POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE
(DEI) MASTER'S DEGREE COURSE IN AUTOMATION ENGINEERING

Robotics 2st Module – Mobile Robotics

"Simulation and recording of urban-like environments with a drone in Carla: an integration of ROS1 and Docker containers"

Professors:

Prof. Eng. Luca De Cicco

Eng. Walter Brescia

Students:

Savino Francesco

Savino Tommaso

Summary

Introduction 3
Technologies and tools used 4
Carla Simulator 5
ROS (Robot Operating System) 6
1. Environment setup 7
1.1 CARLA Simulator Container 7
1.2 ROS-bridge and quad_sim_python Container 7
1.2.1 Changes to ROS2 files 10
2. Workflow overview 13
2.1 Starting Carla Simulator 13
2.2 Launching the Carla-ROS-bridge container 15
2.3 Spawning actors 19
2.3.1 The flying_sensor_full.json file 19
2.3.2 The Carla_spawn_objects.py file 20
2.4 Traffic manager 24
2.5 Capturing data with the drone's camera 26
2.5.1 Rviz and quadcopter positioning 26
2.5.2 Capturing images and videos 27
3. Simulations 30
3.1 Changing weather conditions 30
3.2 Simulations performed 36
4. Conclusions and future implementations 45
References 47

List of figures

Figure 1: Carla Simulator logo 5
Figure 2: Carla Town01 map 17
Figure 3: Carla Town03 map 18
Figure 4: Town03 details 18
Figure 5: Example of an image captured by the drone's RGB camera 27
Figure 6: Example of changed weather conditions 31
Figure 7: Examples of clear weather conditions: Sunrise, Noon, Sunset, Night 32
Figure 8: Examples of foggy weather conditions: Sunrise, Noon, Sunset, Night 33
Figure 9: Examples of cloudy weather conditions: Sunrise, Noon, Sunset, Night 33
Figure 10: Examples of light rain weather conditions: Sunrise, Noon, Sunset, Night 34
Figure 11: Examples of medium rain weather conditions: Sunrise, Noon, Sunset, Night 35
Figure 12: Examples of heavy rain weather conditions: Sunrise, Noon, Sunset, Night 35
Figure 13: First simulation - Clear, sunrise 37
Figure 14: First simulation – Cloudy, sunset 38
Figure 15: First simulation – Clear, night 39
Figure 16: Second simulation - Foggy, noon 40
Figure 17: Second simulation - Foggy, night 41
Figure 18: Second simulation - Light rain, sunrise 41
Figure 19: Third simulation - Medium rain, sunset 43
Figure 20: Third simulation - Heavy rain, noon 43
Figure 21: Third simulation - Heavy rain, night 44

Introduction

The following project is based on the use of the Carla simulator to capture images and video through a drone in an urban environment. Carla is a realistic simulator, capable of representing complex scenarios such as city intersections populated by vehicles and pedestrians. The context is that the drone, equipped with a downward-facing camera, is used to capture images and videos of these scenes from different heights and positions.

One of the main objectives of the project is to obtain repeatable recordings, i.e. to perform acquisitions in which the simulation, by setting a fixed seed, ensures that the scene remains constant in terms of vehicles and pedestrians present. The project consists of several steps, including the configuration of a simulation environment, then the development of Python scripts to populate the urban environment and traffic control, and a script to guarantee the recording of images and videos.

The work was supported using Docker containers, facilitating the management of the simulator and ROS (Robot Operating System), which was integrated for the recording and management of topics and nodes.

Technologies and tools used

In this project, we leveraged several key technologies and tools to achieve the objectives of simulation, image acquisition, and processing with the drone in Carla.

- *GitHub repositories*: We utilized both public and personal GitHub repositories. Public repositories, such as *ros2_quad_sim_python* [1] and *carla-ros* [2], provided essential directories and code for the initial implementation. These repositories contained useful resources to help set up Carla and ROS integration. As the project evolved, we saved our updated code versions in the personal repositories, ensuring proper version control and backup of our work. This approach allowed us to easily collaborate and track progress over time.
- *SSH and Windows Remote Desktop*: To access and work on the remote server, we employed SSH (Secure Shell) for command-line access and Windows Remote Desktop for a graphical interface. These tools enabled us to work remotely on the server, where the heavy computation related to the Carla simulation was performed. This setup provided the flexibility to handle the resource-intensive tasks of the project without relying on local machines.
- *Carla Simulator*: The Carla Simulator [3] was the core environment for modeling urban-like scenarios. Built on Unreal Engine, it provides a realistic and dynamic setting for traffic simulation, enabling the spawning of vehicles, pedestrians, and drones equipped with cameras. Carla's modular architecture allows for seamless integration with custom scripts through Python APIs, making it an ideal tool for capturing high-quality data and replicating real-world conditions.
- *ROS1 and ROS-bridge*: To facilitate communication between Carla and the drone's control system, we employed ROS1 (Robot Operating System) alongside the Carla ROS-bridge. This integration allowed us to publish and subscribe to topics, manage sensor data, and control the drone's movements within the simulation. The various ROS packages streamlined our workflow, especially for handling images from the drone's cameras and ensuring efficient control of simulation elements.

Carla Simulator

The Carla Simulator played a fundamental role in creating a realistic urban-like environment for simulating dynamic scenarios, such as the movement of vehicles and pedestrians in city streets. Designed primarily for research in autonomous driving, Carla offers a modular and flexible platform that addresses a wide range of tasks related to autonomous driving, aiming to democratize research and development in the field.

Figure 1: Carla Simulator logo

Carla employs the ASAM OpenDRIVE standard to define roads and urban environments, providing a comprehensive framework for testing and training autonomous driving systems. Its scalable client-server architecture enhances performance: the server manages core simulation processes—like rendering sensor data, computing physics, and updating actor states—while the client uses the Carla API (available in both Python and C++) to control actor logic and set world conditions.

Key features of Carla include a built-in Traffic Manager, which autonomously manages vehicles in the environment, simulating realistic traffic behaviors essential for urban-like scenarios. Additionally, Carla supports a wide array of sensors, including cameras, radars, and lidars, which can be attached to actors to gather critical data for perception and control applications in autonomous driving research.

The simulator also features a Recorder tool that allows users to reenact simulations step by step, making it easier to analyze actor behavior. Integration with external systems like ROS (via the ros-bridge) enhances its functionality, allowing users to leverage ROS tools within the Carla environment. Furthermore, Carla's open asset library provides customizable maps and the ability to simulate various weather conditions, adding to its versatility for advanced use cases.

ROS (Robot Operating System)

In this project, ROS served as a critical framework for facilitating communication between various components of the simulation and controlling the drone. As a structured communication layer, ROS enables seamless interaction among different software modules through the use of topics. This modular architecture allowed us to easily publish and subscribe to relevant data streams, such as camera images captured by the drone's RGB camera.

By leveraging ROS1, we efficiently managed the acquisition of images during the simulation. The drone's camera published its data on specific ROS topics, enabling us to subscribe to these streams and process the images in real time. This capability was particularly crucial for our objective of collecting and storing images for subsequent analysis, as it facilitated smooth integration with various ROS-based tools and libraries.

Furthermore, ROS played a key role in integrating the *carla_ros_bridge*, which established communication between Carla and ROS. This integration allowed us to dynamically control the drone's movements and interact with the simulated environment. This setup not only enhanced the flexibility of our project but also provided a robust platform for developing and testing algorithms related to image processing, autonomous navigation, and environmental perception.

It is important to note that while our project utilized ROS1, some elements, such as the ros2_quad_sim_python repository, were designed for ROS2. In the following sections of this report, we will discuss the modifications made to adapt these elements for compatibility with ROS1.

1. Environment setup

A fundamental aspect of this project was creating a flexible and efficient simulation environment capable of running the Carla simulator and handling the ROS1 integration that could support the interactions within CARLA while managing sensor data, control commands, and real-time feedback. To achieve this, we designed an architecture that leveraged two distinct Docker containers, each playing a critical role in ensuring smooth integration between the various components.

