Unreal Engine Plugin Project

Daniel Appel, Yuval Ben Simhon

June 2024

1 Introduction

1.1 Abstract

We live in an era where artificial intelligence (AI) is an integral part of our daily lives, whether in an air conditioner, in a car, in a plane, or in a toaster oven. With the need for AI, there naturally comes the need for training the AI models and classifiers using various techniques, whether providing large quantities of data or providing an environment for automated systems to produce and train the models and classifiers.

One such technique that has gained significant attention in recent years is reinforcement learning (RL). RL is a branch of machine learning in which an agent learns to make decisions by performing actions and receiving feedback from those actions in the form of rewards or penalties. This approach is particularly effective and well-suited for training autonomous systems (such as vehicles and drones) to perform complex tasks in dynamic and uncertain settings. The adaptive nature of RL allows these systems to continuously improve their performance through trial and error, making them highly versatile and capable.

In autonomous drone and vehicle (agent) training, the use of simulated environments is often very desirable as it provides an environment for training the agent without using their real-life counterparts, thus drastically reducing the monetary expenses for training and introduces the ability to speed up the training by speeding up the simulations and also use multiple agents simultaneously. This, in addition to potentially avoiding any bureaucratic inconveniences which arise from real world testing on live agents. Game engines are the most popular simulated environments for many such endeavors, since they provide photo-realism in the environments and are easily adapted for RL training purposes.

One such gaming engine is Unreal Engine (UE), which is often chosen due to its powerful capabilities, even compared to other game engines. UE is a state-of-the-art game engine renowned for its high-fidelity simulations and robust physics modeling. These features are essential for creating realistic and photorealistic training scenarios that closely mimic real-world conditions. The use of UE ensures that the training environment is both immersive and accurate, which is crucial for the successful deployment of RL-trained systems in real-world applications. If needed, UE also provides the tools needed to mimic real-world physics (such as enabling gravity), which can also prove to be of great help in training agents.

Python, on the other hand, is the preferred programming language for implementing RL algorithms due to its simplicity and the vast array of machine learning libraries it supports, such as TensorFlow and PyTorch. These libraries provide comprehensive tools for developing, training, and evaluating RL models. Python's popularity in the machine learning community ensures that there is extensive documentation and a large support network, which facilitates the development process and enables rapid prototyping and experimentation.

The combination of Unreal Engine and Python represents a powerful synergy for training autonomous systems using reinforcement learning. The realistic simulations of UE, coupled with Python's extensive machine learning libraries, provide a robust platform for developing sophisticated AI models.

The most prominent library designed to connect Unreal Engine (UE) with Python was AirSim, developed by Microsoft. Although AirSim was initially embraced for its potential to bridge UE's high-fidelity simulations with Python's machine learning capabilities, it became evident that the library had notable shortcomings, especially after Microsoft ceased its development and support. Several projects revealed significant flaws in AirSim, particularly

in regard to its efficiency and performance. These issues included poor scalability when handling multiple agents or complex environments, substantial latency in data transmission, especially with high-resolution sensory data, and limited integration with popular machine learning libraries like TensorFlow and PyTorch. Such inefficiencies in data transmission and processing can severely impede the intensive training regimes required for reinforcement learning (RL), often resulting in sub-optimal model performance and prolonged or stalled development cycles.

Addressing these shortcomings requires the creation of more refined frameworks that go beyond the capabilities of AirSim. These new frameworks must seamlessly integrate the state-of-the-art simulation capabilities of UE with the sophisticated RL algorithms implemented in Python. The design of these frameworks should focus on optimizing the communication pathways between the simulation environment and the learning algorithms to ensure rapid and efficient data processing. This entails enhancing data exchange protocols to reduce latency, improving scalability to manage multiple simultaneous agents effectively, and providing robust support for real-time adjustments in complex training scenarios.

Overcoming the inefficiencies of existing integration tools through the development of optimized frameworks is essential to unlock the full potential of RL in training autonomous systems, paving the way for advanced applications in various industries.

1.2 Previous Related Works

1.2.1 AirSim Drone Racing Lab [MGV+20]

The paper presents the AirSim Drone Racing Lab, a simulation framework for prototyping algorithms and machine learning research in autonomous drone racing. Provides realistic environments, sensor modalities, and tools for benchmarking algorithms. However, additional frameworks and features are needed for comprehensive drone racing research.

1.2.2 A3C for drone autonomous driving using Airsim [VMGMRA21]

In this work, there is using artificial intelligence to autonomously guide a drone to a target point in a virtual environment created by Unreal Engine, utilizing the Airsim plugin. The Asynchronous Actor-Critic Advantage (A3C) algorithm, which requires fewer computing resources, is employed, and experiments show that increasing the number of parallel simulations improves learning stability and efficiency.

1.2.3 Autonomous Flight Control System for Drones to Reach Targets [Abe20]

The paper presents the development of a Python-based controller to navigate quadcopter drones in 3D space, comprising an acceleration controller, a converter for orientation and altitude, and a PID controller. Simulated in AirSim, the controller successfully guides the drone through various goals, demonstrating potential for real-world application.

1.2.4 Drone Simulation Using Unreal Engine [Jir]

This paper presents a photorealistic drone simulator based on Unreal Engine 5, featuring LiDAR, RGB cameras, and multiple drone support. The simulator, which supports fast TCP/IP communication, was tested in various scenarios and confirmed to be effective for training drone models in reinforcement learning tasks.

1.2.5 Flying Free: A Research Overview of Deep Learning in Drone Navigation Autonomy [LMC21]

This paper maps vehicular autonomy levels to drone navigation tasks to classify drone autonomy, providing a top-down examination of research in the area. It serves as a guide to understanding current research and identifying opportunities in drone autonomy, especially focusing on Deep Learning-based solutions.

1.2.6 Automatic Drone Navigation in Realistic 3D Landscapes using Deep Reinforcement Learning [SKK19]

This study uses deep reinforcement learning (RL) to navigate a drone through 3D obstacles, comparing its performance with human pilots. The results show that the Double Dueling DQN algorithm outperforms other RL algorithms and performs better than novice pilots, but expert pilots still surpass all algorithms.

1.2.7 An Efficient Simulation Platform for Testing and Validating Autonomous Navigation Algorithms for Multi-rotor UAVs Based on Unreal Engine [YZP+19]

This paper presents a simulation platform using Unreal Engine to test autonomous navigation algorithms for UAVs without GNSS signals. This realistic simulation addresses the high costs and data demands of real-world testing and deep learning techniques.

1.2.8 A Parameter Sharing Method for Reinforcement Learning Model between AirSim and UAVs [HTL+18]

This study uses reinforcement learning to train UAVs for autonomous landing in a virtual environment, reducing the time and cost of real-world training. Q-learning is used, making the transfer of AI to real-world UAVs more efficient.

1.2.9 Autonomous UAV Navigation Using Reinforcement Learning [PLFSN18]

This paper presents a framework using reinforcement learning for UAV navigation in unknown environments, validated through simulation and real-world implementation. It addresses technical aspects of RL in UAV systems, enabling future research in wildfire monitoring and search and rescue missions.

1.2.10 AirSim: High-Fidelity Visual and Physical Simulation for Autonomous Vehicles [SDLK18]

This paper presents a new simulator built on Unreal Engine for developing and testing autonomous vehicle algorithms. It offers realistic simulations, supports real-time HITL with protocols such as MavLink, and is extensible for various vehicles and hardware, demonstrated with a quadrotor.

1.3 Our impact

In our project and this article, we introduce a new, simpler yet hopefully more efficient framework for connecting UE and Python. By setting Airsim aside, with its large yet clumsy set of capabilities, we developed an ability to connect a Python program to an instance of an Unreal Engine based game. The game in Unreal Engine includes a drone, with the ability to control its flight, sight, etc. through the Python program. The Python program is also able to receive images from a camera (virtual or physical) from which it analyzes the images received from the drone in the simulation. The camera is assumed to be a First-Person View (FPV) of the drone. This schema is shown in 1

2 Project Structure

2.1 Unreal Engine Project Overview

In an attempt to achieve photorealism in our simulation, the Unreal Engine platform for game development was used, which is known for its ability to simulate photorealistic environments. It was decided to choose Unreal Engine 5.3 for our project, as at the time it was the newest and most advanced version of the engine (the project was also tested using earlier versions of UE, which required only minor changes to the project in order to work).

