snapshot of a rendering of an environment in FlightGoggles. Taken from [42].

We have discussed the popular and state-of-the-art simulators currently available. The emerging trend for simulators with realistic rendering is noticeable and can be interpreted as a consequence of the need for realistic data for model training and the continuous availability of stronger hardware.

Chapter 3

Problem overview

In the previous chapter, we have discussed the features of currently available drone simulators. As we can see, many simulators implement a set of common, base features — mainly collision detection, a camera, an IMU sensor, a LiDAR sensor and a GPS sensor. Combined with the required features discussed in Chapter 1, this forms a set of components, that the simulator will present.

In this chapter, we will analyze the elements, that are crucial for the drone simulator. We will discuss the approaches, that can be taken to simulate such elements realistically. At last, we will look into the communication with the drones and the environment.

3.1 Movement and collisions

In the real world, drones fly by gaining thrust by spinning propellors attached to motors on their body. When the propellors spin at the same rate, the drone flies upwards. A difference in propellors spin rate causes the drone to rotate and fly in different directions.

As was already stated in Chapter 1, for the purposes of this simulator, we will not be taking drone dynamics/kinematics into account. Instead, we can move the drone by simply teleporting it across the map. This behavior brings a benefit: as we leave the drone dynamics/kinematics on the client side, there can be any motion model integrated. A smooth transition is then achievable by teleporting the drone in small steps, provided that the teleportation is a fast enough operation. If we want to support drone dynamics in the future, we have two ways of doing so: (i): implement the dynamics as a module, that will stand between the simulator and a client (middleware [38]). This approach is inspired by Flightmare [41], with the difference that the rendering engine must be running nevertheless; and (ii): implement the dynamics as a plugin inside of the simulator.

For this approach to work, we need to check for collisions. As we will be teleporting between two points, we simply need to check that the drone will not collide with anything on the line segment between those two points. If a collision will be present, we need to stop the drone at the collision point. This can be achieved by iteratively calculating multiple positions between the two points and for each of the positions, check that the drone does collide with anything (see Fig. 3.1). We will discuss the implementation more in detail in Chapter 4, which is dedicated to the simulator itself, as this type of transition is something that is baked right into the Unreal Engine.

At one request, we should be able to either change location, change rotation or change both. Changing both location and rotation in one request is especially useful, as drones in the real world usually do the same.

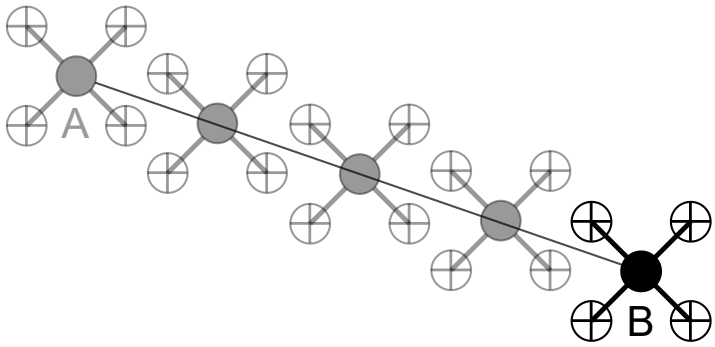


Figure 3.1: A sample a drone transition from point A to B in 2D space. The full line represents the transition line segment. Greyed-out drones represent iterations of the transition.

A similar path can be taken for rotating the drone in any direction. For the collision detection matter, we iteratively calculate multiple rotations in the direction of goal rotation and check for impacts in each of those.

3.2 Sensors

GPS module provides geolocation to the drone. As authors of [44] state, the state-of-the-art Differential GPS method achieves approximately 15 cm error. This error is inadequate for drone orientation, as a few centimeters error could cause a crash. The GPS is prone to error and might not even work in certain conditions — indoors, environments with a lot of interference, etc.

As the environment will be simulated, we should be able to retrieve the exact location of the drone in it. To make the simulation more realistic, we can introduce error generation to the position retrieving mechanism (similar to AirSim [43]).

IMU sensor is primarily used for determining acceleration, angular velocity and rotation. It mainly consists of two components: (i): accelerometers — used for obtaining linear acceleration, and pitch and roll rotation angles; and (ii): gyroscopes — used for obtaining angular velocity and all three rotation anglers. Some modifications of the IMU also provide a magnetometer, which provides a yaw rotation angle. Measurements from mentioned components are combined and provide calibrated output [45].

As the linear and angular velocity contributes mainly to the drone dynamics in the context of the drone simulator and we will not be dealing with the dynamics, we can omit their implementations from the simulator. The rotation that the IMU sensor provides us is important and similar to GPS, we should be able to retrieve it easily from the environment.

LiDAR sensor is responsible for retrieving the distance between it and obstacles. In short, it emits a laser beam and calculates the distance to the obstacle in the direction it was emitted from the time it takes to deflect back. LiDARs have different maximal ranges of operation (how far they "see"), counts of the beams and ranges of a shot (under what angles they "see", both 2D and 3D). The sensor itself can be mounted to various positions on the drone.

The simulated LiDAR should support all of the abilities and properties described above. We should be able to send a beam in a particular direction from the LiDAR, as that is something fairly common in a game engine — e.g. shooting a bullet in a videogame. The Unreal Engine supports this and even supports retrieving the distance out of the box. We will discuss the implementation in Chapter 4.

Camera provides video and photo input for the drone operator or for the model that operates the drone. The cameras attached to drones can have depth-sensing, diverse resolution, Field of View (FOV) angle, color space, exposure settings etc. Similar to LiDAR, the camera can be mounted to various positions on the drone.

As discussed in Chapter 1, the camera should support an RGB color space without depth sensing. Configurable FOV angle is an important setting, as it is commonly diverse in drone cameras and should be implemented. It is crucial to have a camera input in a certain orientation depending on the case and the mounting of the camera must be supported as well.

3.3 Controls

There are three ways of controlling the drone in the real world: (i): a drone is controlled by software, that exists on the board attached to the drone; (ii): a drone is operated by a human from a distance using a controller; and (iii): a drone is remotely controlled by software on another machine.

We will focus on remotely controlling the drone, as it gives us much more flexibility. This way, we can dynamically change its behavior from remote without reloading the drone. Also, it gives us the benefit of controlling the drone from multiple endpoints. We can define various commands (e.g. move, rotate, return lidar data) and send them directly to the drone.

Chapter 4

UEds simulator

The UEds (Unreal Engine drone simulator) is the proposed drone simulator developed in Unreal Engine 5. It consists of source code and graphical assets. In this chapter, we will be discussing the architecture and the capabilities of the proposed simulator. We will also talk about the performance limitations and their dependent root causes. The complications of developing a simulator in Unreal Engine will be explained and clarified.

As stated before, the main focus of this simulator is the utilization of Unreal Engine 5 realistic visualization and collision handling. Another goal is to support multiple clients, which allows the training and evaluation programs to run in parallel and thus those programs can run faster than in a real world environment.

4.1 Architecture

The simulator is composed of three main parts: the *environment*, the *game mode* and the *drones*.

The environment is an abstraction of Unreal Engine's world, map and physics and lightning simulation. It holds references to the objects that exist in it (e.g. walls, trees, terrain, drones, skylight) and acts as a mediator between those objects — e.g. a tree stands on the terrain, a light from the skylight falls on a tree and casts a show onto the terrain (lightning simulation), a drone crashes into a wall (physics simulation). Objects can interact with the environment in various ways depending on their type — e.g. cast shadow, collide.

The game mode is the main control unit of the simulator. It exists on top of the environment and is able to manipulate it. In UEds, it holds references to all drones that exist in the environment and allows for spawning and removing them on demand. Additionally, the game mode is able to extract Frames Per Second (FPS) from the environment, which will become convenient for benchmarking. As the game mode allows for environment manipulation, it could be further used for changing the map, setting skylight/sun angle (time and cardinal directions manipulation), dynamically spawning other objects, etc.

The drone is an object in the environment simulating a real world drone. It is currently comprised of a model of a Tarot 650 frame, motors and propellors and can be switched to any other drone model using an FBX format [12]. A LiDAR and an RGB camera are attached to the drone object and can be manipulated by it. The drone is able to freely move and rotate in the environment, can collide with it and registers the collision point in the environment when it happens. As there can be many drones in the environment, they can not affect each other — they are invisible to each other. This behavior is specifically configured and can be changed.

Both the game mode and the drone objects are holding a reference to their custom TCP server (see Fig. 4.1). This way, we can utilize the larger communication throughput

for the game mode and each drone in comparison to one common server, which would pass requests to the game mode and the drones. Also, this corresponds to the real world, where each drone is being operated by its separate controller. Each server waits for incoming connections and requests. Upon each request, the server parses the request, routes it and executes the desired operation on the game mode or drone. We will discuss the implementations more in Sections 4.2 and 4.7.

At the start of the simulator, the game mode server is started and waits for requests. The start of the simulator does not spawn any drones and thus the game mode server is the only running server in the beginning. On a request for a drone spawn, the game mode creates a drone instance, starts the drone server and holds a reference to the drone in case the user wants to remove the drone.

The simulator can run on top of any Unreal Engine map and environment, as there are no configurations that bound the simulator logic to a specific map or environment. It exists on its own in any environment and interacts with it using Unreal Engine common methods. The only required steps are: (i): import the code of the UEds simulator; and (ii): set the default game mode to the game mode provided by the UEds simulator.

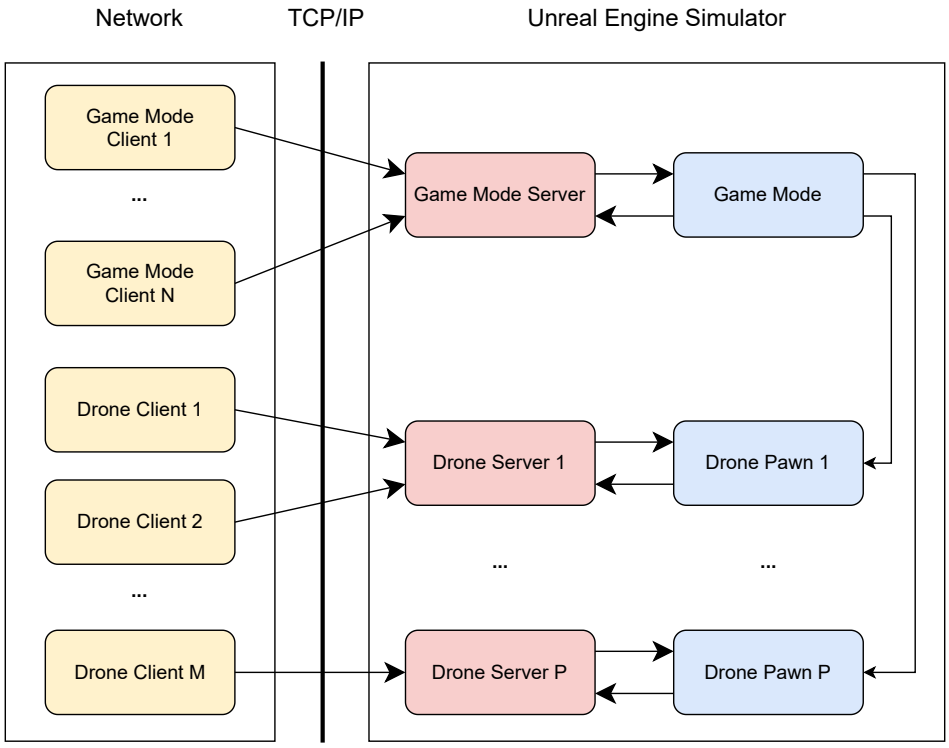


Figure 4.1: A diagram of the simulator architecture. The blue boxes represent the objects existing in the Unreal Engine environment, the red boxes represent the servers of the objects and the yellow boxes represent the clients connected to the servers. Both the objects and the servers exist under the Unreal Engine environment and the clients are connected over TCP/IP socket from the network.

4.2 Game thread

In Unreal Engine and other game engines, the environment is rendered and displayed to a user. The user sees the environment as a continuous sequence of pictures, called frames. Each second, several frames are being rendered and displayed — the metric for that is called FPS. The more FPS the game produces, the smoother it is.

During each frame render, the Unreal Engine processes rendering and post-processing operations and synchronously and serially calls *Tick* function upon each object in the environment, provided the object has the Tick function enabled. The Tick functions are called from the main game thread, which is responsible for the game state, the rendering and the user input handling. As a consequence, the operations that change the state of the objects and the environment (in our context drone changing position and rotation, game mode spawning and removing drones) must be called from the game thread.

