НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

До захисту допущено:
Завідувач кафедри
_____ Сергій СТІРЕНКО
“__” _____ 2024 р.

Дипломний проект
на здобуття ступеня бакалавра
за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

на тему: «Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS»

Виконав: студент 4 курсу, групи ІП-05 (шифр групи) Османов Ервін Якубович (прізвище, ім’я, по батькові) _____ (підпис)

Керівник
ст. викладач, доктор філософії, Таран Владислав Ігорович (посада, науковий ступінь, вчене звання, прізвище та ініціали) _____ (підпис)

Консультант(нормоконтроль)
доцент, к.т.н. Волокита Артем Миколайович (посада, науковий ступінь, вчене звання, прізвище та ініціали) _____ (підпис)

Рецензент
доц. каф. ІСТ, к.т.н., Шимкович Володимир Миколайович (посада, науковий ступінь, вчене звання, прізвище та ініціали) _____ (підпис)

Засвідчую, що у цьому дипломному проєкті
немає запозичень з праць інших авторів
без відповідних посилань.
Студент _____
(підпис)

Київ – 2024

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)
Освітньо-професійна програма
“Інженерія програмного забезпечення комп’ютерних систем”
спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Сергій СТІРЕНКО
(підпис)

« » _____ 2024 р.

ЗАВДАННЯ

на бакалаврський дипломний проект студента

Османова Ервіна Якубовича

1. Тема проекту: «Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS»
   Керівник проекту: Таран Владислав Ігорович, ст. викладач, доктор філософії
2. Термін подання здобувачем вищої освіти роботи: 13.06.2024
3. Вихідні дані до роботи: *попередні дослідження використання комп'ютерного зору для автономних роботизованих систем, технічна документація, теоретичні та статистичні дані, науково-технічна література*
4. Зміст роботи: Опис предметної області, дослідження методів розробки модулю комп’ютерного зору для роботизованих систем із використанням ROS

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) компоненти системи (структурна схема), діаграма структури даних (функціональна схема), алгоритм дій (принципова схема).
6. Консультанта проекту, з вказівкою розділів проекту, які до них вносяться

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Нормоконтроль	Волокита А. М.		

7. Дата видачі завдання: «16» Лютого 2024 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2023-15.12.2023*	
2.	*Вивчення та аналіз завдання*	*15.12.2023-15.03.2024*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2024-25.03.2024*	
4.	*Розробка структур окремих підсистем*	*25.03.2024-5.04.2024*	
5.	*Програмна реалізація системи*	*5.04.2024- 15.04.2024*	
6.	*Оформлення пояснювальної записки*	*15.04.2024- 20.05.2024*	
7.	*Захист програмного продукту*	*6.06.2024*	
8.	*Передзахист*	*06.06.2024*	
9.	*Захист*	*20.06.2024*	

Студент-дипломник _____ Ервін ОСМАНОВ

(підпис)

Керівник проекту _____ Владислав ТАРАН

(підпис)

АНОТАЦІЯ

Дипломна робота присвячена розробці модулю комп'ютерного зору для автономних роботизованих систем на базі Robot Operating System (ROS). Основна увага зосереджена на аналізі, проектуванні та реалізації алгоритмів комп'ютерного зору, що дозволяють роботизованим системам визначати об’єкти в навколишньому середовищи. Проект включає інтеграцію з відкритими бібліотеками та інструментами для обробки зображень, що значно покращує функціональні можливості роботизованих систем. Результати роботи демонструють підвищення точності та швидкості обробки візуальної інформації, а також здатність системи адаптуватися до нових завдань.

Ключові слова: комп'ютерний зір, автономні роботизовані системи, ROS, алгоритми обробки зображень, інтеграція систем, візуальна локалізація.

ABSTRACT

The thesis is devoted to the development of a computer vision module for autonomous robotic systems based on the Robot Operating System (ROS). The main focus is on the analysis, design and implementation of computer vision algorithms that allow robotic systems to detect objects in environment. The project includes integration with open libraries and image processing tools, which significantly improves the functionality of robotic systems. The results of the work demonstrate an increase in the accuracy and speed of visual information processing, as well as the ability of the system to adapt to new tasks.

Keywords: computer vision, autonomous robotic systems, ROS, image processing algorithms, system integration, visual localization

справки	Форм	*Значення*	*Найменування*	Кіл. листів	№ екземпл	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Модуль комп'ютерного	3		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Модуль комп'ютерного	58		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Модуль комп'ютерного	1		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			(структурна схема)			
	А4	***ІАЛЦ.467200.005 Д2***	Модуль комп'ютерного	1		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			(функціональна схема)			
	А4	***ІАЛЦ.467200.006 Д3***	Модуль комп'ютерного	1		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			Алгоритм дій програмного			
			забезпечення			
			(принципова схема)			
	А4	***ІАЛЦ.467200.007 Д4***	Модуль комп'ютерного	20		
			зору для автономних			
			роботизованих систем			
			із використанням ROS			
			Текст програми			

					ІАЛЦ.467200.001 ОА
Зм.	Арк.	№ докум.	Підпис	Дата	
Розробив		Османов Е. Я.			*Модуль комп'ютерного зору для роботизованих систем із використанням ROS* Опис Альбому
Перевірив		Таран В.І			
Реценз.					
Н. Контр.		Волокита А. М.			
Затвердив					

Літ.	Аркуш	Аркушів
	1	

КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05

Технічне завдання
до дипломного проекту

на тему: “Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS”

Київ – 2024

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ			
Зм.	Арк.	№ докум.	Підпис	Дата				
Розробив		Османов Е. Я.			*Модуль комп'ютерного зору для роботизованих систем із використанням ROS* Технічне завдання	Літ.	Аркуш	Аркушів
Перевірив		Таран В.І					1	3
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05		
Н. Контр.		Волокита А. М.						
Затвердив								

1. НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ВИКОРИСТАННЯ

Найменування проекту: Розробка модуля комп'ютерного зору для автономних роботизованих систем з використанням ROS. Область використання: Автономні роботизовані системи, що здійснюють взаємодію з навколишнім середовищем в промислових, дослідницьких або побутових сферах.

2. ПРИЧИНИ ДЛЯ РОЗРОБКИ

Підставою для розробки служить завдання для виконання роботи кваліфікаційно-освітнього рівня «бакалавр інженерії програмного забезпечення», який був затверджений факультетом “Інформатики та обчислювальної техніки” кафедрою обчислювальної техніки Національного технічного Університету України «Київський Політехнічний інститут ім. Ігоря Сікорського».

3. МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою є створення модуля комп'ютерного зору для автономних роботів, що інтегрується з ROS. Призначенням є забезпечення автономних роботів здатністю до взаємодії з різноманітними об'єктами у навколишньому середовищі.

4. ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки є науково-технічна література з комп'ютерного зору, обробки зображень, дослідження та стандарти у галузі робототехніки, технічна документація та інструкції по ROS (Robot Operating System) та статті в мережі Інтернет на дану тему.

5. ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

- Інтеграція з існуючими роботизованими системами.
- Модульність та масштабованість.

• Висока надійність і точність у роботі.

5.2. Вимоги до програмного забезпечення

• Використання бібліотеки OpenCV для обробки зображень.

• Сумісність з останніми версіями ROS.

• Можливість легкої інтеграції з іншими модулями ROS.

5.3. Вимоги до апаратної частини

• Сумісність із стандартними промисловими компонентами.

• Використання камер високої роздільної здатності.

• Підтримка роботи в різноманітних умовах освітлення.

• Мінімум 4GB RAM, 2G Graphics

6. ЕТАПИ РОЗРОБКИ

6.1 Вивчення літератури	06.12.2023
6.2 Складання і узгодження технічного завдання	22.12.2023
6.3 Проектування алгоритмів комп'ютерного зору	15.04.2024
6.4 Інтеграція алгоритмів з ROS	18.05.2024
6.5 Тестування на роботизованих платформах	21.05.2024
6.6 Аналіз результатів і оптимізація системи	25.05.2024
6.7 Підготовка документації та публічний захист роботи	05.06.2024

Пояснювальна записка
до дипломного проекту

на тему: “Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS”

Київ – 2024 р.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

CSRT (Channel and Spatial Reliability Tracker) — це один з алгоритмів трекінгу об'єктів, реалізованих у бібліотеці OpenCV. Цей алгоритм відноситься до класу алгоритмів трекінгу, які використовують кореляційні фільтри.

ОС (Операційна система) — це базовий комплекс програм, що виконує керування апаратною складовою комп'ютера або віртуальної машини; забезпечує керування обчислювальним процесом і організовує взаємодію з користувачем.

RnD (Research and Development) - це структура, яка шукає, фільтрує та аналізує ініціативи в індустрії та всередині компанії.

RTOS — це проста, легка система, яка використовується для обмежених або простих систем, таких як вбудовані пристрої.

DDS (Data Distribution Service) - це стандартний протокол та API для зв’язку, що забезпечує масштабовану, надійну та продуктивну передачу даних між публікаторами та підписниками.

TCP (Transmission Control Protocol) - один із мережевих протоколів, орієнтований на роботу з підключеннями та передачею даних у вигляді потоків байтів

UDP — це один з протоколів транспортного рівня моделі OSI, котрий виконує обмін повідомленнями

RVIZ - інструмент з відкритим вихідним кодом, призначений для візуалізації процесів і налагодження алгоритмів робототехнічної систем.

ВСТУП

У сучасному світі технології стрімко розвиваються, і цей розвиток постійно змінює багато аспектів нашого життя, включаючи такі галузі, як промисловість, логістика і особливо робототехніка. Колись це було лише фантастикою, а зараз автономні роботизовані системи активно інтегруються в наше повсякденне життя, від виробничих ліній до домашніх помічників. Головним елементом ефективності таких систем є комп'ютерний зір. Це технологія, яка дозволяє машинам «бачити» своє оточення і реагувати на нього відповідним чином.

Розвиток і вдосконалення комп'ютерного зору відкриває нові можливості для автономних роботів, зокрема, покращуючи їхню здатність рухатися на основі візуальної інформації, взаємодіяти з об'єктами і навіть самостійно навчатися. Використання Robot Operating System (ROS) як гнучкої та масштабованої платформи для розробки робототехнічних систем надає унікальну можливість інтегрувати останні досягнення в галузі комп'ютерного зору.

Актуальність підтверджується не тільки стрімким розвитком технологій, але й зростаючим попитом промисловості на інтелектуальні автономні системи, здатні ефективно функціонувати в реальному світі без прямого контролю з боку людини. Це особливо важливо при роботі в небезпечних ситуаціях або у важкодоступних місцях, де використання автономних роботів є найбільш логічним рішенням.

Метою цієї дипломної роботи є дослідження, розробка та інтеграція модуля комп'ютерного зору для автономних робототехнічних систем з використанням ROS. Цей модуль матиме змогу виявляти об'єкти в 3D просторы. Модуль буде інтегрований з ROS, що дозволить обмінюватися даними з іншими вузлами системи.

Основним натхненням для цього дослідження стало бажання вивчити роботизовані системи та можливість впровадження комп’ютерного зору в них,

так як дані можливості є дуже актуальними в рамках поточних бойових дій в Україні.

На тлі глобальних технологічних змін це дослідження має на меті перетворити теоретичні знання у практичні технологічні рішення.

РОЗДІЛ 1. ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

1.1 ВИЗНАЧЕННЯ ПОНЯТЬ

Автономні роботизовані системи - це системи, здатні виконувати завдання в неструктурованому середовищі без постійного зовнішнього втручання. Їх автономність включає самостійне прийняття рішень на основі зібраних даних, адаптацію до змін у навколишньому середовищі та навчання на власному досвіді.

Операційна система роботів (ROS) - це гнучка платформа для написання програмного забезпечення для роботів, яка включає в себе набір інструментів і бібліотек спрямованих на спрощення створення складної і надійної поведінки роботизованих систем в різних середовищах. ROS надає сервіси, розробка яких в іншому випадку була б складною і трудомісткою, такі як апаратна абстракція, низькорівневе управління пристроями, реалізація загальновживаних функцій, передача повідомлень між процесами і управління пакетами.

Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS — це спеціалізований програмно-апаратний комплекс, який інтегрується у роботизовані системи для надання їм здатності сприймати та інтерпретувати візуальну інформацію з їхнього оточення. Використання ROS як платформи для розробки та впровадження цього модуля дозволяє використовувати численні існуючі інструменти та бібліотеки, забезпечуючи високий рівень модульності та адаптивності.

Одночасна локалізація та картографування (SLAM) - це набір алгоритмів, що використовуються в робототехніці для одночасного визначення власного положення робота в навколишньому середовищі та створення або оновлення карти цього середовища. SLAM відіграє ключову роль у навігації автономних роботів, особливо в незнайомому або динамічно мінливому середовищі.

Карти глибини - це візуальні дані, які показують відстань від датчика до точок сцени. Карти глибини часто використовуються в комп'ютерному зорі для підтримки різних завдань, таких як виявлення об'єктів, навігація та взаємодія з навколишнім середовищем.

Згорткові нейронні мережі (CNN) - це клас глибоких нейронних мереж, які особливо ефективні в аналізі візуальних зображень. CNN здатні витягувати і вивчати характеристики зображень на різних рівнях абстракції, що робить їх важливим інструментом в модулях комп'ютерного зору для розпізнавання об'єктів, класифікації зображень і виявлення об'єктів.

1.2 КОМП'ЮТЕРНИЙ ЗІР В РОБОТОТЕХНІЦІ

1.2.1 КОМП'ЮТЕРНИЙ ЗІР

Комп'ютерний зір - це галузь комп'ютерних наук, яка фокусується на відтворенні частин складної системи людського зору і дозволяє комп'ютерам ідентифікувати і обробляти об'єкти на зображеннях і відео так само, як це робить людина. Донедавна комп'ютерний зір працював лише в обмежених можливостях.

Завдяки досягненням у галузі штучного інтелекту та інноваціям у сфері глибокого навчання і нейронних мереж, за останні роки ця сфера зробила великий стрибок і змогла перевершити людину в деяких завданнях, пов'язаних з виявленням і маркуванням об'єктів.

Одним із рушійних факторів розвитку комп'ютерного зору є велика кількість даних, які ми генеруємо сьогодні, а потім використовуємо для навчання та вдосконалення комп'ютерного зору.

На певному рівні комп'ютерний зір - це розпізнавання образів. Отже, один із способів навчити комп'ютер розуміти візуальні дані - це надати йому зображення, багато зображень, тисячі, мільйони, якщо можливо, які були позначені, а потім застосувати до них різні програмні методи або алгоритми, які

дозволять комп'ютеру знаходити закономірності у всіх елементах, які пов'язані з цими позначеннями.

Так, наприклад, якщо ви завантажите в комп'ютер мільйон зображень котів, він опрацює їх всіх алгоритмами, які дозволять йому проаналізувати кольори на фото, форми, відстані між формами, місця, де об'єкти межують один з одним, і так далі, щоб визначити профіль того, що означає «кіт». Після завершення роботи комп'ютер (теоретично) зможе використати свій досвід, якщо йому завантажити інші немарковані зображення, щоб знайти ті, на яких зображений кіт.

1.2.2 РОБОЧІ СТАНЦІЇ З КОМП’ЮТЕРНИМ ЗОРОМ

Робочі станції, оснащені системами комп'ютерного зору, є невід'ємною частиною сучасного промислового обладнання. Вони забезпечують високий рівень автоматизації завдяки здатності аналізувати візуальну інформацію, що дозволяє значно підвищити точність і ефективність виробничих процесів. Основні сфери застосування робочих станцій комп'ютерного зору включають:

Контроль якості: Завдяки вбудованим камерам і алгоритмам аналізу зображень ці системи ефективно виявляють виробничі дефекти, такі як неправильні розміри або пошкодження поверхні. Це зменшує потребу в ручному контролі і підвищує загальну надійність продукції.

Автоматизація збірки: Сучасні робочі станції здатні керувати роботизованими системами для правильного складання складних механізмів, що особливо цінно в автомобільній та електронній промисловості.

Логістика та сортування: Швидка класифікація і правильне сортування різних предметів - використання комп’ютерного зору революціонізувало стратегії на складах, забезпечивши ефективний контроль над потоками товарів.

Навігація: Автономні роботи використовують комп'ютерний зір для безпечної навігації на складах і виробничих ділянках, уникаючи перешкод і оптимізуючи маршрути.

Адаптивність та інтерактивність: Здатність систем адаптуватися до змін у виробничому середовищі та взаємодіяти з людьми-операторами розширює сферу їх застосування та підвищує ефективність.

Завдяки постійному вдосконаленню алгоритмів машинного навчання та якості оптичних компонентів, робочі станції комп'ютерного зору відіграють ключову роль у промисловому виробництві, стимулюючи інновації та підвищуючи загальну продуктивність.

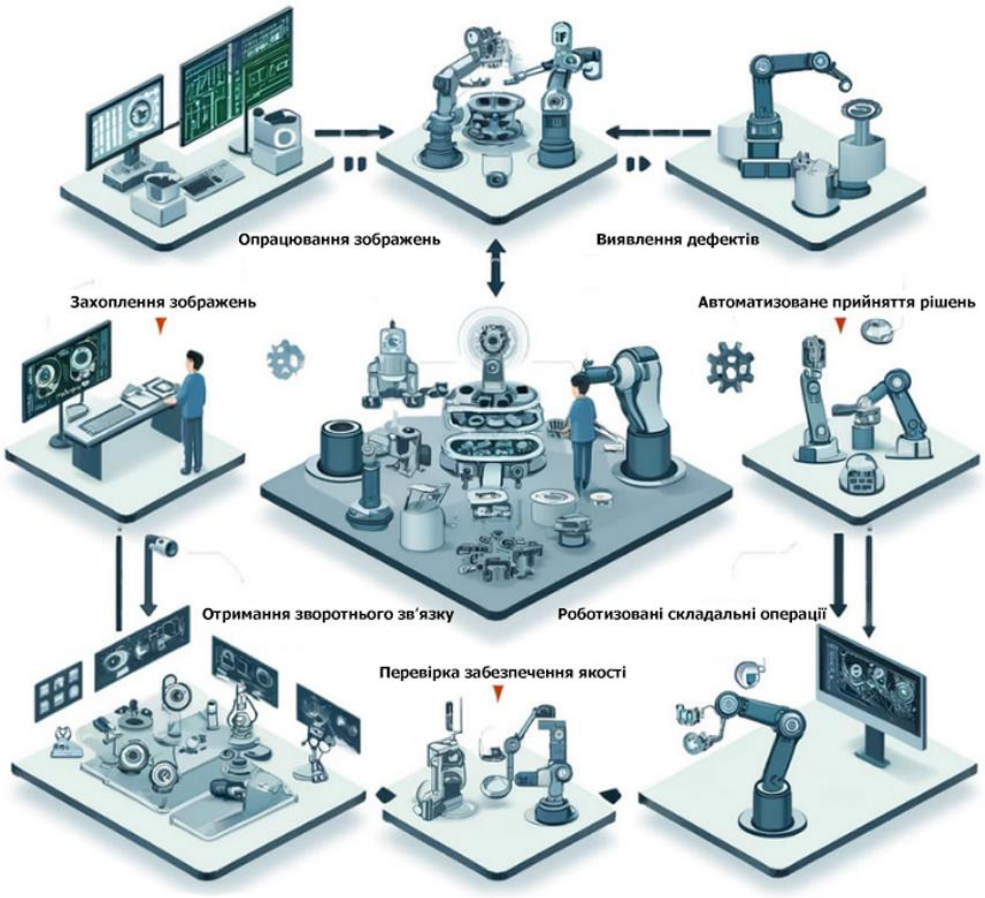


Рисунок 1.1 - Схема технологічного процесу роботизованої робочої станції з комп'ютерним зором

1.2.3 НАВІГАЦІЯ З ВИКОРИСТАННЯМ КОМП’ЮТЕРНОГО ЗОРУ

Навігація з використанням комп'ютерного зору є ключовим компонентом у багатьох сучасних технологічних системах, зокрема у робототехніці та автономних транспортних засобах. Цей метод включає в себе використання камер та інших оптичних сенсорів для збору візуальних даних з навколишнього середовища, які потім обробляються за допомогою алгоритмів комп'ютерного зору для визначення положення, орієнтації та навігації пристрою.

