МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

В. І. Таран,
О. П. Роковий,
Ю. Г. Гордієнко,
С. Г. Стіренко

ОСНОВИ ОПЕРАЦІЙНОЇ СИСТЕМИ ДЛЯ РОБОТІВ

Практикум

Навчальний посібник

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня магістра
за освітніми програмами «Інженерія програмного забезпечення комп’ютерних систем»
спеціальності F2 Інженерія програмного забезпечення
та «Комп’ютерні системи та мережі» спеціальності F7 Комп’ютерна інженерія

Електронне мережне навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2025

УДК 004.8
В50

Автори: *Таран Владислав Ігорович*, д-р філософії, доц.
Роковий Олександр Петрович, канд. техн. наук, доц.
Гордієнко Юрій Григорович, д-р фіз.-мат. наук, ст. наук. співроб.
Стіренко Сергій Григорович, д-р техн. наук, проф.

Рецензент *Ролік О. І.*, д-р техн. наук, проф.,
завідувач кафедри інформаційних систем та технологій
КПІ ім. Ігоря Сікорського

Відповідальний редактор *Новотарський М. А.*, д-р техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського
(протокол № 3 від 12.12.2025 р.)
за поданням вченої ради Факультету інформатики та обчислювальної техніки
(протокол № 4 від 24.11.2025 р.)

Таран В. І.

В50 Основи операційної системи для роботів [Електронний ресурс] : практикум : навч. посіб. для здобувачів ступеня магістра за освіт. програмами «Інженерія програмного забезпечення комп'ютерних систем» спец. F2 Інженерія програмного забезпечення та «Комп'ютерні системи та мережі» спец. F7 Комп'ютерна інженерія / В. І. Таран, О. П. Роковий, Ю. Г. Гордієнко, С. Г. Стіренко ; КПІ ім. Ігоря Сікорського. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2025. – 110 с.

Навчальний посібник розроблено для ознайомлення здобувачів із сучасними підходами, методами, механізмами функціонування та використання інфраструктури для побудови та управління роботизованими системами за допомогою ROS2 в контексті її застосування із безпілотними літальними апаратами, а також розробкою програмних модулів для різних сценаріїв їх використання. Посібник містить інформацію щодо теоретичних та практичних основ використання ROS2, що включає налаштування її компонентів, підготовку інфраструктури симуляції БПЛА та розробку окремих програмних модулів керування.

Навчальний посібник призначений для здобувачів ступеня магістра напряму підготовки за освітніми програмами "Інженерія програмного забезпечення комп'ютерних систем" спеціальності F2 - "Інженерія програмного забезпечення" та "Комп'ютерні системи та мережі" спеціальності F7 - "Комп'ютерна інженерія".

УДК 004.8

Реєстр. № НП 25/26-124. Обсяг 5,0 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Перемоги, 37, м. Київ, 03056
https://kpi.ua

Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

Цей навчальний посібник було підготовлено в рамках міжнародного освітнього проєкту *Штучний інтелект для дотичної індустрії — AI4CI*. Магістерська програма *AI4CI* співфінансується Європейською Комісією, програмою «Цифрова Європа», якою керує *HADEA*, номер контракту: 101123524.

This textbook was prepared in the framework of *Artificial Intelligence for Connected Industries — AI4CI* project. The *AI4CI* master is co-funded by the European Commission, Digital Europe Programme, managed by *HADEA*, contract nb: 101123524.

ЗМІСТ

ВСТУП

Навчальний посібник “Основи операційної системи для роботів. Практикум” призначений для використання в рамках викладання дисципліни “Операційна система для роботів (ROS)” магістратури 1-го курсу. Матеріал спрямований на вивчення підходів, методів, механізмів функціонування та використання інфраструктури для побудови та управління роботизованими системами за допомогою *ROS2*. Необхідність в використанні *ROS2* обумовлена тим, що сучасні підходи до вирішення завдань у робототехніці потребують побудови складних модульних компонентів та їх ефективну інтеграцію в комплексну систему, що в свою чергу вимагає застосування абстракцій рівня апаратного забезпечення, низько-рівневого керування пристроями, реалізації загальновживаних функцій та наявності засобів передачі повідомлень між компонентами роботизованої системи. Навчальний посібник дозволить майбутніми фахівцями набути важливих компетенцій в плані використання нових підходів, програмних інструментів та бібліотеки для написання, побудови та запуску програмного забезпечення для роботизованих систем у децентралізованих та розподілених інфраструктурах.

Метою навчального посібника є підготовка фахівців, здатних розв’язувати комплексні задачі у сфері побудови роботизованих систем та використовувати сучасні засоби для організації їх функціонування в комплексних децентралізованих та розподілених інфраструктурах.

Предметом навчального посібника є:

• підходи, методи та засоби побудови роботизованих систем в децентралізованих та розподілених інфраструктурах;

• механізми організації взаємодії компонентів роботизованих систем.

1 СИМУЛЯЦІЯ ДРОНІВ НА БАЗІ *ARDUPILOT*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 20.04.5 (*minimal installation*);
- *SITL ArduPilot* 4.3.3;
- *QGroundControl* 4.2.4.

1.1 Налаштування *SITL ArduPilot*

Репозиторій з програмним кодом знаходиться за посиланням: (https://github.com/ArduPilot/ardupilot).

Для початку виконується завантаження та налаштування необхідного програмного забезпечення *SITL ArduPilot*.

```
    # Команди виконуються із домашнього каталогу
1.  cd ~
    # Встановлення инструменту Git, в разі його відсутності
2.  sudo apt install git
3.  git clone https://github.com/ArduPilot/ardupilot.git
4.  cd ardupilot
    # Встановлення залежностей
5.  Tools/environment_install/install-prereqs-ubuntu.sh -y
    # Завантаження змінних середовища
6.  . ~/.profile
    # Переміщення на актуальну гілку (в прикладі – версія 4.3.3)
7.  git checkout Copter-4.3.3
8.  git submodule update --init --recursive
```

Після завершення налаштування програмного забезпечення, запуск симулятору для одного дрону виконується із терміналу наступною командою. В результаті запуститься консольна програма для вводу команд керування дроном, а також додаткове вікно *ArduCopter*, де відображаються основні параметри симулятору *ArduPilot* (рис. 1.1).

```
# Запуск симулятора дрону
cd ~/ardupilot/ArduCopter
sim_vehicle.py -w
```

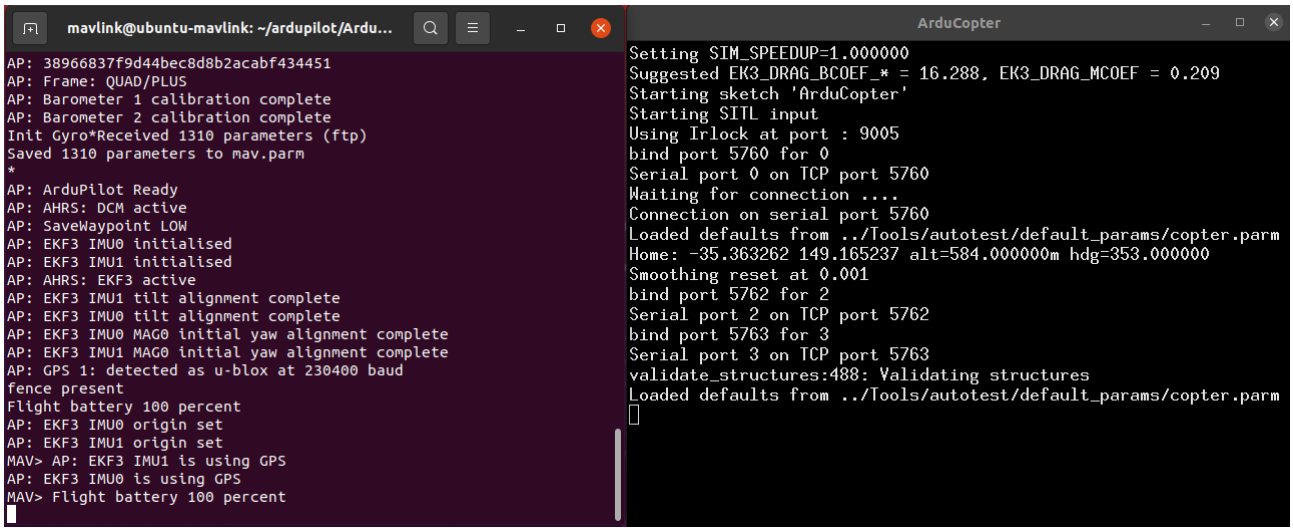


Рисунок 1.1 — Запущений симулятор *SITL ArduPilot*

1.2 Налаштування *QGroundControl*

Програмне забезпечення *QGroundControl* надає графічний інтерфейс для керування через протокол *MAVLink* та завантаження планів польоту для літальних апаратів з прошивками *ArduPilot* та *PX4.*

Основна інформація по *QGroundControl* знаходиться на офіційному сайті: (https://docs.qgroundcontrol.com/master/en/).

Завантаження та налаштування *QGroundControl* виконується наступними командами:

```
# Підготовка середовища та встановлення залежностей
sudo usermod -a -G dialout $USER
sudo apt remove modemmanager
sudo apt install gstreamer1.0-plugins-bad gstreamer1.0-libav gstreamer1.0-gl libqt5gui5 libfuse2
# Завантаження останньої версії QGroundControl (в прикладі – версія 4.2.4)
Див. - https://docs.qgroundcontrol.com/master/en/qgc-user-guide/getting_started/download_and_install.html
chmod +x ~/QGroundControl.AppImage
```

Після завершення налаштувань, запуск *QGroundControl* виконується у терміналі командою нижче. В результаті відкриється вікно програми. В разі,

якщо симулятор дрону *SITL ArduPilot* (рис. 1.1) також запущений, то *QGroundControl* автоматично підключиться до нього, і стануть доступні елементи керування дроном (рис. 1.2).

```
# Запуск QGroundControl
~/QGroundControl.AppImage
```

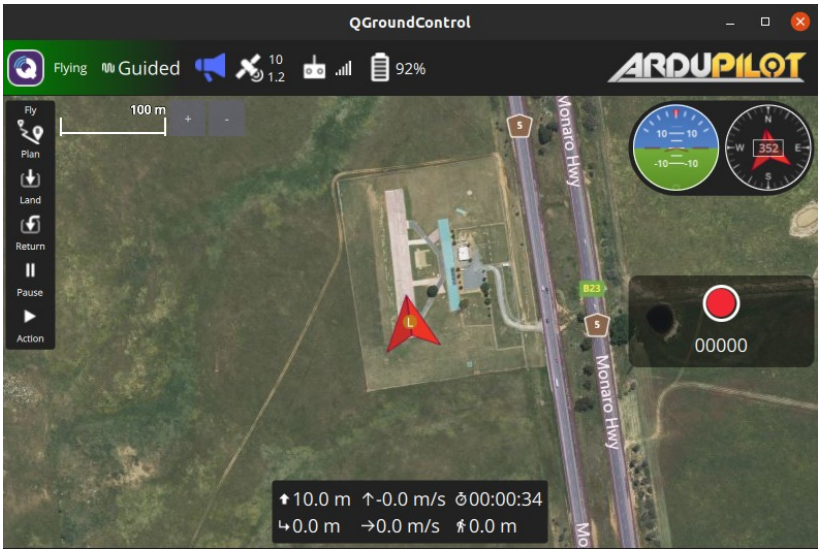


Рисунок 1.2 — Вікно програми *QGroundControl*

1.3 Налаштування для симуляції декількох дронів

Вище було описано як виконати симуляцію для одного дрону, а також як запустити програму керування *QGroundControl*. В разі якщо одночасно необхідно керувати декількома симуляціями дронів, потрібно виконати додаткові налаштування середовища. Оскільки протокол *MAVLink* використовує певні порти для взаємодії із програмою керування, потрібно доповнити параметри симулятора, щоб для різних екземплярів дрону *SITL ArduPilot* виділялись різні порти та присвоювались різні ідентифікатори. Після внесення зазначених змін програма керування *QGroundControl* зможе підключатись та відображати декілька екземплярів дронів.

Спочатку треба доповнити симулятор параметрами для додаткових екземплярів дронів. Для цього необхідно відредагувати файл *vehicleinfo.py* (рис. 1.3). Файл знаходиться у директорії: *~/ardupilot/Tools/autotest/pysim*.

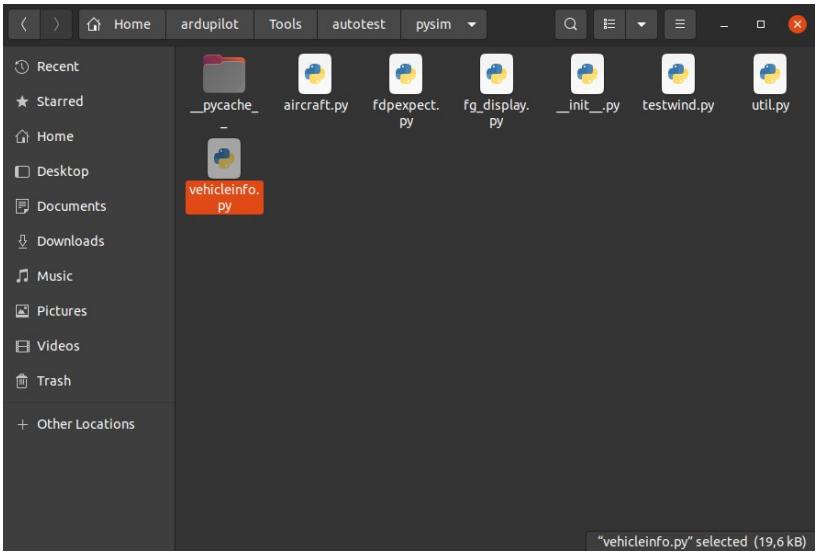


Рисунок 1.3 — Файл *vehicleinfo.py*, який потрібно відредагувати

Для прикладу додамо в цей файл параметри для трьох додаткових екземплярів дронів.

```
"copter-drone1": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sitl-drone1.parm"],
},
"copter-drone2": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sitl-drone2.parm"],
},
"copter-drone3": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sitl-drone3.parm"],
},
```

Після редагування вміст файлу *vehicleinfo.py*, має виглядати наступним чином (рис. 1.4).

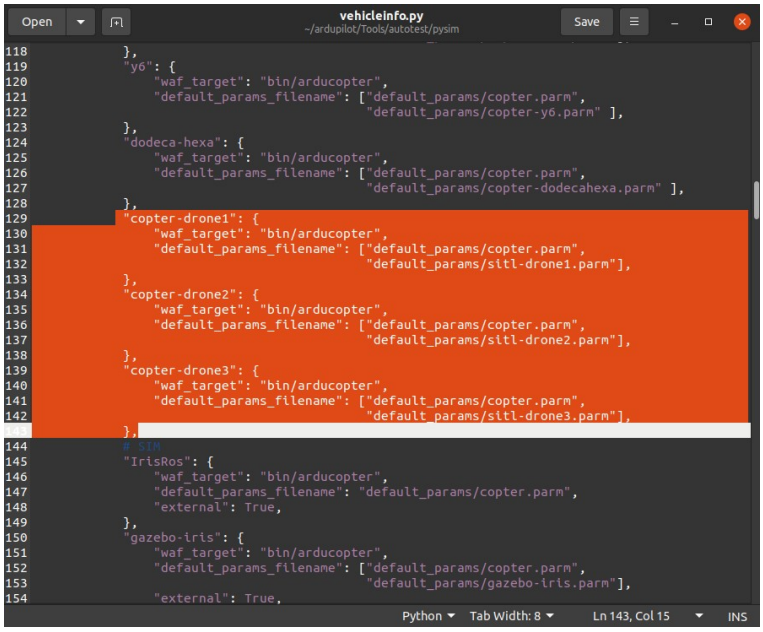


Рисунок 1.4 — Файл *vehicleinfo.py* після редагування

Важливими параметрами тут є:

1. *"copter-drone1"*;

2. *"default_params/sitl-drone1.parm"*.

1-й параметр — задає назву екземпляру, яка в подальшому буде використовуватись під час запуску екземпляру симулятора. Важливим є те, що слово “*copter*” в цій назві вказує на базову модель, яку слід використовувати симулятору. Тому, ця частина назви не може бути випадковою, інакше виникатиме помилка відсутності моделі. 2-й параметр — задає файл де вказані налаштування для відповідної симуляції дрону.

Далі необхідно створити відповідні файли налаштувань, які були вказані у “*default_params_filename”*. В прикладі це буде три файли: *sitl-drone1.parm*, *sitl-drone2.parm* та *sitl-drone3.parm*. Зазначені файли необхідно створити у директорії: *~/ardupilot/Tools/autotest/default_params*. В найпростішому вигляді кожен файл буде містити значення ідентифікатору для окремого екземпляру дрону.

```
1. mavlink@ubuntu-mavlink:~/ardupilot/Tools/autotest/default_params$ cat sitl-drone1.parm
   SYSID_THISMAV 1
2. mavlink@ubuntu-mavlink:~/ardupilot/Tools/autotest/default_params$ cat sitl-drone2.parm
   SYSID_THISMAV 2
3. mavlink@ubuntu-mavlink:~/ardupilot/Tools/autotest/default_params$ cat sitl-drone3.parm
   SYSID_THISMAV 3
```

1.4 Запуск симуляції для декількох дронів

Після виконання зазначених вище кроків, середовище готове для запуску симуляції декількох дронів. Для цього потрібно запустити декілька екземплярів *SITL ArduPilot* із різними параметрами. Важливими параметрами при запуску є:

- *-f* — задає налаштування для кожного екземпляру;
- *-I* — задає унікальний номер відповідного екземпляру;
- *--out* — задає додатковий порт для підключення по *MAVLink*.

Для прикладу запустимо два екземпляри наступними командами (кожна виконується в окремому терміналі).

```
# 1-й екземпляр

1. mavlink@ubuntu-mavlink:~$ sim_vehicle.py -v ArduCopter -f copter-drone1 --console -I0 --
   out=tcpin:0.0.0.0:8100

# 2-й екземпляр

2. mavlink@ubuntu-mavlink:~$ sim_vehicle.py -v ArduCopter -f copter-drone2 --console -I1 --
   out=tcpin:0.0.0.0:8200
```

В результаті запустяться два симулятори і додатково відкриються консолі із телеметрією по кожному із дронів (рис. 1.5).

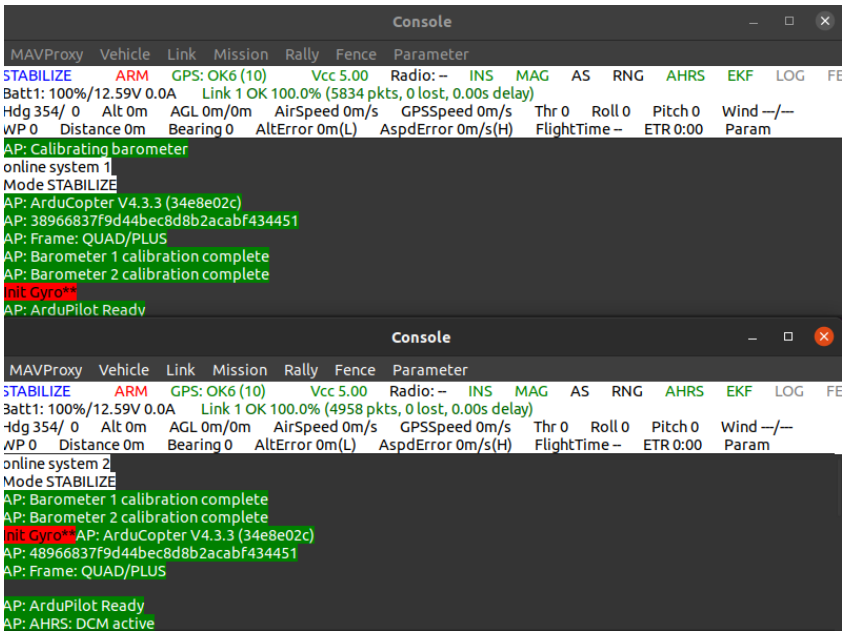


Рисунок 1.5 — Консолі із телеметрією для двох симуляцій дронів

1.5 Керування декількома симуляціями дронів засобами *QGroundControl*

Для керування додатковими екземплярами дронів у *QGroundControl* потрібно внести зміни у налаштування підключень до дронів. Для цього у лівому верхньому куті програми необхідно натиснути на її логотип. Далі у вікні “*Select Tool*”, що відкрилось, обирається “*Application Settings*”. У вікні налаштувань обирається пункт “*Comm Links*” (рис. 1.6).

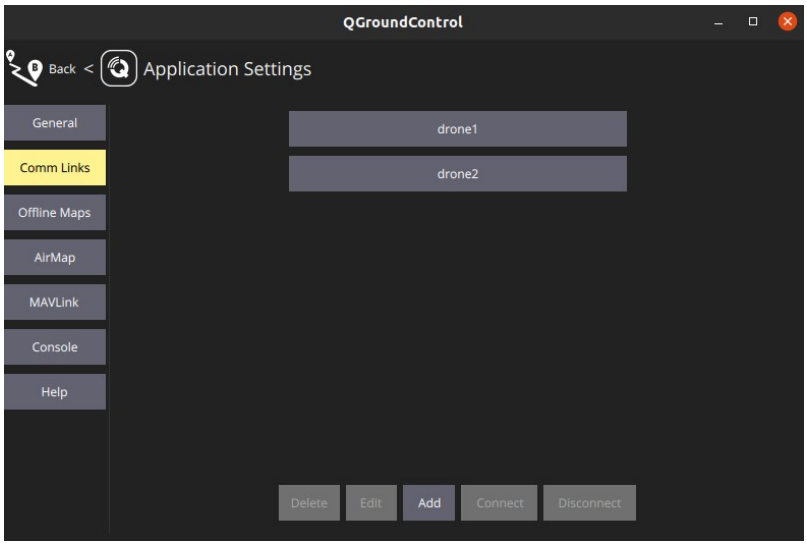


Рисунок 1.6 — Налаштування параметрів підключення

Наступним кроком обирається дія “*Add*” і задаються параметри підключення до кожної із симуляції дрону. Налаштування 2-го дрону показані на рис. 1.7. Для підключення дрону обирається дія “*Connect*”.

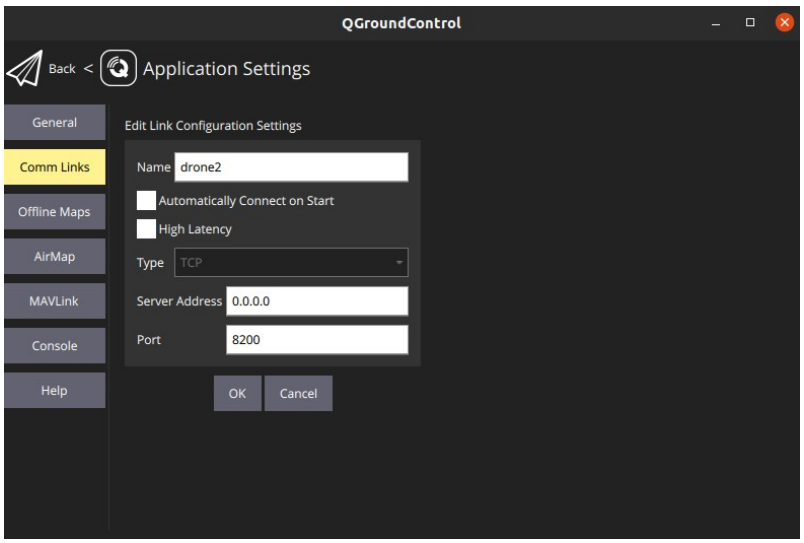


Рисунок 1.7 — Налаштування для 2-го симулятору дрону

Після успішного налаштування та підключення до обох симуляцій дронів, вони мають відобразитись у головному вікні *QGroundControl* (рис. 1.8).

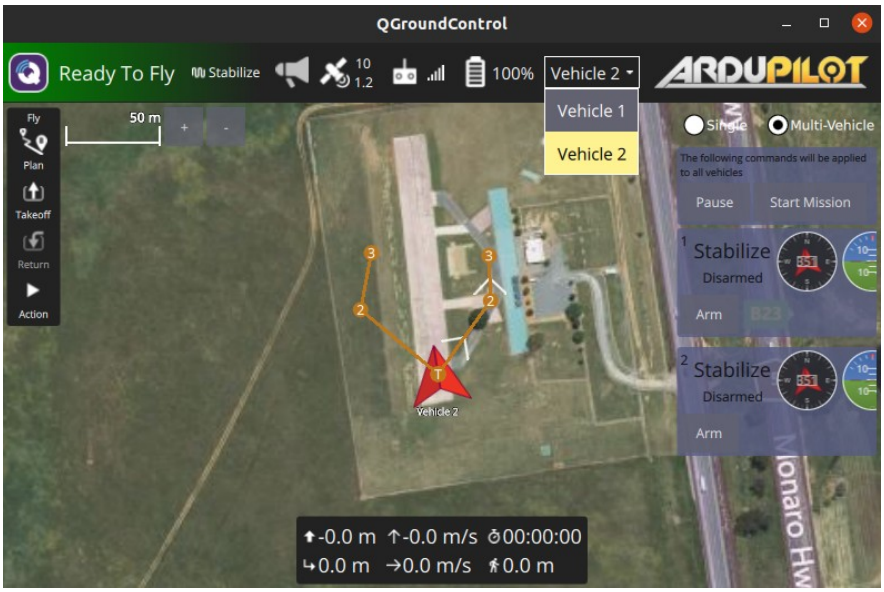


Рисунок 1.8 — *QGroundControl* із двома симуляціями дронів

В прикладі політні плани вже завантажені в обидва дрони. Редагувати ці плани можна обравши дію “*Plan*” у лівій частині вікна програми, а також можна завантажувати оновлені плани польоту дією “*Upload*” або “*Upload Required*” у правому верхньому куті вікна. Щоб одночасно запустити політні плани на обох дронах треба обрати дію “*Start Mission*” у правій частині вікна програми та виконати підтвердження виконання дією “*Slide to confirm*”. Результат польоту двох дронів показано на рис. 1.9.

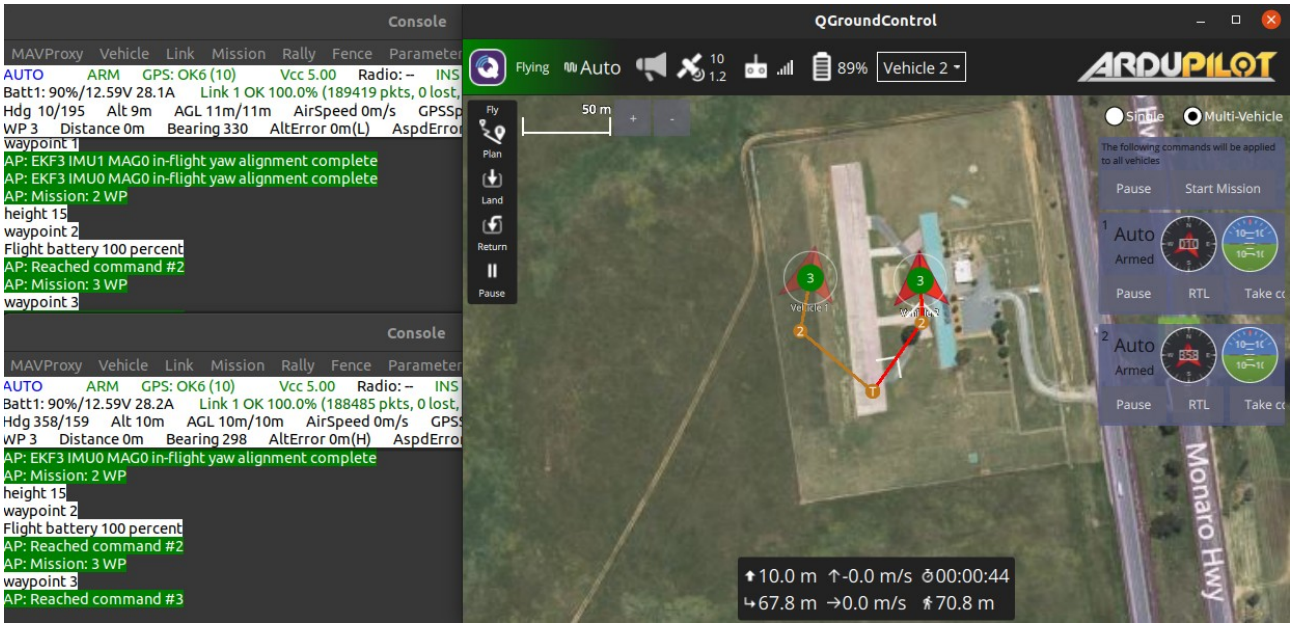


Рисунок 1.9 — Виконання політного плану двома дронами

2 СИМУЛЯЦІЯ ДРОНІВ НА БАЗІ АВТОПІЛОТУ *PX4* ІЗ ВИКОРИСТАННЯМ *ROS*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 20.04.5 (*minimal installation*);
- *ROS Noetic* 1.16.0;
- *PX4 Autopilot* 1.13.3;
- *Gazebo Simulator* 11.12.0;
- *QGroundControl* 4.2.4.

2.1 Налаштування *SITL PX4*

Репозиторій з програмним кодом знаходиться за посиланням: (https://github.com/PX4/PX4-Autopilot)

Офіційна документація *PX4* радить встановлювати ПЗ із майстер гілки репозиторію, але періодично майстер гілка містити бета-версії. Через це функції управління автопілоту можуть не працювати як очікується. Слідуючи рекомендованим крокам по встановленню із офіційної документації може виникнути ситуація, коли буде завантажена актуальна версія, яка наразі має статус “бета” (рис. 2.1).

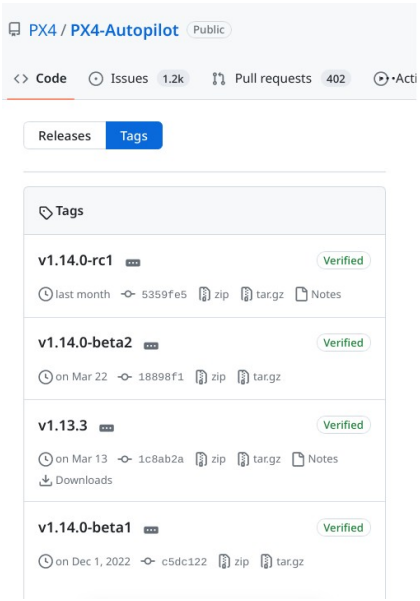


Рисунок 2.1 — Версії репозиторію *PX4 Autopilot*

Наприклад, у версії “*1.14.0-beta1*”, в режимі польоту *offboard* команди керування некоректно працюють через *MAVLink*. Щоб уникнути такої проблеми, слід встановлювати стабільну версію із репозиторію. На момент написання це версія “*1.13.3*”.

Завантаження та налаштування необхідного програмного забезпечення *SITL PX4 Autopilot* виконується наступними командами:

```
# Команди виконуються із каталогу ROS
cd ~/ROS
# Встановлення инструменту Git, в разі його відсутності
sudo apt install git
# Клонування стабільної версії
git clone https://github.com/PX4/PX4-Autopilot.git --recursive --branch v1.13.3
# Встановлення залежностей
bash ./PX4-Autopilot/Tools/setup/ubuntu.sh --no-sim-tools --no-nuttx
sudo apt install libgstreamer-plugins-base1.0-dev
# Налаштування робочого середовища та майбутня інтеграція із ROS
echo "export PX4_ROOT=~/ROS/PX4-Autopilot" >> ~/.bashrc
mkdir ~/.ros && ln -sf ${PX4_ROOT}/ROMFS ~/.ros/
```

Після вказаних кроків *PX4* потрібно запустити. Для цього виконаємо команди нижче (рис 2.2). Під час першого запуску виконується компіляція коду, що може зайняти тривалий час.

```
cd ~/ROS/PX4-Autopilot
# Компіляція без симулятору
make px4_sitl none_iris
```

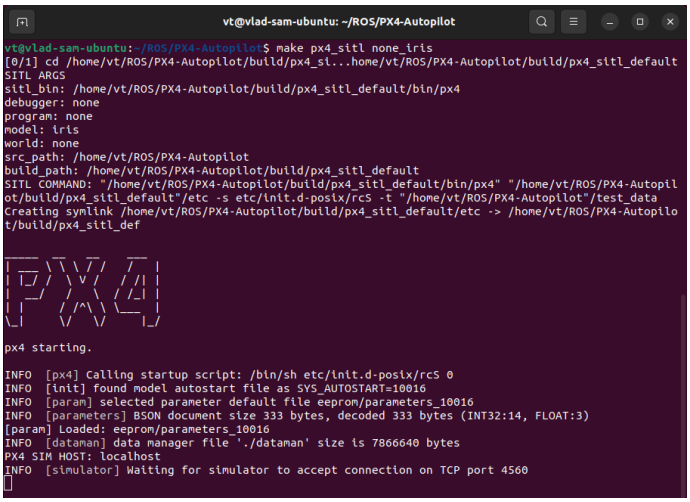


Рисунок 2.2 — Запуск *SITL PX4 Autopilot*

Для роботи *SITL* автопілот *PX4* потребує даних телеметрії, які генеруються ПЗ для симуляції. В цьому прикладі використовуватиметься симулятор віртуального оточення *Gazebo*.

2.2 Завантаження симулятора *Gazebo*

Репозиторій в дистрибутиві *Ubuntu* містить готовий пакет симулятору *Gazebo*, тому його встановлення відбувається через вбудований менеджер пакетів наступною командою.

```
# За потреби додаємо репозиторій Gazebo, якщо потрібної версії не має у репозиторії обраного дистрибутиву
1. sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable `lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
2. wget https://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
# Вказуємо пакет gazebo11 (або просто gazebo, якщо у репозиторії зразу пропонується 11-а версія)
3. sudo apt install gazebo11 libgazebo11-dev libopencv-dev
```

Після успішного встановлення симулятору можна запустити *SITL* автопілот *PX4* разом із симулятором (рис. 2.3).

```
1. cd ~/ROS/PX4-Autopilot
   # Запуск із симулятором Gazebo (додаткова компіляція при першому виконанні)
2. make px4_sitl gazebo
```

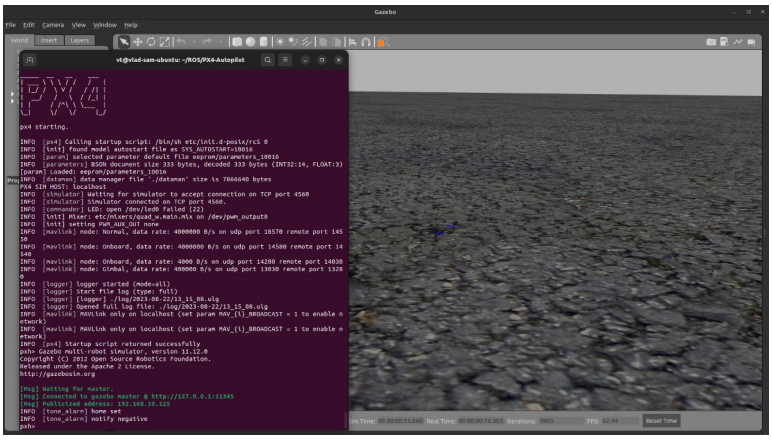


Рисунок 2.3 — Запуск *SITL PX4 Autopilot* із симулятором *Gazebo*

Щоб перевірити роботи автопілоту можна виконати зліт дрону через консоль *PX4* або засобами програми керування *QGroundControl*. У разі роботи через консоль слід ввести команду “*commander takeoff*”. Для успішного результату потрібно, щоб *QGroundControl* був запущений, оскільки

налаштування автопілоту за замовчуванням не дозволяються виконувати зліт без з’єднання із пультом або програмою дистанційного керування.