To structure our environment, the "Middle East" map (see Figure 2) was chosen to simulate an urban middle eastern setting. This, of course, can be changed to any other desirable environment. In addition to the map, two plugins were used:

- “City Sample Crowd” – used to spawn person characters in the simulation, and they would be the targets of the drone.
- “UDP Unreal” – an open-source plugin which enables UDP communications to and from UE via blueprints. This was used for all communications between the UE and the Python client in the project.

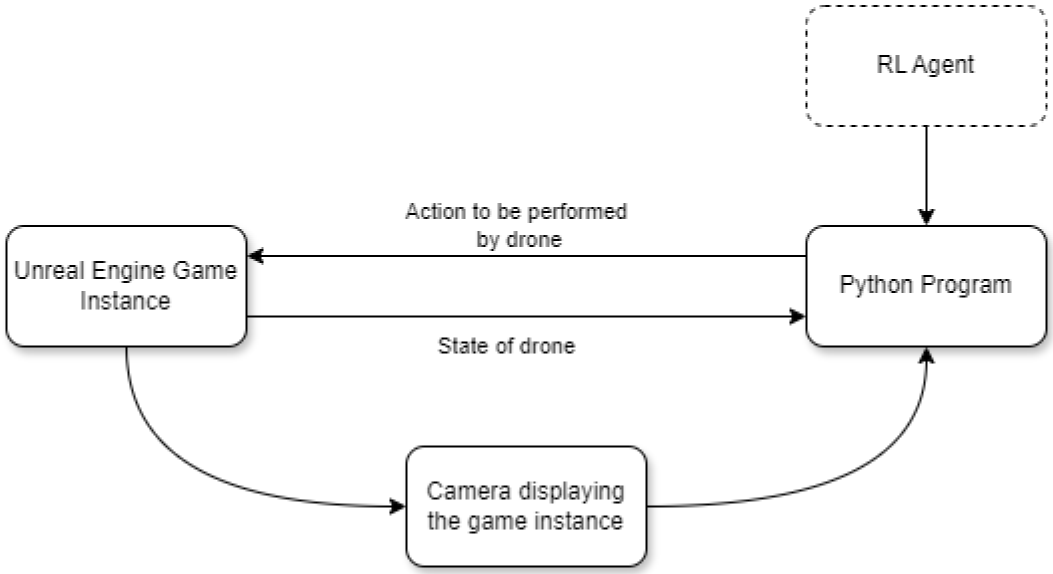

RL Agent
Action to be performed by drone
Unreal Engine Game Instance
Python Program
State of drone
Camera displaying the game instance

Figure 1: Schema of our implementation

2.1.1 Structure of project

The structure and purpose of the working files/folders inside the project are as follows:

- “Crowd” folder – contains the “City Sample Crowd” plugin
- “MiddleEast” folder – contains the map and all the map’s assets
- “QuadcopterSimple” folder – contains the blueprint of the drone, including the visual representation of the drone and the logic of data transfers (here, most of the logic of the simulation is located), the other portion of which can be found inside the level blueprint
- “BP_UDP_Actor” – a blueprint which must exist in the world in order for the UDP communications to work (as stated in the plugin documentation). In the world, the blueprint is invisible.
- “BP_SpawnerPerson” – a blueprint which, when placed in the world, represents a possible spawn location of a person actor. In the current setup of the project, 150 such spawners have been placed on the map.

2.1.2 Level Blueprint

The level blueprint contains the logic of default possession at the start of the game (possession of the drone to enable controls) and the logic for changing the time of day in the simulation. The values of the requested change (the number of degrees) are taken from the quadcopter blueprint.

2.1.3 Quadcopter Blueprint

This blueprint, as stated before, contains the overwhelming majority of the logic of the entire simulation. Here, the ability is given to the user/player to play in two ways (not exclusively, but it is our belief that most use cases will, by their definition, exclude one of the use cases for the sake of maintainability and control):

- The ability to play directly from the simulation, using UE’s built-in controls. This would allow the control of flight in all aspects, but not much else
- The ability to use outside communications to control the drone in terms of flight, the simulation as a whole for object spawning, time-of-day change, all this while allowing to request information regarding the drone in real-time.

  A Python program was developed in parallel, to demonstrate these abilities of control of the simulation, as

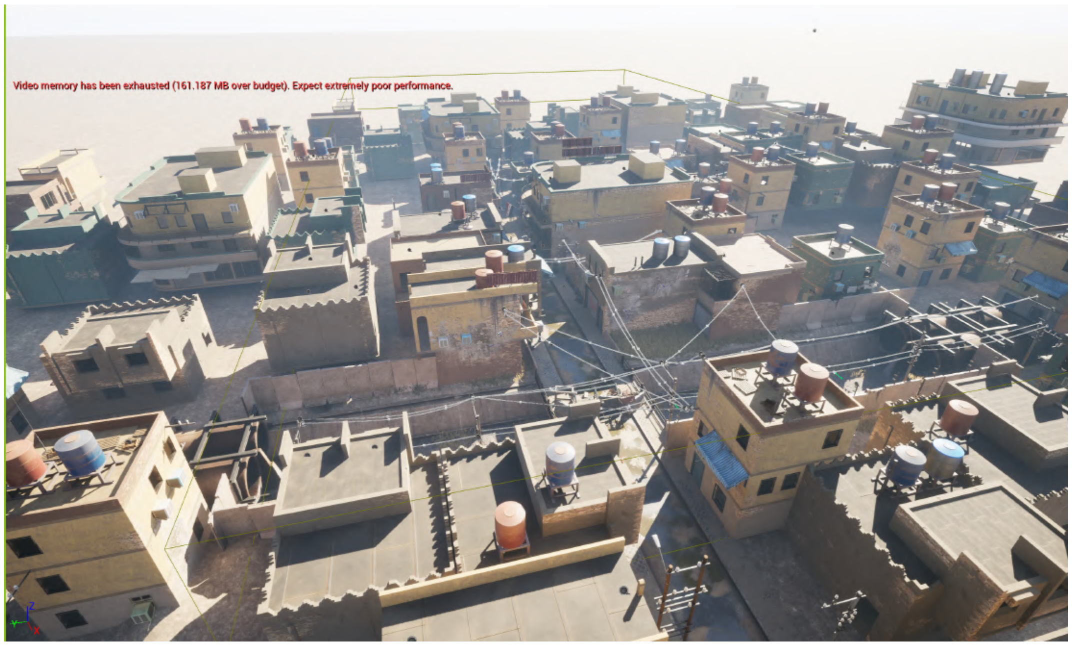

Video memory has been exhausted (161.187 MB over budget). Expect extremely poor performance.

