НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

«На правах рукопису»

УДК 004.8.032.26:004.932](043.3)

До захисту допущено:
В. о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«___» _____ 2025 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-науковою програмою «Комп'ютерні науки»

зі спеціальності 122 «Комп'ютерні науки»

на тему: «Побудова деталізованих 3D карт на основі моделей YOLO»

Виконав:
студент ІІ курсу, групи КІ-41мп
Мірошниченко Михайло Андрійович _____

Науковий керівник:
професор кафедри штучного інтелекту, д.т.н., професор
Данилов Володимир Яковлевич _____

Рецензент:
професор кафедри математичних методів системного аналізу, д.т.н.,
доцент Кузнєцова Наталія Володимирівна

_____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2025 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Навчально-науковий інститут прикладного системного аналізу
Кафедра штучного інтелекту

Рівень вищої освіти – другий (магістерський)
Спеціальність – 122 «Комп'ютерні науки»
Освітньо-наукова програма «Комп'ютерні науки»

ЗАТВЕРДЖУЮ
В.о. завідувачки кафедри
_____ Ірина ДЖИГИРЕЙ
«29» серпня 2025 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Мірошниченку Михайлу Андрійовичу

1. Тема дисертації: «Побудова деталізованих 3D карт на основі моделей YOLO», науковий керівник дисертації Данилов Володимир Яковлевич, професор кафедри штучного інтелекту, д.т.н., затверджені наказом по ННІПСА від «06» листопада 2025 р. № 4837-с.

2. Термін подання студентом дисертації: 07.12.2025.

3. Об'єкт дослідження: система візуальної навігації та побудови тривимірних карт у динамічних середовищах.

4. Вихідні дані: RGB-D послідовності (зокрема TUM RGB-D), відеозаписи динамічних сцен, попередньо навчені моделі YOLOv8, трекер ByteTrack, алгоритми TSDF-ф'южну.

5. Перелік завдань, які потрібно розробити: провести огляд сучасних підходів до Visual SLAM у динамічних середовищах, проаналізувати архітектуру YOLOv8 та можливості інтеграції детектора динамічних об'єктів у SLAM-пайплайн, розробити архітектуру нейромережевої системи для робастного 3D
картування з використанням YOLOv8, ByteTrack та TSDF-ф'южну, реалізувати програмний прототип системи та провести експерименти на статичних і динамічних послідовностях, виконати порівняльний аналіз точності траєкторії та якості карти
(ATE RMSE, візуальна оцінка карти) відносно базових SLAM-методів.

6. Перелік графічного (ілюстративного) матеріалу: схеми архітектури SLAM-пайплайну, приклади карт у статичних і динамічних сценах, графіки

ATE RMSE, ілюстрації роботи детектора YOLOv8 та трекера ByteTrack.

7. Дата видачі завдання: «29» серпня 2025 року.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання етапів магістерської роботи	Примітка
1.	Огляд літератури за тематикою роботи та її опрацювання	10.09.2025	Виконано
2.	Написання першого розділу магістерської дисертації	01.10.2025	Виконано
3.	Написання другого розділу магістерської дисертації	15.10.2025	Виконано
4.	Розробка програмного продукту (SLAM-пайплайн, інтеграція YOLOv8, ByteTrack, TSDF)	20.11.2025	Виконано
5.	Написання третього розділу магістерської дисертації	26.11.2025	Виконано
6.	Написання четвертого розділу магістерської дисертації	28.11.2025	Виконано
7.	Висновки та підготовка презентації для доповіді	29.11.2025	Виконано
8.	Оформлення пояснювальної записки	06.12.2025	Виконано

Студент Михайло МІРОШНИЧЕНКО

Науковий керівник Валерій ДАНИЛОВ

РЕФЕРАТ

Магістерська дисертація: 131 ст., 4 рис., 26 табл., 21 посилань, додаток.

VISUAL SLAM, НЕЙРОМЕРЕЖІ, КОМП'ЮТЕРНИЙ ЗІР, YOLOV8, BYTETRACK, TSDF-Ф'ЮЖН, РОБАСТНЕ 3D КАРТУВАННЯ, ДИНАМІЧНІ СЕРЕДОВИЩА.