2.3 Налаштування *ROS*

ROS (Robot Operating System) — мета-операційна система для робототехніки із відкритим програмним кодом. Вона надає функції, що притаманні операційним системам, такі як, абстракція рівня апаратного забезпечення, низько-рівневе керування пристроями, реалізація загальновживаних функцій, засобів передачі повідомлень між процесами та системи керування пакетами. *ROS* також надає програмні інструменти та бібліотеки для написання, побудови та запуску програмного забезпечення у розподіленій інфраструктурі.

У прикладі використовується актуальна версія із тривалою підтримкою — *ROS Noetic*. Згідно офіційної документації (http://wiki.ros.org/noetic/Installation) ця версія сумісна із версією ОС *Ubuntu* 20.04.5. Щоб встановити та налаштувати *ROS* потрібно підключити сторонній репозиторій та завантажити відповідну версію пакету. Для цього виконуються наступні команди:

```
# Підключення репозиторію ROS
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list
# Встановлення curl в разі відсутності та додавання ключа
sudo apt install curl
curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
# Оновлення списку пакетів та встановлення ROS Noetic
sudo apt update
sudo apt install ros-noetic-desktop
# Встановлення пакету для роботи із Gazebo (якщо не встановлено автоматично)
sudo apt install ros-noetic-gazebo-ros
# Встановлення бібліотеки MAVLink для ROS та її ініціалізація
sudo apt install ros-noetic-mavros ros-noetic-mavros-msgs
sudo /opt/ros/noetic/lib/mavros/install_geographiclib_datasets.sh
```

Після завантаження *ROS* необхідно виконати налаштування робочого середовища для подальшої розробки та запуску програм.

```
# Оновлення змінних середовища
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc
# Встановлення залежностей для побудови та розробки
sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential
# Ініціалізація менеджеру пакетів ROS
sudo rosdep init
rosdep update
```

Перевірити налаштоване середовище *ROS* можна шляхом запуску майстер процесу *“ROS core”*. Для цього в двох окремих вікнах терміналу потрібно виконати наступні команди. Результат має буди подібним до рис. 2.4.

```
# Термінал 1
roscore
# Термінал 2
rostopic list
```

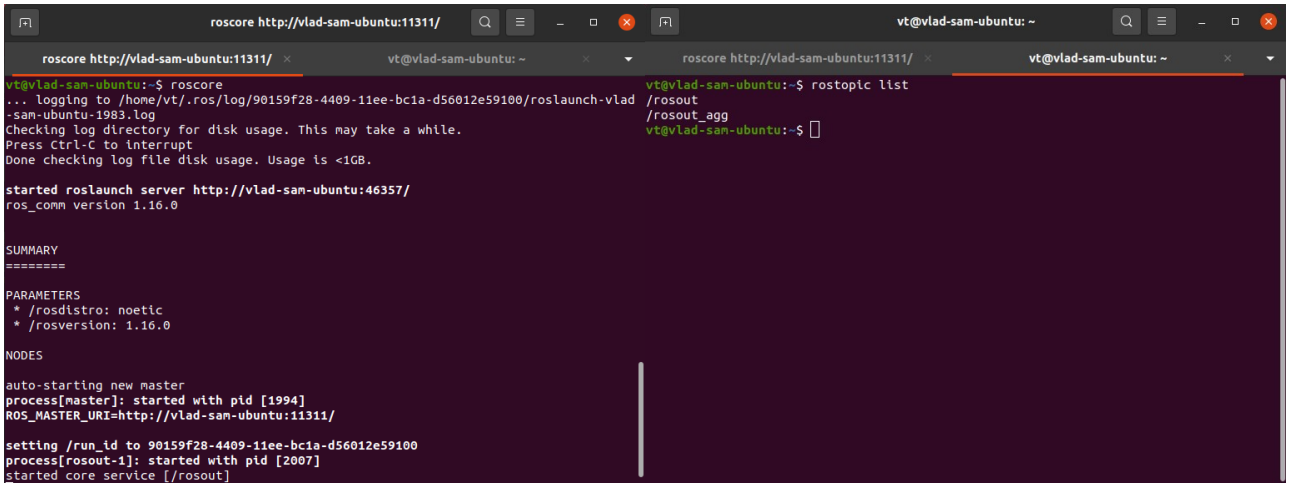


Рисунок 2.4 — Тестовий запуск середовища *ROS*

2.3.1 Інтеграція *PX4 Autopilot* із *ROS*

Програмне забезпечення *PX4* офіційно підтримує *ROS*, тому для інтеграції достатньо внести декілька змін в налаштування середовища у файлі *“.bashrc”*.

```
# Оновлення змінних середовища для PX4
source ~/ROS/PX4-Autopilot/Tools/setup_gazebo.bash ~/ROS/PX4-Autopilot ~/ROS/PX4-Autopilot/build/px4_sitl_default
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/ROS/PX4-Autopilot
export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:~/ROS/PX4-Autopilot/Tools/sitl_gazebo
```

2.4 Приклад програми *ROS* для керування дроном

Для демонстрації роботи із *ROS* для керування дроном на базі автопілоту *PX4* було розроблено програму “*simple_ros_app*”, яка використовує протокол *MAVLink*, щоб відправляти команди дрону. Програмний проєкт має структуру наведену на рис. 2.5.

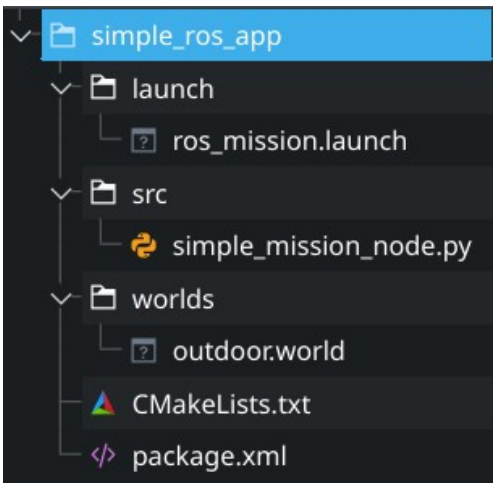


Рисунок 2.5 — Структура програмного проєкту *simple_ros_app*

Ключовими компонентами проєкту *ROS* є:

• Програмний код для процесу *ROS* — *node*. У прикладі код розроблений на мові програмування *Python*;

• Інтегрований файл запуску програми *ROS* — “*launch file*”, який дозволяє запустити програму *ROS* одним командним викликом.

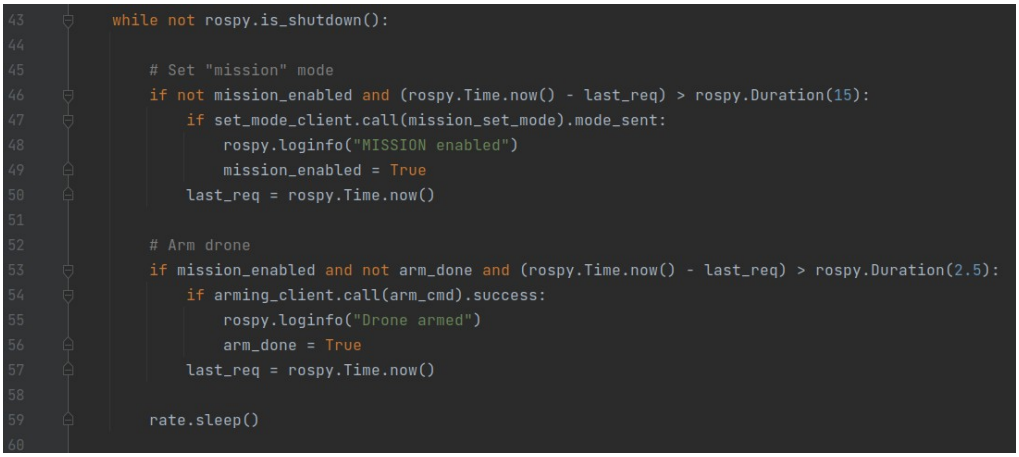
Типову структуру програми *ROS* можна умовно розділити на два етапи:

1. Ініціалізація процесу *ROS* (рис. 2.6);

2. Виконання керуючих дій (рис. 2.7).

```
def run_node():
    rospy.init_node("simple_mission_node")

    state_sub = rospy.Subscriber("/mavros/state", State, callback=state_cb)

    rospy.wait_for_service("/mavros/cmd/arming")
    arming_client = rospy.ServiceProxy("/mavros/cmd/arming", CommandBool)

    rospy.wait_for_service("/mavros/set_mode")
    set_mode_client = rospy.ServiceProxy("/mavros/set_mode", SetMode)

    rate = rospy.Rate(20)

    # Wait for Flight Controller connection
    while not rospy.is_shutdown() and not current_state.connected:
        rate.sleep()

    mission_set_mode = SetModeRequest()
    mission_set_mode.custom_mode = 'AUTO.MISSION'

    arm_cmd = CommandBoolRequest()
    arm_cmd.value = True

    last_req = rospy.Time.now()
    mission_enabled = False
    arm_done = False
```

Рисунок 2.6 — Частина програми ініціалізації процесу *ROS*



```
while not rospy.is_shutdown():

    # Set "mission" mode
    if not mission_enabled and (rospy.Time.now() - last_req) > rospy.Duration(15):
        if set_mode_client.call(mission_set_mode).mode_sent:
            rospy.loginfo("MISSION enabled")
            mission_enabled = True
        last_req = rospy.Time.now()

    # Arm drone
    if mission_enabled and not arm_done and (rospy.Time.now() - last_req) > rospy.Duration(2.5):
        if arming_client.call(arm_cmd).success:
            rospy.loginfo("Drone armed")
            arm_done = True
        last_req = rospy.Time.now()

    rate.sleep()
```

Рисунок 2.7 — Частина програми виконання керуючих дій

Основною функцією розробленої програми є встановлення для дрону режиму польоту “*mission*”, після чого виконання його запуску командою “*arm*”. Після виконання зазначених кроків, надалі автопілот дрону самостійно керує польотом згідно завантаженого через *QGroundControl* політного плану. Інтеграція *MAVLink* із *ROS* забезпечується бібліотекою *MAVROS*. Бібліотека дає можливість транслювати повідомлення *MAVLink — messages*, які генерується ПЗ автопілоту у повідомлення *ROS*. Це дозволяє взаємодіяти із ПЗ дрону шляхом стандартного інтерфейсу повідомлень *ROS — topics*.

Перш за все потрібно підготувати робочу директорію, яка міститиме програмний код для проєктів *ROS*. В прикладі це директорія “*~/ROS/catkin_ws*”. Далі завантажується *zip*-архів демонстраційної програми “*simple_ros_app.zip*”, що розміщено на хмарному ресурсі дисципліни. Вміст архіву, який був

наведений на рис. 2.5, потрібно розпакувати у директорію “*~/ROS/catkin_ws/src/simple_ros_app*”. Збирання проєкту і його налаштування відбувається через систему побудови проєктів *Catkin* наступними командами.

```
# Побудова проєкту
cd ~/ROS/catkin_ws
catkin_make
# Підключення розробленого проєкту (для постійної зміни внесіть у ".bashrc")
source ~/ROS/catkin_ws/devel/setup.bash
```

Після виконання наведених вище кроків, демонстраційна програма готова до запуску. Для цього слід використовувати розроблений файл запуску “*ros_mission.launch*”, який дозволяє одним викликом запустити всі компоненти програми (рис. 2.8). Ці компоненти включають:

- Програму автопілоту *PX4*;
- Симулятор *Gazebo*;
- Майстер процес *ROS*;
- Програма керування дроном засобами *MAVROS*.

```
<?xml version="1.0"?>
<launch>

    <!-- Include the MAVROS node with SITL and Gazebo -->
    <include file="$(find px4)/launch/mavros_posix_sitl.launch">
        <arg name="world" value="$(find simple_ros_app)/worlds/outdoor.world"/>
    </include>

    <!-- Our node to control the drone -->
    <node pkg="simple_ros_app" type="simple_mission_node.py" name="simple_mission_node" required="false" output="screen" />

</launch>
```

Рисунок 2.8 — Файл запуску “*ros_mission.launch*”

Запуск демонстраційної програми виконується наступною командою.

```
# Запуск програми ROS
roslaunch simple_ros_app ros_mission.launch
```

В результаті будуть запущені зазначені вище компоненти програми, після чого виконаються команди керування дроном (рис. 2.9-2.10).

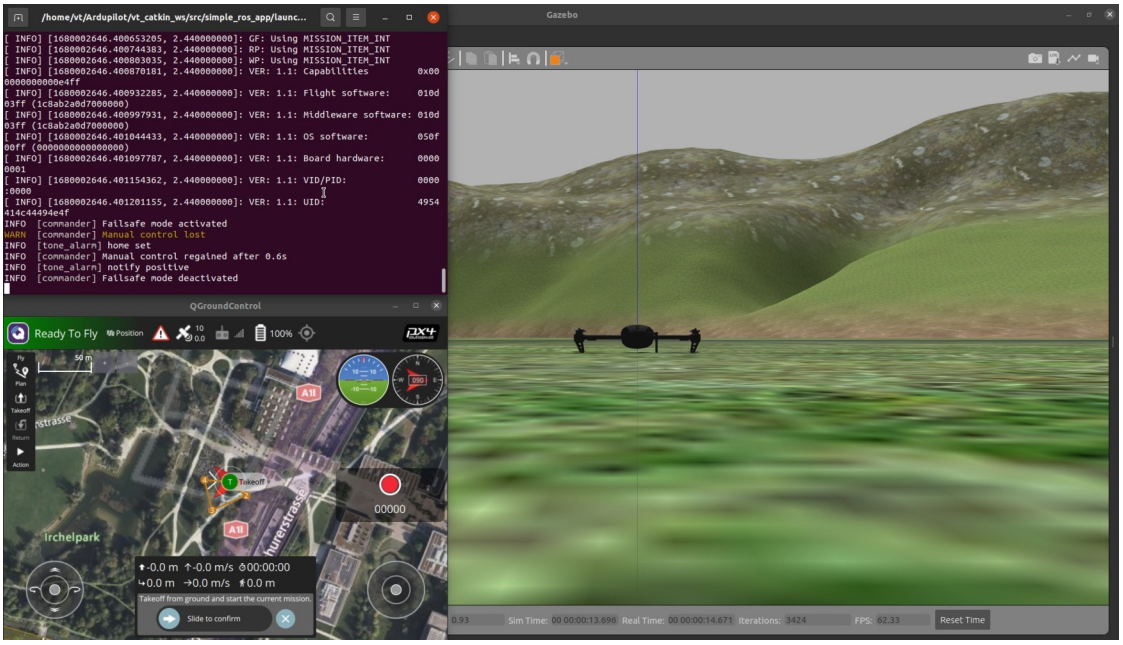


Рисунок 2.9 — Запуск компонентів демонстраційної програми *ROS*

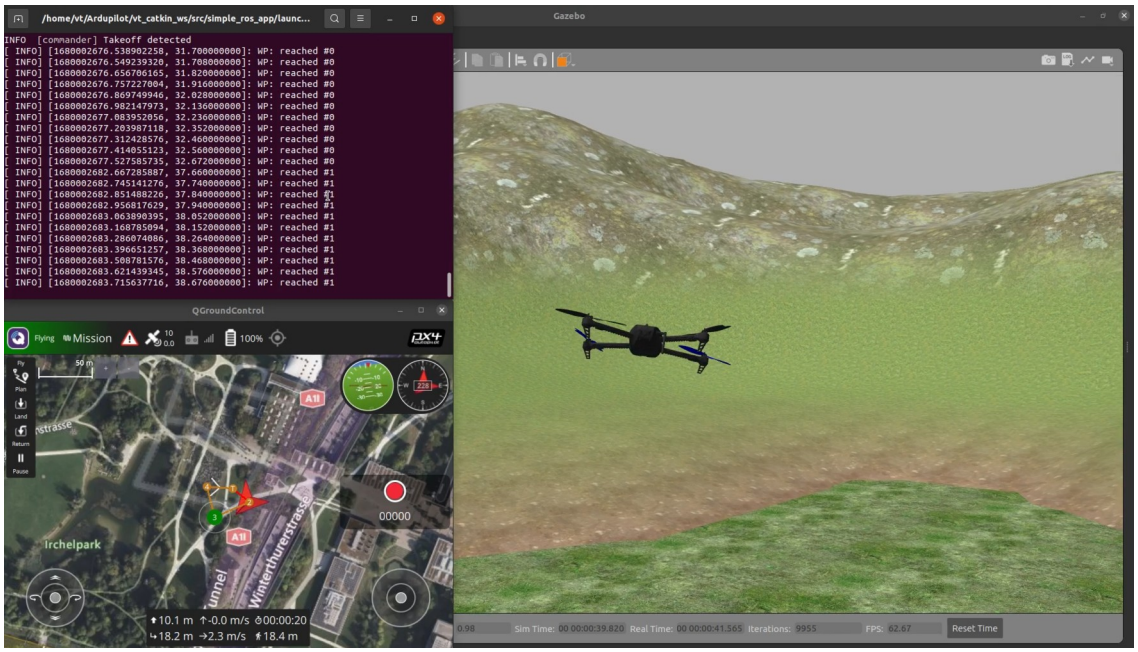


Рисунок 2.10 — Запуск політного плану автопілоту дрону

3 СЕРЕДОВИЩЕ *ROS2*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04;
- *ROS2 Humble* 2023-04-28;
- *PX4 Autopilot* 1.13.3;
- *Gazebo Simulator* 11.10.2;
- *QGroundControl* 4.2.4.

3.1 Різниця між *ROS2* та *ROS1*

З часу коли з’явилися перші версії *ROS* у 2007 році, відбулась велика кількість змін в робототехніці. Метою *ROS2* є адаптуватись до цих змін, використовуючи переваги *ROS,* в той же час вирішуючи проблеми, які присутні при роботі із *ROS1*. До однієї з таких проблем можна віднести централізовану архітектуру *ROS1*, в якій процеси *ROS — “nodes”* взаємодіють між собою через головний-майстер процес — “*roscore*”. Загальна концепція *ROS1* потребує постійного зв'язку із головним процесом для отримання інформації про інші процеси *ROS* в мережі та забезпечення можливості встановлення зв’язку між ними для обміну повідомленнями. Хоча подібна архітектура буде цілком задовільною, наприклад, для завдань розробки, лабораторних досліджень або закритих систем, коли канали зв’язку між процесами є стабільними, для задач із високою ймовірністю мережевих збоїв, це може призвести до відмови програм в середовищі *ROS1*. В ситуації втрати зв'язку із головним процесом, як результат, запущена програма продовжуватиме працювати і обмінюватись повідомленнями по вже відкритим мережевим з’єднанням, проте при спробі створити нове з’єднання виникатиме помилка, оскільки головний процес недоступний. Щоб вирішити подібну проблему, у *ROS2* використовується інша — децентралізована архітектура в якій кожен процес *ROS* може самостійно визначати наявність інших процесів у мережі для подальшої взаємодії.

3.1.1 Служба розповсюдження даних

Служба розповсюдження даних — *data distribution service* (*DDS*) це програмне забезпечення для спрощення взаємодії процесів із використання мережевих каналів. В ньому використовується концепція “публікація-підписка” для відправлення та отримання даних, подій та команд між процесами — “*nodes*”. Процес, який відправляє дані, створює тему та “публікує” повідомлення. Засобами служби розповсюдження даних повідомлення доставляються через мережу до інших процесів згідно тем до яких вони “підписані”.

У *ROS2* використовується служба розповсюдження даних для обміну повідомленнями між процесами, що дозволяє усунути потребу у використанні головного процесу для координації обміну даними. Наразі *ROS2* сумісний із декількома реалізаціями служби розповсюдження даних, а саме: e*Prosima’s Fast DDS, RTI’s Connext DDS, Eclipse Cyclone DDS* та *GurumNetworks GurumDDS.*

3.2 Налаштування *ROS2*

У прикладі використовується актуальна версія із тривалою підтримкою — *ROS2 Humble*. Згідно офіційної документації (https://docs.ros.org/en/humble/Installation.html) ця версія сумісна із версією ОС *Ubuntu* 22.04. Щоб встановити та налаштувати *ROS2* потрібно підключити сторонній репозиторій та завантажити відповідну версію пакету. Для цього виконуються наступні команди:

```
# Додавання репозиторію ROS2
sudo apt install curl
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg
echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(. /etc/os-release && echo $UBUNTU_CODENAME) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null
```

```
# Встановлення пакету ROS2 Humble
sudo apt update
sudo apt install ros-humble-desktop
# Додаткові бібліотеки
sudo apt install ros-humble-mavros ros-humble-mavros-msgs ros-humble-gazebo-ros ros-humble-gazebo-plugins python3-colcon-common-extensions
sudo /opt/ros/humble/lib/mavros/install_geographiclib_datasets.sh
# Оновлення змінних середовища (для постійної зміни внести у файл .bashrc)
source /opt/ros/humble/setup.bash
```

Аналогічно слід встановити симулятор *Gazebo* в разі його відсутності:

```
# Встановлення симулятору Gazebo
sudo apt install gazebo libgazebo-dev libopencv-dev
```

Після успішного завантаження та налаштування ПЗ можна перевірити роботу середовища *ROS2*. Для цього слід виконати команду “*ros2 topic list*”. Результат має бути аналогічним до рис. 3.1.

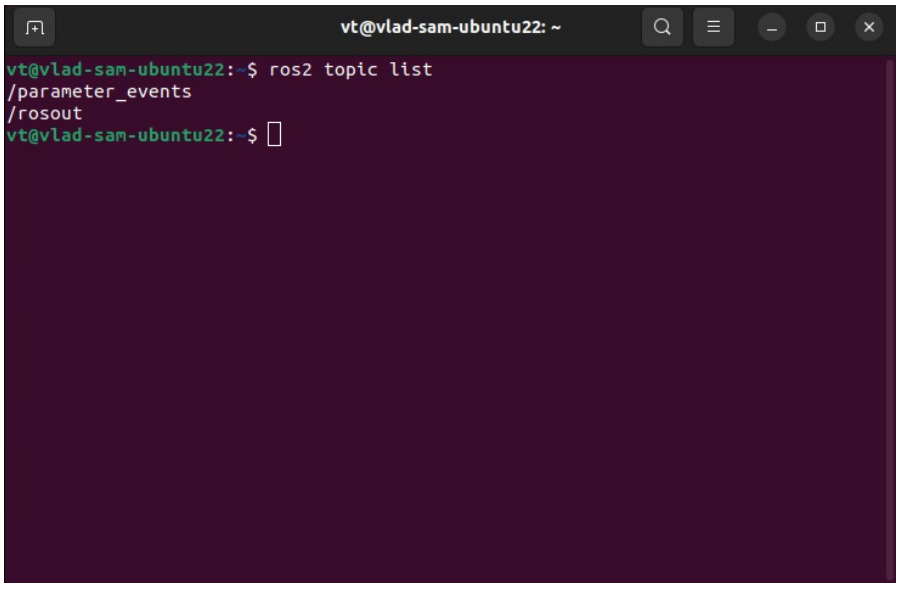


Рисунок 3.1 — Тестовий запуск середовища *ROS2*

3.3 Приклад програми *ROS2* для керування дроном

Щоб продемонструвати керування дроном на базі автопілоту *PX4*, було розроблено програму “*simple_ros2_app*” для *ROS2*. Обмін командами між дроном та програмою відбувається засобами протоколу *MAVLink*. Програмний проєкт має структуру наведену на рис. 3.2.

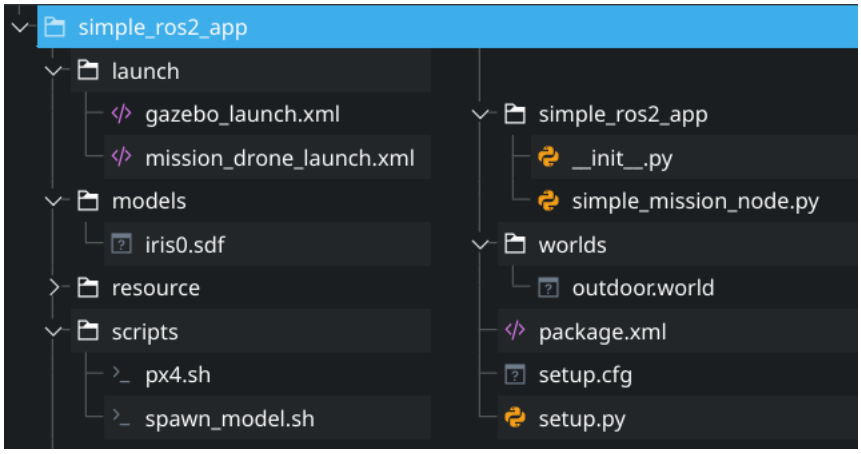


Рисунок 3.2 — Структура програмного проєкту *simple_ros2_app*

Ключовими компонентами наведеного проєкту *ROS2* є:

- Програмний код для процесу *ROS* — *node*. В прикладі код розроблений на мові програмування *Python*;
- Інтегрований файл запуску програми *ROS* — “*launch file*”, який дозволяє запустити програму *ROS* одним командним викликом;
- Файл моделі дрону для симулятора *Gazebo* — файл “*sdf*”, що описує параметри та функції віртуального дрону в симуляторі;
- Скрипти запуску — “*sh*” файли.

Загальна структура програми для процесу *ROS2* подібна до програми для *ROS1*, проте є деякі зміни порівняно із попередньою версією. Код процесу описується в окремому класі, а його ініціалізація виконується в “*main*” методі викликами клієнтської бібліотеки для відповідної мови програмування (рис. 3.3). Для мови Python — це “*rclpy*”. В “*__init__*” методі класу для процесу *ROS2* реєструються виклики процедур, які в ході роботи будуть обробляти повідомлення, публікувати теми, обробляти запити до сервісів, а також

виконувати певні керуючі функції (рис. 3.4). Реалізація керуючих функцій відбувається в окремих процедурах класу (рис. 3.5).

```python
def main(args=None):
    rclpy.init(args=args)

    mission_node = MissionNode()
    mission_node.get_logger().info('Node started')

    try:
        rclpy.spin(mission_node)
    except KeyboardInterrupt:
        pass
    finally:
        rclpy.try_shutdown()
        mission_node.destroy_node()
```

Рисунок 3.3 — Ініціалізація процесу *ROS2*

```python
class MissionNode(Node):

    def __init__(self):
        super().__init__("mission_node")

        # Subscription
        self.current_state = State()
        self.create_subscription(State, "/mavros/state", self.state_callback,
                                 qos.qos_profile_sensor_data)

        # Service clients
        arm_endpoint = "/mavros/cmd/arming"
        self.arming_service = self.create_client(CommandBool, arm_endpoint)
        while not self.arming_service.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting MAVROS service {}'.format(arm_endpoint))

        setmode_endpoint = "/mavros/set_mode"
        self.setmode_service = self.create_client(SetMode, setmode_endpoint)
        while not self.setmode_service.wait_for_service(timeout_sec=1.0):
            self.get_logger().info('Waiting MAVROS service {}'.format(setmode_endpoint))

        self.mode_msg = SetMode.Request()
        self.mode_msg.custom_mode = 'AUTO.MISSION'
        ...
        self.create_timer(5, self.start_mission_callback)
```

Рисунок 3.4 — Опис процесу *ROS2*

```python
def start_mission_callback(self):

    # Set "mission" mode
    if not self.mission_enabled:
        if self.setmode_async_call is None:
            if (self.get_clock().now().seconds_nanoseconds()[0] - self.last_request.seconds_nanoseconds()[0]) > 10:
                self.setmode_async_call = self.setmode_service.call_async(self.mode_msg)
                self.last_request = self.get_clock().now()
        else:
            if self.setmode_async_call.done():
                self.get_logger().info("Mission mode enabled")
                self.mission_enabled = True
    else:
        if not self.arm_done:
            if self.arming_async_call is None:
                if (self.get_clock().now().seconds_nanoseconds()[0] - self.last_request.seconds_nanoseconds()[0]) > 2:
                    self.arming_async_call = self.arming_service.call_async(self.arm_msg)
                    self.last_request = self.get_clock().now()
            else:
                if self.arming_async_call.done():
                    self.get_logger().info("Vehicle armed")
                    self.arm_done = True
```

Рисунок 3.5 — Реалізація функцій керування дроном

Основною функцією розробленої програми є встановлення для дрону режиму польоту “*mission*”, після чого виконання його запуску командою “*arm*”. Після виконання зазначених кроків, автопілот дрону надалі самостійно керує польотом згідно завантаженого через *QGroundControl* політного плану. Інтеграція *MAVLink* із *ROS* забезпечується бібліотекою *MAVROS*. Бібліотека дає можливість транслювати повідомлення *MAVLink — messages*, які генеруються ПЗ автопілоту у повідомлення *ROS*. Це дозволяє взаємодіяти із ПЗ дрону шляхом стандартного інтерфейсу повідомлень *ROS — topics*.

3.3.1 Інтеграція *ROS2* та *SITL* автопілоту *PX4*

В *ROS2* було внесені зміни у процес побудови пакетів через систему побудови проєктів *Colcon*, а також формат файлів запуску “*launch file*”. Через це на момент написання інтеграція *PX4* із *ROS2* потребувало внесення додаткових змін. В цьому проєкті запуск *SITL* автопілоту *PX4* забезпечується через окремий скрипт “*scipts/px4.sh*”. В ньому виконується запуск ПЗ автопілоту після чого очікується підключення до симулятору *Gazebo*, який в подальшому надаватиме телеметрію через *MAVLink* для віртуальної моделі дрону.

Скрипт “scipts/px4.sh”

```
1. #!/bin/bash
2. cd ~/.ros ~/ROS/PX4-Autopilot/build/px4_sitl_default/bin/px4 ~/ROS/PX4-Autopilot/build/px4_sitl_default/etc -s etc/init.d-posix/rcS -i 0 -w sitl_iris_0
3. exit 0
```

Також для подальшого запуску моделі дрону у симуляторі *Gazebo* потрібно окремо згенерувати файли моделі “*sdf*”. Для цього слід запустити програму, яка присутня у ПЗ *PX4*. В результаті виконання буде згенеровано файл моделі дрону для симулятору *Gazebo* з відповідними параметрами. Приклад виконання наведено нижче.

```
# Генерування SDF файлу
cd ~/ROS/PX4-Autopilot/Tools/sitl_gazebo/scripts
python3 jinja_gen.py --mavlink_id=3 --mavlink_udp_port=14560 --
mavlink_tcp_port=4560 --gst_udp_port=5600 --video_uri=5600 --
mavlink_cam_udp_port=14530 ../models/iris/iris.sdf.jinja ..
--output-file=~/ROS/iris0.sdf
# Підключення додаткових бібліотек до Gazebo (оновити .bashrc)
export GAZEBO_PLUGIN_PATH=~/ROS/PX4-Autopilot/build/px4_sitl_default/build_gazebo
export GAZEBO_MODEL_PATH=~/ROS/PX4-Autopilot/Tools/sitl_gazebo/models
```

3.3.2 Інтеграція *ROS2* та симулятору *Gazebo*

Аналогічно відбувається налаштування симулятору *Gazebo* для роботи в *ROS2*. Через оновлений формат файлів запуску “*launch file*” на момент написання *Gazebo* повністю не підтримував оновлений формат через що застарілі файли запуску не працювали. Вирішення цієї проблеми забезпечується окремим скриптом “*scripts/spawn_model.sh*”, який завантажує модель дрону у середовище *Gazebo*. В якості аргументу до цього скрипту передається згенерований файл моделі дрону.

```
Скрипт “scripts/spawn_model.sh”
#!/bin/bash
gz model --spawn-file $1.sdf --model-name $1 -x 0 -y 0 -z 0 -R 0 -P 0 -Y 0
exit 0
```

Також для запуску середовища симулятору *Gazebo* використовується окремий файл запуску “*gazebo_launch.xml*” (рис. 3.6).

```
<?xml version="1.0"?>
<launch>

    <include file="$(find-pkg-share gazebo_ros)/launch/gzserver.launch.py">
        <arg name="world" value="$(find-pkg-share simple_ros2_app)/worlds/outdoor.world" />
    </include>

    <include file="$(find-pkg-share gazebo_ros)/launch/gzclient.launch.py" />

</launch>
```

Рисунок 3.6 — Файл запуску симулятору *Gazebo*

3.3.3 Підготовка робочої директорії та побудова проєкту

Для побудови проєкту потрібно підготувати робочу директорію, яка міститиме програмний код для проєктів *ROS2*. В прикладі це директорія “*~/ROS/ros2_ws*”. Далі завантажується *zip*-архів демонстраційної програми “*simple_ros2_app.zip*”, що розміщено на хмарному ресурсі дисципліни. Вміст архіву, який було наведений на рис. 3.2, потрібно розпакувати у директорію “*~/ROS/ros2_ws/src/simple_ros2_app*”. Збирання проєкту і його налаштування відбувається через систему побудови проєктів *Colcon* наступними командами.

```
    # Побудова проєкту
1.  cd ~/ROS/ros2_ws/
2.  colcon build
    # Оновлення змінних середовища для проєкту
3.  source ~/ROS/ros2_ws/install/local_setup.bash
```

В ході побудови проєкту може з’явитись попередження “*setup.py install is deprecated*”. Це пов’язано з тим, що на момент написання *ROS2* повністю не підтримував нову систему встановлення пакетів у *Python* 3.10. Дане попередження не впливає на подальшу роботу, тому його можна проігнорувати.

Після побудови проєкту демонстраційна програма готова до запуску. Процес запуску розділений на два файли запуску. Перший файл “*gazebo_launch.xml*” запускає симулятор *Gazebo*, а другий файл “*mission_drone_launch.xml*” дозволяє запустити всі складові програми (рис. 3.7). Ці складові включають:

- завантаження моделі дрону у *Gazebo* — “*spawn_model.sh*”;

- запуск автопілоту *PX4 — “px4.sh”*;
- процес трансляції повідомлень *MAVLink* у *ROS — MAVROS*;
- програма керування дроном засобами *MAVROS*.

Рисунок 3.7 — Основний файл запуску програми керування

3.3.4 Запуск програми керування

Запуск розробленої програми керування виконується наступними командами.

```
# Термінал 1
ros2 launch simple_ros2_app gazebo_launch.xml
# Термінал 2
ros2 launch simple_ros2_app mission_drone_launch.xml
```

В результаті буде запущені зазначені вище компоненти програми, після чого виконаються команди керування дроном (рис. 3.8-3.9).