Figure 2: ”MiddleEast” environment map in Unreal Engine

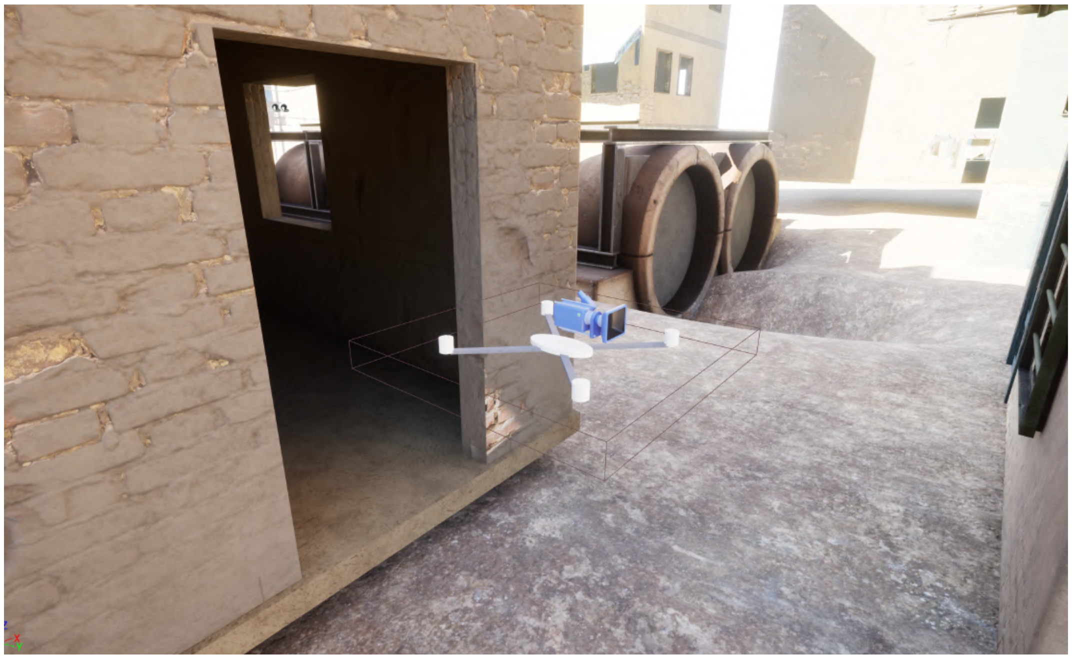


Figure 3: Visual representation of the drone

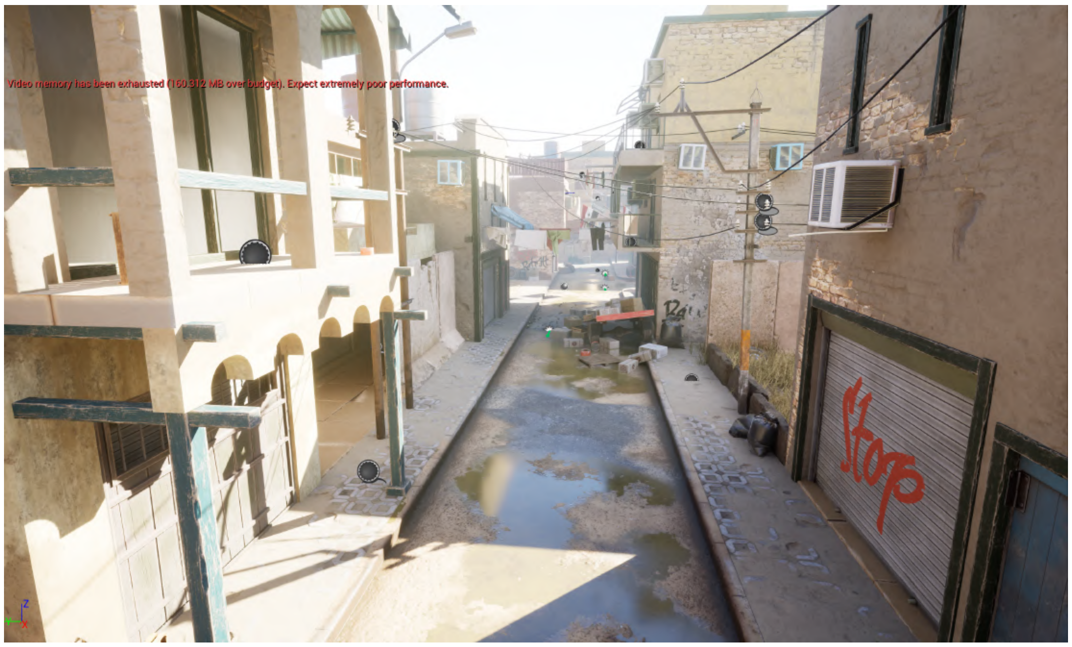

Video memory has been exhausted (160.312 MB over budget). Expect extremely poor performance.
Stop

Figure 4: One of the streets in the simulation

well as a proof of concept that this is indeed possible. The exact capabilities of the simulations can therefore be left for discussion in the Python program later in the essay.

2.2 Python Project Overview

Python was chosen as a language of great potential for use in machine learning, deep learning, and AI, as well as reinforcement learning, which is the focus and main idea behind the project. Python allows for readable and maintainable code, as well as relatively easy implementations of the needed functionality for the purposes of this proof of concept.

The project we have created implements an API to control the simulation (in Unreal Engine), in two main ways: drone controls, with the ability to receive the state of the drone as well, and simulation controls, including spawning objects and time of day change (more on both aspects below). The project also demonstrates two possible ways to run and test the capabilities of the API:

1. The first is a human run, where a human player is, in essence, playing a video game, the purpose of which is to fly a drone (quadcopter), and find the maximum number of targets (people) in the simulation.
2. The second is a simple and deterministic (dummy) algorithm for terrain scanning without the use of a human player. The essence of the algorithm is to fly at a certain altitude, in a certain pattern (see Figure 5) around the map, with the camera pointing slightly down, while trying to recognize (through the use of image recognition – more on this later) the targets.

2.2.1 Project Structure

The project contains 6 directories:

1. "Player": One of two main folders, containing a "main" method. This folder is used for the first use case of the project, e.g. the player-controlled game. The program can be used to establish a baseline for future works with the API for reinforcement and deep learning.
2. "AI": One of two main folders, containing a "main" method. This folder is used for the second use case of the project, e.g. the dummy algorithm. Here, the capabilities of image recognition using YOLOv5 are also shown and presented. The program can, too, be used as a second baseline for future works with the API for

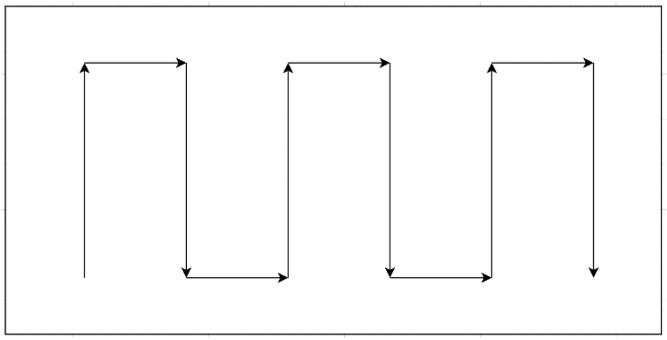


Figure 5: Schematic representation of drone travelling across the map in Unreal Engine using the dummy algorithm

reinforcement and deep learning.

3. “ConfigFiles”: This folder is used to construct (using a python script) and store the configuration files of the project. The configuration files contain the simulation parameters, so that they can easily be modified, without the need to modify the source files. This folder also contains a simple GUI, which can be used as another way to modify the desired parameters.
4. “Core”: Contains the classes which are used by both of the main folders of the project (AI and Player). Consists mainly of data classes and the main API of communication between Python and Unreal Engine, as well as a logger to log the results of runs of the simulation.
5. “LOGS”: The directory where all the logs are being stored from the runs of the simulation.
6. “YoloImpl”: the implementation of YOLOv5 detection. This folder consists mainly of files taken directly from the repository of YOLO, with modifications to the parameters and the output of the model to suit the purposes of this project.

2.2.2 Deep Dive into the project files and their functionality

Here, we shall explain in greater detail the contents of the folders and some of the more important functionality: Core

- Core:
  - ”Requirements” file, which contains the project requirements in terms of libraries and their versions.
  - “Logger”: a class which is responsible for the logging of the project into the log files which are to be saved in the “LOGS” folder.
  - “PublicDroneControl”: The API responsible for all communications between Python and Unreal Engine. This class is explained in detail later in the essay.
  - Data classes, into which some of the responses of the API are mapped.
- Player:
  - Main: The file which aggregates and runs all relevant threads in order to allow player control and evaluation.
  - “PlayerControlThread”: The class and thread responsible for allowing user (player) control over the simulation, including drone controls and a button signifying “detected target” which is verified through UE and when verified, the points are aggregated via the GradePlayer class.
  - “GradePlayer”: The class and thread responsible for documenting the player grade at all points in time.
- AI:
  - Main: The file which aggregates and runs all relevant threads in order to run the and evaluate the dummy algorithm

  - "DummyAlgoThread": The class and thread responsible for running the dummy algorithm in terms of drone control.
  - "GradeAI": The class and thread responsible for documenting the grade of the simulation at all points in time. This class contains the image processing which is used to identify targets in the simulation. The images for the image processing are received by reading a webcam frame whenever requested. This is needed since the UE client and the Python program do not necessarily have to be on the same PC, since all communications are performed via UDP sockets. Hence, the assumption is made that the Unreal Engine video feed is passed to the PC with the Python program running on it. If of the same PC, a virtual camera from OBS studio was used and tested.