Використання комп'ютерного зору для навігації дозволяє автомобілям та дронам самостійно переміщатися по дорогах або в повітрі, розпізнавати дорожні знаки, перешкоди, інших учасників руху і безпечно доїжджати до пункту призначення.

Компактні доставкові роботи, які переміщаються по тротуарах, використовуючи камери для навігації та обходу перешкод, вже стають частиною міського ландшафту в деяких містах.

В умовах, де людське втручання небезпечне або неможливе, роботи, оснащені системами комп'ютерного зору, можуть проводити пошуково-рятувальні місії, визначаючи місцезнаходження людей у завалах або на великій воді.

Компанія Amazon використовує роботів, оснащених системами комп'ютерного зору, для автоматизації складських операцій. Роботи здатні самостійно переміщатися по складу, уникаючи зіткнень з іншими роботами та об'єктами. (Рис 1.2) [1]

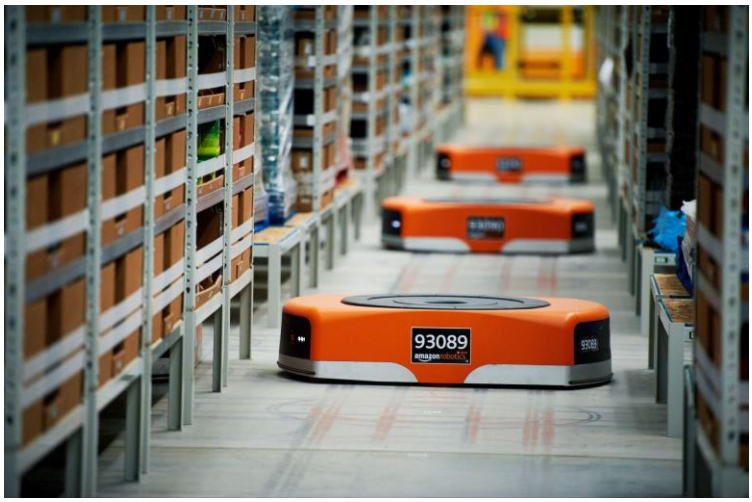


Рисунок 1.2 - Amazon Robotics працюють в цеху

Система автопілота в автомобілях Tesla - Tesla Autopilot використовує декілька камер та датчиків для забезпечення можливості автономного водіння, здатного до реального часу розпізнавати різні дорожні умови та адаптуватися до них.

Також Tesla Autopilot класифікується як система допомоги водієві 2-го рівня (з 5-ти можливих), що означає часткову автоматизацію. Водій повинен постійно контролювати ситуацію та бути готовим негайно взяти управління на себе.

Ще один цікавий приклад - Робот-собака Boston Dynamics' Spot може навігувати в складних промислових та дослідницьких середовищах, використовуючи комп'ютерний зір для мапування території і обходу перешкод.

Також із останніх актуальних новин - Boston Dynamics' Spot будуть використовуватись для розмінування територій України.

1.2.4 ВЗАЄМОДІЯ З ОБ’ЄКТАМИ ЧЕРЕЗ КОМП’ЮТЕРНИЙ ЗІР

Щоб мати змогу автоматизовано отримувати, обробляти та аналізувати візуальну інформацію з реального світу та знаходити корисні дані або приймати рішення - використовують комп’ютерний зір. Це відбувається шляхом вивчення

видимих фактів з камер або різних візуальних датчиків, після чого ця інформація обробляється для виконання певних завдань, включаючи вибір, переміщення, сортування або збирання об'єктів. Комп'ютерний зір дає змогу машинам інтерпретувати і розпізнавати видимі об’єкти з навколишнього середовища, використовуючи найсучасніші алгоритми і передові технології обробки фотографій.

Поява комп'ютерного зору збігається з розвитком штучного інтелекту наприкінці 60-х років. Однак світ почав проявляти справжній інтерес до комп’ютерного зору лише в 90-х роках, який ще більше зріс з появою перших промислових застосувань і впровадженням глибинного навчання.

Проблеми комп'ютерного зору включають автономне водіння або аналіз медичних зображень для покращення процесів діагностики. З вирішенням цих питань виникло багато різних завдань: класифікація зображень, що полягає у віднесенні даного зображення до однієї із заздалегідь визначених категорій, пошук зображень на основі контенту, який ідентифікує зображення з певним змістом у великому наборі даних, розпізнавання об'єктів на зображенні.

У виявленні об'єктів нашою метою є знаходження екземплярів потрібних об'єктів на зображеннях. Це можна розглядати як класифікацію та регресію. Простіше кажучи, ми навчаємо модель малювати рамки навколо потрібних об'єктів на зображенні. (Рис 1.3.)[2]

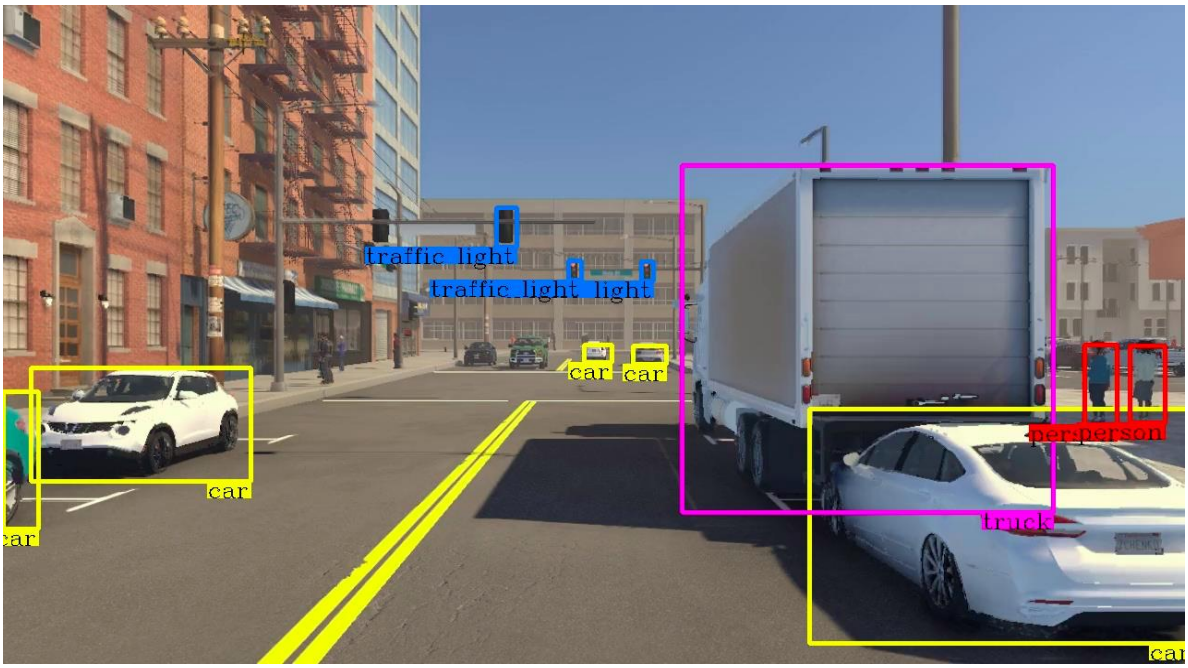


Рисунок 1.3 - Приклад розпізнавання об’єктів на зображенні

Ось приклади того, як компанії використовують комп'ютерний зір у реальних додатках у різних галузях:

Walmart використовує роботів-сканерів, оснащених камерами комп'ютерного зору, для безперервного моніторингу полиць магазинів. Роботи попереджають персонал, коли товари закінчуються або знаходяться в неправильному місці, допомагаючи забезпечити оптимальний рівень запасів.

BMW використовує комп'ютерний зір для контролю якості в процесі виробництва. Камери, оснащені датчиками з високою роздільною здатністю і моделями глибокого навчання, перевіряють деталі автомобіля на наявність дефектів з точністю понад 90%, що значно перевищує точність перевірки людиною.

Програмне забезпечення **InnerEye** від Microsoft*(корпорація комп'ютерних технологій)* використовує моделі комп'ютерного зору з глибоким навчанням для автоматичного аналізу 3D-радіологічних зображень, таких як КТ*(Комп'ютерна томографія)* і МРТ*(Магнітно-резонансна томографія)*. Моделі можуть виявляти, сегментувати та кількісно оцінювати анатомію та патології, допомагаючи радіологам ставити швидші та точніші діагнози.

1.2.5 ОПИС СЕРЕДОВИЩА ROS

Robot Operating System (ROS) є гнучкою платформою, призначеною для полегшення розробки програмного забезпечення для роботизованих систем. Це включає бібліотеки, які спрямовані на створення складної та масштабованої поведінки роботизованих систем у різноманітних середовищах. Важливими аспектами ROS є підтримка спільної розробки, спрощення процесу кодування та тестування, а також взаємодія між різними елементами системи.

До появи операційних систем для роботів кожен конструктор роботів і дослідник робототехніки витрачав значну кількість часу на розробку вбудованого програмного забезпечення для роботів, а також самого обладнання. Це вимагало навичок у машинобудуванні, електроніці та вбудованому програмуванні. Як правило, програми, розроблені таким чином, були більше схожі на вбудоване програмування, подібне до електроніки, ніж на робототехніку в строгому сенсі, як ми можемо зустріти її сьогодні в сервісній робототехніці. Існувало значне повторне використання програм, оскільки вони були тісно пов'язані з основним обладнанням.

Основна ідея ОС для робототехніки полягає в тому, щоб уникнути постійного винаходу колеса і запропонувати стандартизовані функції, що виконують апаратну абстракцію, так само, як і звичайна ОС для ПК, звідси і аналогічна назва.

ROS - це фасилітатор проектів з робототехніки. Дослідники або інженери в науково-дослідних підрозділах, які використовують ROS, більше не витрачають час на створення нової екосистеми для кожного нового проекту з робототехніки. Це також є фінансовою вигодою.

ROS позитивно впливає на R&D, зменшуючи витрати і «час виходу на ринок». Для структур або відділів, метою яких є швидкий запуск нового прототипу або скорочення технологічного розриву, це стає досить привабливим.

Ще однією перевагою операційних систем роботів, таких як ROS, є

поєднання досвіду з різних дисциплін. Фактично, проектування та програмування робота означає:

1. Керування апаратним забезпеченням шляхом написання драйверів
2. Керування пам'яттю та процесами
3. Керування паралельністю, паралелізмом та об'єднанням даних
4. Надання алгоритмів абстрактних міркувань, широко використовуючи штучний інтелект.

ROS знижує технічний рівень, необхідний для роботи над роботизованими системами. Це може полегшити багатьом компаніям початок роботи в галузі робототехніки або пришвидшити розробку складних систем.

ROS не залежить від мови. На даний момент для ROS визначено три основні бібліотеки, які дозволяють програмувати ROS на Python, Lisp або C++. На додаток до цих трьох бібліотек пропонуються дві експериментальні бібліотеки, що дозволяють програмувати ROS на Java або Lua.

Архітектура ROS базується на принципі розподілених обчислень і включає декілька основних компонентів:

Nodes (ноди) - Атомарні будівельні блоки в ROS називаються вузлами. Вузли відповідають за роботу з пристроями та датчиками, управління або обчислювальні алгоритми, в ідеалі кожен вузол виконує окрему задачу. Ці завдання можуть бути як низькорівневими, так і високорівневими. Вузли можуть спілкуватися один з одним за допомогою тем або сервісів. Існує два різних типи вузлів - видавці та підписники. Як випливає з їхніх назв, в той час як видавці публікують повідомлення в темах, підписники підписуються на тему, щоб отримувати повідомлення звідти. Структура ROS базується на розподілених програмних пакетах.

Topics (топіки) - це поняття для опису потоку даних, що використовується для обміну інформацією між різними вузлами. Теми використовуються для надсилання потоку повідомлень одного типу. Це може бути інформація з датчика або зображення з камери. Вузли публікують повідомлення в темах або

підписуються на них, щоб отримувати повідомлення. Кількість різних вузлів, що публікують або підписуються на тему, обмежена системними ресурсами, зокрема оперативною пам'яттю. Кожна тема має унікальне ім'я з визначеним типом повідомлень. Взаємозв'язком між вузлами і темами керує майстер ROS в ROS1 і DDS в ROS2, який є своєрідним проміжним програмним забезпеченням для управління в ROS.

Services (Сервіси) - Альтернативним методом зв'язку між вузлами є сервіси, які нагадують модель архітектури сервер-клієнт, форму віддаленого виклику процедур (RPC). Клієнт надсилає запит, а потім сервер надсилає відповідь на цей запит. Тут і клієнт, і сервер фактично є вузлами ROS. Таким чином, це двонаправлений зв'язок, однак, на відміну від тем, сервіси є одноразовим зв'язком. Сервіси зазвичай використовуються для виклику дії.

Actions (Дії) - Для того, щоб викликати сервіси, але для задач, що потребують тривалого часу, у нас є концепція, яка називається діями. Дія ROS може бути визначена трьома основними повідомленнями ROS, а саме: мета, результат і зворотній зв'язок. В той час як мета представляє стан, на який спрямована дія, а повідомлення про результат представляє фактичний результат після виконання дії. А повідомлення зворотного зв'язку - це повідомлення, яке періодично надсилається для відстеження прогресу виконання завдання.

Parameters (Параметри) - У ROS середовище робота та змінні часу виконання зберігаються на сервері параметрів ROS. Параметри складаються з пар ключ-значення, як словникові структури даних. Оскільки вони не розраховані на високу продуктивність, їх краще використовувати для параметрів конфігурації, таких як статичні, недвійкові дані.

ROS Package (Пакет ROS) - Окремий пакет зазвичай розробляється для виконання одного конкретного завдання, наприклад, керування, навігації тощо, і може містити один або декілька вузлів, переважно повідомлень.

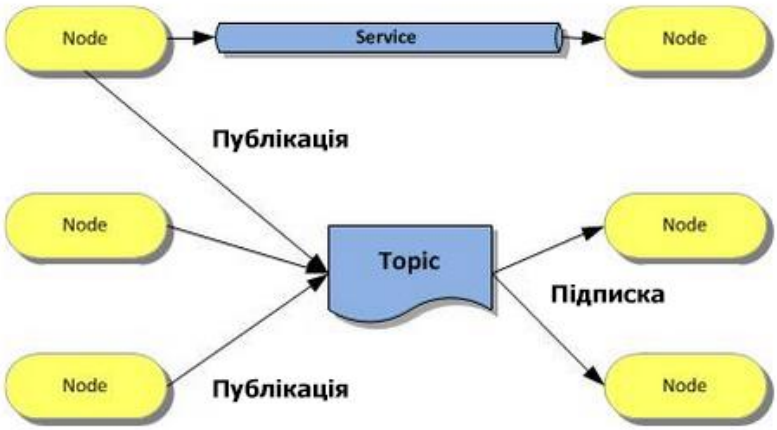


Рисунок 1.4 - Схематичне представлення обчислень системи ROS

ROS та ROS2. Останніми роками спільнота OSRF інтенсивно працювала над вдосконаленням ROS. Сьогодні ці дослідження почали приносити плоди, і нове покоління ROS почало поширюватися під назвою ROS2. Основна відмінність ROS2 від свого попередника ROS полягає в тому, що базова архітектура зв'язку між видавцем і підписником системи взята з TCP/UDP багатоадресного ROS-майстра і повністю замінена на DDS (служба розподілу даних). DDS пропонує дійсно потужні можливості і підсилює ROS для проектування робототехнічних систем реального часу виробничого рівня. Крім того, ROS2 можна використовувати з більшою кількістю операційних систем, таких як Windows, Mac OS та RTOS.
(Рис 1.5)[3]

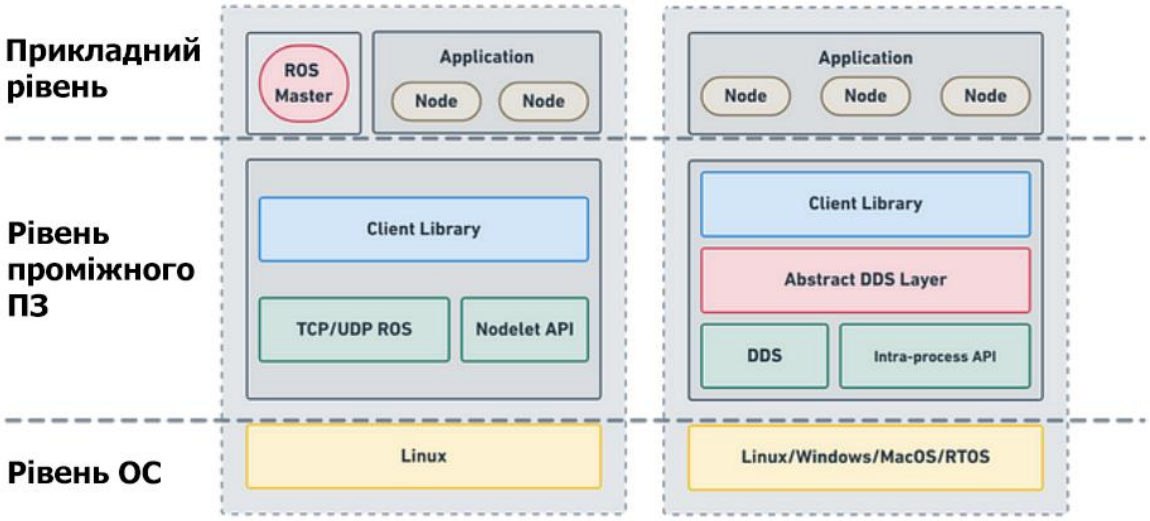


Рисунок 1.5 - Порівняльні схеми ROS та ROS2

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі розглянуто ключові аспекти та поточні рішення у сфері комп'ютерного зору для автономних роботизованих систем, верхньорівнево описав базові поняття для розуміння того, що таке комп’ютерний зір, навігація з його допомогою та взаємодія з об’єктами, а також більш детально заглибився в опис середовища ROS.