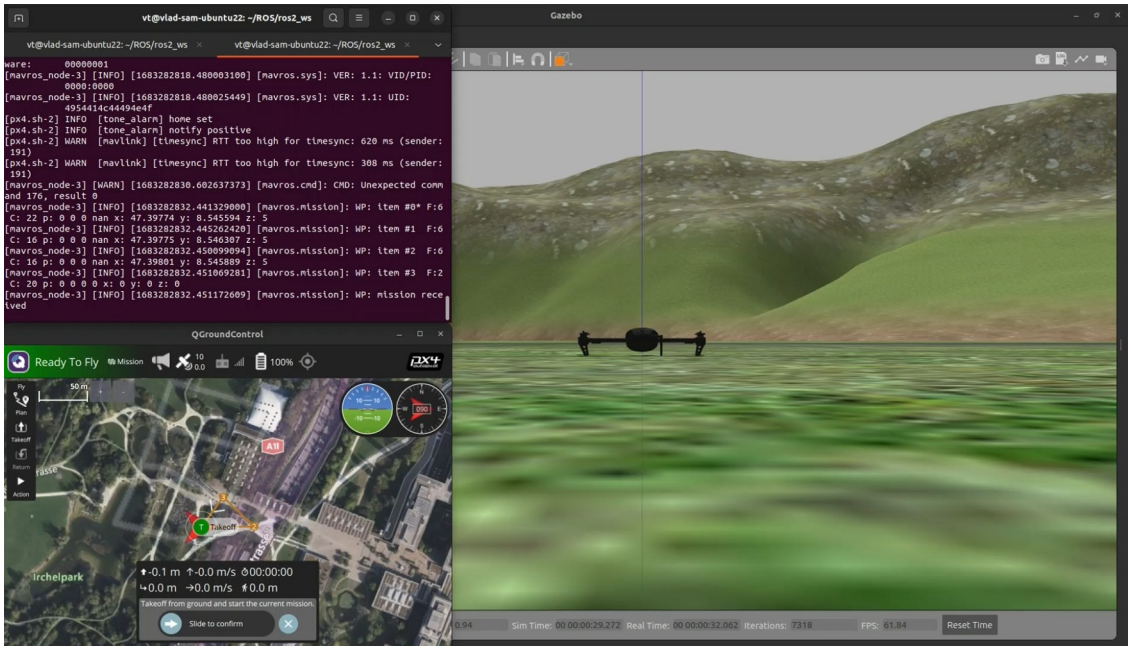


Рисунок 3.8 — Запуск компонентів демонстраційної програми *ROS2*

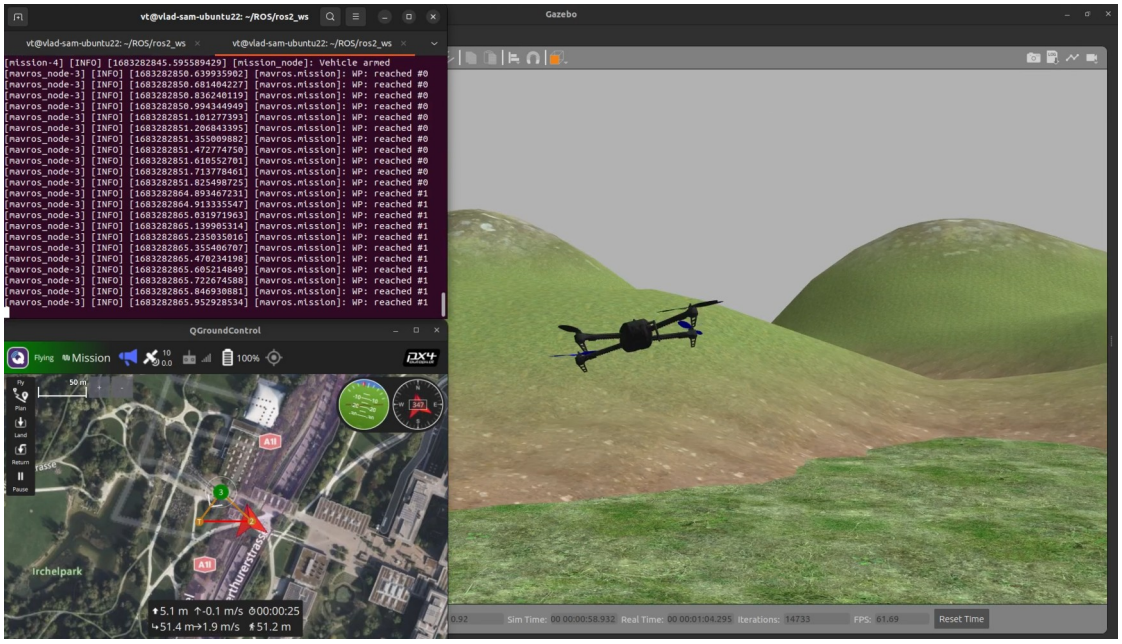


Рисунок 3.9 — Запуск політного плану автопілоту дрону

4 КЕРУВАННЯ ДЕКІЛЬКОМА ЕКЗЕМПЛЯРАМИ ДРОНІВ ІЗ ВИКОРИСТАННЯМ *ROS2*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04;
- *ROS2 Humble* 2023-04-28;
- *PX4 Autopilot* 1.13.3;
- *Gazebo Simulator* 11.10.2;
- *QGroundControl* 4.2.4.

4.1 Режим програмного керування в автопілоті *PX4*

Для ситуацій, коли необхідно надавати зовнішні команди керування, в автопілоті *PX4* передбачений окремий режим польоту “*offboard*”. Цей режим дозволяє передавити координати або вектори переміщення по протоколу *MAVLink* напряму в контролер автопілот. Зазвичай такі команди передаються програмним шляхом з додаткового комп’ютеру “компаньйону” — “*companion computer*”, який розміщується на борту дрону і підключається до контролеру через послідовну шину. Для успішної роботи в описаному режимі автопілот *PX4* вимагає отримання потоку команд керування до переходу в режим “*offboard*” та запуску “*arm*”. Потік команд має надсилатись із частотою не менше 2*Hz*. В разі відсутності команд керування в польоті, автопілот виходить з режиму “*offboard*” і переходить в режим стабілізації.

4.2 Сервіси *ROS*

Для обміну між собою інформацією процеси *ROS* — “*nodes*” використовують концепцію “публікація-підписка”. Це дозволяє організувати схему взаємодії “один-до-всіх” або “всі-до-всіх”. Хоча подібна схема є досить гнучкою для обміну повідомленнями, вона не дає можливості організувати

взаємодію “запит-відповідь”, яка притаманна для віддаленого виклику процедур — *RPC*. Дана взаємодія часто використовується у розподілених системах і в *ROS* забезпечується за рахунок механізму так званих сервісів — “*ROS services*”. Сервіси *ROS* визначають два типи повідомлень: один для “запиту” і один для “відповіді”. Аналогічно до тем повідомлень — “*topics*” процеси *ROS* можуть визначати сервіси — “*service*” у вигляді певного імені. Для взаємодії клієнт робить виклик до відповідного сервісу, відправляючи повідомлення і очікуючи відповідь. Подібна схема може бути використана в разі, коли викликаючій стороні важливо отримати результат або статус виконання певної віддаленої процедури.

4.3 Схема керування декількома дронами у *ROS*

Щоб керувати декількома дронами засобами *ROS*, розглянемо схему відповідно до якої буде організовано взаємодію між всіма компонентами. По-перше, слід забезпечити можливість відправляти команди керування дроном незалежно від наявності зв’язку із оператором. При такій схемі програмне забезпечення дрону, що включатиме в себе окремий процес *ROS* — “*node*” зможе працювати автономно по певній завчасно закладеній програмі. По-друге, щоб оператор міг централізовано керувати або оновлювати політні команди для всіх дронів, також слід доповнити описану схему ще одним центральним процесом *ROS*, який об’єднуватиме керування всіма дронами. При цьому даний центральний процес буде забезпечувати можливість налаштувати початкові параметри дрону, наприклад, координати польоту, але в подальшому програмна частина на борту дрону зможе самостійно керувати польотом і не потребуватиме зв’язку із центральним компонентом.

Для ілюстрації описаної схеми, розглянемо рисунок 4.1. На ньому візуально показано компоненти системи керування та їх логічне розташування. В прикладі маємо три дрони (*uav0*-*uav*2), кожен з них має політний контролер на базі автопілоту *PX4*; так званий комп’ютер “компаньйон”, на якому

працюватиме програмне забезпечення *ROS*; та умовно центральний компонент, з якого можна відправити польотні координати для кожного дрону.

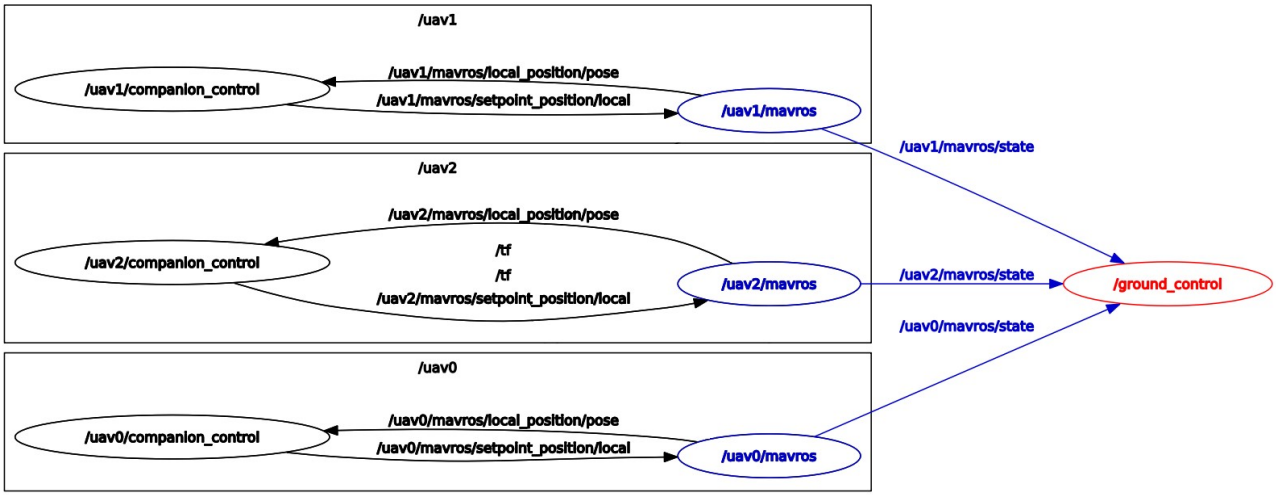


Рисунок 4.1 — Схема керування трьома дронами

Для кожної зазначеної складової схеми реалізується свій окремий процес *ROS*, що виконує відповідні функції. Так, комунікація між автопілотом *PX4* та *ROS* забезпечується за рахунок бібліотеки *MAVROS* та запущеного однойменного процесу *ROS*. Процес — “*companion_control*” відповідає за надсилання політних команд для режиму “*offboard*” в автопілот дрону. Центральний процес — “*ground_control*” дозволяє одночасно відправити політні координати всім комп’ютерам “компаньйонам”, що безпосередньо розташовані на дронах. В подальшому використання цього центрального процесу не є обов’язковим для польоту. Комунікація процесів “*ground_control*” та “*companion_control*” відбувається через механізм сервісів *ROS* — “*services*”.

4.4 Приклад програми *ROS2* для керування декількома дронами

Щоб продемонструвати керування дронами по описаній вище схемі, базову програму “*simple_ros2_app*” було розширено, а також доповнено реалізаціями відповідних процесів *ROS* для керування на кожному логічному рівні. Програмний проєкт має структуру наведену на рис. 4.2.

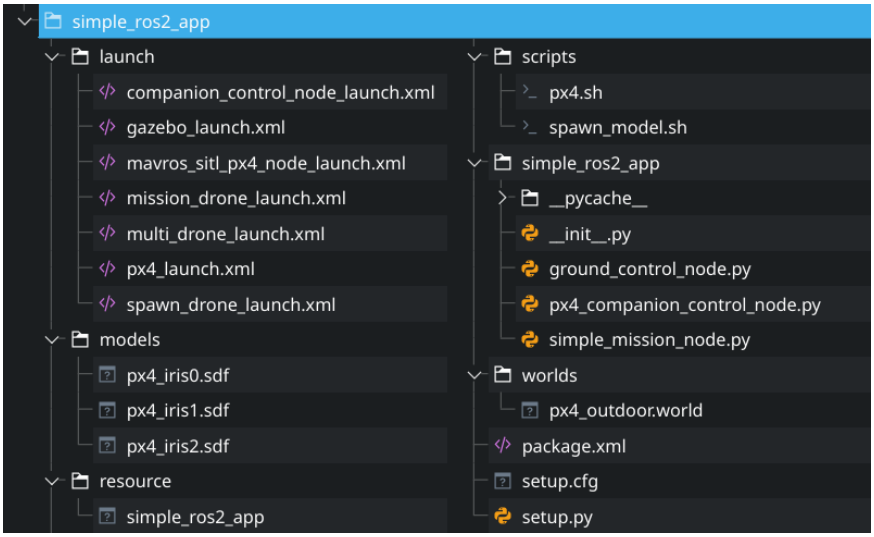


Рисунок 4.2 — Структура програмного проєкту “*simple_ros2_app*” для керування дронами

Ключовими доповненнями в проєкті є:

• Програмний код для нових процесів *ROS* — “*ground_control_node*” та “*px4_companion_control_node*”;

• Додаткові інтегровані файли запуску *ROS* — “*launch file*” для нових процесів *ROS* та загальний файл запуску всієї симуляції — “*multi_drone_launch.xml*”;

• Додаткові файли моделей дронів для симулятора *Gazebo* — файли “*sdf*”, що описують нові параметри та функції двох додаткових віртуальних дронів в симуляторі.

4.4.1 Створення власного типу повідомлення для сервісу

Інструментарій *ROS* дозволяє створювати власні типи повідомлень - “*messages*” для обміну даними між процесами. Тому для взаємодії розроблених процесів *ROS* через механізм сервісів було введено окремий тип повідомлення, що включає в себе інформацію про координати польоту і відмітку часу. Зазначений формат створюється за рахунок його опису у спеціальному файлі інтерфейсу “*msg*” або “*srv*”, який в подальшому використовуватиметься у

повідомленнях або сервісах відповідно. Для того, щоб створити власний тип повідомлення потрібно створити окремий проєкт, в якому описуються необхідні формати власних повідомлень. При описі за основу можна використовувати вбудовані типи повідомлень. При побудові такого проєкту засобами *ROS* будуть автоматично створені реалізації описаних типів повідомлень під необхідні мови програмування, наприклад, *C++*, *Python*, тощо. Приклад проєкту для створення власного типу повідомлення для сервісу наведено на рис. 4.3.

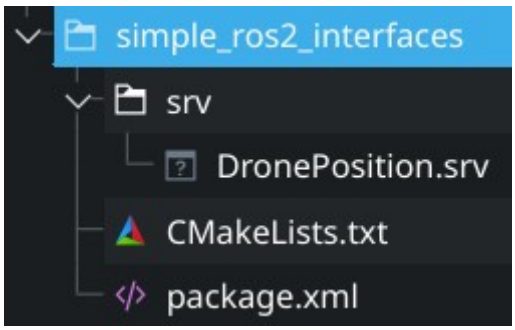


Рисунок 4.3 — Проєкт для створення власного типу повідомлення для сервісу

В проєкті “*simple_ros2_interfaces*” ключовим є файл опису інтерфейсу повідомлення для сервісу — “*DronePosition.srv*”. В ньому описуються поля, які міститиме наше повідомлення. Формат цього файлу є доволі простим. Спочатку вказується тип поля, наприклад, “*builtin_interfaces/Time*”, “*float64*”, “*bool*”, а потім зазначається назва цього поля. При цьому у повідомлення для сервісів спочатку зазначаються поля для запиту, а в кінці через “---” вказуються поля для відповіді. Приклад файлу наведений нижче.

```
    # DronePosition.srv
1.  builtin_interfaces/Time stamp
2.  float64 x
3.  float64 y
4.  float64 z
    ---
5.  bool success
```

В подальшому цей проєкт додається до директорії куди раніше було скопійовано проєкт “*simple_ros2_app*”.

4.4.2 Реалізація програмних компонентів для керування дронами

Згідно описаної на рис. 4.1 схеми керування дронами проєкт було доповнено двома процесами, які дозволяються керувати дроном на кожному логічному рівні. Основний керуючий код для процесу “*px4_companion_control_node*” наведений на рис. 4.4. та 4.5.

```
105     def send_command_callback(self):
106
107         if self.fly_cmd:
108
109             if self.get_flight_mode() != self.offboard_mode and self.setmode_async_call is None and (
110                     self.get_clock().now().seconds_nanoseconds()[0] -
111                     self.last_request.seconds_nanoseconds()[0]) > 2:
112                 # self.get_logger().info("Try offboarding")
113                 self.setmode_async_call = self.set_flight_mode(self.offboard_mode)
114
115                 self.last_request = self.get_clock().now()
116
117             if self.setmode_async_call is not None and self.setmode_async_call.done():
118                 self.get_logger().info("Flight mode enabled")
119                 self.setmode_async_call = None
120
121             if not self.is_armed() and self.arming_async_call is None and (
122                     self.get_clock().now().seconds_nanoseconds()[0] -
123                     self.last_request.seconds_nanoseconds()[0]) > 2:
124                 # self.get_logger().info("Try arming")
125                 self.arming_async_call = self.arm()
126
127                 self.last_request = self.get_clock().now()
128
129             if self.arming_async_call is not None and self.arming_async_call.done():
130                 self.get_logger().info("Vehicle armed")
131                 self.arming_async_call = None
```

Рисунок 4.4 — Фрагмент коду для запуску дрону

Управління дроном можна умовно поділити на два етапи. На першому етапі (рис. 4.4), виконується перемикання автопілоту в режим польоту “*offboard*” і виконується зліт командою “*arm*”. На другому етапі (рис. 4.5), зразу після зльоту, починається передача політних команд до автопілоту. В даній реалізації це два типи команд:

- *mavros/setpoint_position/local* — політні координати; (аналог команди *MAVLink — SET_POSITION_TARGET_LOCAL_NED*)
- *mavros/setpoint_raw/local* — вектор польоту;

В якості тестового польоту, дрон спочатку слідує до зазначених координат, а потім починає обліт області за запрограмованим вектором у вигляді “∞”.

```
            if not self.fly_position_reached:
                if self.is_on_position(self.fly_position[0], self.fly_position[1], self.fly_position[2]):
                    self.fly_position_reached = True
                    self.get_logger().info("Start position reached. Flying")
                    self.yaw = 6.28
                self.goto_position(self.fly_position[0], self.fly_position[1], self.fly_position[2])
            else:
                self.goto_vector(10.0, 0.0, 0.0, self.yaw)
                self.yaw += self.yaw_d
                if self.yaw < 0 or self.yaw > 6.28:
                    self.yaw_d *= -1
                    self.yaw += self.yaw_d
        else:
            if self.is_armed():
                if not self.fly_position_reached:
                    if self.is_on_position(self.fly_position[0], self.fly_position[1], self.fly_position[2]):
                        self.fly_position_reached = True
                        self.get_logger().info("Destination position reached. Landing")
                    self.goto_position(self.fly_position[0], self.fly_position[1], self.fly_position[2])
```

Рисунок 4.5 — Фрагмент коду із командами керування дроном

Для подальшої взаємодії дрону із керуючим процесом “*ground_control_node*” під час ініціалізації процесу “*px4_companion_control_node*” створюються два сервіси:

- *companion_control/start_flight* — початок польоту до заданих координат;
- *companion_control/stop_flight* — повернення до заданих координат та посадка дрону;

Фрагмент коду із описом зазначених сервісів наведено на рис. 4.6.

```
    def __init__(self):

        super().__init__("companion_control")
        self.drone_name = self.get_namespace()

        # Service servers
        self.create_service(DronePosition, '{}/companion_control/start_flight'.format(self.drone_name),
                            self.fly_to_position)
        self.create_service(DronePosition, '{}/companion_control/stop_flight'.format(self.drone_name),
                            self.return_to_position)
        # Subscriptions
```

Рисунок 4.6 — Опис сервісів *ROS* для керування дроном

Реалізація процесу “*ground_control_node*” включає в себе функцію одночасного запуску всіх дронів і їх політ до заданих координат. Для забезпечення цього функціоналу використовується створений сервіс

“*companion_control/start_flight*”. Фрагмент коду із реалізацією наведено на рис. 4.7.

```python
class GroundControlNode(Node):

    drones = ['/uav0', '/uav1', '/uav2']

    drone_position = [[10.0, 0.0, 10.0],
                      [0.0, 10.0, 15.0],
                      [7.0, 7.0, 20.0]]

    fly_service = []

    def __init__(self):
        super().__init__("ground_control")

        for drone in self.drones:
            # Service clients
            drone_endpoint = "{}/companion_control/start_flight".format(drone)
            drone_service = self.create_client(DronePosition, drone_endpoint)
            while not drone_service.wait_for_service(timeout_sec=1.0):
                self.get_logger().info('Waiting drone service {}'.format(drone_endpoint))
            self.fly_service.append(drone_service)

    def start_all_drones(self):
        fly_msg = DronePosition.Request()
        for i in range(len(self.drones)):
            fly_msg.x = self.drone_position[i][0]
            fly_msg.y = self.drone_position[i][1]
```

Рисунок 4.7 — Фрагмент коду реалізації “*ground_control_node*”

Окремо слід відмітити, що перевагою використання механізму сервісів *ROS* є можливість їх виклику поза межами програмного коду. Для цього у *ROS* є окрема утиліта, яка дозволяє виконувати запит до сервісів із командного рядку. Приклад такого виклику для реалізованого сервісу наведено нижче.

```
# Виклик до сервісу

ros2 service call /uav0/companion_control/start_flight
simple_ros2_interfaces/srv/DronePosition "{'x': 0, 'y': 0, 'z': 5}"
```

Доповнення, що стосуються інтегрованих файлів запуску *ROS*, включають в себе виділені в окремі файли запуски відповідних компонентів:

- *px4_launch.xml* — запуск *SITL PX4*;
- *mavros_sitl_px4_node_launch.xml* — запуск процесу *MAVROS* для комунікації з контролером автопілотом;
- *companion_control_node_launch.xml* — запуск процесу керування дроном;

• *spawn_drone_launch.xml* — інтегрований запуск зазначених вище компонентів;

• *multi_drone_launch.xml* — інтегрований запуск зазначених вище компонентів для трьох симуляцій дронів (рис. 4.8).

```xml
<?xml version="1.0"?>
<launch>

    <!-- Drone entity 1 -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/spawn_drone_launch.xml">
        <arg name="instance" value="0"/>
    </include>

    <!-- Drone entity 2 -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/spawn_drone_launch.xml">
        <arg name="instance" value="1"/>
    </include>

    <!-- Drone entity 3 -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/spawn_drone_launch.xml">
        <arg name="instance" value="2"/>
    </include>

    <!-- Ground control -->
    <node pkg="simple_ros2_app" exec="ground_control" output="screen"/>

</launch>
```

Рисунок 4.8 — Файл запуску для розробленої програми керування декількома дронами

4.4.3 Запуск програми керування для симуляції кількох дронів

Запуск розробленого прикладу програми керування виконується наступними командами.

```
# Термінал 1
ros2 launch simple_ros2_app gazebo_launch.xml
# Термінал 2
ros2 launch simple_ros2_app multi_drone_launch.xml
```

В результаті буде запущено зазначені вище компоненти програми, після чого виконаються команди керування дроном (рис. 4.9-4.10). Також на рис. 4.11 показано виклик до сервісу “*companion_control/stop_flight*” всіх дронів через термінал.

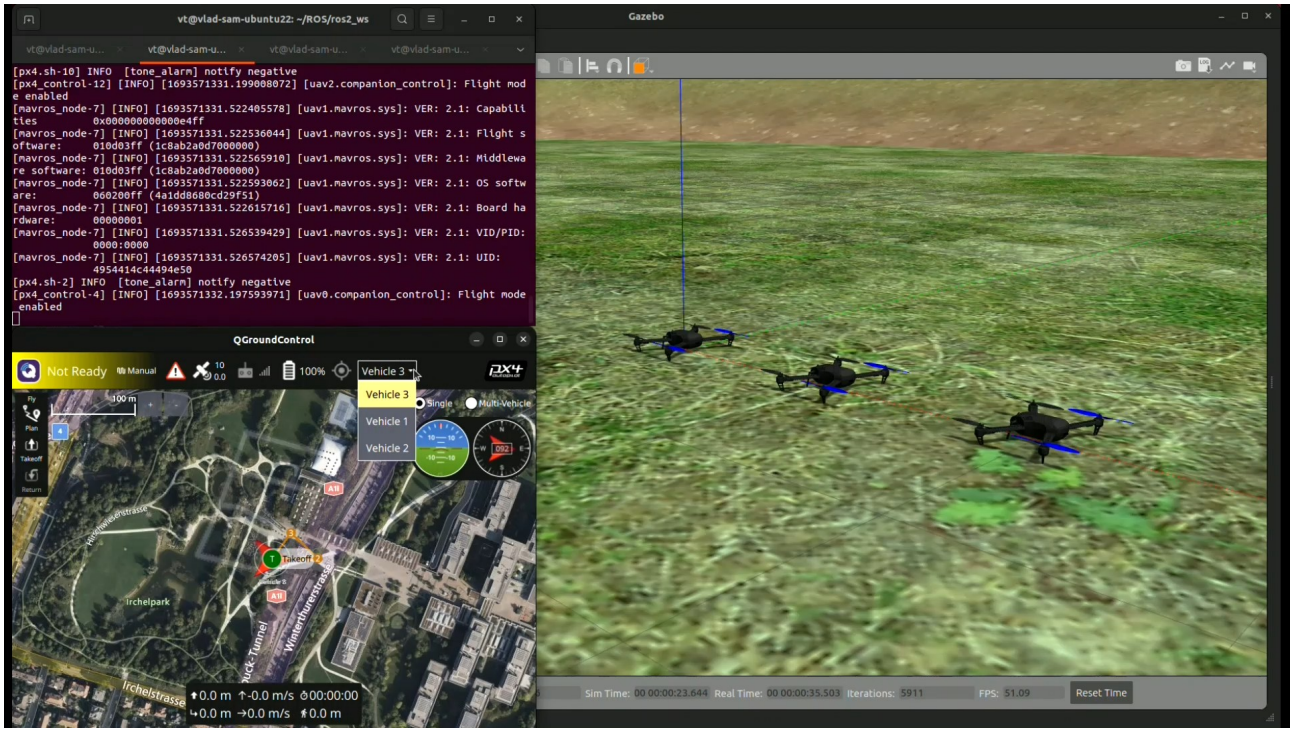


Рисунок 4.9 — Запуск декількох дронів

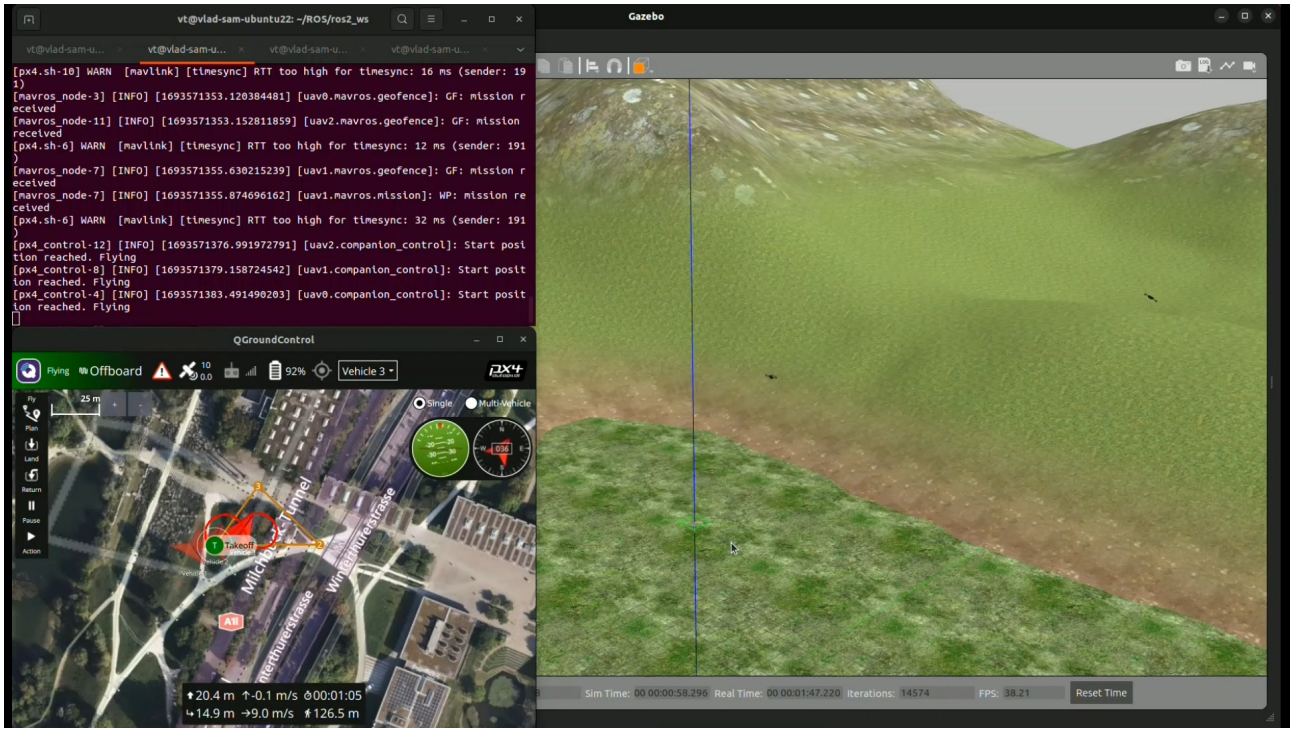


Рисунок 4.10 — Керування польотом декількох дронів

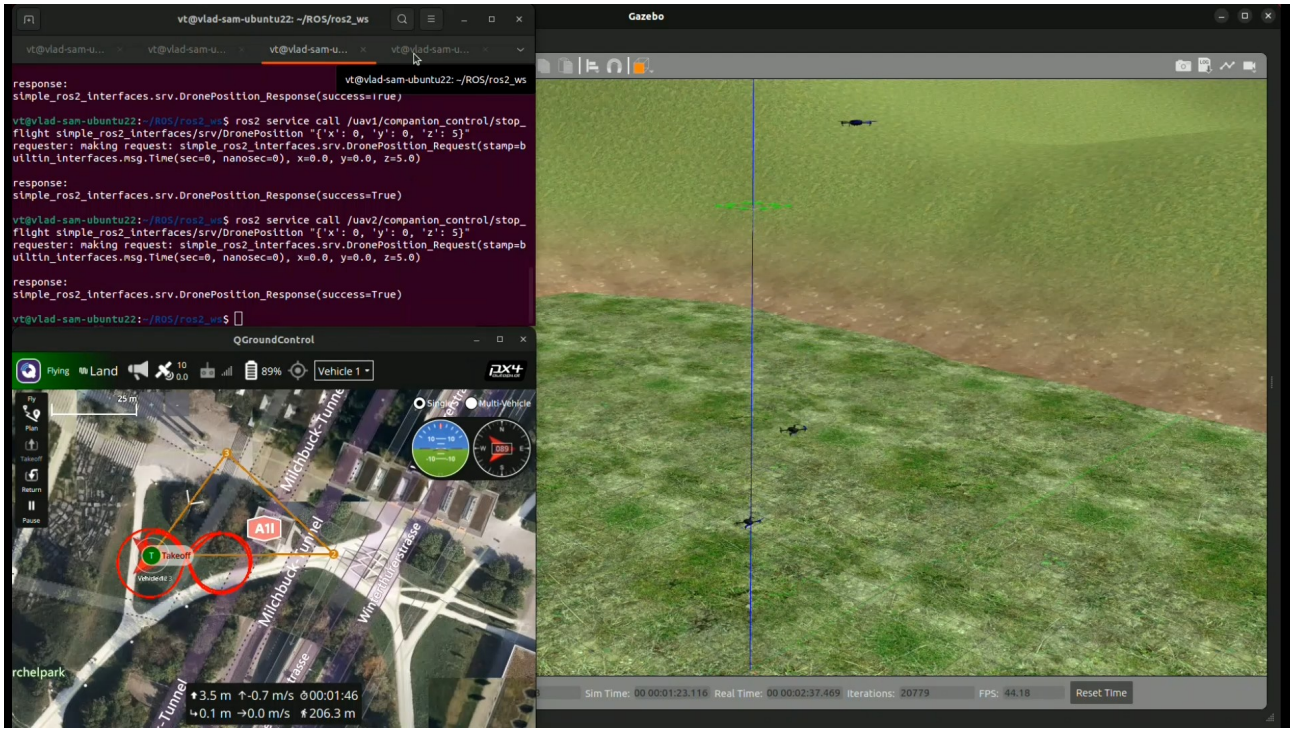


Рисунок 4.11 — Виклик до сервісу “*companion_control/stop_flight*” всіх дронів через термінал

5 ОБРОБКА ВІДЕО ПОТОКУ КАМЕРИ ВІРТУАЛЬНОГО ДРОНУ ІЗ СИМУЛЯТОРА *GAZEBO*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04;
- *ROS2 Humble* 2024-02-17;
- *PX4 Autopilot* 1.13.3;
- *Gazebo Simulator* 11.10.2;
- *QGroundControl* 4.4.3.

5.1 Налаштування робочого середовища *Gazebo* та *ROS*

У програмному забезпечені *Gazebo* окрім самої симуляції польоту дрону у віртуальному середовищі є засоби, щоб захоплювати зображення цього віртуального світу для його подальшої обробки різними методами, наприклад, для задач комп’ютерного зору. Це є зручним, оскільки є можливість зразу протестувати всі аспекти розроблюваної роботизованої системи. В цьому прикладі ми розглянемо можливість додати до нашого віртуального дрону камеру, яка буде видавати відео потік із віртуального оточення в якому переміщується дрон. Для початку потрібно перевірити наявність додаткових програмних бібліотек і в разі їх відсутності встановити їх. Для цього можна використати наступні команди.

```
# Встановлення GStreamer (в разі його відсутності)
sudo apt install libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev libgstreamer-plugins-bad1.0-dev gstreamer1.0-plugins-base gstreamer1.0-plugins-good gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-libav gstreamer1.0-tools gstreamer1.0-x gstreamer1.0-alsa gstreamer1.0-gl gstreamer1.0-gtk3 gstreamer1.0-qt5 gstreamer1.0-pulseaudio
# Встановлення плагіну Gazebo для ROS (в разі його відсутності)
sudo apt install ros-humble-gazebo-plugins
```

Програма *GStreamer* дозволяє створювати та відправляти відео потік за протоколом *UDP.* Другий плагін *ROS* в подальшому дасть можливість симулятору *Gazebo* додатково відправляти відео потік у вигляді повідомлень *ROS.*

5.2 Доповнення файлів моделей дронів віртуальною камерою

Ключові зміни для додавання віртуальної камери до дрону вносяться у відповідний файл моделі дрону. За основу можна взяти готовий файл моделі дрону із попередніх прикладів коду. В цьому прикладі за основу візьмемо модель “*px4_iris0.sdf*” і вкінці доповнимо його фрагментом опису об’єкту камери. Основними логічними блоками в даному описі є:

- *<link name='camera_link'>* - блок верхнього рівня, який містить описи всіх компонентів камери і відповідає за об’єднання елементу камери із корпусом дрону;
- *<sensor name='camera' type='camera'>* - безпосередньо описує параметри самої віртуальної камери;
- *<plugin name="camera_controller" filename="libgazebo_ros_camera.so">* - дозволяє транслювати зображення з віртуальної камери у форматі повідомлень ROS, що дозволяє зчитувати ці дані стандартними методами із процесів ROS;
- *<plugin name="GstCameraPlugin" filename="libgazebo_gst_camera_plugin.so">* - ще одне доповнення, яке дозволяє симулятору *Gazebo* транслювати відео потік за протоколом *UDP.* Це дає можливість переглядати зображення у програмі керування *QGroundControl.*

Після внесення наведених вище змін, можна протестувати новий файл моделі дрону з камерою. В цьому прикладі файли моделей дронів з камерою у своїй назві в кінці містять слово “*_cam*”. Доповнені моделі дронів розміщуються на хмарному сховищі дисципліни.