Let us imagine a simplified version, where the engine calls the Tick function on each object during each frame render and the rest of the frame render execution time is constant. Then each frame calculation time can be expressed as

$$F = \sum_{i=1}^{N} t_i + C, \tag{4.1}$$

where F is the frame calculation time in milliseconds, N is the number of objects with the Tick function, t_i is the execution time of the Tick function of the object i in milliseconds and C is the constant execution time of the rest of the frame render. Using equation (4.1), we can calculate the FPS as

$$FPS = \frac{1000}{F} = \frac{1000}{\sum_{i=1}^{N} t_i + C}, \tag{4.2}$$

where FPS is the Frames Per Second and the rest of the variables correspond to the equation (4.1). From equation (4.2), we can denote that FPS is inversely proportional to the execution time of the Tick function.

The Tick function is the only place in the drone and the game mode objects, from where we could periodically check the server for incoming connections and requests. As we denoted, every increase in the Tick function execution time lowers the FPS of the simulator. Consequently, running the servers in the Tick function would lower the FPS and thus increase the response time of the servers. Instead, we can run the servers from other threads, that will not block the game thread. As we have shown, we can not run operations that change the state of the objects and the environment from any other thread than the game thread. Nevertheless, we can solve this issue by creating an *instruction queue*.

The instruction queue will take incoming operations from the server thread and process them in the Tick function in the game thread. On each incoming request that holds an operation that changes the state of the objects and the environment, the server pushes the operation to the instruction queue and awaits its completion. The objects will check each Tick for any operation in the instruction queue and process them if present. The figure of this flow can be seen in Fig. 4.2. The instruction queue must be thread-safe, as two different threads can access it at the same time. Here, we can utilize the Unreal Engines thread-safe queue [16] implementation.

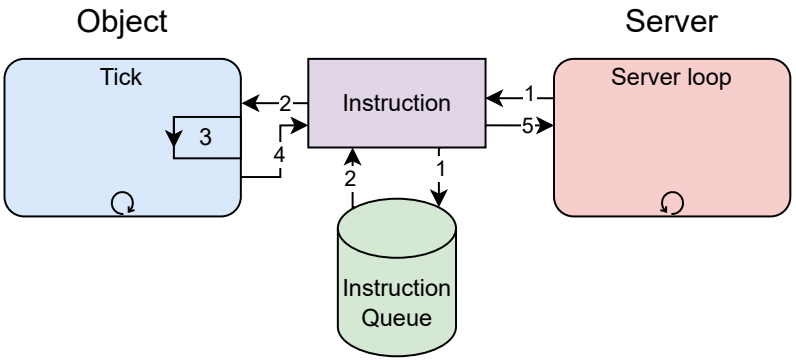


Figure 4.2: Flow of instruction processing. (1): A server adds an instruction to an instruction queue on request. (2): An object reads the instruction from the instruction queue in the Tick function. (3): The object processes the instruction. (4): The object marks the instruction as finished. (5): The server notices that the instruction is finished.

4.3 Movement and collisions

In Chapter 3, we determined that we will move and rotate the drone by teleporting it, provided that collisions will be handled appropriately. We also determined that we will change either location, rotation or both in one request. Changing the location or rotation of the drone are both operations that change the state of the drone and thus need to be run from the game thread.

The Unreal Engine provides methods, that exactly serve the purpose described above [10, 13, 14]. All mentioned methods have an optional flag for turning on and off the collision detection, which can become useful when we want to just teleport the drone to some location (e.g. at the start of some learning scenario). Upon the collision, the drone is stopped and an impact point is returned. The initial model was provided by the MRS Group and consisted of 143,876 triangles. To optimize the performance of the simulator, the model was simplified by the Unreal Engine upon the import to consist of 5,129 triangles (approximately a 96% reduction), which still results in a detailed image (see Fig. 4.6) The collision model was created from the simplified model by wrapping a convex hull around it and can be seen in Fig. 4.3.

For the purposes of the evaluation of trained drone model behavior, we need some way of tracing its path. The debug move line is an optional configuration, which draws a green line between movement points and eventually forms a trace of the drone movement. The usage of it can be seen in Chapter 6.

4.4 Position and tilt sensor

As discussed in Chapter 3, we will be simulating the GPS as a precise location retrieval and the IMU sensor as a precise rotation retrieval. Both properties can be fetched using default Unreal Engine methods [8, 9]. The location is in (X, Y, Z) format in unreal units relative to the world, where one unreal unit is equal to one centimeter in the real world The rotation is in $(pitch, yaw, roll)$ angle format relative to the world. The location and rotation retrieval are both immutable operations and thus can be run from the server thread.

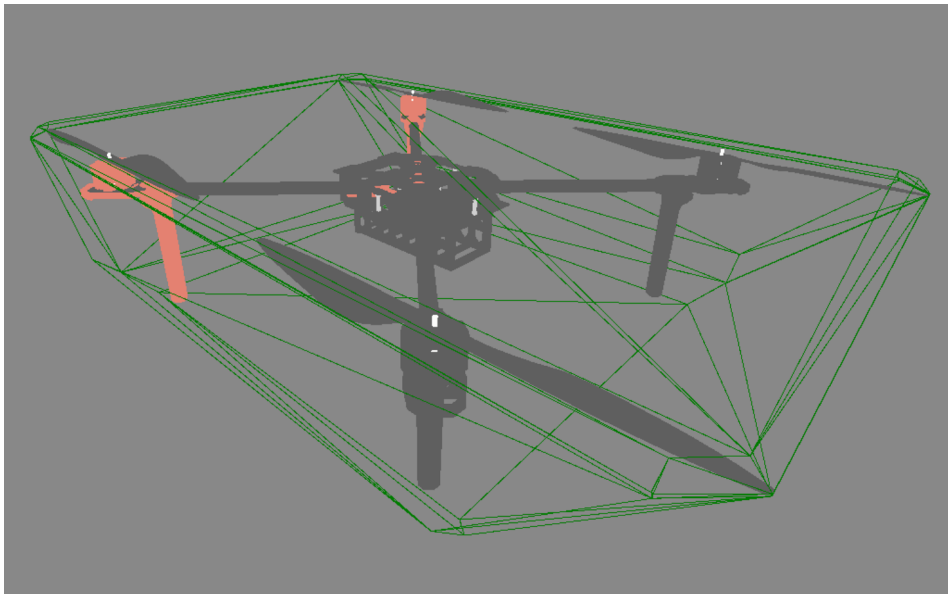


Figure 4.3: A model of Tarot 650 frame, motors, propellors and the corresponding collision model captured using Unreal Engine. The green lines represent the collision model, made of a convex hull wrapped around the model.

4.5 LiDAR

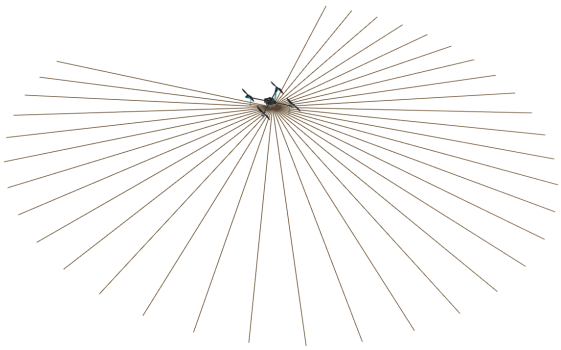
The Unreal Engine supports sending a line trace from a start point to an endpoint including checking for collisions and returning a distance from the starting point to the obstacle if the collision took a place [21]. The starting point is a center of the LiDAR, which can be configured. The endpoint is calculated as a point in the space of configurable distance from the start in the orientation of a beam. The orientation of the beam is calculated from the orientation of the lidar and the configurable FOV angle. This line trace is conducted for each of the beams and the measurements for each of the beams are then returned to the user.

The calculation of the LiDAR hits is not changing the state of the environment and can thus be run in a server thread. We also do not need to precalculate the hits, as the calculation is a fast operation and can be calculated on request. The benchmarks will be later discussed in Chapter 5.

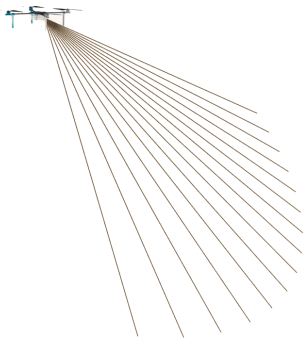
The LiDAR can be configured in several ways. First of all, the orientation and the offset of the LiDAR mounting are configurable. Second, the number of the LiDAR beams, their length (e.g. how far does the LiDAR "see") and the FOV angle are customizable. Third, for debugging purposes, the visibility of the beams is also configurable. Example configurations can be seen in Fig. 4.4.

4.6 Camera

In Unreal Engine, capturing a snapshot from a camera attached to the drone is not a trivial task. First of all, the default Unreal Engine camera component [17] does not support retrieving a snapshot and is used only as a viewport for a Graphic User Interface (GUI). Second, the rendering is being handled on a GPU and for the purpose of sending the snapshot



(a) 35 beams; 270 FOV angle, facing forward



(b) 15 beams; 90 FOV angle, facing forward and down by 35 degrees

Figure 4.4: An example of LiDAR configurations. Captured in the UEds simulator. The colors were inverted to make the beams more visible.

to the client, we need to transfer it to a CPU first. The image data will be compressed and converted to the JPG format before being sent to the client. As the camera will primarily be used for machine learning purposes, the default resolution is set to 640 x 480 px and can be further changed.

The snapshot of an environment can be retrieved using a scene capture component [19], but it needs to be projected to some render target before reading it from the CPU. For that, we can use a simple 2D render target [20], which will be used as a holder of the snapshot. The render target can be updated by the scene capture component only on demand, which will later become useful. The update of the render target changes its state and thus needs to be executed from the game thread. The flow of the camera capture can be seen in Fig. 4.5.

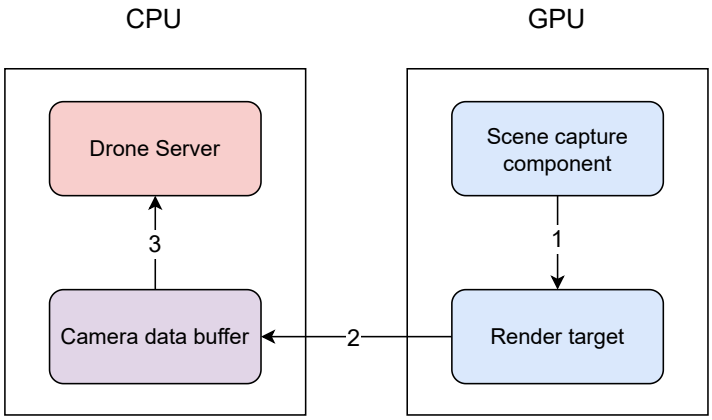


Figure 4.5: Flow of the camera capture. (1): A render target component is updated by a scene capture component. (2): A camera data buffer is filled with data from the render target. (3): The camera data buffer is accessible to a drone server.

As already stated, we need to process a transfer from the GPU to the CPU and this transfer happens when reading the binary image data of the render target. This is a computationally expensive operation, as the data need to transfer a significantly long path between the GPU and the CPU. The update of the render target from the scene capture component is

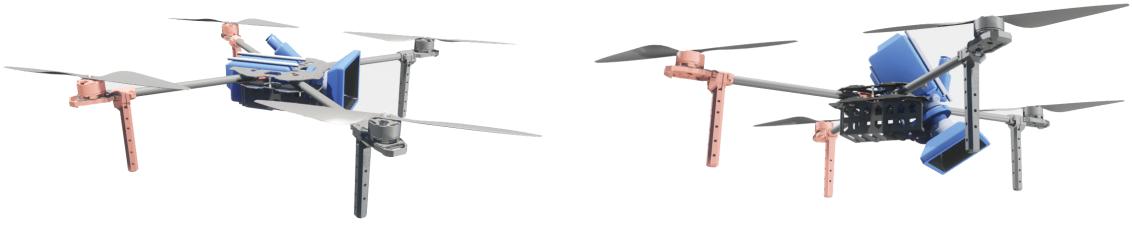
also a demanding operation. Let us imagine that we would process those operations at each Tick, even if we don't need the camera data. This introduces a computational bottleneck, which is even multiplied by the number of drones existing in the simulation.