Комп'ютерний зір відіграє вирішальну роль у сучасній робототехніці, забезпечуючи роботам здатність "бачити" та інтерпретувати своє оточення. Це включає в себе не тільки визначення та класифікацію об'єктів, але й складніше розуміння просторових відносин та динамічних змін у навколишньому середовищі. Технології, такі як навігація за допомогою комп'ютерного зору та взаємодія з об'єктами, вже активно впроваджуються в промисловість, медицину, логістику та інші галузі.

ROS є ідеальним середовищем для розробки таких технологій завдяки своїй модульності, великій кількості доступних інструментів та активній спільноті. Його архітектура та інструменти сприяють швидкій інтеграції нових технологічних рішень та вдосконаленню існуючих систем. Використання ROS дозволяє розробникам зосередитись на інноваціях та функціональності, мінімізуючи необхідність вирішення стандартних задач інтеграції та комунікації.

Головним критерієм саме для мене було те, що використання даних технологій має обширний вплив на якість життя людей саме зараз. Прикладі багатьох техногігантів, які вже використовують дані інструменти для спрощення життя людей дають мені віру в те, що подальший розвиток цієї сфери буде мати колосальний результат і допоможе людству стати більш прогресивним.

РОЗДІЛ 2. АРХІТЕКТУРА СИСТЕМИ ТА АНАЛІЗ ТЕХНОЛОГІЙ ДЛЯ ЇЇ РОЗРОБКИ

2.1 ЗАГАЛЬНІ ДАНІ ДЛЯ РОЗУМІННЯ АРХІТЕКТУРИ СИСТЕМИ

2.1.1 ОПЕРАЦІЙНА СИСТЕМА UBUNTU І ЇЇ ПЕРЕВАГИ ДЛЯ ПРОЕКТУ

Ubuntu - це дистрибутив*(форма розповсюдження програмного забезпечення)* Linux, який походить від Debian і складається здебільшого з вільного програмного забезпечення з відкритим кодом. Він офіційно випускається в декількох редакціях, включаючи Desktop, Server і Core для пристроїв і роботизованих систем з виходом в інтернет. Розроблена британською компанією Canonical та спільнотою інших розробників,

Ubuntu відома своєю стабільністю, зручністю для користувачів та потужною підтримкою спільноти. Canonical надає оновлення безпеки та підтримку для кожного випуску Ubuntu, а версії з довгостроковою підтримкою (LTS) випускаються кожні два роки. Ubuntu широко використовується для загальних цілей, хмарних обчислень і популярна в публічних хмарах, центрах обробки даних і на периферії. Вважається корпоративним дистрибутивом Linux, що пропонує повну комерційну підтримку, виправлення безпеки та спеціальні функції через підписку Canonical Ubuntu Advantage for Infrastructure.

Оскільки Ubuntu є одним з найстабільніших дистрибутивів Linux, він спочатку був основним варіантом для розробників ROS, які шукали підтримку фреймворку. Подорож почалася у 2007 році, щоб спростити процес розробки складної поведінки на широкому спектрі роботизованих платформ. Він випускається під вільною ліцензією BSD. Сьогодні вона підтримується Open Robotics, але здебільшого розвивається спільнотою розробників. ROS була в

основному прийнята компаніями в галузі робототехніки та автоматизації, і вона продовжує зростати.

ROS була створена з урахуванням особливостей Ubuntu. Велика частина розробки і тестування ROS відбувається на Ubuntu, що робить її найбільш стабільною і добре підтримуваною платформою.

Ubuntu забезпечує міцну основу для ROS, з такими важливими інструментами, як GCC, CMake, Python та бібліотеками, як Boost та Eigen, які вже встановлені. Це значно спрощує налаштування середовища розробки ROS.

Також Ubuntu має велику та активну спільноту, що має вирішальне значення для такого складного програмного проекту, як ROS. Спільнота надає підтримку, документацію та пакунки, які добре інтегруються з Ubuntu

ROS слідує циклу випуску Ubuntu, з новим дистрибутивом, випущеним щороку, і версією LTS (Long-Term Support) кожні два роки. Така синхронізація дозволяє розробникам ROS покладатися на стабільні та добре перевірені версії Ubuntu.

Ще Ubuntu можна легко встановити на віртуальну машину за допомогою таких інструментів, як VirtualBox, що дозволяє розробникам налаштовувати середовище розробки ROS, не впливаючи на базову операційну систему.

Хоча ROS можна встановити на інші дистрибутиви Linux, Windows та macOS, Ubuntu залишається основною платформою завдяки тісній інтеграції з ROS та потужній підтримці спільноти. Оскільки індустрія робототехніки продовжує зростати, Ubuntu та ROS стають все більш важливими інструментами для розробників та дослідників у цій галузі.

2.1.2 МОВИ ПРОГРАМУВАННЯ PYTHON ТА C++ ДЛЯ ROS

У контексті розробки модуля комп'ютерного зору для автономних роботизованих систем на базі ROS, вибір мов програмування має вирішальне значення. Python і C++ є двома основними мовами, що підтримуються ROS, і кожна з них пропонує унікальні переваги для різних аспектів системи.

Python вирізняється своєю простотою і швидкістю розробки. Завдяки динамічній типізації та великій кількості готових бібліотек, Python дозволяє розробникам швидко створювати прототипи та реалізувати складні алгоритми без зайвої складності. У контексті ROS, Python часто використовується для написання вузлів, що керують високорівневою логікою та взаємодією, такими як алгоритми планування, обробки даних сенсорів та машинного навчання. Це робить його ідеальним для задач, де необхідна швидка ітерація та тестування.

На відміну від Python, C++ пропонує більшу продуктивність і контроль над системними ресурсами, що є критично важливим для реалізації низькорівневих компонентів системи, таких як драйвери апаратного забезпечення або високопродуктивні алгоритми обробки зображень. Використання C++ у ROS дозволяє максимально ефективно використовувати обчислювальні ресурси, що є необхідним для завдань, які вимагають великої обчислювальної потужності та мінімальної затримки у відгуку, наприклад, для реальної роботи з 3D даними або виконання складних алгоритмів трекінгу в реальному часі.

Інтеграція Python і C++ в ROS відбувається через спеціалізовані інструменти та бібліотеки, які забезпечують їх взаємодію. ROS надає широкі можливості для обміну повідомленнями і виклику сервісів між вузлами, написаними на різних мовах, що забезпечує гнучкість у виборі підходу до розробки кожної частини системи. Зокрема, можна використовувати Python для розробки алгоритмічних прототипів та управління робочими потоками, тоді як C++ може бути використаний для оптимізації швидкодіючих компонентів.

Таким чином, комбінація Python і C++ у контексті ROS дозволяє розробникам оптимально використовувати переваги обох мов, забезпечуючи необхідний баланс між швидкістю розробки та продуктивністю системи.

Python ідеально підходить для розробки скриптів високого рівня в ROS, які відповідають за обробку зображень і взаємодію з іншими компонентами системи.

Використовуючи бібліотеки, такі як TensorFlow або PyTorch, можна легко

імплементувати нейронні мережі для розпізнавання об'єктів на зображеннях з камер.

Python добре інтегрується з OpenCV для виконання різних операцій з обробки зображень, таких як фільтрація, трансформація і навіть деякі аспекти машинного навчання. (Рис 2.1)[4]

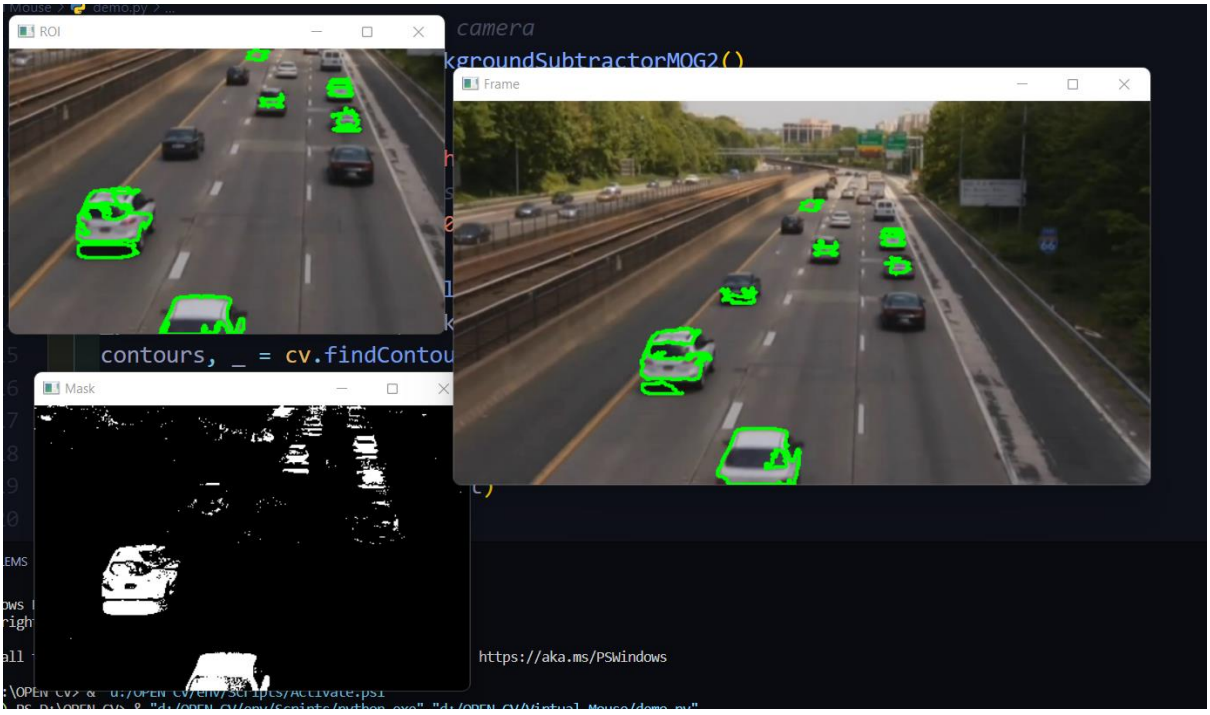


Рисунок 2.1 - Приклад визначення об’єктів за допомогою OpenCV та Python

Також він дає можливість для швидкого тестування і впровадження нових функцій без повної компіляції системи, що значно скорочує час розробки і тестування.

C++ використовується для більш низькорівневих завдань у ROS, де важлива швидкість обробки та мінімізація затримок, особливо при великих обсягах даних, які необхідні для обробки в реальному часі.

Також він часто використовується для написання драйверів для обладнання, такого як камери і лідари, що вимагають високої продуктивності та низької затримки.

Використання PCL (Point Cloud Library) у C++ дозволяє ефективно

обробляти 3D точкові хмари, отримані з лідарів або стереокамер, для завдань, таких як тривимірна реконструкція або SLAM (Simultaneous Localization and Mapping). (Рис 2.2)[5]

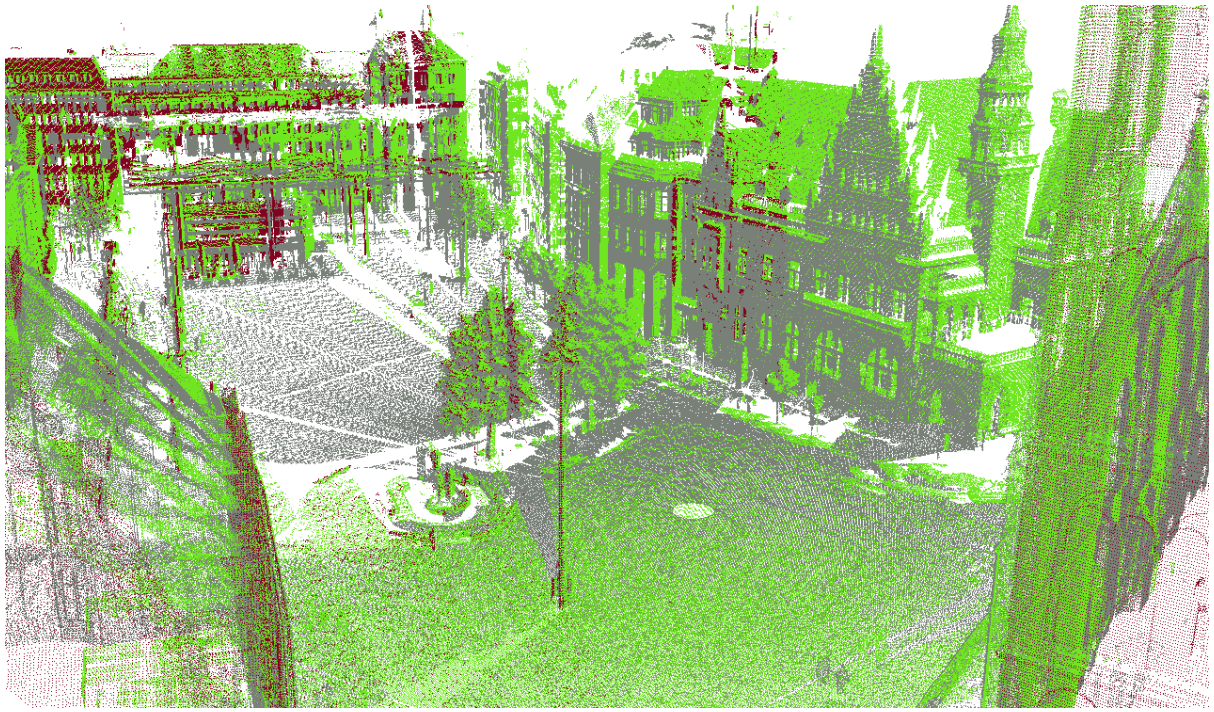


Рисунок 2.2 - Приклад використання PCL

Ще C++ ефективно працює з ROS для управління високопродуктивними процесами в реальному часі, забезпечуючи швидку обробку даних і відповідь системи.

Обидві мови можуть ефективно співпрацювати у проектах на базі ROS, де Python використовується для розробки і тестування алгоритмів на високому рівні, а C++ застосовується для оптимізації критично важливих частин системи. Таке поєднання дозволяє максимально використовувати переваги обох мов:

1. Модульність: Розробка окремих модулів на Python і C++ спрощує тестування, відладку і обслуговування системи.
2. Ефективність: Використання C++ для обробки вимогливих завдань забезпечує необхідну продуктивність системи, в той час як Python дозволяє швидко адаптувати і впроваджувати нові функції.

Цей підхід забезпечує гнучкість у розробці та можливість використання

ефективних технічних рішень для кожного конкретного аспекту проекту.

2.2 ВИКОРИСТАННЯ ROS У ПРОЕКТІ

2.2.1 КОНФІГУРАЦІЯ ТА УПРАВЛІННЯ ВУЗЛАМИ ROS

Управління вузлами в ROS є одним із ключових елементів архітектури модуля комп'ютерного зору, оскільки кожен вузол відповідає за виконання специфічних завдань, таких як обробка зображень, детекція об'єктів, трекінг та реконструкція сцен. Розглянемо детальніше як організована конфігурація та управління цими вузлами.
Основні аспекти конфігурації та управління вузлами:

Кожен вузол у ROS є окремим процесом, що виконується на основі скрипту Python або виконуваного файлу C++. Вузли можуть бути запущені індивідуально через рядок команд або групово через ROS launch файл.

Launch файли у ROS допомагають автоматизувати швидкий запуск вузлів з наперед визначеними параметрами та налаштуваннями. Вони використовуються для визначення простору імен, параметрів і залежностей між вузлами.
Керування вузлами: ROS пропонує ряд утиліт для керування вузлами:

1. rosnode: дозволяє переглядати факти про активні вузли, зв'язки між ними та запобігати їх роботі.
2. rostopic: використовується для моніторингу, публікації та підписки на теми, які вузли використовують для взаємодії.эrosservice: дозволяє взаємодіяти з сервісами, які вузли використовують для синхронної обробки запитів.

Використання Parameter Server: ROS Parameter Server дозволяє вузлам зберігати і отримувати параметри виконання у централізованому сховищі. Це забезпечує гнучке управління конфігурацією системи в реальному часі.

Використання різноманітних логів ROS для діагностики та моніторингу стану вузлів дозволяє швидко виявляти та усувати потенційні проблеми в

системі.

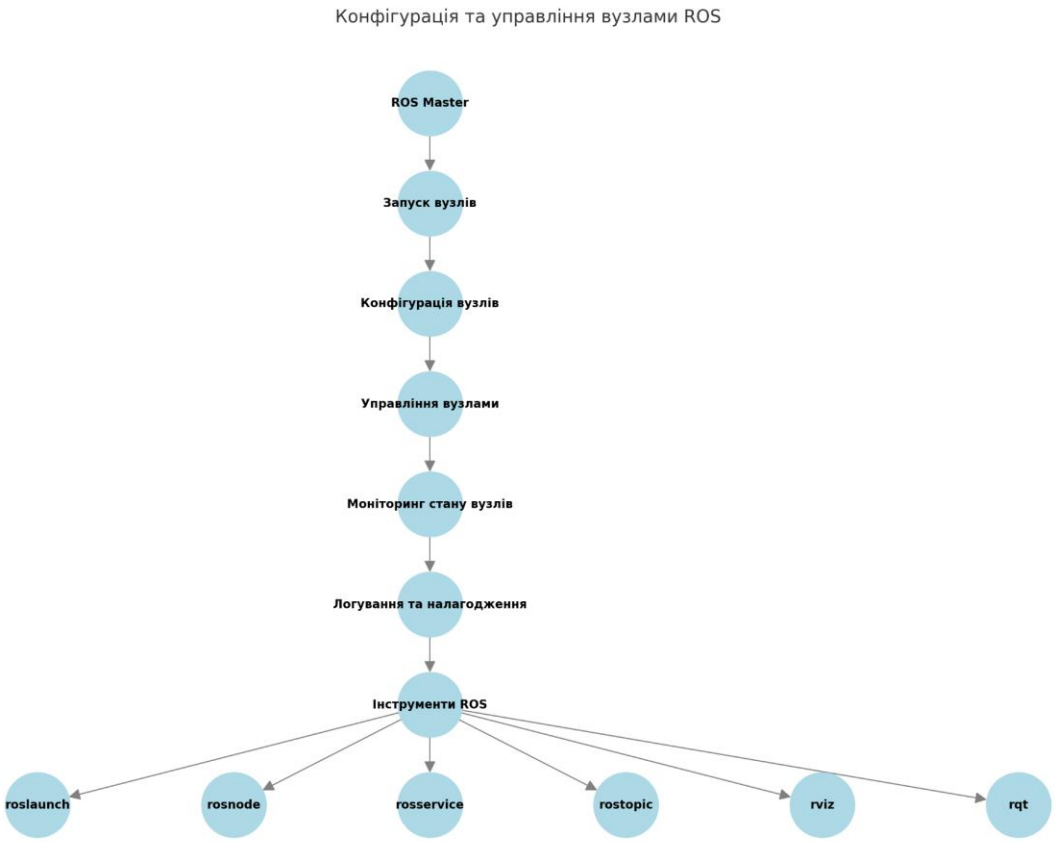


Рисунок 2.3 - Схема конфігурації та управління вузлами ROS

2.2.2 ВЗАЄМОДІЯ МІЖ ВУЗЛАМИ ЧЕРЕЗ TOPICS І SERVICES

В ROS вузли взаємодіють один з одним за допомогою двох основних механізмів: Topics і Services. Ці механізми дозволяють розподілити різні задачі обробки між вузлами, забезпечуючи модульність та гнучкість системи.