5.3 Тестування камери віртуального дрону

Запуск середовища симуляції дронів є аналогічним до минулих прикладів, але тепер замість звичайної моделі віртуального дрону слід використовувати файл моделі дрону з камерою. Після запуску всіх елементів симулятору можна перевірити, що відео потік з камери надсилається і його можна зчитати. Для цього можна використати наступні програми:

- *QGroundControl* — для відображення відео потоку за протоколом *UDP* (рис. 5.1) або напряму через *GStreamer*;
- *RViz2* — для відображення відео потоку, що надсилається у форматі повідомлень *ROS* (рис. 5.2).

Окремо слід відмітити, що для активації функції відображення відео потоку з дрону у *QGroundControl* потрібно внести деякі зміни у параметри програми. Для цього у лівому верхньому куті програми необхідно натиснути на її логотип. Далі у вікні “*Select Tool*”, що відкрилось, обирається “*Application Settings*”. У секції “*Fly View*” в елементі “*Video Settings*” потрібно для параметру “*Source*” обрати “*UDP h.264 Video Stream*” (рис. 5.3). Після цього у лівому нижньому куті на головному вікні програми з’явиться вікно “*Waiting for Video*”, що в подальшому відображатиме відео з камери дрону.

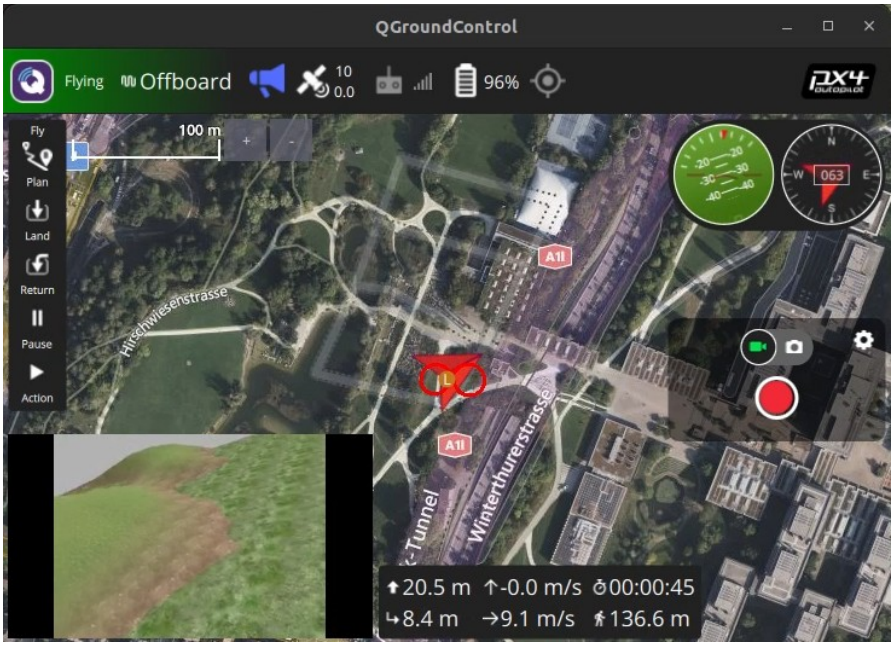


Рисунок 5.1 — Відображення відео з камери дрону у *QGroundControl*

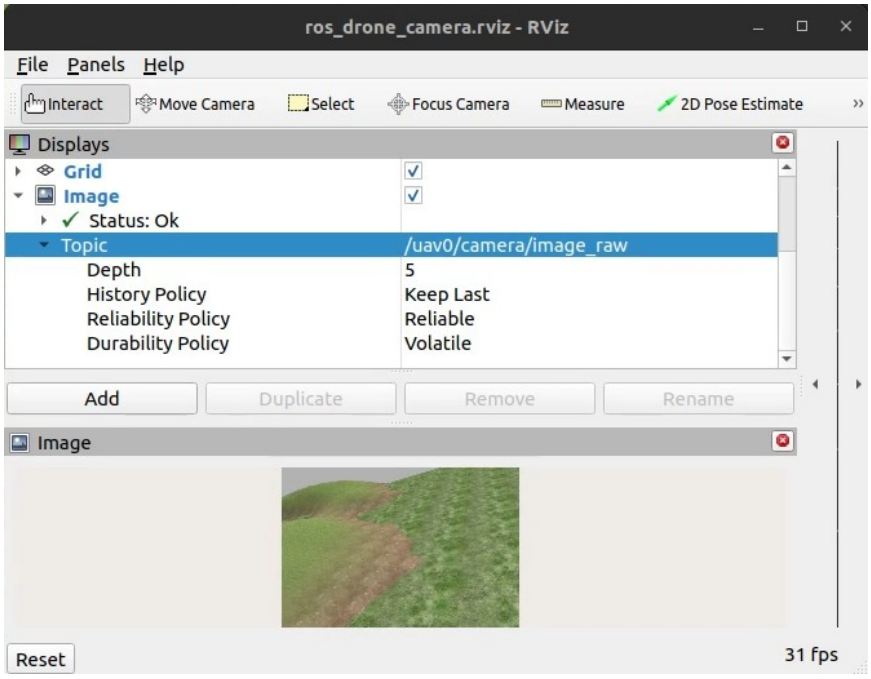


Рисунок 5.2 — Відображення відео з камери дрону у *RViz2*

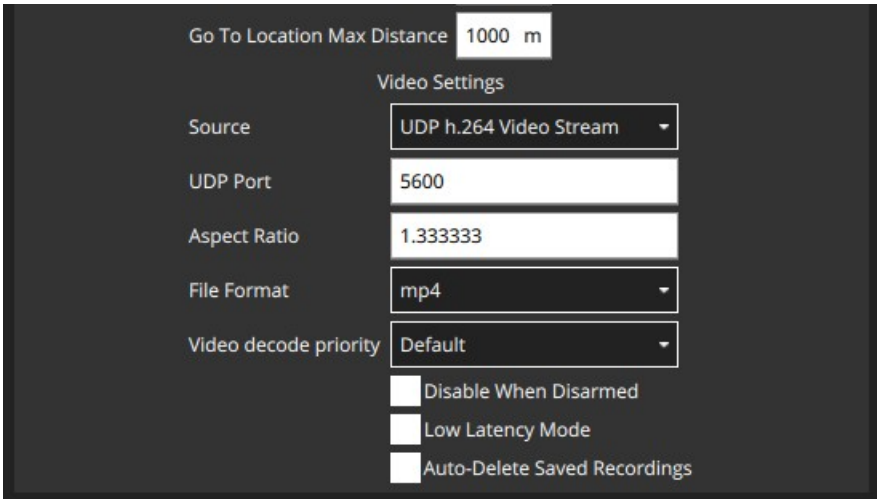


Рисунок 5.3 — Налаштування *QGroundControl* для відображення відео з дрону

В разі якщо для відображення відео з дрону використовується *RViz2*, його можна запустити через окремий термінал. Для цього слід виконати команду нижче:

```
# Запуск RViz2
ros2 run rviz2 rviz2
# Або запуск RViz2 параметрами заданими у конфігураційному файлі
ros2 run rviz2 rviz2 -d ros_drone_camera.rviz
```

Якщо *RViz2* запускається без попередньо заданих параметрів, тоді для відображення відео з камери необхідно додати потрібний тип даних. В такому випадку слід натиснути “*Add*”, обрати тип даних “*Image*” і після додавання нового дисплею внести зміни у назву “*Topic*” згідно назви у файлі моделі дрону — блоки “*<namespace>*” та “*<camera_name>*”. Також слід упевнитись, що в якості “*Reliability Policy*” обрано “*Best Effort*”, інакше повідомлення можуть не доставлятись. В результаті дисплей для “*Image*” має бути аналогічним до показаного на рис. 5.2. Загалом *RViz2* є потужним інструментом у *ROS2*, який дозволяє візуалізувати інформацію різного типу із повідомлень *ROS*.

Щоб відобразити *UDP* відео потік через *GStreamer* можна використати команду показану нижче:

```
# Відображення відео потоку через GStreamer
gst-launch-1.0 -v udpsrc port=5600 caps='application/x-rtp, media=(string)video, clock-rate=(int)90000, encoding-name=(string)H264' ! rtph264depay ! avdec_h264 ! videoconvert ! autovideosink sync=false
```

5.4 Приклад програми обробки відео з дрону

Щоб продемонструвати використання камери віртуального дрону, програму “*simple_ros2_app*” було розширено і доповнено реалізацією процесу *ROS* для виконання трекінгу та *optical flow* по даним з відео. Програмний проєкт має структуру наведену на рис. 5.4.

Ключовими доповненнями в проєкті є:

- Програмний код для нового процесу *ROS* — “*optical_flow_node*”;
- Оновлений файл запуску *ROS* — “*spawn_drone_launch.xml*” для запуску нового процесу *ROS* “*optical_flow*”;
- Додаткові файли моделей дронів із камерою для симулятора *Gazebo* — файли “*sdf*”, що описують параметри відео камери та її функції.

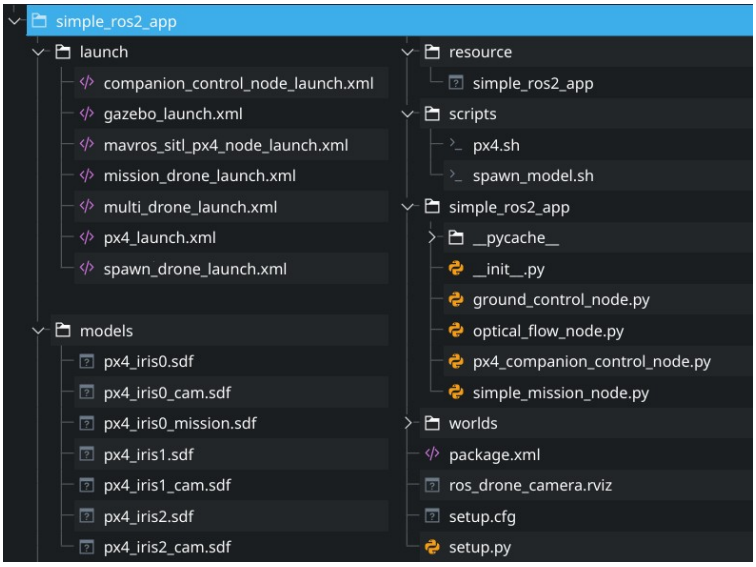


Рисунок 5.4 — Структура програмного проєкту “*simple_ros2_app*” для обробки відео потоку дрону

5.4.1 Алгоритм O*ptical Flow*

Обробка відео потоку, що отримується з віртуального дрону, виконується засобами програмної бібліотеки *OpenCV*. В прикладі використовується алгоритм *Lucas-Kanade* для обчислення оптичного потоку — *optical flow*. Фрагмент коду алгоритму із файлу “*optical_flow_node.py*” наведено на рис. 5.5. Оптичний потік — це відображення траєкторії руху об’єктів або поверхні, яке виникає в результаті переміщення спостерігача. В нашому випадку роль спостерігача відіграє дрон, що літає.

```
42    def frame_processing(self, msg):
43
44        if self.skip_frame > 0:
45            self.skip_frame -= 1
46            return
47
48        try:
49            cv_image = self.cv_bridge.imgmsg_to_cv2(msg, "bgr8")
50            cv_image_gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)
51
52            if not self.optical_flow_init:
53                self.prev_features = cv2.goodFeaturesToTrack(cv_image_gray, mask=None, **self.feature_params)
54                self.mask = np.zeros_like(cv_image)
55
56                if self.prev_features is not None:
57                    self.optical_flow_init = True
58
59            else:
60                features, st, err = cv2.calcOpticalFlowPyrLK(self.prev_frame_gray, cv_image_gray, self.prev_features,
61                                                             **self.lk_params)
62                # Select good points
63                if features is not None:
64                    good_new = features[st == 1]
65                    good_old = self.prev_features[st == 1]
66                    # draw the tracks
67                    for i, (new, old) in enumerate(zip(good_new, good_old)):
```

Рисунок 5.5 — Фрагмент коду обробки зображення для *optical flow*

Також при реалізації самого процесу *ROS — “optical_flow_node”* виконується його ініціалізація та створення підписки для отримання повідомлень із зображеннями, які відправляє симулятор *Gazebo* (рис 5.6).

```python
class OpticalFlowNode(Node):

    def __init__(self):

        super().__init__("optical_flow")
        self.drone_name = self.get_namespace()

        # Camera topic subsription
        self.cv_bridge = CvBridge()
        self.drone_camera = self.create_subscription(Image, "{}/camera/image_raw".format(self.drone_name),
                                                     self.frame_processing, qos.qos_profile_sensor_data)

        # Optical flow
        self.skip_frame = 5
        self.optical_flow_init = False
        self.prev_frame_gray = None
        self.color = np.random.randint(0, 255, (100, 3))
        self.feature_params = dict(maxCorners=100,
                                   qualityLevel=0.3,
                                   minDistance=21,
                                   blockSize=7)
        self.lk_params = dict(winSize=(15, 15),
                              maxLevel=2,
                              criteria=(cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT, 10, 0.03))
        self.prev_features = None
        self.mask = None
```

Рисунок 5.6 — Код ініціалізації процесу *ROS — “optical_flow_node”*

Щоб запускати новий процес, файл запуску *“spawn_drone_launch.xml”* доповнено блоком запуску відповідного процесу (рис. 5.7).

```
    <!-- Companion control node -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/companion_control_node_launch.xml">
        <arg name="instance" value="$(var instance)"/>
        <arg name="sitl" value="$(var sitl)"/>
    </include>

    <node pkg="simple_ros2_app" exec="optical_flow" namespace="/uav$(var instance)" output="screen" />

</launch>
```

Рисунок 5.7 — Доповнення файлу запуску новим процесом

5.4.2 Запуск програми *ROS* для обробки відео з дрону

Запуск розробленої програми виконується аналогічно до попередніх прикладів. В результаті буде запущена симуляція дрону із відео камерою та

відео потік відображатиметься у *QGroundControl* (рис. 5.8-5.9). Також додатково його можна окремо вивести у *RViz2* або *GStreamer.*

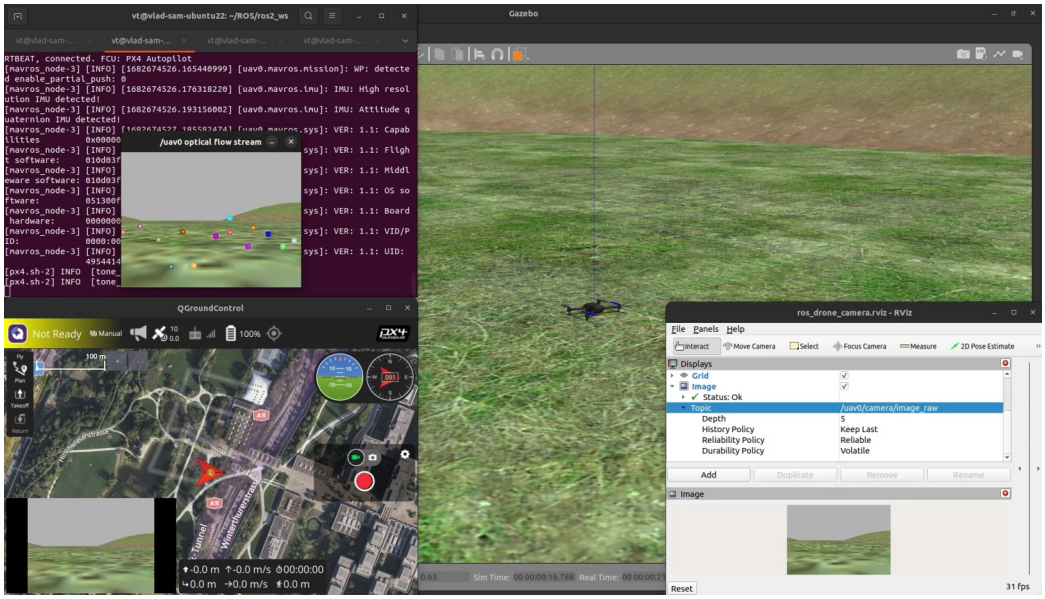


Рисунок 5.8 — Запущена симуляція дрону із відео камерою

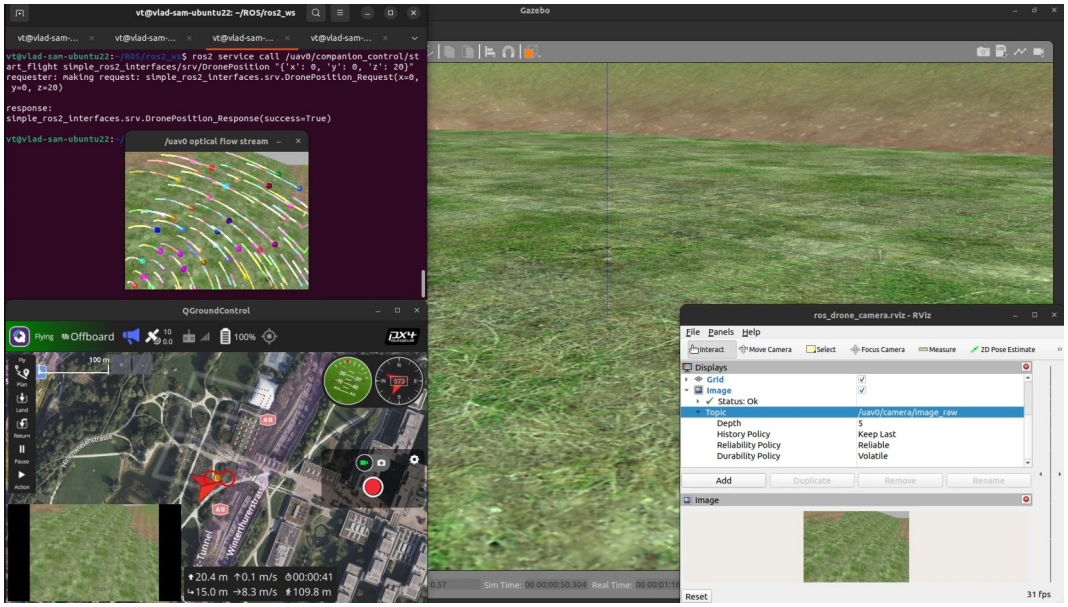


Рисунок 5.9 — Приклад застосування алгоритму *optical flow*

6 РОЙОВЕ КЕРУВАННЯ ДРОНАМИ ІЗ ВИКОРИСТАННЯМ *ROS2*

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04;
- *ROS2 Humble* 2024-09-09;
- *SITL ArduPilot* 4.3.3;
- *Gazebo Simulator* 11.10.2;
- *QGroundControl* 4.4.3.

6.1 Ройове керування із ROS2

Концепція ройового керування це схема організації при якій певна група дронів — “рій” працюють над спільною задачею і можуть розповсюджувати команди між собою. Це в певній мірі можна назвати прикладом децентралізованої системи, коли кожен дрон працює незалежно, але може координувати свої дії з іншими. В такому випадку кожен дрон зв’язується з іншим дроном, що знаходиться неподалік. Оскільки в *ROS2* використовується децентралізована служба розповсюдження даних, це робить його зручним для організації подібного роду керування дронами. Наприклад, можна розглянути ситуацію, коли певний дрон знаходиться на достатньо великому віддалені від станції керування, з якої він передбачає отримання нових політних координат (рис. 6.1). При виході за межі радіо зв’язку подібний дрон в подальшому не зможе оновлювати координати і в результаті досягнення останніх переданих координат, не зможе продовжувати політ. Це звісно не означає, що дрон не зможе летіти, але відправити його у нові координати буде неможливо. Але залежно від закладеної бортової програми, яка працює на комп’ютері компаньйоні дрону він, наприклад, все рівно зможе повернутись назад до точки зльоту.

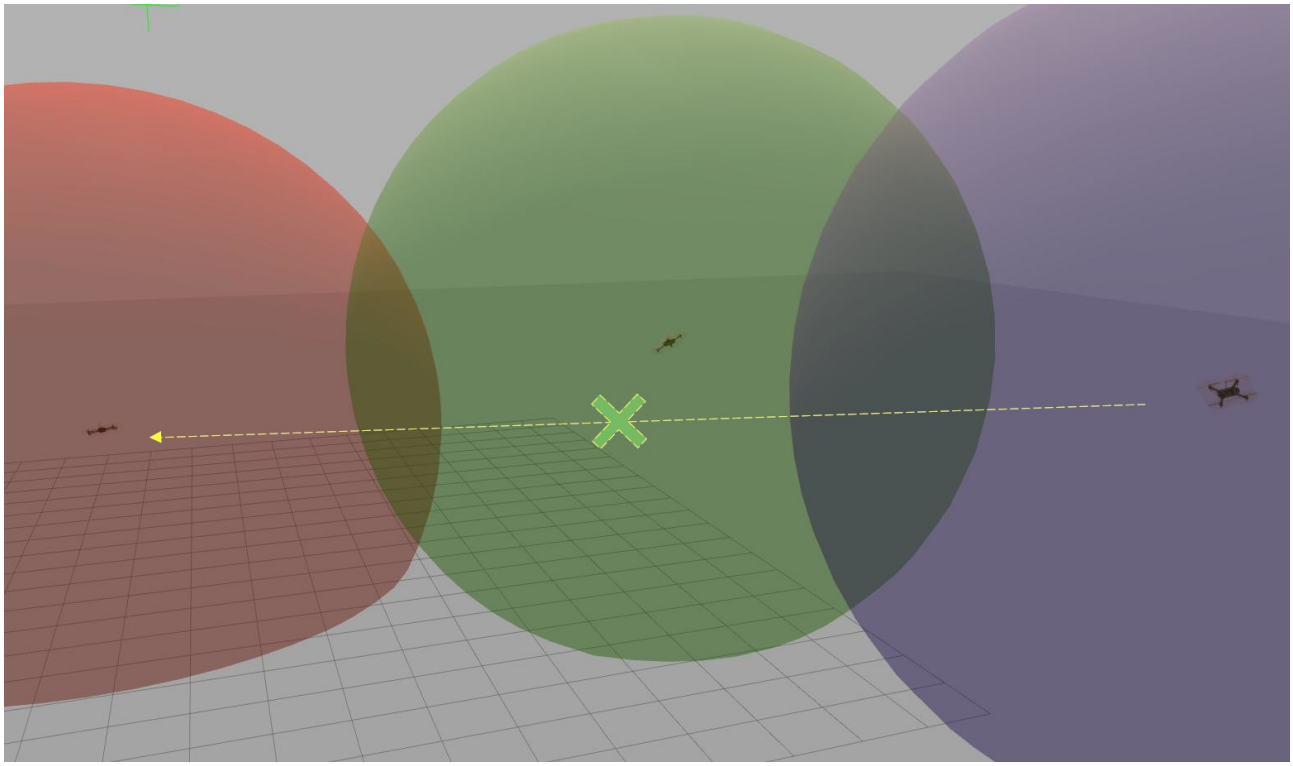


Рисунок 6.1 — Ситуація коли не має прямого зв’язку із найвіддаленіший дроном

З іншого боку, використання рою дронів і відповідно ройового керування в певній мірі дозволяє обійти описану проблему. В такому випадку можна уявити, що в певній області є деяка група дронів. Між ними є радіозв’язок, але не обов’язково у форматі “кожен-з-кожним”. В такій ситуації нові політні координати можуть бути передані до найвіддаленішого дрону через проміжних дронів, що знаходяться на всьому проміжку між керуючою станцією та найвіддаленішим дроном (рис. 6.2).

Також подібний сценарій можна розширити до ситуації, коли прямого зв’язку із станцією керування немає не тільки між найвіддаленішим дроном, але і взагалі з роєм дронів. Аналогічний механізм можна використати і для вирішення такої проблеми. Оскільки акумуляторна батарея дрону дозволяє йому працювати обмежений час, є логічним, що через деякий час дрон буде повертатись назад до точки зльоту. Протягом цієї операції інформацію для дрону можна буде оновити і запустити його повторно. Коли він повернеться до

групи дронів, то маючи оновлену інформацію або команду, зможе розповсюдити її між іншими дронами у рої.

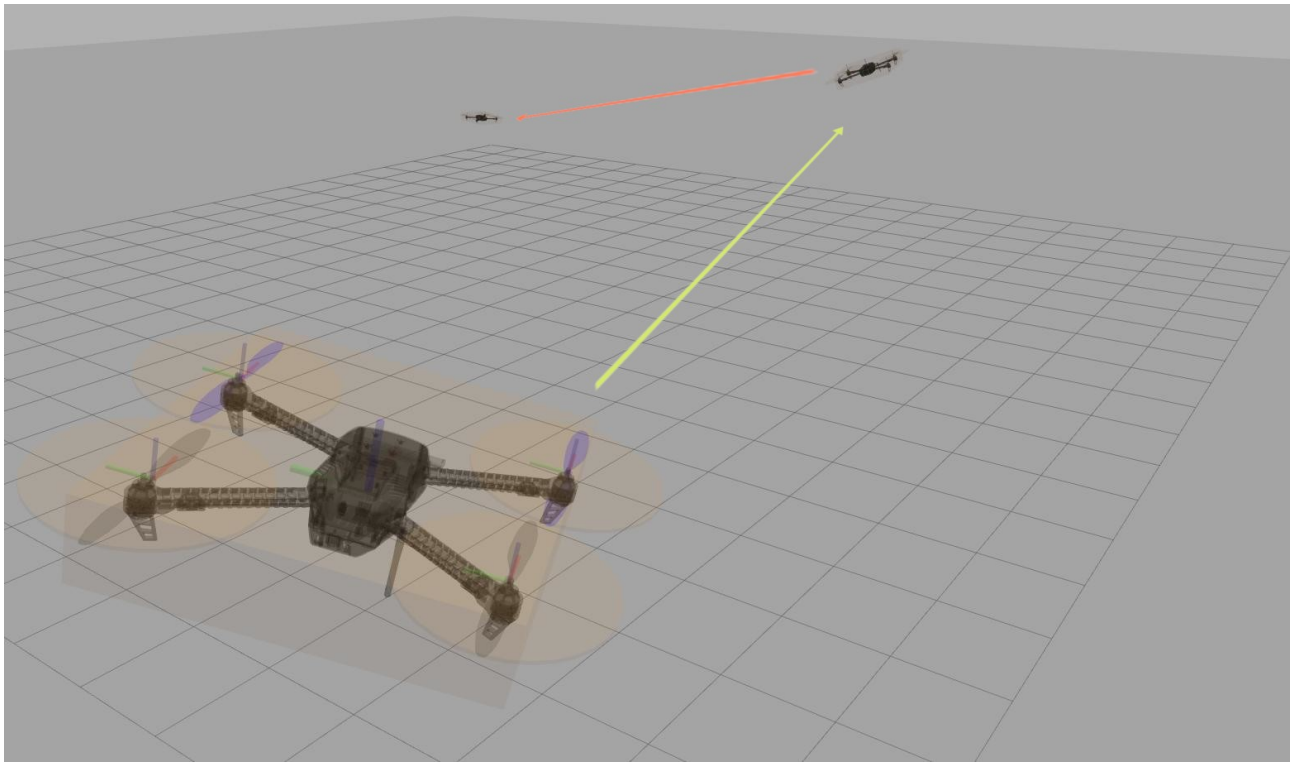


Рисунок 6.2 — Варіант розповсюдження інформації у рої дронів

Подібна схема організації керування може бути корисною при застосуванні дронів, наприклад, в задачі патрулювання певної місцевості.

6.2 Варіанти взаємодії станції керування із групою дронів

Розглянемо можливі схеми взаємодії станції керування польотом із декількома безпілотними літальними апаратами. У цьому випадку до уваги береться обмін повідомленнями між дронами на високому рівні абстракції, що не вносить обмеження на можливість застосування різного апаратного забезпечення. В такому разі взаємодія включає в себе наступні компоненти:

- Станція керування (*ground control*) — фізичний або віртуальний сервер на якому працює програмне забезпечення для управління польотом дронів;

- Бездротова мережа передачі даних — засоби за допомогою яких організовується радіо зв’язок між станцією керування та дронами;
- Безпілотний літальний апарат (дрон) — один або певна множина дронів з якими зв’язується станція керування. Окремим випадком може бути організація множини дронів у групу (рій) і керування ним як логічною одиницею.

Серед схем взаємодії станції керування та дронів можна виділити три основні:

1. Пряма взаємодія станції керування із одним або декількома дронами (рис. 6.3).

   При даній схемі передача повідомлень/команд до дронів відбувається напряму до кожного. Наведена схема є найпростішою та не потребує додаткового апаратного забезпечення, що встановлюється на дроні для прийняття команд. Взаємодія відбувається шляхом використання пропрієтарних рішень, що надаються виробниками конкретних комерційних дронів;
2. Взаємодія станції керування із декількома дронами, що об’єднані у групу (рій) (рис. 6.5). В такому випадку команда зі станції передається у рій, а потім розповсюджується серед усіх дронів, які наразі присутні у рої. Хоча з боку станції керування подібна схема не сильно відрізняється від попередньої, з боку дронів подібний підхід потребуватиме встановлення додаткового програмного та можливо апаратного забезпечення для організації децентралізованої взаємодії дронів між собою. Дана схема може бути не придатною для комерційних дронів через закритість їх програмної та апаратної складових, що в свою чергу не дозволить доповнити їх необхідними функціями.

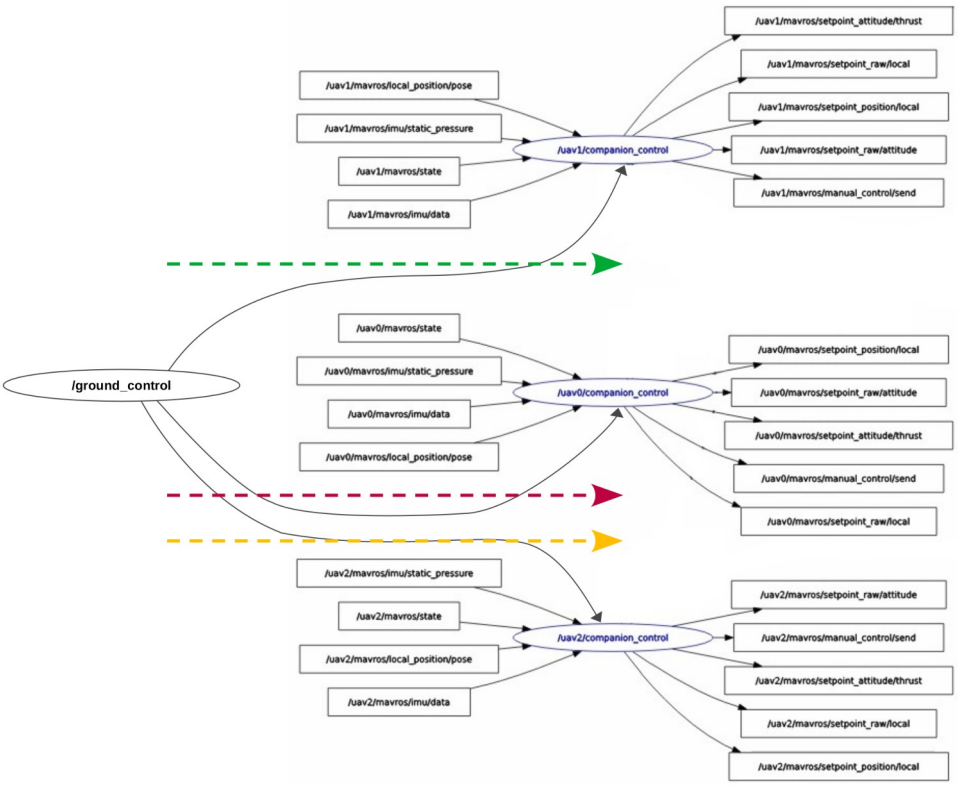


Рисунок 6.3 — Схема прямої взаємодії станції керування та дронів

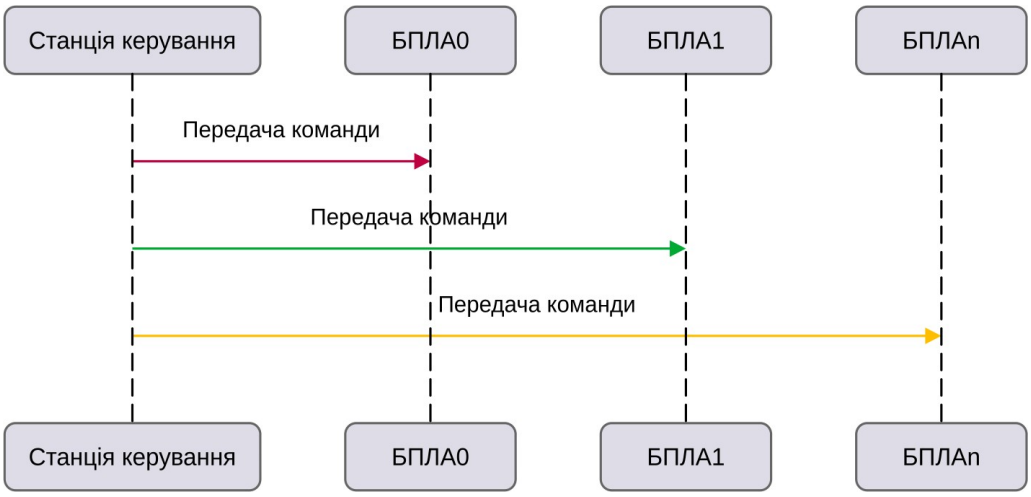


Рисунок 6.4 — Діаграма взаємодії станції керування та дронів

3. Гібридна взаємодія станції керування із віддаленим дроном з яким не має прямого радіо зв’язку (рис. 6.7).

