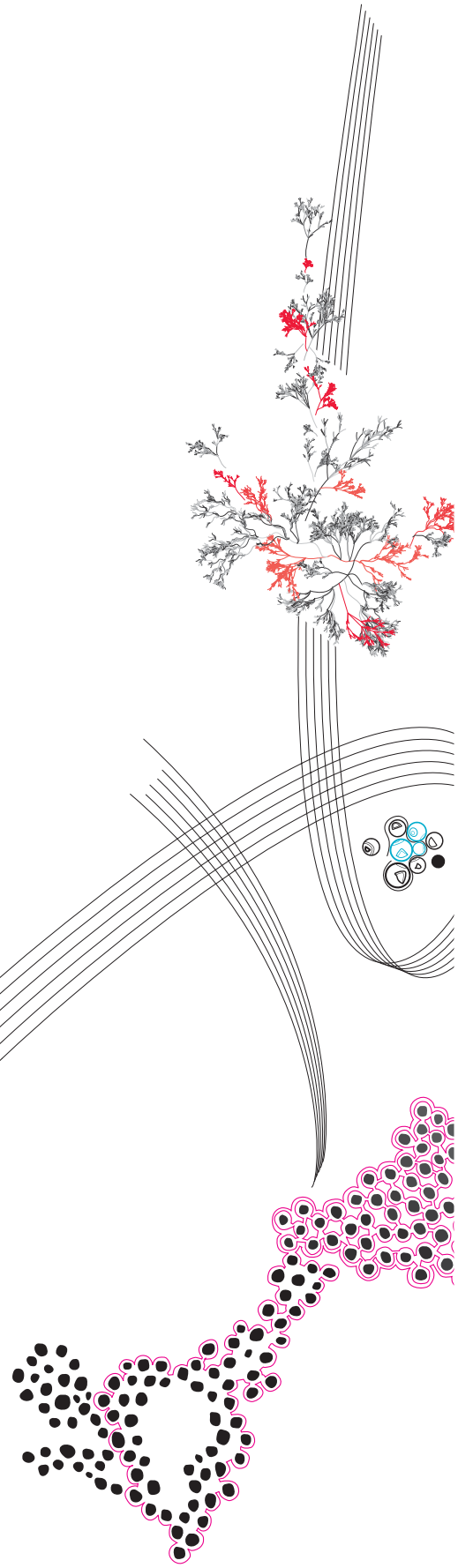

BSc Computer Science
Design Project

Drone Swarm Simulator

Raul Botea (s3145786)
Pedro Cecchini Said (s2960796)
Stella-Maria Horvath (s2998491)
Iván Peñate Gómez (s3163660)
Carim Popa (s3147851)
David Szentpeteri (s3148726)

Supervisor: dr.ir. A. Chiumento

April, 2026

Department of Computer Science
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Acknowledgements

We would like to express our gratitude to our supervisor, dr. Alex Chiumento, for his guidance, feedback and support throughout this design project. His supervision allowed for open and constructive discussions, which contributed to an enjoyable project experience.

Contents

Chapter 1

Introduction

The operation of drone swarms in semi-autonomous conditions presents a rather enthralling challenge, since each agent must navigate through a dynamic environment and interact with fellow agents. The complexity of coordinating a drone swarm, while simultaneously ensuring safety, efficiency, and adaptability, highlights the importance of this research area with applications in military, search and rescue (SAR), and logistics.

This project requests the extension of an existing two-dimensional simulation, as developed by [1], into a three-dimensional collision avoidance module. This system enables unmanned aerial vehicle (UAVs) swarm(s) to navigate towards a given destination while avoiding both static and dynamic obstacles, as well as one another. The goal of this project is to design, implement and validate the collision avoidance module and provide an easily adaptable framework for people to train efficient models, facilitating both the training and evaluation of further models.

The implementation builds upon the original Proximal Policy Optimization[1] (PPO)-based reinforcement learning (RL) approach and transitions to a multi-agent Soft Actor-Critic[2] (SAC) algorithm, which is supported by high-fidelity sensors within the 3D environment, namely LiDAR. Each drone operates as a semi-autonomous agent, meaning it has partial observability of its surrounding environment and must make decisions based purely on local information. Through this iterative training, the drones learn policies that allow them to avoid collisions, maintain a safe distance from each other, and reach destinations efficiently.

Upon completion of the project, the deliverables included an extensive 3D environment extended with the implementation of the reinforcement learning algorithm, as well as a model for collision avoidance of drone swarms within a 3D environment. This environment was carefully curated by ensuring the successful creation, consistency and collaboration of participating entities, as well as developing an environment suitable for RL to learn in. Furthermore, an implementation of SAC is implemented, along with an excelling expert policy for the purpose of speeding up training. Although the final trained model is not representative of pure RL and has

[1]Proximal policy optimisation (PPO) is a state-of-the-art deep reinforcement learning algorithm that trains agents by iteratively improving a policy with algorithmic stability as a key feature.

[2]Soft-Actor-Critic (SAC) is another state-of-the-art deep reinforcement learning algorithm that combines a critic learning the value of states and an actor that uses the estimated values to drive decision-making.

expert influence, it still demonstrates that learning and autonomy of drones is possible with a longer training time. This project therefore serves as a proof of concept, and provides the framework with which others may train an optimal 3D collision avoidance model.

The following report provides insight into the development of the collision avoidance module. Chapter 2 presents the functional and non-functional requirements, along with the constraints and assumptions of the project. Chapters 3 and 4 describe the system design, first as a high-level overview of the global design and then in detail, including key design choices and elements of the RL model such as the state representation and action space. The implementation is discussed as part of chapter 4. Chapters 5 and 6 focus on testing for the custom environment, wrappers, training processes and LiDAR, and reflect on the performance of the trained SAC model. A guide on accessing and reviewing the source code is given in chapter 7, while chapter 8 outlines the specific contributions of each team member during this project. Finally, Chapters 9 and 10 present the discussion and conclusion.

Chapter 2

Requirements Specifications

This chapter defines both the functional and non-functional requirements of the system, which were formulated alongside the client and their requests. Functional requirements outline what the system must be able to do, while non-functional requirements are focused more towards quality characteristics, such as performance, reliability and scalability.

The following list of requirements are prioritised using the MoSCoW method (Must, Should, Could, Won't), which represents their importance in the success of delivering a production-ready module.

2.1 Functional Requirements

2.1.1 Must Have Requirements

User-level

1. The system shall provide an interactive user interface that enables them to configure and execute drone simulations.
2. The system shall allow the user to configure one or more target locations, which the respective drone swarm will navigate towards.
3. The system shall allow the user to configure the number of static and dynamic obstacles within the environment.
4. The system shall allow the user to specify the number of drones in a swarm.
5. The system shall allow the user to specify the number of swarms in the environment.
6. The system shall allow the user to specify drone spawn locations.

System-Level Must Requirements

1. The system shall operate in a fully custom, three-dimensional simulated environment.

2. The system shall enable drones to start at a user-defined (or random) spawn location and navigate to a user-defined (or random) destination.

3. The system shall dynamically adapt drone paths to avoid both static and dynamic obstacles.

4. The system shall implement a real-time collision avoidance system capable of avoiding:

   - Dynamic obstacles
   - Other drones within the swarm

5. The system shall utilise LiDAR-based sensing to support real-time obstacle detection.

2.1.2 Should Have Requirements

1. The system should support changes in global parameters that affect the entire swarm simultaneously.

2.1.3 Could Have Requirements

1. The system could support the implementation of a swarm leader model, where an assigned drone coordinates communication or path guidance.

2. The system will maintain coordination among drones within a swarm and ensure that a maximum and minimum distance between drones is maintained.

3. The system shall allow the user to modify parameters that do not require retraining of RL models, i.e. speed, drag, gravity of a specific drone or as a whole swarm.

2.1.4 Wont Have Requirements

1. The system will not include physical hardware deployment or real-world UAV integration.

2. The system will not implement regulatory compliance mechanisms.

3. The system will not include advanced swarm communication modelling.

2.2 Non-Functional Requirements

1. **Performance:** The system shall support real-time simulation and collision avoidance for multiple drones and swarms operating simultaneously.

Due to a lack of time, the could have requirements were not fully implemented, and are thus features to include in future work. This was discussed with the client.

2. **Scalability:** The system shall support the increase in the number of drones, obstacles, world scale, swarms and targets (up to a defined limit) without significant degradation in performance.

3. **Reliability:** The system shall maintain stable operation under diverse environments with varying static and dynamic obstacles.

4. **Accuracy:** The high-fidelity sensors (LiDAR) shall provide a meaningfully accurate detection of the environment from the drone's point of view to allow for accurate obstacle detection and navigation.

5. **Modularity:** The system shall be designed in modular components to allow for the separation and ease of understanding of the interconnected components.

6. **Usability:** The user interface shall be intuitive, user-friendly and easy to navigate. Furthermore, the training set-up and other learning configurations are easily modified without requiring in-depth technical knowledge.

2.3 Constraints & Assumptions

2.3.1 Constraints

1. The system can only be implemented and tested in a simulated environment, and will not be validated on physical UAV hardware.

2. The system itself is constrained by the limitations of the simulation environment (i.e. GymPybullet Drones).

3. The training duration and accuracy is limited on account of the module period in which the system is to be developed.

4. Environmental conditions such as wind and turbulence are omitted for this project.

2.3.2 Assumptions

1. LiDAR rays give an accurate reading of the environment surrounding the drones.

    - Violation consequence: If violated, the observation space becomes unreliable, invalidating the learned policy.
    - Fallback: No fallback is implemented, the correctness is guaranteed by the environment. The observation vector per drone keeps track of nearest obstacle and neighbouring drone entities as well, which serves as a back up / check for the LiDAR.

2. It is assumed that each drone operates in the same manner and has the same capabilities as one another.

- Violation consequence: If violated, a lower success rate is expected.
- Fallback: The distance between the closest neighbour impacts the learned policy. The policy is learned under this assumption of symmetry, so performance may degrade if drones differ in capabilities or dynamics.

Chapter 3

Global Design

3.1 System Overview Diagram

The system was implemented with a focus on modularity. This was done with the purpose of allowing every member to independently develop and improve features within any of the system files, while also allowing future improvements to be done without unexpected errors arising.

The system is made up of multiple components that can best be grouped by logical functionality as seen in Figure 3.1. The diagram provides a high-level view of each logical component, the relation between them and the main external libraries used. The component order is meaningful, with components on the left being dependencies of those on the right. This is shown through the usage of association and dependency arrows combined with the provided/required interface commonly seen in component diagrams. At the top of the diagram the most important external libraries used can be seen, with dependency lines that show which components use them.

The "Training Pipeline" component contains files responsible for training the machine learning algorithm. The results of training are stored in a predefined location which forms the "Model Artifacts" component. The "Evaluation Pipeline" is closely associated to the "Training Pipeline" as it contains files responsible for evaluating the result of a trained model, and it does so by reading from the "Model Artifacts" component. The files are all connected to the "Drone Environment Core" component as they require a working simulation in order to function. The "Drone Environment Core" is a component that contains files responsible for creating a 3D environment and it glues together the whole project. In order to create all the necessary assets it borrows them from the "World Building Pipeline" which contains all the files related to object building, attributes and rules.

In the next chapter the following components are discussed together, in one single group, as on a higher-level they are all required to have a meaningful role for the system: "Training Pipeline", "Evaluation Pipeline", "Model Artifacts" are part of "Reinforcement Learning"; "User Interface", "Runtime Controller" are part of "Graphical User Interface"; "World Building Pipeline", "Drone Environment Core" are part of "3D Environment". These changes were not applied to the diagram, as this separation provides no additional insight into possible issues and potential

improvements compared to a slightly lower-level system overview.

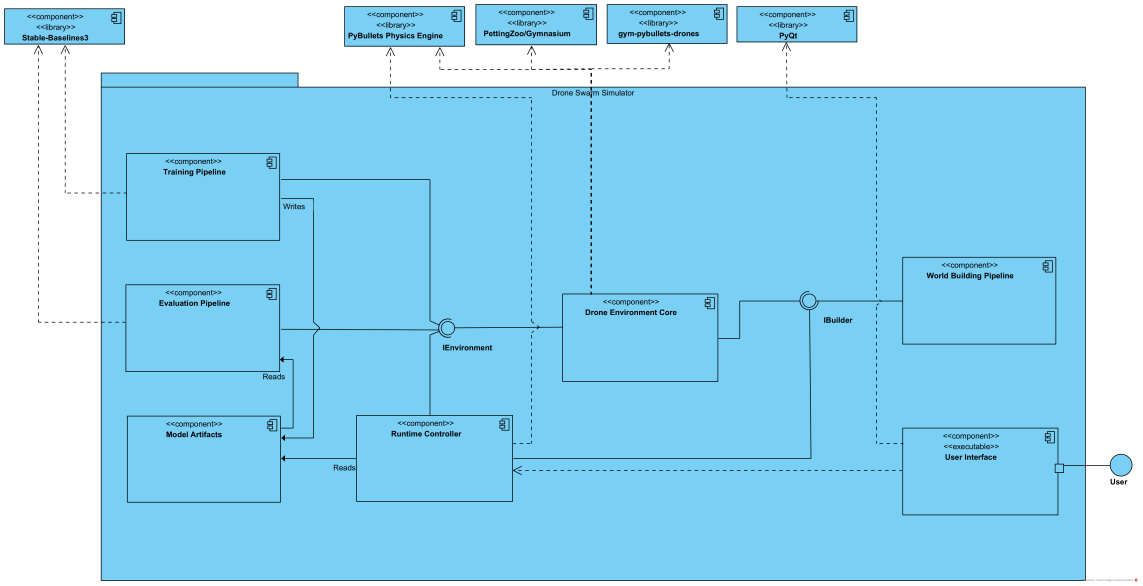

FIGURE 3.1: Component Diagram

3.2 Description of Major Components

3.2.1 3D Environment

To support both the training and the evaluation of the collision avoidance module, a custom three-dimensional simulation environment was developed, which is built on top of the `gym_pybullet_drones` framework. This library provides the base functionality of spawning an empty `PyBullet` environment containing drones, and the custom environment extends it by introducing swarm-spawning logic, target creation, static and dynamic obstacle generation, LiDAR integration, collision monitoring, and both step and episode bookkeeping customised for the RL training. This three-dimensional environment, implemented as `DroneSwarm3D`, provides a somewhat realistic setting, yet still computationally manageable as far as training is concerned for (multiple) drone swarms to learn to navigate and avoid collisions in.

Moreover, the `VelocityAviary` environment was chosen as the simulation pillar. In this environment, drones are controlled via velocity vectors given for each of the three spatial axes, i.e. [vx, vy, vz]. This environment was chosen because it is simple enough to be trained and controlled by an RL algorithm, yet still sufficiently robust to give good control of the drones.

The custom environment is implemented as a PettingZoo `ParallelEnv`, which allows all drones to act simultaneously at each timestep. This was essential for swarm navigation rather than single-agent observations, and allows for the modelling of multiple drones in a shared world and that they are purely interacting through the space, observations, shared obstacles and collision logic.

The three-dimensional world implemented by the environment is originally empty, only allowing for the specification of the number of drones and their spawn positions. This logic was extended to suit the spawning of drone swarms. Additionally, in order to create a more realistic training environment, functions were created for the spawning of static and dynamic obstacles. This was done using `Pybullet`, which provides useful methods for such tasks.