    The class can run an Aruco or human target detection (or both). Both of these are run on separate threads so as not to clog up the runtime of the program. Whenever a new recognition is made, a request to add points is made to the Grade class.

2.2.3 Public API for Communications

In this section, the API of drone controls and simulation is explained in detail. The API is thread-safe, and the reception of messages from UE is run on a separate thread.

- **moveDroneUp**(self, speedMultiplier: float): This function is used to move the drone up.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **moveDroneDown**(self, speedMultiplier: float): This function is used to move the drone down.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **moveDroneForward**(self, speedMultiplier: float): This function is used to move the drone forward.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **moveDroneBackward**(self, speedMultiplier: float): This function is used to move the drone backward.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **moveDroneRight**(self, speedMultiplier: float): This function is used to move the drone right.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **moveDroneLeft**(self, speedMultiplier: float): This function is used to move the drone left.
  - Receives:
    - speedMultiplier: float: speed at which to move the drone (multiplier of default speed)
  - Returns: None
- **rotateDroneRight**(self, speedMultiplier: float): This function is used to rotate the drone right.
  - Receives:

    - speedMultiplier: float: speed at which to rotate the drone (multiplier of default speed)
  - Returns: None
- **rotateDroneLeft**(self, speedMultiplier: float): This function is used to rotate the drone left.
  - Receives:
    - speedMultiplier: float: speed at which to rotate the drone (multiplier of default speed)
  - Returns: None
- **rotateCameraDown**(self, speedMultiplier: float): This function is used to rotate the camera down.
  - Receives:
    - speedMultiplier: float: speed at which to rotate the camera (multiplier of default speed)
  - Returns: None
- **rotateCameraUp**(self, speedMultiplier: float): This function is used to rotate the camera down.
  - Receives:
    - speedMultiplier: float: speed at which to rotate the camera (multiplier of default speed)
  - Returns: None
- **hoverDrone**(self): This function is used to hover the drone.
  - Receives: None
  - Returns: None
- **rotateDroneXDegreesAtSpeed**(self, degrees: float, speed: float = 90): This function is used to rotate the drone a certain number of degrees at a certain speed.
  - Receives:
    - degrees: float: number of degrees to rotate
    - speed: float: speed at which to rotate (in degrees per second). Defaults to 90.
  - Returns: None
- **rotateDroneTowardsLocation**(self, x: float, y: float, z: float, speed: float = 90): This function is used to rotate the drone towards a certain location at a certain speed.
  - Receives:
    - x: float: x coordinate of target location on UE grid
    - y: float: y coordinate of target location on UE grid
    - z: float: z coordinate of target location on UE grid
    - speed: float: speed at which to rotate (in degrees per second). Defaults to 90.
  - Returns: None
- **moveDroneToLocation**(self, x: float, y: float, z: float, speed: float = 2, turnWithMove: bool = True): This function is used to move the drone to a certain location at a certain speed. The function blocks the runtime until a reply of "done" is received from UE.
  - Receives:
    - x: float: x coordinate of target location on UE grid
    - y: float: y coordinate of target location on UE grid
    - z: float: z coordinate of target location on UE grid
    - speed: float: speed at which to rotate (in degrees per second). Defaults to 90.

    - turnWithMove: boolean: if true, drone will turn while moving to face the target location at the end of the travelling
  - Returns: None

- **turnCameraXDegreesAtSpeed**(self, degrees: float, speedMultiplier: float = 45): This function is used to turn the camera a certain number of degrees at a certain speed.
  - Receives:
    - degrees: float: number of degrees to turn. value must be in the range of -89 to 89. If not, the value will be clamped to this range. Positive values turn down, negative values turn up
    - speedMultiplier: float: speed at which to turn (multiplier of default speed)
  - Returns: None
- **getDroneState**(self): This function is used to get the current state of the drone.
  - Receives: None
  - Returns: DroneState object. Includes the location of the drone (in UE grid coordinates) and the number of collisions up to that point.
- **sendDroneGrade**(self, grade: float): This function is used to send the grade of the drone to the UE to be displayed on the screen.
  - Receives:
    - grade: float: grade of the drone
  - Returns: None
- **getDistanceToCameraDirection**(self): This function is used to get the distance to the nearest object in the direction of where the camera is facing.
  - Receives: None
  - Returns: distance to nearest object in direction of camera in meters.
- **getCameraTarget**(self): This function is used to get the information regarding the nearest object in the direction of where the camera is facing.
  - Receives: None
  - Returns: Target object containing display name of target, class name of target, location of target (in X, Y, Z coordinates in UE units).
- **getTargetOfPoint**(self, coordinateX: float, coordinateY: float): This function is used to get the information regarding the nearest object in the direction of a set of coordinates on screen (in 2D space). UE parses this into 3D space and returns the relevant information.

  The coordinates passed should be of the simulation only, without borders or any blank spaces.
  - Receives:
    - coordinateX: float: Normalized X coordinate in 2D space of the simulation (on screen)
    - coordinateY: float: Normalized Y coordinate in 2D space of the simulation (on screen)
  - Returns: Target object containing display name of target, class name of target, location of target (in X, Y, Z coordinates in UE units)
- **requestDaytimeChange**(self, addDegrees: float): This function is used to request a change in the time of day in the simulation.
  - Receives:
    - addDegrees: float: add this number of degrees to the current positioning of the sun (direction of the light)

    - Returns: None

- **spawnXActors**(self, numOfActorsToSpawn: int): The method should be called at start of simulation This method spawns people NPC in the simulation, where numOfActorsToSpawn is the number of actors which will be spawned randomly. All NPC are randomized in their appearance (male/female/skin and body types/clothing).
    - Receives:
        * addDegrees: float: numOfActorsToSpawn: int: The number of actors which will be spawned randomly (max value for this simulation is 150)
    - Returns: list of Coordinate object (x, y, z coordinates) of the spawned actors.
- **verifyAndDestroyActorFromCamera**(self): Method to verify that the target is a spawned actor, and if verified, remove the actor from the simulation.
    - Receives: None
    - Returns: True if verified, false otherwise
- **verifyAndDestroyActorFromPoint**(self, normalizedX, normalizedY): Method to verify that the target is a spawned actor, and if verified, remove the actor from the simulation.
    - Receives:
        * coordinateX: float: Normalized X coordinate in 2D space of the simulation (on screen)
        * coordinateY: float: Normalized Y coordinate in 2D space of the simulation (on screen)
    - Returns: True if verified, false otherwise

3 The Framework - System Description

The following chapter is a detailed overview of how we developed and used the framework, focusing on the installation and running instructions, to allow a reproduction of our results, as well as future works and expansions on the framework. Moreover, a demonstration of the possibilities and capabilities of the framework is provided in the form of two variations of running scenarios (one using deterministic AI and another with a player-controlled simulation), which are to be presented in due time.