Об'єкт дослідження – система побудови тривимірних карт на основі RGB-D даних у динамічних середовищах. Предмет дослідження – методи комп'ютерного зору та глибинного навчання, що інтегруються у SLAM-пайплайн для підвищення робастності до рухомих об'єктів.

Метою роботи є розробка та аналіз нейромережевої системи Visual SLAM, яка забезпечує точне тривимірне картування в присутності динамічних об'єктів завдяки використанню YOLOv8, ByteTrack та TSDF-ф'южну.

Розглянуто сучасні підходи до SLAM у динамічних середовищах, проаналізовано переваги та обмеження методів семантичного маскування та геометричної фільтрації. Описано архітектуру запропонованої системи, що поєднує детекцію об'єктів YOLOv8, трекінг ByteTrack, маскування динамічних областей та щільне картування на основі TSDF.

Реалізовано програмний прототип, проведено експерименти на статичних та динамічних наборах даних, зокрема на послідовностях TUM RGB-D. Виконано порівняльний аналіз точності траєкторії (ATE RMSE) та візуальної якості карти у порівнянні з базовими SLAM-системами без семантичної фільтрації.

Результатом дослідження є побудова та програмна реалізація нейромережевої системи Visual SLAM для динамічних середовищ; експериментальне підтвердження зменшення впливу рухомих об'єктів на якість тривимірної карти та траєкторії камери; отримання високоточних результатів на статичних послідовностях (ATE RMSE ≈ 0.0229 м) та покращення стійкості SLAM у динамічних сценах; формування рекомендацій щодо інтеграції детекторів об'єктів і трекерів у SLAM-пайплайни реального часу.

ABSTRACT

Master’s Thesis: 131 pages, 4 figures, 26 tables, 21 references, appendix.

VISUAL SLAM, NEURAL NETWORKS, COMPUTER VISION, YOLOV8, BYTETRACK, TSDF FUSION, ROBUST 3D MAPPING, DYNAMIC ENVIRONMENTS.

The object of the research is an RGB-D based 3D mapping system operating in dynamic environments.

The subject of the research is computer vision and deep learning methods integrated into a SLAM pipeline in order to increase robustness to moving objects.

The purpose of the thesis is to develop and analyze a neural Visual SLAM system that performs accurate 3D mapping in the presence of dynamic objects by using YOLOv8 detection, ByteTrack tracking and TSDF fusion.

The thesis reviews modern approaches to SLAM in dynamic environments and analyzes the advantages and limitations of semantic masking and purely geometric filtering methods. It describes the architecture of the proposed system that combines YOLOv8-based object detection, ByteTrack tracking, motion masking, and dense 3D reconstruction with TSDF.

A software prototype of the system is implemented and evaluated on static and dynamic datasets, including TUM RGB-D sequences. A comparative analysis of camera trajectory accuracy (ATE RMSE) and map quality is performed against baseline SLAM systems without semantic filtering.

The results of the study are design and implementation of a neural Visual SLAM system for dynamic environments; experimental confirmation of reduced influence of moving objects on the quality of the 3D map and camera trajectory; achievement of high accuracy on static sequences (ATE RMSE ≈ 0.0229 m) and improved robustness of SLAM in dynamic scenes; formulation of practical recommendations on integrating object detectors and trackers into real-time SLAM pipelines.

ЗМІСТ

ВСТУП

Актуальність[1] теми. В умовах стрімкого розвитку автономних систем, мобільної робототехніки та технологій доповненої (AR) і віртуальної (VR) реальності, критично важливою вимогою є здатність цих систем до високоточного та надійного тривимірного картування (3D–картування) навколишнього середовища в режимі реального часу. Центральним механізмом, що забезпечує цю можливість, є Simultaneous Localization and Mapping (SLAM) – процес, який одночасно оцінює позу сенсора та будує карту невідомої сцени.

Проте, традиційні Візуальні SLAM–системи (V–SLAM), які ґрунтуються виключно на геометричних обмеженнях (наприклад, ORB–SLAM), виходять із фундаментального, але часто нереалістичного, припущення про статичне середовище. Внаслідок цього, у складних, динамічних сценаріях, таких як міський рух або приміщення з великою кількістю рухомих об'єктів (людей, транспорту), ці системи накопичують значну похибку траєкторії, відому як глобальний дрейф (drift). Геометричні робустизатори, такі як RANSAC, не здатні повністю подолати цю проблему, оскільки рухомі об'єкти створюють систематичні та локально узгоджені хибні відповідності.