1.1 CARLA Simulator Container

The first container hosts the CARLA simulator. For our project, we used CARLA version 0.9.13, which provided a detailed and dynamic simulation of urban environments, complete with moving vehicles, pedestrians, and environmental variability.
The container running CARLA was built using a public GitHub repository [4], which provided a pre-configured setup that allowed us to run CARLA in a headless mode on a remote server, efficiently handling the heavy computational demands without burdening local machines. CARLA's client-server architecture also made it easier to manage the simulation and interact with the virtual world through external scripts and APIs.

1.2 ROS-bridge and quad_sim_python Container

The second container was specifically designed to handle the communication with the first container. In fact, this ROS package is a bridge that enables two-way communication between ROS and CARLA. The information from the CARLA server is translated to ROS topics. In the same way, the messages sent between nodes in ROS get translated to commands to be applied in CARLA.

To achieve this, we needed to integrate the CARLA ROS-bridge and quad_sim_python packages, which facilitated the publishing and subscribing of topics between the simulated drone and CARLA's environment.

The ROS1-based container was built from a custom *Dockerfile* which defined the environment, dependencies, and tools required for the simulation. Initially, the setup relied on public repositories, such as ros2_quad_sim_python and carla-ros. However, since these repositories were designed for ROS2, we had to adapt them

to ROS1, the version we chose for this project due to compatibility reasons and pre-existing workflows. These modifications involved refactoring sections of the codebase to ensure that topics, nodes, and services adhered to ROS1 standards.

The Dockerfile for this container was hosted in a personal GitHub repository [5], and it contained the necessary instructions to automate the setup.

The container is based on the `osrf/ros:noetic-desktop-full` image, which provides the full installation of ROS Noetic (ROS1). Below is a breakdown of the Dockerfile used to build this container:

- **Base image and initial setup:** The container is built from the official ROS Noetic desktop-full image, which already includes many essential ROS tools. From here, we escalate privileges to root for package installation purposes:

```
FROM osrf/ros:noetic-desktop-full USER root
```

- **Installing basic dependencies:** We begin by updating the package lists and installing essential tools such as python3-pip, git, and wget. Additionally, we install python3-rosdep, which helps in resolving ROS dependencies later. We also install the basic networking and debugging tools like net-tools and x11-apps, which are useful for visualization with RViz.

```
RUN apt-get update && apt-get upgrade -y && apt-get install -y \
      python3-pip \
      python3-rosdep \
      git \
      wget \
      nano
RUN apt install -y net-tools iputils-ping x11-apps
```

- **Setting up the workspace:** We then create the project directory structure within the container.

```
RUN mkdir -p /home/gruppo2/Progetto-MR-
Savinox2/carlaLab/catkin_ws/src
WORKDIR /home/gruppo2/Progetto-MR-Savinox2/carlaLab/catkin_ws
```

- **Cloning required repositories:** At this stage, we clone the ROS-bridge and quadcopter simulation packages. Initially, public repositories were used (carla-ros and ros2_quad_sim_python), but these were forked and modified for

compatibility with ROS1 and Carla 0.9.13. These modified repositories are cloned from our own GitHub links:

```
RUN git clone https://github.com/FrankSav80/provax.git src/ros-
bridge
RUN git clone https://github.com/FrankSav80/provay.git
src/quad_sim_python
```

- **Installing python requirements:** Next, we install the Python dependencies required for CARLA and the custom ROS packages. The specific version of CARLA used is 0.9.13, and we use pip to install any additional Python libraries needed for both ROS-bridge and the drone control package (quad_sim_python).

```
RUN python3 -m pip install --upgrade pip
RUN python3 -m pip install carla==0.9.13
RUN python3 -m pip install -r src/ros-bridge/requirements.txt
RUN python3 -m pip install quad_sim_python
```

- **ROS dependencies and workspace compilation:** With the code and dependencies in place, we set up ROS dependencies using rosdep, ensuring that any external ROS packages required by the project are installed. After that, we compile the ROS workspace using catkin_make:

```
RUN if [! -f /etc/ros/rosdep/sources.list.d/20-default.list]; then
rosdep init; fi && \
    rosdep update && \
    rosdep install --from-paths src --ignore-src -r -y
RUN /bin/bash -c 'source /opt/ros/noetic/setup.bash; catkin_make
clean; catkin_make'
```

- **Environment configuration:** To simplify the development process, we configure the container's environment to automatically source ROS and CARLA settings on every new session. This involves adding relevant lines to the *".bashrc"* file, which ensures the necessary environment variables are set up whenever a terminal session is opened.

```
RUN echo "source /opt/ros/noetic/setup.bash" >>
/home/gruppo2/.bashrc
RUN echo "source /home/gruppo2/Progetto-MR-
Savinox2/carlaLab/catkin_ws/devel/setup.bash" >>
/home/gruppo2/.bashrc
RUN echo "export
PYTHONPATH=\$PYTHONPATH:/home/gruppo2/.local/lib/python3.8/site-
packages" >> /home/gruppo2/.bashrc
RUN echo "export PATH=\$PATH:/home/gruppo2/.local/bin" >>
/home/gruppo2/.bashrc
```

- **Finalizing the docker container:** Finally, the working directory is set to the catkin workspace, and the default command for the container is set to open a bash session, ready for interaction.

```
WORKDIR /home/gruppo2/Progetto-MR-Savinox2/carlaLab/catkin_ws
CMD ["bash"]
```

In the upcoming sections, we will explore the modifications applied to the public repositories to adapt them for compatibility with ROS1.

1.2.1 Changes to ROS2 files

As previously mentioned, the original repository [1] for the quadcopter simulation includes ROS2 packages. To avoid any issues with using these packages, we decided to modify all the relevant files to ensure compatibility with ROS1, even though not all of them were ultimately used.

There are three subdirectories in the *src* directory: *quad_sim_python_msgs*, *rclpy_param_helper*, *ros2_quad_sim_python*.

Changes have been made to the CMakeLists.txt file for the *quad_sim_python_msgs* and *ros2_quad_sim_python* directories, mainly because ROS2 uses the ament build system while ROS1 is based on catkin. One notable difference is how packages and dependencies are handled: in ROS2, the directive find_package(ament_cmake REQUIRED) is used, which is typical of the ament system, while in ROS1, this is replaced by find_package(catkin REQUIRED COMPONENTS ...).

Message generation is another key aspect. In ROS2, it is managed through the function rosidl_generate_interfaces, which allows for automatic generation of interfaces for the messages defined in the .msg files. In transitioning to ROS1, this function had to be replaced with add_message_files and generate_messages, which follow a similar process but use different syntax and are integrated into the catkin workflow. Additionally, the declaration of message dependencies, such as std_msgs, shifts from DEPENDENCIES in ROS2 to CATKIN_DEPENDS in ROS1.

In summary, converting from ROS2 to ROS1 involved removing all directives related to the ament system and its dependency management, replacing them with the equivalent catkin functions.

The *package.xml* file also underwent changes, as the build system differs significantly between ROS2 and ROS1.

Changes have also been made to the launch files in the ros2_quad_sim_python directory and the python files linked to them. Initially, the overall structure of the launch files underwent a significant change: ROS2 uses a dynamic Python file format, while ROS1 adopts a static XML format. This transition required a thorough review of how arguments and nodes are declared, as the functionalities available in ROS2 are not directly transferable to ROS1. For example, in ROS2, functions such as DeclareLaunchArgument and Node are used to simplify the definition of arguments and nodes. In ROS1, however, we had to rely on the <arg> tag to declare arguments and use the <node> tag for nodes.

The logic for defining the quadcopter parameters in ROS1 is more complex. In ROS2, parameters were handled more directly, whereas in ROS1 we had to specify each parameter using separate arguments within the launch file, increasing redundancy.