3.1 Framework Usage instructions

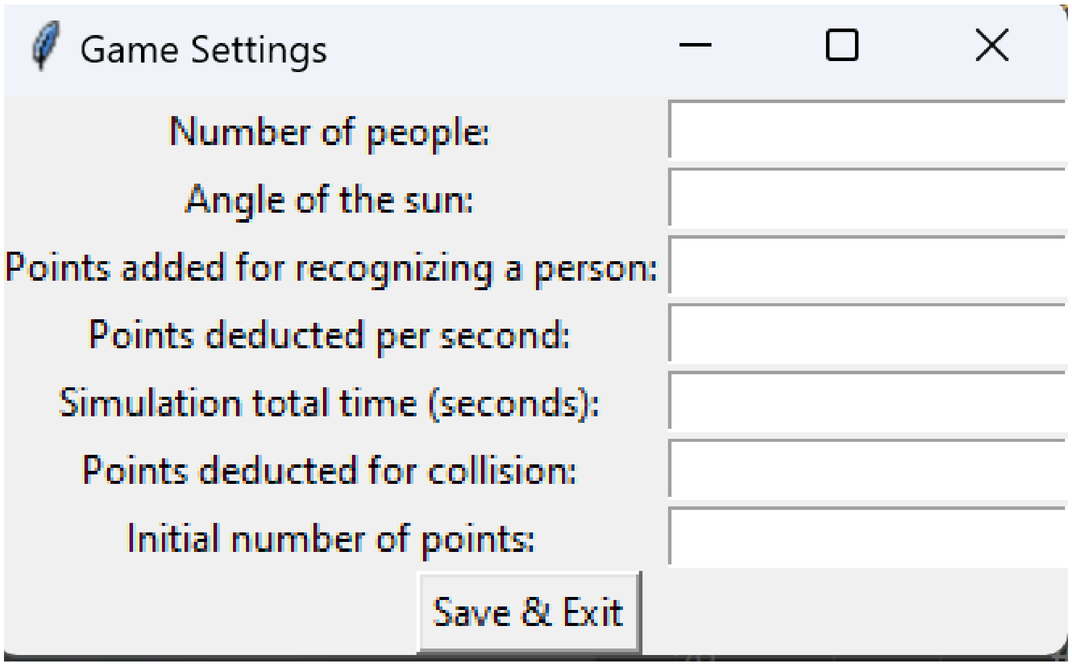
3.1.1 Python Setup

The Python requirements and installation requirements are presented in the form of two "requirements" files, one with the bare requirements of the project, and the other with the addition of all dependency requirements which are downloaded together with the bare minimum requirements. We shall not delve into the specifics of said requirements; however, Python 3.9 or newer is required for them to work (previous versions of Python were not tested).

3.1.1.1 Configuration Files

Two configuration files are present in the project, which can be edited before the start of each simulation, if such customization is required. The first (gradeConfig.json) contains general parameters of the simulation, such as the number of people in the simulation, the angle of the Sun, initial points, and the number of points added/deducted for an event. The second (graphPointsConfig.json) contains a list of points in the 3D space corresponding to the dummy algorithm's flight path.

A GUI has been implemented to modify the former of the two files (see 6), which can be run in the Gui.py file. Another option is to modify the files directly, or via a script provided in the same folder. The files are loaded in runtime into the program, hence all three variations are equivalent for the purposes of parameter initialization of our project. Note: the location (the file hierarchy) of the configuration files cannot change without a modification to the source code.


Game Settings

Number of people:

Angle of the sun:

Points added for recognizing a person:

Points deducted per second:

Simulation total time (seconds):

Points deducted for collision:

Initial number of points:

Save & Exit

Figure 6: Python GUI to initialize the simulation configuration file

3.1.1.2 Logger

Upon running one of the two simulations, a logger is run, where all actions are transcribed. The logs of the previous runs of the simulations are saved inside the ”logs” directory, which must be present in the project hierarchy in the root folder (an exception is thrown otherwise.

3.1.1.3 YOLO Settings

The YOLO framework, in itself, provides numerous settings that could be changed to improve or change the result. During our research, we encountered some that were crucial in our attempts to optimize the functionality and enable the working of YOLO. The changes were all made in the file: MainYoloDetect.py. These are presented below:

1. In order to change the input source from images to webcam, an exchange was made from the default line to the following:

```
parser.add_argument('--source', type=str, default=0,
    help='file/dir/URL/glob/screen/0(webcam)')
```

2. In order to change the threshold for object detection, we changed the default value to 0.25. By changing the threshold probability value, the model's output can directly be affected. The line was changed as follows:

```
parser.add_argument('--conf-thres', type=float, default=0.25, help='confidence
    threshold')
```

3. Since YOLO performs CPU-intensive tasks on images, an ability to use GPU instead of CPU exists to perform the necessary calculations and thus improve efficiency and lessen the burden on the CPU. This can be done by using CUDA and specifying this in a parameter as follows:

```
        parser.add_argument('--device', default='', help='cuda device, i.e. 0 or 0,1,2,3 or cpu')
```

The examples provided with the framework, which use YOLO, only use its ability to detect people, but another use might be to detect any of the multitude of objects which YOLO has been trained on (the full list may be found in the YOLO documentation). The filter to detect only persons and disregard anything else was implemented in the file GradeAI.py in the method handleYoloDetection.

3.1.1.4 Running

In order to run the Python program, one of the two simulation settings must be chosen: Either a player-driven simulation, where a player controls the drone movements via the keyboard, or the dummy AI-driven simulation, where a simple deterministic map-scanning algorithm is implemented, moving the drone following the schema 5. In accordance with the choice made, either MainPlyayer.py or MainAi.py files are to be run, respectively.

If a custom or more complex algorithm (or agent) is to be implemented, an API in the file PublicDroneControl.py is provided to connect to the UE client. An object is to be created of the said class, upon which all available methods can be run. For inspiration on image detection, see the GradeAI.py file. For inspiration on the usage of the API, see PlayerControlsThread.py, DummyAlgoThread.py, and others.

3.1.2 Unreal Engine Setup

First and foremost, a choice must be made on the subject of the necessity and/or the willingness to change anything and everything in the simulation. If the simulation provided is sufficient for one's purposes, then the only setup needed is downloading the compiled game and running it on the client PC (more on this in 3.1.3). Since "such simplicity" is doubtfully the case, this section of the essay will delve into the specifics regarding how the source project is structured, what are its dependencies, and how to edit any and all aspects of the simulation.

3.1.2.1 Dependencies

The main dependency of the simulation is Unreal Engine 5.3, which must be downloaded through the Epic Store. The simulation also heavily relies on two distinct plugins, both are located in the "plugins" directory:

- "City Sample Crowd" – used to spawn person characters in the simulation, and they would be the targets of the drone in the provided simulation (if human recognition is not needed, the plugin may be removed, and all its logic and references removed from the blueprints of the project).
- "UDP Unreal" – an open source plugin that enables UDP-based communications to and from UE via blueprints. This was used for all communications between the UE and the Python client in the project and thus is an integral part of the framework, and must be present in all simulations.

3.1.2.2 Possible Modifications

There are an almost infinite number of modifications possible in UE, and more specifically in the project presented, but some can be combined into categories, and briefly explained on their limitations with regards to the framework presented:

- Map Modifications: All current map assets can be plainly seen when opening the UE project of the framework and can be replaced at will. They are located in the three directories: Lighting, Map Actors, RenderFx. That said, there is a pseudo-dependency to the Spawners directory.
- Spawners: These are objects of the "BP_Spawner" blueprint class. These are empty (visually) objects, but they indicate the locations where persons may be spawned randomly. Thus, they may be removed (if not needed), moved (if for example the map was changed), or more can be added to map. Note: If their number is changed, the relevant parameter must be changed in the UE and in Python for consistent results (since the verification is performed on both sides).
- BP_QuadcopterSimple is an object where the main functionality of the framework is located. The only change which may be done here without risk of program failure is the visual representation of the drone (no use is

done on it). Note: FloatingPawnMovement and UDPComponent components must be present and preferably unchanged in the blueprint hierarchy.

3.1.2.3 Finishing Editing

When the editing of the environment is complete, the simulation must be packaged and exported (only Windows packaging was tested). This can be done by following 7.