Актуальність роботи обумовлена необхідністю подолання цієї фундаментальної вразливості шляхом інтеграції передових засобів глибокого навчання, здатних надавати семантичні пріори. Моделі детекції об'єктів у реальному часі, зокрема YOLO (You Only Look Once), забезпечують проактивну фільтрацію динамічних об'єктів, тим самим перетворюючи

[1]Тут і нижче використано такий інструмент штучного інтелекту як чат-бот з генеративним штучним інтелектом ChatGPT, виключно для корегування та редагування тексту, створеного автором цієї дипломної роботи, на основі автоматизованої перевірки граматики, структури та стилю, що відповідає Політиці використання штучного інтелекту для академічної діяльності в КПІ ім. Ігоря Сікорського (протокол №11 Вченої ради КПІ ім. Ігоря Сікорського від 11 грудня 2023 р.).

семантику з допоміжної інформації на фундаментальну умову стійкості 3D–картування. Це дозволяє досягти феноменального зниження Абсолютної Помилки Траєкторії (ATE) (до 98.37% на динамічних бенчмарках), що є критичним для застосування SLAM на вбудованих платформах (NVIDIA Jetson) в реальних умовах.

Метою магістерської роботи є розробка та аналіз архітектурних рішень для інтеграції високоточних 2D семантичних пріорів на базі моделі YOLOv8 у Visual SLAM системи, спрямованих на забезпечення робустної 3D локалізації та картування в динамічних середовищах, включаючи дослідження апаратних оптимізацій для реалізації в режимі реального часу.

Як завдання дослідження було визначено такі компоненти.

1. Провести теоретичний та методологічний аналіз еволюції систем 3D картування, включаючи SfM та V–SLAM, з акцентом на чисельну нестабільність геометричних методів (8–точковий алгоритм, нормалізація, RANSAC).

2. Обґрунтувати парадигмальний зсув від чисто геометричного до семантично–орієнтованого SLAM, спричинений кризою припущення про статичне середовище.

3. Виконати архітектурний аналіз YOLOv8, зокрема, його функції втрат $\left(L_{CIoU}, L_{DFL}\right)$ для встановлення його надійності як геометричного пріора для 3D проекції.

4. Синтезувати та проаналізувати стратегії 2D–to–3D семантичної фузії та їхній вплив на бекенд оптимізації Graph SLAM, включаючи зменшення коваріації оцінки пози.

5. Дослідити ключові апаратні та програмні оптимізації (TensorRT, квантизація INT8, DLA) для забезпечення високочастотної роботи YOLO–SLAM на вбудованих системах.

6. Розробити архітектуру робустного SLAM–конвеєра, що використовує семантичне маскування для фільтрації динамічних ознак.

Об'єктом дослідження є процеси одночасної локалізації та картування

(SLAM) в умовах підвищеної динамічності навколишнього середовища.

Предметом дослідження є методології інтеграції нейромережевих моделей детекції об'єктів (YOLO) у візуальні SLAM–системи та їхній вплив на геометричну стійкість, точність (ATE) та обчислювальну ефективність.

У роботі використано такі методи дослідження, представлені нижче.

1. Теоретичний аналіз та синтез для вивчення еволюції алгоритмів 3D картування та архітектури глибоких нейронних мереж (YOLOv8).

2. Методи комп'ютерного зору та багатокамерної геометрії для аналізу Фундаментальної Матриці, епіполярних обмежень та методів робустизації (RANSAC).

3. Порівняльний та кількісний аналіз емпіричних результатів роботи семантичних SLAM–систем на динамічних бенчмарках (TUM Dynamic) для підтвердження зниження похибки ATE.

4. Методи нелінійної оптимізації для аналізу впливу семантичної фільтрації на бекенд Graph SLAM та коваріацію оцінки пози.

Наукова новизна.

1. Вперше проведено ригористичний аналіз функцій втрат YOLOv8 $\left(L_{CIoU}\text{та}L_{DFL}\right)$ з точки зору їхньої ролі у забезпеченні високонадійного геометричного пріора для 3D реконструкції, підтверджуючи, що ці компоненти мінімізують не лише 2D, але й масштабовану 3D невизначеність.