   Подібна схема комбінує обидва попередні підходи для розширення радіусу радіозв’язку за рахунок того, що команда зі станції керування спочатку передається до найближчого дрону з яким є зв’язок, після чого цей дрон розповсюджує отриману команду у рої в який входить дрон до

якого необхідно надіслати команду зі станції керування. Наведеній схемі притаманні недоліки від попередньої схеми, оскільки організація подібної взаємодії може потребувати встановлення додаткового програмного та апаратного забезпечень.

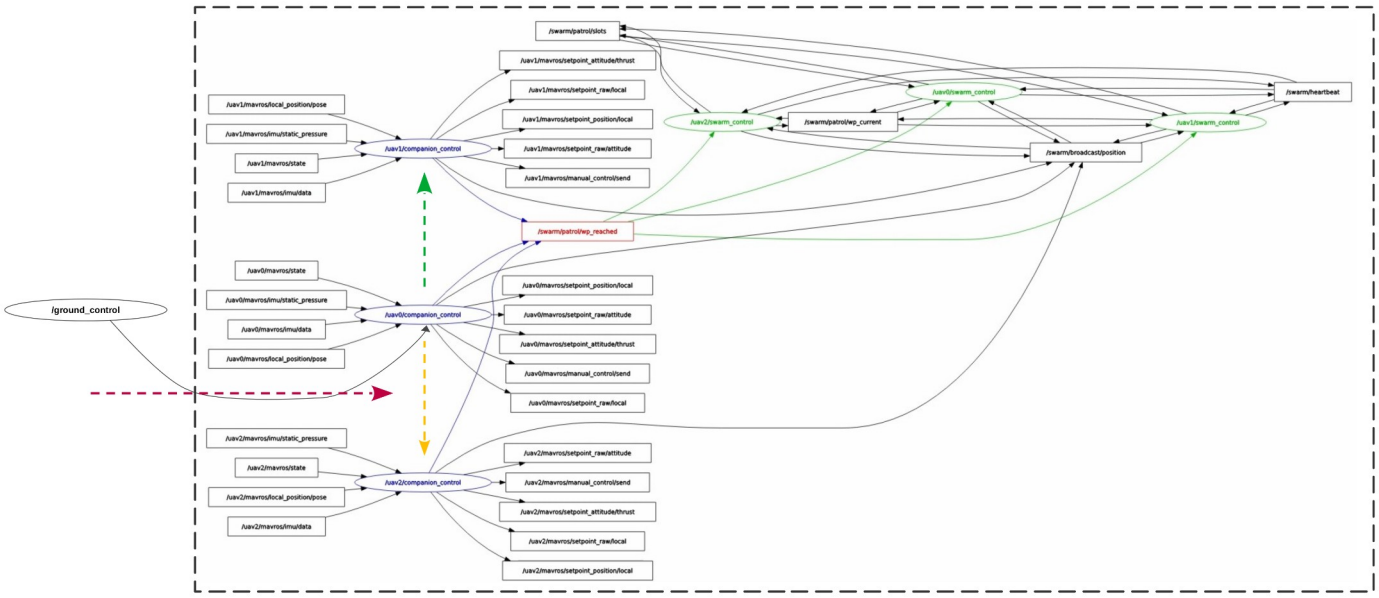


Рисунок 6.5 — Схема взаємодії станції керування із групою (рій) дронів

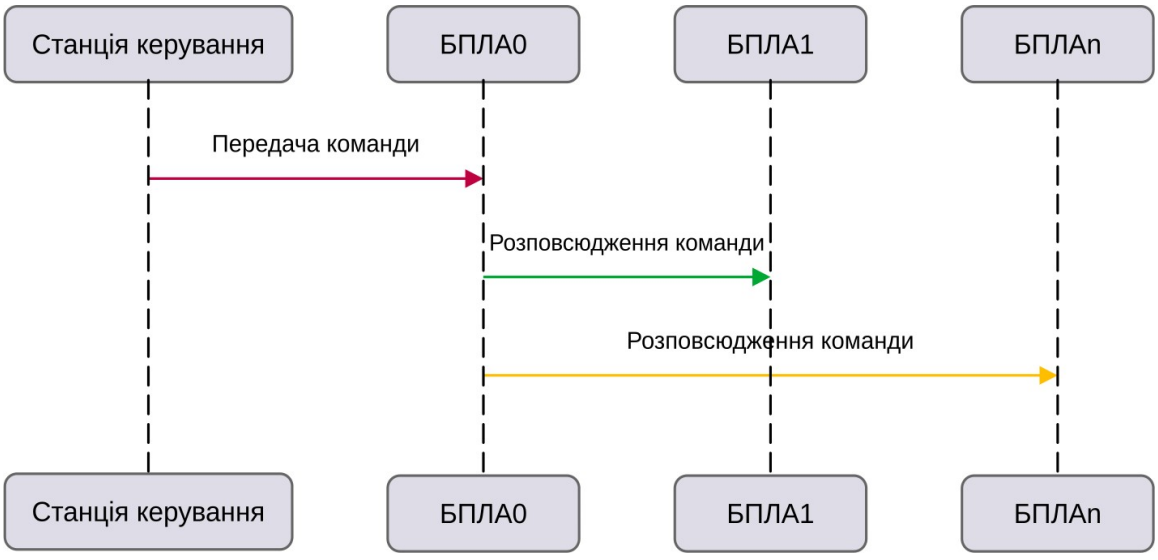


Рисунок 6.6 — Діаграма взаємодії станції керування із групою (рій) дронів

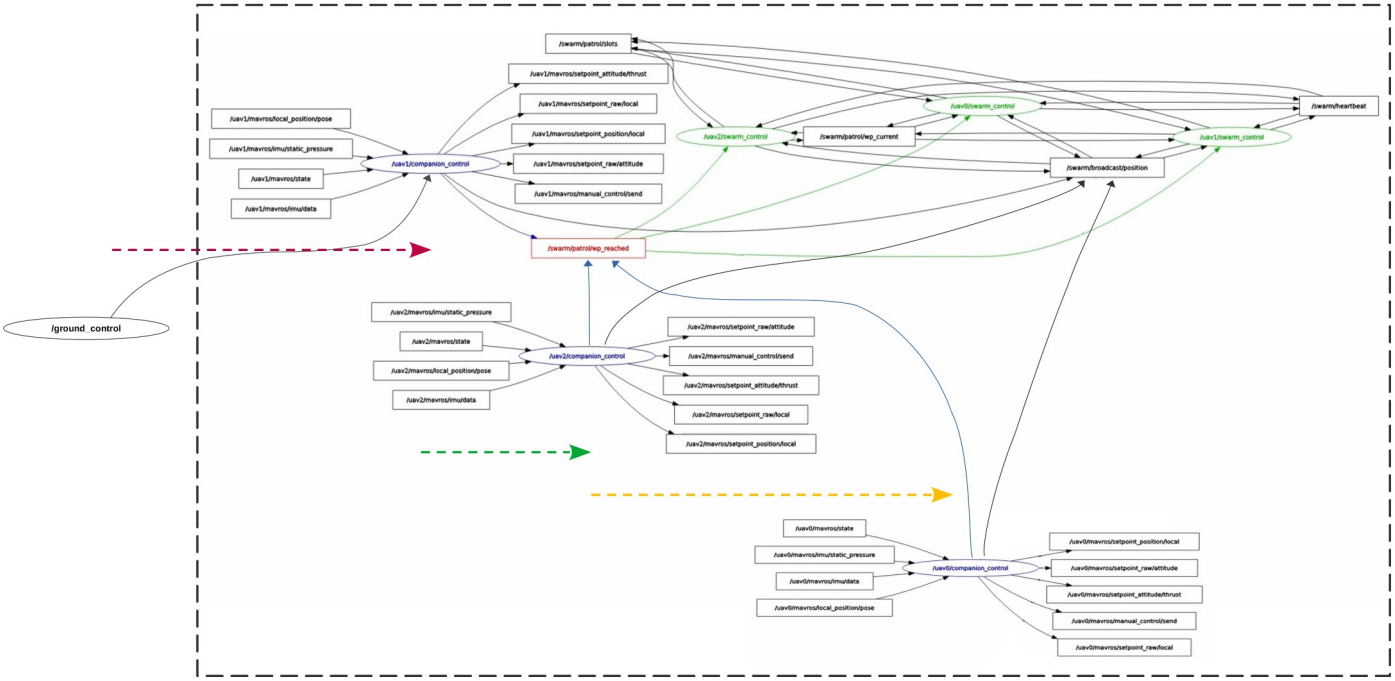


Рисунок 6.7 — Схема гібридної взаємодії станції керування із віддаленим дроном

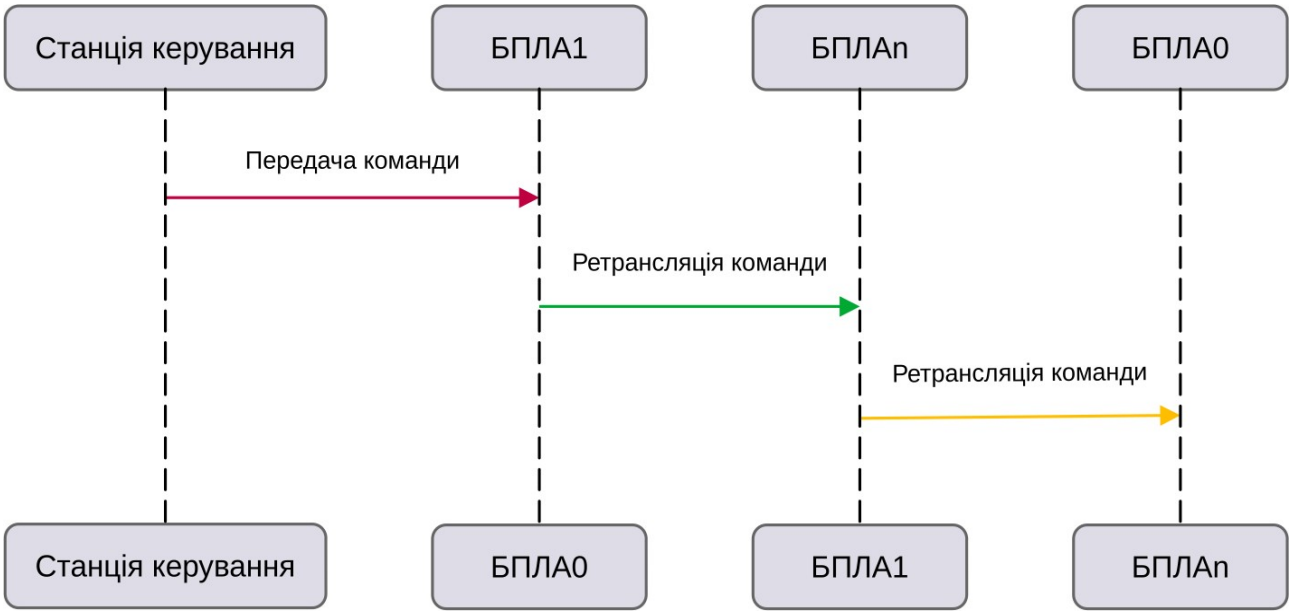


Рисунок 6.8 — Діаграма гібридної взаємодії станції керування із віддаленим дроном

6.3 Приклад програми ройового керування дронами із *ROS2*

Щоб продемонструвати ройове керування дронами по описаній вище схемі, програму “*simple_ros2_app*” було розширено і доповнено реалізаціями відповідних процесів *ROS* для ройового керування. В прикладі в якості програмного забезпечення для автопілоту використовується *Ardupilot*. Також, щоб змоделювати ситуацію відсутності зв’язку із роєм, наряду із віртуальними дронами використовується справжній дрон на базі політного контролеру *Pixhawk 2.4.8*. Програмний проєкт має структуру наведену на рис. 6.9.

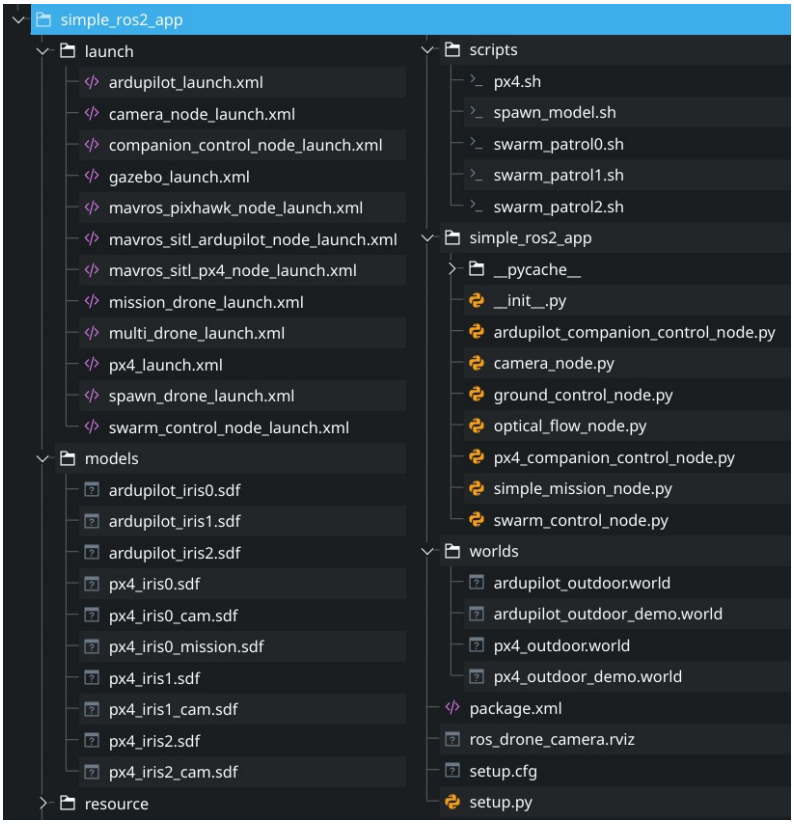


Рисунок 6.9 — Структура програмного проєкту “*simple_ros2_app*” для ройового керування дронами

Ключовими доповненнями в проєкті є:

• Програмний код для нових процесів *ROS* — “*swarm_control_node*”, “*ardupilot_companion_control_node*” та “*camera_node*”;

• Оновлені існуючі та додаткові файли запуску *ROS* — “*swarm_control_node_launch.xml*” для запуску нового процесу *ROS*

“*swarm_control*”, а також: “*mavros_pixhawk_node_launch.xml*”, “*mavros_sitl_ardupilot_node_launch.xml*”, “*ardupilot_launch.xml*”, “*camera_node_launch.xml*”;

• Додаткові файли моделей дронів на базі автопілоту *Ardupilot* для симулятора *Gazebo*.

6.3.1 Розповсюдження команд у рої дронів

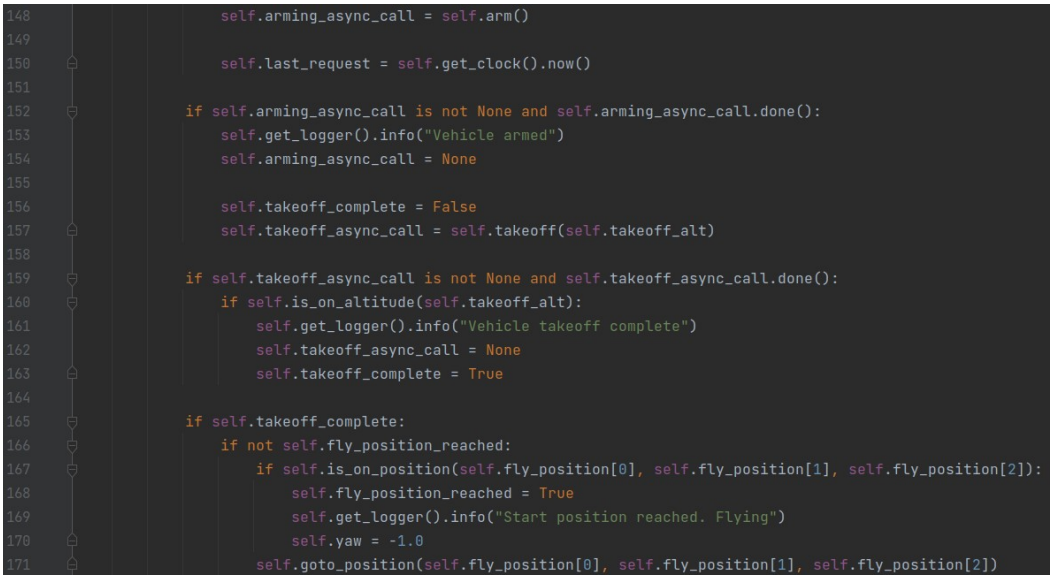
Для реалізації описаного вище механізму розповсюдження повідомлень або команд серед групи дронів було розроблено окремий процес *ROS*, який виконується на комп’ютері компаньйоні дрону на ряду із програмою керування. Задача цього додаткового процесу — виконувати передачу — “*broadcast*” найновішої команди. В цьому прикладі це набір політних координат. Щоб розрізняти команди за новизною, у форматі повідомлення присутнє окреме службове поле, яке містить час створення команди. Наявна у дрона команда з певним інтервалом публікується, так би мовити “в ефір”, за адресою “*/swarm/broadcast/position*”. Кожен дрон в межах радіо зв’язку приймає ці повідомлення один від одного. Якщо на якомусь дроні наявна команда більш застаріла, тоді дрон зберігає прийняту команду собі і в подальшому починає передавати вже оновлену команду. В такій ситуації досягається результат, коли найновіша команда розповсюджується по рою дронів. Фрагмент коду, що реалізує описаний функціонал наведено на рис. 6.10.

```
    def position_callback(self, msg):
        assert isinstance(msg, PoseStamped)
        if self.position_msg is None or msg.header.stamp.sec > self.position_msg.header.stamp.sec:
            self.position_msg = msg
            self.get_logger().info('New swarm position {} {} {}'.format(self.position_msg.pose.position.x,
                                                                        self.position_msg.pose.position.y,
                                                                        self.position_msg.pose.position.z))
            # drone start_flight
            self.update_drone_flight_position()

    def broadcast_callback(self):
        if self.position_msg is not None:
            self.swarm_position.publish(self.position_msg)

        if self.drone_async_call is not None and self.drone_async_call.done():
            self.get_logger().info("{} flight position updated from swarm".format(self.drone_name))
            self.drone_async_call = None

    def update_drone_flight_position(self):
        self.drone_position.stamp = self.position_msg.header.stamp
        self.drone_position.x = self.position_msg.pose.position.x
        self.drone_position.y = self.position_msg.pose.position.y
        self.drone_position.z = self.position_msg.pose.position.z

        self.drone_async_call = self.drone_service.call_async(self.drone_position)
```

Рисунок 6.10 — Фрагмент коду із розповсюдженням повідомлення

6.3.2 Реалізація керування для *Ardupilot* засобами *ROS*

Загальна схема управління автопілотом в режимі програмного керування на базі *Ardupilot* мало відрізняється від аналогічної схеми для *PX4*, але присутні деякі відмінності. По-перше, успішний зліт дрону в зазначеному режимі потребує передачі додаткової команди “*takeoff*”. Також, в автопілоті *Ardupilot* використовується система координат *NED — North East Down*. Через це, звична задача висоти польоту програмним шляхом вимагатиме передачі від’ємного значення бажаної висоти. Ці особливості керування було враховано у окремому процесі *ROS* — “*ardupilot_companion_control_node*” для керування автопілотом на базі *Ardupilot*. Фрагмент коду показано на рис. 6.11.



```
            self.arming_async_call = self.arm()

            self.last_request = self.get_clock().now()

        if self.arming_async_call is not None and self.arming_async_call.done():
            self.get_logger().info("Vehicle armed")
            self.arming_async_call = None

            self.takeoff_complete = False
            self.takeoff_async_call = self.takeoff(self.takeoff_alt)

        if self.takeoff_async_call is not None and self.takeoff_async_call.done():
            if self.is_on_altitude(self.takeoff_alt):
                self.get_logger().info("Vehicle takeoff complete")
                self.takeoff_async_call = None
                self.takeoff_complete = True

        if self.takeoff_complete:
            if not self.fly_position_reached:
                if self.is_on_position(self.fly_position[0], self.fly_position[1], self.fly_position[2]):
                    self.fly_position_reached = True
                    self.get_logger().info("Start position reached. Flying")
                    self.yaw = -1.0
                self.goto_position(self.fly_position[0], self.fly_position[1], self.fly_position[2])
```

Рисунок 6.11 — Фрагмент коду для *Ardupilot* з використанням додаткової команди “*takeoff*”

6.3.3 Налаштування *SITL Ardupilot* для роботи в симуляторі *Gazebo*

Саме по собі програмне забезпечення *SITL Ardupilot* не містить засобів для симуляції дрону у симуляторі *Gazebo*. Але є можливість запустити *SITL Ardupilot* із подальшим його підключенням до віртуального дрону в *Gazebo*. Для цього слід завантажити невеликий пакет додаткових програмних засобів, які міститимуть модель дрону для *Gazebo*, яка підходить для роботи з *Ardupilot*. Далі наведений приклад завантаження та встановлення зазначеного пакету із відкритого репозиторію.

```
    # Завантаження додаткових засобів Gazebo для Ardupilot
1.  git clone https://github.com/khancyr/ardupilot_gazebo
2.  cd ardupilot_gazebo/
    # Побудова проєкту
3.  mkdir build
4.  cd build
5.  cmake ..
6.  make -j4
    # Встановлення в робоче середовище
7.  sudo make install
```

Файли моделей “*ardupilot_iris*” в прикладі базуються на моделях отриманих з наведеного вище репозиторію. Також віртуальне середовище для *Ardupilot* дещо відрізняється від використаного раніше “*px4_outdoor.world*”, тому для симуляцій потрібно використовувати саме оновлене середовище “*ardupilot_outdoor.world*”.

Також для можливості запуску декількох екземплярів дронів на базі *Ardupilot* слід виконати налаштування по створенню параметрів для кожного додаткового екземпляру автопілоту дрона, що показувалось у прикладах до розділу “Симуляція дронів на базі *Ardupilot*”.

Після виконання всіх налаштувань екземпляри *SITL Ardupilot* можна запустити наступним чином. В подальшому цей процес можна інтегрувати у файли запуску для *ROS*.

```
    # Перехід до директорії SITL Ardupilot
0.  cd ~/ROS/ardupilot/ArduCopter/
    # Термінал 1
1.  ../Tools/autotest/sim_vehicle.py -f gazebo-iris0 --console -I0
    # Термінал 2
2.  ../Tools/autotest/sim_vehicle.py -f gazebo-iris1 --console -I1
    # Термінал 3
3.  ../Tools/autotest/sim_vehicle.py -f gazebo-iris2 --console -I2
```

6.3.4 Запуск програми *ROS* для ройового розповсюдження повідомлень

Як результат програму було доповнено функцією ройового керування, а саме ройового обміну повідомленнями, коли кожен дрон обмінюється актуальною командою з іншими сусідніми. Тестування відбувається на змодельованій ситуації із двома віртуальними і одним реальним дроном, щоб перевірити як відбуватиметься обмін повідомленнями. Віртуальні дрони запускаються на комп’ютері з вимкненим *WiFi*, що імітує групу із застарілою командою (рис. 6.12). А реальний дрон підключений до мережі *WiFi* містить свіжу команду і імітує дрон, який повертається у групу. У цьому тесті “команда”

— це летіти у певні координати. Далі *WiFi* на комп’ютері з віртуальними дронами вмикається — імітуючи ситуацію коли всі дрони вийшли на зв'язок (рис. 6.13). Як результат команда від реального дрону з новими координатами розповсюджується між всіма віртуальними дронами і всі дрони прямують до нової координати. Таким чином дрони у рої можуть обмінюватись між собою інформацією.

Запуск всіх компонентів програми відбувається аналогічно до попередніх прикладів. Спочатку відбувається запуск *SITL Ardupilot*, що було показано вище. Потім виконується запуск середовища *Gazebo*, а також створюється потрібна кількість екземплярів дронів.

```
# Термінал 1
ros2 launch simple_ros2_app gazebo_launch.xml world:=ardupilot_outdoor
# Термінал 2
ros2 launch simple_ros2_app spawn_drone_launch.xml instance:=0 sitl:=ardupilot
# Термінал 3
ros2 launch simple_ros2_app spawn_drone_launch.xml instance:=1 sitl:=ardupilot
```

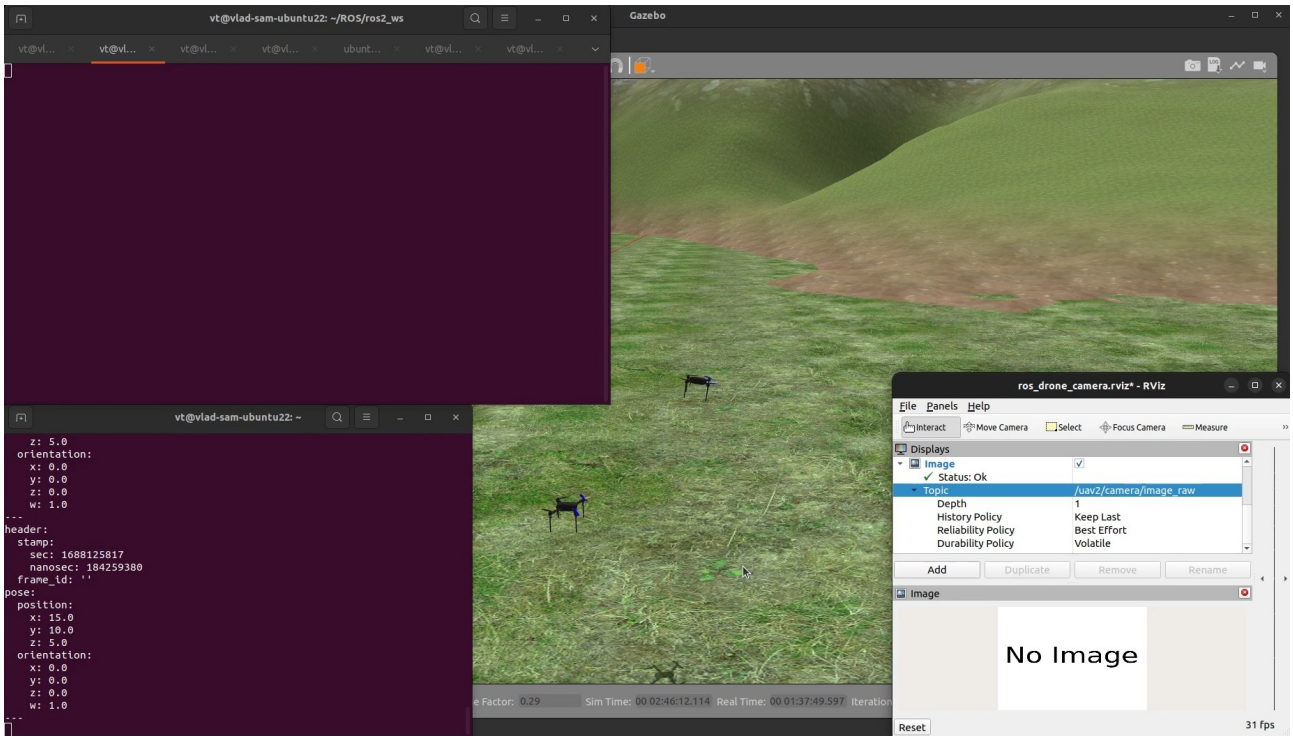


Рисунок 6.12 — Імітація групи дронів із застарілою командою

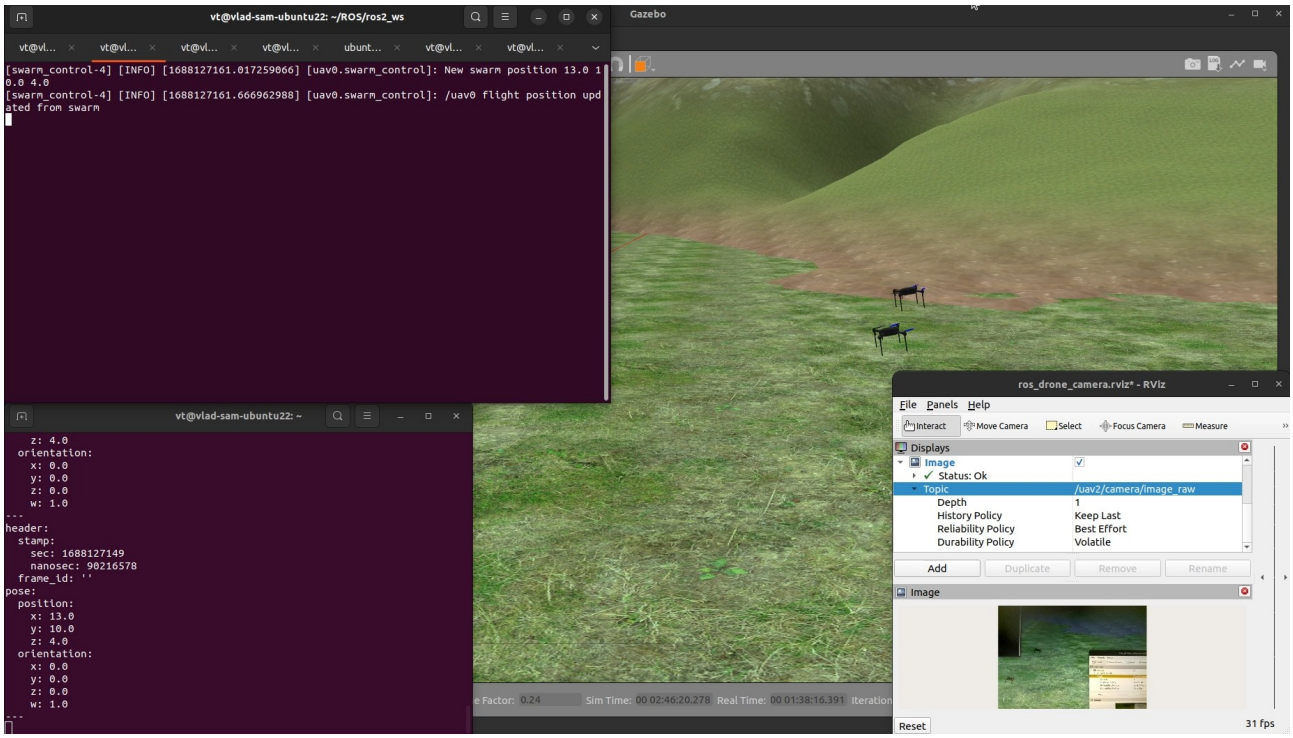


Рисунок 6.13 — Імітація ситуації коли всі дрони вийшли на зв'язок

6.4 Приклад застосування розробленої програми для задачі патрулювання

Маючи наявний розроблений проєкт програми для ройового керування, а також розроблені до цього інтерфейси та сервіси для керування дронами засобами *ROS*, є можливим додатково реалізувати функцію патрулювання групою дронів певної області у віртуальному середовищі (рис. 6.14-6.15). У найпростішому наближені, вирішити поставлену задачу можна використовуючи наявний у кожного дрону сервіс “*companion_control/start_flight*”. Він дозволяє відправити дрон у потрібні координати. Беручи до уваги, що координати можуть розповсюджуватись між групою дронів, можна зімітувати цей процес через звичайні скрипти: “*swarm_patrol1.sh*”, “*swarm_patrol2.sh*” та “*swarm_patrol3.sh*”.

В цьому прикладі використовуються моделі віртуальних дронів з камерою, а також середовище “*_outdoor_demo.world*” де у певній області, яку

патрулюватимуть дрони розміщено декілька моделей транспортних засобів.

Запуск всіх компонентів симуляції відбувається аналогічно процесу описаному у попередніх прикладах, але віртуальне середовище задається “*ardupilot_outdoor_demo.world*” або “*px4_outdoor_demo.world*” залежно від використаного *SITL*.

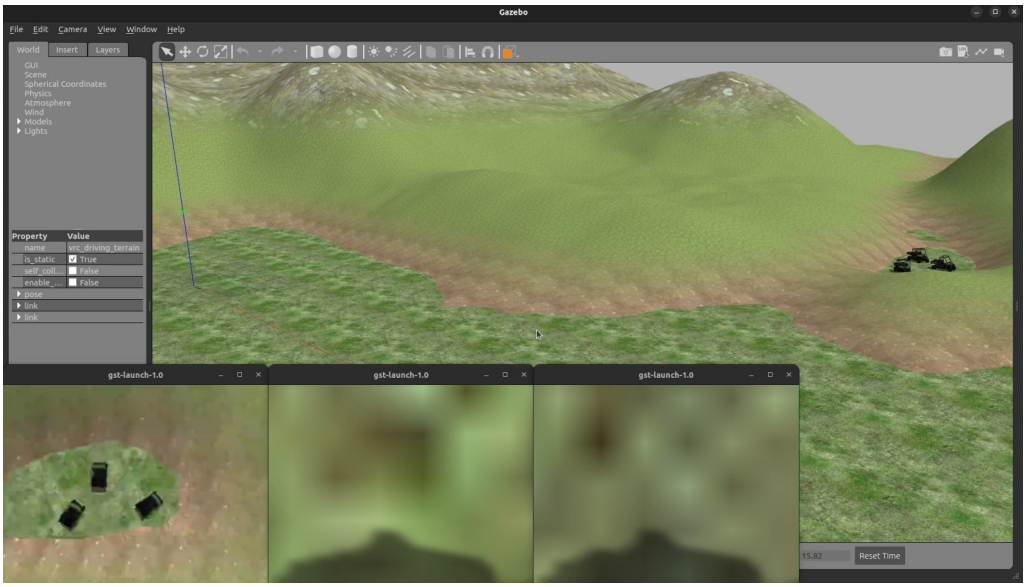


Рисунок 6.14 — Запуск патруля з групою дронів

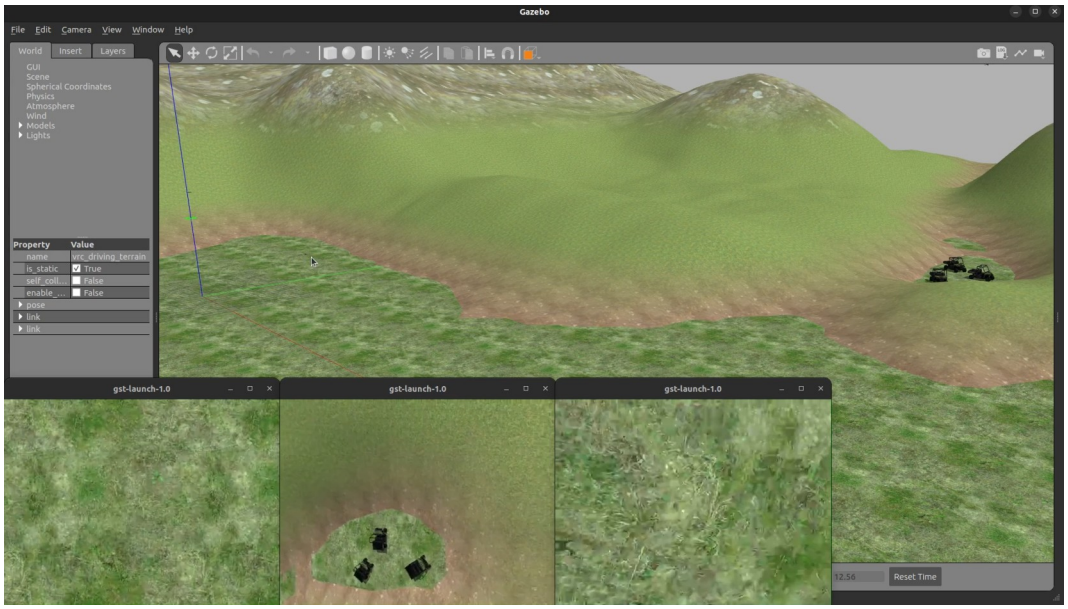


Рисунок 6.15 — Зміна дронів на патрулі

7 СЦЕНАРІЙ ПАТРУЛЮВАННЯ ПРИ РОЙОВІЙ ОРГАНІЗАЦІЇ ДРОНІВ

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04.4 *LTS*;
- *ROS2 Humble* 2025-02-11;
- *Ardupilot — ArduCopter* 4.3.3;
- *Gazebo Simulator* 11.10.2.