At a high-level, the custom three-dimensional environment is responsible for the following tasks:

- generating the three-dimensional world and the simulator
- spawning one or more drone swarms
- creating, spawning and assigning targets
- creating and spawning both static and dynamic obstacles
- constructing the observation vectors for each drone
- receiving and converting actions
- stepping the simulator
  - computing rewards at each step
  - bookkeeping statistics
- detecting collisions of drones with walls, other drones and obstacles

This exhaustive list proves the `DroneSwarm3D` environment to be more than just a wrapper around `PyBullet`. It is responsible for the orchestration of many internal processes that contribute to the success of training and evaluation. This combination of choices allows for a useful balance between realism and tractability of the models. The drones operate within a three-dimensional world governed by proper positioning, velocities, speeds, gravity and collisions, but the action space remain simple enough to make learning feasible.

3.2.2 Graphical User Interface

The Graphical User Interface (GUI) makes use of the environment provided by the `gym-pybullet-drones` library to show the simulation. This was not changed throughout the development as it ensures that the physics rules used to train the machine learning algorithm remain unchanged, resulting in a slightly improved performance.

The GUI is designed to be simple and easy to use. It starts with a configuration page where the user can modify almost all of the simulation environment parameters and select a model to control the drones. The user can then go to the simulation page where more detailed settings can be chosen. This page allows the user to modify attribute values of entities in the environment, play and replay a simulation, save and load preset environments, and see statistics of the most recent simulation run. Other features such as replaying simulations or filtering objects are also possible.

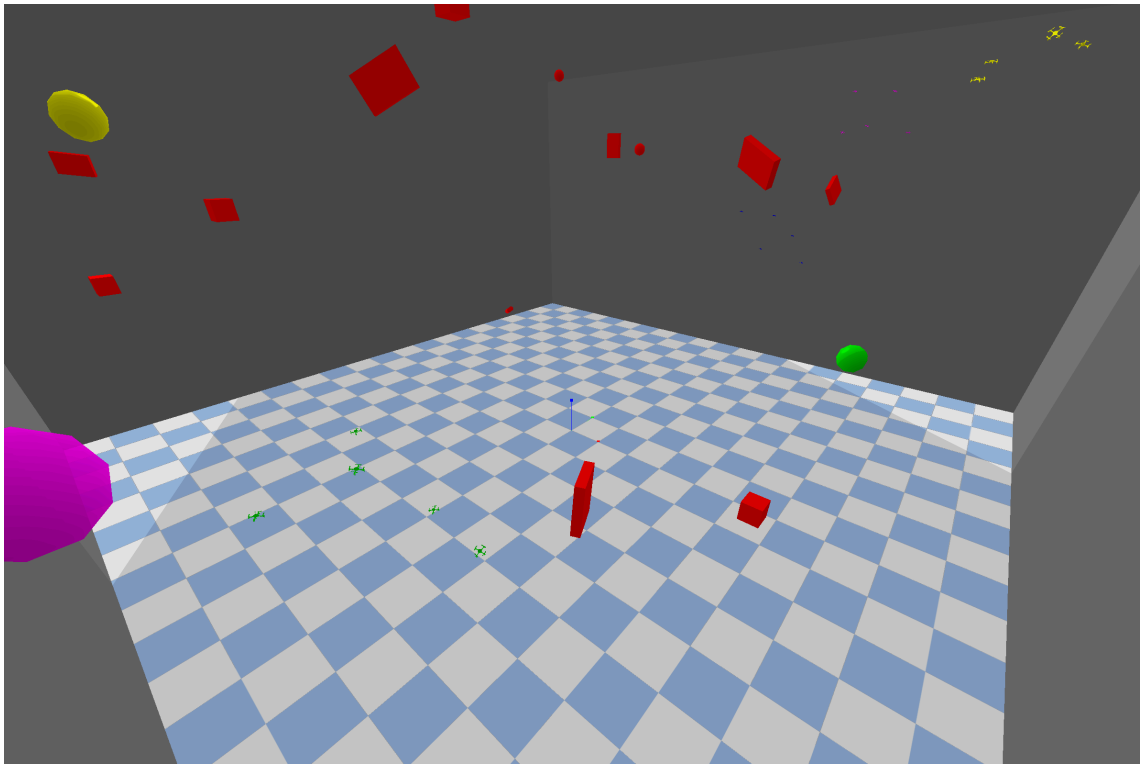

FIGURE 3.2: Visualisation of custom 3D environment. The circular and colourful spheres are targets for each drone swarm. These correspond according to colours, i.e. the blue swarm of drones need to navigate towards the blue target. The red objects of different shapes and sizes are the obstacles, and the grey surroundings represent the walls.

3.2.3 Reinforcement Learning Implementation

The (semi)-autonomous drones learn the collision avoidance behaviour within the environment using a reinforcement learning approach, specifically making use of the Soft-Actor-Critic algorithm. Essentially, the training is done in a centralised manner, meaning observations are continuously learned from all participating drones, and then executed in a decentralised way. Each drone is controlled by a shared policy trained to maximise the long-term reward.

To accelerate the learning process and improve the early-stage transitions from which the RL learns, an expert-guided action blending was also incorporated. During the initial steps of training, the action executed by each drone is representative of a weighted combination of the expert policy and the RL policy output. The influence is gradually reduced over time, allowing the agent to acquire some assistance at the beginning, while eventually relying entirely on learned behaviour.

Additionally, training is also carried out using a curriculum based learning strategy. This allows for a steady increase in the difficulty of the training environment without having to undergo vast exploration in the event that the drone is immediately thrown in a very complex space. Early stages involve simple scenarios with few drones, no obstacles and a smaller world scale, while later stages progress to spaces occupied by multiple drones and obstacles. At each stage, the trained model is trans-

ferred to the newly configured environment, and preserves all learned parameters and normalisation statistics.

The final output of the training is a shared policy that is capable of controlling multiple drones within a shared environment, with a prioritised feature of collision avoidance with walls, obstacles and other drones. Overall, the final model combines a state-of-the-art RL algorithm with training enhancements, including expert blending and curriculum training.

3.3 Chosen Tools

This section describes the tools, libraries and frameworks used in the system together with an explanation on why they were selected.

The simulation environment uses `PyBullet` and `gym_pybullet_drones`. `PyBullet` was chosen due to its lightweight physics engine, while `gym_pybullet_drones` was added as it extends `PyBullet` to work with drones, more specifically for quadcopters. In addition, `gym_pybullet_drones` is also compatible with reinforcement libraries such as `Stable-Baselines3` and `Gymnasium`. Although `PyBullet` offers less realistic physics compared to other more specialized libraries, it is easier to integrate and is more computationally efficient, both important requirements in order to reduce the time spent on training a model. The simulation created through the usage of `gym_pybullet_drones` has pre-built drone models, control interfaces and functionalities like zooming in and out, turning the angle of visualisation and moving across the 3D environment.

As an extension of the visualisation, an interface was created to set up the initial simulation, as a light application that had the minimum to work was needed. The GUI Toolkit used is `PyQt`, which works similarly to making an app by building up with pre-made blocks. This allowed having a working application that executed simulation early on. From there onward, it was very easy and modular to add functionalities and expand the interface as much as needed.

When training the model with Reinforcement Learning, `Stable-Baselines3` and `SuperSuit` were used. The former has a suitable for implementing the algorithm needed of SAC. The latter is used to preprocess and clean the environment between episodes and stages. When working on the system, it was also important to take into account that the system is for a swarm of drones, therefore a multi-agent system. To allow for a clear structure of the machine learning, given the complexity of managing the swarms, the library `PettingZoo` was used. The library is perfect for this type of system. It is a standard for multi agent RL which enables parallel agents running concurrently.

Lastly, GitLab was utilized to ensure that the latest version of the project is always accessible, minimize code conflicts and enable rollbacks in case of faulty updates. Furthermore, it also enabled task splitting and development to be more efficient through the usage of branches. Although not formally enforced, the main branch was reserved for working, error-free code; therefore, each member was responsible with analysis and testing the correctness of their branch.

Chapter 4

Design

4.1 Core Environment Components

4.1.1 Drone Spawning

The custom environment represents each drone as a `PettingZoo` agent with a fixed identifier. There is a list of `possible agents` in the world, which is essentially the full fixed-capacity list of drones that may be exposed, as well as list of `agents`, which contains all active drones during an episode. An active drone is essentially one that is given a target destination and participates by providing transitions for the purpose of RL training. To preserve compatibility with frameworks such as `PettingZoo` and `Stable-Baselines3`, the environment keeps a fixed maximum number of drone slots and flags the inactive ones with dummy positions far outside the actual world. It was required to maintain a fixed vector shape from which the RL model learns (i.e. prevents the curriculum stages from changing the vector shape of transitions that the RL model sees during training). The fixed-capacity was chosen to be 10, as this represents a sufficient swarm size in order to capture enough collisions and also allows for a reasonable training time.

Unlike other objects in the simulation, the spawning of the drones is handled by `VelocityAviary`. The goal of this project is obstacle avoidance for drone swarms; however, there is no pre-defined function for swarm spawning, so specific logic was introduced.

The first step is to generate a point as the designated centre of the swarm, and, given the swarm radius, generate random spawn positions for drones within the area. Nonetheless, it must satisfy constraints, such as being sufficiently far from world boundaries and ensuring that drones maintain a minimum separation between them to avoid collisions shortly after spawning. This process can be repeated for the total number of swarms.

Additionally, the environment also tracks episode-specific drone state through boolean arrays that record whether

- a drone has collided or is considered dead (collided / inactive)
- a drone has already reached the target
- a drone has been visually despawned from the GUI simulation

Finally, it was chosen to set the default speed of a drone to 3 meters per second. This was chosen after doing a manual grid search among speeds within the range 1 to 10 meters per second, and 3 proved to be the most stable, especially for training. It was observed that any faster speed caused the drones to go head first into the ground or lose control and spiral into obstacles.

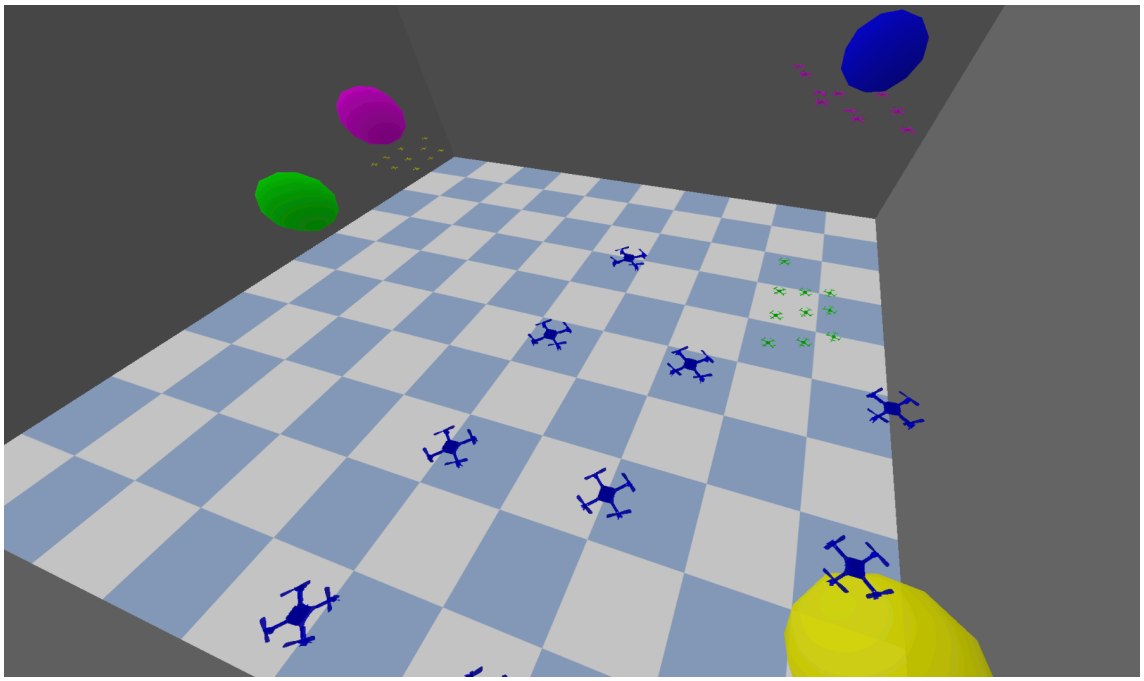

FIGURE 4.1: Visualisation of the initial spawning of drone swarms in the environment.

4.1.2 Obstacle & Target Spawning

The spawning of obstacles is handled by another class called `EnvObject`. This class follows Object Oriented Programming (OOP) design, making it more modular and easily reusable. Furthermore, it allows for the abstraction of information for users and makes it easily extendable and flexible for different tasks, not just for the training of an obstacle-avoiding drone.

The `EnvObject` class has predefined parameters for cubes, spheres, walls, and cylinders. It was implemented in a way that no internal knowledge is needed to spawn an obstacle; simply specify the shape, position, size, and orientation. Additionally, increasing the options of obstacle types is simple, only requiring a few modifications. The class also has helper functions to ensure the object is being placed in an adequate position. This includes functions to check whether the object is sufficiently far from drones and targets, and to verify that objects do not overlap and are not on the paths of dynamic obstacles.

To spawn dynamic obstacles, an additional class was created that extends `EnvObject`, meaning it inherits the spawn position check functions, however, it takes a few more arguments, such as speed and other variables, to determine its path. Furthermore,

due to the way the `PyBullet` library works, in order to make a dynamic obstacle, every step of the simulation, the object must be manually moved, thus, helper functions were made to carry out this logic as well, namely `update_obstacle()`.

Each swarm is assigned its own target. This is also easily noticeable on the GUI as being the same colour as that of a swarm's drones. The generation of targets are done in such a manner to create a meaningful traversal of the world: they are spawned on the opposite side of the world as the swarm spawn to encourage further exploration. Certain validity checks are also in place to ensure proper spawning. A target may not overlap with drones, obstacles and walls, and it must also maintain a strict distance away from these entities.

The final class added was `Target`, which as its name suggest is responsible for spawning targets for drones, which are spheres. However, the targets are different than other obstacles as they do not have a collision area, and are only visually displayed in the GUI. This decision was made early in the project, before the RL training was implemented. Keeping the target with no hit-box allows drones to fly inside the target, which prevents them from crashing into other drones once they reach the target and freeze. Nonetheless, for this to work, the target has two parameters, `TARGET_RADIUS` and `TARGET_REACH_RADIUS`. The former influences the size of the target, it is important to have this sufficiently large, otherwise drones are trying to reach a rather small area, which causes crashes near the target. The latter is the distance from the centre of the target, which drones need to be to officially "reach" the target. This variable is always smaller than the `TARGET_RADIUS`, ensuring the logic explained above works.

4.1.3 World Configuration

The simulated world is a bounded three-dimensional space which is adjusted using the `world_scale` parameter. All drone swarms, targets and obstacles are spawned within the confined space, while 6 virtual wall objects are also added to represent the floor, ceiling and adjacent sides.

Upon simulation initialisation, the world is empty with only the drones spawning. This is then extended in a structure promoting modular coding to include:

- swarm spawn generation
- target placement
- static and dynamic obstacle placement
- wall creation
- LiDAR visualisation (which is optional)

4.1.4 LiDAR Sensing

LiDAR is the high-fidelity sensor used within the environment by each drone for the purpose of directional range sensing. This allows for a significant improvement in obstacle detection.

The LiDAR model works by precomputing a set of ray directions with respect to the drone's body, which are configurable through the view field of the horizontal direction, the view field of the vertical direction, the number of rays, and the number of vertical layers.
For each ray, the environment records a hit value from zero to one.

- A value near one suggests that no obstacle is detected within that direction's ray.
- A value near zero implies that an object is detected nearby.

These values constitute the rest of the observation vector, i.e. following the base vector as can be viewed in Table 4.2. Rays are optionally cast into the GUI for visualisation using `PyBullet`'s batched ray testing. Hit rays and free rays are given different colours to make distinguishing easier.

Overall, the inclusion of LiDAR was especially valuable for training, as it allowed for a greater spatial sensing and understanding of the environment, instead of purely relying on the nearest obstacle positioning.