Another difference is in the inclusion of other launch files. In ROS2, the *IncludeLaunchDescription* feature allows for the modular and manageable incorporation of launch files, while in ROS1, inclusion occurs through the *<include>* tag.

In the python files, such as *ros_quad_ctrl.py*, *ros_quad_sim.py* and *ros_quad_sim_and_ctrl.py,* the reference libraries and their handling have been changed *(*e.g. if in ROS2 you use *rclpy.spin()* to keep the node active, in ROS1 you use *rospy.spin()*).

Regarding the *carla-ros-bridge* package, it provides support for both ROS1 and ROS2 through separate implementations that share a common interface. This design allows for flexibility depending on the version of ROS being used. To run the ROS bridge, the only requirement is to set up the appropriate ROS environment, which, in our case, was efficiently managed using a Dockerfile. The Dockerfile automates the environment setup, ensuring that all necessary dependencies are installed, making it straightforward to launch the bridge and seamlessly connect Carla with ROS.

It is important to note that some files within the various carla-ros-bridge packages had to be modified to meet the specific needs of our project. These changes will be explained in detail in the following sections.

2. Workflow overview

In this section, we will walk through the complete workflow followed to achieve the project's objectives, from initializing the Carla simulator to integrating ROS functionalities via the carla-ros-bridge. Each step is described in detail, outlining the commands and configurations used to set up the simulation environment and the control systems. Additionally, any modifications, updates, or custom scripts that were required to tailor the system to our needs will be discussed.

The workflow is structured to reflect the chronological order of operations, beginning with the setup of the Carla environment, followed by launching the container that contains the necessary ROS packages, and ending with the control and data collection processes.

2.1 Starting Carla Simulator

To begin the simulation, we use a pre-built Docker image for Carla version 0.9.13, configured to run in headless mode. The image, named *ricardodeazambuja/carlasim:0.9.13_headless*, is stored on the server and takes up approximately 16.6GB of space. This specific version and setup were chosen for its compatibility with our system, allowing us to execute Carla without the graphical interface, reducing the computational load.

To launch the Carla simulator in headless mode, the following Docker command is executed in the first terminal:

```
docker run --rm -it \
--name carla-container \
--hostname=carla-container \
--user carla \
-p 2004-2006:2000-2002 \
--gpus 0 \
ricardodeazambuja/carlasim:0.9.13_headless ./launch_headless.sh
```

This command initiates the container, maps the required ports (2000-2002 for Carla's communication, and 2004-2006 for additional services if needed), and runs it using the available GPU. The `--rm` flag ensures that the container is removed after execution, keeping the environment clean.

The script `launch_headless.sh`, executed within the container, contains the following instructions:

```
#!/bin/sh

# echo "Starting avahi..."
# sudo avahi-daemon -D

unset SDL_VIDEODRIVER

echo "Starting simulator..."
./CarlaUE4.sh -vulkan -RenderOffscreen -nosound &

echo "Press ENTER to kill it!"
echo
read USELESS

echo
echo "Killing simulator..."
echo

pkill Carla
```

This script performs the following actions:

1. It clears the SDL_VIDEODRIVER variable to disable any unnecessary video driver settings required to run in headless mode.
2. It starts the Carla simulator using the `CarlaUE4.sh` script with the `-vulkan`, `-RenderOffscreen`, and `-nosound` flags. These flags ensure that Vulkan is used as the rendering engine, with offscreen rendering and no audio output, optimizing performance for a non-graphical environment.
3. It waits for user input to terminate the simulation. Once the Enter key is pressed, the script gracefully shuts down the simulator using the `pkill` command.

This setup ensures that Carla runs efficiently in a resource-constrained environment, allowing the subsequent steps in the workflow to interact with the simulation through the networked ROS-bridge setup.

2.2 Launching the Carla-ROS-bridge container

After successfully starting the Carla simulator in the first terminal, the next step is to establish a connection between Carla and ROS using the Carla-ROS bridge. This requires launching another Docker container dedicated to managing ROS communication. This container will also be used later in multiple terminals to execute various scripts for spawning vehicles and pedestrians, managing the traffic, controlling the drone, and more.

To allow the Carla-ROS bridge container to connect to the Carla simulator, we first need to find the IP address of the Carla container. This is done by opening a new terminal and executing the following command:

```
docker inspect carla-container | grep -A 10 "Networks"
```

Once we have the IP of the Carla container, we proceed by launching the Carla-ROS bridge container. In the same terminal, we execute the following Docker command:

```
docker run --rm -it --privileged --net host --
device=/dev/dri:/dev/dri -v /tmp/.X11-unix:/tmp/.X11-unix -e
DISPLAY=$DISPLAY -v $HOME/.Xauthority:/home/$(id -un)/.Xauthority -
e XAUTHORITY=/home/$(id -un)/.Xauthority carlav2:gruppo2new
/bin/bash
```

This command does the following:

- *--privileged*: Allows the container to access certain devices and capabilities, such as GPU resources.
- *--net host*: Ensures the container uses the host's network configuration, facilitating communication with the Carla container.
- *--device=/dev/dri:/dev/dri*: Grants access to the host's display rendering interface for graphical rendering, if needed.
- *-v /tmp/.X11-unix:/tmp/.X11-unix*: Mounts the X11 socket for GUI applications.
- *-e DISPLAY=$DISPLAY and -e XAUTHORITY=/home/$(id -un)/.Xauthority*: Ensures the container can use the host's display settings for GUI-related tasks.

Once inside the container, we set up the environment for ROS by sourcing the necessary setup files:

```
source /opt/ros/noetic/setup.bash
source devel/setup.bash
```

We then edit the `/etc/hosts` file inside the ROS bridge container to associate the IP address of the Carla container with a hostname. This ensures that when the ROS bridge attempts to communicate with Carla, it can resolve the hostname to the correct IP address:

```
echo "172.17.0.2 carla-container.local" >> /etc/hosts
```

With the ROS environment ready and the containers properly connected, we navigate to the directory containing the ROS launch file for the Carla-ROS bridge:

```
cd src/ros-bridge/carla_ros_bridge/launch/
```

Finally, we execute the ROS launch file to start the Carla-ROS bridge:

```
roslaunch carla_ros_bridge.launch
```

The launch file, `carla_ros_bridge.launch` [6], contains several parameters and configurations. In fact, this file starts the ROS node 'carla_ros_bridge', which is responsible for creating and maintaining the connection between CARLA and ROS and executes the 'bridge.py' script [7].

In 'bridge.py', the class 'CarlaRosBridge' is defined, which is mainly characterised by the following functions:

- Two-way communication: The class acts as a bridge between CARLA and ROS, allowing both to receive commands from ROS (such as the generation of new actors) and to send information from the simulator to ROS (e.g. the status of vehicles and environmental conditions).

- Synchronous mode support: If the simulator is configured in synchronous mode (as is the case here), the bridge controls the world update cycle in a coordinated manner with ROS, also waiting for commands from ego vehicles (user-controlled vehicles) before proceeding with the next update. The

synchronous mode synchronises the simulation time with ROS to keep both systems in sync and adds a timeout to avoid connection delays. The *fixed_delta_seconds* has been set to 0.05 seconds; this sets the time step for the simulation [8].

- ROS services and topics: The bridge exposes various services and topics to interact with the simulator, making it possible to dynamically create actors, the modification of weather conditions, and the collection of real-time status data.

We can also choose which map to load into Carla. For our initial setup and environment configuration, we decided to use the default map, *"Town01"* [9]. This is a small town with numerous T-junctions and a variety of buildings, surrounded by coniferous trees and featuring several small bridges spanning across a river that divides the town into two halves.