Because of the reasons mentioned above, we need some way of turning off the camera updates (render target update and reading the render target data) if desired. We introduce three camera capture modes. Each of the camera capture modes has its pros and cons.

- *Capture all frames*
  - The mode updates the camera at each Tick and consequently slows down the game thread every Tick.
  - The image retrieval is instant, as the image data are always ready.
- *Capture on movement*
  - The mode updates the camera at each Tick if the drone moves or rotates and consequently slows down the game thread only on movement or rotation.
  - The image retrieval is instant, as the image data are always ready.
  - The mode is not suitable for dynamic environments, as it does not reflect the changes in the environment.
- *Capture on demand*
  - The mode updates the camera only on a request and thus slows down the game thread only when requested.
  - The client needs to wait for the camera update.

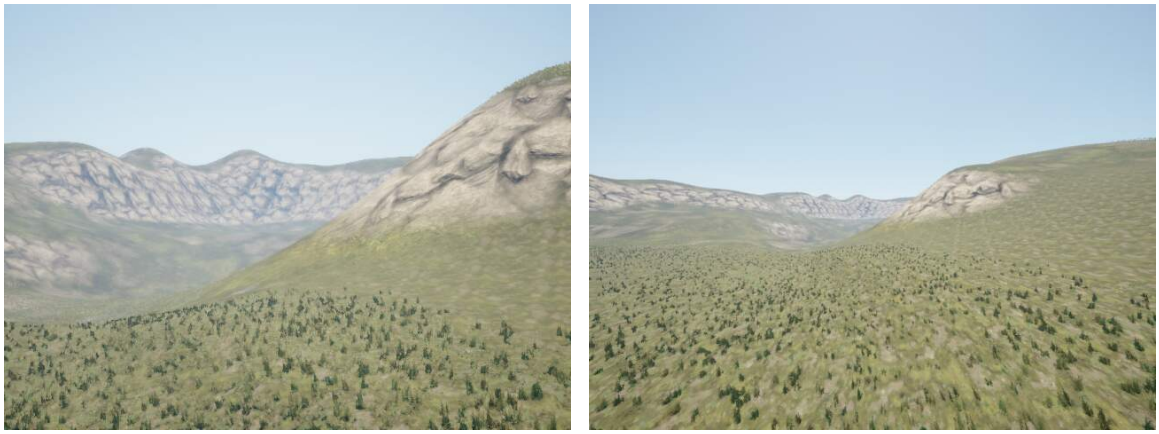
The benchmarks of the camera capture modes will be later discussed in Chapter 5.

Similar to LiDAR, the offset and the orientation of the camera can be configured. As discussed in Chapter 3, the FOV angle configuration is also present. For debugging purposes, the visibility of the scene capture component is also configurable. Example configurations of mounting can be seen in Fig. 4.6 and example images with various FOV angles can be seen in Fig. 4.7.



(a) facing forward (b) facing forward and down by 60 degrees

Figure 4.6: An example of camera configurations. Captured in the UEds simulator. The background color of the drones was modified to white for better visibility of the camera.



(a) 60 FOV angle (b) 120 FOV angle

Figure 4.7: An example of captured camera images with various FOV angles. Captured in the UEds simulator.

4.7 Communication

In Section 4.1, we discussed that both the game mode and the drones are holding a reference to their custom server. Later in Section 4.2, we have shown an example of communication between the object and its server using the instruction queue. Some of the operations do not need to be run on a game thread and can thus be run on a server thread, with the benefit of not impacting the overall simulator performance. Both the game mode and the drones expose an API composed of several operations. The overview of such APIs can be found in Table 4.1 and Table 4.2.

The messages between the server and the client are being sent in a serialized form using a cereal serialization library [35]. The library allows for the serialization and the deserialization of C++ data structures from and into several formats, including binary, JSON and XML. As there is no need for a human-readable message format, we can use the binary format. This brings a benefit, as serializing and deserializing binary format is faster and the serialized output is smaller than in JSON and XML formats. If there is a need for using a human-readable format, we can easily switch the formats.

As mentioned in Chapter 1, part of this work is also to create a client for the simulator written in C++. The client is called *UEds-connector* and exposes two main interfaces/controllers — the game mode controller and the drone controller. Both controllers are TCP clients pointing to their relevant server with a mapping to serialized requests and responses. The controllers follow the operations described in Table 4.1 and Table 4.2.

4.8 Support

The simulator was tested, validated and proved to be working on all three major platforms — Windows, macOS and Linux. The simulator has not shown any defects on all three platforms and worked as expected. The table of the exact test platforms can be seen in Table 4.3.

Operation	Game thread	Description
Ping	✗	Reachability test
GetDrones	✗	Returns an array of ports of spawned drones
SpawnDrone	✓	Spawns a drone
RemoveDrone	✓	Removes a drone
GetCameraCaptureMode	✗	Returns a camera capture mode of all drones
SetCameraCaptureMode	✓	Sets a camera capture mode of all drones
GetFps	✗	Returns FPS

Table 4.1: A list of operations implemented in the game mode server.

Operation	Game thread	Description
Ping	✗	Reachability test
GetLocation	✗	Returns the location of the drone
SetLocation	✓	Teleports the drone
GetCameraData	✓	Returns a JPG binary data of a camera output
GetRotation	✗	Returns the rotation of the drone
SetRotation	✓	Rotates the drone
SetLocationAndRotation	✓	Teleports and rotates the drone
GetLidarData	✗	Returns an array of distances and orientations of the LiDAR beams
GetLidarConfig	✗	Returns a configuration of the LiDAR
SetLidarConfig	✗	Sets a configuration of the LiDAR
GetCameraConfig	✗	Returns a configuration of the camera
SetCameraConfig	✓	Sets a configuration of the camera
GetMoveLineVisible	✗	Returns visibility of the move line
SetMoveLineVisible	✗	Sets visibility of the move line

Table 4.2: A list of operations implemented in the drone server.

Operating system	CPU	RAM	GPU
Fedora 37 with kernel 6.2	Intel Core i5-13600KF 14-Core 20-Threads	32GB DDR4	NVIDIA GeForce GTX 1070
Windows 11	Intel Core i5-13600KF 14-Core 20-Threads	32GB DDR4	NVIDIA GeForce GTX 1070
macOS 13	M1 Pro 10-Core (ARM)	32GB LPDDR5	M1 Pro

Table 4.3: A list of platforms, where the simulator was tested, validated and proved to be working.

Chapter 5

Benchmarks

The operations that are invokable upon the drone can take some time to execute. The execution time depends on several factors — the number of drones in the environment, LiDAR and camera settings etc. To make the connection between the performance and the factors clearer, we will benchmark three scenarios and discuss the results. The scenarios are: (i): the drone movement, as it is an operation that must be run on the game thread; (ii): the LiDAR data retrieval, as there are many configurable parameters; and (iii): the camera data retrieval, as there are configurable camera modes and it must be run on the game thread.

We will be performing a stress test — e.g. pushing the simulator to its limits by executing the same operation for a certain amount of time all over again. Two metrics will be captured: (i): the FPS of the simulator, which shows us the performance of the game thread; and (ii): the response time of each operation, which tells us the overall performance (game thread and server thread). All of the benchmarks will be conducted under a fresh start of the OS with no unusual background applications running.

For later benchmark analysis, we need to examine the architecture of the CPU first. The benchmarks will be performed on a desktop from Table 5.1. The CPU uses a new Intel architecture, which uses a combination of performance and efficiency cores and thus the performance of calculations on the CPU differs based on which type of core the calculation is being run on. The Intel Core i5-13600KF offers 6 performance and 8 efficiency cores and 12 performance and 8 efficiency threads. Whether some calculation will be run on the performance or the efficiency thread depends on the OS scheduler. The CPU architecture needs to be taken into account when reviewing the benchmarks for the reasons described above.

Operating system	CPU	RAM	GPU
Fedora 37 with kernel 6.2	Intel Core i5-13600KF 14-Core 20-Threads	32GB DDR4	NVIDIA GeForce GTX 1070

Table 5.1: A platform, where the simulator benchmarked.

For instance, let us imagine, that we benchmark the simulator by running some operations on nine drones repetitively and that the OS does not take any CPU resources. This means, that one CPU thread will be fully utilized by the game thread, nine CPU threads will be utilized by the drone server responding to the operations and another nine CPU threads will be utilized by the client asking the drone servers to do something. The game thread can be considered blocked because the Unreal Engine tries to push the thread to its limits. The other 18 threads have “windows”, e.g. wait for request/response and will let other computations take their place for some while. From our example, we can conclude, that out of 20 available CPU threads, there will be only one CPU thread left and some of the servers or clients will be running on the efficiency cores. If we modify our example by adding one or more drone servers and drone clients, there will be no more available CPU threads and the servers and the clients will have to take turns on the CPU threads.

5.1 Drone movement

As already stated, the drone movement changes the state of the drone and thus must be run on the game thread. In Chapter 4, we have shown that an operation running in the game thread slows down the performance of the simulator and the execution of other operations in the game thread. The assumption then is, that the more drone movement operations will be performed, the less FPS and higher response time we will get.

For various numbers of drones, we will be moving the drone by a random distance from interval $< -10\ cm; 10\ cm >$ in each of the coordinates for 60 seconds all over again. Each drone will be called from a different thread to ensure that the calls do not block each other. We will be capturing the FPS for the 60 seconds and also the response time for each request at each drone. The response times of each of the drones in one batch (number of drones) will be merged before plotting statistics. We will project the basic statistics and compare the results based on the drone counts.

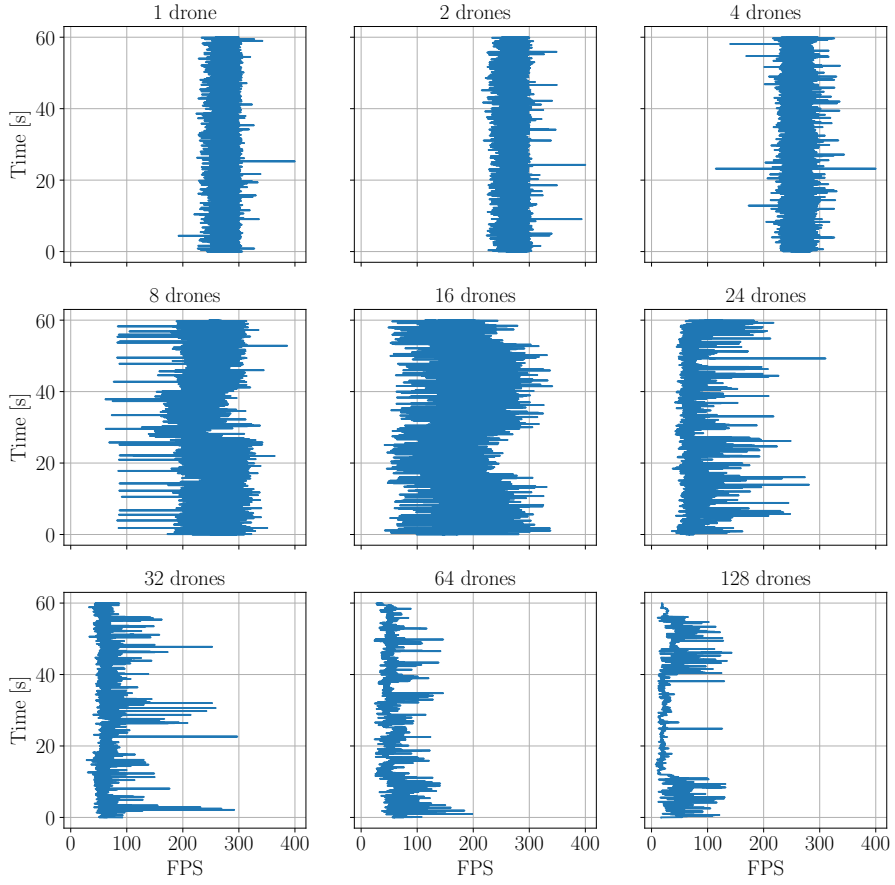


Figure 5.1: FPS in time relative to drone count measured at drone movement benchmark.