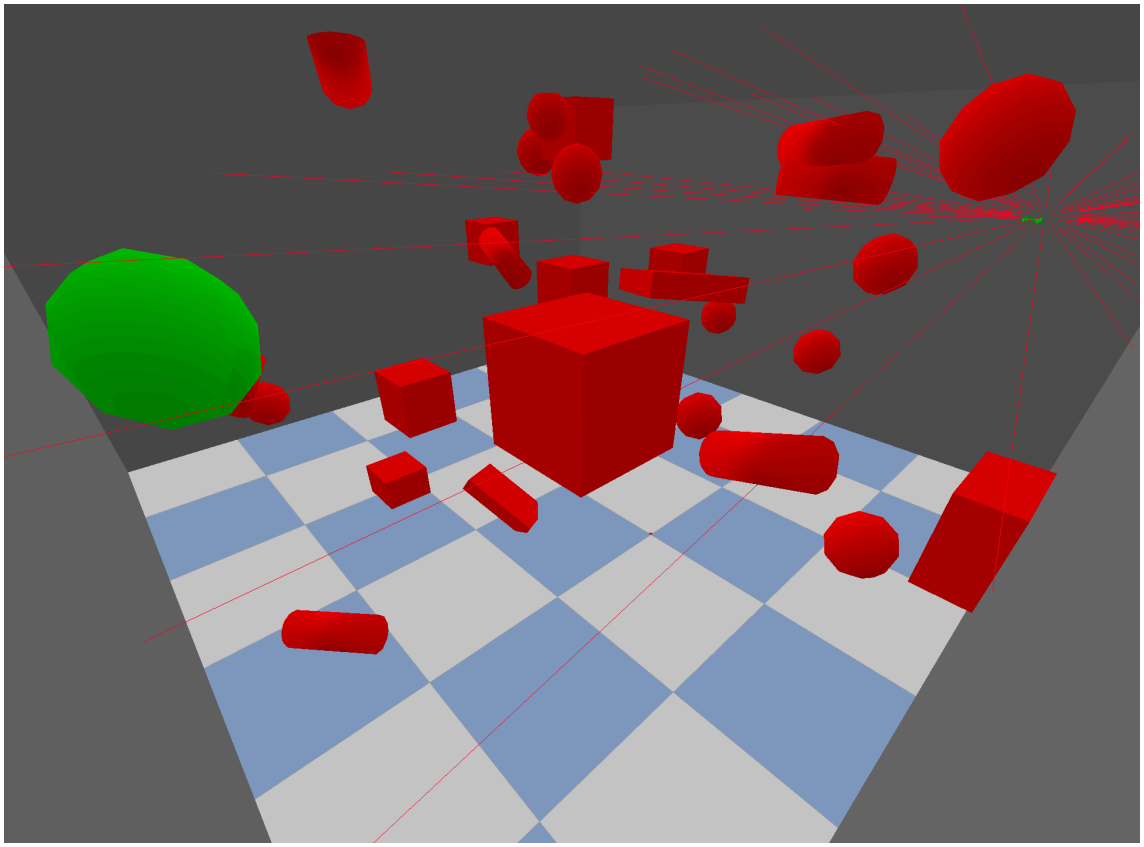

FIGURE 4.2: Visualisation of the LiDAR rays emerging from a drone.

4.1.5 Despawning Logic

In order to avoid the contamination of statistics with dead or inactive drones transitions and to not see these drones within the simulated GUI environment, it was

necessary to ensure that despawning logic was implemented. When a drone reaches its target or collides, it is despawned. This process does not occur by deleting the drone body, but rather by moving the drone to a very far location outside the confined world space and setting their speed and positioning to a zero vector. In this manner, they do not contribute any useful transitions for the RL to mistakenly learn from and it avoids having to reconstruct the simulator. It also preserves a stable simulator structure and fixed drone indexing, which is essential for multi-agent RL training.

As far as the RL model is concerned, the presence of both active drone and inactive drone transitions meant that the replay buffer using during training contained a lot of useless transitions, and that when batch sampling was done to update the network gradients, a large portion was dominated by inactive drone transitions. For this reason, a drone is clearly marked as being inactive and active to allow for a custom replay buffer to be designed from which only active drone transitions are uniformly sampled from. This is further discussed in the training section.

4.1.6 Training-Specific Environment

While creating an environment for training the model, it is important to include varying situations, so that it can learn from many different experiences and adapt to unforeseen circumstances. In order to replicate this, between episodes, the drone, target, and obstacle positions change. It is important to mention that there is a correlation between the swarm and its target spawn location. Drone swarms are created in one of 4 edges of the world (+x, -x, +y, -y) and their respective target spawns on the opposite edge. For example, if a swarm spawns in +x edge, its target will always be in -x edge, however, the y and z values for this swarm centre and target are still randomized. Also, to ensure both spawn in an acceptable position, variables like `SWARM_EDGE_MARGIN` and `TARGET_EDGE_MARGIN`, which set a minimum distance from the world boundaries at which targets and swarms can spawn. The motivation for this is to spawn drones as far away as possible from their target, forcing them to travel a longer distance and encounter more obstacles along the way, in an attempt to increase the exploration done by drones.

Another approach to optimize the training diversity is with the spawn position of obstacles. Randomly creating a place for them to spawn ensures many different scenarios for the model to learn from. However, during early implementation, it was noticed that most of the time, even with a large number of obstacles, there would still be a safe, straight path from the drones to their targets (optimal path). Thus, to make the environment more beneficial for learning, a new type of obstacles was used, called *blocking obstacles*. When spawning static obstacles, there is a chance, defined by the user, for them to be this new type. The only difference is that *blocking obstacles* are spawned randomly in the optimal path of a swarm, forcing drones to avoid them.

When generating obstacles for the training environment there are necessary parameters that the obstacle classes need to take in, as previously mentioned. However, to introduce more variability into the training, these values are generated each episode from a pre-determined range. Such values include:

- Size
- Orientation
- Speed (for dynamic obstacles only)
- Displacement (for dynamic obstacles only)

This allows for more efficient training, but also makes it adjustable for future use. For example, when first implemented, the speed of dynamic obstacles was set too high, which was only realized later during training. Nonetheless, adjusting and testing new speeds is easy, due to the design choices.

Although most of the training was done in randomly generated environments, certain more realistic tasks were taken into account. In order to train the drones on more interesting and complex movements three custom environments were created: a hallway, variations of rectangles in walls and a maze.

The hallway environment allows drones to reach the target exclusively through a tight passage. The environment aims to improve the coordination of drones when clustered together, while also reducing the collision rate between drones under any circumstances.

The rectangles in walls environments restrict the possible paths to the target to gaps in walls. The environment aims to improve the drones ability to manoeuvrer between large obstacles that are close to each other. This mimics slightly a zig-zag movement, which would be common if a city were to be modelled in the environment.

The maze environment restricts the drones to a single possible path, with all other potential paths resulting in dead ends. The environment aims to combine the two other environments and provide both benefits at the same time.

4.1.7 Collision Query World

The `CollisionQueryWorld` class provides a lightweight, headless `PyBullet` physics instance running in `DIRECT` mode (no GUI, no gravity) that exists purely for collision geometry queries. Its purpose is entirely separate from the main `VelocityAviary` simulation: rather than stepping physics, it is used to test whether object geometries intersect at arbitrary positions and times, without disturbing the live simulation state.

On initialisation, the query world opens its own isolated `PyBullet` client connection. Objects are registered into it in three categories: static obstacles, dynamic obstacles, and temporary bodies used transiently for a single query. Each object is reconstructed from its `collision_shape_spec`, meaning only the collision geometry is mirrored over, while visual shapes are omitted entirely.

The primary use case is swept-volume collision checking for dynamic obstacle placement. When a new dynamic obstacle is being spawned, `candidate_collides_over_time()` is called, which adds the candidate object as a temporary body, then steps through a short time horizon, advancing both the candidate's and all registered dynamic obstacles' poses via `pose_at_time(t)`. At each step, `getClosestPoints` is queried between candidate and every dynamic body using a configurable margin. If any intersection is found, then the candidate is rejected. The temporary body

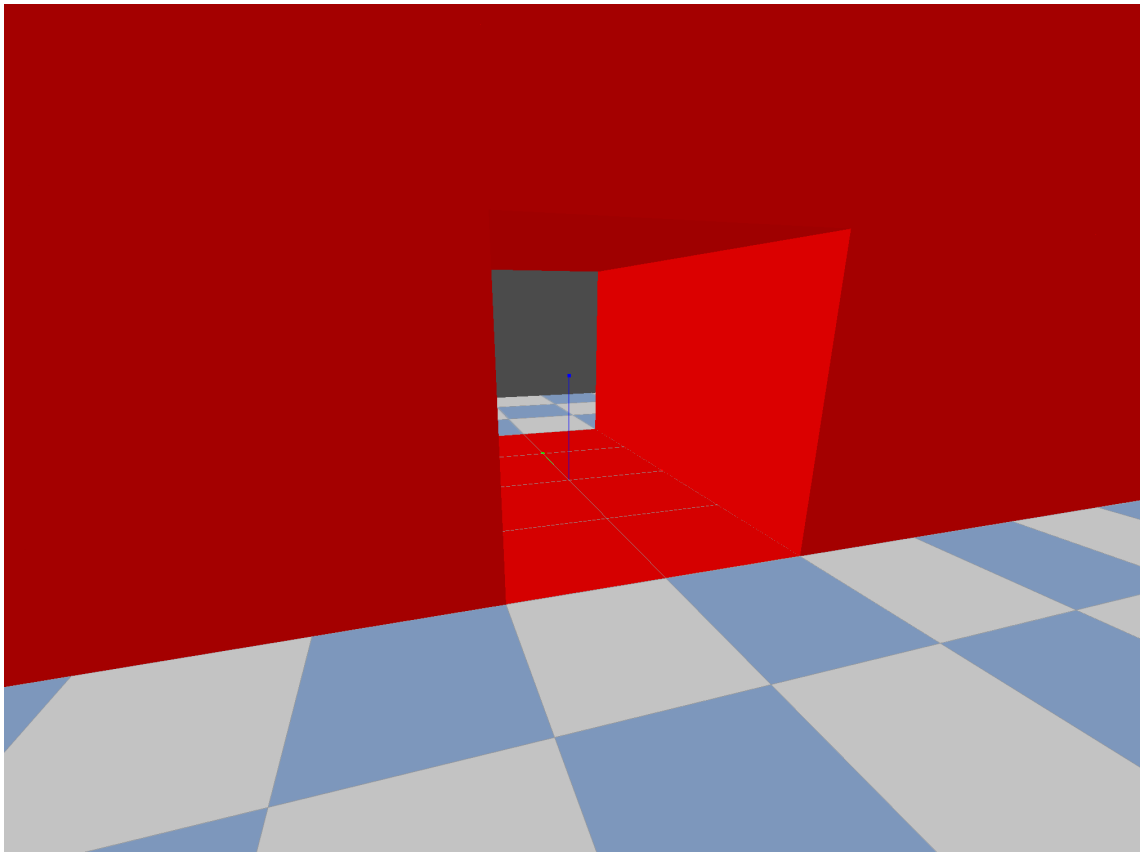

FIGURE 4.3: Visualisation of the hallway complex environment.

is removed in a `finally` block regardless of the outcome, keeping the query world clean.

Between episodes, `_rebuild_query_world_from_current_scene()` is called, which clears and repopulates the query world to match the freshly spawned obstacle set. This is also triggered after each dynamic obstacle is placed during `_spawn_obstacles()`, so that subsequent candidates are checked against an updated scene. The `clear()` method handles teardown by disconnecting the client and reconnecting a fresh one, avoiding residual body state from prior episodes.

The key design benefit here is the clean separation between simulation and querying: the main `PyBullet` client handles the live physics steps and rendering, while `CollisionQueryWorld` handles all speculative collision queries independently. This avoids the need to temporarily add and remove bodies from the live simulation during spawning.

4.1.8 Collision Avoidance Logic

The environment distinguishes between three different collisions:

- Collisions with the world edges, i.e. walls.
- Collisions with other drones (either in a drone's own swarm or with drones from other swarms)

- Collisions with static and dynamic obstacles

For each active drone, the environment queries `PyBullet` to check whether collisions have occurred among the object bodies of the drones, obstacles and walls. These results are stored as boolean arrays, and allow statistics to be collected such as:

- Whether a drone has collided during an episode
- Whether a drone just collided at the given step
- What type of collision was detected
- Which drone was involved with the collision

4.1.9 Step Logic and Episode Flow

The custom three-dimensional environment follows a structured per-step execution flow in order to ensure that statistics and records remain consistent across the simulation, agent behaviour and RL training. At every time step, a fixed sequence of actions are performed to either update the world specifications, process recent actions, evaluate these actions and prepare for successive states.

At the beginning of each step, the internal step counter is incremented and the dynamic obstacles are updated (in order to simulate their movement within the environment). This order also allows for an obstacle to be updated before the drone acts. Next, the actions stemming from any inactive drones are frozen, and all their next actions are forced to be zero-motion commands so that it no longer influences the simulation and contaminates further statistics.

After this, the action produced by the policy is converted into a simulation action, i.e. VelocityAviary commands. This serves as a bridge between the learned policy and simulator dynamics.

Once the physics has been incorporated and the step occurs in the environment, the `observation_dict` is created, whose structure is outlined in Section 4.3.1 as the state space. This is passed to a separate class in order to compute step-state logic and evaluate the current spatial configuration of the swarm in question. Each type of collision, that being with the wall, a drone or an obstacle, is tracked separately so that the environment can provide a more detailed diagnostic information. When a collision occurs, they are marked as dead and despawned from the simulation interface. Similarly, if the drone has reached the target, they are marked as successful and are also despawned.

Furthermore, pairwise drone distances between drones in the same swarm are also recorded to check when the swarm has become too dispersed or too compacted. This was initially added with the purpose of allowing the reward function to potentially account for inter-swarm cohesion penalties or bonuses. Based on the resulting step state, the environment also computes rewards, terminations, truncations and an information dictionary containing plenty auxiliary information, as shown in Table 4.1. An episode is considered finished once all drones have either collided or reached the target. This logic, is however, changed for during training, which is

elaborated in Section 4.2.9. Additionally, truncation is triggered when the maximum allowed steps is attained during an episode. It was important to define two different terminations, namely the task-related ones and the time-related ones, in order for the RL algorithm to distinguish between them and reward accordingly.

Overall, the step state logic acts as the central control of the environment. It is responsible for coordinating control of the simulation, updating the necessary parameters and variables when necessary and assisting the RL training significantly through the use of computed rewards and drone statuses.

Category	**Records**
General	`drone_idx`, `step`, `active`
Collision (overall)	`collided`, `wall_collision`, `obstacle_collision`, `drone_collision`
Collision (current step)	`collision_now`, `wall_collision_now`, `obstacle_collision_now`, `drone_collision_now`
Goal tracking	`reached`, `reached_now`, `dist_to_target`
Motion	`cmd_speed`, `actual_speed`
Swarm metrics	`swarm_max_sep`, `swarm_min_sep`, `swarm_too_far`, `swarm_too_close`
Episode status	`episode_done_collision`, `episode_done_all_reached`

TABLE 4.1: Summary of `info_dictionary` contents

4.1.10 Statistics and Logging

The environment supports the inclusion of per-episode statistics tracking. This records per-step and per-episode statistics to a CSV file, allowing the team to observe a continuous record of rewards, collisions, active drones and step state outcomes. Such logging is important to the success of the RL development as relying purely on training performance is not enough and does not assist in guidance on how to modify / create the reward function.