Figure 2: Carla Town01 map

However, for the simulations, we also use the map *"Town03"*:

- Town 3 [10] is a larger town with features of a downtown urban area. The map includes some interesting road network features such as a central roundabout, a residential cul-de-sac (death-end residential road) underpasses and overpasses, along with numerous 4-way junctions and T-

junctions. The town also includes a raised metro track and a large building under construction.

Figure 3: Carla Town03 map

Figure 4: Town03 details

2.3 Spawning actors

At this point, the Carla-ROS bridge is successfully running, and the environment is ready to proceed with additional tasks, such as spawning vehicles and pedestrians.

We then open a new terminal where we enter the same container:

```
docker exec -it nome_container bash
```

Once inside the container, you can execute the carla_spawn_objects.launch [11] file with the specified object definition file using the following command:

```
roslaunch carla_spawn_objects.launch
objects_definition_file:=/home/gruppo2/Progetto-MR-
Savinox2/carlaLab/catkin_ws/src/quad_sim_python/src/ros2_quad_sim_p
ython/cfg/flying_sensor_full.json
```

2.3.1 The flying_sensor_full.json file

The flying_sensor_full.json file [12] defines a set of objects and sensors that are used in the Carla simulation. The primary purpose of this file is to configure a flying sensor (representing a drone) equipped with various sensors to capture environmental data during the simulation.

This configuration file (objects_definition_file) is read by the *Carla_spawn_objects.py* script, which extracts the specified object definitions and sends spawn requests via the ROS service /carla/spawn_object. The script attempts to obtain the spawn point from three sources: the launch file, the configuration file or, as a last resort, a random point via the spawn service.

The file specifies three virtual sensors:

- *sensor.pseudo.objects*: collects information about all objects in the simulation environment.
- *sensor.pseudo.actor_list*: gathers data on all actors present in the simulation, such as vehicles and pedestrians.
- *sensor.pseudo.markers*: captures markers used to visualize or identify specific elements in the simulation.

The central element of this setup is the *flying_sensor*, which simulates a drone positioned 2.4 meters above the ground with an initial orientation of zero roll, pitch, and yaw. Attached to this drone are several important sensors:

- *GNSS (Global Navigation Satellite System)*: provides accurate geographical coordinates (latitude, longitude, altitude) without any noise, simulating GPS functionality.
- *Depth cameras*: Two depth cameras are mounted on the drone, one facing forward and one downward. These capture depth images of the environment, which can be used to analyse distances to objects or features in the scene.
- *Semantic segmentation cameras*: These cameras classify objects in the environment based on semantic categories (e.g., vehicles, pedestrians, buildings). Like the depth cameras, they are positioned both forward and downward to cover different perspectives.
- *RGB cameras*: These standard colour cameras capture high-quality RGB images of the environment, again positioned both forward and downward to provide a comprehensive view.
- *Actor control*: A pseudo actor that allows the drone to be controlled programmatically within the simulation.

Each camera has a field of view of 73 degrees and a resolution of $640x480\ pixels$, ensuring that the drone captures detailed visual data from multiple angles. This configuration is designed to simulate a multi-modal sensor system for a drone, providing a rich set of data (depth, segmentation, and RGB images) for a variety of tasks, such as mapping, object detection, and tracking.

2.3.2 The Carla_spawn_objects.py file

The next step involves spawning vehicles and pedestrians into the Carla environment. This is achieved using a custom Python script, carla_spawn_objects.py [13], which has been adapted from the original version found in the carla-ros package.

The most notable improvements include the use of service calls from *carla_msgs.srv* for actor management, the introduction of a random seed for repeatability, and robust mechanisms for handling blueprints and spawn points.

As already mentioned, a crucial addition is the integration of services from the carla_msgs.srv module, which provides essential functionalities for interacting with Carla. The imported services include:

```
from carla_msgs.srv import SpawnObject, DestroyObject, GetBlueprints,
SpawnPoints
```

Where:

- SpawnObject: Responsible for spawning vehicles and pedestrians in the simulation.
- DestroyObject: Allows for clean removal of spawned actors.
- GetBlueprints: Retrieves available blueprints [14] for vehicles and pedestrians.
- SpawnPoints: Obtains valid locations within the simulation where actors can be spawned.

One of the major additions is the introduction of a *random seed* to ensure the repeatability of actor spawning. This is particularly important in simulations to reproduce results and test specific scenarios.

By setting a seed value (42 in our case) for the random number generator, the simulation guarantees that the same vehicles and pedestrians will be spawned in the same locations and orientations. The seed is set using the `random.seed()` function.

The core of the modifications lies in the `spawn_actors_with_blueprints()` function. Using the *GetBlueprints* service, the function retrieves available blueprints for vehicles and pedestrians and filters them accordingly.

```
# Example for vehicles
get_blueprints_service =
rospy.ServiceProxy('/carla/get_blueprints', GetBlueprints)

vehicle_request = GetBlueprints()
vehicle_request.filter = "vehicle.*"

vehicle_response = get_blueprints_service(vehicle_request.filter)
```

By doing so, it dynamically adapts to the simulation's available actors, which can include different types of vehicles (e.g., cars, trucks, motorcycles) and pedestrians.

Additionally, the code prompts for user input regarding the number of actors to spawn in the simulation. Specifically, it retrieves the desired number of vehicles and pedestrians using ROS parameters. By default, the system is set to spawn five vehicles and ten pedestrians, but this can be adjusted based on the specific requirements of the simulation.

```
# Ask for input on the choice of actors to spawn
num_vehicles = max(0, rospy.get_param('~num_vehicles', 5))
num_walkers = max(0, rospy.get_param('~num_walkers', 10))
```

This flexibility allows users to tailor the simulation environment to their needs, enhancing its applicability for various testing scenarios. For example, when starting up the node:

```
roslaunch carla_spawn_objects.launch objects_[…]ll.json
num_vehicles:=50 num_walkers:=20
```

The `get_spawn_points()` function has been integrated to retrieve valid spawn positions via the *SpawnPoints* service. If no spawn points are available, the function handles the situation by recording a warning and returning None, ensuring that the simulation does not proceed without valid data. A list is created with all available spawn points.

```
# “Spawnpoint service” request
get_spawn_points_service =
rospy.ServiceProxy('/carla/get_spawn_points', SpawnPoints)


# Spawnpoint list
spawn_points = self.get_spawn_points()
```

The `generate_random_pose` function was introduced to improve the process of spawning actors. This function works in tandem with the seed mechanism, ensuring that the generation of random poses can be deterministic when necessary. Initially, it retrieves valid spawn points using the *SpawnPoints* service, taking care to avoid previously used positions to maintain variety within the simulation. However, it also retains the flexibility to reuse spawn points if necessary.

When generating poses for actors, the function introduces an element of randomness, but the fixed seed ensures that this randomness remains predictable and repeatable across different simulations. The function effectively generates valid pose configurations that include both the positions and orientations of the actors, thus ensuring their correct positioning within the environment.

The script introduces two important functions for managing actors in the simulation:

- `spawn_actor()`: This function takes the selected blueprint and a generated pose and uses the *SpawnObject* service to spawn an actor in the simulation. It logs detailed information about the spawned actors and ensures proper tracking through the *spawned_actors* list. In case of failure, it logs appropriate warnings without halting the simulation.

```
# “Spawn service” request
spawn_service = rospy.ServiceProxy("/carla/spawn_object",
SpawnObject)
```

- `destroy_actors()`: This function handles the clean removal of all previously spawned actors using the *DestroyObject* service. It iterates through the *spawned_actors* list, ensuring that each actor is properly destroyed and freeing up resources in the simulation.

```
# “Destroy service” request
destroy_service = rospy.ServiceProxy('/carla/destroy_object',
DestroyObject)
```

2.4 Traffic manager

Having successfully implemented the mechanisms for spawning vehicles and pedestrians, it is essential to address the management of traffic within the simulation environment. The Traffic Manager serves as a crucial component in this context, providing the necessary functionality to control and coordinate the movements of spawned actors. By leveraging the Traffic Manager, we can simulate realistic traffic scenarios that not only incorporate the presence of vehicles and pedestrians but also facilitate their interactions in a manner that closely resembles real-world dynamics.