2. Виконано синтез вимог до апаратної оптимізації (TensorRT, INT8, DLA [8]) як технологічного каталізатора для реалізації семантичних SLAM–систем у реальному часі на вбудованих платформах, де ресурсні обмеження є критичними.

3. Обґрунтовано, що інтеграція YOLO є пре–кондиціонером даних для геометричного бекенда, що призводить до кращої обумовленості Інформаційної Матриці та зменшення коваріації оцінки пози, що є ключовим для боротьби з глобальним дрейфом.

Практичне значення роботи полягає у тому, що отримані результати та синтезована архітектура дають змогу створювати високоробустні та точні

SLAM–системи, які можуть бути безпомилково застосовані в:

- автономній навігації мобільних роботів та дронів у складних, непередбачуваних міських та індустріальних середовищах;
- розробці AR/VR додатків, які вимагають стійкого та точного трекінгу в динамічних сценах;
- оптимізації апаратних конвеєрів для вбудованих пристроїв (NVIDIA Jetson), забезпечуючи високу частоту кадрів для семантичної обробки.

Апробація результатів. Основні положення роботи були представлені та обговорені на наукових семінарах кафедри, а також планується їх публікація у збірниках праць міжнародних та всеукраїнських науково–технічних конференцій.

РОЗДІЛ 1 ОГЛЯД, АНАЛІЗ ТА ПАРАДИГМАЛЬНИЙ ЗСУВ У МЕТОДАХ 3D КАРТУВАННЯ

1.1 Еволюція методологій відновлення 3D простору: від SfM до V–SLAM

Побудова точних тривимірних моделей оточення традиційно базувалася на двох ключових, але принципово різних, методологіях: Structure from Motion (SfM) та Simultaneous Localization and Mapping (SLAM). Розуміння їхніх архітектурних обмежень є необхідним для визначення потреби у впровадженні семантичних пріорів, що надаються моделями YOLO.

Structure from Motion є класичним підходом, розробленим для створення високоточних 3D моделей на основі статичного набору зображень. Ключова функція SfM полягає в одночасній оцінці пози камери та 3D структури сцени, що вимагає вичерпного набору даних, зібраного заздалегідь.

Головна архітектурна особливість SfM – це використання глобальної оптимізації, відомої як Bundle Adjustment (BA), яка мінімізує помилку репроєкції всіх 3D точок та параметрів камери [3, 4]. Незважаючи на високу точність, BA є обчислювально надзвичайно витратним, оскільки вимагає пакетної обробки всього набору даних [3, 4]. Через цю високу обчислювальну складність, SfM залишається офлайн–методом, несумісним з вимогами навігації та картування в режимі реального часу. Ця несумісність стала головною причиною для індустріального та академічного переходу до розробки оперативних SLAM–систем.

Системи SLAM були розроблені для вирішення подвійної проблеми: картування невідомого середовища та одночасна локалізація самого сенсора, що критично важливо для мобільної робототехніки та автономних систем [3]. Архітектура візуального SLAM (V–SLAM) функціонує в режимі реального часу, або онлайн.

Ключові компоненти архітектури V–SLAM включають Local Mapping,

де 2D ознаки, витягнуті з зображень, перетворюються на 3D хмару точок через процес тріангуляції, та Local Optimization, що забезпечує локальну узгодженість [3, 7]. Значний прогрес у V–SLAM було досягнуто завдяки використанню RGB–D камер, які надають інформацію про глибину D безпосередньо для кожного пікселя [3]. Це забезпечує вирішальну перевагу, оскільки усуває необхідність у складній та чутливій до помилок тріангуляції, яка є обов'язковою для монокулярних систем, значно полегшуючи формування точних 3D структур у реальному часі.

1.2 Фундаментальна математична база: геометрія та чисельна стійкість

Ефективність V–SLAM критично залежить від надійності геометричних обмежень, які забезпечують відновлення 3D–координат. Ці обмеження формуються багатокамерною геометрією.

Епіполярна геометрія описує внутрішній геометричний зв'язок між двома зображеннями, отриманими з різних позицій. Це є критично важливим для процесу пошуку відповідних точок (feature matching), оскільки вона обмежує пошук до одновимірного простору – епілінії. Це значно підвищує ефективність та швидкість пошуку відповідностей.