7.1 Інфраструктура для симуляції

Симуляція сценарію патрулювання в ройовій організації використовує налаштовану інфраструктуру *ROS2* та *Gazebo*, що дозволяє протестувати роботу розробленого алгоритму групового керування у віртуальному середовищі (рис. 7.1). Тестове середовище обране з відкритого набору, який можна завантажити за посиланням (https://github.com/leonhartyao/gazebo_models_worlds_collection)

Апаратне забезпечення інфраструктури включає наступні компоненти:

- ЦП — *Intel Core i5-4460 CPU @ 3.20GHz*
- ОЗП — 24 ГБ *PC3-12800*
- ГП — *NVIDIA GeForce RTX 3080 Ti*

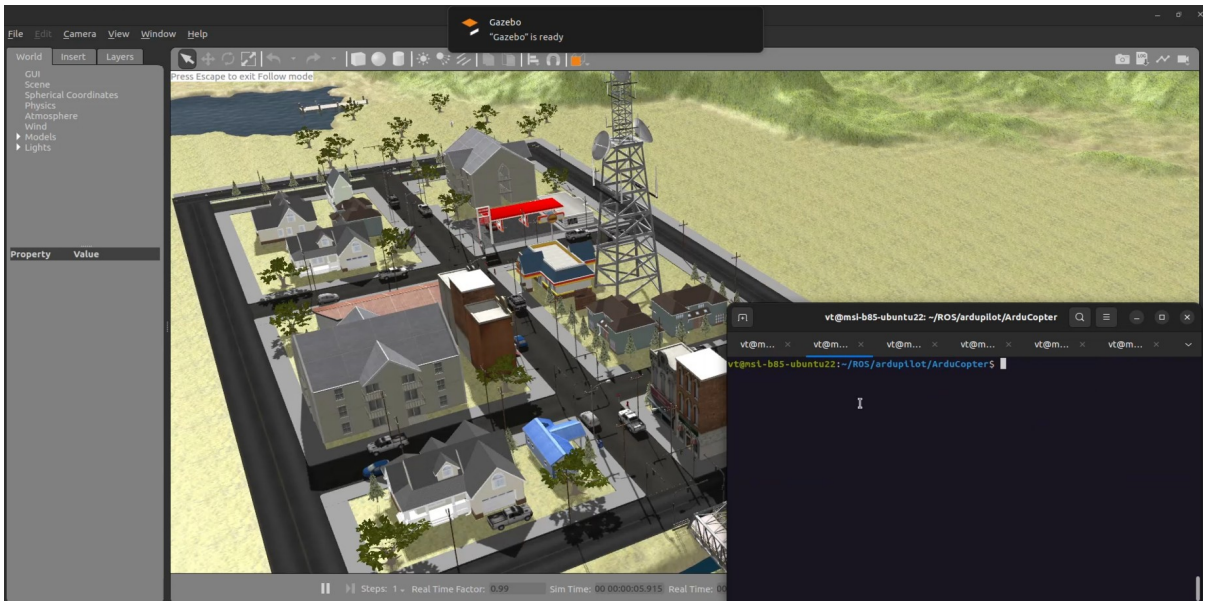


Рисунок 7.1 — Тестове віртуальне середовище для симулятору *Gazebo*

7.2 Групове керування дронами засобами ROS2

Групова або “ройова” організації взаємодії дронів має ключову перевагу у більшій гнучкості та надійності системи в цілому за рахунок того, що кожен дрон у групі є автономним і може переймати функції та задачі інших учасників групи в разі їх відмови. Децентралізована архітектура подібної організації додатково покращує дану систему, оскільки в ній не має єдиної точки відмови або центрального компоненту, який може вплинути на працездатність всієї групи. При такій організації кожен учасник — дрон взаємодіє з іншими шляхом передачі службової інформації по підтримці злагодженості рою, а також обмінюється з іншими інформацією про виконані задачі, що дозволяє синхронізувати стани всіх учасників. В разі відмови одної з частин подібної групової системи, її швидко можна замінити іншим дроном, який зможе отримати необхідну інформацію для продовження виконання певної задачі від інших дронів у групі, оскільки дані в рої є синхронізованими.

Використання *ROS2* з його децентралізованою службою обміну повідомленнями (*DDS*) дає можливість організувати описану вище схему взаємодії дронів. В прикладі використовується модульна архітектура при якій керування дроном оформлюється у вигляді модулів *ROS* із специфічними призначеннями, які забезпечують роботу окремих функцій дрону. Наразі функції дрону розділені на дві складові, а саме:

- політні функції, що забезпечують взаємодію бортового комп’ютеру “компаньйону” із політним контролером дрону, а також передачу планів польоту;
- групові функції рою, які відповідають за обмін інформацією між дронами, відстеження стану дронів у групі, а також синхронізацію задач у рої, що в свою чергу дозволяє відновлювати працездатність рою при виході з ладу окремих учасників.

Зазначені функції реалізуються відповідними модулями *ROS*, а саме:

- *companion_control_node* — забезпечує політні функції через засоби *MAVROS*;
- *swarm_control_node* — забезпечує групові “ройові” операції.

Для тестування групової організації роботи дронів використовується симуляція із трьох дронів *UAV0-UAV2*. Відповідна схема взаємодії “політного” (рис. 7.2) та “ройового” (рис. 7.3) модулів наведена нижче.

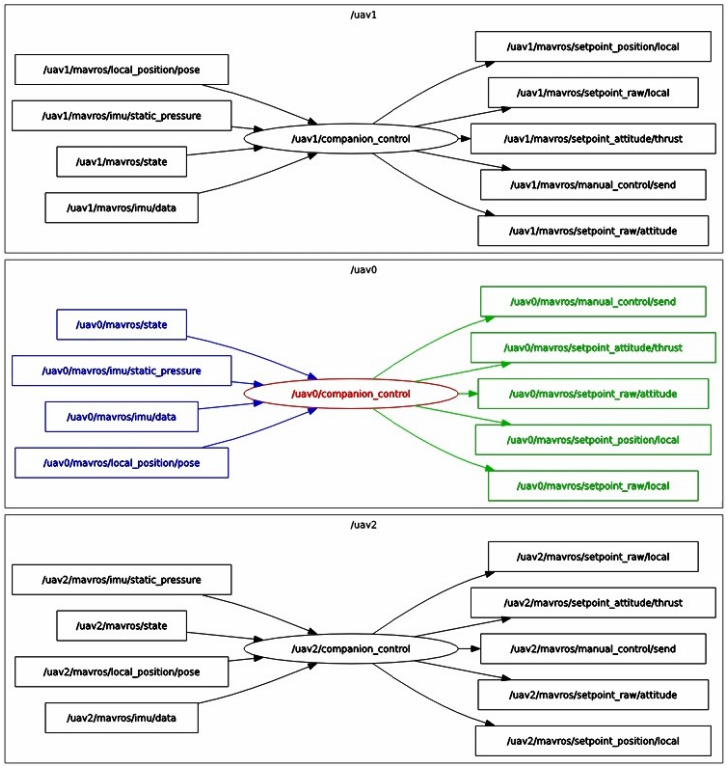


Рисунок 7.2 — Схема взаємодії “політного” модуля для трьох дронів

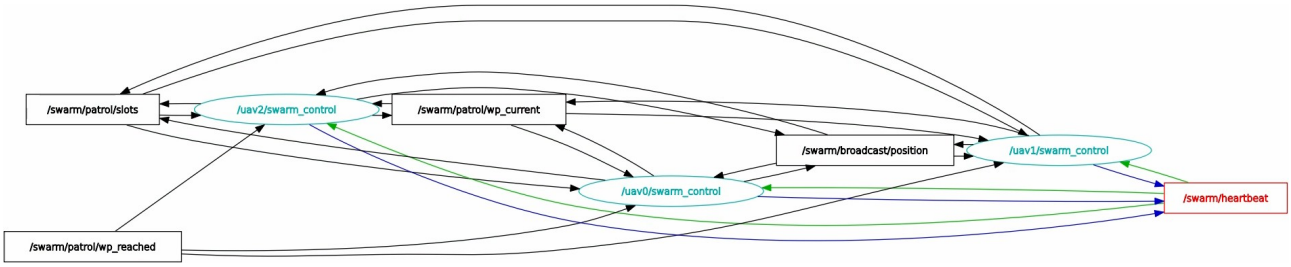


Рисунок 7.3 — Схема взаємодії “ройового” модуля для трьох дронів

При цьому, кожен модуль відповідає за свої функції дрону і не перешкоджає роботі іншого, тому в разі збоїв одного з модулів інший

продовжуватиме працювати. Загальна схема взаємодії та обміну інформацією між всіма модулями всіх дронів наводиться на рис. 7.4.

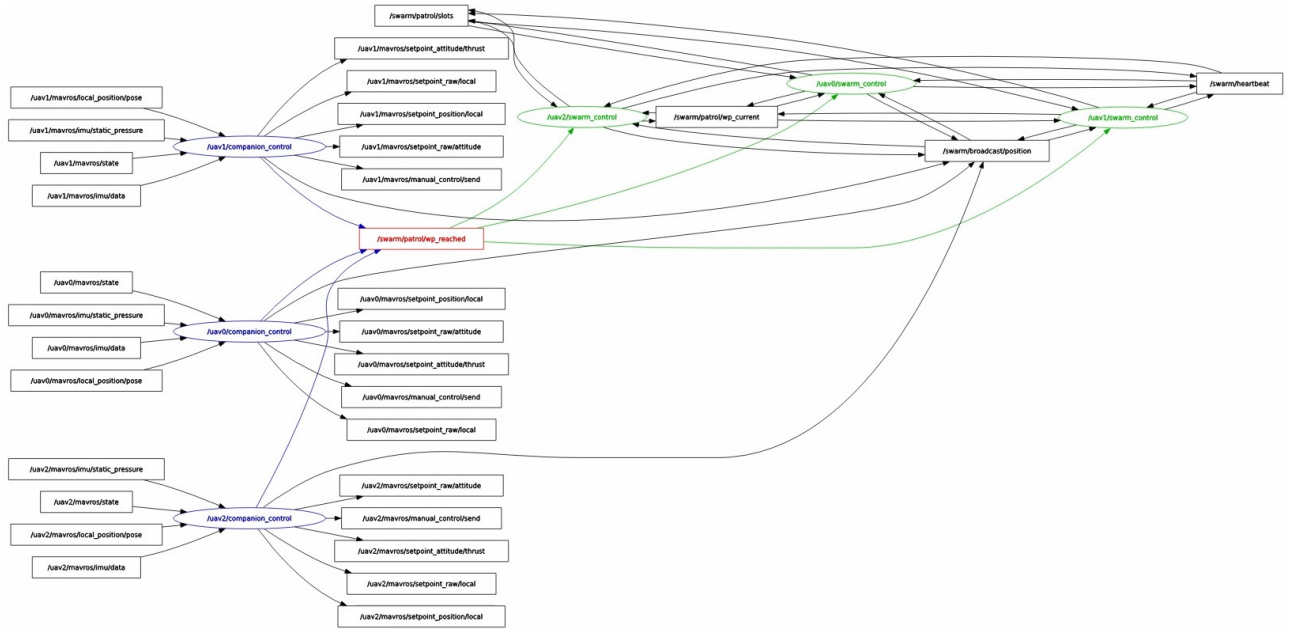


Рисунок 7.4 — Загальна схема взаємодії зазначених модулів для трьох дронів

Як результат такого розділення функцій, обмін інформацією між двома типами модулів відбувається через відповідні засоби:

- *companion_control/start_flight* — сервіс для передачі політної інформації та/або запуску дрону;
- *companion_control/stop_flight* — сервіс повернення дрону у задані координати;
- */swarm/broadcast/position* — розповсюдження інформації про координати дронів у рої;
- */swarm/heartbeat* — відстеження станів дронів у рої.

А також специфічні засоби взаємодії для сценарію патрулювання:

- */swarm/patrol/slots* — обмін інформацією про зайняті місця у патрулі;
- */swarm/patrol/wp_current* — синхронізація виконаних дронами задач у патрулі;
- */swarm/patrol/wp_reached* — інформування інших дронів у групі про виконану задачу. Дані повідомлення використовуються для синхронізації

роботи двох модулів “політного” та “ройового” (рис. 7.5).

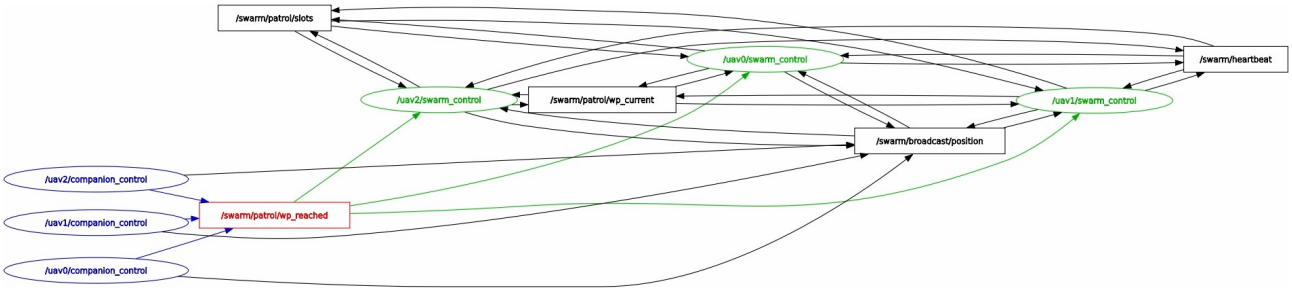


Рисунок 7.5 — Використання повідомлень */swarm/patrol/wp_reached* для синхронізації двох типів модулів

7.3 Алгоритм групової організації дронів для сценарію патрулювання

В тестовому сценарії передбачається використання трьох дронів, які взаємодіють між собою для забезпечення більшої відмовостійкості системи при виконанні поставленої задачі. Для патруля використовуються два дрони, при цьому третій дрон залишається у резерві і може заміщати дрон, який вийде з ладу. Хоч цей тестовий приклад має невелику кількість дронів, запропонований алгоритм групової організації може бути масштабований на більшу кількість дронів у рої.

Основні дії, що виконують дрони у зазначеній груповій організації для патруля, включають:

1. Спочатку дрон запускає процедуру “виявлення” (*swarm discovery*) інших дронів у рої через обмін повідомленнями “*/swarm/heartbeat*”;
2. По завершенню процедури всі дрони дізнаються один про одного і починають відстежувати “статуси” одне одного;
3. В прикладі, режим патрулювання має 2 вільні місця — “слоти”. Два перші дрони “займають” вільні слоти, повідомляють про це інших через повідомлення “*/swarm/patrol/slots*” і відповідно отримують статус “*active*”, третій дрон — чекає звільнення місця у патрулі, наприклад, в разі відмови інших дронів, і має статус “*standby*”;

4. Всі дрони “перевіряють” статус одне одного, і якщо будь-який інший дрон довго не виходив на зв’язок — не оновлював свій статус “*/swarm/heartbeat*”, тоді цей дрон відмічається зниклим і отримує статус “*offline*”. При чому, якщо цей дрон займав слот у патрулі, тоді він звільняється, а його місце займає вільний дрон, що чекає зі статусом “*standby*”;
5. Інформація про “виконані команди” у патрулі розповсюджуються кожним дроном у рої через повідомлення “*/swarm/patrol/wp_reached*”, тому новий дрон “продовжує” патруль з місця зламаного, синхронізуючи виконані задачі через “*/swarm/patrol/wp_current*”;
6. Якщо у рої всі учасники вийшли з ладу і мають статус “*offline*”, вважається, що “рій розпався”, і всі поточні статуси групи втрачаються. В такій ситуації нові дрони організовуються у нову ройову групу і починають свою внутрішню запрограмовану поведінку спочатку.

Фрагмент програмного модулю ROS — “*swarm_control_node*”, що відповідає за групову організацію роботи дронів наведено на рис. 7.6.

```
def patrol_callback(self):
    # Initialization and swarm discovery
    if self.swarm_discovery or self.get_clock().now().seconds_nanoseconds()[0] - self.init_time.seconds_nanoseconds()[0] > 10:
        if not self.swarm_discovery:
            self.get_logger().info('Swarm discovery completed, waiting free patrol slot')
        # Check all heartbeats
        for k in list(self.swarm_heartbeat_data):
            if (k in self.patrol_slots_data and
                    self.get_clock().now().seconds_nanoseconds()[0] - self.swarm_heartbeat_data[k] > 5):
                self.get_logger().info('{} heartbeat timeout, assuming offline'.format( *args: k, self.get_clock().now().seconds_nanoseconds()[0] - self.swarm_heartbeat
                # Remove offline drone from patrol slot
                self.patrol_slots_data = self.patrol_slots_data.replace(k, '{}')

        # Check patrol available slots
        if not self.patrol_mode_active:
            # Drone idle and has free slot in patrol
            if '{}' in self.patrol_slots_data:
                self.get_logger().info('{} taking free patrol slot'.format(self.drone_name))
                self.patrol_slots_data = self.patrol_slots_data.replace('{}', self.drone_name, 1)
                self.patrol_mode_active = True

                # Start patrol program
                ps = self.patrol_slots_data.find(self.drone_name) % 5
                self.get_logger().info('Current slot {}, patrol program {}'.format( *args: ps, self.patrol_slots_wp_data[ps]))
                self.send_new_patrol_waypoint(self.patrol_slots_wp_data[ps])

        # Send patrol slots after swarm discovery
        self.patrol_slots_msg.data = self.patrol_slots_data
        self.patrol_slots.publish(self.patrol_slots_msg)

        # Send slot waypoints to swarm
        self.patrol_slots_wp_msg.data = '{} {}'.format( *args: self.patrol_slots_wp_data[0], self.patrol_slots_wp_data[1])
        self.patrol_waypoint.publish(self.patrol_slots_wp_msg)

        self.swarm_discovery = True

    # Send heartbeat
    self.swarm_heartbeat.publish(self.drone_heartbeat_msg)
```

Рисунок 7.6 — Процедура організації групової взаємодії дронів

7.5 Тестування групової організації дронів

Запуск симуляції дронів виконується у середовищі *Gazebo*. Імітація роботи політного контролеру дрону відбувається засобами *SITL Ardupilot*. Програмні компоненти *ROS* для забезпечення функцій керування “польотом” та “роєм” запускаються, використовуючи відповідні файли запуску “*companion_control_node_launch.xml*” та “*swarm_control_node_launch.xml*”. Основні етапи групової — “ройової” взаємодії дронів показані на рис. 7.7-7.10.

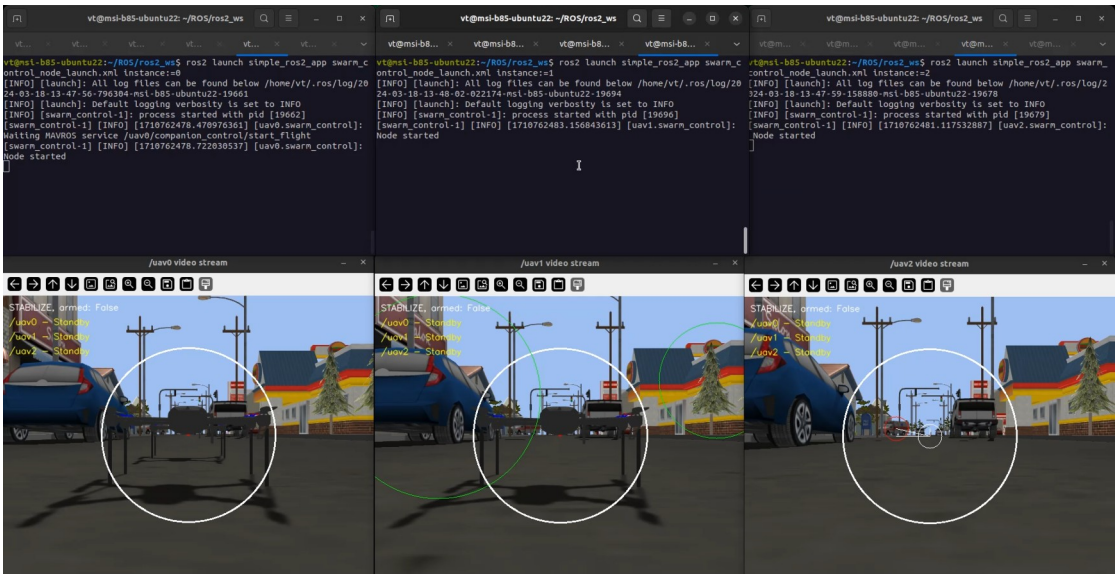


Рисунок 7.7 — Запуск та організація дронів “*/uav0-/uav2*” у ройову організацію

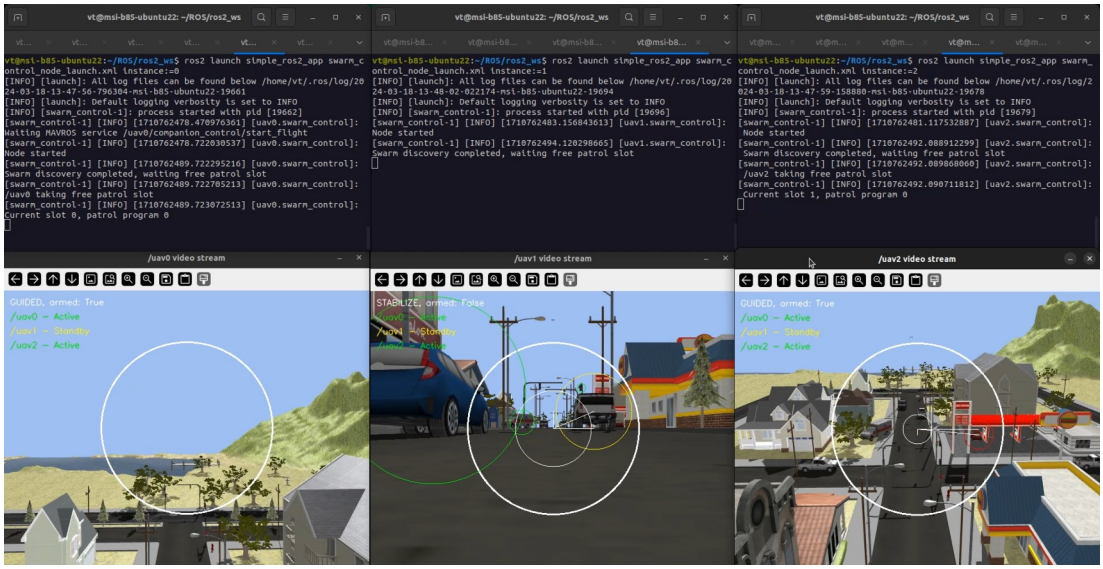


Рисунок 7.8 — Виконання двома дронами “*/uav0*” та “*/uav2*” задачі патрулювання, третій дрон “*/uav1*” очікує

Імітація виходу із ладу дрону виконується шляхом вимкнення зазначених керуючих модулів *ROS* та примусового вимкнення двигунів дрона.

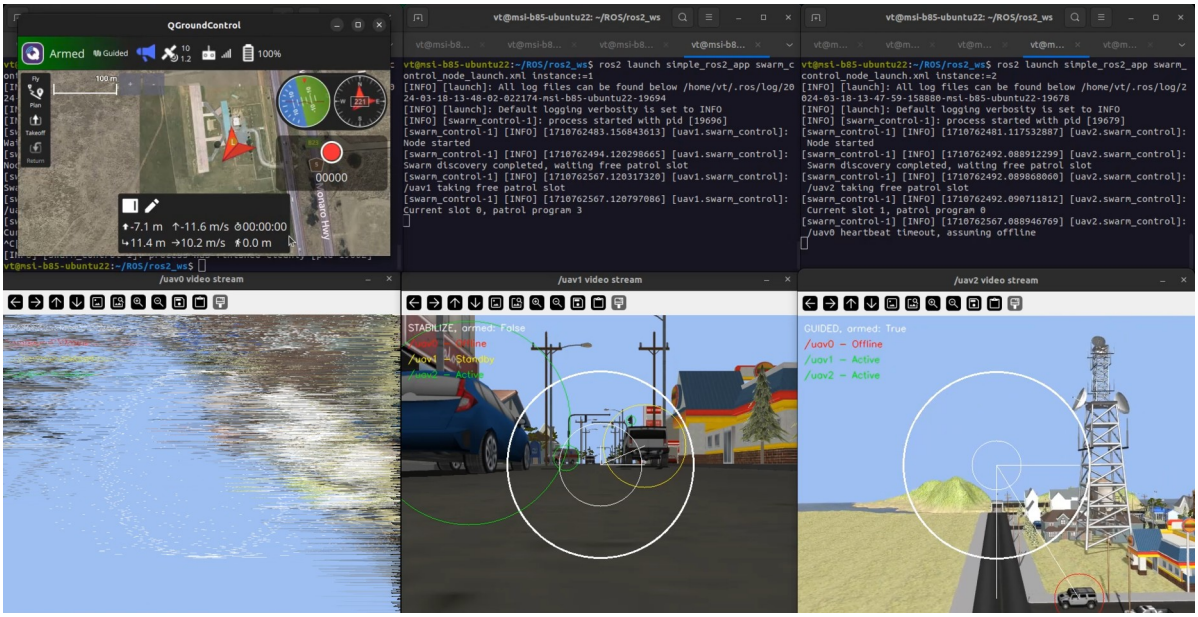


Рисунок 7.9 — Заміщення резервним дроном “*/uav1*” місця у патрулі виведеного з ладу першого дрону “*/uav0*”

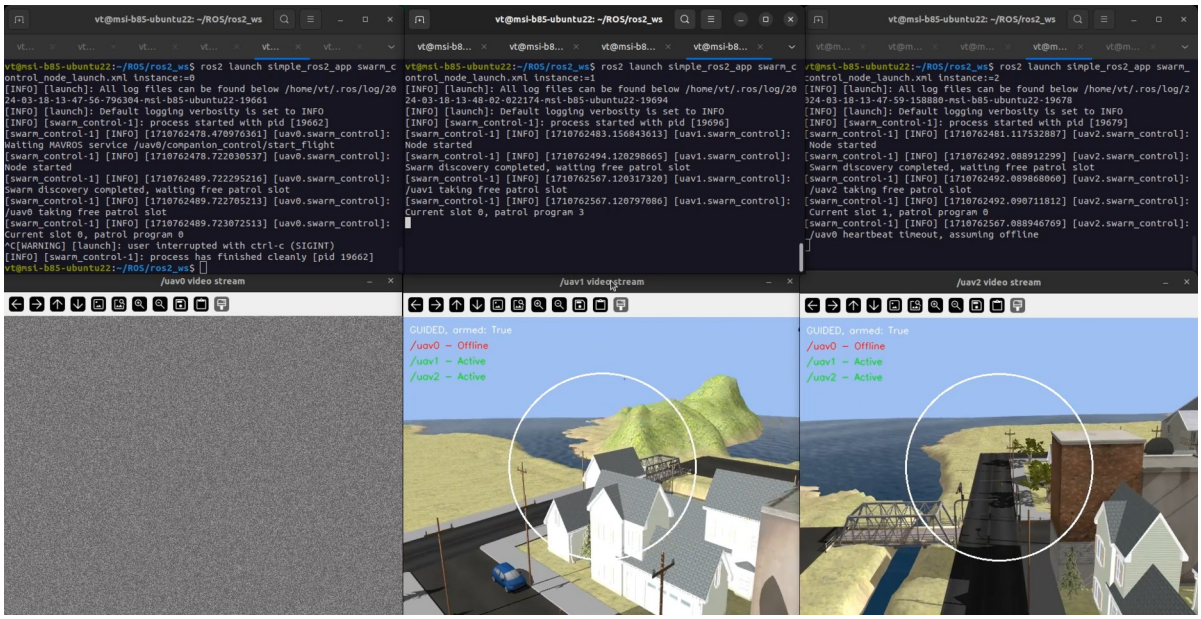


Рисунок 7.10 — Продовження резервним дроном “*/uav1*” задачі патрулювання з місця першого дрону “*/uav0*”

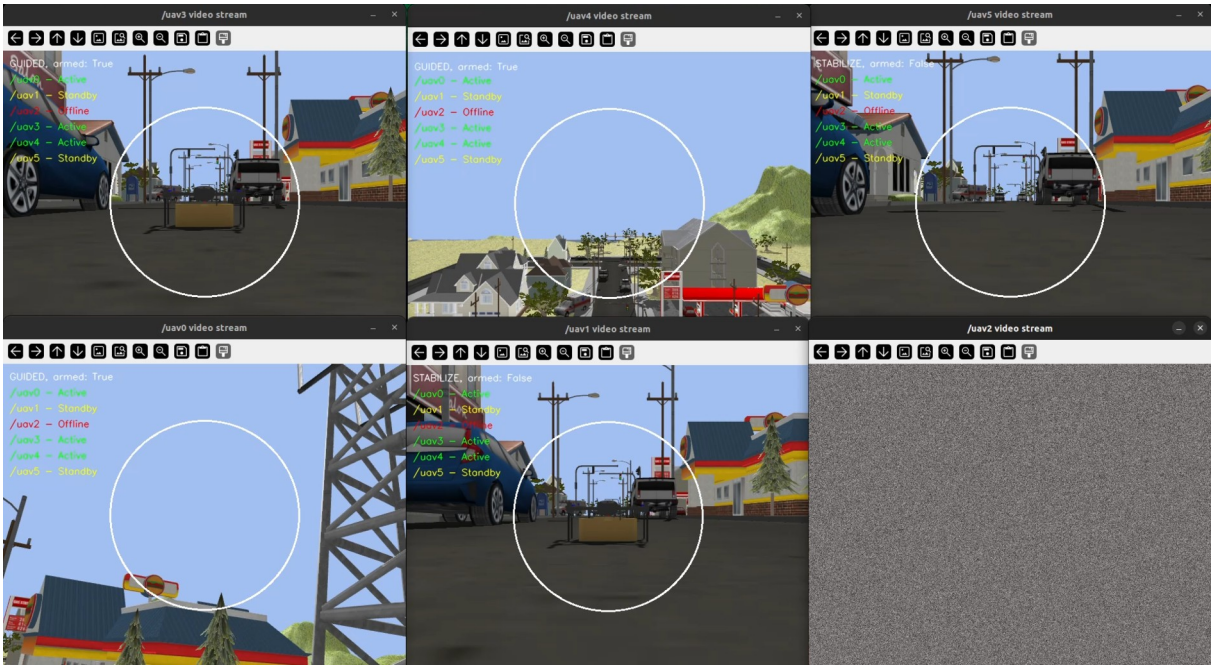


Рисунок 7.11 — Тестування сценарію для більшої кількості дронів

7.6 Тестування групової організації дронів у 2D режимі

Виконання симуляції дронів для великої кількості екземплярів може вимагати значних потужностей процесора, що в свою чергу ускладнює перевірку різних сценаріїв використання, застосовуючи слабкі персональні комп’ютери. Через це актуальною є можливість запуску подібних тестів на потужних серверах, наприклад, використовуючи хмарні платформи. При цьому доцільно економити кошти при використані подібних платних інфраструктур. Як варіант, можливо проводити симуляції без використання потужних графічних прискорювачів, обмежившись лише простою 2D графікою.

Для такого варіанту тестування основні фрагменти програмних модулів, що описувались раніше залишаються без змін. Єдина модифікації вноситься у запуск екземплярів віртуальних дронів при якому не використовуватиметься віртуальне середовище *Gazebo*. Програмне забезпечення *SITL Ardupilot* дає можливість симулювати апаратні складові дрону вбудованими засобами. Для цього необхідно доповнити файл “*vehicleinfo.py*” у директорії “*~/ROS/ardupilot/Tools/autotest/pysim*” відповідними параметрами екземплярів дронів як показувалось у розділі “Симуляція дронів на базі *Ardupilot*”.

Важливим тут є відсутність в описі дрону параметру “*"external": True,*”, який передає функції симуляції у *Gazebo*. Після внесення змін, описаний файл має виглядати подібним чином (рис. 7.12).

```
"quad0": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sysid-iris0.parm"],
},
"quad1": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sysid-iris1.parm"],
},
"quad2": {
    "waf_target": "bin/arducopter",
    "default_params_filename": ["default_params/copter.parm",
                                "default_params/sydid-iris2.parm"],
},
"quad3": {
    "waf_target": "bin/arducopter",
```

Рисунок 7.12 — Приклад параметрів для симуляції дронів через *SITL Ardupilot*

Надалі запуски всіх компонентів для симуляції одного екземпляру дрона було об’єднано у вигляді файлу запуску “*demo_swarm_2d_drone_launch.xml*” (рис. 7.13).

```
<?xml version="1.0"?>
<launch>

    <arg name="instance" default="0" />
    <arg name="sitl" default="ardupilot"/>

    <!-- Autopilot software -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/$(var sitl)_launch.xml">
        <arg name="instance" value="$(var instance)"/>
        <arg name="model" value="quad"/>
    </include>

    <!-- MAVROS node -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/mavros_sitl_$(var sitl)_node_launch.xml">
        <arg name="instance" value="$(var instance)"/>
    </include>

    <!-- Companion control node -->
    <include file="$(find-pkg-share simple_ros2_app)/launch/companion_control_node_launch.xml">
        <arg name="instance" value="$(var instance)"/>
        <arg name="sitl" value="$(var sitl)"/>
    </include>

    <!-- Swarm node -->
    <node pkg="simple_ros2_app" exec="swarm_control" namespace="/uav$(var instance)" output="screen" />

</launch>
```

Рисунок 7.13 — Файлу запуску одного екземпляру дрона

7.6.1 Запуск сценарію патрулювання для рою дронів

Запуск симуляції 6 дронів виконується через створений файл запуску, а відображення середовища забезпечується на карті у програмі *QGroudControl.* Після проходження зазначених вище процедур 3 дрони починають задачу патрулювання згідно закладеної програми (рис. 7.14).

Рисунок 7.14 — Три екземпляри дронів “*Vehicle 1-3*” починають програму патрулювання

Надалі при відмові і втраті зв’язку з певним дроном до його останньої позиції вилітає інший дрон, який надалі продовжує програму патрулювання зламаного дрону (рис. 7.15-7.18).