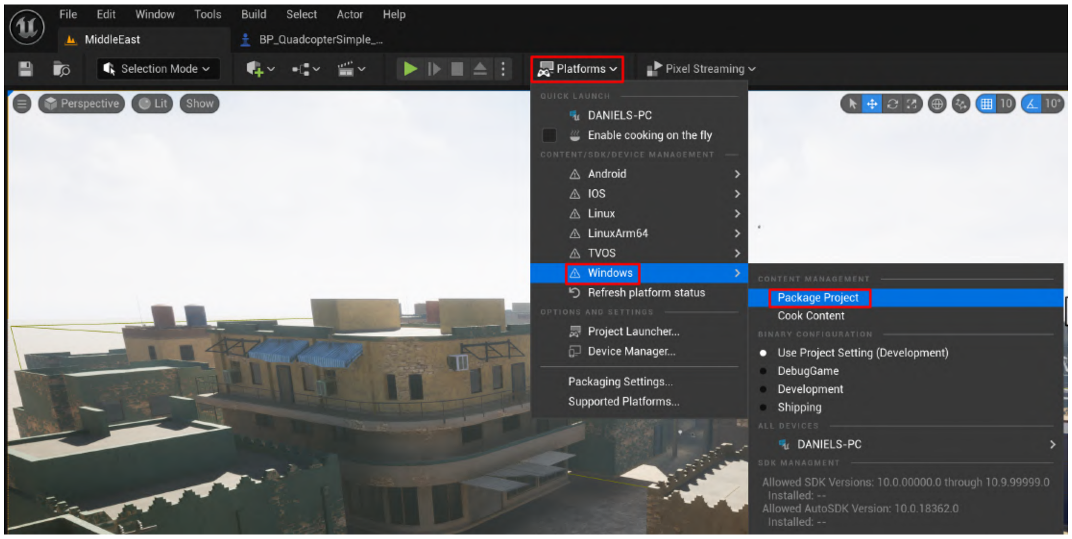


Figure 7: Screenshot on the way to compile, package and export the UE project

3.1.3 Simulation Setup

In this section, the final preparations and the running procedures of the framework and its components are described:

1. Have a completed Python program.
2. Have a Packaged Unreal Engine Project.
3. If running both the simulation and the Python program on a single PC, the program "OBS Studio" is advised to be installed, and the virtual camera it provides is advised to be turned on to show the simulation screen (as shown in 8), in order to enable image processing via Python. If the UE and the Python are run on separate PCs, they must be connected to the same local network for UDP communications. Also, in order to enable fast and reliable screen capture from one PC to another, an HDMI-to-USB cable is advised to be used.
4. Run the simulation (UE project).
5. Toggle the virtual camera or connect the HDMI-to-USB cable and configure it to display the simulation.
6. Run the Python program and perform the simulation as required.

3.2 Results

The framework that was developed and tested on the simple use cases which were explained previously, was proven successful after multiple runs of both scenarios, averaging at 30 FPS after the connection to Python was established, with constant actions sent from Python to UE and with constant frame extraction and image processing (the image processing frame-rate was not measured, since it is mostly dependant on hardware of the specific PC). Some screenshots were taken during simulation runs (see 9, 10, 11, 12, 13, 14)

The PC on which the tests were performed was using Windows 11 64-bit OS, running an i7-9700k CPU, NVIDIA GeForce 1660 Ti GPU and 48GB of RAM.

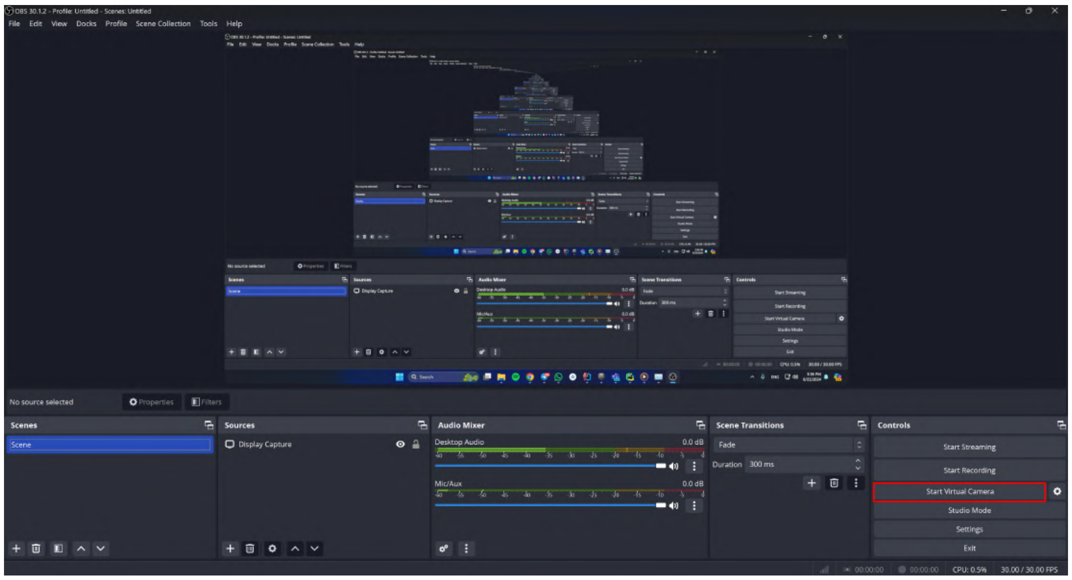


Figure 8: OBS Virtual Camera toggle

The results of the simulation runs were as follows: The dummy AI algorithm recognized on average 5 people out of 30 present in the simulation. The results of the simulation controlled by the player were extreme, ranging from 1 to 20 target recognitions in the allotted time. It is our belief that the variance in the results arises both from player proficiency and target distribution around the map (randomly distributed). For example, a mediocre gamer has scored slightly better than a non-gamer, and a simulation where the targets were located on rooftops scored better than when they were mostly on the streets and in buildings.

3.3 Negative Results

During our research, we encountered some things that we believed were "a good idea to implement", but after some trials with varying degrees of failure, they were discarded and declared "out of scope" for the current article and project.

The most prominent of these is implementing the drone through physics, since currently, the drone is using the UE's standard "move in direction X" movement system. An alternative to that, which may be more realistic and close to reality is implementing the movements of the drone using physics, with up forces from the drone rotors turning and tilting the drone to move in the desired direction.

Another example of what could be a meaningful addition to the framework is a deeper example regarding using the framework in an RL environment, this by implementing and training an RL agent to accomplish some real task. Due to time deficiency, it was decided to postpone this aspect to a future project, thus leaving this project lacking in "real world" examples.

On a more technical note, a complication arises when trying to modify the UE project. UE as a program requires high-end PCs to run smoothly, otherwise stalling. An idea was brought to us to use a 3D rendering of a university campus instead of the "Middle East" map in our project, but our PCs were incapable of loading the map into UE. We believe that a PC with greater resources might be able to accomplish the task, thus providing an even more realistic scenario for the drone to be trained on.

4 Future Works

The current project has established a solid foundation for combining high-fidelity simulations with reinforcement learning (RL) algorithms, specifically targeting the control of autonomous drones. This integration showcases the potential and initial feasibility of using sophisticated simulated environments to train RL models for complex tasks.

Figure 9: Man on a roof in UE, during a trial run

However, to push the boundaries of what these simulations can achieve, it is crucial to explore further enhancements and applications of this framework. This future work could significantly improve the system's capabilities, making the simulations more robust and widely applicable across various domains.

4.1 Enhanced Realism and Physics-Based Simulation

The enhancement of simulation realism remains a priority for future development, critical to bridge the gap between simulated training environments and real-world operational contexts. Current simulations are based on simplified movement dynamics, which do not adequately account for the complexity of physical interactions a drone would experience in a real environment. To address this, future work should focus on developing and integrating sophisticated physics models that replicate the nuanced behaviors of aerodynamic forces, rotor dynamics, and environmental factors such as wind shear and turbulence.

Additionally, enhancing the realism of these simulations could involve the integration of realistic sensory feedback systems, such as visual, infrared, and ultrasonic sensors, to better train drones on navigating and responding to environmental cues. This would provide RL algorithms with a more comprehensive data set from which to learn, enabling more nuanced decision-making capabilities. By simulating complex physical environments and sensory inputs, drones can be better prepared for unforeseen challenges during actual deployment, increasing their reliability and effectiveness in real-world operations.