	FPS								
Drone count	#	**Mean**	Std	Min	Max	5th percentile	25th percentile	75th percentile	95th percentile
1	11695	279.02	13.22	192.19	399.98	257.09	269.47	289.15	297.05
2	11692	273.32	13.89	215.08	399.96	252.81	263.90	284.15	294.38
4	11689	263.82	12.05	114.49	399.99	246.46	257.92	269.20	284.26
8	11665	252.44	22.45	61.73	386.22	211.43	246.26	263.18	275.38
16	11708	185.46	41.18	41.76	341.07	120.55	159.24	213.11	250.18
24	11738	80.42	25.94	36.03	310.53	55.26	64.98	86.34	133.91
32	11683	65.52	18.46	28.31	297.32	47.95	56.33	68.79	99.27
64	11535	55.79	16.41	23.94	198.32	35.84	45.58	63.09	84.55
128	11751	33.49	19.86	7.50	143.06	13.52	19.24	44.75	70.14

Note: # refers to the number of measurements.

Table 5.2: FPS statistics relative to drone count measured at drone movement benchmark.

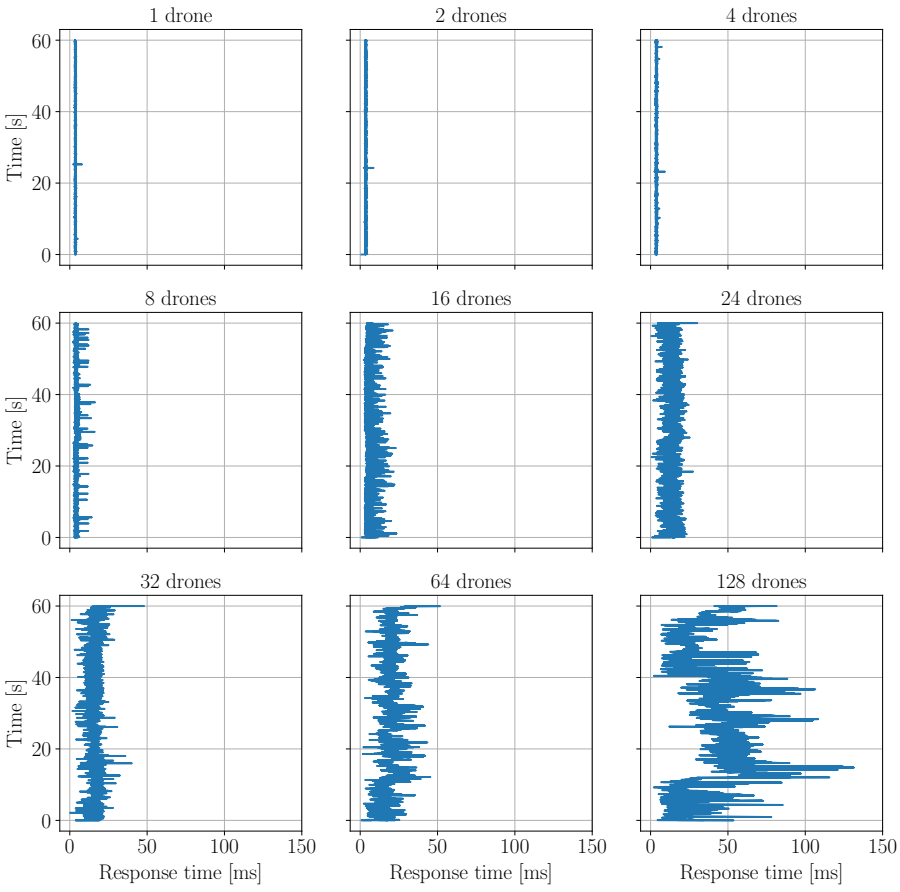


Figure 5.2: Response time in time relative to drone count measured at drone movement benchmark. Data are based on a 35% sample of the original data.

	Response time [ms]								
Drone count	#	**Mean**	**Std**	Min	Max	5th percentile	25th percentile	75th percentile	95th percentile
1	16705	3.53	0.22	1.17	7.98	3.21	3.37	3.67	3.91
2	32718	3.60	0.25	0.46	8.69	3.24	3.43	3.75	4.00
4	63179	3.73	0.24	0.38	9.36	3.37	3.60	3.86	4.10
8	119918	3.94	0.58	0.87	16.41	3.47	3.72	4.03	4.72
16	170137	5.58	1.85	0.67	23.45	3.66	4.50	6.18	8.22
24	111490	12.86	3.63	0.14	71.20	6.56	10.18	15.29	18.47
32	122141	15.66	3.51	0.10	76.58	9.18	14.08	17.73	20.81
64	207649	18.40	4.90	0.52	66.25	11.15	14.92	21.22	26.66
128	241258	31.70	16.94	1.61	147.30	10.84	19.38	43.77	63.07

Note: # refers to the number of measurements.

Table 5.3: Response time statistics relative to drone count measured at drone movement benchmark.

From Table 5.2 and Fig. 5.1, we can conclude that the original assumption that the FPS are getting lower with the increasing number of drones stands. The stability of the frame rate is approximately the same for each drone count, as the standard deviation does not noticeably differ. The standard deviation for 16 drones is higher in contrast to the other drone counts and is probably caused by the OS scheduler. The biggest FPS change is noticeable between 8 to 24 drones and is probably caused by the CPU hitting its limit, as described in the introduction of this chapter.

Table 5.3 and Fig. 5.2 show us, that the response time is getting higher with a higher drone count. As described in the introduction of this chapter, the CPU is not pushing its limits until 10 drones. This is visible in both the mean and the standard deviation of the response times, as until the 8 drones, both of the measurements stay approximately the same and grow with higher drone counts.

5.2 LiDAR

In Chapter 4, we concluded that the LiDAR does not need to be run on the game thread and that it is calculated on request. The execution time relies on the beam count and thus this will be our main focus of this benchmark. The operation runs on the server thread and we can assume that the performance of the game thread will not be influenced by growing beam count.

During the benchmark, the drone can stay in a stationary position because the LiDAR measurements are recalculated on each request. The setup of this benchmark can be seen in Fig. 5.3. We will be retrieving the LiDAR data for various beam counts all over again for 30 seconds. Similar to the drone movement benchmark, we will be capturing the FPS and the response times of the requests.

The assumption that the game thread is not affected by the LiDAR beam count is correct, as the FPS stay approximately the same for various beam counts (see Table 5.4). Similar to the drone movement benchmark, there is a noticeable change in FPS between 8 and 24 drones, which is probably caused by the CPU limitation. The Table 5.5 shows us, that the response time is getting higher with increasing beam count. Nevertheless, the standard deviation of the response time is low for drone counts within the CPU limitation which implies that the LiDAR retrieval is a stable operation.

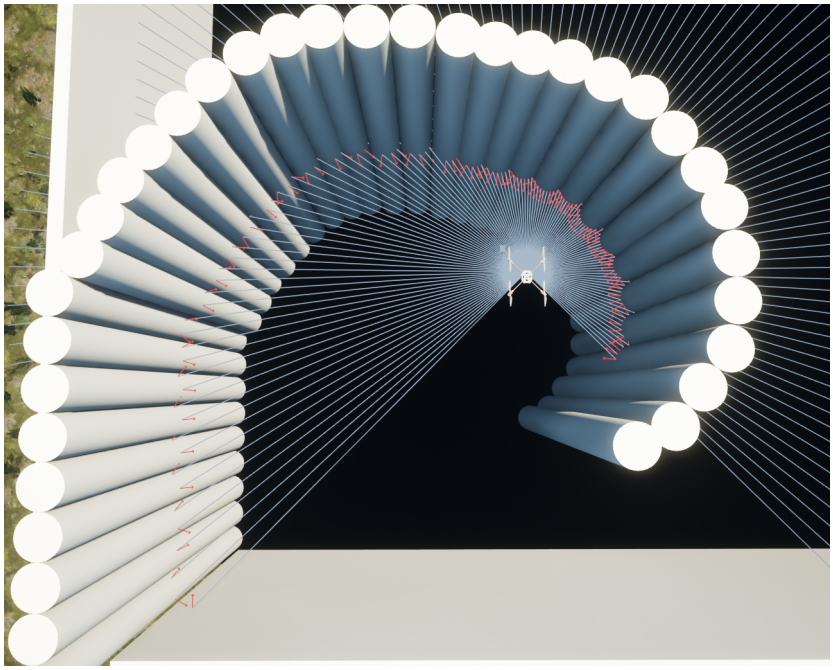


Figure 5.3: A LiDAR benchmark setup with 125 beams.

FPS															
Beam count	35			125			250			500			1000		
Drone count	#	**Mean**	Std	#	**Mean**	Std	#	**Mean**	Std	#	**Mean**	Std	#	**Mean**	Std
1	5855	292.07	13.69	5854	291.90	13.24	5852	292.21	13.53	5849	292.03	13.40	5844	291.46	14.02
2	5856	286.47	13.49	5853	286.66	13.29	5851	285.24	13.84	5849	285.75	13.30	5848	285.12	13.62
4	5852	278.14	13.51	5851	277.68	14.57	5848	277.12	13.85	5846	277.39	15.33	5841	276.29	15.85
8	5829	243.80	26.57	5842	265.60	16.24	5834	258.24	22.28	5829	254.37	24.37	5824	254.45	25.94
16	5873	137.41	38.20	5854	166.87	34.99	5853	169.71	35.20	5852	167.99	37.09	5843	165.37	40.48
24	5877	115.29	33.71	5863	131.66	31.15	5855	130.23	31.36	5846	129.50	32.85	5832	128.31	37.02
32	5865	102.94	28.60	5851	112.49	25.93	5839	111.46	26.92	5815	108.93	29.11	5783	109.33	32.50
64	5752	59.89	27.40	5758	61.57	28.88	5725	59.06	29.61	5694	56.43	30.40	5673	54.32	31.41
128	4429	23.40	15.32	4373	22.59	18.40	4226	17.32	11.74	4209	16.49	13.26	4055	18.56	19.88

Note: # refers to the number of measurements.

Table 5.4: FPS statistics relative to beam count measured at LiDAR benchmark.

Response time [ms]															
Beam count	35			125			250			500			1000		
Drone count	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**
1	235120	0.13	0.03	84381	0.35	0.07	46087	0.65	0.11	24495	1.22	0.17	12632	2.37	0.32
2	444573	0.13	0.03	158623	0.38	0.08	85246	0.70	0.13	44498	1.35	0.24	23044	2.60	0.42
4	785555	0.15	0.04	266192	0.45	0.12	138665	0.86	0.22	72028	1.66	0.42	37111	3.23	0.80
8	1103039	0.22	0.07	409492	0.58	0.14	198812	1.20	0.30	99322	2.41	0.60	50889	4.71	1.13
16	1452261	0.33	0.17	469994	1.02	0.35	240996	1.99	0.58	121594	3.94	1.01	61910	7.75	1.75
24	1503879	0.48	0.30	473711	1.52	0.62	239183	3.01	1.06	121936	5.90	1.77	61125	11.78	2.96
32	1467605	0.65	0.44	460877	2.08	0.97	235894	4.07	1.58	116188	8.26	2.68	57796	16.61	4.41
64	1265446	1.52	1.78	385607	4.99	3.77	196924	9.75	5.75	99371	19.32	9.02	49409	38.87	13.96
128	870900	4.40	4.80	258002	14.88	11.59	131139	29.29	20.70	65643	58.50	32.16	34015	113.10	49.40

Note: # refers to the number of measurements.

Table 5.5: Response time statistics relative to beam count measured at LiDAR benchmark.

5.3 Camera

In Chapter 4, we demonstrated that camera capture is a resource-heavy operation, which must be furthermore run on the game thread. We have also shown that there are situations, where we do not need the latest camera data at each time, which led to the introduction of the camera modes — capture all frames, capture on movement and capture on demand. The performance impact of the complexity of the environment that is being captured is unknown and should be benchmarked.

Similar to the other benchmarks, we will be calling the camera capture and recording the FPS and response time. We will compare all of the camera capture modes side-by-side with various drone counts. For each of the camera capture modes, we will compare capturing of two environments — a simple and a complex one, differing in the complexity of the objects visible to the camera (the environments can be seen in Fig. 5.4). Because of the logical differences in the capture modes, there will be two scenarios tested: (i): drone stays stationary; and (ii): drone moves before taking a snapshot from the camera. The measurements from the scenarios should reveal the differences in the performance impact of the capture modes.

(a) A simple environment. (b) A complex environment.

Figure 5.4: Samples of the environments captured during camera benchmark.

The Table 5.6 shows us the differences in the game thread performance based on camera capture mode, drone count, scenario and environment. Similarly, the Table 5.7 displays the differences in the performance of the servers based on camera capture mode, drone count, scenario and environment. The performance impact of the complexity of the environments that are being captured is noticeable in both FPS and response time metrics.