Topics в ROS використовуються для асинхронної взаємодії між вузлами. Вузол публікує повідомлення на певну тему, які можуть підписуватися інші вузли. Це дозволяє розподілити потоки даних у системі, де кожен вузол може незалежно від інших виконувати свої завдання, отримуючи та відправляючи необхідну інформацію.

Вузол, відповідальний за обробку зображень, може публікувати оброблені

дані (наприклад, виявлені об'єкти) на тему detected_objects. Інші вузли, такі як вузол навігації, можуть підписатися на цю тему для отримання інформації про об'єкти та використання їх для маршрутизації.

```
# Вузол публікує тему
pub = rospy.Publisher('detected_objects', String, queue_size=10)
pub.publish("object_detected")


# Вузол підписується на тему
def callback(data):
    rospy.loginfo("Received object info: %s", data.data)


sub = rospy.Subscriber('detected_objects', String, callback)
```

Рисунок 2.4 - Приклад публікації та підписки
на мові програмування Python

На відміну від Topics, Services в ROS забезпечують синхронну взаємодію. Вони дозволяють одному вузлу викликати сервіс іншого вузла, який виконує певні дії та повертає результат. Services ідеально підходять для виконання завдань, які потребують прямої взаємодії між вузлами та швидкої відповіді.

Вузол для трекінгу об'єктів може надавати сервіс track_object, який приймає дані про об'єкт і повертає інформацію про його поточне положення.

```python
# Визначення сервісу
from your_custom_srvs.srv import TrackObject, TrackObjectResponse

def handle_track_object(req):
    # Логіка для трекінгу об'єкта
    return TrackObjectResponse(result)
rospy.Service('track_object', TrackObject, handle_track_object)

# Виклик сервісу
rospy.wait_for_service('track_object')
try:
    track_object = rospy.ServiceProxy('track_object', TrackObject)
    resp = track_object("object_id")
    print("Tracking result:", resp.result)
except rospy.ServiceException as e:
    print("Service call failed:", e)
```

Рисунок 2.5 - Приклад вузла для трекінгу об’єктів на мові програмування Python

За допомогою Topics і Services, вузли в ROS можуть ефективно спілкуватися та координувати свої дії, забезпечуючи високу продуктивність та надійність роботизованих систем. Це також дозволяє системі бути масштабованою та легко адаптуватися до нових завдань і змін в оперативному середовищі.

2.2.3 ВИКОРИСТАННЯ PARAMETER SERVER ДЛЯ УПРАВЛІННЯ КОНФІГУРАЦІЯМИ СИСТЕМ

Parameter Server в ROS відіграє ключову роль у централізованому зберіганні та управлінні параметрами, що використовуються різними вузлами системи. Це дає мені змогу легко адаптувати поведінку системи до змін у

зовнішніх умовах або до нових вимог без необхідності змінювати і перезавантажувати весь код.

Основне призначення Parameter Server полягає у тому, що він зберігає всі конфігураційні параметри в форматі ключ/значення, забезпечуючи універсальний доступ до них з будь-якого місця в системі. Це дуже зручно, адже параметри можуть бути прочитані або змінені в реальному часі, що дає можливість системі бути гнучкою та адаптивною.

Parameter Server у моєму проекті дасть змогу виконувати деякі речі більш ефективно та якісно, наприклад:

Параметри, пов'язані з обладнанням, до прикладу - роздільна здатність камер, це може централізовано управлятись, що дозволяє легко змінювати їх в залежності від вимог до якості зображення або продуктивності системи.

Для отримання значення параметра використовую команду rospy.get_param, а для встановлення нового значення — rospy.set_param. Це дозволяє мінімізувати жорстку залежність коду від конкретних значень, роблячи систему більш модульною та легкою для тестування.

Також я використовую можливість видалення або перевірки наявності параметра через rospy.delete_param та rospy.has_param, що забезпечує додаткову гнучкість у управлінні станом системи.

Використання Parameter Server сприяє кращому розділенню логіки програми та конфігураційних даних, забезпечуючи ефективне масштабування проекту та полегшуючи управління складними системами з багатьма залежними параметрами.

Parameter Server в ROS дозволяє не лише зберігати і змінювати параметри на льоту, але й організовувати їх у ієрархічну структуру, що може дуже зручно для керування складними системами з багатьма модулями та рівнями налаштувань.

Параметри можуть бути організовані у вигляді дерева, де кожен вузол може мати власні підпараметри. Наприклад, для системи автоматичного водіння

можна мати окремі групи параметрів для детекції перешкод, навігації, та управління швидкістю.

```yaml
# Example
vision_system:
  resolution: "1920x1080"
  fps: 30
navigation:
  max_speed: 10
  obstacle_distance: 5
```

Рисунок 2.6 - Приклад ієрархічної структури

ROS дозволяє вузлам підписуватися на оновлення параметрів. Це означає, що коли параметр змінюється через Parameter Server, вузол може автоматично отримати сповіщення про це і відповідно адаптувати свою поведінку.

```python
import rospy

def param_callback(config, level):
    rospy.loginfo("Received reconfigure request with level: %d", level)

    update_system_configuration(config.max_speed, config.obstacle_distance)
    return config

if __name__ == "__main__":
    rospy.init_node("parameter_subscriber_example")
    srv = dynamic_reconfigure.server.Server(ConfigType, param_callback)
    rospy.spin()
```

Рисунок 2.7 - Приклад підписки на параметри

Управління доступом до Parameter Server є важливим аспектом у розробці безпечних роботизованих систем. ROS пропонує механізми для контролю доступу до параметрів, забезпечуючи, що лише авторизовані вузли можуть

змінювати критичні налаштування.

Можна використовувати схеми авторизації або шифрування для забезпечення того, що доступ до важливих параметрів матимуть лише вузли з відповідними правами доступу.

2.2.4 ВИКОРИСТАННЯ ІНСТРУМЕНТІВ ROS ДЛЯ ДЕБАГІНГУ ТА ТЕСТУВАННЯ СИСТЕМИ

У процесі розробки модуля комп'ютерного зору для автономних систем на базі ROS, важливо мати можливість ефективно відлагоджувати та тестувати кожен компонент системи. ROS надає декілька потужних інструментів, які допомагають в цьому, забезпечуючи глибоке розуміння внутрішніх процесів та спрощуючи виявлення помилок.

Перший інструмент, який було використано — це rqt. Це графічний інструментарій, який допомагає візуалізувати структуру системи та потоки даних. З його допомогою можна легко побачити, як різні вузли комунікують між собою, які дані вони обмінюють, що є незамінним при відладці зв'язків у складних системах. Ще один важливий інструмент — rviz, який є особливо корисним для роботи з комп'ютерним зором, оскільки дозволяє візуалізувати 3D-дані в реальному часі, такі як моделі оточення, що відображають, що "бачить" робот.

Коли справа доходить до запуску і тестування різних частин системи одночасно, я вдаюся до використання roslaunch. Цей інструмент дозволяє мне запускати комплексні сценарії з багатьма вузлами, що працюють взаємозалежно, з усіма необхідними параметрами та конфігураціями. Це робить процес тестування більш гладким та контрольованим.

Не менш важливим є інструмент rosbag, який дозволяє записувати всі повідомлення, що передаються між вузлами в системі, та відтворювати їх згодом. Це надзвичайно корисно для тестування, оскільки можна відтворити точний

сценарій багаторазово для відлагодження або демонстрації системи без необхідності повторного виконання всіх операцій в реальному часі.

2.3 ОБРОБКА ЗОБРАЖЕНЬ ТА ВИКОРИСТАННЯ БІБЛІОТЕК

2.3.1 РОЛЬ GAZEBO У СИМУЛЯЦІЇ ТА ДЕТЕКЦІЇ ОБ'ЄКТІВ

Gazebo - це потужне симуляційне середовище з відкритим вихідним кодом, яке широко використовується для тестування та розробки роботизованих систем. Воно дозволяє створювати реалістичні фізичні симуляції, які включають динамічні моделі роботів, сенсори та оточення.

Детектування об'єктів у Gazebo здійснюється за допомогою сенсорів, таких як камери, та лідарами. Ці сенсори можуть генерувати дані, які далі обробляються за допомогою алгоритмів детекції, інтегрованих в ROS. Для цього можна використовувати пакет simple_grasping, що обробляє дані з PointCloud сенсорів для виявлення об'єктів.

Використання Gazebo для обробки зображень у моєму проекті дозволяє точно моделювати роботу роботизованої системи, оптимізувати взаємодію з фізичним оточенням і тестувати різні сценарії без необхідності фізичного обладнання. Інтеграція цих технологій з ROS забезпечує високу адаптивність і реактивність системи.

2.3.2 ІНТЕГРАЦІЯ БІБЛІОТЕК З ROS

Інтеграція зовнішніх бібліотек з ROS є фундаментальною для розширення можливостей та ефективності автономних роботизованих систем. Це дозволяє використовувати спеціалізовані алгоритми та методи обробки даних, які не входять в стандартний набір інструментів ROS, але є критично важливими для

задач комп'ютерного зору та обробки 3D-даних.

Пакет simple_grasping є прикладом бібліотеки, що інтегрується з ROS для обробки 3D-даних. Інтеграція цих бібліотек вимагає знання про те, як правильно зв'язати їхні функції з вузлами ROS та як ефективно обмінюватися даними між різними частинами системи.

Процес інтеграції зазвичай включає встановлення та налаштування бібліотек. Пакети, як simple_grasping, часто доступні як репозиторії, які можна клонувати та компілювати за допомогою інструментів ROS, таких як colcon. Наприклад, для інтеграції simple_grasping з вашим проектом на базі ROS2, вам може знадобитися встановити додаткові залежності та налаштувати середовище розробки.

Вузли можуть бути написані на C++ або Python, і вони можуть імпортувати і використовувати функції з simple_grasping для обробки даних. Наприклад, вузол, який отримує дані з 3D камери, може використовувати функції simple_grasping для їх обробки та аналізу перед тим, як публікувати результати на інші теми або сервіси ROS. Важливою частиною інтеграції є ефективний обмін даними. Інструменти ROS дозволяють перетворювати зображення та інші типи даних між форматами, що використовуються ROS та зовнішніми бібліотеками, забезпечуючи гладку взаємодію між різними компонентами системи.

Використання цих бібліотек разом з ROS дозволяє створювати потужні, гнучкі та ефективні рішення для роботизованих систем, значно розширюючи можливості стандартних інструментів ROS і забезпечуючи високу продуктивність обробки зображень та 3D-даних.

2.4 АРХІТЕКТУРА СИСТЕМИ

2.4.1 СТРУКТУРА АРХІТЕКТУРИ. ВИЗНАЧЕННЯ ОСНОВНИХ КОМПОНЕНТІВ СИСТЕМИ ТА ЇХ ВЗАЄМОЗВ'ЯЗКИ

Архітектура системи для виявлення та відстеження об'єктів базується на трьох основних компонентах, які взаємодіють між собою за допомогою ROS Topics. Кожен з цих компонентів виконує певну функцію, забезпечуючи злагоджену роботу системи в цілому.

Першим компонентом є вузол **camera_node**, який відповідає за захоплення зображень з камери та їх публікацію в топіку /camera/image_raw. Камера, встановлена на робота, захоплює зображення навколишнього середовища, які потім передаються до вузла ROS. Вузол використовує бібліотеку OpenCV для отримання зображень з камери та перетворення їх у формат, сумісний з ROS. Це забезпечує ефективне та надійне захоплення даних для подальшої обробки.

Другим компонентом є вузол **object_detection_node**, який отримує зображення з топіку /camera/image_raw, обробляє їх для виявлення об'єктів та публікує результати у топіку /detected_objects/image. Цей вузол використовує методи машинного навчання та бібліотеку OpenCV для виявлення об'єктів на зображеннях. Обробка зображень включає перетворення їх у формат HSV та застосування маски для виділення об'єктів певного кольору.

Третім компонентом є вузол **object_tracking_node**, який отримує оброблені зображення з топіку /detected_objects/image, відстежує виявлені об'єкти та публікує результати у топіку /tracked_objects/image. Вузол використовує алгоритми трекінгу з бібліотеки OpenCV для відстеження об'єктів у реальному часі. Цей компонент забезпечує точне та ефективне відстеження об'єктів на зображеннях, що дозволяє роботу реагувати на зміни у навколишньому середовищі.

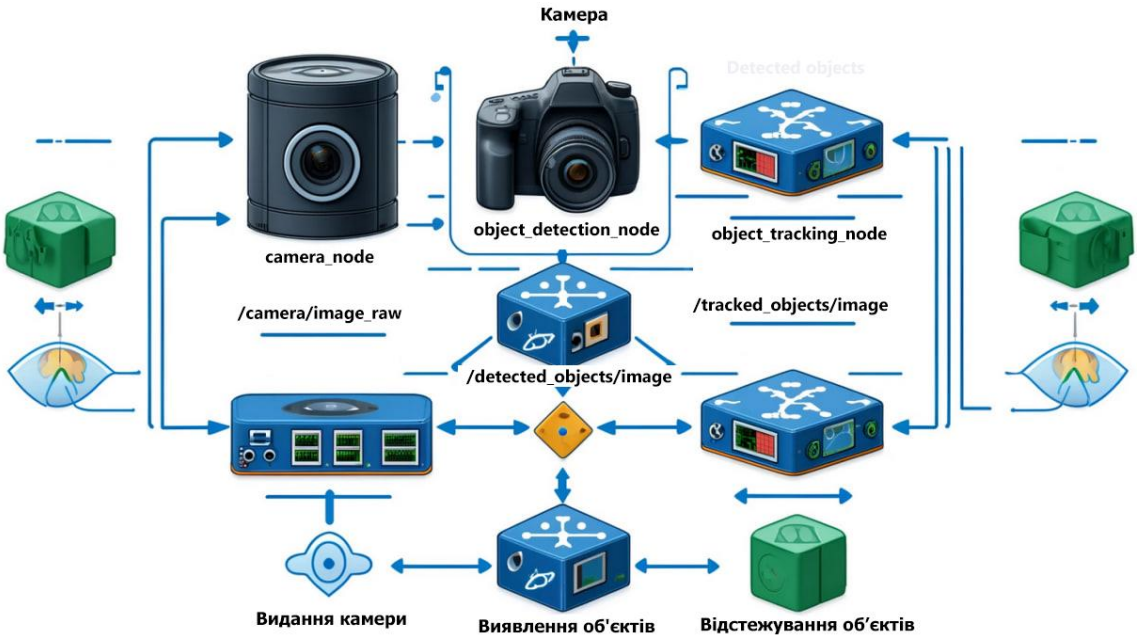


Рисунок 2.8 - Архітектурна схема системи виявлення та відстеження об'єктів

2.4.2 ВИКОРИСТАННЯ ROS. ЯК ІНТЕГРОВАНО РІЗНІ ВУЗЛИ, ЩОБ УПРАВЛЯТИ ДАНИМИ ТА ПРОЦЕСАМИ

Кожен вузол у нашій системі має чітко визначені завдання та взаємодіє з іншими вузлами через публікацію та підписку на певні Topics. Це дозволяє реалізувати гнучку та модульну систему, де кожен вузол можна легко замінити або оновити без значних змін у загальній архітектурі.

Вузол camera_node захоплює зображення з камери та публікує їх у топіку /camera/image_raw. Це забезпечує потік даних від камери до системи обробки зображень.

Вузол object_detection_node підписується на топік /camera/image_raw, обробляє зображення для виявлення об'єктів та публікує результати в топік /detected_objects/image. Це дозволяє вузлу object_tracking_node отримувати оброблені зображення для трекінгу. Вузол object_tracking_node підписується на топік /detected_objects/image, відстежує виявлені об'єкти та публікує результати

у топік /tracked_objects/image. Цей вузол використовує алгоритми трекінгу з бібліотеки OpenCV для відстеження об'єктів у реальному часі. Для налаштування та конфігурації вузлів використовуються параметри, які дозволяють змінювати поведінку вузлів без необхідності зміни коду. У launch-файлі можна вказати параметри, які будуть використовуватися вузлами. Це дозволяє зручно запускати всі вузли одночасно та задавати необхідні параметри для їх роботи. Використання параметрів у ROS дозволяє зручно змінювати налаштування вузлів без необхідності зміни коду. Вузол camera_node можна налаштувати для використання різних параметрів камери. Це дозволяє задавати індекс камери через параметр camera_index, що можна змінювати в launch-файлі або через команду rosparam.

Завдяки такій архітектурі та використанню інструментів ROS, ми отримуємо гнучку, модульну систему, яка легко налаштовується та адаптується до різних вимог та умов експлуатації.

2.4.3 ОБРОБКА ДАНИХ

Обробка зображень для виявлення об'єктів включає кілька кроків, які забезпечують точне і ефективне виявлення. Основним методом є використання кольорових масок для виявлення об'єктів певного кольору.

Алгоритм виявлення об'єктів використовує перетворення зображень у кольорову модель HSV (Hue, Saturation, Value) і застосування масок для виділення об'єктів певного кольору. Цей метод дозволяє ефективно виділяти об'єкти на зображенні, використовуючи прості кольорові маски.

Алгоритм CSRT забезпечує високу точність трекінгу об'єктів навіть у складних умовах. Вузол object_tracking_node використовує цей алгоритм для відстеження об'єктів на зображеннях.

Вузли системи обмінюються даними через ROS Topics, що забезпечує гнучкість та модульність системи. Дані передаються у вигляді повідомлень, які можуть бути легко оброблені та використані іншими вузлами.