4.2 Reinforcement Learning

4.2.1 State Representation

The observation space for this environment is given per drone and it includes all relevant data an agent needs to describe their current situation. For this project, the observation space is an 18-dimensional vector, plus the number of LiDAR rays, which are adjustable. This vector includes the drone's position in the world [x, y, z], velocity [vx, vy, vz], and vector distance from the target. Additionally, it has the vector distance to the nearest obstacle, nearest drones, and vector distance from the swarm centroid. The latter was included to be used in the reward function

to ensure drones fly as swarms and not individually. Moreover, the distance to the nearest obstacle and the drone are needed for the implementation of expert blending. Finally, it includes LiDAR data, which, per ray, returns a number between 0 and 1. Zero, meaning there is an obstacle extremely close to the drone, and 1, meaning the LiDAR ray is not hitting anything.

LiDAR was added to the drone's observation space to enrich the available data. The environment the drone traverses is too complex to be navigated without additional sensing mechanisms to perceive its surroundings. The other option was to make the drone simply know the position of every drone and obstacle, which would result in a large observation space with a lot of irrelevant information, if no obstacles or drones were near. Additionally, it would be unrealistic for a drone to have such information, thus making the results of this project not applicable to real-world situations.

If a drone is inactive, dead or despawned, it receives an observation vector of zero. This preserves a consistent observation shape while clearly marking the drone as a non-participant in the world.

To provide the reader with a simple setup of the base of the observation vector, Table 4.2 shows these inclusions.

Component	**Dim**	**Description**
Drone position	3	Current position of the drone in world coordinates
Drone velocity	3	Linear velocity extracted from the simulator state
Vector to target	3	Relative vector from the drone to its assigned target
Vector to nearest obstacle	3	Relative vector pointing to the closest obstacle
Vector to swarm centroid	3	Relative vector to the centroid of active drones in the swarm
Vector to nearest teammate	3	Relative vector to the closest active drone in the swarm
Total	**18**	

TABLE 4.2: Base Observation Components (excluding LiDAR observations)

4.2.2 Action Space

The action space defines the control interface that the reinforcement learning agent uses to interact with the simulation environment. For this project, a continuous space making the moves smoother rather than a discrete one with more abrupt movement as this is more unrealistic.

The action space used was composed of four dimensions and a speed vector, $action = [\text{dir}_x, \text{dir}_y, \text{dir}_z, \text{speed}]$. Here, the three directions represent a normalized direction vector indicating the desired flight direction, and speed is a scalar value in the range $[-1, 1]$ where -1 corresponds to minimum speed and 1 corresponds to

maximum speed. The way speed is applied is simply by calculating the dot product of speed to the directions. $[v_x, v_y, v_z] = \text{speed} \cdot [\text{dir}_x, \text{dir}_y, \text{dir}_z]$. Once the dot product is made, the values are normalized to the physical velocity restrictions of the drone and thus suitable for `VelocityAviary`.

This separated representation of directions and speed was used for several reasons. First, separating direction from speed allows the agent to learn these aspects independently, reducing the complexity of the learning problem. Second, the normalizing the direction vector improves consistency with the orientation, preventing random changes in direction. Finally, having this format is easier to integrate to curriculum learning, where direction can be guided by predetermined rules at the start of the training while speed is learned, then directions are learned together with speed.

All action components are bounded to $[-1, 1]$, which aligns with the output range of the reinforcement learning algorithm used, namely Soft Actor-Critic (SAC).

4.2.3 RL Algorithm

The initial simulator in a two-dimensional environment, of which this project is an extension, utilised the Proximal Policy Optimisation (PPO) multi-agent reinforcement algorithm [1]. PPO is a widely used on-policy [1] gradient method best known for its training stability, and was successfully applied to the multi-agent drone navigation and collision avoidance task, as demonstrated in [1]. A study in which three different deep reinforcement algorithms, namely Deep Q-Networks (DQN), PPO and Soft-Actor-Critic (SAC), were compared for vision-based navigation of UAVs and collision avoidance with both static and dynamic obstacles also proved PPO to be a strong performer in this domain [2].

Despite these advantages, PPO also has certain limitations as far as this project is concerned. Firstly, being an on-policy algorithm, PPO requires newly collected transitions for each update, which ultimately results in a lower sample efficiency. This introduces problems such as longer training times in complex three-dimensional environments with multiple agents navigating simultaneously. [2] highlights that of the three tested algorithms, PPO performed the poorest due to the weaknesses of an on-policy algorithm in an extensive 3D environment with dynamic entities.

To address these limitations, it was decided to select the SAC algorithm for the purpose of this project. Given the complexity of the environment and the multi-agent setting, SAC is the better performer when compared to algorithms such as DQN and PPO [1] [2]. SAC is an off-policy algorithm that focuses not solely on maximising the long-term rewards but also the entropy of the policy [2]. This allows for a balance between exploration and exploitation. SAC also operates over a continuous action space, making it well suited for the velocity-based control that is used within this project.

In short, SAC operates with an actor-critic structure. The critic returns the

[1] An on-policy algorithm essentially learns from its own policy, i.e. the actions that the drone just performed by following the guidance of the RL policy, while an off-policy algorithm utilises all past experiences during learning with the aid of a replay buffer that stores these historical transitions.

estimated discounted value of the cumulative long-term rewards, while the actor simply returns the action that would maximise the discounted value. The actor makes use of a policy gradient method during training, and instead of purely going off of the rewards, it learns by studying the critics' feedback. Similarly, the critic learns through a value-based approach so that it can provide the feedback and criticism to the actor [2].

4.2.4 Reward Function

The reward function was carefully designed at the beginning to embody all possible aspects that the team wanted to control during training, such as the target reaching, collision avoidance and team coordination. Therefore, it initially consisted of multiple components that capture progress, safety constraints, control and swarm structure. This section briefly discusses the original design as well as the final chosen implementation.

Progress-Based Reward When a drone has neither collided nor reached its target, it receives a reward based on its progress towards the goal. This is defined as the reduction in distance to the target between consecutive timesteps:

$$r_{\text{progress}} = k_{\text{prog}} \cdot (d_{\text{prev}} - d_{\text{current}}) \tag{4.1}$$

This encourages the drone to continuously move closer to its assigned target.

Shaping Penalties Several penalty terms are applied at each timestep to discourage undesirable behaviour:

- **Distance penalty:** discourages remaining far from the target

$$r_{\text{dist}} = -k_{\text{dist}} \cdot d_{\text{current}} \tag{4.2}$$

- **Time penalty:** encourages faster task completion

$$r_{\text{time}} = -k_{\text{time}} \tag{4.3}$$

- **Control penalty:** discourages large or aggressive actions, where $\|\mathbf{a}\|$ is the euclidean distance of the action

$$r_{\text{ctrl}} = -k_{\text{ctrl}} \cdot \|\mathbf{a}\| \tag{4.4}$$

- **Alignment penalty:** penalty for not being aligned with the direction in which the target lies

$$r_{\text{align}} = -k_{\text{align}} \cdot \left(1 - \frac{\mathbf{a} \cdot \mathbf{d}_{\text{target}}}{\|\mathbf{a}\| \, \|\mathbf{d}_{\text{target}}\|}\right) \tag{4.5}$$

- **Obstacle proximity penalty:** penalises flying too close to obstacles

$$r_{\text{obs}} = -\frac{k_{\text{near}}}{d_{\text{obs}}} \tag{4.6}$$

- **Cohesion penalty:** penalises drones spreading too far apart, where $\Delta_{\max}$ is the maximum distance between two drones in a swarm

$$r_{\text{cohesion}} = -k_{\text{coh}} \cdot \Delta_{\max} \tag{4.7}$$

- **Separation penalty:** penalises drones being too close to each other, where $\Delta_{\min}$ is the minimum distance between two drones in a swarm

$$r_{\text{separation}} = -k_{\text{sep}} \cdot \Delta_{\min} \tag{4.8}$$

- **Zero-speed penalty:** discourages hovering without progress

$$r_{\text{zero}} = -k_{\text{zero}} \tag{4.9}$$

Terminal Rewards Certain one-time bonuses are given when key events occur:

- **Wall collision:**

$$r = -k_{\text{wall}} \tag{4.10}$$

- **Drone collision:**

$$r = -k_{\text{drone}} \tag{4.11}$$

- **Obstacle collision:**

$$r = -k_{\text{obs-coll}} \tag{4.12}$$

- **Goal reached:**

$$r = +k_{\text{goal}} \tag{4.13}$$

- **Truncation:**

$$r = -k_{\text{trunc}} \tag{4.14}$$

Overall Reward The total reward at each timestep is then given by the sum of all above components:

$$r = r_{\text{progress}} + r_{\text{dist}} + r_{\text{time}} + r_{\text{ctrl}} + r_{\text{obs}} + r_{\text{cohesion}} + r_{\text{separation}} + r_{\text{zero}} + r_{\text{align}} \quad (4.15)$$

with terminal rewards overriding the sum when collisions or goal completion occur, i.e. they are occur one-time.

Optimizing the reward function so it is suitable for the entire training is not a trivial task. The rewards and their coefficients should be adapted to match the current goals, so they were implemented in a way that is easy to change. The rewards implemented were based on other SAC implementations for drone collision avoidance [1] [3] [4], but it is important to note that not all rewards were used.

Once the reward function was tested and established, the coefficients representing the penalties and bonuses were tuned via a long process of testing as well as trial and error. The most optimal tune found during the limited training time for these coefficients was a soft time penalty with a harsh negative reward for collisions, a slightly less harsh negative reward for timing out, a large reward bonus of reaching the target and reward based on the distance of the respective drones from the target minus a positive reward for progressing towards it and a negative from getting further from the target. The time penalty and the time out penalty were decided not to be so harsh as to give the drones time to explore and since it was decided the drones not colliding were a bigger priority than the drones reaching the target faster. The collision penalty both with walls and obstacles were set quite high since them not reaching the target is an instant failure. Lastly, a small alignment penalty also helped to contribute to the learning. All of these respective coefficient values used in the final trained model can be found in Table 4.3, and the function itself was then:

$$Reward_{\text{Function}} = r_{\text{progress}} + r_{\text{align}} + r_{\text{time}} r_{\text{dist}} \quad (4.16)$$

This is of course including the one time values for the collisions and goal reaching events.

Term Coefficient	Coefficients Value
k_{prog}	25
k_{dist}	0.03
k_{time}	-0.03
k_{align}	0.05
$k_{\text{drone}}/k_{\text{wall}}k_{\text{obstacle}}$	-50
k_{goal}	250
k_{trunc}	-40

TABLE 4.3: Reward Function Coefficients

4.2.5 Training Curriculum

The final curriculum has the following parameters:

- `world_scale` - indicates the size of the environment
- `num_drones` - the number of drones in the environment at the same time
- `num_static_obstacles` - the number of non moving obstacles in the environment
- `num_dynamic_obstacles` - the number of moving obstacles in the environment
- `stage_training_timesteps` - the maximum number of time steps per episode
- `episode_horizon` - the time spent on that stage of training
- `lidar_enabled` - indicates if LiDAR is turned on for the drones

The curriculum was designed to start training with multiple drones so that the shared policy receives multiple useful policy samples per environment step. Throughout the training, both the number of drones was increased, as were the other parameters of the environment. Each stage of the curriculum, the environment was incrementally made harder, while the time per episode was increased proportionally to the difficulty of the environment. Designing the curriculum and finding the most optimal training path, similar to hyperparameter tuning was based on testing and trial and error.

Table 4.4 provides the curriculum used for the training of the pure SAC model, while table 4.5 provides the curriculum used for the training of the model with an initial 80% expert influence.

world_scale	num_drones	num_static_obstacles	num_dynamic_obstacles	stage_training_timesteps	episode_horizon	lidar_enabled
2	1	0	0	100000	2300	True
2	1	1	0	200000	3000	True
3	1	0	0	100000	2300	True
3	2	1	0	200000	3000	True

TABLE 4.4: Simple Curriculum

world_scale	num_drones	num_static_obstacles	num_dynamic_obstacles	stage_training_timesteps	episode_horizon	lidar_enabled
3	3	0	0	300000	1500	True
3	5	0	0	350000	1700	True
3	5	2	0	400000	1700	True
4	5	3	0	450000	2000	True
4	5	4	1	450000	2200	True
5	5	5	1	500000	2500	True
5	8	5	1	500000	2700	True
6	8	6	2	550000	3000	True
7	8	8	2	550000	3300	True
8	10	8	3	600000	3600	True
9	10	10	4	700000	3900	True

TABLE 4.5: Curriculum used for the training of the model with 80% initial expert influence.

4.2.6 Custom Replay Buffer

A custom replay buffer was introduced to better handle the complex multi-agent environment. In standard SAC, the replay buffer is used to store the transitions of drones collected during their interactions with the world, and is then uniformly sampled from in batches during learning. Due to the fact that there are dead, despawned and inactive drones present in the environment, not all stored transitions contribute meaningfully to the learning, and thus impact the success thereof.

To address this issue, the replay buffer was extended such that every stored transitions is marked as either stemming from an active, participating drone or an inactive drone at the time of insertion. This is done through an additional boolean mask, referred to as the active mask, which is stored alongside the usual replay buffer contents.

During sampling, the custom replay buffer then uniformly selects from only the transitions marked as active, i.e. the algorithm only learns from experiences generated by active participants in the environment. This prevents the RL from learning transitions that carry little or no behavioural value, and from repeatedly sampling zero vectored actions that introduce noise into the training.

4.2.7 Expert Policy and Action Blending

To accelerate the learning phase and stabilise training, an expert-guided action mechanism was created to be used within the SAC framework. Rather than allowing a high-intensity of exploration at the beginning, the expert blending assists the agent in completing their necessary tasks.

The expert policy is simply a controller that maps a raw observation to an action. It is designed in such a way to allow for reasonable navigation towards the target and collision avoidance behaviour with all environment entities.

The expert itself works using a priority-based approach. With a high-level overview, it operates by combining the following components into a single direction vector:

- movement towards the target,
- obstacle avoidance,
- LiDAR-based environmental awareness,
- wall avoidance,
- teammate avoidance,
- adaptive speed control.

The core objective is to guide the drones towards the target, which is achieved by computing the unit vector pointing from the drone's current position to the target position. This forms the base of the direction, which will be referred to here on out as the motion vector. Furthermore, to avoid collisions with nearby obstacles, when an obstacle is detected within a predefined radius, the component of the motion

vector that points directly to that obstacle is removed. This results in a tangential motion that allows the drone to navigate around the obstacle. In the case that the obstacle lies directly in front of the path of motion, the drones movement is adjusted slightly to go sideways as well as a little up in order to go over the obstacle. This logic prevents the drones oscillating around the obstacles. Since LiDAR is integrated in the drone's abilities, the expert also makes use of this sensing. Horizontal LiDAR rays generate a repulsive bias away from cluttered regions, while vertical layers influence upward or downward motion.

Additionally, when a drone is near the wall, a small repulsion is added to the motion vector to steer it back toward the interior of the environment. This force increases linearly with the distance of the drone to the wall and eventually disappears after a safe predefined distance. Lastly, to avoid collisions with other drones, the expert makes use of the recorded minimum and maximum separation distances to add soft repulsions away from neighbouring drones. Here it is preferred to steer horizontally away from them rather than vertically.

All behavioural components are combined through a weighted summation:

$$\mathbf{d}_{\text{final}} = \text{normalize}(\mathbf{d}_{\text{target}} + \mathbf{d}_{\text{obs}} + \mathbf{d}_{\text{lidar}} + \mathbf{d}_{\text{wall}} + \mathbf{d}_{\text{team}}) \tag{4.17}$$

In addition to direction control, the expert dynamically adjusts the drones speed based on the level of risk in the environment. The speed is reduced under the following conditions:

- when approaching obstacles,
- when nearing the target,
- when in close proximity to other drones.

This slowdown is implemented using continuous scaling rather than discrete thresholds, resulting in smooth deceleration and a more stable controlled behaviour.