Цей зв'язок формально описується фундаментальною матрицею F. Її ключова властивість полягає у тому, що вона є придатною для використання з некаліброваними камерами, оскільки включає матриці внутрішніх параметрів K та K′. Ключове епіполярне обмеження виражається як:

$$p'^{T} F p = 0, \tag{1.1}$$

де $p = (u, v, 1)^{\top}$ і $p^{'} = (u^{'}, v^{'}, 1)^{\top}$ – відповідні точки у двох кадрах (однорідні координати), $F \in R^{3\times3}$ – фундаментальна матриця (ранг 2).

Рівність означає, що $p^{'}$ лежить на епіполярній прямій $l^{'} = Fp$ (і симетрично $l = F^{\top}p^{'}$ для першого кадру).

Традиційна оцінка Фундаментальної Матриці, що виконується за допомогою 8–точкового алгоритму, часто призводить до значних числових проблем [3]. Матриця, сформована для розв'язання системи лінійних рівнянь, як правило, виявляється погано обумовленою (*ill–conditioned*). Це порушує теоретичні вимоги до сингулярних добутків F і призводить до неточностей у обчисленій матриці пози.

Для подолання цієї чисельної нестабільності, критично важливим етапом є попередня обробка даних, відома як нормалізація координат [3]. Точки транслюються та масштабуються таким чином, щоб забезпечити початкову стійкість геометричних алгоритмів. Якщо нормалізація не виконується, початкова чисельна оцінка може бути сильно зміщена, що призводить до накопичення помилок.

Навіть після нормалізації, набір відповідностей часто містить викиди (*outliers*), спричинені як помилками вимірювання, так і, що важливіше для реальних сценаріїв, рухомими об'єктами [3]. Для відсіювання цих викидів та забезпечення робастності геометричних оцінок використовуються робастні ітераційні методи, як–от RANSAC (Random Sample Consensus) та його сучасні варіанти. Ці методи дозволяють системі оцінити модель (наприклад, позу камери) лише на основі статичних, внутрішньо узгоджених ознак, ігноруючи ті, що не відповідають епіполярному обмеженню.

1.3 Парадигмальний зсув до семантичної стійкості: інтеграція YOLO

Незважаючи на інновації у геометричних методах та робастизації, традиційні V–SLAM системи не могли повністю подолати проблему дрейфу, викликаного динамічними об'єктами.

Традиційні системи V–SLAM, такі як ORB–SLAM2, ґрунтуються на фундаментальному, але часто нереалістичному, припущенні про статичне середовище. У реальних сценаріях, таких як міський дорожній рух (набори даних KITTI) або офісні сцени з людьми, що рухаються (TUM Dynamic), рухомі об'єкти створюють хибні відповідності ознак. Ці помилкові відповідності не можуть бути повністю відфільтровані лише за допомогою геометричних методів (навіть RANSAC), оскільки вони можуть бути численними та локально узгодженими.

Ці хибні дані є домінуючим джерелом помилок локалізації, що призводить до значного накопичення помилки траєкторії, відомого як дрейф (drift). Саме ця фундаментальна вразливість до динамічних об'єктів спричинила необхідність у Семантичному SLAM.

Інтеграція нейронних мереж, що виконують детекцію об'єктів у реальному часі, як–от YOLO (You Only Look Once), призвела до трансформації семантичної інформації з простого доповнення на фундаментальну умову стійкості системи. YOLO виявляє потенційно динамічні регіони у 2D зображенні. Це дозволяє системі проактивно фільтрувати (маскувати) точки ознак, що знаходяться в межах обмежувальних рамок рухомих об'єктів, залишаючи для оцінки пози камери виключно статичні ознаки.

Наприклад, у системі YPR–SLAM використовується YOLOv5 для виявлення потенційно динамічних областей, а потім алгоритм Depth–PROSAC, який, використовуючи глибинну інформацію RGB–D

камери, точно класифікує точки як динамічні або статичні на основі їхніх геометричних обмежень. Цей двоступеневий підхід забезпечує феноменальне підвищення стійкості локалізації.