In contrast to the other modes, the game thread in the capture all frames mode is under a constant load as the FPS are lower. The standard deviation of the FPS metric is the lowest one and thus the capture all frames mode game thread performance is the most stable one. The response time metric in the stationary drone scenario is lower than in the drone moving scenario and is most probably caused by the client asking for the camera capture data getting ahead of the camera data refresh in the game thread. Compared to the capture on demand mode, retrieving the camera data is more stable, as the standard deviation of the response time metric is noticeably lower.

The capture on movement mode does not seem to impact the performance of the game thread in the stationary drone scenario, other than the FPS drop between the 8 to 24 drones

		FPS											
	Scenario	Stationary drone & data retrieved recurrently						Drone moves recurrently & data retrieved recurrently					
		Simple			Complex			Simple			Complex		
Camera mode	Drone count	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**
Capture all frames	1	2936	82.87	3.77	2939	57.20	5.86	2936	78.59	4.17	2938	56.58	5.55
	2	2939	48.37	1.90	2940	31.96	2.25	2937	46.06	2.16	2938	31.71	2.49
	4	2938	25.35	0.62	2939	16.49	0.32	2937	24.27	1.08	2940	16.22	0.68
	8	2936	13.01	0.45	2931	8.37	0.39	2939	12.38	0.44	2938	8.57	0.10
	16	2777	3.94	0.47	2855	2.68	0.24	2929	5.73	0.38	2940	4.02	0.26
	24	2503	2.54	0.17	2122	2.50	0.00	2923	2.93	0.21	2905	2.50	0.00
Capture on movement	1	2924	294.96	19.04	2922	298.41	20.33	2933	311.04	201.68	2933	343.97	172.10
	2	2923	281.91	19.92	2921	281.50	17.29	2937	53.14	17.36	2939	41.08	16.93
	4	2920	270.97	16.91	2916	266.91	17.34	2937	25.53	20.08	2940	17.92	19.04
	8	2881	228.81	30.09	2881	193.93	34.68	2938	14.39	22.07	2938	10.66	21.03
	16	2661	102.22	33.05	2685	79.05	24.04	2931	7.09	9.71	2935	5.43	10.26
	24	2376	68.15	11.70	2443	62.89	14.11	2911	4.98	10.16	2928	4.91	12.06
Capture on demand	1	2931	291.98	202.18	2932	337.65	173.00	2928	320.06	130.25	2931	264.70	143.50
	2	2937	57.04	18.97	2939	41.36	16.41	2933	201.66	174.23	2935	182.32	181.42
	4	2937	26.83	18.32	2939	18.19	20.09	2934	90.46	109.85	2936	66.20	102.42
	8	2934	15.31	23.73	2936	10.57	18.96	2935	27.56	9.01	2935	45.74	77.04
	16	2936	6.91	11.55	2944	5.72	14.79	2938	21.67	9.78	2950	14.30	15.89
	24	2887	4.98	8.62	2907	4.05	7.89	2928	12.54	5.80	2872	18.07	24.60

Note: # refers to the number of measurements.

Table 5.6: FPS statistics relative to camera capture mode, drone count, scenario and environment measured at camera benchmark.

		Response time [ms]											
	Scenario	Stationary drone & data retrieved recurrently						Drone moves recurrently & data retrieved recurrently					
		Simple			Complex			Simple			Complex		
Camera mode	Drone count	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**	#	**Mean**	**Std**
Capture all frames	1	1756	8.54	5.34	845	17.76	1.96	700	12.70	0.66	417	19.64	1.83
	2	52812	0.57	1.84	13028	2.30	4.40	1198	11.87	1.16	758	18.04	2.50
	4	149421	0.40	1.19	37080	1.62	2.84	1455	12.16	1.16	975	18.81	2.06
	8	320945	0.37	0.86	70142	1.71	2.17	1491	12.90	1.14	1035	19.07	1.78
	16	361110	0.66	1.29	89965	2.66	2.83	1386	14.27	3.11	984	20.59	3.14
	24	351430	1.02	1.81	87287	4.12	3.99	1081	17.59	3.99	828	23.46	4.34
Capture on movement	1	53187	0.28	0.04	13312	1.12	0.14	958	14.19	1.46	650	20.99	1.69
	2	101631	0.29	0.04	24263	1.23	0.22	1333	12.28	1.32	908	18.79	2.50
	4	188971	0.32	0.05	43583	1.37	0.21	1439	12.24	1.30	988	18.58	1.93
	8	344131	0.35	0.09	64700	1.85	0.38	1486	12.94	1.27	1021	19.49	1.99
	16	307206	0.78	0.82	73348	3.27	2.01	1378	14.34	3.24	972	20.97	3.54
	24	342371	1.05	1.18	78718	4.57	3.06	1113	17.07	3.91	811	24.54	4.37
Capture on demand	1	1043	14.38	1.78	670	22.41	2.38	793	17.42	1.31	543	25.49	1.44
	2	1411	21.27	4.42	906	33.13	3.96	1084	19.14	4.67	751	24.78	5.77
	4	1528	39.30	4.95	996	60.33	9.06	1295	30.82	9.94	878	41.07	11.70
	8	1567	76.77	8.58	1055	114.22	11.15	1393	51.66	16.84	951	71.05	28.19
	16	1295	186.52	24.60	905	267.77	32.44	1273	104.49	44.45	840	154.36	72.20
	24	1221	297.68	34.56	880	414.65	45.21	1141	174.48	83.24	793	244.40	121.37

Note: # refers to the number of measurements.

Table 5.7: Response time statistics relative to camera capture mode, drone count, scenario and environment measured at camera benchmark.

caused by the already discussed CPU limitation. The game thread performance of the capture on movement mode in the drone moves recurrently scenario is better than in the capture all frames mode, as the FPS are higher. The standard deviation of the FPS metric is higher and thus less stable than in the capture all frames mode. The response time of the capture on movement mode in the stationary drone scenario is low, which is caused by the camera not being updated in the scenario. The response time of the capture on movement mode in the drone moves recurrently scenario is comparable to the capture all frames mode.

The game thread performance of the capture on demand mode is slightly better than in the capture all frames mode and is most probably caused by the camera not being refreshed at each Tick because of the delay between the client and the server. On the other hand, the standard deviation of the FPS metric is very high compared to the other modes and is caused by the intermittent camera refreshing in the game thread. Based on that observation, the Unreal Engine seems more optimized for a constant load. The response time of the capture on demand mode is much higher and unstable compared to the other modes, which is caused by the waiting for the camera capture.

The performance of the modes differs and is tightly coupled to the scenario. The capture all frames mode is suitable for dynamic environments, where the camera data are being retrieved at a high rate. The capture on movement mode is suitable for static environments, where the camera data are being retrieved at a high rate. The capture on demand mode is suitable for both dynamic and static environment, where the camera data are expected to be retrieved on an infrequent basis.

5.4 Conclusion

We benchmarked the UEds simulator on various scenarios. As already stated, the game thread runs on a single thread and thus the overall performance of the game thread and the FPS count depends on the CPU single-thread performance. The servers run on separate threads and as the stress tests have shown, the performance of the servers (response time) in multi-drone scenarios depends on the thread count of the CPU. The performance impact of the GPU is unknown, but it is expected to gain higher performance in the camera capture benchmark with a more powerful GPU than the one that was used for the benchmarks described above.

The simulator is resource-heavy and requires a powerful multi-core CPU for running a multi-drone scenario. Nevertheless, there is a limit to how many cores and threads can a CPU pack and nowadays, the most high-performance and expensive CPUs have up to 64 cores and 128 threads. The CPU with a high thread count might not even be fully utilized, as the game thread depends on the single-thread performance and might bottleneck the servers while executing the instructions that change the state of the environment.

To gain the highest performance out of the UEds simulator, we can run multiple instances of the UEds simulator on single or multiple machines in cases where the environment is static and the drones do not impact each other. This can become especially useful when training a drone model based on a static environment.

Chapter 6

Experiments

In previous chapters, we discussed the architecture of the UEds simulator and benchmarked it on various scenarios. As already stated, the simulator is meant to be focused on realistic rendering and collision handling and we need to verify that this is fulfilled. Both the LiDAR sensor and the RGB camera must be not only benchmarked but also validated that they work properly. As one of the primary usages of this simulator is training various drone controllers, we can validate both the LiDAR sensor and the RGB camera in reinforcement learning tasks, which will also be the focus of this chapter.

6.1 Prerequisites

Reinforcement learning is a machine learning subfield, where agents learn by taking actions in an environment with some prior observation of the environment and subsequentially getting a reward [52]. In the UEds simulator context, the agent is the drone, the environment is the map and the objects that exist in it and the observation can be for instance a direction vector to some goal, a measurement from LiDAR or an RGB camera data. Using a combination of UEds-connector and some reinforcement learning framework, we can effortlessly train a drone model for various tasks.

The UEds-connector is written in C++ and no major reinforcement learning frameworks exist in the C++ ecosystem. Most of the reinforcement learning frameworks are based on Python, as it is a standard for AI and scientific applications. However, there is a way of binding code written in C++ to Python using pybind11 library [29]. The UEds-connector controllers can be then wrapped using the library and used in the Python ecosystem.

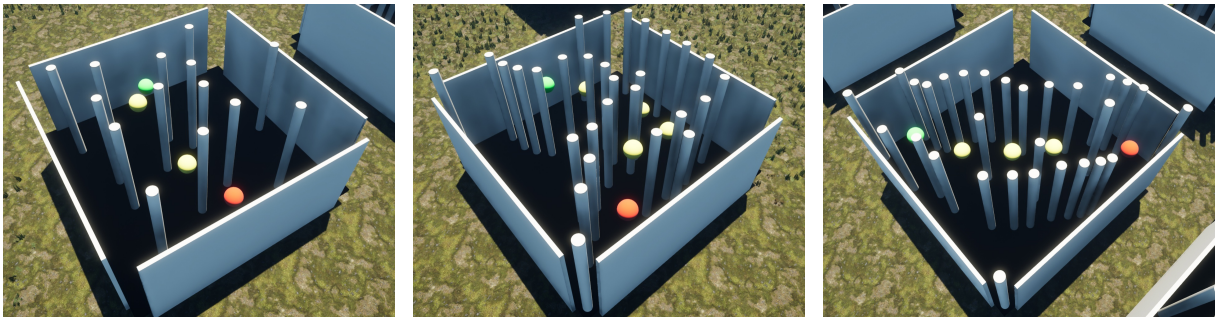
OpenAI Gym is an open-source reinforcement learning library written in Python, which exposes a unified API of the reinforcement learning environments [24]. The environments are easily pluggable into many libraries implementing reinforcement learning algorithms. It features a set of predefined environments (e.g. Atari games) and also allows for creating custom ones. The custom environment can be created by extending the base OpenAI Gym environment class and implementing common methods. We can develop a custom OpenAI Gym environment by holding an UEds-connector drone controller inside of it and calling the methods of the controller where necessary.

Stable-Baselines3 is an open-source state-of-the-art library containing reinforcement learning algorithms using PyTorch as a backend [40]. It is tightly coupled with the OpenAI Gym environment and provides an out of the box support for it. The library provides a set of reinforcement learning algorithms and a common API for using them. The algorithms can run in parallel using Vectorized Environments [7] and allow for training a single model from multiple scenarios and environments. This can become useful for fast training of a drone model in different environments.

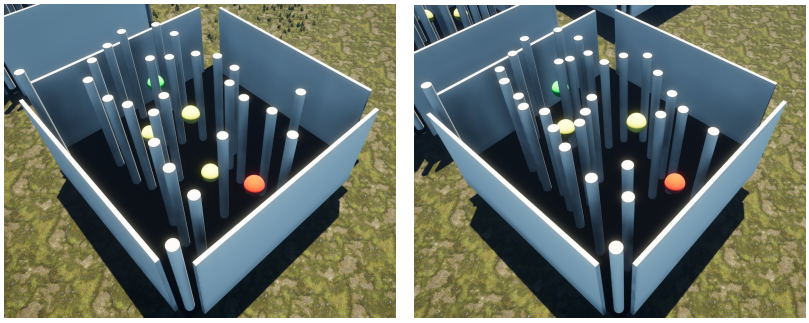
The UEds-connector is extended by the libraries described above. The drone and the game mode controllers are wrapped using the pybind11 library and are accessible from Python. The OpenAI Gym environments for the experiments described below are implemented and are directly connected to the UEds simulator using the UEds-connector. The Stable-Baselines3 algorithms and Vectorized Environments will be used for the experiments described below. This whole integration of the UEds-connector and the libraries discussed above forms a reinforcement learning framework over the UEds simulator.