At each training timestep, the final action executed in the environment is a combination of:

- the action predicted by the SAC policy, and
- the action proposed by the expert policy.

Formally, the blended action is given by:

$$a_t = \alpha \cdot a_t^{\text{expert}} + (1 - \alpha) \cdot a_t^{\text{RL}}, \tag{4.18}$$

where $\alpha \in [0, 1]$ is the *blend ratio*. A higher value of α means the expert has a heavier influence on what the action of the agent, while lower values allow the learned policy to take control.

Crucially, the expert is only used during *action selection*. The learning process itself remains unchanged:

- Transitions stored in the replay buffer do reflect the blended actions.
- Gradient updates follow the standard SAC objective.

Over time, the influence of the expert is reduced via a scheduling mechanism. It starts out with 80% influence (blend ratio) and proceeds until it gives full control to the RL policy. The expert is also only influencing actions for the first 20% of all training time steps.

4.2.8 Training Callbacks

Training is managed through various callbacks that extend the default functionalities of SB3. The role of the callbacks is to automate check points, perform evaluation and log different training statistics. It makes it possible to monitor the progress and react to intermediate results rather than relying on just a final outcome.

The checkpoint callback is responsible for periodically saving the current trained model, and is done after a certain pre-defined number of training steps. This includes not only the learned policy parameters, but also the replay buffer and the `Vecnormalize` statistics. As previously mentioned, SAC is an off-policy algorithm and samples from all past experiences, so this makes saving the replay buffer an essential task. By storing these together with the model, training can be resumed from specific points without losing the prior distribution of past experiences.

The evaluation callback evaluates the current policy on a separately created 3D environment. This environment is purely for evaluation and is kept distinct from the training environment so that performance can be measured consistently at regular intervals and without the interference of intermediate training states. This callback also saves the best-performing model observed up until that point. Furthemore, this callback goes hand in hand with a second one, namely the `StopTrainingOnNoModelImprovement`, which stops training if repeated evaluations fail to improve after a pre-defined number of *patience* evaluations. This prevents unnecessarily further training.

A custom callback, `StepStatsCallback`, was implemented to collect detailed rollout- and episode-level statistics specific to the multi-drone setting.

- number of drones that reached their targets,
- wall, obstacle, and inter-drone collision rates,
- mean distance to target,
- commanded and realised drone speed,
- swarm-spacing indicators such as drones being too close or too far apart,
- episode outcome categories, including success, collision type, and time limit.

These statistics are logged both to the SB3 logger and to the console, making them available for later analysis in Weights & Biases. This custom callback also implements a curriculum-specific early stopping callback. A stage is considered sufficiently learned if it reaches a certain success rate and maintains a low collision avoidance rate for a certain number of consecutive rollouts. Once these conditions are satisfied, the callback signals that the current stage can end and that the next curriculum stage may begin.

In addition to the main callback stack, a separate `BlendScheduleCallback` is used when the model is trained using the expert blending. This callback ensures that the blend ratio decreases gradually over the specified percentage of time steps.

4.2.9 Episode Termination

When to terminate an episode during training is important as it can heavily influence training speed and outcome. For this project, there were two main options, terminating the episodes when all drones finished or when the first drone finishes. If the former were chosen, as soon as a drone finishes, sampling from that drone would stop, but other drones would keep going. This would make training faster as fewer environments would have to be generated, and more information would be gathered per environment. Furthermore, since decentralised learning occurs, such an approach makes sense, as the rewards are per drone, and not per swarm, nonetheless, the other option was selected. The first reason is that when dealing with drones in real-life scenarios, a safety-critical mindset must be used, and the reliability of the drones can be more important. Additionally, by ending episodes when the first drone finishes, this keeps the custom replay buffer always full of useful information and to not take samples from inactive drones.

4.3 Justification of Design Choices

4.3.1 Libraries and Frameworks

The physics simulation layer is built on **PyBullet**, accessed through the `gym-pybullet-drones` wrapper which exposes the `VelocityAviary` environment. **PyBullet** provides a physics engine with support for collision detection, contact point queries, and real-time 3D simulation. All of these are essential for modelling the flight dynamics of drones and detecting interactions between drones, obstacles, and world boundaries. While `gym-pybullet-drones` does provide built-in RL-ready environments, these require precise low-level motor commands in the form of RPM values per rotor, which makes them difficult to train an RL algorithm on directly. They also come with predefined tasks that offer no support for additional sensors such as LiDAR, no obstacle avoidance, and no goal-reaching behaviour, which are all core requirements of this project. Extending them for future work would likewise be heavily constrained. `VelocityAviary` was chosen instead because it accepts normalised velocity commands, making it straightforward to control with an RL policy, while still providing sufficiently realistic drone dynamics.

Compared to alternatives such as **Gazebo**, **PyBullet** was preferred because of its straightforward Python API, and its tight integration with `gym-pybullet-drones`, which already provided validated drone dynamics models. Another reason is that `Gazebo` comes with considerably more setup overhead and is harder from programmatic standpoint.

The environment extends **PettingZoo's** `ParallelEnv` because all drones act simultaneously each step, which is exactly what `ParallelEnv` is designed for. It also supports the `possible_agents` and `agents` split, which is used to despawn

drones mid-episode. Where a drone that collides or reached its target is removed from `agents` while the fixed observation and action space shapes stay unchanged, keeping everything compatible with SB3's vectorised wrappers.

The reinforcement learning component of this project is built primarily using `Stable-Baselines3 (SB3)`, which is a widely-used `PyTorch` library that provides reliable and ready-to-use modern RL algorithms. Specifically, the implementation of their SAC algorithm was used for training purposes due to its stability, sampling efficiency and suitability for this project and its continuous action space. SB3 was the chosen library implemented in the previous 2D version of which this project is an extension. For this reason, it was decided to continue using it, but also on account of its simplicity and strong documentation which made it easy to integrate with the custom 3D environment. SB3 also supports the vectorised execution which helps make the training process more efficient. The downside of SB3 is that it unfortunately does not support MARL, and thus had to be wrapped using `SuperSuit`, which allows `PettingZoo` environments to be compatible with SB3 by converting the multi-agent case into a format suitable with single-agent RL algorithms.

Additionally, to monitor the training progress and see the progression of statistics over each curriculum stage's time steps, `Weights and Biases (wandb)` was integrated into a training script that functions purely with the SAC model, i.e. no expert blending policy. This framework allows for real-time logging and plotting of the key performance metrics, as outlined in Section 6.1. It also allows training runs to be stored and analysed externally, making it easier to perform hyperparameter tuning and compare different configurations. This is essential in such reinforcement learning settings with a complex 3D environment where plenty of stochasticity is present in both the environment and the learning process.

4.3.2 Architectural Choice

The project follows a **modular environment-centred architecture**, in which a single top-level environment class, `DroneSwarm3D`, coordinates the full simulation loop while specialized components handle world entities, collision querying, configuration, and reward computation. This architecture was chosen because the project combines several concerns that should remain separable: realistic drone simulation, procedural generation of 3D scenes, multi-agent episode management, sensor construction, and RL-specific training logic. Keeping these responsibilities inside one monolithic class would make the system much harder to extend and reason about, especially when adding new sensors, obstacle types, or reward terms.

At the core of the system, `DroneSwarm3D` acts as the orchestration layer. It is responsible for storing the environment configuration, creating the simulator instance, managing agent state, resetting episodes, advancing the simulation, and assembling the observation dictionary returned to the learning algorithm. This makes it the natural integration point between the physical simulation and the reinforcement learning interface. The environment also exposes a fixed action and observation interface, while internally supporting curriculum-style variation in the number of active drones through the separation between `possible_agents` and the currently active drones. This design keeps the external API stable for training frameworks

while still allowing drones to despawn after collisions or goal completion.

A second architectural layer handles **world entities**. Static obstacles, dynamic obstacles, and goals are represented as separate object abstractions rather than being hard-coded into the environment logic. `EnvObject` provides a generic representation for shaped 3D bodies together with shared functionality such as shape specification, spawning, overlap checks, and validity checks. `Dynamic` extends this with time-dependent motion along a chosen axis, including both online updates during simulation and position queries at arbitrary times. `Target` is modelled separately because it behaves as a goal marker rather than a collidable obstacle. This split was chosen to make world generation extensible: new entity types can be added by extending the object layer without restructuring the full environment loop.

Collision handling is also separated architecturally. Normal collision checks during an episode are performed inside the main simulation, but the system additionally introduces a dedicated `CollisionQueryWorld` that mirrors obstacle geometry in a separate PyBullet client. This allows hypothetical or future-time collision checks to be carried out without interfering with the active training environment. The advantage of this design is that predictive validation of moving obstacles remains isolated from the main simulator state, reducing coupling and making it possible to reject unsafe candidates before they affect an episode.

Finally, configuration and reward logic are externalized. Numerical constants such as spawn margins, LiDAR settings, collision thresholds, and reward coefficients are stored in a dedicated defaults module, while reward shaping is computed in a separate rewards module. This keeps the main environment focused on state transitions rather than parameter management or reward formula. The result is an architecture that is easier to tune, debug, and extend: changing the curriculum, observation richness, or reward structure can be done with minimal impact on the rest of the codebase.

Overall, this architecture was chosen because it balances clarity, extensibility, and compatibility with reinforcement learning frameworks. The central environment class provides a single controlled interface to the learning algorithm, while the supporting modules isolate object modelling, predictive collision checking, reward shaping, and shared configuration. This makes the system suitable not only for the current target-reaching task, but also for future extensions such as richer obstacle behaviours, additional sensors, or more multi-swarm coordination.

Chapter 5

Testing Environment

5.1 Testing Strategy

The testing of the system was carried out in several steps. The first goal was to check whether the custom environment itself behaved correctly. The second goal was to check whether the environment still behaved correctly when connected to the reinforcement learning pipeline. The third goal was to check whether the final trained policy could solve meaningful navigation tasks in fixed environments that were designed in advance. Since the project combines simulation, sensing and learning in one system, testing could not be limited to isolated helper functions. Instead, a combination of smaller functional checks with full simulation runs and final evaluation was chosen.

A first part of the strategy was component-level functional testing. This was used for behaviours that needed to remain correct regardless of the exact scene. Examples of this are the fixed-capacity agent interface, inactive drone handling, reset stability and stage transitions during the curriculum-based training. These parts were tested separately because they define the correctness of the environment as an RL interface. If they are wrong, then observations, rewards and stored transitions can already become invalid before any meaningful training takes place.

A second part of the strategy was integration testing of the full environment loop. In these tests, the environment was reset and stepped repeatedly while the drone positions, rewards, collisions, terminations and target-reaching status were monitored. This was important because many parts of the system only make sense when they are running together. Obstacle updates, action conversion, collision checks, observation construction, despawning logic and reward computation all interact inside the same simulation step, so they had to be tested in the running environment and not only as separate units.

A third part of the strategy was controlled scenario testing. Alongside the usual random generation of worlds, the environment was also tested in predefined scene layouts. This made it possible to observe the same navigation situation more than once and compare behaviour across repeated runs. Such a strategy was useful for obstacle-heavy simulations, because it removed part of the randomness from the test and made failures easier to replicate and inspect.

Another important part of the strategy was sensor and reward inspection. Li-

DAR and the reward function are both central to the learning behaviour of the drones, so they were not treated as simple black boxes. Instead, they were tested while the simulation was running by observing how LiDAR values changed in the presence of known obstacles and how reward values changed as drones acted in the environment.

Furthermore, testing included training-pipeline debugging. Since the environment was built specifically for SAC training with curriculum learning and expert blending, it was necessary to inspect what the learning code actually receives from the environment. For this reason, the testing strategy also covered normalized and raw observations, active and inactive drone slots, replay buffer insertions, active masks, and the relation between expert actions, RL actions and the final blended action. This helped verify that the simulator was not only correct in isolation, but also correct in the exact training setup.

Finally, the strategy also included the post-training end-to-end evaluation. After training the final policy was tested in a set of prebuilt environments, such as obstacle layouts with narrow openings or more structured scenes like maze or forest like worlds. The purpose here was different from earlier environment tests. Here, the goal was no longer to check one internal mechanism at a time, but to see whether the full trained system could guide a swarm through a fixed scene and reach the target without collisions. This gave a more practical view of the final result and complemented the random training environments with repeatable evaluation oriented towards the task completion.

5.2 Test Cases and Solutions

5.2.1 Environment Generating

Initial tests of the environments were simple, and each component was tested manually to ensure functionality and later stress tested.

5.2.2 Drones & Targets

For drones and targets, the first test cases were spawning a swarm of one drone and one target. This test was used to ensure drones always spawned opposite to their target, and that both were far enough from the walls. This was latter ramped up by increasing the number of drones per swarm and swarms. Additionally, this was stress-tested by increasing the number of drones per swarm until drones did not spawn or spawned overlapping other drones. This helped determine how many drones could realistically fit in it and also make sure the environment did not crash in such circumstances.

The tests revealed issues with the spawning behaviour, as sometimes swarms would spawn too close to their respective targets. During debugging, it was realized that the spawn positions were correctly generated, however, the issue persisted. The root of the problem was the order of the reset function, causing previous spawn positions to be used instead of the current one. However, after changing the environment

for the fixed number of agents, for Expert Blending, the issue re-emerged. Originally,`VelocityAviary` was responsible for the repositioning of drones during resets, but with the addition of drones with dummy positions, it failed. Thus, as a solution, since the drones are never truly despawned, just moved far away, active drones are manually moved to their new spawn positions. Additionally, stress-testing helped determine the adequate distance between drones in a swarm, within the given swarm radius.

5.2.3 Obstacles

Similar testing was done for obstacles. First, manual tests with a few obstacles were done, focusing on the spawn position, and scenarios where the object was placed near the target. This helped ensure, independent of the object orientation, that it did not go into the target or overlap the world walls. This was extended with dynamic obstacles to see if its path was correct and did not collide with anything during its motion. Similarly, stress testing was also done for obstacle spawning. These tests allowed for analysing the obstacle-spawning behaviour in high object-density environments.

These tests revealed issues in how the object overlap was being originally checked. The overlap was done simply by ensuring that the object spawn position was within a pre-determined distance from other object spawn positions. This approach did not take into consideration how object orientation could affect overlapping and was not best suited for obstacles of different shapes. This created scenarios were, even though the distance between objects were met, they would still overlap, as shown in figure 5.1. Thus, a different approach was used, where object spawning would be simulated behind the scenes, allowing for more precise overlapping checks.

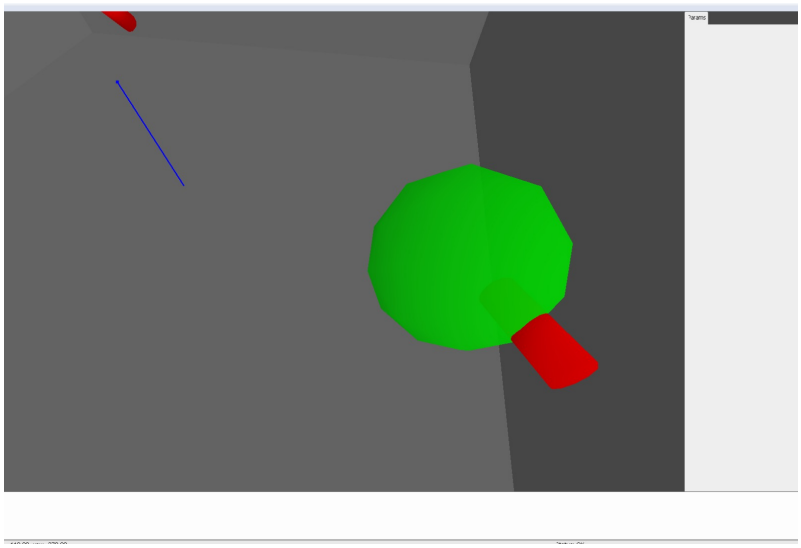

FIGURE 5.1: Obstacle Overlap Issue

With the original approach, when static obstacles were spawned first, it was harder to guarantee that a dynamic obstacle would not later move through a static one. The solution was to spawn dynamic obstacles first, and only then spawn static

obstacles. Once a dynamic object is created, its future path can be approximated by a swept area and every static obstacle candidate can then be rejected if it lies inside that area. Spawning static obstacles was therefore simpler and more reliable than trying to validate each new dynamic obstacle against a world that already contained many static obstacles.