The Traffic Manager integration was realised through two key files: a launch file called *traffic_manager.launch* [15] and a Python script called *traffic_manager.py* [16]. The *traffic_manager.launch* file is used as an entry point to launch the Traffic Manager within the ROS ecosystem.

The launch file starts the ROS node associated with the Traffic Manager using the *carla_spawn_objects* package, which includes the Python script. This Python script contains the main logic for traffic management, facilitating the control of actors already present in the simulated world.

To launch the Traffic Manager, a new terminal must be opened and entered the same container where *ros-bridge* and *spawn-objects* have already been launched. We use the `roslaunch` command to execute the launch file, thus starting the node responsible for traffic management. This approach allows the different processes to be kept separate, ensuring more efficient management of the system, without interfering with other components such as the actor spawn or *ros-bridge*.

The script is designed to interact directly with Carla's simulator. Once the connection with the simulated world is established the script initialises the Traffic Manager and sets a seed to ensure repeatability of the simulations:

```
# Connection to Carla's world
self.client = carla.Client('carla-container.local', 2000)
self.world = self.client.get_world()

# Initialising the Traffic Manager
self.traffic_manager = self.client.get_trafficmanager(9000)

# Seed for repeatability
self.traffic_manager.set_random_device_seed(seed_value)
random.seed(seed_value)
```

This seed ensures that vehicles and pedestrians behave consistently between different runs of the simulation, maintaining the same behaviour.

The script manages two types of actors: vehicles and pedestrians. In the case of vehicles, these are connected to the Traffic Manager, which enables the autopilot for each of them, ensuring that they follow traffic rules, such as obeying traffic lights, and configuring other specific properties, such as speed, distance to leading vehicle and automatic lane changing.

```
# Autopilot and properties
vehicle.set_autopilot(True, self.traffic_manager.get_port())
self.traffic_manager.vehicle_percentage_speed_difference(vehicle,-
20)
self.traffic_manager.ignore_lights_percentage(vehicle, 0)
self.traffic_manager.auto_lane_change(vehicle, True)
```

As for pedestrians, the script creates controllers that allow them to move along predefined routes, also assigning customised maximum speeds. The Traffic Manager ensures that vehicles and pedestrians move in a coordinated manner, thus simulating complex and realistic traffic scenarios.

```
# Spawn of the pedestrian controller
controller_bp =
self.world.get_blueprint_library().find('controller.ai.walker')
controller = self.world.spawn_actor(controller_bp,
carla.Transform(), attach_to=pedestrian)

# Create the controller
controller.start()

# Random location for pedestrians
target_location = self.world.get_random_location_from_navigation()
```

2.5 Capturing data with the drone's camera

To introduce the data capture phase with the drone camera, it is necessary to first describe the steps to start the simulation system and position the drone correctly.

2.5.1 Rviz and quadcopter positioning

This process begins with the use of new terminals to execute commands within the Docker container.

First, a new terminal is opened, and the usual Docker container is accessed. Next, one moves to the project directory relating to the quadcopter simulation, and *Rviz*, the ROS visualisation tool, is launched. Rviz is launched with the following command:

```
cd src/quad_sim_python/src/ros2_quad_sim_python/cfg
rosrun rviz rviz
```

To ensure that the visual configuration is correct and adapted to the use of the drone sensor, a customised configuration file [17] for Rviz is used:

```
rosrun rviz rviz -d rviz_carla_RGBdown.rviz
```

In a second terminal, again by accessing the container, a message is published on the ROS topic controlling the position of the drone. The `rostopic pub` command is used to set the initial position of the drone, typically at a road junction:

```
rostopic pub /carla/flying_sensor/control/set_transform
geometry_msgs/Pose "{position: {x: 94.5, y: -57.5, z: 30.0},
orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}"
```

This command sends a message that positions the drone at a height of 30 metres above the selected intersection, providing a good view of the area to be analysed. The choice of this spawn point allows the drone to monitor traffic and activities at the intersection, collecting useful data from an aerial perspective. Once the drone is positioned, the data collection phase can begin via the *RGB Camera* mounted on the drone, which is used to take photos and record videos. This camera provides detailed, high-resolution images of the scene, allowing precise

visual information on traffic and objects in the simulated environment to be captured.

Below is an image representing what the drone's RGB camera captures as it flies over the intersection:

Figure 5: Example of an image captured by the drone's RGB camera

2.5.2 Capturing images and videos

Once the mechanism for capturing data from the drone's camera has been set up, it is important to manage the capture of images and video in a structured way. The primary tool for this task is the *drone_image_saver* node, which enables the storage of images and video data captured by the RGB camera mounted on the drone.

The implementation of the image and video capture is realised through two key files: a launch file called *drone_image.launch* [18] and a Python script called *drone_image.py* [19]. The launch file defines the necessary parameters for specifying where the images and videos will be stored, as well as the duration of the recordings and the frame rate. You can customise these parameters by passing variables via the associated launch file, such as *record_duration* or *frame_rate*.

```
# Variation in registration time at node startup
roslaunch drone_image.launch record_duration:=20
```

This file initiates the ROS node associated with the *drone_image_saver*, using the *carla_spawn_objects* package, which includes the Python script that contains the main logic for capturing and saving the RGB camera data. This script subscribes to the */carla/flying_sensor/rgb_down/image* topic, where the images from the drone's camera are published, and handles the process of saving these images locally. In addition, it uses the *cv_bridge* package [20] to convert ROS image messages into OpenCV-compatible formats, allowing images to be manipulated and saved in the filesystem.

```
# Starting CvBridge
self.bridge = CvBridge()

[...]

# Image conversion
cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")
```

The script subscribes to two main ROS topics:

- */carla/flying_sensor/rgb_down/image*: the topic that transmits the images captured by the camera mounted on the drone. The callback associated with this topic is called each time an image message is received.

- */toggle_recording*: a topic allowing video recording to be started or stopped via Bool messages. When recording is active, the received images are written into the video. Initially, the *.avi* video extension with XVID encoding was used. This did not allow the video to be displayed correctly. After various trials, a solution was found by saving the file with the *.avi* extension and using the MJPG codec.

```
# Subscription to the RGB Camera topic
rospy.Subscriber('/carla/flying_sensor/rgb_down/image', Image,
self.image_callback)

# Subscription to the topic for registration control
rospy.Subscriber('/toggle_recording', Bool,
self.toggle_video_recording)
```

To start and stop video recording, it is necessary to open a new terminal and access the usual Docker container. Once inside the container, a message must be posted to the topic */toggle_recording* using the following command:

```
rostopic pub /toggle_recording std_msgs/Bool "{data: true}"
```

Once the recording is started and completed, the images and videos captured by the drone are saved within the Docker container. However, it is important to remember that the files saved in the container are temporary: once the container is closed or restarted, all changes to the file system are lost. Therefore, it is essential to copy the files locally to avoid losing them.

To transfer files (images and videos) from the Docker container to the local server, it is necessary to use the `docker cp` command:

```
# Copying the entire directory
docker cp NAME_container:/home/gruppo2/[...]/drone_images

# Copying a single file
docker cp NAME_container:/home/gruppo2/[...]/drone_videos/vid2.avi
```

Once the files have been copied to the server, they can be downloaded to your local machine. This can be done using the `scp` (Secure Copy Protocol) command for secure file transfer.

```
# Copying file in our machine
scp gruppo2@[Ip address]:/home/gruppo2/video_4265.avi .
```

3. Simulations

In the context of autonomous systems simulation, visual scene analysis is one of the key aspects in assessing the vehicle's ability to perceive and recognise the environment. In this chapter, we will focus on the images captured by the drone's RGB camera inside the Carla simulator, exploring how weather conditions affect the quality and visibility of the footage.