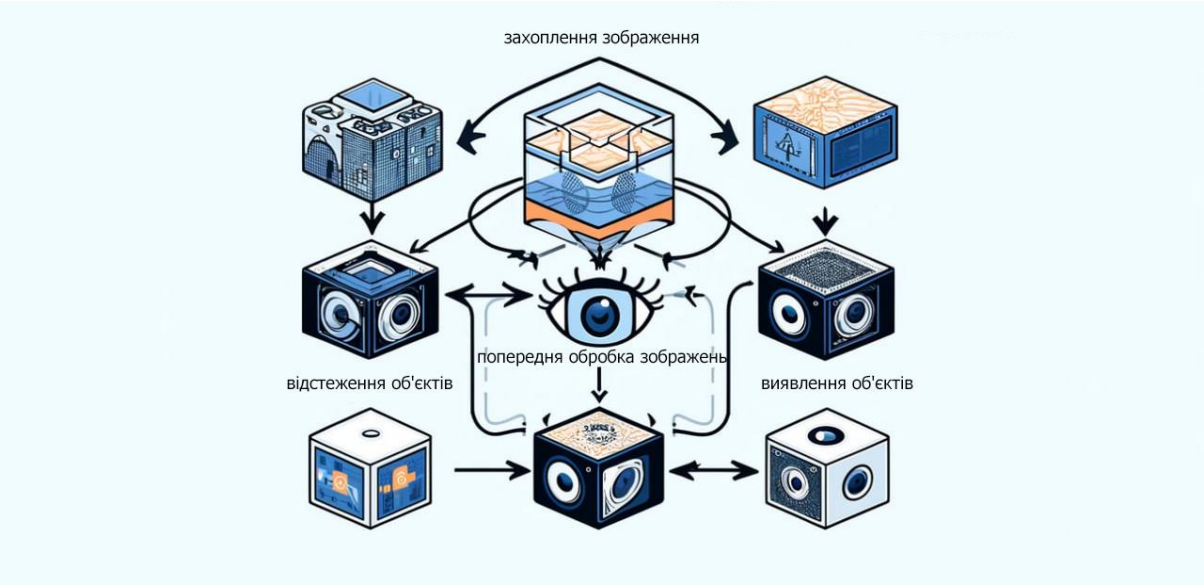


Рисунок 2.9 - Алгоритм дій системи виявлення та відстеження об'єктів

Захоплення зображень з камери (camera_node). Попередня обробка зображень (object_detection_node). Перетворення у HSV-колірний простір та застосування кольорових масок. Виявлення об'єктів (object_detection_node). Обробка зображень для виявлення об'єктів. Відстеження об'єктів (object_tracking_node). Відстеження виявлених об'єктів за допомогою алгоритму CSRT.

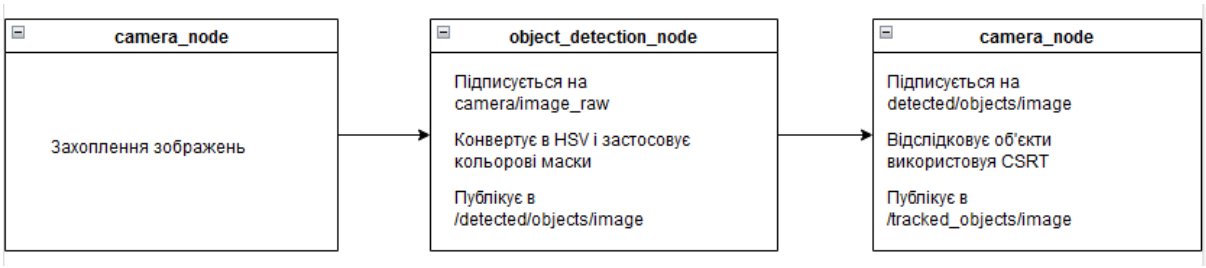


Рисунок 2.10 - Функціональна схема системи виявлення та відстеження об'єктів

2.4.4 ІНТЕРФЕЙС КОРИСТУВАЧА

Візуалізація результатів роботи модуля є критично важливою, оскільки дозволяє користувачу в реальному часі спостерігати, як система виявляє та відстежує об'єкти. Для цього ми використовуємо бібліотеку OpenCV, яка надає можливість відображення зображень у вікні. Відображення результатів включає виділення об'єктів кольоровими рамками, що робить процес більш наочним і зрозумілим для користувача.

Інтерфейс також повинен дозволяти користувачу зупиняти та запускати обробку зображень, а також переглядати різні етапи обробки зображень, такі як попередня обробка, виявлення та трекінг об'єктів. Це дозволяє краще розуміти, як працює система, і знаходити можливі помилки або місця для покращення.

Для забезпечення гнучкості та точності роботи системи необхідно надати користувачу можливість налаштовувати параметри виявлення та трекінгу об'єктів. Це включає налаштування таких параметрів, як діапазон кольорів для виявлення об'єктів, параметри алгоритму трекінгу, чутливість до руху та інші важливі параметри.

Для реалізації цього завдання можна використовувати графічні інтерфейси, створені за допомогою бібліотек, таких як Tkinter або PyQt. Ці інтерфейси можуть включати слайдери, кнопки та поля введення, які дозволяють легко змінювати параметри та одразу бачити результати змін. Наприклад, користувач може налаштувати діапазон кольорів у HSV-колірному просторі для більш точного виявлення об'єктів.

Крім налаштування параметрів, інтерфейс користувача також може забезпечувати зворотний зв'язок, який дозволяє користувачу бачити, як зміна параметрів впливає на результати виявлення та трекінгу. Це може бути реалізовано через виведення повідомлень, графіків або інших візуальних елементів, які показують ефективність роботи системи.

Користувач може також зберігати та завантажувати налаштування, що

дозволяє швидко перемикатися між різними конфігураціями для різних умов експлуатації. Це особливо корисно у випадках, коли система використовується в різних середовищах або для різних задач, де потрібні різні налаштування параметрів.

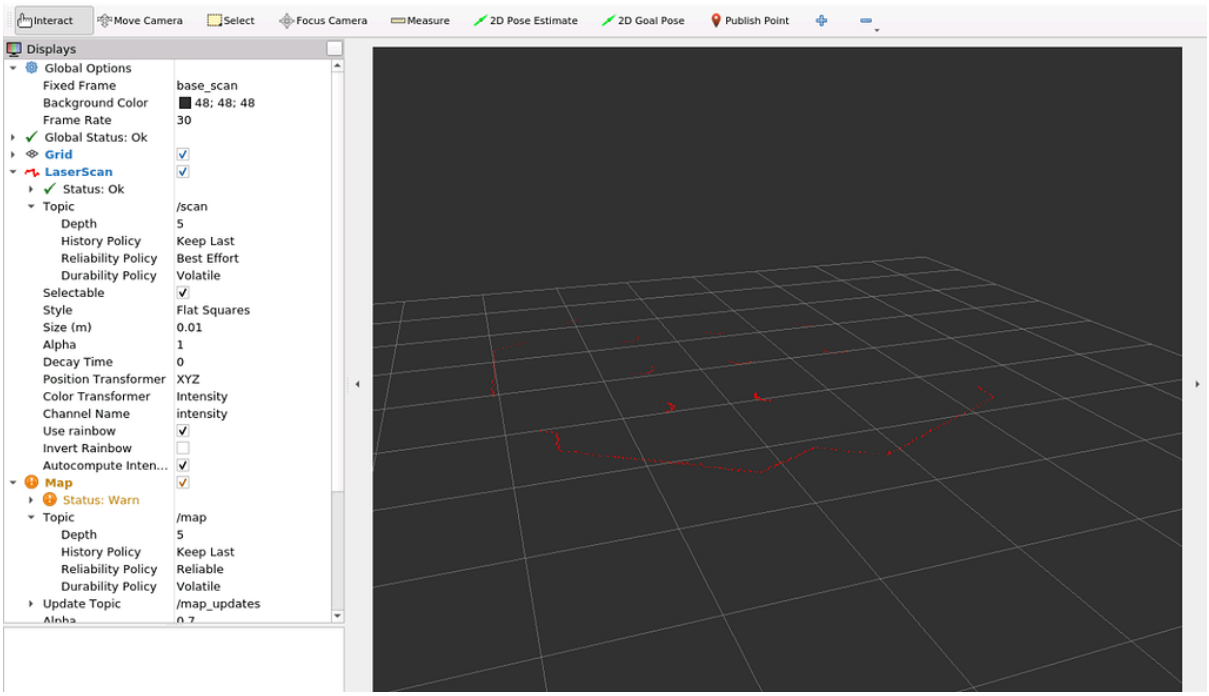


Рисунок 2.11 - Графічний інтерфейс користувача - Rviz2

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі було розглянуто архітектуру системи для виявлення та відстеження об'єктів, яка базується на використанні ROS та бібліотеки OpenCV. Ця система була детально описана, починаючи з вибору операційної системи та мов програмування, і завершуючи інтеграцією вузлів та обробкою даних.

Вибір операційної системи Ubuntu був обґрунтований її надійністю та підтримкою інструментів ROS, що робить її ідеальною для розробки робототехнічних систем. Використання мов програмування Python та C++ дозволяє максимально ефективно реалізувати вузли ROS, поєднуючи простоту написання коду та високопродуктивні можливості.

Ми також детально розглянули, як конфігурувати та керувати вузлами ROS, забезпечуючи їхню взаємодію через Topics і Services. Ці аспекти є ключовими для побудови гнучкої та масштабованої системи, яка може легко адаптуватися до різних умов і завдань.

Обробка зображень є центральною частиною системи виявлення та відстеження об'єктів. Використання бібліотеки OpenCV дозволяє реалізувати складні алгоритми обробки зображень, такі як перетворення у HSV-колірний простір та застосування кольорових масок для виявлення об'єктів. Завдяки цьому система може ефективно виділяти та відстежувати об'єкти в реальному часі.

Інтеграція вузлів ROS, які відповідають за захоплення зображень, їхню попередню обробку, виявлення та трекінг об'єктів, була представлена у вигляді блок-схем та детальних описів. Це дозволяє краще розуміти, як дані передаються і обробляються на кожному етапі.

Інтерфейс користувача є важливою складовою системи, яка забезпечує зручний спосіб візуалізації результатів та налаштування параметрів. Відображення результатів роботи модуля у режимі реального часу дозволяє користувачам бачити, як система виявляє та відстежує об'єкти, що значно полегшує процес налагодження та оптимізації.

РОЗДІЛ 3. РОЗРОБКА СИСТЕМИ ДЛЯ ДЕТЕКЦІЇ ОБ'ЄКТІВ З ВИКОРИСТАННЯМ ROS

3.1 СИМУЛЯЦІЯ ОБ'ЄКТІВ В 3D ПРОСТОРІ

Створення середовища для тестування системи детекції об'єктів починається з налаштування Gazebo, одного з інструментів для створення тривимірних симуляцій роботів. Gazebo дозволяє моделювати фізичні властивості, такі як гравітація, тертя, зіткнення, що є важливими для відтворення умов роботи роботизованих систем.

Спочатку необхідно встановити Gazebo та інтегрувати його з ROS2. Це передбачає налаштування відповідних плагінів, які забезпечують взаємодію між Gazebo та ROS2, дозволяючи отримувати зображення з камер та дані з інших сенсорів. Після встановлення та налаштування Gazebo, створюється базова сцена. Ця сцена включає освітлення та камери, що забезпечують базову інфраструктуру для симуляції. Важливо додати об'єкти для детекції, такі як кубики, циліндри або більш складні форми, та налаштувати їх фізичні властивості.

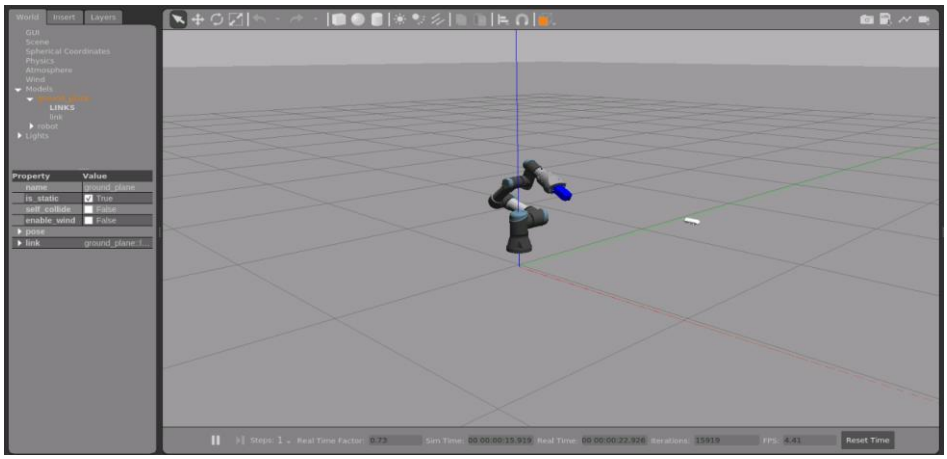


Рисунок 3.1 - Створена базова симуляційна сцена в Gazebo із моделлю універсального робота

Організація робочого простору ROS (modeling) має велике значення для ефективної розробки та тестування системи. Структура робочого простору включає три основні директорії: build, devel та src. Директорія devel містить файли та бібліотеки, створені під час розробки, а також скрипти для налаштування середовища. У директорії src зберігаються вихідні файли проекту, включаючи основну директорію для пакетів integrations.

У пакеті integrations розміщені різні пакети, які необхідні для симуляції та інтеграції з ROS. Пакет capture _common містить опис захоплювача, який використовується для маніпулювання об'єктами. Пакет gazebo_ros_pkgs призначений для інтеграції Gazebo з ROS, забезпечуючи обмін даними між симуляційним середовищем та ROS. Додаткові функції забезпечуються пакетами gazebo-pkgs, roboticsgroup_gazebo_plugin, які включають плагіни та сенсори для розширення можливостей симуляції. Пакет robot містить моделі універсальних роботів, які можуть бути використані для тестування різних сценаріїв.

Файл grasp_box.urdf містить опис об'єкта для симуляції, який використовується для тестування алгоритмів детекції та маніпулювання. Файл CMakeLists.txt використовується для конфігурації збірки проекту, забезпечуючи правильне налаштування залежностей та параметрів збірки.

Інтеграція симуляції з ROS2 здійснюється за допомогою запуску симуляції в Gazebo з підтримкою ROS2. Це забезпечується використанням спеціальних плагінів, які дозволяють отримувати дані з камер та інших сенсорів, а також налаштовувати теми (topics) та сервіси для обміну даними між вузлами ROS2 та Gazebo. Наприклад, для отримання зображень з камери використовуються теми типу /camera/image_raw, які передають зображення в реальному часі до вузлів ROS2, що обробляють ці дані.

Симуляція об'єктів в 3D просторі за допомогою Gazebo створює реалістичне середовище для тестування системи детекції об'єктів. Це дозволяє забезпечити високий рівень точності та надійності розробленої системи, що є

критично важливим для успішної роботи роботизованих систем в реальних умовах. Використання Gazebo спрощує процес розробки, дозволяючи швидко і легко тестувати різні сценарії та налаштування, виявляти та усувати помилки на ранніх етапах розробки.

3.2 ВИЯВЛЕННЯ ОБ’ЄКТІВ

3.2.1 СПРИЙНЯТТЯ ОБ’ЄКТІВ

Сприйняття об’єктів у системі детекції об’єктів починається з отримання даних з камер, які можуть бути як звичайними RGB-камерами, так і камерами глибини. Використання камер глибини забезпечує додаткову інформацію про відстань до об’єктів, що критично важливо для точного визначення їхнього положення у просторі. Дані з камер передаються до ROS2 за допомогою спеціально налаштованих топіків. Наприклад, топік /camera/image_raw використовується для передачі зображень з RGB-камери, а /camera/depth/image_raw — для передачі зображень з камери глибини. Ці топіки дозволяють системі отримувати зображення в реальному часі та використовувати їх для подальшого аналізу.

Після отримання даних з камер починається етап попередньої обробки зображень. Цей процес включає кілька методів, які допомагають підвищити якість зображень та спростити подальше розпізнавання об'єктів. Наприклад, медіанний фільтр використовується для зменшення шуму на зображеннях, а алгоритми нормалізації дозволяють вирівняти яскравість зображень. Підвищення контрасту також може бути корисним для покращення видимості об'єктів на зображеннях.

Після попередньої обробки зображення аналізуються за допомогою алгоритмів комп’ютерного зору. Існує багато різних методів, які можуть бути

використані для цього завдання, включаючи алгоритми машинного навчання та глибокого навчання.

Калібрування камер є ще одним важливим аспектом сприйняття об'єктів. Процес калібрування дозволяє точно визначити параметри камер, такі як фокусна відстань, центр зображення та інше. Це необхідно для забезпечення точного співвідношення між зображенням і реальним простором. Калібрування включає зйомку зображень спеціального калібрувального шаблону з різних кутів та подальшу обробку цих зображень для визначення параметрів камери.

Проект структурований у робочому просторі ROS (ros2_ws), що містить декілька основних директорій: build, install, log та src. Директорія src містить кілька ключових пакетів. Пакет, який забезпечує інтерфейс бібліотеки Transform Library для ROS2. Пакет, який містить повідомлення, необхідні для пакету main_capturing. Також використовується пакет, який забезпечує інтерфейс Point Cloud Library (PCL) для ROS2, яка є важливою для обробки тривимірних даних. Пакет main_capturing містить основні програми для виявлення об'єктів і визначення їх координат для подальшого захоплення.

В проекті було сконцентровано увагу саме на створенні пакету main_capturing. Додаткові пакети для візуалізації та відображенні були використані для полегшення тестування та презентації результатів.

У пакеті main_capturing, в піддиректорії src, розміщені основні програми пакету, такі як capturing_sensing.cpp та capturing_segmentation.cpp. Перший скрипт, написаний на C++, забезпечує сервер дій /find_objects, який можна викликати для запуску конвеєра детекції об'єктів. Другий скрипт, також написаний на C++, обробляє дані, отримані з 3D камери PointCloud, для виявлення об'єктів, які можна захопити в сцені.

Таким чином, сприйняття об’єктів включає кілька ключових етапів: збір даних з камер, попередню обробку зображень, аналіз зображень за допомогою алгоритмів комп’ютерного зору та калібрування камер. Ці етапи забезпечують основу для подальших процесів, таких як розпізнавання об'єктів та їх

маніпулювання. Ключові пакети проекту забезпечують всі необхідні функції для успішної реалізації системи сприйняття об'єктів у автономних роботизованих системах.

3.2.2 ПАКЕТ main_capturing

Пакет main_capturing є ключовим компонентом системи детекції об'єктів, розробленої для використання в ROS2. Цей пакет містить основні програми, які дозволяють здійснювати виявлення об'єктів і визначення їх координат для подальшого захоплення. Розглянемо детально його структуру і основні функції.

В рамках проекту пакет main_capturing включає такі піддиректорії:

- **include**: містить заголовкові файли.
- **scripts**: містить різні скрипти.
- **src**: основні програмні файли пакету.
- **test**: тести для перевірки функціональності пакету.