6.2 LiDAR

To verify the functionality of the LiDAR sensor, we will try to train a drone controller based mainly on the data provided by the sensor. We will spawn a drone in a playground with obstacles and train it to fly to a certain predefined point. To ease the process of the training, the drone will have to fly through predefined waypoints before reaching the goal without crashing into anything. We will use five different training playgrounds upon which will be one drone controller trained in parallel using the Stable-Baselines3 Vectorized Environments. The training playgrounds can be seen in Fig. 6.1.



(a) Training playground 1. (b) Training playground 2. (c) Training playground 3.



(d) Training playground 4. (e) Training playground 5.

Figure 6.1: Training playgrounds used in the LiDAR experiment. All of the playgrounds have the same 1,400 cm × 1,400 cm × 525 cm dimensions. The pillars in the playgrounds have a diameter of 50 cm. The red sphere corresponds to the start of the playground, the yellow spheres correspond to the waypoints and the green sphere corresponds to the goal of the playground.

To ease the process of the training, the action space will be set as a discrete space rather than a continuous one. We will let the drone move in six directions — up, down, left, right,

forward and backward. The step will be set to 10 cm, which is around a ninth of the drone length (the drone dimensions are 93 cm × 59 cm × 20 cm). The rotation will not be taken into account and the playgrounds are designed with this in mind.

The observation space consists of two parts — the direction to the current goal and the LiDAR measurements. The direction is set either to the waypoint or the goal, depending on the current state. To make the model work with any direction, the direction is a normalized direction vector to the current goal. Similarly, the LiDAR measurements are also normalized for the same reason. The LiDAR is set with a 270 degrees FOV angle, 35 beams, a beam length of 300 cm and is facing forward.

The reward function is based on the reward function from [37] and consists of five different variables — distance reward, crash penalty, free space reward, step penalty and acceleration penalty. The distance reward is calculated as

$$r_d = \begin{cases} R_{goal} & \text{if } d < T_{goal} \\ \Delta d & \text{otherwise,} \end{cases} \tag{6.1}$$

where d is the distance to the current goal, R_{goal} is the reward for achieving the current goal and T_{goal} is the threshold for the current goal distance. The crash penalty is calculated as

$$p_c = \begin{cases} -R_{goal} & \text{if drone crashed} \\ \min_{l \in L} l - M_{OD} & \exists l \in L \colon l < M_{OD} \\ 0 & \text{otherwise,} \end{cases} \tag{6.2}$$

where L is the set of normalized LiDAR measurements, R_{goal} is the reward for achieving a goal and M_{OD} is the maximal obstacle distance. The free space reward is calculated as

$$r_f = \frac{\sum_{l \in L} (\frac{l}{M_{OD}} - 1)}{\frac{|L|}{M_{OD}} - |L|}, \tag{6.3}$$

where L is the set of normalized LiDAR measurements and M_{OD} is the maximal obstacle distance. The step penalty is calculated as

$$p_s = -\frac{S}{M_S}, \tag{6.4}$$

where S is the current number of the learning step and M_S is the maximal step number. The acceleration penalty is calculated as

$$p_a = \begin{cases} 0 & \text{if the current step is the same as the last one} \\ -1 & \text{otherwise.} \end{cases} \tag{6.5}$$

The reward is then calculated as

$$R_l = r_d + p_c + r_f + p_s + p_a. \tag{6.6}$$

The parameters used for the training can be seen in Fig. 6.1.

The Stable-Baselines3 library provides several reinforcement learning algorithms. DQN, A2C and PPO algorithms [4–6] were tested for the LiDAR experiment. Only the PPO algorithm has shown promising results and the learning is thus based on the PPO algorithm. The

Parameter	Notation	Value
LiDAR beam count	$\|L\|$	35
Goal threshold	T_{goal}	50 cm
Goal reward	R_{goal}	10000
Maximal obstacle distance	M_{OD}	0.2
Maximal step number	M_S	3000

Table 6.1: A list of parameters used for the reward function of the LiDAR experiment.

Parameter	Value
Learning rate	0.0003
Discount factor	0.99
Clipping parameter	0.2
The number of steps to run for each environment per update	2048
Policy	MlpPolicy

Table 6.2: A list of parameters used for the PPO algorithm in the LiDAR experiment.

algorithm was run with the default parameters provided by the Stable-Baselines3. A subset of the PPO algorithm parameters can be seen in Fig. 6.2. The training will be conducted with 40 drones — eight for each of the training playgrounds.

The training was conducted for 36,864,000 steps with an average speed of 1,327 steps per second. The progress of the training reward value can be seen in Fig. 6.2. The training was able to achieve a positive reward value just after 3,358,720 steps. The trained model is able to fly from the start to the goal of all of the training playgrounds without crashing (several learned paths in the training playgrounds can be seen in Fig. 6.3). Two validation playgrounds were created and the trained model is able to find a path from the start to the goal without crashing in both of the validation playgrounds (see Fig. 6.4).

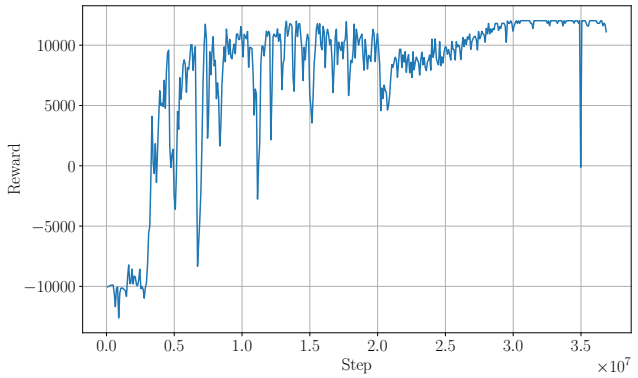
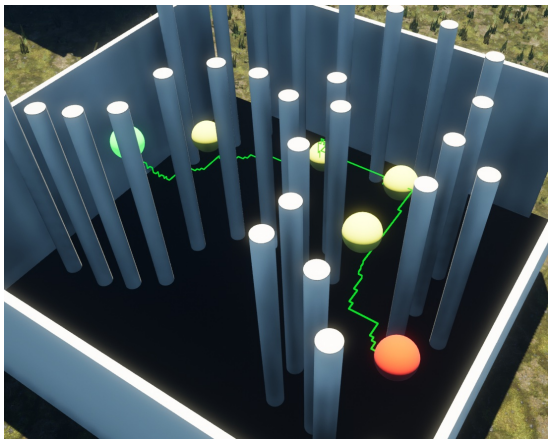
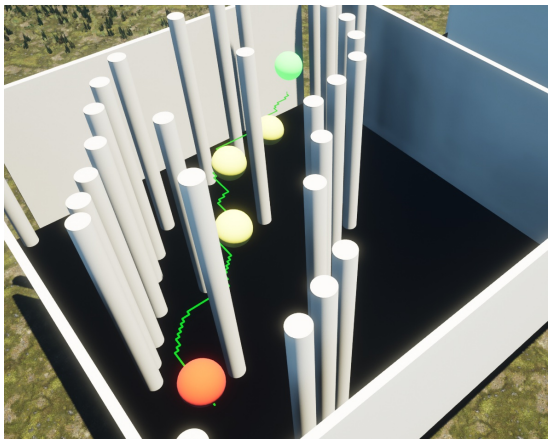


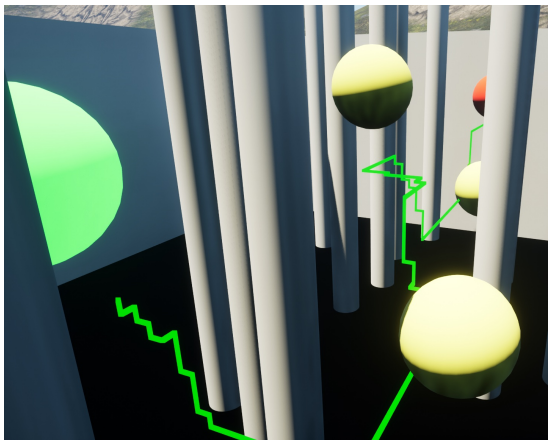
Figure 6.2: Reward value in steps of the LiDAR experiment training.



(a) Training playground 2 (reward 11,344.6)

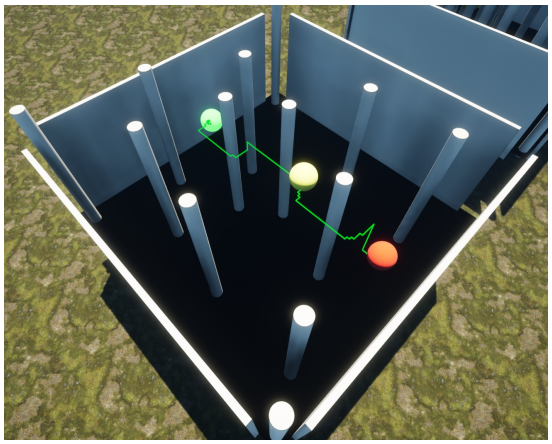


(b) Training playground 3 (reward 11,435.5)

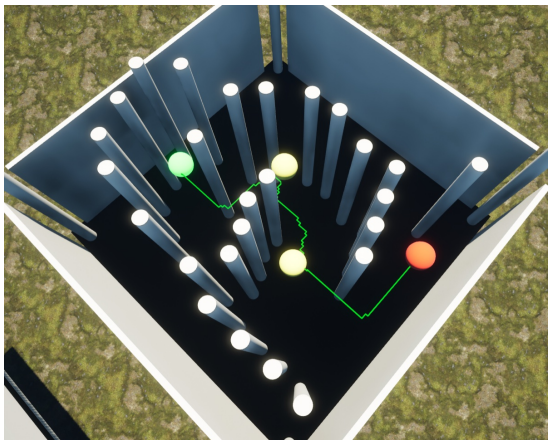


(c) Training playground 4 (reward 11,250.5)

Figure 6.3: Learned paths in the training playgrounds of the LiDAR experiment. The green line represents the trail of the drone.



(a) Validation playground 1 (reward 11,422.6)



(b) Validation playground 2 (reward 11,475.2)

Figure 6.4: Learned paths in the validation playgrounds of the LiDAR experiment. The green line represents the trail of the drone.

6.3 Camera

In the last experiment, we gave the drone controller a direction to a predefined goal and LiDAR measurements as an observation and trained it to reach it without crashing into anything. To verify the functionality of the RGB camera, the direction to the goal will be calculated from an image captured using the RGB camera attached to the drone. We will find the location of the green sphere (goal) in the image and convert it to a direction, which will be used as an observation. Similarly to the LiDAR experiment, the playgrounds consist of obstacles and a goal, which is the green sphere. We will use two different training playgrounds upon which will be one drone controller trained in parallel using the Stable-Baselines3 Vectorized Environments.

At each step, the drone controller will capture an image using its camera. Green color space will be filtered-out from the image using opencv-python library [25], which produces a mask, where the green color space has switched-on bits. If there are any switched-on bits in the mask, we find a boundary box around it using Pillow library [27]. We create a vector from the center of the original image to the center of the boundary box and thus retrieve a direction to the goal. If no green color space is present, the direction will be set as a null vector. The steps of extracting the direction can be seen in Fig. 6.5. The floors and the walls of the playgrounds were set to a white color to ease the green color space localization. The playgrounds used for the training can be seen in Fig. 6.6.

The action space is similar to the LiDAR experiment but was simplified to accelerate the process of the training. The drone will move by 10 cm in three directions — forward, left and right. The playgrounds are built with this in mind and the LiDAR FOV angle will be reduced to 180 degrees.

As already discussed, the first part of the observation space will be set as a direction to the green sphere captured from the camera. The action space permits the drone to move in the up and down direction, so only a direction in the width of the image will be used. To make the model work with any size of the captured image, the direction is normalized. We also pass the boundary box area to the captured image area ratio, which indicates the drone how close it is to the green sphere. Similar to the LiDAR experiment, the model needs to avoid collisions and normalized measurements from the LiDAR will be passed to the observation space as well. The LiDAR is configured with 180 degrees FOV angle, 20 beams, a beam length of 300 cm and is facing forward. The camera is configured with 90 degrees FOV angle and is facing forward.