Chapter 6

Performance Evaluation

6.1 Performance Metrics

The following performance metrics were chosen to evaluate the success of the trained model. The following metrics reflect the performance of the final trained RL model by evaluating the project requirements.

6.1.1 Success Rate

This metric tests the requirement of the drones successfully reaching the target calculated by the following formula.

$$\text{Success Rate} = \frac{\text{Number of successful episodes}}{\text{Total number of episodes}} \tag{6.1}$$

6.1.2 Collision Episode Rate

This metric tests the requirement of the drones colliding with each other, walls and obstacles - both moving and static calculated by the following formula. (failure case)

$$\text{Collision Episode Rate} = \frac{\text{Number of episodes with collision}}{\text{Total number of episodes}} \tag{6.2}$$

6.1.3 Collision-Free Rate

This metric tests the requirement of the drones not colliding with each other, walls and obstacles - both moving and static calculated by the following formula. (success case)

$$\text{Collision-Free Rate} = \frac{\text{Number of collision-free episodes}}{\text{Total number of episodes}} \tag{6.3}$$

6.1.4 Wall-Hit Episode Rate

This metric tests the requirement of the drones not colliding with walls only, calculated by the following formula. This metric was included dictated by a necessity introduced by initial training where drones would often collide with walls.

$$\text{Wall-Hit Episode Rate} = \frac{\text{Number of episodes with wall collision}}{\text{Total number of episodes}} \tag{6.4}$$

6.1.5 Average Episode Length

This metric tests the requirement of the average time taken by the drones to reach the target of collide or potentially time out - calculated by the following formula. This metric reflects the speed/efficiency of the drones as well as reflecting the time out rate of the drones.

$$\text{Average Episode Length} = \frac{1}{N}\sum_{i=1}^{N} L_i \tag{6.5}$$

6.2 Performance of Trained RL Model

To evaluate the performance of the model, a model evaluation file was created, namely `evaluate_final_model.py`. A curriculum was designed just for the evaluation process, shown in 6.1, and by using the same curriculum to evaluate performance, it reduces bias when comparing different models. This evaluation curriculum is simply the current curriculum found in `curriculum.py` and can be modified according to the user's goal. Furthermore, the model is evaluated 50 times per stage to create reliable performance statistics.

world_scale	num_drones	num_static_obstacles	num_dynamic_obstacles	stage_training_timesteps	episode_horizon	lidar_enabled
2	1	0	0	0	1000	True
2	1	1	0	0	3000	True
2	2	0	0	0	3000	True
3	1	0	0	0	3000	True
3	1	2	0	0	4000	True
3	2	2	0	0	4000	True
4	5	3	0	0	5000	True
4	5	4	1	0	5000	True
5	5	5	1	0	5000	True
4	4	4	6	0	5000	True

TABLE 6.1: Test Curriculum Configuration

Due to the high number of episodes, the evaluation process can take time. Thus, for simple and faster model testing, another evaluation file was made, `test_model.py`. Additionally, it also allows for viewing the behaviour of the drones, to better understand what it is doing, instead of only relying on statistics.

Before the implementation of the expert policy, some SAC models were trained, and the evaluation of one of the early models is shown in 6.2.

Although manual tests were done to ensure the expert policy had good performance, a model that was completely controlled by the expert was evaluated, and

Stage	World Scale	Drones	Static	Dynamic	Max Steps	Coll-Free	Success	Coll Ep.	Wall Ep.	Avg Len
0	2	1	0	0	1000	10.00	10.00	90.00	90.00	234.62
1	2	1	1	0	3000	26.00	26.00	74.00	68.00	287.94
2	2	2	0	0	3000	10.00	10.00	90.00	90.00	305.72
3	3	1	0	0	3000	2.00	2.00	98.00	98.00	320.68
4	3	1	2	0	3000	6.00	6.00	94.00	86.00	471.86
5	3	2	2	0	4000	14.00	14.00	86.00	84.00	537.44
6	4	5	3	0	5000	2.00	2.00	98.00	86.00	848.74
7	4	5	4	1	5000	12.00	12.00	88.00	84.00	816.90
8	5	5	5	1	5000	0.00	0.00	100.00	92.00	1000.96
9	4	5	2	6	5000	4.00	4.00	96.00	78.00	786.62

TABLE 6.2: SAC Model Performance

the results are shown in 6.3. The evaluation confirms that the expert algorithm is sufficiently good as it has over 90% reach rate in most stages. However, there are two exceptions, the first and last stages. In the first stage, the success rate is 74%, which is lower than in other, more complex stages. It is important to note that even though the success rate is not as high, it still has 100% collision free episodes, meaning that the other 26% of stages ended due to a timeout. In the last stage, the model gets its lowest success rate, and the main difference from this stage compared to others, is the increase of dynamic obstacles.

From these result, it is possible to conclude that the expert policy can avoid obstacles efficiently in drone environments. This shows that its use for expert blending is feasible and has promising results.Nonetheless, the expert policy is not perfect, as it struggles with time constraints and dynamic obstacles, both of which would most likely be resolved by the RL algorithm.

Stage	World Scale	Drones	Static	Dynamic	Max Steps	Coll-Free	Success	Coll Ep.	Wall Ep.	Avg Len
0	2	1	0	0	1000	100.00	74.00	0.00	0.00	837.32
1	2	1	1	0	3000	90.00	90.00	10.00	10.00	1633.80
2	2	2	0	0	3000	100.00	100.00	0.00	0.00	613.78
3	3	1	0	0	3000	100.00	100.00	0.00	0.00	1698.88
4	3	1	2	0	3000	98.00	92.00	2.00	0.00	2857.26
5	3	2	2	0	4000	98.00	98.00	2.00	0.00	1619.82
6	4	5	3	0	5000	100.00	100.00	0.00	0.00	2802.96
7	4	5	4	1	5000	96.00	96.00	4.00	0.00	3161.70
8	5	5	5	1	5000	92.00	92.00	8.00	8.00	3621.48
9	4	5	2	6	5000	68.00	68.00	32.00	0.00	3419.12

TABLE 6.3: Expert Policy Performance

The idea of training the model with expert blending was introduced after training the initial SAC model, allowing for faster training and a simpler reward function. The initial plan was to focus on training a successful RL model with the use of the expert policy. The first models trained with it, showed promising results, and so other parts of the project were prioritised. However, with 1 week left, through testing, it was discovered that the model relied on the expert policy more than expected, and it was not a full RL model. So, due to time constraints, not a lot of training was done with the use of expert blending, and so no pure RL model was made. However, a file for the training with expert blending was created, called `sac_training_with_expert.py`, which does work.

Chapter 7

User Manual

7.1 Installation and Setup

The installation and set up procedure required for the project is explained in the README.md file found in the projects GitLab repository [5]. The README file also contains more information about what configurations can be changed to train different models and where they can be found.

7.2 Interface

Once all libraries are installed, you can launch the simulation by running:

```
python -m interface.interface_viewer
```

Executing this command opens the landing page (Figure 7.1), where you can configure the simulation environment. The interface is divided into two sections, the left has the quick settings, the essential configuration features required for the application. The right has some additional features. After adjusting the settings, click `Launch Simulation` to proceed.

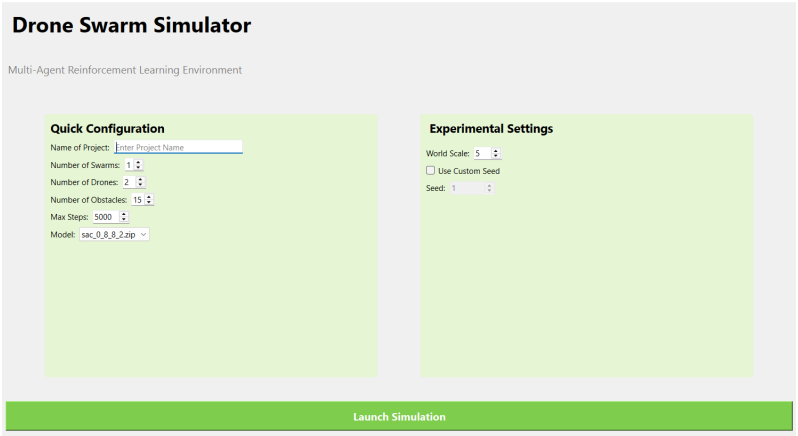

FIGURE 7.1: Quick settings setup

7.2.1 Environment Configuration

Once the settings are confirmed, the environment setup viewer opens (Figure 7.2). This interface provides several configurable components: the type, so if its static or dynamic, shape, coordinates (x,y,z), size and some properties if its dynamic such as speed and direction. The number of objects can be adjusted in real time. Multiple swarms can be generated. For each swarm you can specify the colour of the swarm, and a name. For each drone in the swarm you can set the relative position of each drone with respect to the swarm centre. Targets can be created and placed anywhere in the environment, for each swarm there is one target.

Once configured, the environment can be stored for future use and modification or loaded from a previously saved configuration.

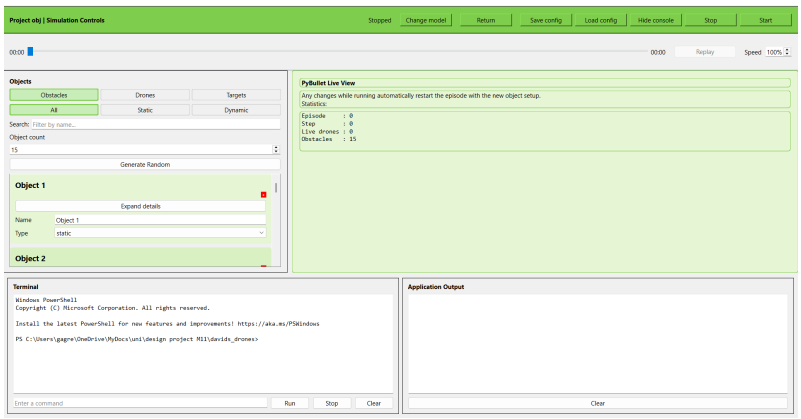

FIGURE 7.2: Advanced environment setup

7.2.2 Running the Simulation

When you are ready to run the simulation, click `Start`. A new visualization window will open, displaying the drones, objects and targets. To move around the 3D environment, zoom in and out and to change directions, hold `Ctrl` while using the trackpad or mouse.

While the simulation is running, no modifications can be made to the environment. Changing any attribute (e.g., an obstacle property) will reset the simulation. To view terminal outputs alongside the simulation, click `Open Console` before starting the simulation. This allows you to monitor all terminal outputs while using another terminal for other tasks.

7.2.3 Simulation Control and Replay

Once the simulation completes, you can either, wait for a new simulation to start automatically, or click `Stop` to end the current simulation. After stopping, the `Play Replay` option becomes enabled. Clicking it replays the entire simulation, allowing you to pause and resume playback, rewind to previous moments and increase or decrease playback speed for improved analysis. All this can be done with the buttons at the top of the screen and the scroll bar.

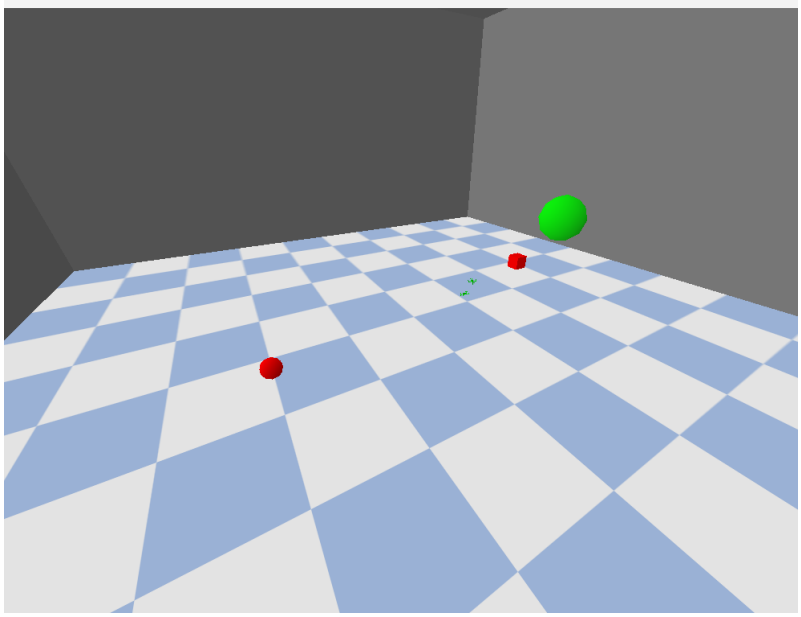

FIGURE 7.3: Running simulation view

7.3 Training the Model

All that has been explained is perfect to check how the drones could arrive to the target, but to ensure this, a trained model needs to be used. To train the model using our algorithm, you must use the commands below to start the training.

- `python -m train.sac_training`:

  This script trains a model using the standard SAC algorithm without any expert guidance. The agent learns entirely through reinforcement learning by interacting with the environment, collecting rewards, and updating its policy. The outcome is a model developed that explored creative or adaptable solutions. Performance depends heavily on the reward function and training duration.

- `python -m train.sac_training_with_expert`: This script also uses SAC but integrates an expert policy to guide the agent's actions during training. The expert policy significantly improves learning efficiency and final accuracy, but the resulting model exhibits little true reinforcement.

- `python -m train.evaluation.testmodel`: This script loads a previously trained model (either pure SAC or expert-augmented) and tests it across multiple custom environments. It systematically measures two key metrics, collision rate and target reach rate.

Chapter 8

Individual Contributions

Component	**Contributor(s)**	**Description**
3D Environment (`DroneSwarm3D`)	Stella, Pedro, David	Custom environment design and implementation
	Stella, Pedro	(De-)spawning logic
	Stella, Pedro	Environment testing
	Stella, Carim, Pedro	Collision detection logic
Obstacle System	Pedro, Carim	Static and dynamic obstacle generation and validation
LiDAR Integration	Stella, Pedro, David	LiDAR integration and testing
Reinforcement Learning (SAC)	Stella	Algorithm research
	David, Pedro, Stella	Training, Testing & tuning parameters
Reward Function	Stella, Pedro, David	Design and implementation
Expert Policy	Stella	Design, implementation and testing
Custom Replay Buffer	Stella	Design, implementation and testing
Graphical User Interface	Raul, Ivan	GUI development and simulation interface
Diagrams	Raul	Data Flow, Class and Component Diagrams
Evaluation and Testing	All	Component-level testing by respective contributors
Documentation and Report	All	

Table 8.1: Summary of Individual Contributions

8.1 Green Card

Given the individual contributions, we considered that the contribution of one of our members was exceptional compared to everyone else. We would like to give the Green Card to Stella-Maria Horvath to attempt to show our appreciation for her work during the project.

Chapter 9

Discussion & Conclusion

9.1 Discussion

The initial scope of the project was developing and implementing a “Production-Ready" 3D collision avoidance module that integrates high-fidelity sensor data with the existing PPO logic. Although most of this requirement has been achieved, with the consent of the client and mentor the PPO logic has been replaced with the better performing SAC logic [1].