Each weather condition brings with it specific difficulties, such as reduced visibility or reflections caused by rain, which can negatively affect the quality of the captured images. In this chapter, we will analyse simulations carried out under different conditions and on various intersections in different virtual cities, using images and videos recorded by the drone camera.

3.1 Changing weather conditions

One of the most useful aspects of the Carla simulator is the ability to dynamically change weather conditions during a simulation.

Carla allows weather conditions to be dynamically modified via the topic */carla/weather_control*, by publishing a message of type *CarlaWeatherParameters.msg*. This message allows us to control various environmental parameters that simulate realistic weather conditions. Using ROS, we can customise various aspects such as cloudiness, rain, wind, fog and sun intensity, creating scenarios that affect the visibility and illumination of the images captured by the drone's sensors.

The CarlaWeatherParameters.msg message includes the following fields:

- *cloudiness*: indicates the percentage of cloud cover.
- *precipitation*: level of precipitation, such as rain or snow.
- *precipitation_deposits*: deposits left by rain, such as puddles.
- *wind_intensity*: intensity of wind in the environment.
- *fog_density*: density of fog, which reduces visibility.
- *fog_distance*: distance at which fog begins.
- *wetness*: indicates the wetting of surfaces, which creates reflections.

- *sun_azimuth_angle*: horizontal angle of the sun, which controls the direction of shadows.
- *sun_altitude_angle*: vertical angle of the sun, which affects the overall brightness.

Below are some advanced parameters that simulate the behaviour of light through particles in the air, such as fog or mist, and can influence the overall lighting of the scene:

- fog_falloff: the rate at which fog density decreases with distance. The higher the value, the faster the fog disappears with distance.
- scattering_intensity: intensity of light scattering, which affects the appearance of the sky and scene.
- mie_scattering_scal: controls the amount of “Mie” scattering (related to larger particles, such as water droplets).
- rayleigh_scattering_scale: controls the amount of ‘Rayleigh’ scattering (related to smaller particles, such as air molecules).

An example of a ROS command to change the weather conditions in Carla is as follows:

```
rostopic pub /carla/weather_controlcarla_msgs/
CarlaWeatherParameters "{cloudiness: 0.0, precipitation: 0.0,
precipitation_deposits: 0.0, wind_intensity: 5.0, fog_density: 0.0,
fog_distance: 1000.0, wetness: 0.0, sun_azimuth_angle: 90.0,
sun_altitude_angle: 5.0}"
```

In this example, we are simulating a clear day with clear skies (no cloudiness) and no precipitation or wetness on the ground. The wind intensity is set to a moderate value, and the angle of the sun simulates a morning light.

Figure 6: Example of changed weather conditions

Below are all the types of weather obtained in Carla:

1. Clear - Sunrise, Noon, Sunset, Night:

Figure 7: Examples of clear weather conditions: Sunrise, Noon, Sunset, Night

2. Foggy - Sunrise, Noon, Sunset, Night:

Figure 8: Examples of foggy weather conditions: Sunrise, Noon, Sunset, Night

3. Cloudy - Sunrise, Noon, Sunset, Night:

Figure 9: Examples of cloudy weather conditions: Sunrise, Noon, Sunset, Night

4. Light Rain - Sunrise, Noon, Sunset, Night:

Figure 10: Examples of light rain weather conditions: Sunrise, Noon, Sunset, Night

5. Medium Rain - Sunrise, Noon, Sunset, Night:

Figure 11: Examples of medium rain weather conditions: Sunrise, Noon, Sunset, Night

6. Heavy Rain - Sunrise, Noon, Sunset, Night:

Figure 12: Examples of heavy rain weather conditions: Sunrise, Noon, Sunset, Night

Each image will depict a snapshot of the weather conditions obtained during the simulation. Visibility and lighting conditions will vary significantly depending on the time of day (Sunrise, Noon, Sunset, and Night) and the weather setup (Clear, Foggy, Cloudy, Rainy).

Additionally, the Excel file [21] contains the exact settings for each simulated weather condition, allowing easy reproduction of the various scenarios via the listed ROS commands. This file serves as a guide for replicating the conditions shown in the report's images.

3.2 Simulations performed

In this section we present the results of simulations conducted with the drone under a range of weather conditions in the Carla simulator. By adjusting the main environmental parameters such as cloudiness, fog, precipitation and illumination at different times of the day, we attempted to evaluate how these factors affect the visual data acquired by the drone. Images and videos were recorded to assess the impact of atmospheric conditions on visibility and image quality, particularly in difficult conditions such as fog, heavy rain and low illumination.

In addition, simulations were conducted to assess the repeatability of the environment. By keeping weather parameters consistent between runs, we wanted to ensure that the conditions of each scenario could be reliably reproduced, allowing for a fair comparison of results between different types of weather.

As already mentioned, to evaluate the performance of our system on different urban scenarios, we conducted simulations in Carla Simulator's Town03 map. To switch from the Town01 map to Town03, we ran the command:

```
roslaunch carla_ros_bridge carla_ros_bridge.launch town:=Town03
timeout:=5
```

Due to the heaviness of the map, we increased the timeout to ensure a proper connection between the simulator and the ROS bridge. This allowed the loading of a larger and more complex city. In this scenario, 85 vehicles and 20 pedestrians were spawned, managed by the Traffic Manager.

We divide the simulations into three different locations, a roundabout, a 5-way junction and a 4-way junction with dual carriageways, with different weather conditions. All the pictures and videos taken can be found at the link [22].

1. First simulation – Roundabout

With the drone we move to the roundabout through command:

```
rostopic pub /carla/flying_sensor/control/set_transform
geometry_msgs/Pose "{position: {x: 0, y: 0, z: 60.0}, orientation:
{x: 0.0, y: 0.0, z: 0.0, w: 1.0}}"
```

This positions the drone, and thus the RGB camera, at a height of 60 metres.

a. Clear – Sunrise

We set clear weather at sunrise with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 0.0, precipitation:
0.0, precipitation_deposits: 0.0, wind_intensity: 5.0, fog_density:
0.0, fog_distance: 1000.0, wetness: 0.0, sun_azimuth_angle: 90.0,
sun_altitude_angle: 10.0}"
```

The image of the drone is as follows:

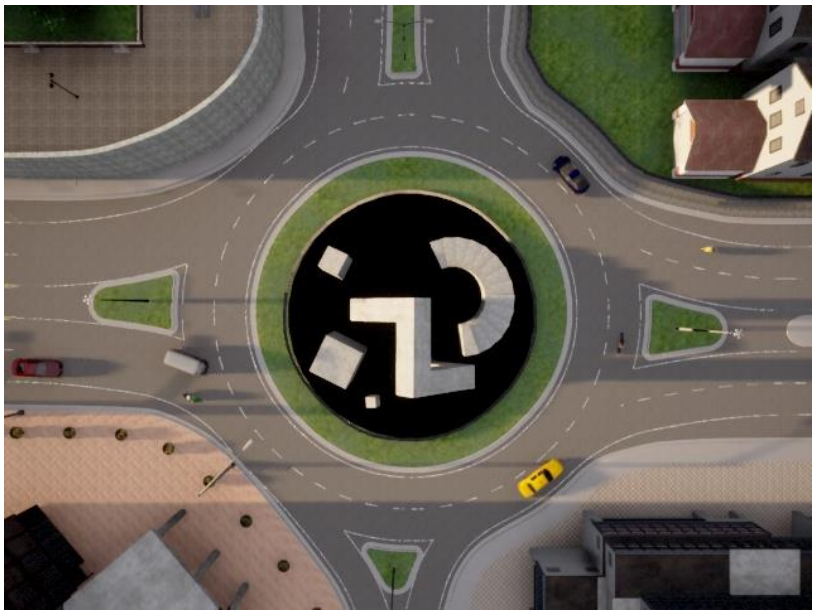

Figure 13: First simulation - Clear, sunrise

The video shows how all vehicles obey the rules of the road and travel at a moderate speed. One can also notice the presence of different types of cars and motorbikes and the presence of pedestrians walking on the pavement

b. Cloudy – Sunset

We set cloudy weather at sunset with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 100.0,
precipitation: 0.0, precipitation_deposits: 0.0, wind_intensity:
5.0, fog_density: 3.0, fog_distance: 0.75, wetness: 0.0,
sun_azimuth_angle: 270.0, sun_altitude_angle: 10.0}"
```