Емпіричні дані, отримані на динамічних бенчмарках (наприклад, послідовності TUM Dynamic), надають переконливе кількісне підтвердження ефективності семантичної фільтрації. Системи, що інтегрують YOLO, демонструють радикальне зниження Абсолютної Помилки Траєкторії (ATE) порівняно з базовою лінією, представленою чисто геометричними алгоритмами, такими як ORB–SLAM2.

Наприклад, YPR–SLAM досяг зниження ATE до 97.8% порівняно з ORB–SLAM2 у динамічному середовищі, а YKD–SLAM, який використовує YOLOv8 у поєднанні з епіполярними обмеженнями, продемонстрував зниження ATE на 98.37%. Таке драматичне зменшення ATE є головним доказом того, що інтеграція YOLO успішно вирішує проблему глобального дрейфу, яка раніше була нерозв'язною для чисто геометричних систем. Це підтверджує, що семантичне маскування є не додатковим поліпшенням, а фундаментальною вимогою для забезпечення глобальної узгодженості та надійності SLAM–систем у реальних умовах.

Порівняння основних методологій 3D картування, що демонструє цей парадигмальний зсув, представлено у таблиці 1.1, а ключові кількісні результати – у таблиці 1.2.

Таблиця 1.1 – Порівняння ключових методів 3D картування

Характеристика	**Structure from Motion (SfM)**	**Visual SLAM (V–SLAM)**	**Гібридні Методи (Semantic SLAM)**
Режим роботи	Офлайн (Пакетна обробка)	Режим реального часу (Online)	Режим реального часу / Псевдо–реальний час

Кінець табл. 1.1

Характеристика	**Structure from Motion (SfM)**	**Visual SLAM (V–SLAM)**	**Гібридні Методи (Semantic SLAM)**
Основна мета	Створення статичної, точної 3D моделі	Одночасна локалізація та картування	Семантичне розуміння та фільтрація динамічних об'єктів
Ключовий компонент	Bundle Adjustment	Local Optimization, Loop Closure	2D Детекція (YOLO) та 3D Фільтрація
Стійкість до Динаміки	Низька (Статичне припущення)	Низька (без семантики), Висока (з семантикою)	Дуже Висока (Проактивне маскування)
Типові сенсори	RGB	RGB, RGB–D, IMU	RGB–D, Multi–View RGB

Джерело: створено автором на основі [4, 10, 11]

Таблиця 1.2 – Кількісний порівняльний аналіз динамічних SLAM систем (ATE RMSE)

Алгоритм	**Основа**	**Набір Даних**	**Зниження Помилки (vs. ORB–SLAM2/3)**	**Ключовий Механізм Динаміки**
ORB–SLAM2 (Baseline)	Геометричний SLAM	TUM Dynamic	N/A (Базова лінія)	Припущення про статичне середовище
YPR–SLAM	YOLOv5 + Depth–PROSAC	TUM Dynamic	До 97.8%	Семантична фільтрація + Геометричний PROSAC
YKD–SLAM	YOLOv8 + Epipolar Geo.	TUM Dynamic	До 98.37%	YOLOv8 Детекція + Епіполярні Обмеження

Кінець табл. 1.2

Алгоритм	**Основа**	**Набір Даних**	**Зниження Помилки (vs. ORB–SLAM2/3)**	**Ключовий Механізм Динаміки**
ADD–SLAM	3D Gaussian Splatting	RGB–D Dynamic	Category–Agnostic	Аналіз геометричної консистентності сцени

Джерело: створено автором на основі [4, 6,10, 17]

Висновки до розділу 1

На підставі огляду еволюції 3D-картування – – від класичних геометричних методів до інтеграції глибинного навчання – можна стверджувати таке. Перехід від SfM до V-SLAM був зумовлений потребою роботи в реальному часі: глобальний Bundle Adjustment у SfM є надто ресурсоємним і пакетним, тоді як V-SLAM, перейшовши до локальної оптимізації, виграв у швидкості, але втратив у глобальній точності, що сприяє накопиченню дрейфу. Ефективність V-SLAM крити чно спирається на чисельну стабільність геометричних процедур: запровадження нормалізації координат усунуло погану обумовленість 8-точкового алгоритму оцінювання фундаментальної матриці й тим самим зменшило початкові похибки пози – головний тригер подальшого дрейфу. Водночас межі чисто геометричної робастизації стали очевидними: RANSAC добре протидіє випадковому шуму, однак у динамічних сценах рухомі об’єкти породжують структуровані хибні відповідності, які цей підхід не відсіює надійно, що неминуче веде до глобального дрейфу. У результаті семантика стає необхідною умовою стійкості: інтеграція детекторів на кшталт YOLO перетворює семантичне

маскування на проактивний фільтр динаміки, залишаючи для оцінювання поз лише статичний бекенд і забезпечуючи різке зниження ATE – до/на рівні 98.37% відносно суто геометричних підходів – тобто фактичне подолання проблеми дрейфу в динамічних середовищах.