Основні програми пакету розташовані в директорії src. Зокрема, зосередимося на двох важливих скриптах: capturing_sensing.cpp та capturing_segmentation.cpp.

В файлі capturing_sensing.cpp розташована логіка, яка написана на C++. Вона забезпечує сервер дій під назвою /find_objects. Сервер дій може бути викликаний для запуску конвеєра детекції об'єктів. Основна функція цього скрипту полягає у використанні алгоритмів комп'ютерного зору для аналізу зображень, отриманих з камер, і визначення координат об'єктів, які можуть бути захоплені.

При виклику сервера дій /find_objects скрипт починає обробку зображень, застосовуючи попередньо визначені алгоритми для виявлення об'єктів. Цей процес включає попередню обробку зображень для зменшення шуму і підвищення контрасту, а також застосування методів глибокого навчання для точного розпізнавання об'єктів.

В файлі capturing_segmentation.cpp розташований скрипт, який також написаний на C++ і виконує основну функцію обробки даних, отриманих з 3D камери PointCloud. Основна мета цього скрипту — виявлення об'єктів, які можна захопити у сцені. Він аналізує тривимірні дані, щоб визначити положення об'єктів і їхню доступність для захоплення.

Скрипт обробляє PointCloud дані, використовуючи бібліотеку PCL (Point Cloud Library), що дозволяє здійснювати фільтрацію, сегментацію та класифікацію точкових хмар. Завдяки цьому, система може точно визначити об'єкти, які знаходяться у полі зору камери, і передати їх координати для подальшого захоплення маніпулятором.

Пакет main_capturing інтегрується з іншими пакетами ROS2, такими як geometry2 та grasping_msgs. Пакет geometry2 забезпечує інтерфейс бібліотеки Transform Library для роботи з геометричними даними, необхідними для точного позиціонування маніпулятора відносно об'єктів. Пакет grasping_msgs містить необхідні повідомлення для пакету main_capturing, що дозволяють здійснювати обмін даними між різними компонентами системи.

Процес використання пакету main_capturing включає кілька ключових етапів:

1. Запуск сервера дій /find_objects для ініціації процесу детекції об'єктів.
2. Обробка зображень, отриманих з камер, за допомогою скрипту capturing_sensing.cpp.
3. Обробка PointCloud даних для виявлення об'єктів за допомогою скрипту capturing_segmentation.cpp.
4. Передача координат виявлених об'єктів до маніпулятора для їх захоплення.

Завдяки пакету main_capturing, система детекції об'єктів отримує необхідні інструменти для точного визначення положення об'єктів і їх подальшого захоплення, що є критично важливим для автономних роботизованих систем.

3.2.3 ДАТЧИК КАМЕРИ ГЛИБИНИ

Для виявлення об'єктів у тривимірному просторі використовуються дані з камер глибини, які надають важливу інформацію про відстань до об’єктів. Камера глибини дозволяє створювати тривимірне зображення сцени, що є критично важливим для точного визначення положення об'єктів і успішного їх захоплення маніпулятором. Камери глибини, такі як RGB-D камери, забезпечують одночасно і колірне зображення, і глибину, що дозволяє більш точно ідентифікувати об'єкти в просторі.

Спочатку потрібно розташувати об'єкт у симуляції. Це об'єкт, який буде використовуватися для тестування системи захоплення. Після підготовки середовища ROS і запуску симуляції з потрібною моделлю об'єкта, наприклад, grasp_box, об'єкт з’являється в сцені перед маніпулятором. Це дозволяє системі бачити і взаємодіяти з об'єктом у контрольованому середовищі.

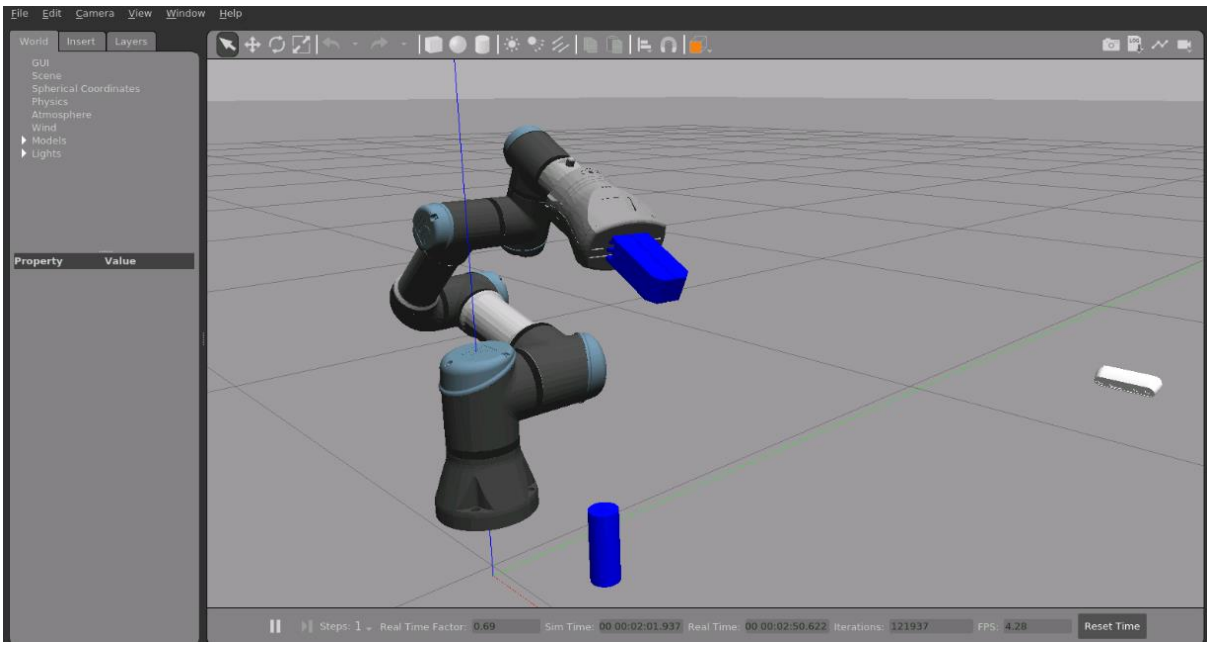


Рисунок 3.3 - Розташований циліндричний об’єкт у симуляції

Після розташування об'єкта починається аналіз даних, що публікуються камерою глибини. Камера глибини постійно генерує і публікує дані у вигляді зображень і точкових хмар (PointCloud). Ці дані передаються через теми ROS,

що дозволяє іншим компонентам системи отримувати і обробляти їх. Для аналізу даних камери можна використовувати інструменти ROS, такі як rostopic, для перегляду всіх тем, які публікує камера.

Топік, який містить PointCloud дані, є ключовою для виявлення об'єктів, оскільки вона надає тривимірну інформацію про сцену. Ці дані використовуються програмами сприйняття для визначення місцезнаходження об'єктів у сцені.

Оскільки дані публікуються в ROS1, їх потрібно конвертувати до ROS2, оскільки система працює на ROS2. Для цього використовується спеціальний вузол, який називається parameter_bridge. Цей вузол забезпечує конвертацію даних між ROS1 і ROS2, що дозволяє системі використовувати всі доступні дані незалежно від версії ROS, в якій вони були створені. Спочатку завантажуються всі необхідні параметри, а потім запускається вузол parameter_bridge, який виконує конвертацію даних.

Після конвертації даних можна перевірити, чи топіки правильно конвертуються і передають дані. Для цього можна використовувати Rviz2 — потужний інструмент для візуалізації даних у ROS2. У Rviz2 можна налаштувати відображення PointCloud даних, що дозволяє побачити тривимірне зображення сцени і переконатися, що дані конвертуються правильно. Візуалізація в Rviz2 дозволяє оцінити якість даних і переконатися, що вони можуть бути використані для виявлення об'єктів.

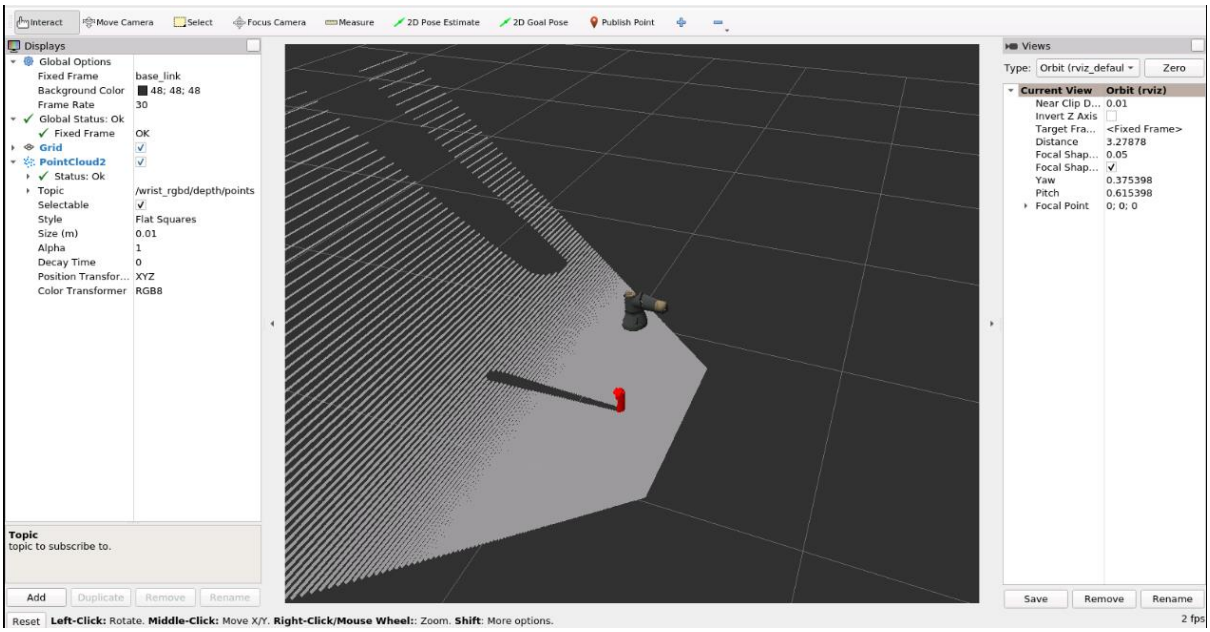


Рисунок 3.4 - Візуалізація в Rviz2

Інтеграція камери глибини з ROS2 забезпечує ефективне отримання і обробку тривимірних даних, що є важливим етапом у процесі виявлення і захоплення об'єктів у симуляції. Точні дані глибини дозволяють системі точно визначити положення об'єктів і забезпечити їх успішне захоплення маніпулятором. Використання даних з камер глибини значно підвищує точність і надійність системи детекції об'єктів, що є критично важливим для автономних роботизованих систем.

Таким чином, процес виявлення об'єктів за допомогою камери глибини включає кілька важливих етапів: підготовку середовища ROS, спавн об'єкта у симуляції, аналіз даних з камери, конвертацію даних з ROS1 до ROS2 і візуалізацію даних у Rviz2. Виконання цих кроків забезпечує ефективне використання тривимірних даних для виявлення і захоплення об'єктів у просторі, що значно підвищує можливості і надійність автономних роботизованих систем.

3.2.4 ЗАПУСК ВУЗЛІВ ВИЯВЛЕННЯ ОБ’ЄКТІВ ТА ВІЗУАЛІЗАЦІЯ ВЖЕ ВИЯВЛЕНИХ ОБ’ЄКТІВ В RVIZ2

Запуск вузлів виявлення об'єктів є важливим етапом у процесі створення системи комп'ютерного зору для автономних роботизованих систем. Це включає не лише налаштування середовища ROS2, але й запуск відповідних вузлів, які здійснюють обробку даних з камер і сенсорів для виявлення об'єктів. Ось більш детальний опис цього процесу.

Спочатку потрібно підготувати середовище ROS2, забезпечити підключення камер глибини та RGB, а також налаштувати всі необхідні компоненти системи. Після цього можна переходити до запуску вузлів, що відповідають за виявлення об'єктів. Основні вузли для цієї задачі знаходяться в пакеті main_capturing який містить кілька важливих скриптів, зокрема capturing_sensing.cpp та capturing_segmentation.cpp.

Щоб запустити вузли виявлення об'єктів, спочатку слід запустити основні сервіси ROS2, які забезпечують обмін даними між різними вузлами. Після цього можна запустити скрипти для обробки зображень і даних з камер глибини. Основний скрипт, який потрібно запустити, це capturing_sensing_node. Цей вузол відповідає за обробку даних з камер і визначення координат об'єктів, які можуть бути захоплені маніпулятором.

Після запуску вузла capturing_sensing_node створюється новий сервер дій під назвою /find_objects. Цей сервер дій може бути викликаний для запуску конвеєра детекції об'єктів. Коли сервер дій запущений, можна перевірити його наявність за допомогою команди ros2 action list. Ця команда покаже всі доступні сервери дій, і серед них має бути /find_objects.

Запуск вузла також створює дві додаткові теми: /object_cloud і /support_cloud. Ці теми використовуються для візуалізації об'єктів і поверхонь підтримки у Rviz2. Тема /object_cloud містить інформацію про об'єкти, які були виявлені, а тема /support_cloud показує поверхні, на яких розміщені ці об'єкти. Це дозволяє візуально оцінити, що саме бачить і розпізнає робот у своїй робочій зоні.

Щоб активувати виявлення об'єктів, потрібно викликати сервер дій /find_objects. Це можна зробити за допомогою команди ros2 action send_goal, вказавши потрібні параметри. Параметр plan_grasps визначає, чи потрібно обчислювати можливі захоплення для виявлених об'єктів. Наприклад, якщо plan_grasps встановлено в false, система лише виявляє об'єкти, але не обчислює можливі способи їх захоплення.

Після виклику сервера дій система починає шукати об'єкти у робочій зоні робота. Це може зайняти деякий час, оскільки обробка тривимірних даних і виявлення об'єктів є досить складним завданням. Коли система знайде об'єкти, вона поверне результат, який містить інформацію про виявлені об'єкти та їхні координати. Результат також містить інформацію про поверхні підтримки, що дозволяє зрозуміти, де саме розташовані виявлені об'єкти.

Результат, який повертає сервер дій, може бути досить великим і містити багато інформації. Наприклад, він може містити масив об'єктів GraspableObject[] objects і масив поверхонь Object[] support_surfaces. Ця інформація використовується для подальшої обробки і визначення оптимальних способів захоплення об'єктів. Візуалізація цих даних у Rviz2 значно спрощує розуміння того, що саме виявляє і бачить робот.

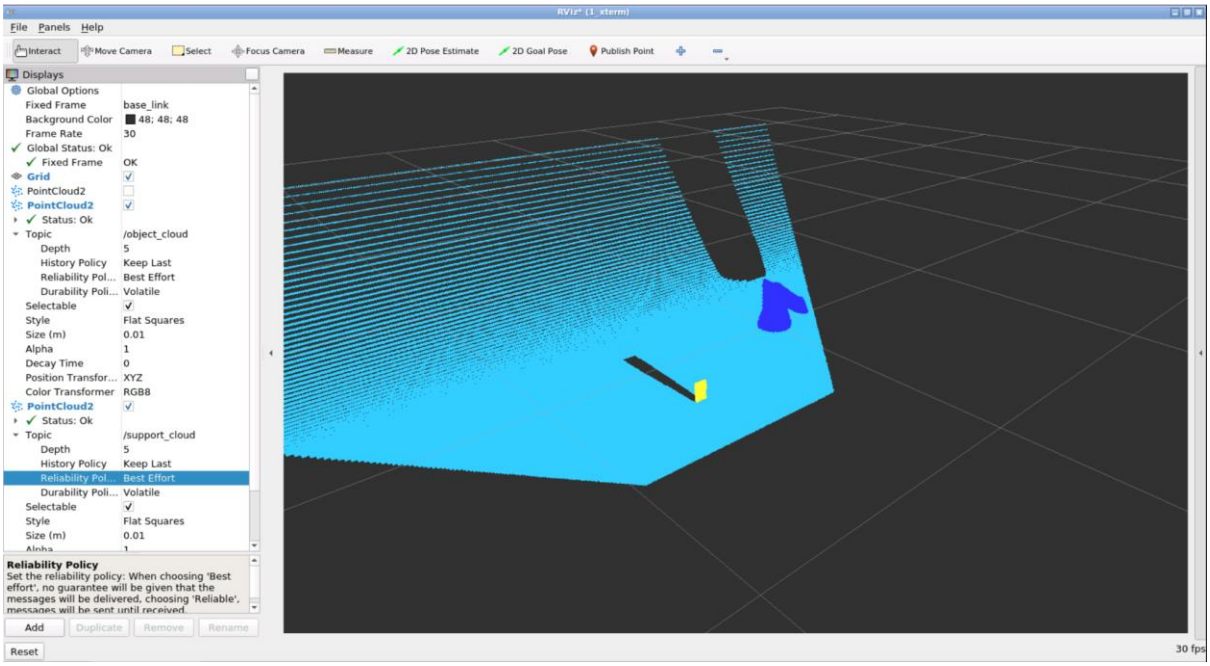


Рисунок 3.5 - Візуалізація в Rviz2 виявлених об’єктів

На (рис. 3.5) можна побачити, як система знайшла та відобразила розташовані в полі зору камери об’єкти.

Процес запуску вузлів виявлення об'єктів включає кілька важливих кроків, кожен з яких є критично важливим для забезпечення успішної роботи системи. Підготовка середовища, запуск вузлів, активація сервера дій і візуалізація результатів у Rviz2 забезпечують ефективне виявлення об'єктів і підготовку до їх захоплення.

3.3 ТЕСТУВАННЯ

Під час розробки системи для виявлення об'єктів особливу увагу було приділено тестуванню кожного окремого компонента. Це дозволило переконатися, що всі частини системи працюють належним чином ще до їх інтеграції.

Перше, що варто було протестувати - це взаємодія між ROS1 та ROS2. Для цього я використовував міст (ROS1 Bridge), який давав змогу налаштовувати середовище в ROS1, але комунікувати та знаходити об’єкти за допомогою ROS2.

Робота над модулем capturing_sensing.cpp розпочалася з тестування сервера дій /find_objects. Я відправляв різні запити на сервер, використовуючи тестові зображення з різними об'єктами та умовами освітлення. Мета полягала в тому, щоб переконатися, що сервер правильно обробляє запити та повертає очікувані результати.

Одним із важливих аспектів було перевірити, чи сервер правильно розпізнає об'єкти на зображеннях. Для цього я використовував кілька наборів зображень з різними об'єктами та аналізував точність розпізнавання. Також важливою була стабільність роботи сервера під час обробки великої кількості запитів.

Тестування показало, що всі компоненти системи працюють коректно. Сервер дій /find_objects успішно обробляв тестові запити та правильно визначав об'єкти на зображеннях.