As discussed in this paper, the change has provided higher performance, albeit with a great increase in complexity as well. Due to the difficulties encountered during the development of the SAC model an expert policy was created, with the purpose of speeding up the training process while also improving the success rate of the model. The expert policy provided great results when combined with the SAC model, however, due to the inexperience of the team in regards to reinforcement learning, the SAC models over-reliance on the expert policy has gone unnoticed until the latter stage of the project. Despite the fairly high success rate a model that is over-reliant on the expert policy is not desired. This issue has led the team to modify the scope of the project such that it also acts as a framework that allows users to train their own reinforcement learning model.

9.1.1 Team & Software Organisation

The team held weekly meetings with the mentor and client to present the progress, get feedback, discuss potential improvements and get information on the best course of action for future work. Exceptions to the weekly meetings happened during the last two weeks when communication was done mainly via email.

Additional meetings between team members were scheduled at least once a week to ensure that the current situation was clear, to distribute tasks and resolve any type of issues. The meetings began with a brainstorming process to find the most urgent tasks, then a discussion on which groups of at least two people should be assigned to them. The vast majority of the tasks have had two people assigned so that potential problems could be solved without requiring a meeting while simultaneously allowing each committed code to be peer reviewed by at least one more person. Tasks that were larger either in person-hours required or complexity had at most three people

assigned to prevent miscommunication between the members, however, the other members were expected to test and check the correctness of the work. In general, each task should not exceed a duration of three days; however, as it is always hard to know the exact difficulty before implementation started, this was not always the case. For tasks that would take longer than expected, the team would discuss the best course of action and decide if more people should be assigned or in very rare cases scrapping the feature and dropping the task.

Overall, the development framework used is a hybrid between the waterfall and agile frameworks. Although the predictability and stability offered by a waterfall development approach was greatly desired, the previously discussed difficulties have made it hard to exclusively follow the principles of one framework. As combining the two opposite framework can be harmful if done incorrectly, a separation was done by modules so that the chances that there are any significant detriments to the development are lowered.

In general, the project has followed the waterfall model phases, with certain modules acting as exceptions. The first two weeks were spent on requirements gathering and analysis. The next part, which took approximately one and a half weeks were spent on design and research. The implementation has then started and lasted until the start of week nine. The testing was done thorough the entire duration of the implementation, however, extensive stress testing and system testing have started in week nine and lasted until week ten. The last week is the deployment phase, and includes a live demo for the client.

This organisation method has helped the team in preventing unexpected bottlenecks throughout the development while also allowing them to work independently on their tasks.

9.2 Conclusion

The project originally started with developing a PPO based collision avoidance model for drone swarms finding their way to a target and visualizing it with an interface. The goal was closely met, with the additional benefit that, due to the system's modularization, it can be expanded and improved further. Making use of research papers offered by the mentor, we produced an environment designer and editor for many types of objects, an option to train new models using our reward functions and our model training with SAC and expert policy, and finally a way to visualize and test model performance across many scenarios.

The original requirements were substantial. On the functional side, the system had to provide an interactive interface where users could configure drone swarms, set target locations, place static and dynamic obstacles, and specify spawn points. All of this needed to run in a custom 3D simulation with real-time collision avoidance using LiDAR sensing. Drones had to avoid both obstacles and each other.

Among the should have features was supporting global parameter changes across an entire swarm. The could have features included swarm leader coordination, maintaining distances between drones, and adjusting physics parameters without retraining models. On the non functional side, expectations included real-time performance, scalability, reliability, sensor accuracy, modular code structure, and a

usable interface. Several constraints applied: simulation only, limitations from the GymPybullet Drones environment, and a fixed project timeline. The key assumptions were that LiDAR readings would be accurate, environmental factors like wind would be ignored, and all drones would behave identically.

Now that we have spent the last weeks working on the project, we can say that we have achieved most of the initial requirements given the constraints. We implemented all the interface features for 3D visualization of a simulation where you can set objects, targets, and drone configurations, except for drone speed. For the drones themselves, using SAC with expert policy, they reached the goal with very few collisions. However, with the pure RL model (SAC without expert), collisions remain frequent. One feature we did not manage to implement was the swarm leader.

On the non functional side, we delivered a system that is modular, reliable, and has a usable interface. Scalability depends on the computational power available. The 3D render we used is resource intensive and handles many objects and drones poorly, so scalability is limited in practice. We knew this limitation from the start, so given our constraints, the system performs as expected. With a more efficient physics engine, it would be easily expandable.

The main functional requirements were mostly achieved, as the system provides a configurable GUI, supports user defined drones, swarms, spawn locations, targets, and static and dynamic obstacles and works in a customizable 3D environment. Lidar was also successfully implemented. The should have requirements of supporting global swarm-level parameter changes were done to some extent via shared environment and configuration parameters. Regarding the could have requirements explicit swarm-leader coordination was not implemented and physics changes without retraining the model may be challenging.

The non-functional requirements: modularity, usability, and scalability were achieved to a great extent as the code is easy to understand and separated into logical components. Performance and reliability are perhaps the weaker non-functional requirements compared to the others, simply due to the fact that the complexity of the training performance of the drones was challenging, and thus the reliability of the drones making it to the target.

9.2.1 Future Work

There are still many improvements that can be done so that the project can evolve from the initial goal of transitioning a simulation environment from 2D to 3D into an application capable of simulating real-life scenarios.

On the reinforcement learning side, there is currently no pure RL model that can reliably navigate the environment available, as the high-success-rate model remains over-reliant on the expert policy. Achieving a similarly performing pure RL model would require significant processing power and time, given the slow training times and the need for reward function refinement. As the project's goal is to provide a framework for training models, a more technically knowledgeable user with a high-performance setup could potentially achieve this. Another possible solution would be to outsource a server that is capable of speeding up the training process. The

fixed swarm capacity of 10 drones was chosen as a balance between swarm size and training time; future work could investigate whether the shared policy scales effectively to larger swarms, and what architectural modifications would be necessary to preserve collision avoidance performance as the number of agents grows.

A natural and significant direction for future work would also be the deployment and validation of trained policies on physical UAV hardware. While the simulation provides a controlled and reproducible training environment, it remains to be seen whether behaviours learned in PyBullet transfer effectively to real drones, and whether the sim-to-real gap introduces new problems obscured by the simulation. The realism of the simulation itself could further be improved by introducing real-life physical forces such as wind and turbulence, which were explicitly excluded from this project. Their inclusion would make training considerably more challenging, but the resulting policies could be more meaningful and applicable to real-world deployment scenarios. On the coordination side, a leader-follower architecture could be implemented, where an assigned drone coordinates the path to the target for the rest of the swarm. Such a model could improve cohesion and navigation efficiency, particularly in complex or constrained environments, and could include explicit inter-drone communication modelling.

Regarding the simulation environment itself, obstacles are currently limited to three basic geometric shapes and a binary dynamic/static property. This could be extended to include more complex shapes and real-life objects such as trees and buildings, with placement rules governing how they are arranged. This would add realism to the environment and could eventually lead to procedurally generated, real-world-like scenes. It should be noted, however, that the simulation's performance constraints must be taken into account, as entity counts exceeding roughly 100 begin to cause noticeable lag.

Finally, user testing and interviews were conducted to evaluate the usability of the GUI. The general feedback was that it was a functional product, though without standout features. The most significant UX improvements identified relate to entity creation and replay capabilities. Currently, positioning entities requires manually entering coordinates, which proved more time-consuming than expected for users to orient themselves spatially. This could be addressed by introducing a visual editor page where users can drag, click, and delete entities directly in the environment. The replay function, while useful for identifying anomalies, does not allow any modifications during playback, limiting its value for exploring alternative outcomes. Enabling interactive replay editing would significantly improve its utility for research and analysis purposes.

9.2.2 Relation of Results to Requirements

Requirement	Evidence from results	Achievement
3D target navigation	Table 6.3 shows high success rates in most stages, demonstrating successful navigation in the custom 3D environment.	Mostly achieved
Collision avoidance	High collision-free rates and low wall-hit rates for the expert policy show effective avoidance of walls, obstacles, and other drones.	Mostly achieved
LiDAR-based sensing	LiDAR was integrated into the observation space and supported strong performance in obstacle-rich stages.	Achieved
Interactive user interface	The implemented GUI allows users to configure drones, swarms, targets, and obstacles, and to run simulations.	Achieved
Modularity and usability and scalability	The system is separated into clear components and provides a usable interface for simulation and evaluation.	Achieved
Reliability and performance	The framework worked across repeated runs and harder stages, but performance decreased as complexity increased.	Partially achieved
Pure RL autonomy	Table 6.2 shows that pure SAC remained weak, so strong performance was mainly achieved with expert guidance.	Not fully achieved

Table 9.1: Relation of results to the main project requirements

Chapter 10

Use of AI

During the preparation of this project, the team used ChatGPT and Claude Sonnet to support code refinement, including generating suggestions and basic tests, and identifying potential bugs. Grammarly was used to improve the clarity and correctness of the written report. After using these tools/services, the team thoroughly reviewed and edited the content as needed, taking full responsibility for the final outcome.

Bibliography

[1] V. Lysenko, "Multi-agent collision avoidance using ppo in decentralized reinforcement learning for drone simulated environment," in *Proceedings of the 43rd Twente Student Conference on IT (TScIT 43)*, Authors version. Not for redistribution, Enschede, The Netherlands: ACM, Jul. 2025.

[2] A. P. Kalidas, C. J. Joshua, A. Q. Md, S. Basheer, S. Mohan, and S. Sakri, "Deep reinforcement learning for vision-based navigation of uavs in avoiding stationary and mobile obstacles," *Drones*, vol. 7, no. 4, p. 245, 2023.

[3] P. Junare, *Rl-uav: Reinforcement learning for autonomous indoor uav navigation*, https://github.com/pranay-junare/RL-UAV, GitHub repository, 2024.

[4] G. Varshini, *Automated uav navigation in simulated environments*, https://github.com/GarrepelliVarshini/Automated-UAV-Navigation-in-Simulated-Environments, GitHub repository, 2024.

[5] R. Botea, P. C. Said, S.-M. Horvath, I. P. Gómez, C. Popa, and D. Szentpeteri, *3d drone swarm simulator*, https://gitlab.utwente.nl/s3148726/3D_Drone_Swarm_Simulator, GitHub repository, 2024.

[6] Keeeor, *Sar-sac: State-aware regulated soft actor-critic for safe uav navigation in complex environments*, https://github.com/Keeeor/SAR-SAC, GitHub repository, 2024.

Chapter 11

Appendix

11.1 Model Configurations

The results displayed in 6.2 were from a model that was trained with the following configurations. The hyperparameters in Table 11.1 were primarily chosen based on observations from other implementations of 3D drone collision avoidance modules implemented with SAC [4] [3]. We set both the training frequency and gradient steps to 1 to ensure that the model updates continuously, which promotes incremental learning.

Hyperparameter	**Value**
Discount factor (γ)	0.99
Batch size	128
Replay buffer size	300000
Learning starts	2000
Training frequency	1
Gradient steps	1
Learning rate	3×10^{-4}
Entropy coefficient	auto