Figure 14: First simulation – Cloudy, sunset

Unlike the previous case, there is no sunshine as the sky is covered by clouds. From the video, one can see the repetition of the actions of vehicles and pedestrians.

c. Clear – Night

We set clear weather at night with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 0.0, precipitation:
0.0, precipitation_deposits: 0.0, wind_intensity: 5.0, fog_density:
0.0, fog_distance: 1000.0, wetness: 0.0, sun_azimuth_angle: 0.0,
sun_altitude_angle: -10.0}"
```

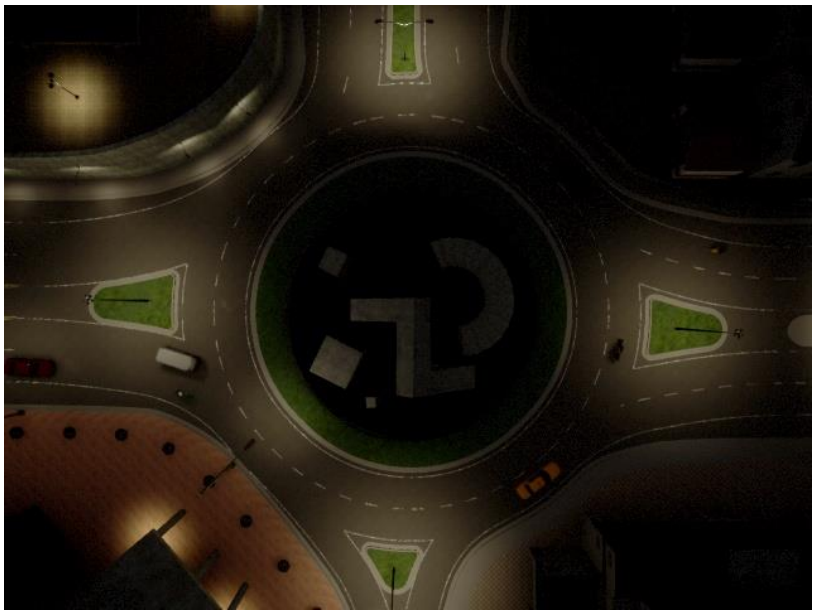

Figure 15: First simulation – Clear, night

The picture shows the roundabout at night. There is much less detail in the image and it is noticeable that the only active lights are those of the street lamps.

2. Second simulation – 5-way Junction

With the drone we move to the junction through command:

```
rostopic pub /carla/flying_sensor/control/set_transform
geometry_msgs/Pose "{position: {x: -82, y: 0, z: 40.0},
orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}"
```

This positions the drone, and thus the RGB camera, at a height of 40 metres.

a. Foggy – Noon

We set foggy weather at noon with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 70.0,
precipitation: 0.0, precipitation_deposits: 0.0, wind_intensity:
5.0, fog_density: 90.0, fog_distance: 30.0, wetness: 0.0,
sun_azimuth_angle: 180.0, sun_altitude_angle: 90.0}"
```

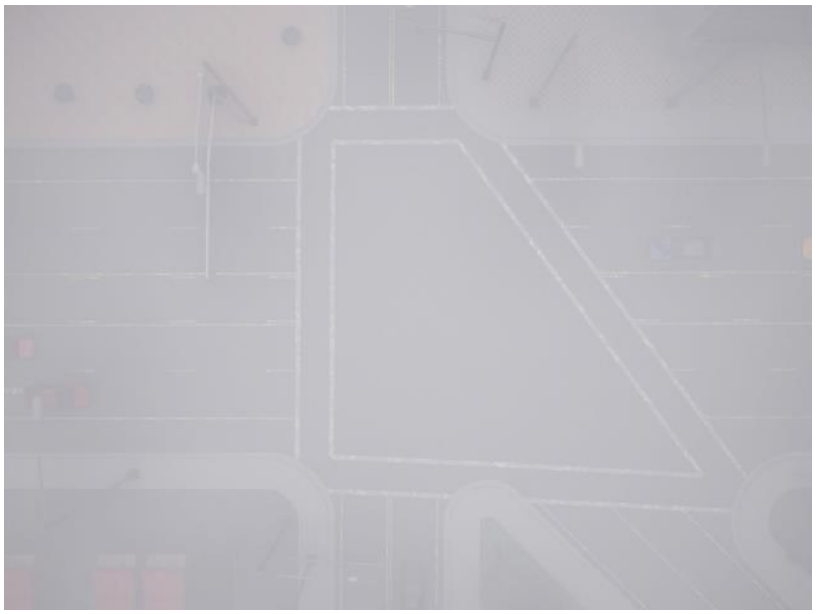

Figure 16: Second simulation - Foggy, noon

The image shows an overhead view of an intersection in a city, shrouded in thick fog or mist, which significantly reduces visibility. The urban environment is barely discernible, with some details such as road stripes, pavements and traffic lights, but vehicles and other elements are blurred and difficult to identify.

In a simulation context, this scene can represent an unfavourable atmospheric condition, ideal for testing the response of autonomous vehicle sensors or urban traffic in situations of limited visibility.

b. Foggy – Night

We set foggy weather at night with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 70.0,
precipitation: 0.0, precipitation_deposits: 0.0, wind_intensity:
5.0, fog_density: 90.0, fog_distance: 30.0, wetness: 0.0,
sun_azimuth_angle: 0.0, sun_altitude_angle: -10.0}"
```

In the following image, in addition to the night lighting, a slight haze or fog can be observed, which makes the atmosphere denser and further reduces visibility. The street lights struggle to penetrate the haze, creating a diffuse lighting effect that makes the contours of buildings and vehicles less defined.

Figure 17: Second simulation - Foggy, night

c. Light Rain – Sunrise

We set light rain weather at sunrise with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 80.0,
precipitation: 30.0, precipitation_deposits: 10.0, wind_intensity:
5.0, fog_density: 10.0, fog_distance: 600.0, wetness: 40.0,
sun_azimuth_angle: 90.0, sun_altitude_angle: 10.0}"
```

Figure 18: Second simulation - Light rain, sunrise

Small raindrops create reflections on the surface of the road surface, increasing the shine effect and making the pavement slightly slippery.

Evaluating the different videos, one can still see the reeptibility of the traffic, so the same cars choose the same route in the three cases shown above.