Оскільки протягом тесту сценарію віртуальні екземпляри дронів неодноразово перезапускаються, імітуючи заміну відмовивших дронів, це дозволяє виконати симуляцію нескінченної кількості дронів, які заміщують одне одного.

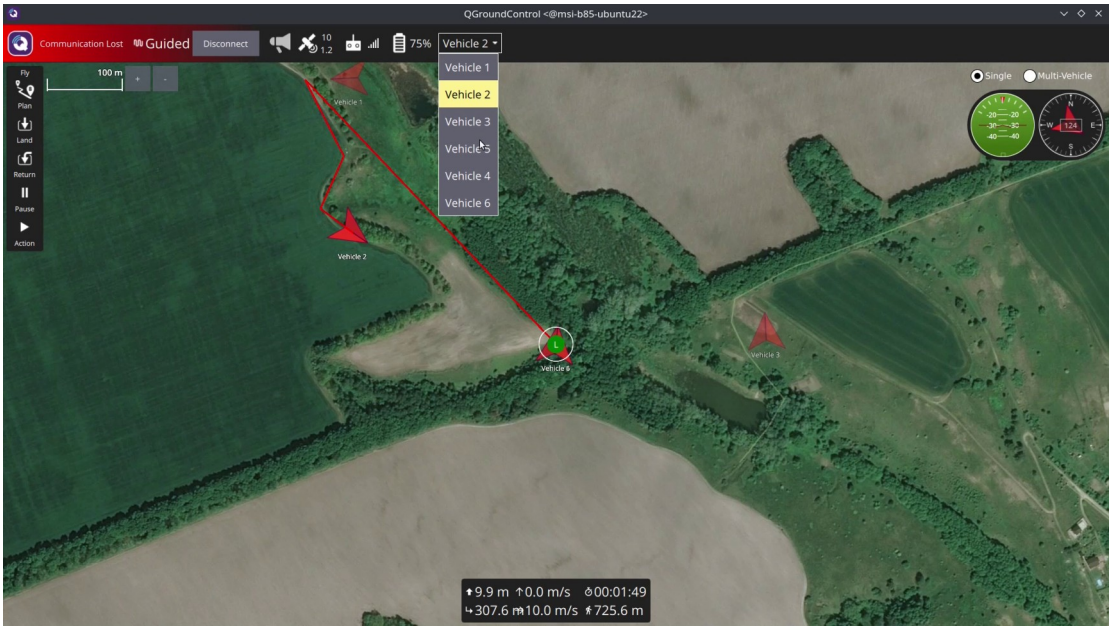


Рисунок 7.15 — Відмова дрону “*Vehicle 2*”

Рисунок 7.16 — Дрон “*Vehicle 5*” замінює дрон “*Vehicle 2*” у патрулі

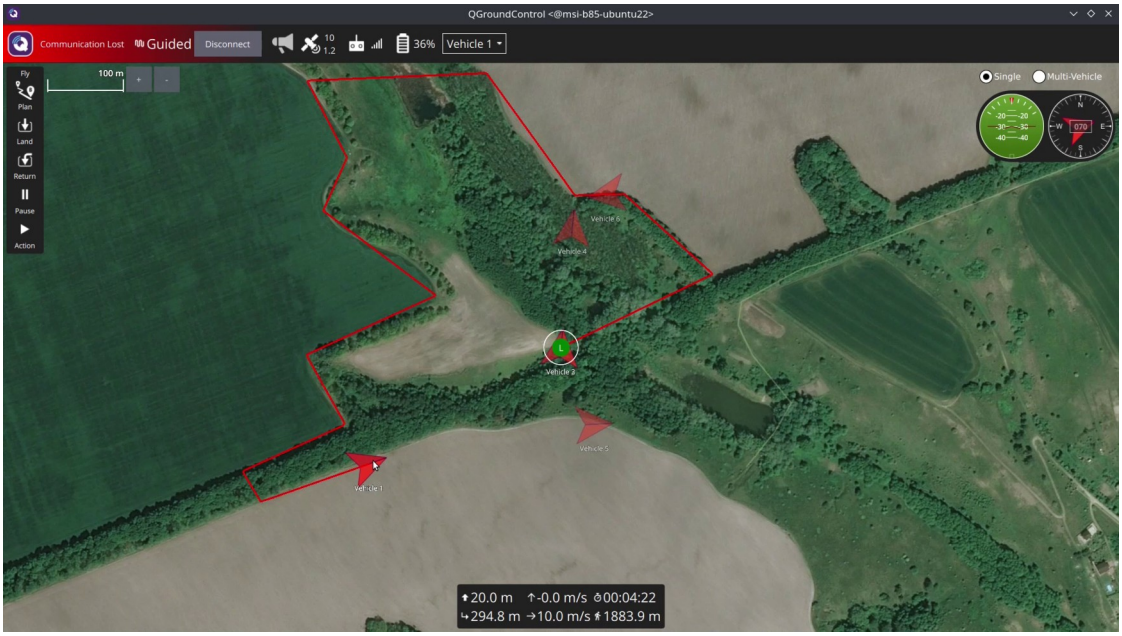


Рисунок 7.17 — Відмова дронів “*Vehicle 1*” (позиція лівіше) та “*Vehicle 6*” (позиція вище)

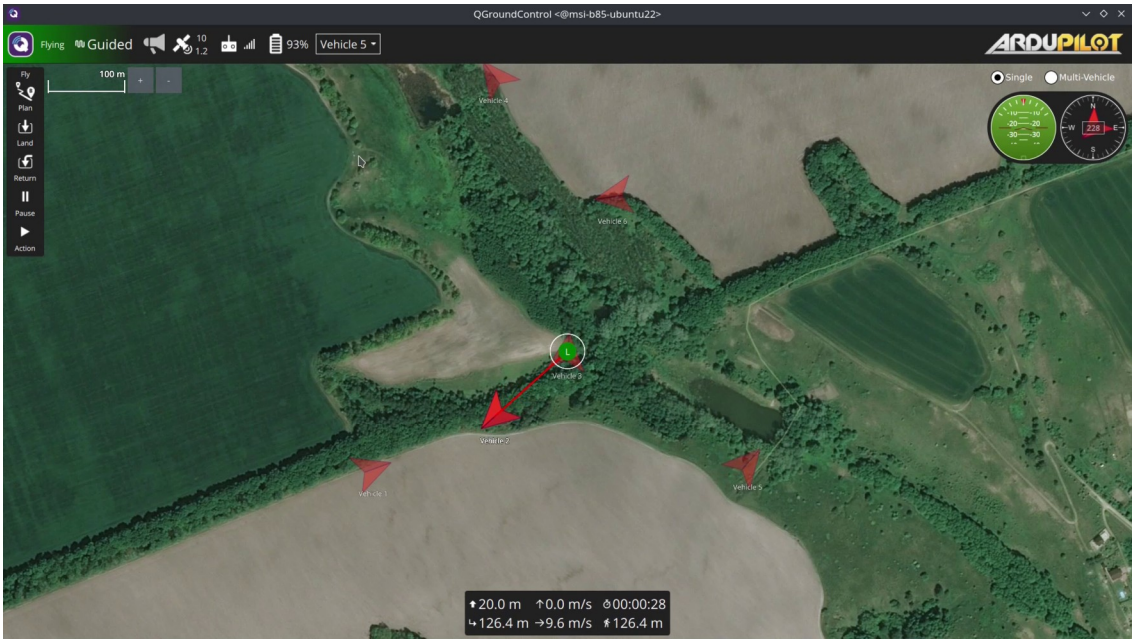


Рисунок 7.18 — Дрони “*Vehicle 2*” та “*Vehicle 4*” замінюють дронів “*Vehicle 1*” та “*Vehicle 6*” відповідно

8 ЗАСТОСУВАННЯ МЕТОДІВ КОМП’ЮТЕРНОГО ЗОРУ ДЛЯ КЕРУВАННЯ ДРОНОМ

В розділі використовуються наступні версії ОС та ПЗ:

- *Ubuntu Desktop* 22.04 *LTS*;
- *ROS2 Humble* 2025-02-11;
- *Ardupilot — ArduCopter* 4.3.3;
- *Gazebo Simulator* 11.10.2.

8.1 Захоплення відео потоку з камери *Raspberry Pi*

Захоплення відео потоку зі справжньої камери засобами *ROS2* та його подальша обробка особливо не відрізняється від аналогічного процесу, що розглядався для віртуальної камери, яка відображала віртуальне середовище у *Gazebo*. Єдина різниця полягає у тому, що тепер необхідно додатково розробити модуль, який зчитуватиме потік зі справжньої камери та направлятиме дані у форматі повідомлень через відповідну тему *ROS*.

Для прикладу цього процесу розглянемо захоплення відео потоку із камери, що підключена до одноплатного комп’ютеру *Raspberry Pi 3B+* через інтерфейс *MIPI CSI*. За замовчуванням для такої камери створюється окремий файл пристрою, наприклад, “*/dev/video0*”, який можна використовувати в якості відео пристрою. У програмному коді зчитування кадрів відео можна зробити через бібліотеку *OpenCV*. Надалі ці кадри конвертуються у повідомлення *ROS* та публікуються через окрему тему. Приклад цього процесу наведено на рис. 8.1.

Назва цієї теми може бути аналогічною темі, що використовує симулятор “*camera/image_raw*” для віртуальної камери. Це дозволяє замінювати цим модулем інформацію з камери для інших залежних модулів при реальному застосуванні.

```python
        # Pi camera
        self.CAMERA_FPS = 20
        self.pi_camera = cv2.VideoCapture('/dev/video0', cv2.CAP_V4L)
        self.pi_camera.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
        self.pi_camera.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)
        self.pi_camera.set(cv2.CAP_PROP_FPS, self.CAMERA_FPS)
        self.pi_camera.set(cv2.CAP_PROP_BUFFERSIZE, 1)

        # Register main control callback
        self.create_timer(1.0 / self.CAMERA_FPS, self.camera_processing_callback)

    def camera_processing_callback(self):  1 usage

        if self.skip_frame > 0:
            self.skip_frame -= 1
            return

        try:

            ret, frame = self.pi_camera.read()

            if ret:
                image_msg = self.cv_bridge.cv2_to_imgmsg(frame, "bgr8")

                self.drone_camera.publish(image_msg)

        except CvBridgeError as e:
            self.get_logger().info(e)

```

Рисунок 8.1 — Приклад зчитування кадрів із відео пристрою та конвертація у повідомлення *ROS*

8.2 Виділення графічних результатів обробки зображення в окремий модуль

У попередніх розділах вже наводився приклад модуля *ROS*, що підписувався на тему із кадрами з віртуальної камери у середовищі *Gazebo*, обробляв їх за певним алгоритмом та відображав результуюче зображення у графічному вікні. У тестових сценаріях це дозволяє зменшити кількість програмного коду, зосереджуючи всю логіку в одному модулі. Проте при реальному використанні можливі випадки, коли графічно відображати оброблене зображення не потрібно. Через це може бути доцільним винести частину функцій по відображенню зображень в окремий модуль *ROS*, що дозволить викликати його за потреби, залишаючи без змін інший модуль, який відповідатиме за обробку кадрів відео.

В такому разі аналогічний функціонал по відображенню кадрів, що базується на бібліотеці *OpenCV*, був перенесений в окремий спеціалізований модуль, модифікований та доповнений можливістю збереження отриманих кадрів в окремий відео файл. Як і раніше для цього виконується підписання на повідомлення *ROS* з теми із відео потоком, і збереження кадрів через відповідну функцію, так званий *callback*. Проте не слід забувати, що ця функція викликається лише при умові надходження нового повідомлення і в разі, якщо з боку відправника відбувається затримка у відправці, наприклад, через тривалу обробку даних, *callback*-функція буде виконуватись зі змінною частотою. З точки зору відображення відео з постійною частотою кадрів, наприклад, 20 кадрів на секунду, такий факт викликатиме підвисання графічного вікна при відображенні кадру, в разі якщо цю логіку додавати безпосередньо до тієї самої *callback*-функції. Тому, щоб покращити продуктивність цього модуля *ROS* в цілому, бажано модифікувати його виконання у багатопоточний формат. Це стосуватиметься як простого відображення поточного кадру у вікні програми, так і функції запису кадрів у відео файл.

```
class UICameraWindowNode(Node):

    def __init__(self, args):

        super().__init__("camera_window")
        self.drone_name = self.get_namespace()

        # Camera topic subsription
        self.cv_bridge = CvBridge()

        qos_profile = qos.QoSProfile(reliability=qos.ReliabilityPolicy.BEST_EFFORT,
                                     history=qos.HistoryPolicy.KEEP_LAST,
                                     depth=1)  # was depth=5
        # Subscriptions
        self.create_subscription(Image, "{}/camera/image_detection".format(self.drone_name),
                                 self.frame_processing, qos_profile, callback_group=MutuallyExclusiveCallbackGroup())

        self.mode = ''
        self.skip_frame = 5
        self.frame_center = [320, 240]

        # Recording
        from datetime import datetime
        self.RECORDING_FPS = 20
        self.camera_recording = cv2.VideoWriter(datetime.now().strftime('recording-%Y-%m-%d_%H:%M:%S.mp4'),
                                                cv2.VideoWriter.fourcc(*'avc1'), self.RECORDING_FPS, (640, 480), True)
        self.camera_last_window_frame = None
        self.camera_last_frame = None
        self.camera_last_frame_time = None
        self.camera_frame_buffer = []
        self.camera_frame_buffer_sync = []

        self.recording_interval = 1.0 / self.RECORDING_FPS
        # self.recording_interval_x2 = self.recording_interval
        # Enable recording
        self.create_timer(self.recording_interval, self.camera_recording_callback, callback_group=MutuallyExclusiveCallbackGroup())

        self.window_name = "{} tracking flow stream".format(self.drone_name)
```

Рисунок 8.2 — Фрагмент коду ініціалізації модуля для відображення кадрів відео

Для відображення кадру у графічному вікні створюється окремий потік засобами пакету “*threading*”, який надалі виконує відображення кадрів із заданою частотою. У випадку, якщо на час чергового оновлення вікна нового кадру не надходило, відображатиметься останній збережений кадр.

```
208 def show_opencv_ui(client_node, stop_event):
209
210     while not stop_event.is_set():
211         if client_node.camera_last_window_frame is not None:
212             cv2.imshow(client_node.window_name, client_node.camera_last_window_frame)
213             # cv2.waitKey(1000)
214             cv2.waitKey(int(client_node.recording_interval * 1000))
215         else:
216             time.sleep(1)
```

Рисунок 8.3 — Фрагмент коду для відображення графічного вікна із кадром

8.2.1 Запис кадрів у відео файл

Аналогічна ідея покладена і в основу алгоритму запису кадрів в окремий відео файл. Засоби бібліотеки *OpenCV* дозволяють доволі легко організувати цей процес через виклик декількох вбудованих процедур. Але в зазначеному випадку їх просте використання також призводитиме до артефактів у записаному відео. Річ у тім, що під час створення об’єкту “*cv2.VideoWriter*”, вказування параметру “частоти кадрів” призводить лише до відтворення записаного відео файлу із цією заданою кількістю кадрів на секунду, але ніяким чином не синхронізує кадри під час додавання чергового кадру у файл. Через це, в разі якщо процедура запису кадрів виконується із затримками або непостійною частотою, це призводитиме до візуально нерівномірної швидкості зміни кадрів у записаному відео, так званий ефект «таймлапсу». Тобто якщо у модуль *ROS* повідомлення із кадрами надходитимуть раз на 5 секунд, а потрібно записувати відео файл із частотою 20 кадрів на секунду, то простий запис першого кадру, а потім через 5 секунд — наступного, який прийшов із затримкою, у записаному відео виглядатиме як просто дуже швидке відтворення цих двох кадрів в межах першої секунди у відео файлі (рис. 8.4).

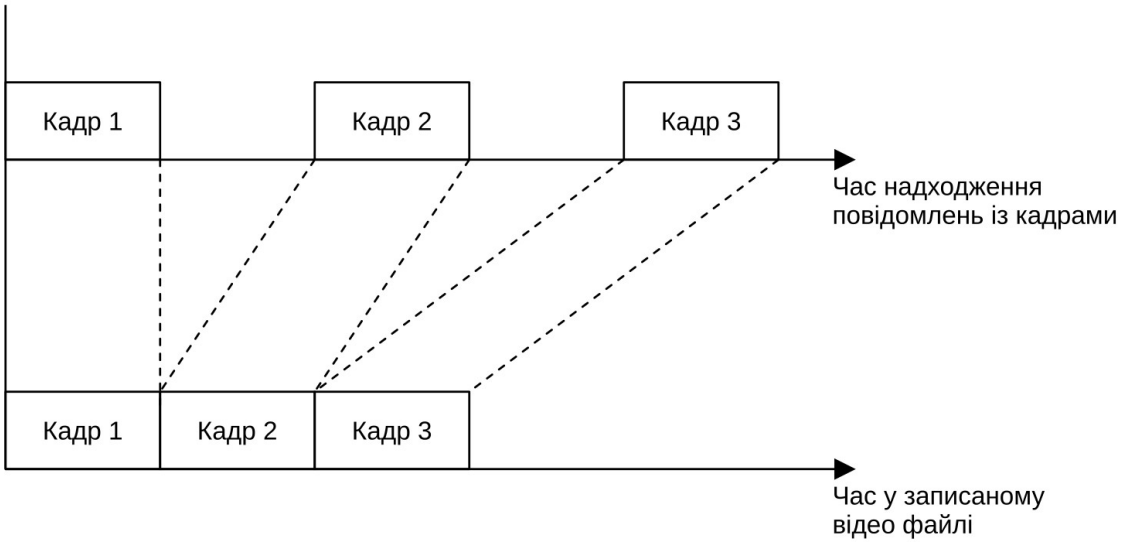


Рисунок 8.4 – Часова діаграма при записі кадрів без синхронізації часу надходження повідомлення

8.2.2 Реалізація синхронізації кадрів з часом запису

Щоб уникнути зазначеного часового артефакту при записі відео файлу необхідно доповнити цю процедуру додатковою синхронізацією кадрів за часом їх надходження. Відповідно для досягнення цієї мети можна доповнити програмний код, який записує кадри, буфером, що накопичуватиме кадри, які надходять із відповідної теми *ROS*, а також дозволятиме відстежувати час коли надійшов черговий кадр. За рахунок цього можна синхронізувати час у відео файлі із часом за яким надходять нові кадри. В такому разі, відбувається відкладений запис у відео файл. Із буферу обирається поточний кадр, час його надходження порівнюється із часом наступного кадру в цьому буфері. Якщо він більший ніж час чергового кадру, який має бути записаний у відео файл, враховуючи задану частоту відтворення, виконується повторний запис поточного кадру з буферу. В іншому разі обирається наступний кадр з буферу і повторюється аналогічна логіка (рис. 8.5).

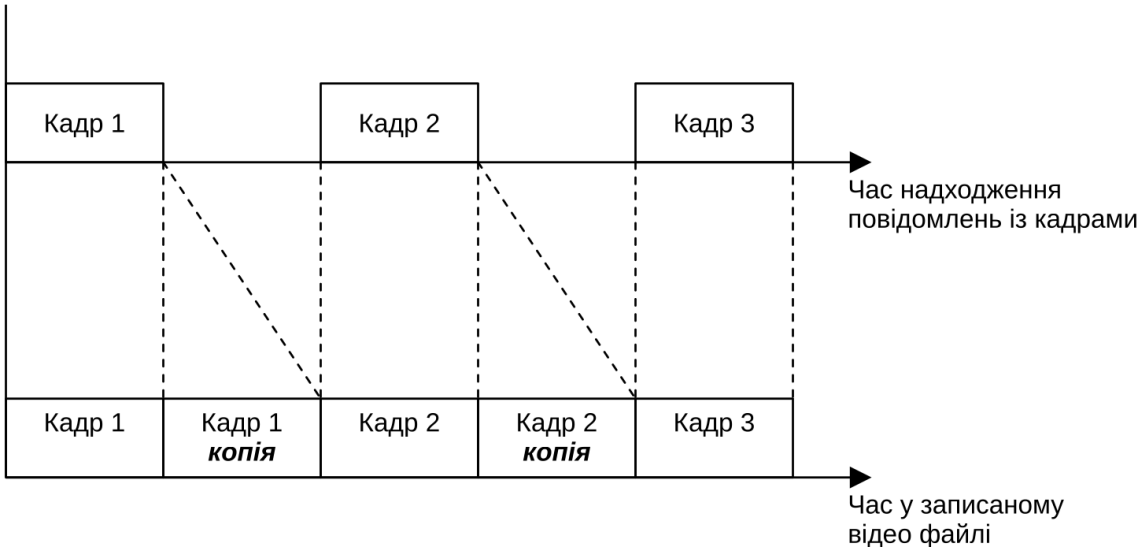


Рисунок 8.5 — Часова діаграма при записі буферизованих кадрів із синхронізацією часу надходження повідомлення

Програмна реалізація описаного алгоритму наведено на рисунку нижче.

```
118
119     def record_frame(self):
120
121         if len(self.camera_frame_buffer) > 0:
122             # have frames in buffer
123             if self.camera_last_frame is not None:
124                 # check next frame time
125                 next_frame_time = self.camera_last_frame_time + self.recording_interval
126                 if self.camera_frame_buffer_sync[0] - next_frame_time < 0.005:
127                     # read next frame from buffer
128                     self.camera_last_frame = self.camera_frame_buffer.pop(0)
129                     self.camera_last_frame_time = self.camera_frame_buffer_sync.pop(0)
130                 else:
131                     # update time for last frame
132                     self.camera_last_frame_time = next_frame_time
133             else:
134                 self.camera_last_frame = self.camera_frame_buffer.pop(0)
135                 self.camera_last_frame_time = self.camera_frame_buffer_sync.pop(0)
136             self.camera_recording.write(self.camera_last_frame)
137         else:
138             if self.camera_last_frame is not None:
139                 self.camera_recording.write(self.camera_last_frame)
                    # update time for last frame
                    self.camera_last_frame_time += self.recording_interval
```

Рисунок 8.6 — Фрагмент коду для запису буферизованих кадрів у відео файл

8.3 Передача команд на пересування для політного контролеру

У попередніх розділах вже розглядались деякі політні режими автопілотів дронів (*PX4 – OFFBOARD*, *Ardupilot – GUIDED*), які дозволяли направляти його в бажані координати, при цьому інші параметри такі, як необхідні оберти двигунів, швидкість, орієнтація у просторі та інше розраховував політний контролер автоматично. Подібні режими зручні в разі якщо у польоті використовується система позиціонування — *GPS* і відповідно дрон може визначати поточні координати і слідувати у потрібному напрямі. Проте в деяких сценаріях є потреба напряму керувати дроном, аналогічно тому як це відбувалося б при використанні простого пульта керування. В такому разі нас цікавитиме не передача політних координат у які потрібно летіти дрону, а більш прості команди, наприклад, летіти прямо/назад, вправо/вліво, обертання, зміна висоти.

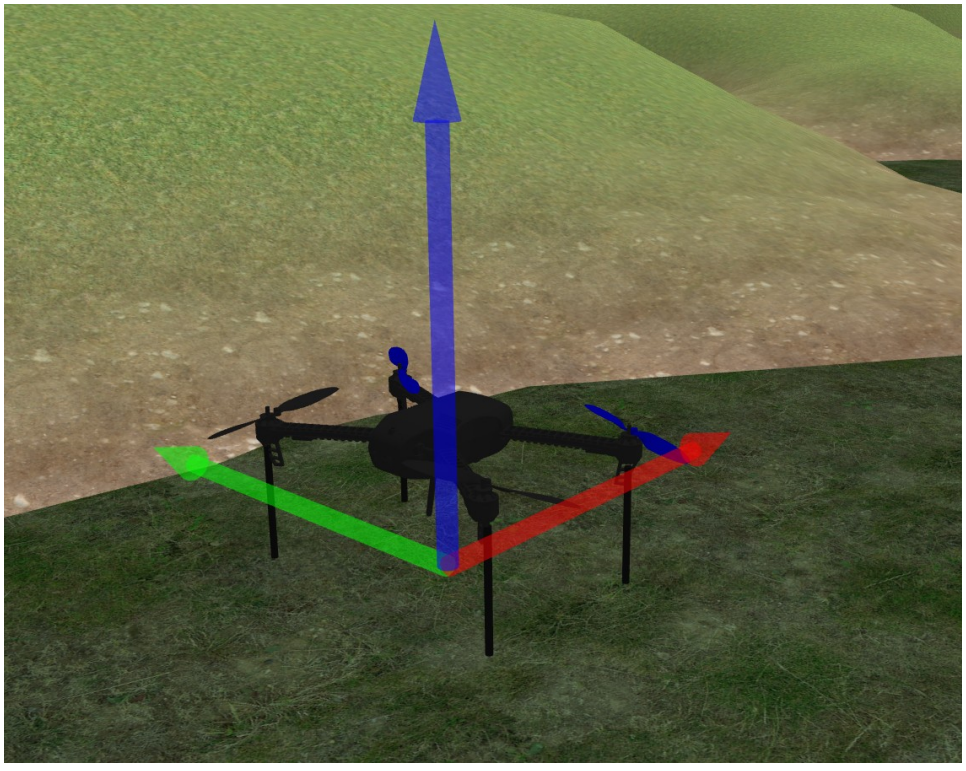


Рисунок 8.7 — Безпосереднє керування напрямом руху дрону

У згаданих автопілотах дронів є подібні спеціалізовані політні режими із можливістю прямого керування, наприклад, *ALT_HOLD* у *Ardupilot*. На відміну від режиму *GUIDED*, тепер дрон не відслідковує свої координати і відповідно не має можливості чітко утримувати позицію в разі, якщо команди на пересування перестають передаватись. В цьому режимі, якщо перестати передавати команди, що є аналогічним виставленню у центральні позиції елементів керування на пульті, дрон продовжуватиме рух по інерції за попередньою траєкторією. Відповідно, щоб зупинити його треба буде подати команди протилежні цьому руху — стабілізувати позицію вручну. Однак згідно назви цього політного режиму, дрон все таки має можливість самостійно утримувати свою висоту за рахунок вбудованих сенсорів, наприклад, барометру. Подібний режим є доволі гнучким в плані керування, проте якщо є можливість використовувати *GPS*, режим *LOITER* у *Ardupilot*, буде більш зручним, оскільки стабілізація позиції виконуватиметься автоматично політним контролером, що спрощуватиме алгоритм керування.

8.3.1 Повідомлення *MAVROS* для ручного керування

Оскільки радіус дії пульта керування є обмеженим, бажано мати можливість відправляти політному контролеру аналогічні команди програмним шляхом, що в подальшому дозволить застосувати це у алгоритмі керування за рахунок використання комп’ютерного зору. У *MAVLink* передбачене відповідне повідомлення *MANUAL_CONTROL*, через яке можна програмно передавати команди на пересування, що є подібним до пульта керування. Засобами бібліотеки *MAVROS* таке повідомлення генеруватиметься через відправку повідомлення *ROS* у тему “*mavros/manual_control/send*”. Однією особливістю формату повідомлення у згаданих політних режимах є необхідність передавати значення 500 для частини, яка відповідає за висоту — сигнал “z”, щоб дрон утримував поточну висоту. Як результат, значення менше цієї величини призводитимуть до зниження висоти польоту дрону, більше — збільшення. Що стосується польоту в інших площинах: вперед/назад — сигнал

“*x*”, ліво/право — сигнал “*y*”, обертання ліворуч/праворуч — сигнал “*r*”, то для руху у відповідних напрямах необхідно встановлювати значення більше/менше нуля у відповідних частинах повідомлення.

8.3.2 Керування дроном через клавіатуру

Враховуючи вищесказане тепер є можливість реалізувати подібний режим керування програмно у вигляді модуля *ROS*. Відповідно замість справжнього пульта керування з’явиться можливість додати до дрону функцію керування через клавіатуру комп’ютеру. Для цього застосовуватиметься пакет “*pynput*”, який дозволяє зчитувати натискання клавіш.

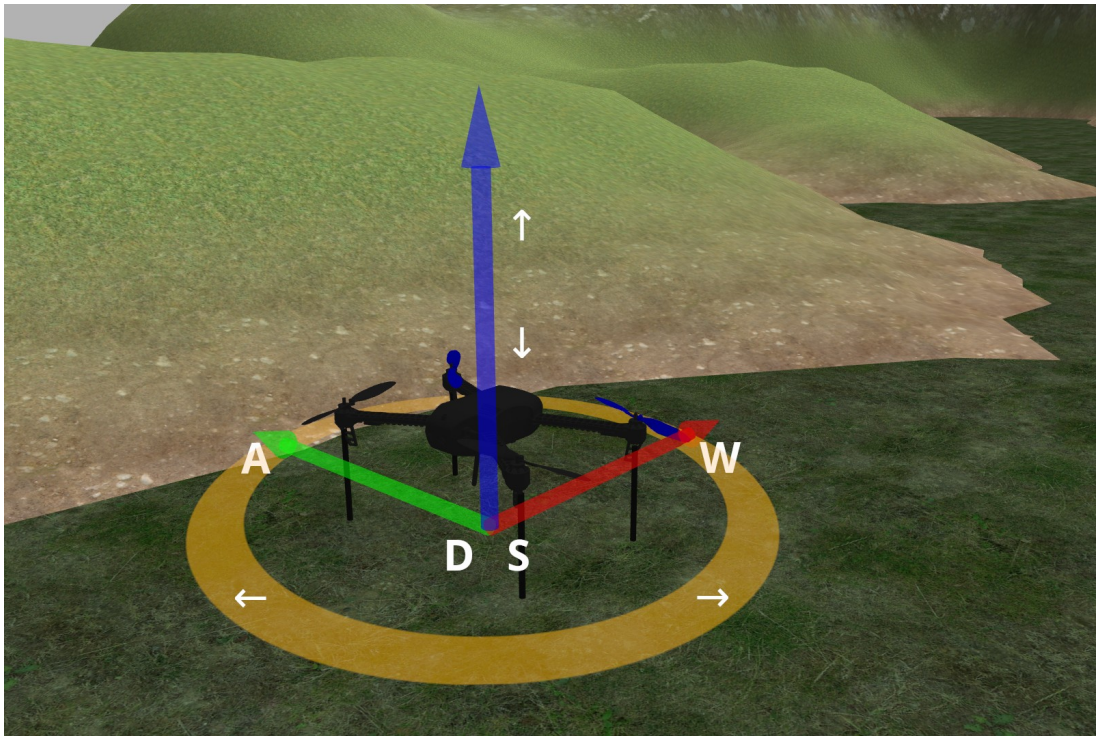


Рисунок 8.8 — Команди керування дроном через клавіатуру

Для відповідних напрямів руху дрону можна обрати клавіші клавіатури аналогічно тим, які використовуються у більшості комп’ютерних симуляторів. Клавіші “*WASD*” для команд руху “вперед-вбік”, клавіші “←→↑↓” — “зміна висоти/обертання навколо центральної вісі”. Фрагмент програмного коду, який демонструє зчитування натискання клавіш наведено на рис. 8.9.

```
    def key_press_callback(self, key):
        self.key_press[key] = 1.0

    def key_release_callback(self, key):
        self.key_press[key] = 0.0

    def enable_full_keyboard_callback(self, key):...

    def switch_full_key_listener(self):
        if self.full_key_listener is None:
            self.full_key_listener = keyboard.Listener(on_press=self.key_press_callback,
                                                       on_release=self.key_release_callback, suppress=True)
            self.full_key_listener.start()
            return True

        else:
            self.full_key_listener.stop()
            self.full_key_listener = None
            return False
```

Рисунок 8.9 — Фрагмент коду для зчитування натискання клавіш

Надалі зареєструвавши чергове натискання клавіш, його можна конвертувати у повідомлення для керування дроном. Ця процедура є аналогічною тій, що використовувалась у попередніх розділах про керування дроном через відправку політних координат, проте тепер замість політних координат відправлятиметься повідомлення із сигналами керування напрямами руху у відповідну тему *MAVROS*. Для цього використовуються певні константи для кожного із напрямів руху, які додаються до відповідних сигналів в разі натискання клавіш на клавіатурі (рис. 8.10).

```
            if self.arming_async_call is not None and self.arming_async_call.done():
                self.get_logger().info("Vehicle armed")
                self.arming_async_call = None
                self.failsafe_lock_switch_mode = True

        if not self.keyboard_ignore and self.is_armed():
            x = self.MOVE_FORWARD * self.key_press[self.KEY_W] + self.MOVE_BACKWARD * self.key_press[self.KEY_S]
            y = self.MOVE_LEFT * self.key_press[self.KEY_A] + self.MOVE_RIGHT * self.key_press[self.KEY_D]
            z = self.MOVE_UP * self.key_press[self.KEY_UP] + self.MOVE_DOWN * self.key_press[self.KEY_DOWN]
            r = self.ROT_LEFT * self.key_press[self.KEY_LEFT] + self.ROT_RIGHT * self.key_press[self.KEY_RIGHT]

            self.send_manual_control(x, y, 500.0 + z, r, buttons: 0)
```

Рисунок 8.10 — Фрагмент коду для формування сигналів для керування напрямом руху

8.3.3 Перемикання режимів трекінгу/ручного керування

Також використання клавіатури дозволяє розширити можливість керування функціями інших модулів *ROS*, як наприклад, вмикати/вимикати функцію керування засобами комп’ютерного зору, за рахунок аналогічного відстеження натискання деяких клавіш і відправки відповідних повідомлень до модулів *ROS* (рис. 8.11).

```
129         if self.key_press[self.KEY_T] > 0.0:
130             self.get_logger().info("Switching tracking")
131             self.key_press[self.KEY_T] = 0.0
132             self.keyboard_ignore ^= True
133             self.tracker_control.publish(self.tracker_msg)
134 
135         if self.key_press[self.KEY_R] > 0.0:
136             self.get_logger().info("Resetting tracking")
137             self.key_press[self.KEY_R] = 0.0
138             self.tracker_reset_control.publish(self.tracker_msg)
```

Рисунок 8.11 — Фрагмент коду для перемикання функцій іншого модуля *ROS*

8.3.4 Зчитування сигналів пульта радіо керування

Засобами *MAVROS* передбачено можливість зчитувати сигнали із пульта радіо керування через окрему тему “*mavros/rc/in*”. Це в свою чергу дозволяє реалізувати керування власними модулями *ROS*, не тільки через взаємодію з ними через вбудовані програмні утиліти, а і через сигнали, наприклад, тумблерів на пульті керування (рис. 8.12).

```python
    def rc_callback(self, msg):
        if not self.kb_mode_enabled:
            if not self.rc_mode_enabled and int(msg.channels[self.rc_control_mode_index]) > 1100:
                if self.switch_full_key_listener():
                    self.rc_mode_enabled = True
                    self.get_logger().info("RC control enabled")

            if self.rc_mode_enabled and int(msg.channels[self.rc_control_mode_index]) < 1800:
                if not self.switch_full_key_listener():
                    self.rc_mode_enabled = False
                    self.get_logger().info("RC control disabled")

            if int(msg.channels[self.rc_tracking_mode_index]) > 1600:
                if int(msg.channels[self.rc_tracking_mode_index]) > 1800:
                    if not self.rc_switch_tracking:
                        self.rc_switch_tracking = True
                        self.key_press[self.KEY_T] = 1.0
```

Рисунок 8.12 — Фрагмент коду для зчитування сигналів пульта радіо керування

8.4 Керування засобами комп’ютерного зору

Сфера комп’ютерного зору включає в себе методи для обробки візуальних зображень та виокремлення з них корисної інформації. В якості однієї із підзадач можна розглянути детекцію об’єктів, мета якої знайти певний об’єкт та визначити його положення на зображенні. В задачі детекції об’єктів наразі широко застосовуються глибокі нейронні мережі, які з високою точністю дозволяють виявляти об’єкти різних класів на зображеннях. Але беручи до уваги, що обчислювальні ресурси на борту дрону зазвичай невеликі, це може обмежувати їх використання на малопотужних пристроях, наприклад, на комп’ютері компаньйоні, що знаходиться на борту.