4.2 Advanced Reinforcement Learning Algorithms

The adoption of more sophisticated RL algorithms is a key component of the future direction of the project. The current reliance on deterministic algorithms provides a stable but limited framework for navigating predictable environments. To truly enhance the drone's operational capabilities, future iterations should explore the incorporation of advanced neural network architectures, such as convolutional neural networks (CNNs) for spatial decision making and recurrent neural networks (RNNs) for tasks that require understanding temporal dynamics.

These advanced models, combined with techniques such as transfer learning, where knowledge gained in one context is applied to different but related problems, could vastly improve the efficiency and adaptability of the learning process. Furthermore, exploring the implementation of unsupervised learning algorithms could enable drones to independently identify and adapt to new challenges without explicit prior programming, thus enhancing their

Figure 10: Woman on in a building in UE, during a trial run

autonomy and effectiveness in dynamic environments. Using cutting-edge RL techniques, the project can push the boundaries of what autonomous drones are capable of achieving.

4.3 Multi-Agent Systems

Exploring the development of multi-agent systems within high-fidelity simulations represents a significant opportunity for future research. Moving beyond the scope of single-drone operations, integrating multiple drones into the simulation can facilitate the study of complex interactions and collaborative strategies among autonomous agents. This approach is particularly relevant for applications such as search and rescue missions, where teams of drones can optimize search patterns and resource allocation to enhance efficiency and effectiveness. Similarly, in environmental monitoring and surveillance tasks, multiple drones can achieve comprehensive area coverage more swiftly and thoroughly than is possible with a single drone.

The technical implementation of such systems would involve the use of advanced multi-agent RL techniques, where each agent (drone) learns not only from its own experiences but also from the interactions with other agents. This can lead to emergent behaviors and cooperative strategies that are difficult to program directly but can be learned and optimized through interaction. By simulating realistic multi-drone operations, researchers can also investigate the dynamics of drone swarming behaviors, collision avoidance protocols, and optimal task distribution among different agents, each equipped with different capabilities.

Furthermore, expanding these simulations to include adversarial scenarios, where drones must compete for resources or strategically hinder other drones, could provide additional information on the resilience and adaptability of RL strategies under competitive pressures. Such environments would not only improve the robustness of the control algorithms but could also contribute to the safety and reliability of drone operations in complex multi-agent settings.

4.4 Enhanced Environment Interactions

To further enhance the training and operational capabilities of drones, it is advisable to expand the range of their interactions with the environment. Currently, the simulated environments offer limited scenarios, which may not fully represent the complexity of real-world operations. Future developments could focus on creating more dynamic and interactive training scenarios that include moving obstacles, active threats, and unpredictably

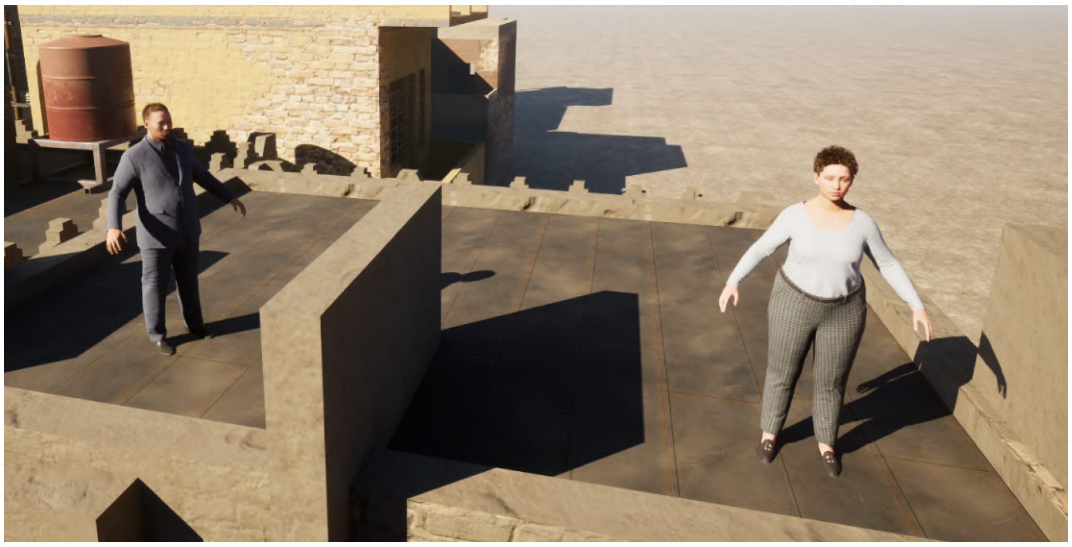


Figure 11: 2 people on a roof in UE, during a trial run

changing conditions, all of which are typical in natural and urban environments.

Utilizing the Unreal Engine's advanced simulation capabilities can facilitate the creation of diverse weather conditions, varied terrain types, and fluctuating environmental factors like lighting and visibility. By introducing these elements, drones can be trained to navigate through and adapt to a wide array of operational challenges, from sudden weather changes to navigating in densely populated urban landscapes with moving vehicles and pedestrians. Such simulations would test not only the drone's navigational skills but also its decision-making process under stress and dynamically changing conditions.

Moreover, incorporating interactive elements such as civilian and wildlife avatars could test drones' abilities to operate safely and effectively around living beings, ensuring their preparedness for deployment in sensitive or crowded areas. The ultimate goal is to create a simulation environment that mirrors the unpredictability of the real world, preparing drones for virtually any scenario they might encounter.

Another avenue to explore is the possibility of the addition of drone interactions with objects to achieve certain goals, such as pulling levers, lifting objects, pushing objects, etc., which could enhance the depth of learning and the depths and complexity of tasks assigned to the drone/s to perform.

4.5 Expanding Application Domains

While the current project focuses on drone navigation, the versatility and scalability of the developed simulation framework hold potential for application in a broader range of autonomous systems. The next phase of research could extend the application of this framework to include autonomous ground vehicles, which face distinct challenges such as navigating complex road networks and interacting with human-driven vehicles. Similarly, underwater robots could benefit from simulations that mimic the challenging conditions of marine environments, such as variable water currents, pressure changes, and obscured visibility, as well as wildlife and plant-life.

Exploring the application of the framework in space exploration scenarios, such as simulating rover operations on lunar or Martian surfaces, could also provide invaluable insights. Each of these domains presents unique challenges that require specialized adaptations of RL algorithms and simulation parameters. By tailoring learning and simulation environments to the specific characteristics and needs of these diverse platforms, the framework could facilitate significant advances in the field of autonomous systems. This expansion not only broadens the impact of current research, but also opens up new avenues for the practical application of RL in high-fidelity simulations, driving forward the capabilities of autonomous technologies in exploring and operating within complex, unstructured, and often hazardous environments.

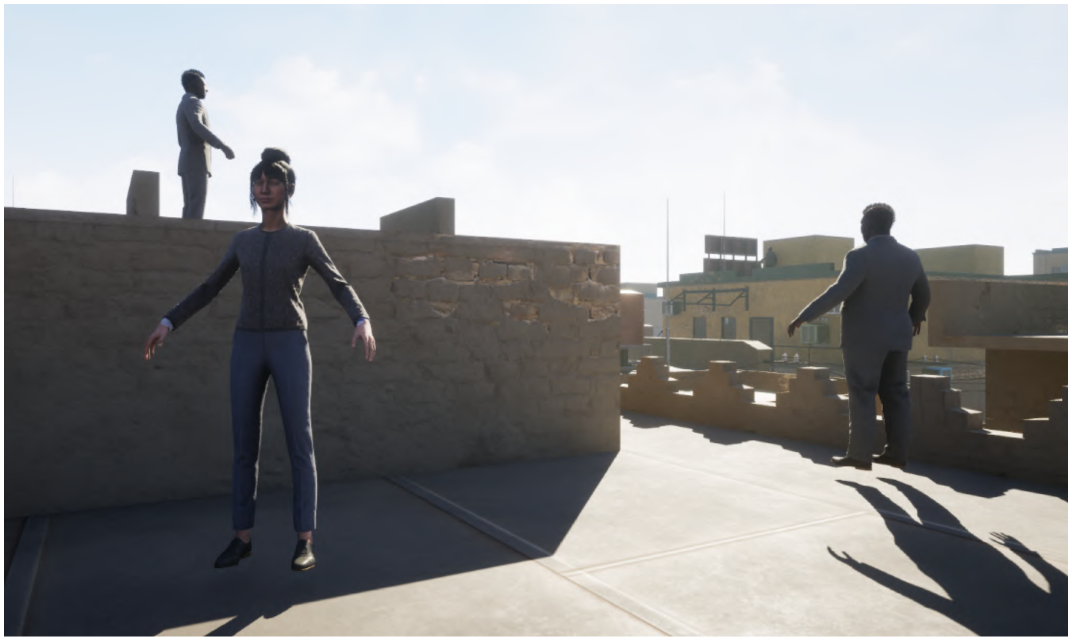


Figure 12: 3 people on a roof in UE, during a trial run

4.6 Conclusion

The Unreal Engine Plugin Project has successfully demonstrated a novel approach to integrating high-fidelity simulations with RL algorithms for autonomous drone control. By addressing the areas for future work, researchers and developers can further enhance the realism, efficiency, and applicability of this framework. The continued advancements in this field hold great promise for the development of sophisticated autonomous systems capable of performing complex tasks in diverse and dynamic environments.