3. Third simulation – 4-way Junction

With the drone we move to the junction through command:

```
rostopic pub /carla/flying_sensor/control/set_transform
geometry_msgs/Pose "{position: {x: -2.5, y: -133, z: 50.0},
orientation: {x: 0.0, y: 0.0, z: 0.0, w: 1.0}}"
```

This positions the drone, and thus the RGB camera, at a height of 50 metres.

a. Medium rain – Sunset

We set medium rain weather at sunset with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 90.0,
precipitation: 60.0, precipitation_deposits: 40.0, wind_intensity:
5.0, fog_density: 20.0, fog_distance: 400.0, wetness: 70.0,
sun_azimuth_angle: 270.0, sun_altitude_angle: 10.0}"
```

The following picture represents a four-way intersection with dual carriageways. The rain, of moderate intensity, soaked the road surface heavily, creating obvious puddles and reflections of water on the asphalt.

Figure 19: Third simulation - Medium rain, sunset

b. Heavy rain – Noon

We set heavy rain weather at noon with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 100.0,
precipitation: 90.0, precipitation_deposits: 70.0, wind_intensity:
5.0, fog_density: 30.0, fog_distance: 200.0, wetness: 90.0,
sun_azimuth_angle: 180.0, sun_altitude_angle: 90.0}"
```

Figure 20: Third simulation - Heavy rain, noon

The picture shows how during heavy torrential rain, water almost completely covers the road surface, forming large puddles and creating irregular reflections on the asphalt. Visibility is reduced due to the heavy rain and mist.

c. Heavy rain – Night

We set heavy rain weather at night with the following command:

```
rostopic pub  /carla/weather_control
carla_msgs/CarlaWeatherParameters "{cloudiness: 100.0,
precipitation: 90.0, precipitation_deposits: 70.0, wind_intensity:
5.0, fog_density: 30.0, fog_distance: 200.0, wetness: 90.0,
sun_azimuth_angle: 0.0, sun_altitude_angle: -10.0}"
```

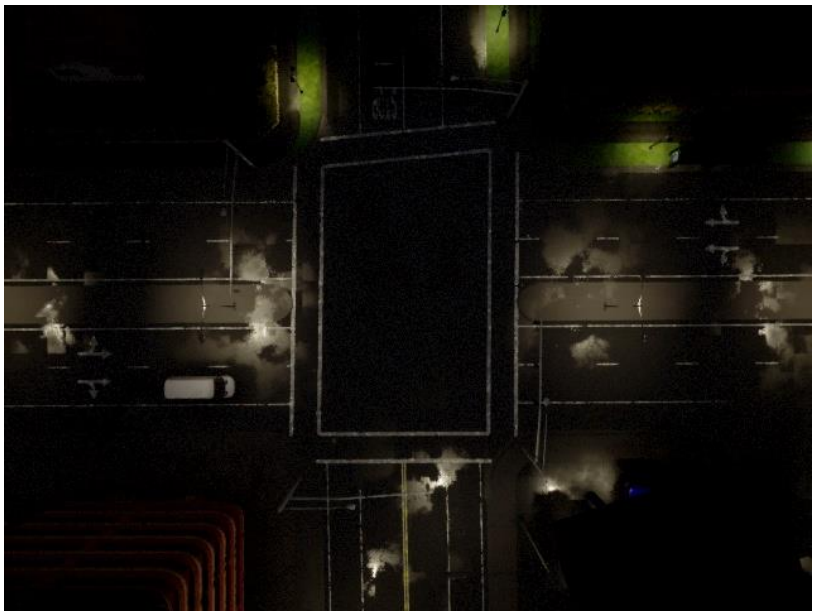

Figure 21: Third simulation - Heavy rain, night

Lighting comes exclusively from streetlamps, which cast a dim, flickering light due to the heavy rain curtain. The streets are covered with large puddles that weakly reflect the light, creating halos and shadows that add to the atmosphere of poor visibility.

Again, comparing all the videos, the vehicles perform the same manoeuvres consistently. This again confirms the repeatability of the system.

4. Conclusions and future implementations

In this project, we developed a system for managing vehicle and pedestrian traffic in an advanced simulation environment, using the Carla simulator integrated with ROS1. The main objective was to simulate complex traffic scenarios and to capture images and videos via an RGB camera installed on a drone. Thanks to the use of ROS Bridge and Carla's API, such as the Traffic Manager, we achieved our goal in a way that was certainly not easy.

During the development of the project, some significant problems emerged:

- *Difficulties in creating a complete and working Docker container*: it was necessary to create a Dockerfile based on files originally designed for ROS2, which were then completely rewritten for ROS1.

- *Sub-optimal spawning of vehicles and pedestrians*: Initially, vehicles were spawned in wrong positions, e.g. in the opposite direction of travel, while pedestrians were spawned in inappropriate positions, such as on buildings or in the middle of the road.

- *Synchronisation of the Traffic Manager*: activation of the synchronous mode for the Traffic Manager caused problems in the movement of vehicles, with the server crashing. The solution was to keep the Traffic Manager in asynchronous mode and synchronise the simulator instead. In fact, the ROS bridge operates in synchronous mode by default, waiting for all expected sensor data within the current frame to ensure reproducible results. When running multiple clients in synchronous mode (e.g. Traffic Manager), only one client can update the world. By default, the ROS bridge is the only client authorised to perform this update, unless passive mode is activated. By activating passive mode, the ROS bridge withdraws, allowing another client to attempt to update the world. However, this caused problems when starting the bridge.

- *Synchronisation for photos and videos*: this problem was solved by using the same frequency as the ROS bridge, verified with the following command:

```
rostopic hz /carla/flying_sensor/rgb_down/image
```

The `rostopic hz` command is used to monitor the frequency with which messages are posted to a particular topic, in this case /carla/flying_sensor/rgb_down/image. This ensures that images are captured consistently and that synchronisation with the ROS bridge is correct.

This work represents a solid basis, but there are numerous opportunities for future extensions, including:

- *Development of a script for vehicle light management*: implement logic that activates vehicle headlights based on environmental conditions, such as time of day or visibility. This would improve simulation fidelity in night or adverse weather scenarios.

- *Integration with environmental sensing sensors*: enhance the simulation with additional vehicle sensors, such as light or rain sensors, that can adapt the behaviour of actors, further improving traffic realism

References

[1] https://github.com/ricardodeazambuja/ros2_quad_sim_python - Ricardo De Azambuja, ROS2 packages to simulate a quadcopter.

[2] https://github.com/ricardodeazambuja/carla-ros - Carla Ros, ROS package that enables two-way communication between ROS and CARLA.

[3] https://carla.org/ - Carla Simulator.

[4] https://github.com/ricardodeazambuja/carla-simulator-python - GitHub repository for Carla Simulator container and launch file.

[5] https://github.com/FrankSav80/Progetto-MR-Savinox2.git - Dockerfile private repository.

[6] https://github.com/FrankSav80/provax/blob/master/carla_ros_bridge/launch/carla_ros_bridge.launch – File carla_ros_bridge.launch.

[7] https://github.com/FrankSav80/provax/blob/master/carla_ros_bridge/src/carla_ ros_bridge/bridge.py – File bridge.py.

[8] https://carla.readthedocs.io/en/latest/adv_synchrony_timestep/ - Synchrony and time-step documentation.

[9] https://carla.readthedocs.io/en/latest/map_town01/ - Carla Town01.

[10] https://carla.readthedocs.io/en/latest/map_town03/ - Carla Town03.

[11] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/launch/carla_spawn_objects.launch - File carla_spawn_objects.launch.

[12] https://github.com/FrankSav80/provay/blob/main/src/ros2_quad_sim_python/cfg/flying_sensor_full.json - File flying_sensor_full.json.

[13] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/src/carla_spawn_objects/carla_spawn_objects.py - File carla_spawn_objects.py.

[14] https://carla.readthedocs.io/en/latest/bp_library/ - Blueprint Library.

[15] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/launch/traffic_manager.launch - File traffic_manager.launch.

[16] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/src/carla_spawn_objects/traffic_manager.py - File traffic_manager.py.

[17] https://github.com/FrankSav80/provax/blob/master/rviz_carla_RGBdown.rviz - File rviz_carla_RGBdown.rviz.

[18] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/launch/drone_image.launch - File drone_image.launch.

[19] https://github.com/FrankSav80/provax/blob/master/carla_spawn_objects/src/carla_spawn_objects/drone_image.py - File drone_image.py.

[20] https://wiki.ros.org/cv_bridge - cv_bridge Package.

[21] https://github.com/FrankSav80/provax/blob/master/Weather%20properties.xlsx – File Excel for weather.

[22] https://www.dropbox.com/scl/fo/tuhgh7e87fqexrcxvye9q/AJ0CdbNS0UzjltcKa3lT2L4?rlkey=68kqr1dumfde5up8zfdiqdm9y&st=mdci8ey8&dl=0