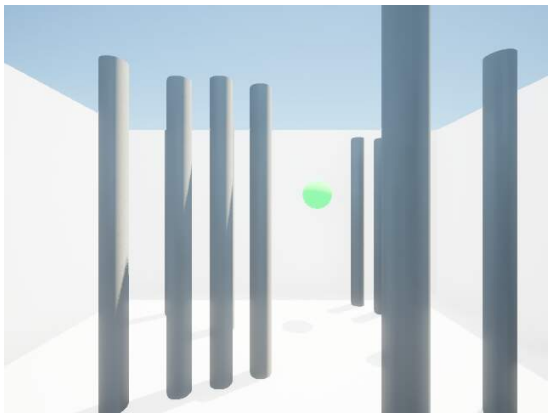
The reward is calculated in the same way as in the LiDAR experiment with an addition of an outrun penalty. We introduce the outrun penalty, because the drone can not rotate, moves only in three directions and might outrun the goal. The outrun penalty is calculated as

$$p_o = \begin{cases} -\frac{R_{goal}}{10} & \text{if } x > X_{goal} + T_{goal} \\ 0 & \text{otherwise,} \end{cases} \tag{6.7}$$

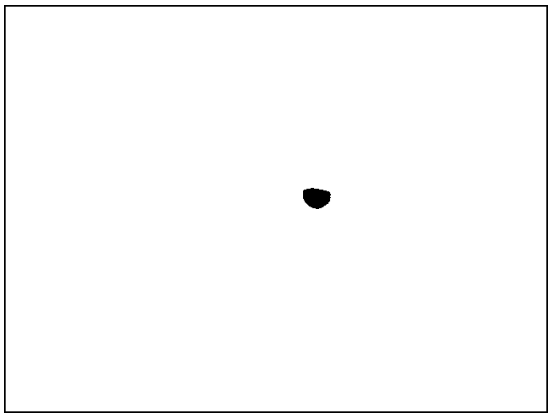
where R_{goal} is the reward for achieving the current goal, x is an X coordinate of the current location of the drone, X_{goal} is an X coordinate of the current goal and T_{goal} is the threshold for the current goal distance. The reward is then calculated as

$$R_c = R_l + p_o, \tag{6.8}$$

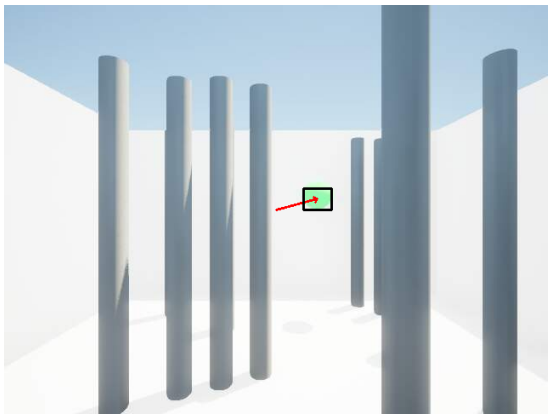
where R_l is the reward from (6.6). The parameters used for the training are similar to the ones used in the LiDAR experiment and can be seen in Fig. 6.3.



(a) A camera capture.



(b) The filtered-out green color space. The black color space represents the filtered-out green color space and the white color space represents a blank space. A black border was added and the colors of the original mask were inverted for better visibility.

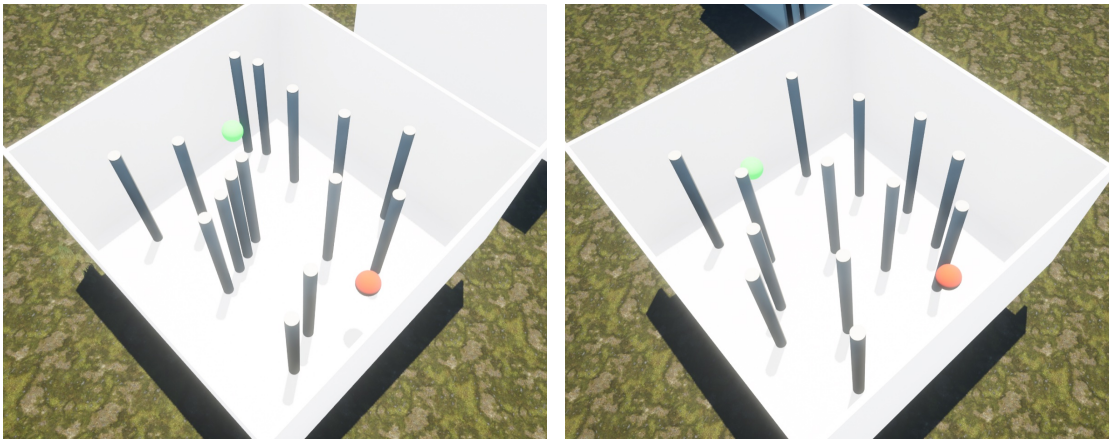


(c) The original camera capture with a direction from the center of the image to the center of the green color space, represented by the red arrow. The black box represents the boundary box of the green color space.

Figure 6.5: Steps of extracting the direction from the center of the camera capture to the green color space. The green color space is filtered from the original camera capture (Fig. 6.5a) and results in a mask (Fig. 6.5b). The center of the boundary box of switched-on bits of the mask represents the center of the green color space and the direction is calculated as a vector from the center of the original image to the center of the boundary box.

Parameter	Notation	Value
LiDAR beam count	$\|L\|$	20
Goal threshold	T_{goal}	50 cm
Goal reward	R_{goal}	10000
Maximal obstacle distance	M_{OD}	0.2
Maximal step number	M_S	3000

Table 6.3: A list of parameters used for the reward function of the camera experiment.



(a) Training playground 1. (b) Training playground 2.

Figure 6.6: Training playgrounds used in the camera experiment. Both of the playgrounds have the same 1,400 cm × 1,400 cm × 525 cm dimensions. The pillars in the playgrounds have a diameter of 50 cm. The red sphere corresponds to the start of the playground and the green sphere corresponds to the goal of the playground.

Similar to the LiDAR experiments, DQN, A2C and PPO algorithms were tested for the training. Only the PPO algorithm was able to train the model in a reasonable time and thus is used for this experiment. A subset of the PPO algorithm parameters can be seen in Fig. 6.4. The camera capture and the green sphere localization is substantially slower operation than obtaining an exact position of the goal and it is expected that the training will be slower than in the LiDAR experiment. The experiment will also block the game thread more than the LiDAR experiment due to the camera capture. Because of the reasons described above, we will perform the training with eight drones — four for each of the training playgrounds.

Parameter	Value
Learning rate	0.0001
Discount factor	0.99
Clipping parameter	0.2
The number of steps to run for each environment per update	2048
Policy	MlpPolicy

Table 6.4: A list of parameters used for the PPO algorithm in the camera experiment.

We conducted the training for 4,400,000 steps with an average speed of 74 steps per second, which is about 95.5% slower than in the LiDAR experiment. We trained the model to be able to fly through both training playgrounds without crashing and successfully reach the goal. The progress of the training reward in steps can be seen in Fig. 6.7 and the learned path can be seen in Fig. 6.8. The model was also successfully verified in two validation playgrounds and the learned paths in the validation playgrounds can be seen in Fig. 6.9.

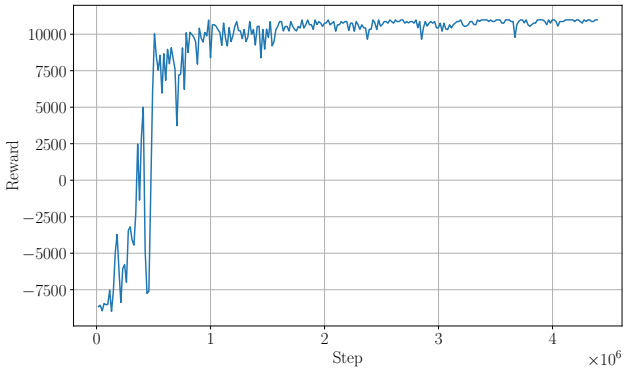
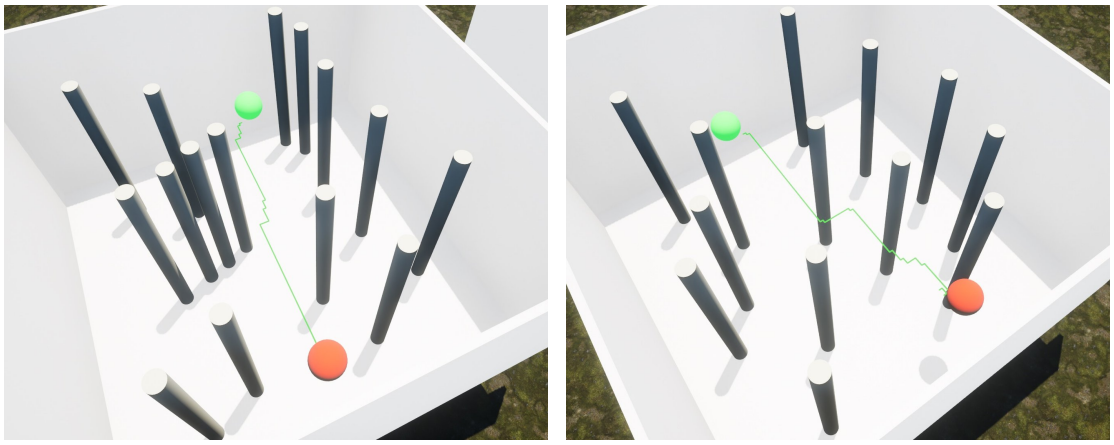
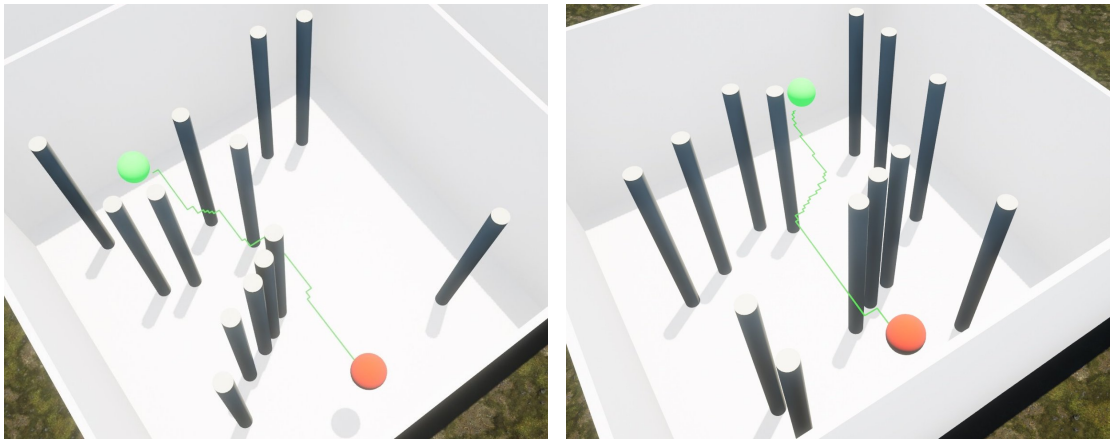


Figure 6.7: Reward value in steps of the camera experiment training.



(a) Training playground 1 (reward 10,975.6) (b) Training playground 2 (reward 10,993.7)

Figure 6.8: Learned paths in the training playgrounds of the camera experiment. The green line represents the trail of the drone.



(a) Validation playground 1 (reward 10,976.8) (b) Validation playground 2 (reward 10,991.8)

Figure 6.9: Learned paths in the validation playgrounds of the camera experiment. The green line represents the trail of the drone.

6.4 Conclusion

A reinforcement learning framework over the UEds simulator was created using the OpenAI Gym environment and the Stable-Baselines3 library. We were able to train drone controllers based on LiDAR measurements and camera data to fly from a start to a goal through obstacle playgrounds. As expected based on the benchmarks, the training in tasks requiring camera data refreshes was substantially slower. Nevertheless, we were able to achieve 1,327 steps per second in the LiDAR experiment and 74 steps per second in the camera experiment. The LiDAR and the camera implementations proved to be working and we verified their applicability in machine learning tasks. The collision handling functionality was confirmed during the training, as crashing the drone into a pillar, a wall or a floor resulted in a highly negative reward.