References

[Abe20] Adrian Abedon. Autonomous flight control system for drones to reach targets. 2020.

[HTL+18] Chia Yu Ho, Shau Yin Tseng, Chin Feng Lai, Ming Shi Wang, and Ching Ju Chen. A parameter sharing method for reinforcement learning model between airsim and uavs. In *2018 1st International Cognitive Cities Conference (IC3)*, pages 20–23. IEEE, 2018.

[Jir] Jakub Jirkal. Drone simulation using unreal engine.

[LMC21] Thomas Lee, Susan Mckeever, and Jane Courtney. Flying free: A research overview of deep learning in drone navigation autonomy. *Drones*, 5(2):52, 2021.

[MGV+20] Ratnesh Madaan, Nicholas Gyde, Sai Vemprala, Matthew Brown, Keiko Nagami, Tim Taubner, Eric Cristofalo, Davide Scaramuzza, Mac Schwager, and Ashish Kapoor. Airsim drone racing lab. In Hugo Jair Escalante and Raia Hadsell, editors, *Proceedings of the NeurIPS 2019 Competition and Demonstration Track*, volume 123 of *Proceedings of Machine Learning Research*, pages 177–191. PMLR, 08–14 Dec 2020.

[PLFSN18] Huy X Pham, Hung M La, David Feil-Seifer, and Luan V Nguyen. Autonomous uav navigation using reinforcement learning. *arXiv preprint arXiv:1801.05086*, 2018.

[SDLK18] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and Service Robotics: Results of the 11th International Conference*, pages 621–635. Springer, 2018.

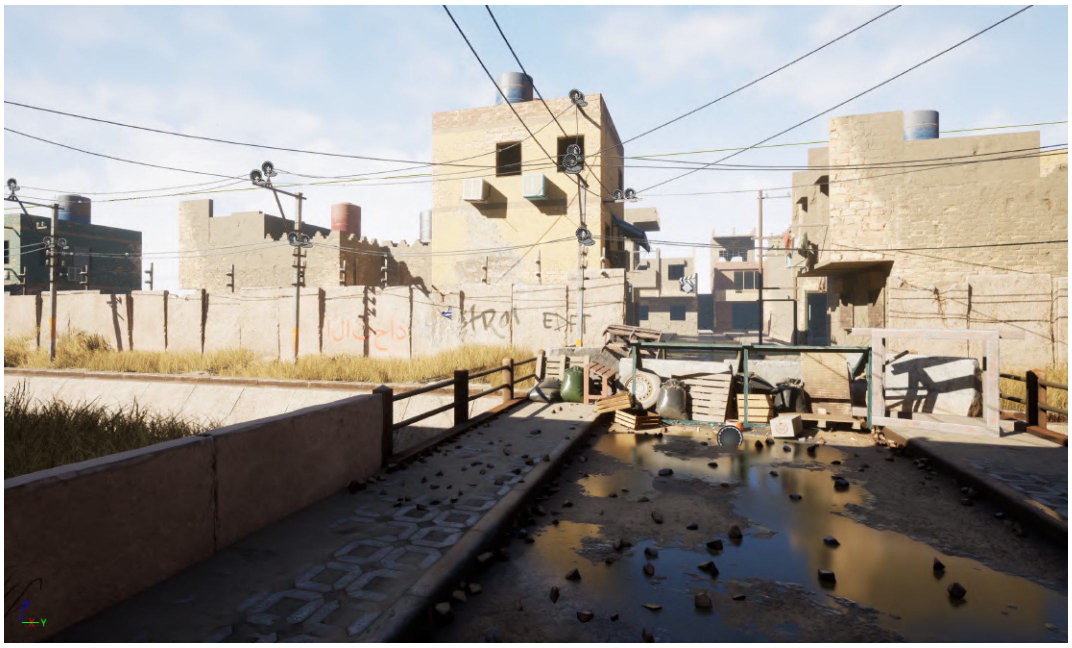


Figure 13: A street in FPV during a simulation

[SKK19] Sang-Yun Shin, Yong-Won Kang, and Yong-Guk Kim. Automatic drone navigation in realistic 3d landscapes using deep reinforcement learning. In *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)*, pages 1072–1077. IEEE, 2019.

[VMGMRA21] David Villota Miranda, Montserrat Gil Martínez, and Javier Rico-Azagra. A3c for drone autonomous driving using airsim. In *XLII Jornadas de Automática*, pages 203–209. Universidade da Coruña, Servizo de Publicacións, 2021.

[YZP$^+$19] Xinglong Yang, Danping Zou, Ling Pei, Daniele Sartori, and Wenxian Yu. An efficient simulation platform for testing and validating autonomous navigation algorithms for multi-rotor uavs based on unreal engine. In *China Satellite Navigation Conference (CSNC) 2019 Proceedings: Volume II*, pages 527–539. Springer, 2019.

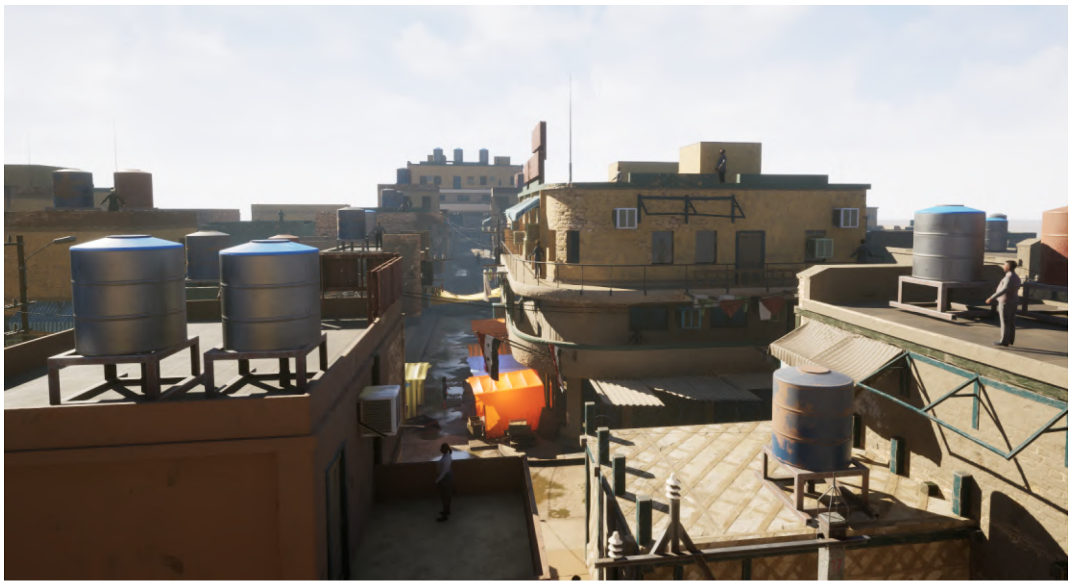


Figure 14: A street in FPV during a simulation with visible people