З іншого боку, задача трекінгу є похідною від задачі детекції об’єктів. Її ідея полягає у слідкуванні/виявленні певного об’єкту на серії кадрів — зображень на відео. Порівнюючи із детекцією, трекінг об’єкту потребує менше обчислювальних ресурсів, оскільки в такому разі наявною є інформація про положення об’єкту, його напрям та швидкість переміщення на попередніх кадрах, що в подальшому полегшує його виявлення на наступних кадрах відео. Напротивагу детекція об’єкту завжди виконується відносно одного поточного кадру, незважаючи на інформацію з попередніх кадрів.

8.4.1 Використання алгоритмів трекінгу та нейронних мереж у керуванні дроном

В контексті безпілотних літальних апаратів, їх доповнення можливістю визначати об’єкти на зображеннях, що надходять з бортової камери, дозволить побудувати комплексну систему керування дроном засобами комп’ютерного зору. Враховуючи обмеженість обчислювальних потужностей на борту дрону, застосування алгоритмів трекінгу є непоганим рішенням. Порівняно із детекцією глибокими нейронними мережами, їх використання потребує менші витрати ресурсів. Так, алгоритм трекінгу *MedianFlow* може обробляти 40-50 кадрів розміром 640*x*640 на секунду, працюючи при цьому на одноплатному комп’ютері *Raspberry Pi 3B+*. Проте у даного класу алгоритмів є важливе обмеження. Для початку роботи їх треба ініціалізувати первинним положенням об’єкту, переміщення якого надалі вони визначатимуть на наступних кадрах. Відповідно, залишається потреба у глибоких нейронних мережах, які здатні знайти будь-який об’єкт на зображенні. Таким чином нейронна мережа може початково знаходити об’єкт на зображенні, а потім передавати його положення до алгоритму трекінгу для його подальшого відслідковування.

Нейронна мережа сімейства *YOLO* це потужний алгоритм для детекції об’єктів, який з високою точністю дозволяє виявляти об’єкти різних класів на зображеннях. Її найменша версія *YOLOn* може виконувати обробку одного зображення в межах 1 секунди на *Raspberry Pi 3B+*. Такої продуктивності звісно не достатньо для обробки відео потоку, проте цілком вистачає для автоматичної ініціалізації більш швидкого алгоритму трекінгу.

8.4.2 Реалізація модуля відстеження об’єкту комп’ютерним зором та формування політних команд

Описаний підхід у відстеженні положення об’єкта був реалізований у окремому модулі *ROS* “*tracking_control_node*” для керування дроном засобами комп’ютерного зору. В реалізації використано підхід із відправкою дрону

команд на пересування аналогічно тому, як це описувалось для модуля керування через клавіатуру, але відповідні сигнали на переміщення динамічно розраховуються відносно положення виявленого об’єкту (рис. 8.13). В частині обробки зображень із бортової камери використовується алгоритм трекінгу із автоматичною ініціалізацією за допомогою детекції об’єкту глибокою нейронною мережею.

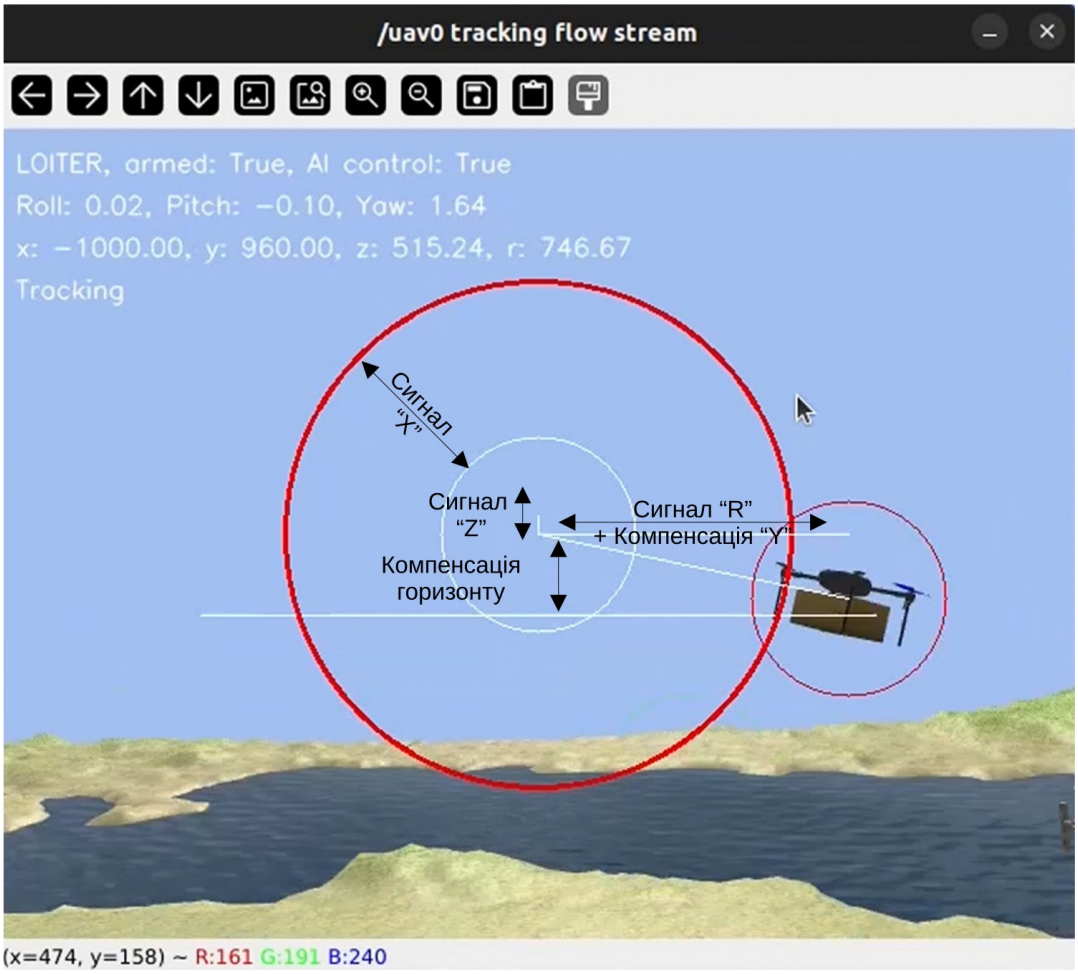


Рисунок 8.13 – Конвертація положення об’єкту у команди напряму польоту

На початку нейронна мережа *YOLO* виконує детекцію всіх об’єктів певних класів на зображенні, після чого автоматично обирається об’єкт який потрапляє у центральну область поля зору дрону і є найближчим до центру серед інших. Положення цього об’єкту відносно поля зору камери дрону передається до алгоритму трекінгу, який надалі починає відслідковувати

переміщення цього об’єкту. Фрагмент коду для обробки положення об’єкту наведено нижче.

```
    def frame_detection(self, msg):

        if self.skip_frame > 0:
            self.skip_frame -= 1
            return

        try:
            frame = self.cv_bridge.imgmsg_to_cv2(msg, "bgr8")

            # Tracking
            if not self.track_object:
                # detect object
                cv2.putText(frame, "Detecting", (10, 125),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)
                pred, pred_all = self.detector.update(frame, self.frame_center, 0)
                pred_fail = None
                if pred is not None:
                    # track object
                    self.track_object = True
                    # try init tracker
                    self.tracker.restart(frame, pred)
            else:
                # track object
                pred_all = None
                pred, pred_fail = self.tracker.update(frame)
                if pred is None:
                    # switch to detection
                    cv2.putText(frame, "Localizing", (10, 125),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)
                    pred, pred_all = self.detector.update(frame, self.track_object_prev[:2], 1)
                    if pred is not None:
                        # reinit tracker
                        self.tracker.restart(frame, pred)
                    else:
                        # tracking lost
                        self.track_object = False
                else:
                    cv2.putText(frame, "Tracking", (10, 125),
                                cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE_AA)

            # save new position
            self.track_object_prev = pred
```

Рисунок 8.14 — Фрагмент програмного коду для відслідковування об’єкту в полі зору камери дрону

Надалі положення виявленого об’єкту конвертується у команди на переміщення дрону та відправляються до політного контролеру (рис. 8.15).

```
        # control commands
        if self.auto_tracking and self.state.armed:
            if pred is not None and not self.failsafe_auto_tracking:
                # Forward
                self.manual_ctl.x = self.X_MAX * ((self.AUTO_RADIUS - min(pred[2:4])) / self.AUTO_RADIUS)

                # Y-shift compensation
                self.manual_ctl.y = self.Y_MAX * (pred[0] - self.frame_center[0]) / self.AUTO_RADIUS

                # Z-altitude
                self.manual_ctl.z = 500.0 + (self.Z_DIFF_MAX * (self.frame_center[1] - self.Z_COMPENSATE - pred[1]) / self.AUTO_RADIUS)

                # Rotation
                self.manual_ctl.r = self.ROT_MAX * (pred[0] - self.frame_center[0]) / self.AUTO_RADIUS

                # Failsafe control signals limits
                self.manual_ctl.x = max(min(300.0, self.manual_ctl.x), -300.0)
                self.manual_ctl.y = max(min(150.0, self.manual_ctl.y), -150.0)
                self.manual_ctl.z = max(min(750.0, self.manual_ctl.z), 250.0)
                self.manual_ctl.r = max(min(150.0, self.manual_ctl.r), -150.0)
            else:
                # Idle control
                self.manual_ctl.x = 0.0
                self.manual_ctl.y = 0.0
                self.manual_ctl.z = 500.0
                self.manual_ctl.r = 0.0

                # Failsafe tracking disable on object loss
                self.failsafe_auto_tracking = True
                #

            self.send_manual_control(self.manual_ctl)
```

Рисунок 8.15 — Фрагмент програмного коду для конвертації положення об’єкту у команду напряму польоту

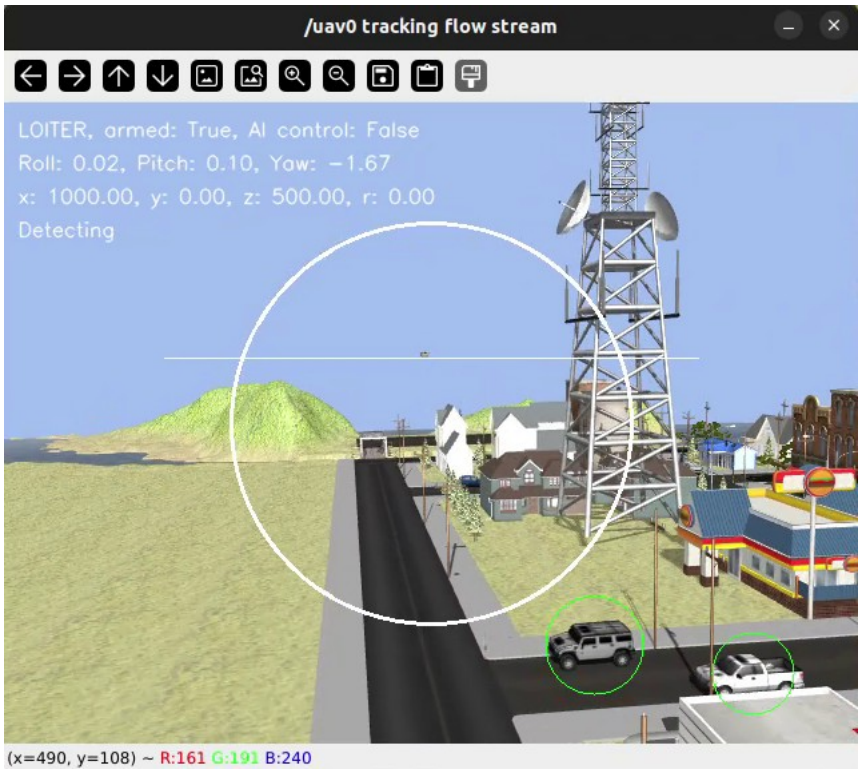


Рисунок 8.16 — Приклад детекції об’єктів певних класів нейронною мережею — виділено «зеленим» колом

В разі якщо алгоритм трекінгу втрачає об’єкт, наприклад, відбувається його різке зміщення, виконується автоматична корекція положення за рахунок детекції всіх об’єктів нейронною мережею та обрання з них найближчого за положенням до попереднього відомого об’єкту трекінгу (рис. 8.17).

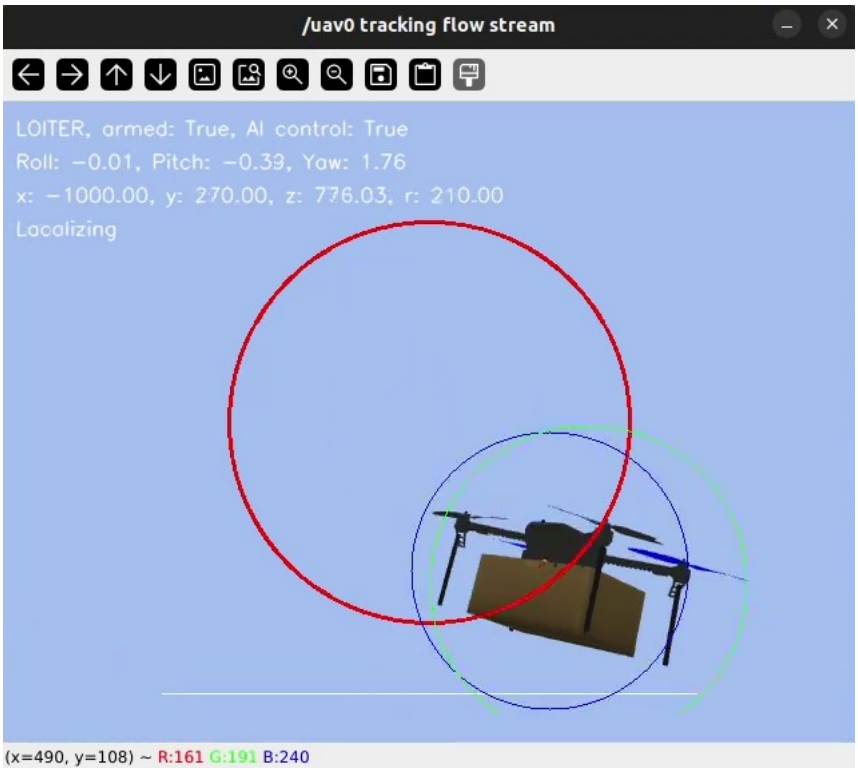


Рисунок 8.17 — Приклад автоматичної корекції положення об’єкту трекінгу — попереднє положення виділено «синім» колом

ОПИС ЛАБОРАТОРНИХ РОБІТ В РАМКАХ НАВЧАЛЬНОЇ ДИСЦИПЛІНИ

В цьому розділі наведений перелік лабораторних робіт, що покривають практичну складову в рамках вивчення дисципліни “Операційна система для роботів (ROS)” магістратури 1-го курсу. Опис включає перелік завдань до кожної лабораторної та додаткову інформацію по налаштуванню окремих компонентів, представлених у теоретичній частині.

Перелік лабораторних, завдання та вимоги наведені з метою ознайомлення та можуть корегуватись безпосередньо на онлайн ресурсі дисципліни, щоб відображати актуальні зміни у навчальному процесі.

Вимоги до оформлення результатів роботи

1. У протоколі детально описуються та пояснюються кроки, які були зроблені для виконання завдання лабораторної. Це може включати текст, скріншоти результатів, фрагменти коду, тощо;
2. В разі якщо виконання завдання потребує налаштування віртуальної машини, її “*хостнейм*” робиться згідно схеми: “*Ваше ім'я-прізвище-група-назва віртуалки*”;
3. Для отриманого результату записується невелика відео-демонстрація (аналогічно до демонстрацій, які надаються до методичних рекомендацій на онлайн ресурсі дисципліни — “*demo*”).

Лабораторна робота 1. Підготовка та налаштування інфраструктури *ROS2*

Мета роботи: ознайомитись із програмним забезпеченням для роботи із віртуальними моделями дронів.

Завдання:

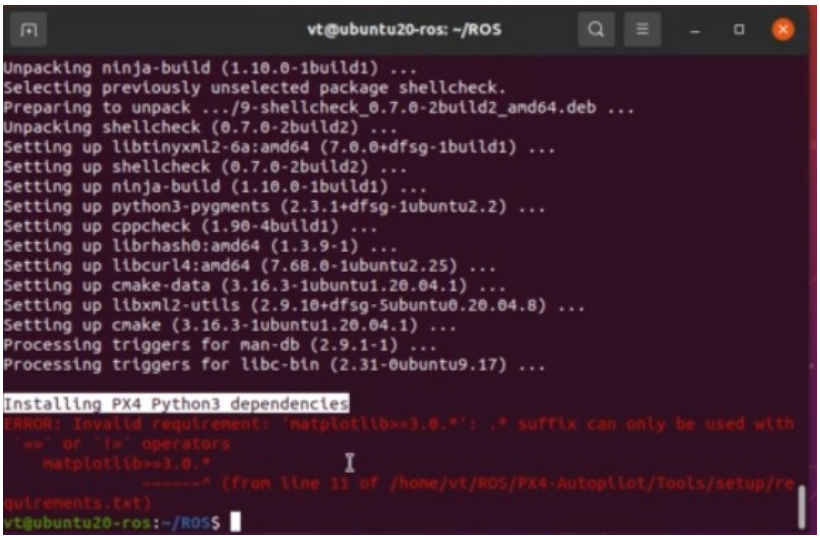
1. ознайомитись та налаштувати симулятор автопілоту на базі *Ardupilot;*
2. ознайомитись та налаштувати програмне забезпечення *QGroundControl* для віддаленого керування дроном;
3. ознайомитись та налаштувати симулятор автопілоту на базі *PX4*;
4. ознайомитись із симулятором віртуального середовища *Gazebo*;
5. ознайомитись із *ROS* 1-ї версії, розглянути її архітектурні відмінності від *ROS* 2.

Порядок виконання

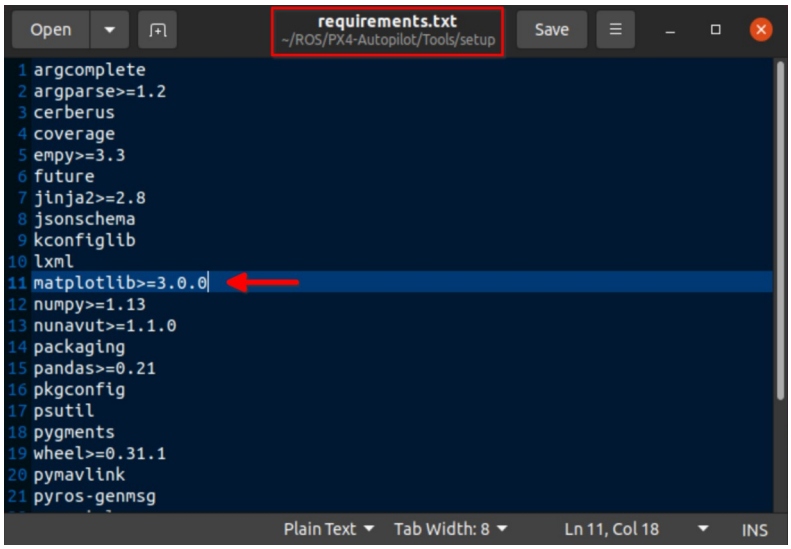
- Детальний опис кроків для виконання завдань 1 та 2 представлено у розділі “Симуляція дронів на базі *Ardupilot*”.
- Завдання 3-5 — у розділі “Симуляція дронів на базі автопілоту *PX4* із використанням *ROS*”.

Вирішення можливих помилок в ході налаштувань

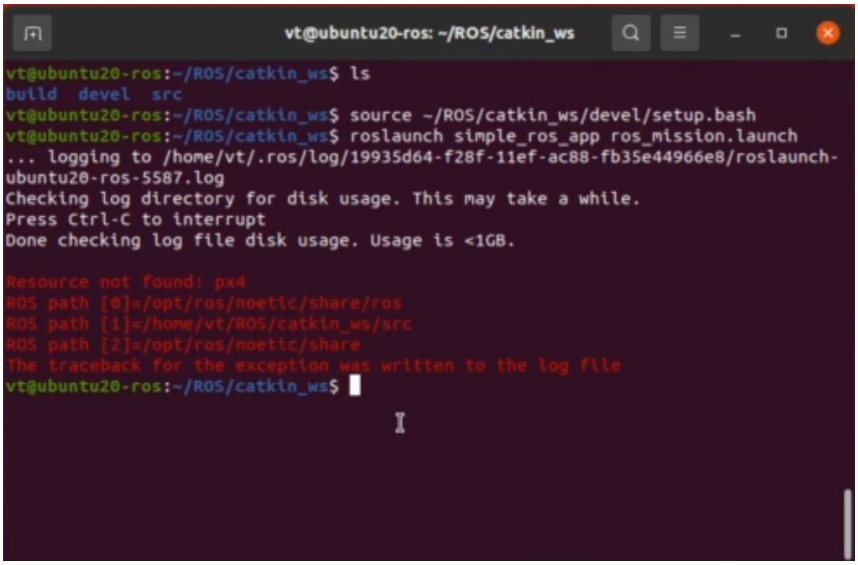
- Під час встановлення залежностей PX4 командою: *bash ./PX4-Autopilot/Tools/setup/ubuntu.sh --no-sim-tools --no-nuttx* виникає помилка.



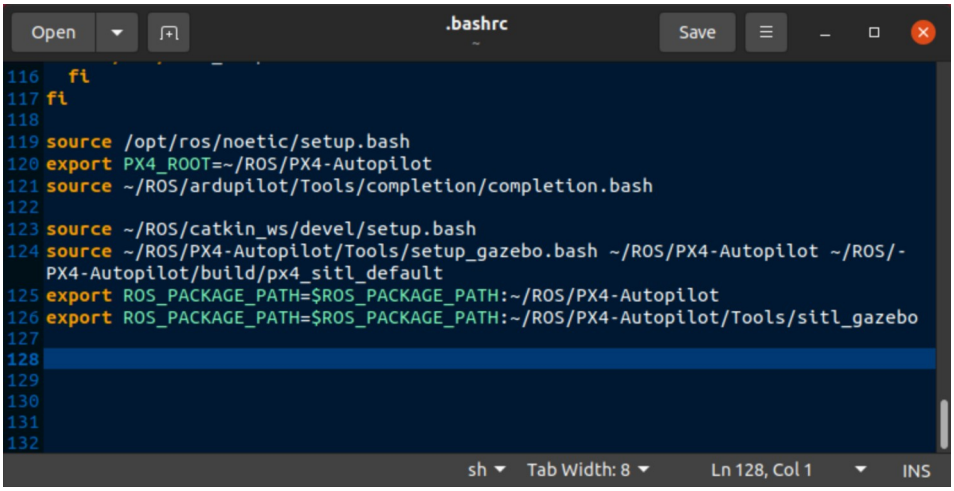
Рішення: відредагувати файл “*requirements.txt*” згідно ілюстрації нижче. Змінити “*matplotlib>=3.0.**” на “*matplotlib>=3.0.0*”



- При запуску програми *ROS* командою: *roslaunch simple_ros_app ros_mission.launch* виникає помилка , що ресурс *PX4* не знайдено.



Можливе рішення: перевірити налаштування змінних середовища у файлі “*.bashrc*” (*послідовність записів може впливати на виникнення помилки*). Приклад робочих налаштувань, які використовувались при тестуванні наведено нижче на ілюстрації. *Додатково можна замінити спеціальне посилання на домашній каталог користувача “~” на абсолютний шлях від кореня.



Лабораторна робота 2. Створення віртуального середовища для керування дроном у симуляторі

Мета роботи: ознайомитись із програмним забезпеченням для запуску та управління моделями дронів у віртуальному середовищі засобами *ROS2*.

Завдання:

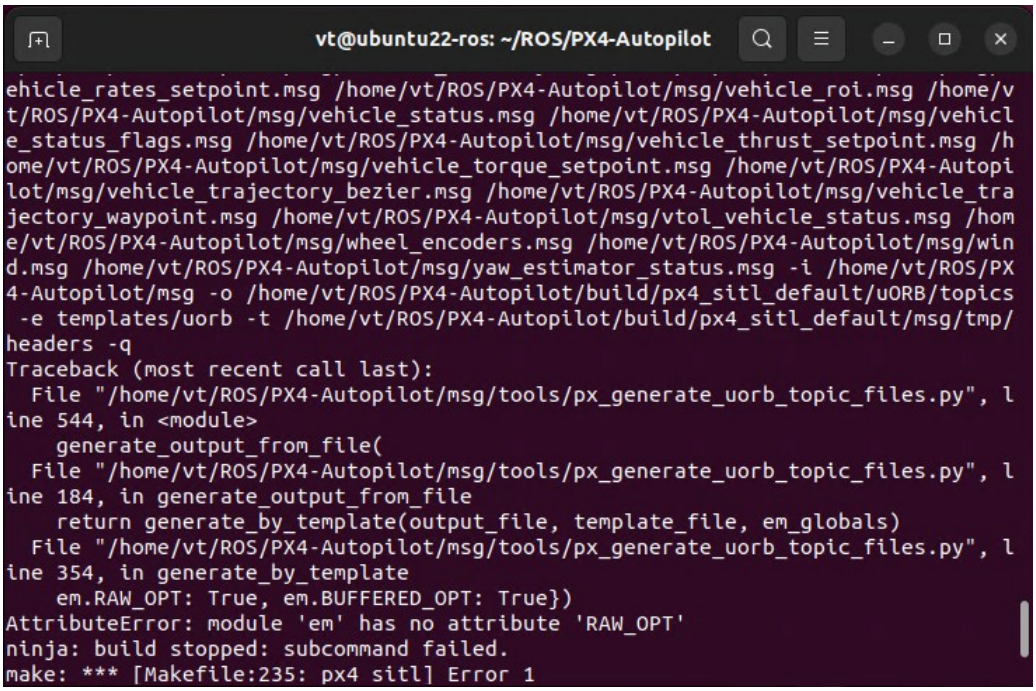
1. підготувати програмне забезпечення для симуляторів автопілотів на базі *Ardupilot* та *PX4*, а також програму управління *QGroundControl;*
2. підготувати віртуальне середовище для польотів у симуляторі *Gazebo*;
3. ознайомитись із основними поняттями та функціями *ROS2*;
4. розробити просту програму для керування моделлю дрону засобами *ROS2*.

Порядок виконання

- Детальний огляд кроків налаштування для завдань 1-2 представлено у лабораторній №1. Додаткові кроки по підготовці в контексті *ROS2* наведено у розділі “Середовище *ROS2*”;
- Завдання 3 — у розділі “Середовище *ROS2*”;
- Базовий код для завдання 4 надано у файлі “*src/simple_ros2_app.zip*” на онлайн ресурсі дисципліни.

Вирішення можливих помилок в ході налаштувань

- Під час компіляції *PX4* командою: “*make px4_sitl none_iris”* виникає помилка.



Рішення: перевстановити пакет “*empy*” командами:

1. *pip3 uninstall empy*
2. *sudo apt install python3-empy*

- Виконання команди з п. 3.3.1 “*python3 jinja_gen.py --mavlink_id=3 --mavlink_udp_port=14560 --mavlink_tcp_port=4560 --gst_udp_port=5600 --video_uri=5600 --mavlink_cam_udp_port=14530 ../models/iris/iris.sdf.jinja .. --output-file=~/ROS/iris0.sdf*” призводить до помилки: “*FileNotFoundError: [Errno 2] No such file or directory: '~/ROS/iris0.sdf*”

Рішення: замість “~” задати абсолютний шлях для параметру “*output-file”*

Лабораторна робота 3. Програмне керування автопілотом дрону засобами ROS2

Мета роботи: ознайомитись із керуванням автопілотом дрону програмним шляхом, розробити програмні компоненти *ROS2* для передачі політних команд моделям дронів.

Завдання:

1. доповнити віртуальне середовище налаштоване у лабораторній роботі 2 додатковими екземплярами дронів;
2. ознайомитись із поняттями та функціями автопілоту *PX4*, *MAVLink*, *MAVROS*, та *ROS2* для програмного керування польотом дрону;
3. розробити власні програмі компоненти *ROS2* для керування трьома моделями дронів.

Порядок виконання

- Необхідна теоретична інформація та детальний огляд кроків налаштування для завдань 1-3 наведено у розділі “Керування декількома екземплярами дронів із використанням *ROS2*”;
- Окремі приклади компонентів/ресурсів в контексті завдання 3 надано у директорії “*src/*” на онлайн ресурсі дисципліни.

Корисна інформація

- Для вимкнення захисного “*Failsafe*” режиму при відсутності зв’язку з пультом керування/політною програмою, що перешкоджає запуску двигунів коптеру, необхідно в автопілоті кожного екземпляру дрону встановити відповідне значення параметру: *COM_RCL_EXCEPT* = 7
- Опис формату повідомлення *MAVROS* для програмного керування: https://wiki.ros.org/mavros#mavros.2FPlugins.setpoint_position

- Додаткова інформація про політний режим Offboard: https://docs.px4.io/main/en/flight_modes/offboard.html

Лабораторна робота 4. Ройове керування та комунікація засобами ROS2

Мета роботи: ознайомитись із способом організації декількох моделей дронів у рій — групу для спільного керування, розробити програмні компоненти *ROS2* для передачі політних команд моделям дронів, які входять у спільну групу, доповнити модель дронів додатковими компонентами.

Завдання:

1. розглянути спосіб та сценарії групової передачі команд між декількома екземплярами дронів засобами *ROS2*;
2. доповнити програмний проєкт із лабораторної роботи 3 додатковими компонентами, що відповідатимуть за ройову — групову комунікацію дронів між собою;
3. реалізувати синхронізацію політної команди у групі дронів на базі автопілоту *PX4* або *Ardupilot*;
4. доповнити модель дрону віртуальною камерою;
5. розробити програмні компоненти *ROS2* для обробки відео потоку з камери дрону.

Порядок виконання

- Необхідна теоретична інформація та детальний огляд кроків налаштування для завдань 1-3 наведено у розділі “Ройове керування дронами із використанням *ROS2*”;
- Теоретична інформація та детальний огляд кроків налаштування для завдань 4-5 наведено у розділі “Обробка відео потоку камери віртуального дрону із симулятора *Gazebo*”. Ви можете реалізувати власний метод

обробки відео потоку з камери, відмінний від наведеного у методичних рекомендаціях.

- Окремі приклади компонентів/ресурсів в контексті завдань 2-5 надано у директорії “*src/*” на онлайн ресурсі дисципліни.

Корисна інформація

- Опис алгоритму *Lucas-Kanade* для обчислення оптичного потоку – *optical flow* у бібліотеці *OpenCV*: https://docs.opencv.org/4.5.4/d4/dee/tutorial_optical_flow.html

Лабораторна робота 5. Керування дроном засобами штучного інтелекту

Мета роботи: ознайомитись із способами керування моделями дронів шляхом передачі політних команд у форматі напрямів руху, розробити програмні компоненти *ROS2* для детекції об’єктів на зображеннях із бортової камери і формування команд руху для дрону.

Завдання:

1. розглянути додатковий тип команд керування дронами за рахунок передачі сигналів напрямів руху без використання системи координат;
2. доповнити програмний проєкт із лабораторної роботи 4 додатковими компонентами, що відповідатимуть за керування коптером шляхом передачі команд на пересування за напрямами руху;
3. реалізувати модуль комп’ютерного зору для коптеру на базі автопілоту *PX4* або *Ardupilot*, який оброблятиме зображення із бортової камери, виконуватиме детекцію об’єкту та формуватиме команди для пересування дрону, щоб утримувати знайдений об’єкт по середині поля зору бортової камери.

Порядок виконання

- Необхідна теоретична інформація та детальний огляд методів керування і комп'ютерного зору для завдань 1-3 наведено у розділі "Застосування методів комп'ютерного зору для керування дроном". Ви можете використовувати власний підхід у керуванні та реалізувати метод обробки зображення засобами комп'ютерного зору, що відмінний від наведених у методичних рекомендаціях;
- В контексті завдання 3, мінімально необхідним є реалізація керування дроном лише за однією віссю руху, наприклад, повороти вліво/вправо, щоб утримувати об'єкт в центрі поля зору камери. Реалізація керування дроном за всіма вісями руху оцінюватиметься на більшу оцінку.

Корисна інформація

- Доступні режими польоту в автопілоті *Ardupilot:* https://ardupilot.org/copter/docs/flight-modes.html
- Доступні режими польоту в автопілоті *PX4:* https://docs.px4.io/v1.13/en/flight_modes/
- Формат повідомлення "*manual_control/send*" у *MAVROS:* https://wiki.ros.org/mavros#mavros.2FPlugins.manual_control
- Нейронна мережа *YOLO:* https://github.com/ultralytics/ultralytics
- Алгоритми трекінгу у *OpenCV:* https://learnopencv.com/object-tracking-using-opencv-cpp-python/
- Додаткові віртуальні середовища для *Gazebo*: https://github.com/leonhartyao/gazebo_models_worlds_collection

СПИСОК ЛІТЕРАТУРИ

1. Francisco Martín Rico. A Concise Introduction to Robot Programming with ROS2. – CRC Press Chapman & Hall, 2022.
2. ROS2 Humble. URL: https://docs.ros.org/en/humble/index.html
3. ROS2 DDS. URL: https://docs.ros.org/en/humble/Installation/RMW-Implementations.html
4. Протокол MAVLink. URL: https://mavlink.io/en/
5. Робота з MAVLink у автопілоті Ardupilot. URL: https://ardupilot.org/dev/docs/mavlink-basics.html
6. Розширення MAVROS. URL: https://wiki.ros.org/mavros
7. Ardupilot SITL. URL: https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html
8. PX 4 Autopilot. URL: https://docs.px4.io/main/en/
9. QGC Drone Control. URL: https://qgroundcontrol.com/
10. Gazebo Simulator. URL: https://gazebosim.org/home
11. Нейронна мережа YOLO. URL: https://github.com/ultralytics/ultralytics