Chapter 7

Conclusion

We proposed a drone simulator focused on realistic visualization and based on Unreal Engine 5. We analyzed and researched the parts that are required for developing such a simulator. The simulator supports operating multiple drones at once and exposes a server for each of the drones and the main control unit. The main control unit is able to spawn and remove the drones on the fly. The collision detection is being handled appropriately and sensors such as LiDAR, RGB camera and position/tilt sensor were implemented. The simulator is easily importable to any Unreal Engine environment and is supported on all three major operating systems.

We implemented a client for the simulator written in C++ programming language. The client communicates with the server over a TCP/IP socket in a fast and reliable way. We wrapped the client using the pybind11 library to be able to use it from the Python ecosystem. The OpenAI Gym environment for the simulator was created and can be used by various reinforcement learning frameworks.

We benchmarked the simulator in various scenarios and revealed the performance impact of particular configurations. All of the benchmarks were conducted for several drone counts and disclosed the CPU thread count limitation under a stress test. The benchmarks have shown the performance limitation of the single-threaded game loop and advise us to use the simulator in multiple instances under a heavy load.

We created two reinforcement learning tasks based on the LiDAR sensor and the RGB camera. The models trained in both of the tasks were able to fly through their corresponding obstacle playgrounds without crashing. The created OpenAI Gym environments proved to be working correctly and we formed a reinforcement learning framework over the UEds simulator. We confirmed the ability of the simulator to serve machine learning purposes.

The proposed simulator proved to be working and stable on a mid-end desktop in various scenarios. Essential features of the drone simulator were developed and tested. The benchmarks and the reinforcement learning experiments demonstrated the capabilities of the simulator in multi-drone scenarios. The requirements from the MRS Group were fulfilled and there is a space for the development of more features.

7.1 Future work

We discussed several points of possible improvements and new features throughout this thesis. Some of the features are implemented in the related drone simulators and implementing them in the proposed simulator would be highly beneficial. The drone dynamics/kinematics was not taken into account in this thesis and as discussed, should be implemented either as a middleware or as a plugin directly in the simulator. An RGBD camera should be implemented, as it is a fairly common feature in the drone simulation field. Implementing an endpoint for

reading the map mesh data (point cloud) would be valuable for path planning purposes. Sensor flaw settings (errors, delays, etc.) should be implemented in order to achieve an even more realistic experience. Implementing an endpoint for loading and creating a simple environment (e.g. obstacle playground) would be beneficial, as many of the simple environments for machine learning purposes are generated by an algorithm and creating them manually in the simulator takes time.

Chapter 8

References

[1] Anton Babushkin. *DrTon/jMAVSim: Simple multirotor simulator with MAVLink protocol support*. https://github.com/DrTon/jMAVSim. [Online; accessed 28-April-2023].

[2] Cyberbotics Ltd. *Webots documentation: DJI' Mavic 2 PRO*. https://cyberbotics.com/doc/guide/mavic-2-pro. [Online; accessed 10-April-2023].

[3] Cyberbotics Ltd. *Webots documentation: License Agreement*. https://cyberbotics.com/doc/guide/webots-license-agreement. [Online; accessed 10-April-2023].

[4] DLR-RM. *A2C — Stable Baselines3 2.0.0a8 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html. [Online; accessed 18-May-2023].

[5] DLR-RM. *DQN — Stable Baselines3 2.0.0a8 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/dqn.html. [Online; accessed 18-May-2023].

[6] DLR-RM. *PPO — Stable Baselines3 2.0.0a8 documentation*. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html. [Online; accessed 18-May-2023].

[7] DLR-RM. *Vectorized Environments — Stable Baselines3 2.0.0a8 documentation*. https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html. [Online; accessed 15-May-2023].

[8] Epic Games. *AActor::GetActorLocation — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/GameFramework/AActor/GetActorLocation. [Online; accessed 1-May-2023].

[9] Epic Games. *AActor::GetActorRotation — Unreal Engine Documentation*. https://docs.unrealengine.com/4.27/en-US/API/Runtime/Engine/GameFramework/AActor/GetActorRotation. [Online; accessed 1-May-2023].

[10] Epic Games. *AActor::SetActorLocation — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/GameFramework/AActor/SetActorLocation. [Online; accessed 1-May-2023].

[11] Epic Games. *Content by Epic Games - UE Marketplace*. https://www.unrealengine.com/marketplace/en-US/profile/Epic+Games?count=20&sortBy=effectiveDate&sortDir=DESC&start=0. [Online; accessed 15-May-2023].

[12] Epic Games. *FBX Scene Import in Unreal Engine — Unreal Engine 5.1 Documentation*. https://docs.unrealengine.com/5.1/en-US/fbx-scene-import-in-unreal-engine. [Online; accessed 8-May-2023].

[13] Epic Games. *SetActorLocationAndRotation — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/GameFramework/AActor/SetActorLocationAndRotation. [Online; accessed 1-May-2023].

[14] Epic Games. *SetActorRelativeRotation — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/GameFramework/AActor/SetActorRelativeRotation. [Online; accessed 1-May-2023].

[15] Epic Games. *The most powerful real-time 3D creation tool - Unreal Engine*. https://www.unrealengine.com/en-US. [Online; accessed 8-May-2023].

[16] Epic Games. *TQueue — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Core/Containers/TQueue. [Online; accessed 25-April-2023].

[17] Epic Games. *UCameraComponent — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Camera/UCameraComponent. [Online; accessed 1-May-2023].

[18] Epic Games. *Unreal Engine Marketplace — Store of UE Assets for Games and 3D Rendering - UE Marketplace*. https://www.unrealengine.com/marketplace/en-US/store. [Online; accessed 8-May-2023].

[19] Epic Games. *USceneCaptureComponent2D — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Components/USceneCaptureComponent2D. [Online; accessed 1-May-2023].

[20] Epic Games. *UTextureRenderTarget2D — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Engine/UTextureRenderTarget2D. [Online; accessed 1-May-2023].

[21] Epic Games. *UWorld::LineTraceSingleByChannel — Unreal Engine Documentation*. https://docs.unrealengine.com/5.1/en-US/API/Runtime/Engine/Engine/UWorld/LineTraceSingleByChannel. [Online; accessed 1-May-2023].

[22] MAVLink. *Introduction · MAVLink Developer Guide*. https://mavlink.io/en. [Online; accessed 28-April-2023].

[23] Multi-robot Systems Group. *Multi-robot Systems*. http://mrs.felk.cvut.cz/. [Online; accessed 8-May-2023].

[24] OpenAI. *Gym Documentation*. https://www.gymlibrary.dev. [Online; accessed 8-May-2023].

[25] OpenCV. *Automated CI toolchain to produce precompiled opencv-python, opencv-python-headless, opencv-contrib-python and opencv-contrib-python-headless packages*. https://github.com/opencv/opencv-python. [Online; accessed 21-May-2023].

[26] Oracle Corporation. *Java 3D API*. https://www.oracle.com/java/technologies/javase/java-3d.html. [Online; accessed 28-April-2023].

[27] Pillow. *Python Imaging Library (Fork)*. https://github.com/python-pillow/Pillow. [Online; accessed 21-May-2023].

[28] PX4 AutoPilot. *jMAVSim with SITL — PX4 User Guide*. https://docs.px4.io/main/en/simulation/jmavsim.html. [Online; accessed 28-April-2023].

[29] pybind. *pybind11 — Seamless operability between C++11 and Python*. https://github.com/pybind/pybind11. [Online; accessed 11-May-2023].

[30] Smith, Russell. *Open Dynamics Engine*. http://www.ode.org. [Online; accessed 28-April-2023].

[31] The MathWorks, Inc. *Simulink Documentation*. https://www.mathworks.com/help/simulink. [Online; accessed 29-April-2023].

[32] The MathWorks, Inc. *Unreal Engine Simulation for Unmanned Aerial Vehicles - MATLAB & Simulink*. https://www.mathworks.com/help/uav/ug/3d-simulation-for-unmanned-aerial-vehicles.html. [Online; accessed 29-April-2023].

[33] The MathWorks, Inc. *US city block Unreal Engine environment - MATLAB*. https://www.mathworks.com/help/uav/ref/uscityblock.html. [Online; accessed 29-April-2023].

[34] Unity Technologies. *Unity Real-Time Development Platform — 3D, 2D, VR & AR Engine*. https://unity.com/. [Online; accessed 25-April-2023].

[35] USCiLab. *cereal Docs - Main*. https://uscilab.github.io/cereal. [Online; accessed 5-May-2023].

[36] UZH Robotics and Perception Group. *DJI Mavic 2 PRO Simulation in Webots*. https://www.youtube.com/watch?v=-hJssj_Vcw8. [Online; accessed 10-April-2023].

[37] Yongsheng Yang et al. "DRL-based Path Planner and its Application in Real Quadrotor with LIDAR". In: *Journal of Intelligent & Robotic Systems* 107.3 (2023), p. 38.

[38] Alexandros Gazis and Eleftheria Katsiri. "Middleware 101". In: *Commun. ACM* 65.9 (Aug. 2022), pp. 38–42. ISSN: 0001-0782. DOI: 10.1145/3546958. URL: https://doi.org/10.1145/3546958.

[39] Jack Collins et al. "A Review of Physics Simulators for Robotic Applications". In: *IEEE Access* 9 (2021), pp. 51416–51431. DOI: 10.1109/ACCESS.2021.3068769.

[40] Antonin Raffin et al. "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22.268 (2021), pp. 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.

[41] Yunlong Song et al. "Flightmare: A Flexible Quadrotor Simulator". In: *Conference on Robot Learning*. 2020.

[42] Winter Guerra et al. "Flightgoggles: A modular framework for photorealistic camera, exteroceptive sensor, and dynamics simulation". In: *arXiv preprint arXiv:1905.11377* (2019).

[43] Shital Shah et al. "AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles". In: *Field and Service Robotics*. 2017. eprint: arXiv:1705.05065. URL: https://arxiv.org/abs/1705.05065.

[44] Mahanth Gowda et al. "Tracking Drone Orientation with Multiple GPS Receivers". In: *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. MobiCom '16. New York City, New York: Association for Computing Machinery, 2016, pp. 280–293. ISBN: 9781450342261. DOI: 10.1145/2973750.2973768. URL: https://doi.org/10.1145/2973750.2973768.

[45] Norhafizan Ahmad et al. "Reviews on various inertial measurement unit (IMU) sensor applications". In: *International Journal of Signal Processing Systems* 1.2 (2013), pp. 256–262.

[46] Aaron Staranowicz and Gian Luca Mariottini. "A Survey and Comparison of Commercial and Open-Source Robotic Simulator Software". In: *Proceedings of the 4th International Conference on PErvasive Technologies Related to Assistive Environments*. PETRA '11. Heraklion, Crete, Greece: Association for Computing Machinery, 2011. ISBN: 9781450307727. DOI: 10.1145/2141622.2141689. URL: https://doi.org/10.1145/2141622.2141689.

[47] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[48] Dave Shreiner, Bill The Khronos OpenGL ARB Working Group, et al. *OpenGL programming guide: the official guide to learning OpenGL, versions 3.0 and 3.1*. Pearson Education, 2009. ISBN: 9780321552624.

[49] N. Koenig and A. Howard. "Design and use paradigms for Gazebo, an open-source multi-robot simulator". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2149–2154 vol.3. DOI: 10.1109/IROS.2004.1389727.

[50] Olivier Michel. "Cyberbotics ltd. webots™: professional mobile robot simulation". In: *International Journal of Advanced Robotic Systems* 1.1 (2004), p. 5.

[51] Edward M Mikhail, James S Bethel, and J Chris McGlone. *Introduction to modern photogrammetry*. John Wiley & Sons, 2001. ISBN: 9780471309246.

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

Chapter A

Appendix A — Included attachments

```
attachments.zip
├── ueds/
│   ├── Source code of the UEds simulator
│   └── README.md file containing the project instructions
├── ueds-connector/
│   ├── Source code of the UEds-connector (UEds simulator client)
│   ├── Source code of the UEds reinforcement learning framework
│   └── README.md file containing the project instructions
├── models/
│   └── Reinforcement learning models used in the Chapter 6
└── photos/
    └── Photos and plots used in this thesis
```

Assets (meshes, materials, etc.) used in the UEds simulator were obtained using the Unreal Engine marketplace. The assets can be used free of charge, but can not be published in their raw, unbuilt form. Therefore, we disclose only the source code of the UEds simulator and the drone model provided by the MRS Group. The full version of the simulator including the assets is stored in a version control system of the MRS Group.