TABLE 11.1: Training hyperparameters

11.2 Class Diagrams

11.2.1 Interface

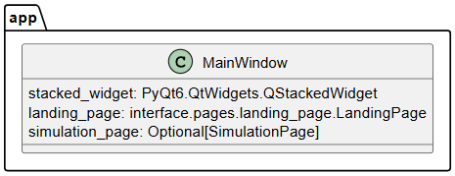

FIGURE 11.1: Interface Entry Point

FIGURE 11.2: Interface Controllers

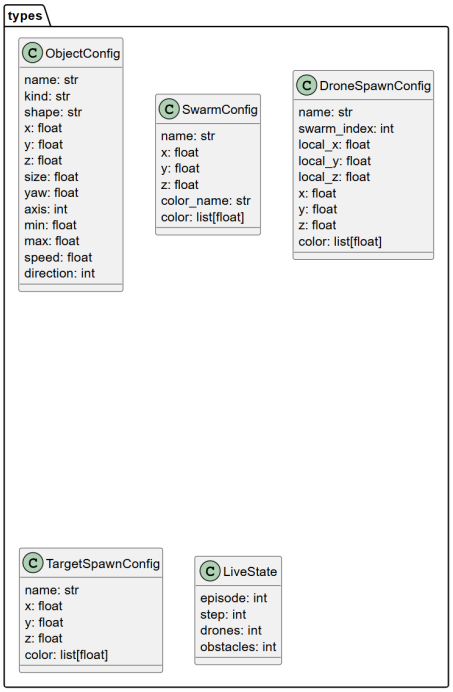

FIGURE 11.3: Interface Data Types

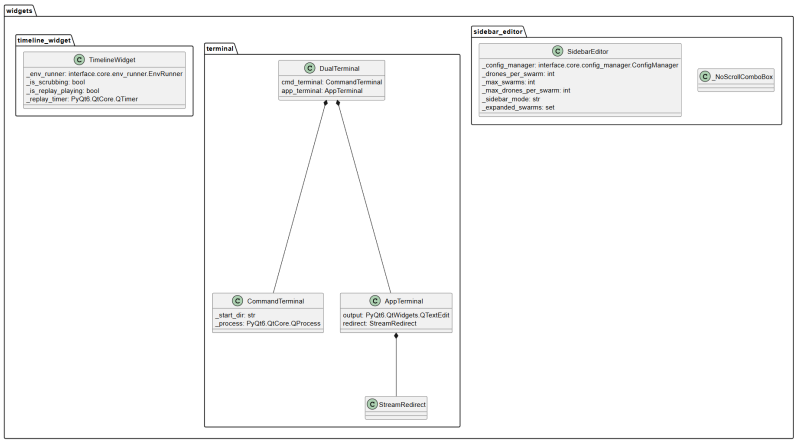

FIGURE 11.4: Interface Widgets

11.2.2 Wrappers

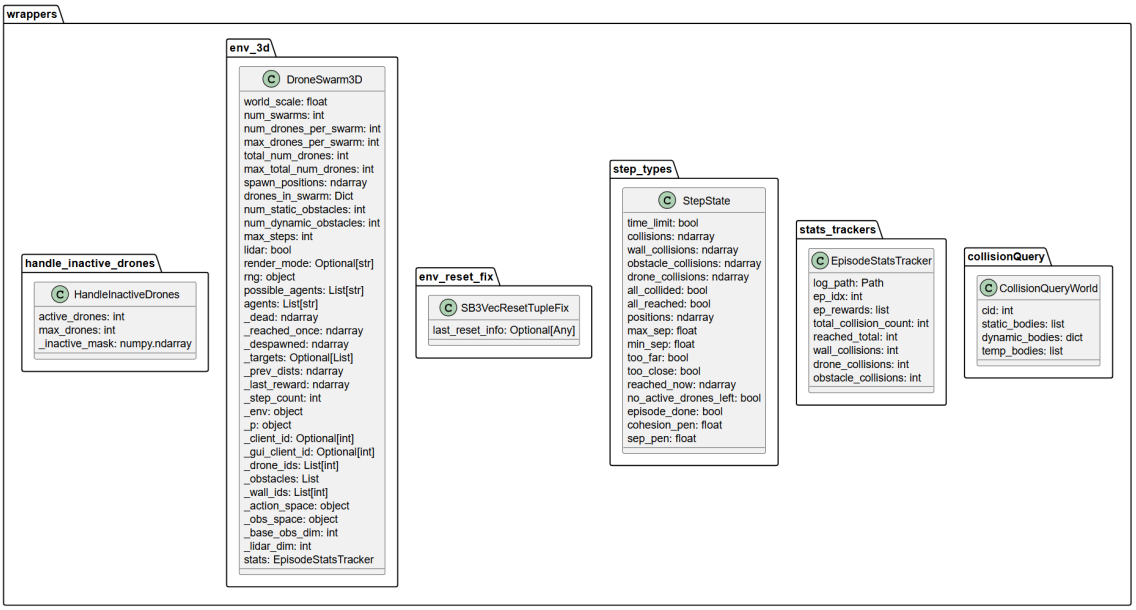

FIGURE 11.5: Wrappers

11.2.3 Training and Evaluation

FIGURE 11.6: Training Callbacks

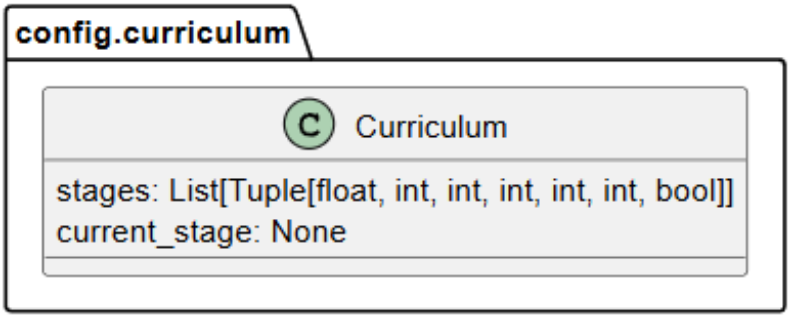

FIGURE 11.7: Training Curriculum

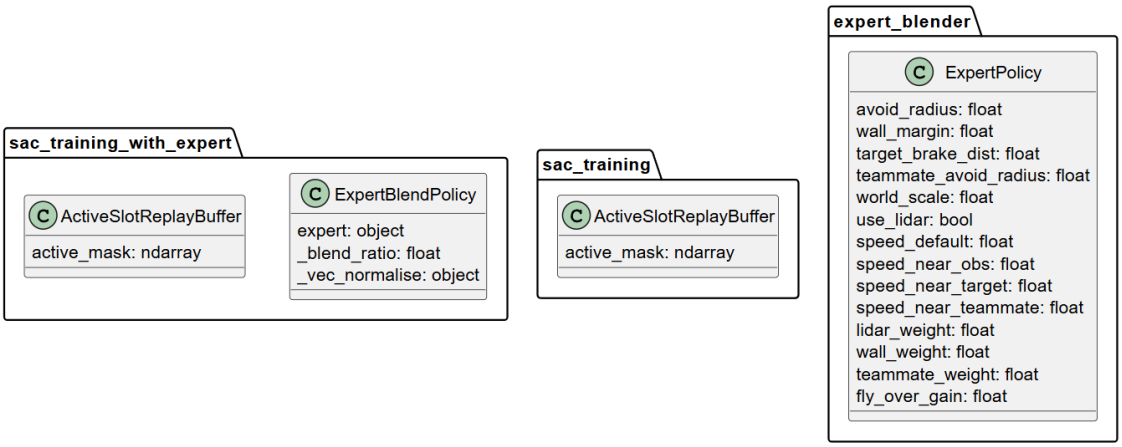

FIGURE 11.8: SAC and Expert

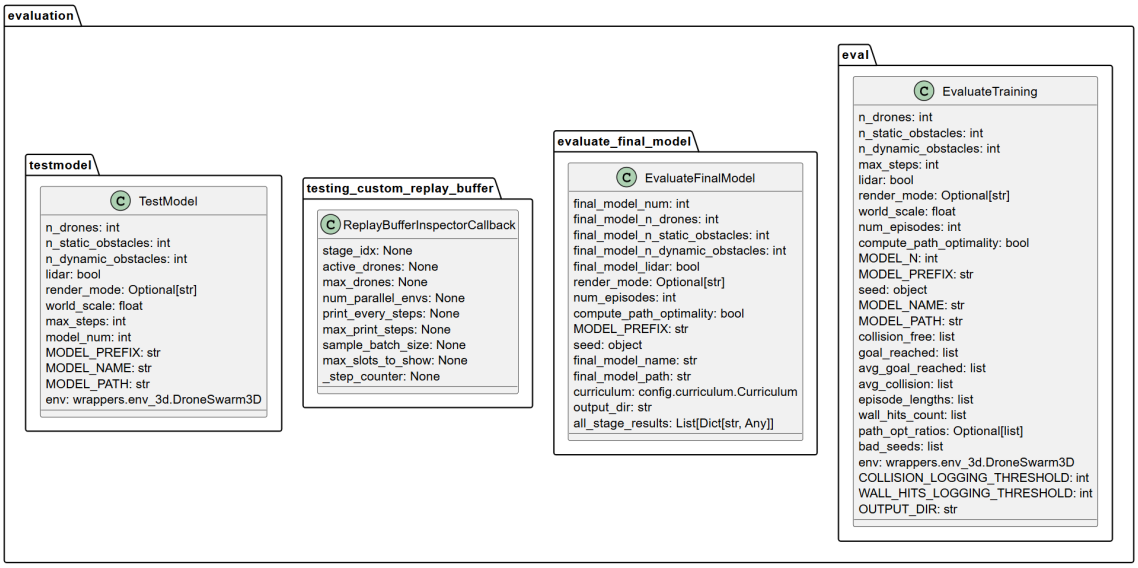

FIGURE 11.9: Evaluation

11.2.4 Object Building

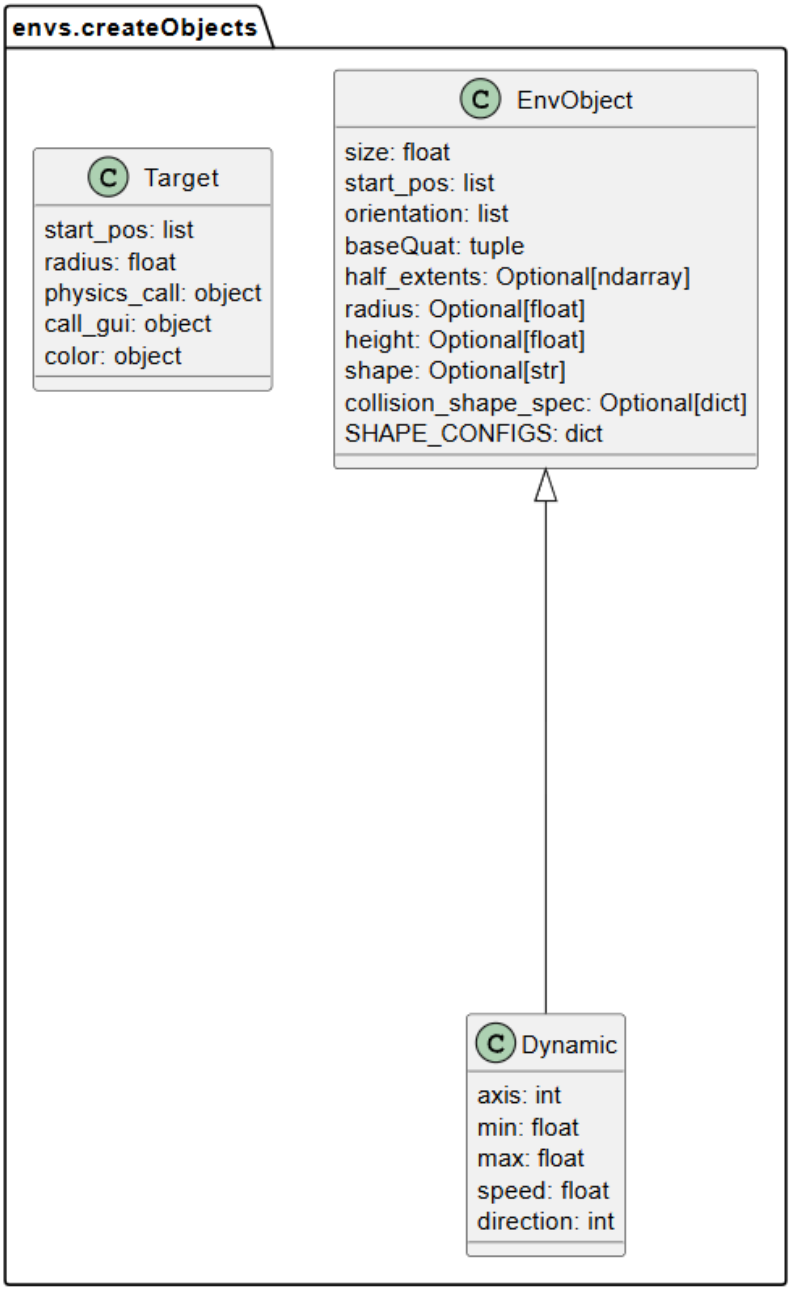

FIGURE 11.10: Object Factory

11.3 Data Flow Diagrams

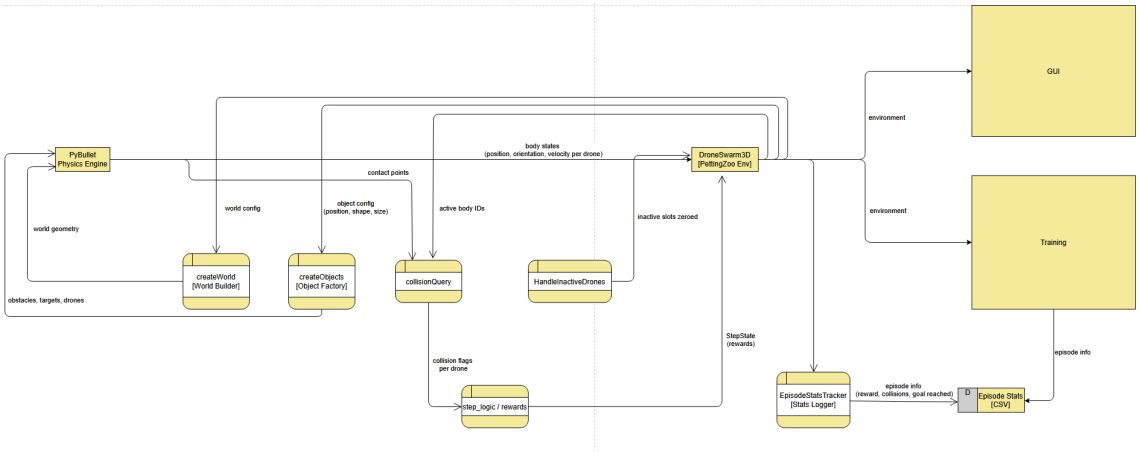

FIGURE 11.11: Environment Data Flow

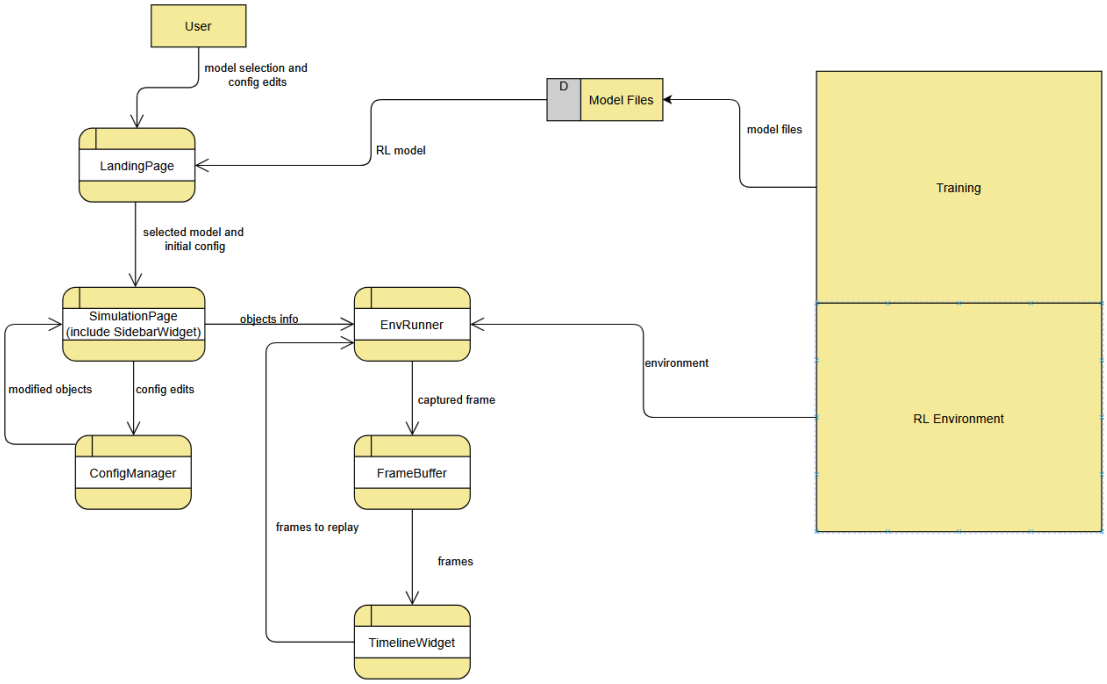

FIGURE 11.12: Interface Data Flow

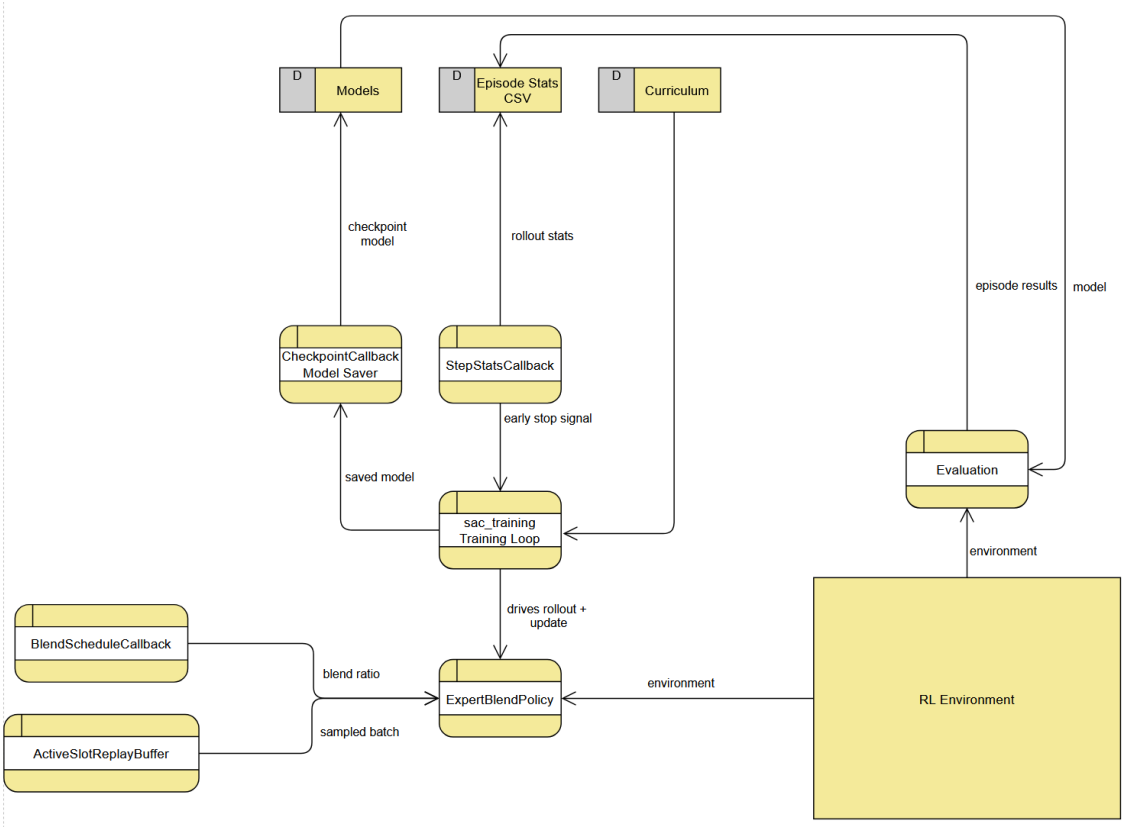

FIGURE 11.13: Training Data Flow