РОЗДІЛ 2 АРХІТЕКТУРНИЙ АНАЛІЗ YOLO МОДЕЛЕЙ ТА КОМПЛЕКСНІ СТРАТЕГІЇ

Ефективність гібридних SLAM–систем, що використовують семантичні пріори, безпосередньо залежить від точності та надійності 2D детектора. Висока геометрична точність YOLOv8 є результатом його архітектурних інновацій та складної функції втрат, спрямованих на максимізацію якості локалізації.

2.1 Архітектурна глибина YOLOv8 та оптимізація локалізації

YOLOv8 використовує сучасну архітектуру з Decoupled, Anchor–Free Head, що забезпечує кращу узагальненість і вищу точність (mAP) порівняно з попередніми версіями, як–от YOLOv5x [5]. У його основі лежить комплексна функція втрат L_{Total} , яка являє собою зважену суму трьох компонентів: класифікаційної втрати (L_{Cls}), втрати регресії обмежувальної рамки (L_{CIoU}) та втрати розподілу (L_{DFL}).

$$L_{Total} = \lambda_{cls} L_{Cls} + \lambda_{box} L_{CIoU} + \lambda_{dfl} L_{DFL}, \quad (2.1)$$

де L_{Cls} – класифікаційна втрата для позитивних осередків/якорів, вимірює, наскільки впевнено модель присвоює правильний клас, L_{CIoU} – регресія бокса через Complete-IoU:

$$L_{CIoU} = 1 - IoU + \frac{\rho^2(b, b^g)}{c^2} + \alpha v$$

(штрафує за малий перетин, зсув центрів і невідповідність пропорцій), L_{DFL} – Distribution Focal Loss для тонкого локалізаційного регресу. Кожну з 4 сторін бокса (left, top, right, bottom) подають як розподіл по K бінів. Для цільового скаляра

$$y \in [0, K): \; l = \lfloor y \rfloor, \; r = l + 1, \; w_r = y - l, \; w_l = 1 - w_r.$$

Якщо $q = softmax(z)$ – ймовірності по бінах, то для однієї сторони

$$L_{DFL} = -\left(w_l \log log \, q_l + w_r \log log \, q_r\right),$$

і сумується по 4 сторонах. Це дає субпіксельну точність меж.

$\lambda_{cls}, \lambda_{box}, \lambda_{dfl}$ – ваги-гіперпараметри для балансування градієнтів (підбираються під датасет/архітектуру).

Вагові коефіцієнти λ необхідні для балансування внеску кожної втрати в загальний градієнт, що запобігає домінуванню одного компонента і забезпечує спільну оптимізацію всіх трьох аспектів для підвищення точності детекції.

Для класифікаційної втрати YOLOv8 використовує Varifocal Loss (VFL). VFL розроблена для забезпечення IoU–Aware Classification Score (IACS) [9]. Це має вирішальне значення, оскільки традиційні скоринги класифікації часто не корелюють з фактичною якістю локалізації, виміряною за допомогою Intersection over Union (IoU).

У традиційних системах, високоточна рамка (з високим IoU) могла мати низький класифікаційний скоринг і помилково відкидатися під час процедури Non–Maximum Suppression (NMS). VFL вирішує цю проблему, надаючи більшу вагу позитивним прикладам, які мають високу якість локалізації. Це змушує модель концентруватися на об'єктах, які вона може виявити та

локалізувати з високою геометричною точністю, підвищуючи надійність семантичного пріора для подальшого 3D картування.

Для регресії обмежувальної рамки використовується Complete IoU Loss (L_{CIoU}) [9]. L_{CIoU} значно перевершує базовий IoU, оскільки вона враховує не лише площу перетину, але й три додаткові геометричні фактори: відстань між центрами $\left(\rho^2\left(b, b^g\right)/c^2\right)$, співвідношення сторін v, та їхній ваговий коефіцієнт α.