Це тестування підтвердило, що всі окремі компоненти системи працюють

належним чином і готові до подальшого інтеграційного тестування.

ВИСНОВОК ДО РОЗДІЛУ 3

У цьому розділі було розглянуто кілька ключових аспектів розробки системи для виявлення об'єктів з використанням ROS2. Цей процес є надзвичайно важливим для створення ефективних і надійних автономних роботизованих систем, здатних виконувати складні завдання в різних умовах.

По-перше, було розглянуто створення симуляційного середовища за допомогою Gazebo. Це дозволяє розробникам тестувати і налагоджувати систему в умовах, що максимально наближені до реальних. Використання Gazebo забезпечує можливість моделювання різноманітних фізичних властивостей, таких як гравітація, тертя та зіткнення. Це є критично важливим для реалістичного моделювання роботи роботизованих систем.

По-друге, було детально розглянуто процес сприйняття об'єктів. В цьому контексті особливу увагу було приділено камерам глибини та їхньому калібруванню. Камери глибини забезпечують тривимірну інформацію про об'єкти, що є необхідним для точного визначення їхнього положення у просторі. Калібрування камер забезпечує точність вимірювань, що є критично важливим для успішної роботи системи.

Третім ключовим аспектом є створення пакету main_capturing, який забезпечує основні функції для виявлення об'єктів і визначення їх координат. Цей пакет включає кілька важливих компонентів, які дозволяють здійснювати детекцію об'єктів у реальному часі. Використання алгоритмів комп'ютерного зору дозволяє аналізувати дані, отримані з камер, і визначати координати об'єктів, які можуть бути захоплені маніпулятором.

Запуск вузлів виявлення об'єктів включає кілька важливих кроків, таких як підготовка середовища, запуск основних сервісів та активація серверів дій. Це забезпечує можливість обробки даних у реальному часі та виявлення об'єктів у робочій зоні робота. Візуалізація результатів у Rviz2 дозволяє розробникам оцінити якість роботи системи і переконатися в її правильній конфігурації.

Загалом, цей розділ демонструє, що розробка системи для виявлення об'єктів є складним і багатоступеневим процесом, який вимагає ретельного планування і виконання. Кожен з розглянутих етапів є критично важливим для забезпечення високої точності і надійності системи. Це включає не лише налаштування симуляційного середовища та сприйняття об'єктів, але й інтеграцію спеціалізованих пакетів і запуск вузлів. Виконання цих етапів забезпечує створення ефективної і надійної автономної роботизованої системи, здатної виконувати складні завдання у різних умовах.

Варто зазначити, що розробка такої системи вимагає міждисциплінарного підходу, що включає знання у сфері робототехніки, комп'ютерного зору, обробки зображень, машинного навчання та програмування. Взаємодія цих дисциплін забезпечує можливість створення систем, які можуть не лише виявляти і розпізнавати об'єкти, але й адаптуватися до змінних умов і виконувати завдання з високою точністю.

Підсумовуючи, можна сказати, що розробка системи для виявлення об'єктів є складним, але дуже важливим завданням у контексті створення автономних роботизованих систем. Виконання всіх розглянутих етапів дозволяє створити систему, яка може ефективно виконувати завдання в реальних умовах, забезпечуючи високу точність і надійність. Це є основою для подальшого розвитку і вдосконалення роботизованих систем, здатних працювати в різноманітних середовищах і виконувати складні завдання.

ЗАГАЛЬНІ ВИСНОВКИ

У даній дипломній роботі на тему "Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS" було досліджено та реалізовано систему, як може виявляти об'єкти у тривимірному просторі.

Перший розділ роботи присвячений огляду існуючих рішень у сфері комп'ютерного зору та робототехніки. Були розглянуті основні поняття, а також застосування комп'ютерного зору в різних аспектах робототехніки, таких як робочі станції, навігація, взаємодія з об'єктами та опис середовища ROS.

Другий розділ був зосереджений на архітектурі системи та аналізі технологій, необхідних для її розробки. Описано переваги використання операційної системи Ubuntu для проекту, а також обрані мови програмування Python та C++ для роботи з ROS. Була детально розглянута конфігурація та управління вузлами ROS, взаємодія між ними через topics і services, а також використання parameter server для управління конфігураціями системи. Крім того, розглянуто роль Gazebo у симуляції та детекції об'єктів, а також інтеграцію бібліотек з ROS для обробки зображень.

Третій розділ присвячений безпосередній розробці системи для детекції об'єктів. Було створено на основі існуючих пакетів симуляційне середовище в 3D просторі з використанням Gazebo, що дозволило моделювати реальні умови для роботи роботизованих систем. Особлива увага була приділена процесу виявлення об'єктів, що включає сприйняття об'єктів за допомогою камер глибини та RGB, а також створенню пакету main_capturing для детекції об'єктів. Детально розглянуто процес запуску вузлів виявлення об'єктів та візуалізації результатів у Rviz2.

Реалізація системи в умовах симуляції показала, що такі технології можуть бути успішно застосовані у реальних сценаріях, де необхідна точна і швидка обробка тривимірних даних. Проект також підкреслив важливість міждисциплінарного

підходу, що включає знання в сфері робототехніки, комп'ютерного зору, програмування та машинного навчання.

ВИКОРИСТАНА ЛІТЕРАТУРА

1. Inside Amazon's robot-run warehouse - https://www.drapersonline.com/insight/analysis/inside-amazons-robot-run-warehouse
2. Deploying a Scalable Object Detection Pipeline: The Inferencing Process - https://developer.nvidia.com/blog/deploying-a-scalable-object-detection-pipeline-the-inferencing-process-part-2/
3. ROS1 vs ROS2 comparison - https://www.researchgate.net/figure/ROS1-ROS2-architecture-for-DDS-approach-to-ROS-We-clarify-the-performance-of-the-data_fig1_309128426
4. Object Tracking with Opencv and Python - https://medium.com/@MrBam44/object-tracking-with-opencv-and-python-7db8b233fab6
5. 3D point cloud library - https://ieee-dataport.org/documents/3d-point-cloud-library
6. Robot Operating System (ROS) for Absolute Beginners: Robotics Programming Made Easy by Lentin Joseph - http://wiki.iranros.com/wp-content/uploads/2019/10/Lentin-Joseph-Robot-Operating-SystemROSfor-Absolute-Beginners_IRANROS.COM2018.pdf
7. Autopilot and Full Self-Driving Capability | Tesla Support - https://www.tesla.com/support/autopilot
8. Walmart використовує роботів в магазинах - https://rau.ua/novyni/walmart-robotov-v-magazinah/
9. Supercharge complex – InnerEye - https://innereye.ai/
10. Discover Spot Boston Dynamics | Robotic solutions for industry - https://www.plainconcepts.com/spot-boston-dynamics-robot/
11. Programming Robots with ROS: A Practical Introduction to the Robot Operating System by Morgan Quigley, Brian Gerkey, and William D. Smart
12. ROS Robotics By Example by Carol Fairchild and Dr. Thomas L. Harman - https://sceweb.sce.uhcl.edu/harman/CENG5437_MobileRobots/Webitems2020/ROS_ROBOTICS_BY_EXAMPLE_SECOND_EDITION.pdf

13. Learning ROS for Robotics Programming by Enrique Fernández and Luis Sánchez
14. Mastering ROS for Robotics Programming by Lentin Joseph
15. Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython by Wes McKinney
16. Effective Python: 59 Specific Ways to Write Better Python by Brett Slatkin
17. C++ Primer by Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo - https://zhjwpku.com/assets/pdf/books/C++.Primer.5th.Edition_2013.pdf
18. The C++ Programming Language by Bjarne Stroustrup
19. Point Cloud Library (PCL) Tutorial by Alexandru-Eugen Ichim
20. Practical Computer Vision with SimpleCV: The Simple Way to Make Technology See by Kurt Demaagd, Anthony Oliver, Nathan Oostendorp, and Katherine Scott - https://books.google.com.ua/books/about/Practical_Computer_Vision_with_SimpleCV.html?id=4st9l6QlkpUC&redir_esc=y
21. Hands-On ROS for Robotics Programming: Learn the Fundamentals of ROS to Build Powerful and Scalable Robotic Applications by Ramkumar Gandhinathan
22. Computer Vision: Algorithms and Applications by Richard Szeliski
23. Robotics, Vision and Control: Fundamental Algorithms in MATLAB by Peter Corke
24. Artificial Intelligence: A Modern Approach by Stuart Russell and Peter Norvig
25. ROS Developers Open Class n.140: Object Detection with ROS2 - https://app.theconstruct.ai/rosjects/796450/
26. Learning Robotics using Python by Lentin Joseph
27. Simple grasping - https://github.com/mikeferguson/c/tree/ros2/src
28. Computer Vision: Models, Learning, and Inference by Simon J.D. Prince

ДОДАТОК 1

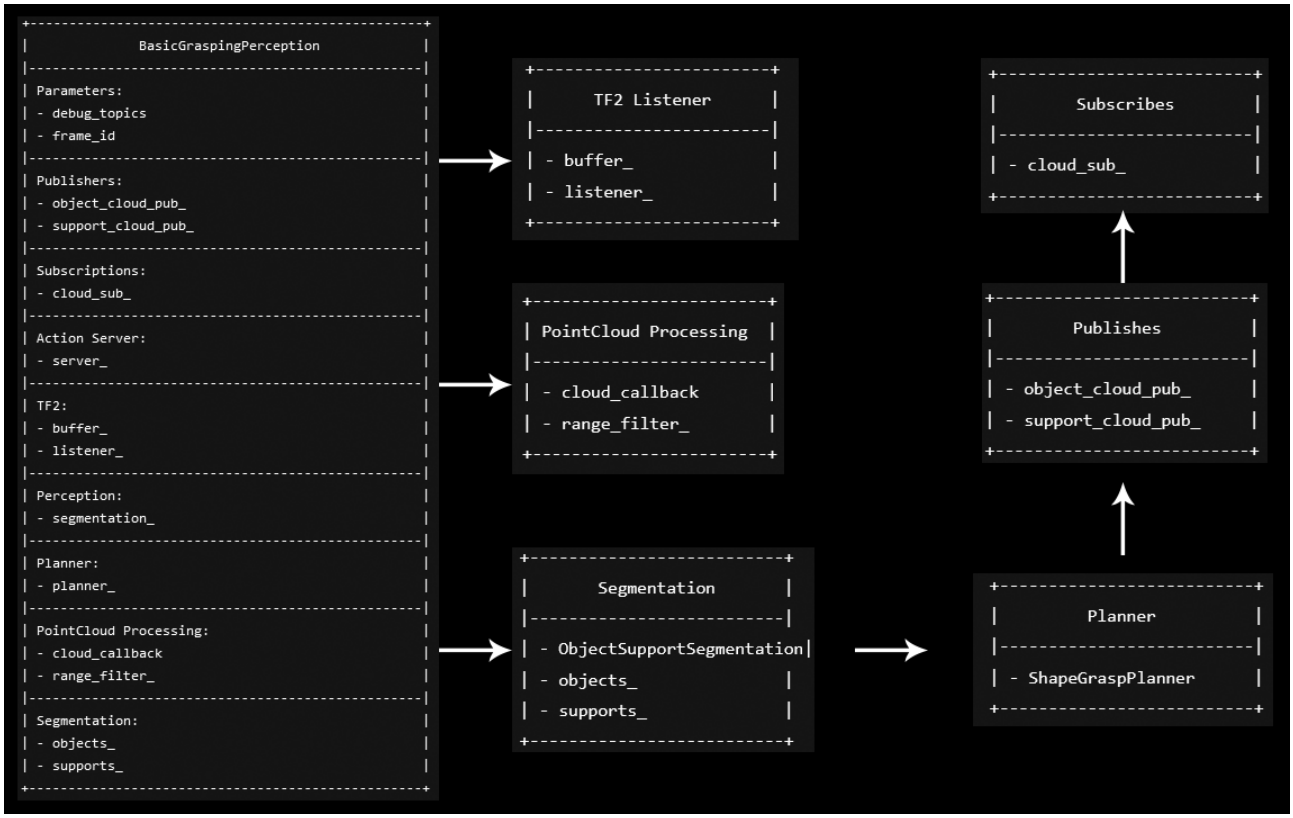
Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS

Структурна схема

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2024



					ІАЛЦ.467200.004 Д1		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Османов Е. Я.			*Модуль комп'ютерного зору для роботизованих систем із використанням ROS* (структурна схема)	Літ.	Аркуш 1 / Аркушів
Перевірив		Таран В.І					
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05	
Н. Контр.		Волокита А. М.					
Затвердив							

ДОДАТОК 2

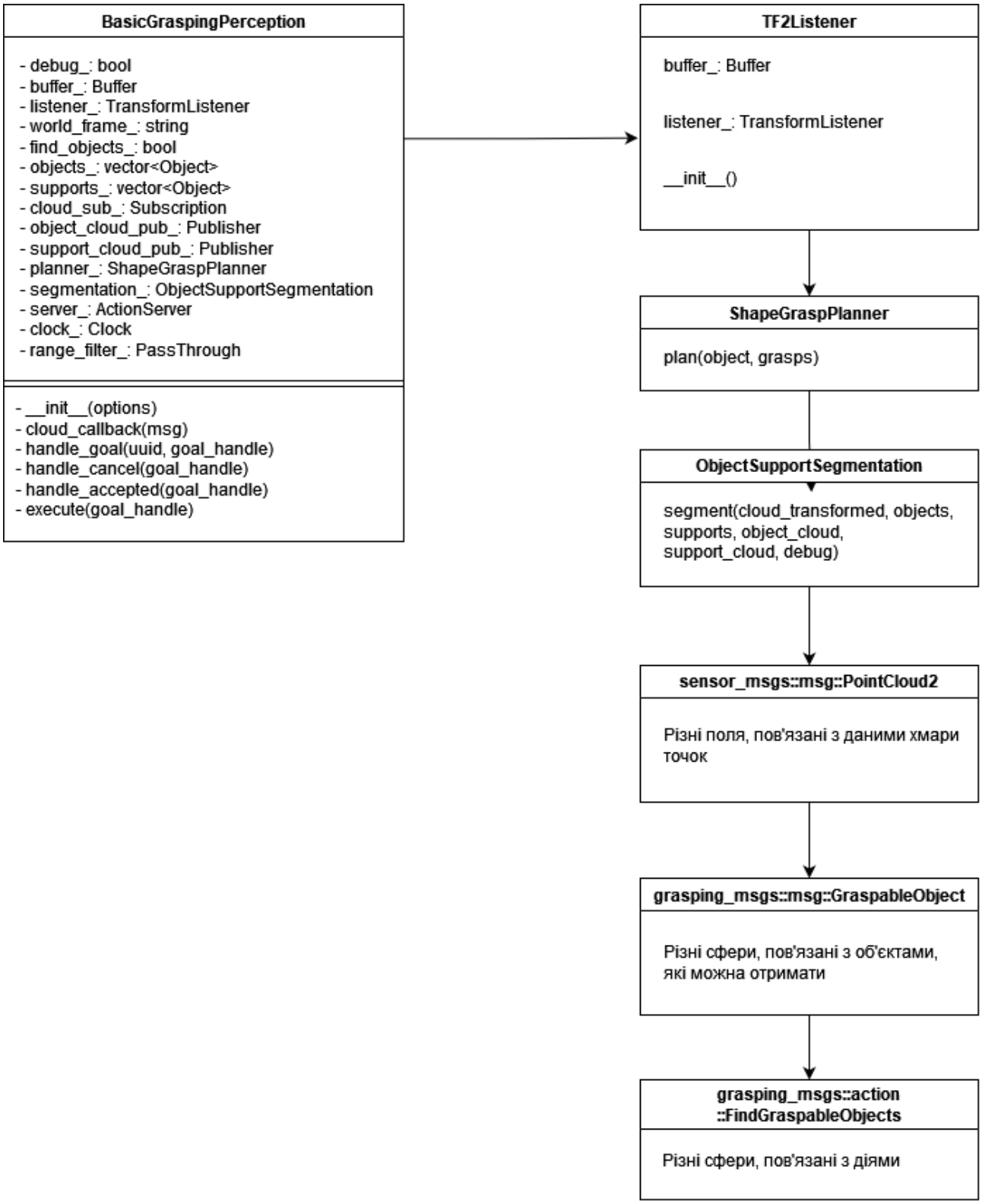
Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS

Функціональна схема

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2024



					ІАЛЦ.467200.005 Д2		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Османов Е. Я.			*Модуль комп'ютерного зору для роботизованих систем із використанням ROS* (функціональна схема)	Літ.	Аркуш 1 / Аркушів
Перевірив		Таран В.І					КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05
Реценз.							
Н. Контр.		Волокита А. М.					
Затвердив							

ДОДАТОК З

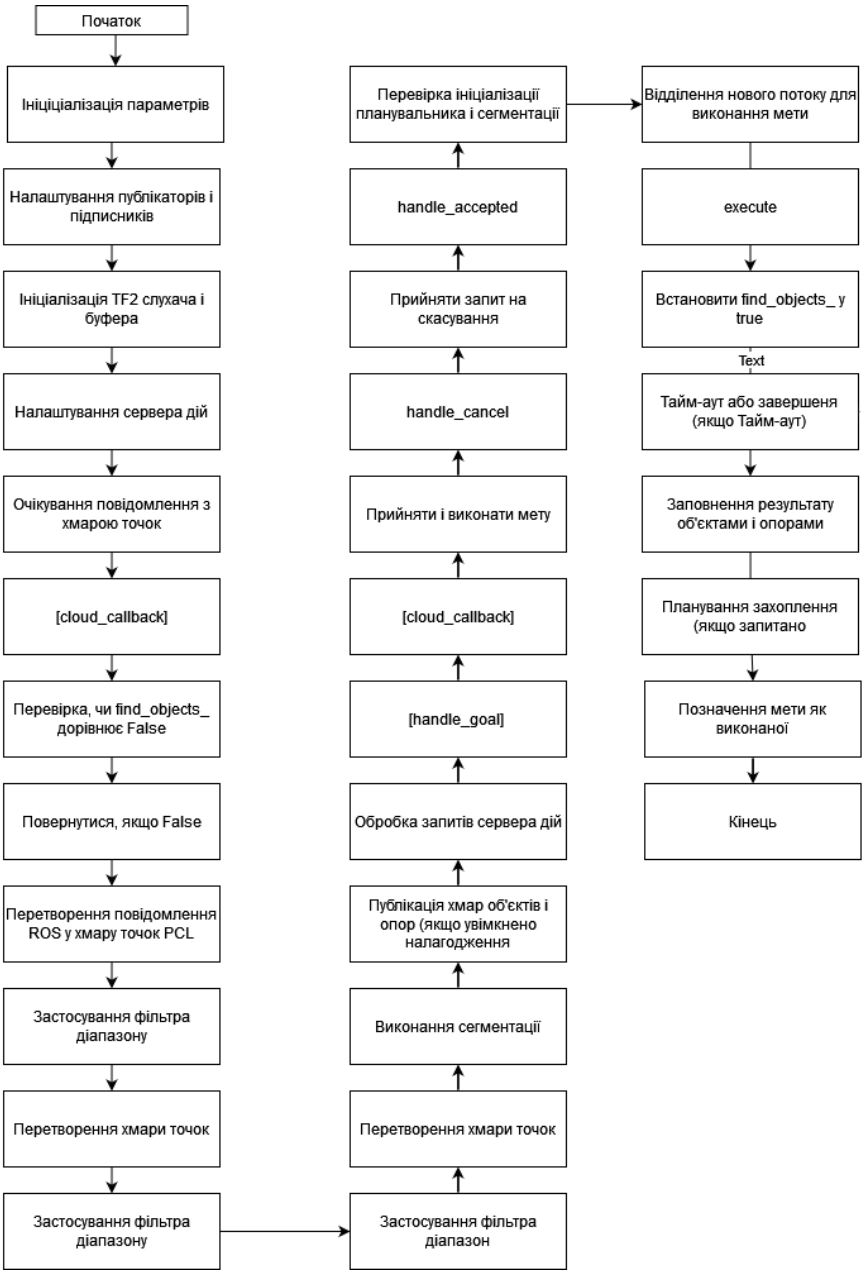
Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 ДЗ

Аркушів 1

Київ 2024



					ІАЛЦ.467200.006 Д3		
Зм.	Арк.	№ докум.	Підпис	Дата			
Розробив		Османов Е. Я.			*Модуль комп'ютерного зору для роботизованих систем із використанням ROS* Алгоритм дій програмного забезпеченя (принципова схема)	Літ.	Аркуш 1 / Аркушів
Перевірив		Таран В.І					
Реценз.						КПІ ім. Ігоря Сікорського, ФІОТ, ІП-05	
Н. Контр.		Волокита А. М.					
Затвердив							

ДОДАТОК 4

Модуль комп'ютерного зору для автономних роботизованих систем із використанням ROS

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 18

Київ 2024

capturing_sensing.cpp

```
#include <rclcpp/rclcpp.hpp>

#include <rclcpp_action/rclcpp_action.hpp>

#include <sensor_msgs/msg/image.hpp>

#include <image_transport/image_transport.hpp>

#include <cv_bridge/cv_bridge.h>

#include <opencv2/opencv.hpp>

#include <capturing_msgs/action/find_objects.hpp>


class CapturingSensing : public rclcpp::Node

{

public:

    using FindObjects = your_custom_action::action::FindObjects;

    using GoalHandleFindObjects = rclcpp_action::ServerGoalHandle<FindObjects>;


    CapturingSensing()

        : Node("capturing_sensing")

    {

        this->action_server_ = rclcpp_action::create_server<FindObjects>(

            this,

            "/find_objects",

            std::bind(&CapturingSensing::handle_goal, this, std::placeholders::_1, std::placeholders::_2),

            std::bind(&CapturingSensing::handle_cancel, this, std::placeholders::_1),

            std::bind(&CapturingSensing::handle_accepted, this, std::placeholders::_1));
```

```
        this->rgb_subscription_ = image_transport::create_subscription(
            this,
            "/camera/image_raw",
            std::bind(&CapturingSensing::image_callback, this, std::placeholders::_1),
            "raw");
    }

private:
    rclcpp_action::Server<FindObjects>::SharedPtr action_server_;
    image_transport::Subscriber rgb_subscription_;
    cv::Mat current_image_;

    rclcpp_action::GoalResponse handle_goal(
        const rclcpp_action::GoalUUID &uuid,
        std::shared_ptr<const FindObjects::Goal> goal)
    {
        RCLCPP_INFO(this->get_logger(), "Received goal request with order %d", goal->order);
        (void)uuid;
        return rclcpp_action::GoalResponse::ACCEPT_AND_EXECUTE;
    }

    rclcpp_action::CancelResponse handle_cancel(
```

capturing_sensing.cpp

```
    const std::shared_ptr<GoalHandleFindObjects> goal_handle)

{

    RCLCPP_INFO(this->get_logger(), "Received request to cancel goal");

    (void)goal_handle;

    return rclcpp_action::CancelResponse::ACCEPT;

}



void handle_accepted(const std::shared_ptr<GoalHandleFindObjects> goal_handle)

{

    std::thread{std::bind(&CapturingSensing::execute, this, std::placeholders::_1), goal_handle}.detach();

}



void execute(const std::shared_ptr<GoalHandleFindObjects> goal_handle)

{

    RCLCPP_INFO(this->get_logger(), "Executing goal");



    const auto goal = goal_handle->get_goal();

    auto result = std::make_shared<FindObjects::Result>();



    if (!current_image_.empty())

    {

        cv::Mat gray_image;

        cv::cvtColor(current_image_, gray_image, cv::COLOR_BGR2GRAY);
```

```
        cv::Mat filtered_image;
        cv::medianBlur(gray_image, filtered_image, 5);

        cv::Mat binary_image;
        cv::threshold(filtered_image, binary_image, 100, 255, cv::THRESH_BINARY);

        result->objects_detected = true;
    }
    else
    {
        result->objects_detected = false;
    }

    goal_handle->succeed(result);
    RCLCPP_INFO(this->get_logger(), "Goal succeeded");
}

void image_callback(const sensor_msgs::msg::Image::ConstSharedPtr &msg)
{
    try
    {
        current_image_ = cv_bridge::toCvShare(msg, "bgr8")->image;
```

capturing_sensing.cpp

```
        }

        catch (cv_bridge::Exception &e)

        {

            RCLCPP_ERROR(this->get_logger(), "cv_bridge exception: %s", e.what());

        }

    }

};



int main(int argc, char **argv)

{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<CapturingSensing>();

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;

}
```

capturing_segmentation.cpp

```
#include <rclcpp/rclcpp.hpp>

#include <sensor_msgs/msg/point_cloud2.hpp>

#include <pcl_conversions/pcl_conversions.h>

#include <pcl/point_cloud.h>

#include <pcl/point_types.h>

#include <pcl/filters/passthrough.h>

#include <pcl/segmentation/sac_segmentation.h>

#include <pcl/segmentation/extract_clusters.h>

#include <pcl_ros/transforms.hpp>

#include <visualization_msgs/msg/marker_array.hpp>


class CapturingSegmentation : public rclcpp::Node

{

public:

   CapturingSegmentation()

      : Node("capturing_segmentation")

   {

      this->pointcloud_subscription_ = this->create_subscription<sensor_msgs::msg::PointCloud2>(

         "/camera/depth/points",

         10,

         std::bind(&CapturingSegmentation::pointcloud_callback, this, std::placeholders::_1));


      this->object_publisher_ = this->create_publisher<visualization_msgs::msg::MarkerArray>(
```

```
            "/object_cloud",
            10);

        this->support_publisher_ = this->create_publisher<visualization_msgs::msg::MarkerArray>(
            "/support_cloud",
            10);
    }

private:
    rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr pointcloud_subscription_;
    rclcpp::Publisher<visualization_msgs::msg::MarkerArray>::SharedPtr object_publisher_;
    rclcpp::Publisher<visualization_msgs::msg::MarkerArray>::SharedPtr support_publisher_;

    void pointcloud_callback(const sensor_msgs::msg::PointCloud2::SharedPtr msg)
    {
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
        pcl::fromROSMsg(*msg, *cloud);

        pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_cloud(new pcl::PointCloud<pcl::PointXYZ>);
        pcl::PassThrough<pcl::PointXYZ> pass;
        pass.setInputCloud(cloud);
        pass.setFilterFieldName("z");
        pass.setFilterLimits(0.0, 1.5);
```

```
    pass.filter(*filtered_cloud);


    pcl::SACSegmentation<pcl::PointXYZ> seg;
    pcl::PointIndices::Ptr inliers(new pcl::PointIndices);
    pcl::ModelCoefficients::Ptr coefficients(new pcl::ModelCoefficients);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane(new pcl::PointCloud<pcl::PointXYZ>());
    seg.setOptimizeCoefficients(true);
    seg.setModelType(pcl::SACMODEL_PLANE);
    seg.setMethodType(pcl::SAC_RANSAC);
    seg.setMaxIterations(100);
    seg.setDistanceThreshold(0.02);
    seg.setInputCloud(filtered_cloud);
    seg.segment(*inliers, *coefficients);


    if (inliers->indices.size() == 0)
    {
       RCLCPP_WARN(this->get_logger(), "Could not estimate a planar model for the given dataset.");
       return;
    }


    pcl::ExtractIndices<pcl::PointXYZ> extract;
    extract.setInputCloud(filtered_cloud);
    extract.setIndices(inliers);
```

```
        extract.setNegative(false);

        extract.filter(*cloud_plane);



        visualization_msgs::msg::MarkerArray support_markers;

        create_markers(cloud_plane, "support_plane", support_markers);



        this->support_publisher_->publish(support_markers);



        extract.setNegative(true);

        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_objects(new pcl::PointCloud<pcl::PointXYZ>());

        extract.filter(*cloud_objects);



        pcl::search::KdTree<pcl::PointXYZ>::Ptr tree(new pcl::search::KdTree<pcl::PointXYZ>);

        tree->setInputCloud(cloud_objects);



        std::vector<pcl::PointIndices> cluster_indices;

        pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;

        ec.setClusterTolerance(0.02);

        ec.setMinClusterSize(100);

        ec.setMaxClusterSize(25000);

        ec.setSearchMethod(tree);

        ec.setInputCloud(cloud_objects);

        ec.extract(cluster_indices);
```

```
        visualization_msgs::msg::MarkerArray object_markers;
        int id = 0;
        for (const auto &indices : cluster_indices)
        {
            pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster(new pcl::PointCloud<pcl::PointXYZ>);
            for (const auto &idx : indices.indices)
                cloud_cluster->push_back((*cloud_objects)[idx]);
            cloud_cluster->width = cloud_cluster->size();
            cloud_cluster->height = 1;
            cloud_cluster->is_dense = true;

            create_markers(cloud_cluster, "object_cluster", object_markers, id++);
        }

        this->object_publisher_->publish(object_markers);
    }

    void create_markers(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud, const std::string &ns,
visualization_msgs::msg::MarkerArray &markers, int id_start = 0)
    {
        visualization_msgs::msg::Marker marker;
        marker.header.frame_id = "camera_frame";
```

```
        marker.header.stamp = this->now();

        marker.ns = ns;

        marker.id = id_start;

        marker.type = visualization_msgs::msg::Marker::POINTS;

        marker.action = visualization_msgs::msg::Marker::ADD;

        marker.pose.orientation.w = 1.0;



        marker.scale.x = 0.01;

        marker.scale.y = 0.01;

        marker.color.r = 1.0;

        marker.color.g = 0.0;

        marker.color.b = 0.0;

        marker.color.a = 1.0;


        for (const auto &point : cloud->points)

        {

           geometry_msgs::msg::Point p;

           p.x = point.x;

           p.y = point.y;

           p.z = point.z;

           marker.points.push_back(p);

        }
```

capturing_segmentation.cpp

```
        markers.markers.push_back(marker);

    }

};



int main(int argc, char **argv)

{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<CapturingSegmentation>();

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;

}
```

shape_extraction.cpp

```
#include <rclcpp/rclcpp.hpp>

#include <sensor_msgs/msg/point_cloud2.hpp>

#include <pcl_conversions/pcl_conversions.h>

#include <pcl/point_cloud.h>

#include <pcl/point_types.h>

#include <pcl/filters/passthrough.h>

#include <pcl/segmentation/sac_segmentation.h>

#include <pcl/segmentation/extract_clusters.h>

#include <pcl_ros/transforms.hpp>

#include <visualization_msgs/msg/marker_array.hpp>

#include <pcl/common/common.h>


class ShapeExtraction : public rclcpp::Node

{

public:

   ShapeExtraction()

      : Node("shape_extraction")

   {

      this->pointcloud_subscription_ = this->create_subscription<sensor_msgs::msg::PointCloud2>(

         "/camera/depth/points",

         10,

         std::bind(&ShapeExtraction::pointcloud_callback, this, std::placeholders::_1));
```

shape_extraction.cpp

```
        this->shape_publisher_ = this->create_publisher<visualization_msgs::msg::MarkerArray>(
            "/detected_shapes",
            10);
    }

private:
    rclcpp::Subscription<sensor_msgs::msg::PointCloud2>::SharedPtr pointcloud_subscription_;
    rclcpp::Publisher<visualization_msgs::msg::MarkerArray>::SharedPtr shape_publisher_;

    void pointcloud_callback(const sensor_msgs::msg::PointCloud2::SharedPtr msg)
    {
        pcl::PointCloud<pcl::PointXYZ>::Ptr cloud(new pcl::PointCloud<pcl::PointXYZ>);
        pcl::fromROSMsg(*msg, *cloud);

        // Filter the point cloud
        pcl::PointCloud<pcl::PointXYZ>::Ptr filtered_cloud(new pcl::PointCloud<pcl::PointXYZ>);
        pcl::PassThrough<pcl::PointXYZ> pass;
        pass.setInputCloud(cloud);
        pass.setFilterFieldName("z");
        pass.setFilterLimits(0.0, 1.5);
        pass.filter(*filtered_cloud);

        // Detect and publish shapes
```

```
        visualization_msgs::msg::MarkerArray shape_markers;

        detect_and_publish_shapes(filtered_cloud, shape_markers);

        this->shape_publisher_->publish(shape_markers);

    }


    void detect_and_publish_shapes(pcl::PointCloud<pcl::PointXYZ>::Ptr cloud,
visualization_msgs::msg::MarkerArray &markers)

    {

        pcl::SACSegmentation<pcl::PointXYZ> seg;

        pcl::PointIndices::Ptr inliers(new pcl::PointIndices);

        pcl::ModelCoefficients::Ptr coefficients(new pcl::ModelCoefficients);


        // Plane segmentation

        seg.setOptimizeCoefficients(true);

        seg.setModelType(pcl::SACMODEL_PLANE);

        seg.setMethodType(pcl::SAC_RANSAC);

        seg.setMaxIterations(100);

        seg.setDistanceThreshold(0.02);

        seg.setInputCloud(cloud);

        seg.segment(*inliers, *coefficients);


        if (inliers->indices.size() != 0)

        {

            create_plane_marker(coefficients, markers, "plane", markers.markers.size());
```

```
    }


    // Cylinder segmentation
    seg.setModelType(pcl::SACMODEL_CYLINDER);
    seg.setMethodType(pcl::SAC_RANSAC);
    seg.setMaxIterations(10000);
    seg.setDistanceThreshold(0.05);
    seg.setRadiusLimits(0, 0.1);
    seg.setInputCloud(cloud);
    seg.segment(*inliers, *coefficients);

    if (inliers->indices.size() != 0)
    {
       create_cylinder_marker(coefficients, markers, "cylinder", markers.markers.size());
    }

    // Sphere segmentation
    seg.setModelType(pcl::SACMODEL_SPHERE);
    seg.setMethodType(pcl::SAC_RANSAC);
    seg.setMaxIterations(10000);
    seg.setDistanceThreshold(0.05);
    seg.setRadiusLimits(0, 0.2);
    seg.setInputCloud(cloud);
```

```
        seg.segment(*inliers, *coefficients);



        if (inliers->indices.size() != 0)

        {

            create_sphere_marker(coefficients, markers, "sphere", markers.markers.size());

        }

    }



    void create_plane_marker(pcl::ModelCoefficients::Ptr coefficients,
visualization_msgs::msg::MarkerArray &markers, const std::string &ns, int id)

    {

        visualization_msgs::msg::Marker marker;

        marker.header.frame_id = "camera_frame";

        marker.header.stamp = this->now();

        marker.ns = ns;

        marker.id = id;

        marker.type = visualization_msgs::msg::Marker::CUBE;

        marker.action = visualization_msgs::msg::Marker::ADD;

        marker.pose.position.x = 0;

        marker.pose.position.y = 0;

        marker.pose.position.z = 0;

        marker.pose.orientation.w = 1.0;

        marker.scale.x = 1.0;

        marker.scale.y = 1.0;
```

```
        marker.scale.z = 0.01;

        marker.color.r = 0.0;

        marker.color.g = 1.0;

        marker.color.b = 0.0;

        marker.color.a = 0.5;



        markers.markers.push_back(marker);

    }



    void create_cylinder_marker(pcl::ModelCoefficients::Ptr coefficients,
visualization_msgs::msg::MarkerArray &markers, const std::string &ns, int id)

    {

        visualization_msgs::msg::Marker marker;

        marker.header.frame_id = "camera_frame";

        marker.header.stamp = this->now();

        marker.ns = ns;

        marker.id = id;

        marker.type = visualization_msgs::msg::Marker::CYLINDER;

        marker.action = visualization_msgs::msg::Marker::ADD;

        marker.pose.position.x = coefficients->values[0];

        marker.pose.position.y = coefficients->values[1];

        marker.pose.position.z = coefficients->values[2];

        marker.pose.orientation.x = coefficients->values[3];

        marker.pose.orientation.y = coefficients->values[4];
```

```
        marker.pose.orientation.z = coefficients->values[5];

        marker.pose.orientation.w = coefficients->values[6];

        marker.scale.x = coefficients->values[6] * 2;

        marker.scale.y = coefficients->values[6] * 2;

        marker.scale.z = coefficients->values[7];

        marker.color.r = 0.0;

        marker.color.g = 0.0;

        marker.color.b = 1.0;

        marker.color.a = 0.5;



        markers.markers.push_back(marker);

    }



    void create_sphere_marker(pcl::ModelCoefficients::Ptr coefficients,
visualization_msgs::msg::MarkerArray &markers, const std::string &ns, int id)

    {

        visualization_msgs::msg::Marker marker;

        marker.header.frame_id = "camera_frame";

        marker.header.stamp = this->now();

        marker.ns = ns;

        marker.id = id;

        marker.type = visualization_msgs::msg::Marker::SPHERE;

        marker.action = visualization_msgs::msg::Marker::ADD;

        marker.pose.position.x = coefficients->values[0];
```

```
        marker.pose.position.y = coefficients->values[1];

        marker.pose.position.z = coefficients->values[2];

        marker.pose.orientation.w = 1.0;

        marker.scale.x = coefficients->values[3] * 2;

        marker.scale.y = coefficients->values[3] * 2;

        marker.scale.z = coefficients->values[3] * 2;

        marker.color.r = 1.0;

        marker.color.g = 0.0;

        marker.color.b = 0.0;

        marker.color.a = 0.5;



        markers.markers.push_back(marker);

    }

};



int main(int argc, char **argv)

{

    rclcpp::init(argc, argv);

    auto node = std::make_shared<ShapeExtraction>();

    rclcpp::spin(node);

    rclcpp::shutdown();

    return 0;

}
```