Формула LCIoU відображає цей комплексний підхід:

$$L_{CIoU} = 1 - IoU + \frac{\rho^2\left(b,b^g\right)}{c^2} + \alpha v, \tag{2.2}$$

де b – передбачуваний бокс, b^g – GT-бокс (обидва осьово вирівняні), IOU – перетин-над-об'єднання для двох боксів,

$$\rho^2(b, b^g) = (x - x^g)^2 + (y - y^g)^2 -$$

квадрат відстані між центрами, c^2 – квадрат діагоналі найменшого спільного охоплювального прямокутника для bі b^g,

$$v = \frac{4}{\pi^2}\left(\arctan\ arctan\ \frac{w^g}{h^g} - \arctan\ arctan\ \frac{w}{h}\right)^2 -$$

невідповідність співвідношень сторін, $\alpha = \frac{v}{1-+v}$ – ваговий коефіцієнт для v.

Ця функція гарантує, що процес оптимізації може тривати, навіть якщо передбачувана рамка (b) та істинна рамка $\left(b^g\right)$ не перетинаються (тобто IoU дорівнює нулю). Врахування додаткових факторів забезпечує швидшу збіжність форми рамки та покращує її кінцеву геометричну відповідність.

Distribution Focal Loss $\left(L_{DFL}\right)$ є ключовим архітектурним рішенням, запозиченим із Generalized Focal Loss, і спрямована на моделювання невизначеності. Традиційно координати обмежувальної рамки моделювалися як фіксоване точкове значення (розподіл Дірака), що не відображало невизначеності, притаманної складним візуальним сценам.

DFL вирішує це, моделюючи координати як довільний розподіл ймовірностей. Це змушує нейронну мережу концентрувати ймовірності між двома найближчими дискретними бінами, які оточують істинну безперервну цільову координату y. Це забезпечує локальну статистичну точність.

Математично DFL (для частини координати y між бінами i та i+1) виражається так:

$$L_{DFL}(S, y) = -\left((y - i)\log log\left(S_{i+1}\right) + (i + 1 - y)\log log\left(S_{i}\right)\right) \quad (2.3)$$

де $y \in [0, K)$– цільове дійсне значення (відстань до межі) у бінованому інтервалі, $i = \lfloor y \rfloor$– індекс лівого біна, $S = softmax(z) \in R^{K}$ – передбачений **розподіл** по Kбінів; S_{j} – ймовірність j-го біна.

Стратегічна синергія L_{CIoU} та L_{DFL} забезпечує високу точність YOLOv8: L_{CIoU} відповідає за глобальну геометричну відповідність та швидку збіжність, тоді як L_{DFL} забезпечує локальну статистичну точність, мінімізуючи невизначеність на піксельному рівні. Цей дуальний механізм є ключем до використання YOLOv8 як високоточного семантичного пріора для подальшої проекції у 3D простір.

Таблиця 2.1 – Компоненти та математична роль функції втрат YOLOv8

Компонент Втрати	**Математична Основа**	**Основне Призначення**	**Роль у Локалізації/Класифікації**
Classification Loss $\left(L_{Cls}\right)$	Varifocal Loss (VFL)	Спільна оптимізація класу та якості IoU	Пріоритезація високоточних позитивних прикладів (IoU–Aware Classification)
Bounding Box Loss $\left(L_{CIoU}\right)$	Complete IoU	Регресія геометричних параметрів	Оптимізація перетину, відстані центрів та співвідношення сторін для швидкої збіжності
Distribution Focal Loss $\left(L_{DFL}\right)$	Generalized Focal Loss	Моделювання розподілу координат	Зменшення невизначеності, фокусування на точних безперервних значеннях

Джерело: створено автором на основі [7]

2.2 Стратегії проекції семантики з 2D у 3D (2D–to–3D fusion)

Ефективна інтеграція 2D семантики (отриманої від YOLO) у 3D простір є обов'язковою для створення семантичної карти.

Ранні гібридні підходи використовували 2D обмежувальні рамки для визначення області інтересу у 3D просторі, відомої як зрізаний конус (frustum). Хоча цей підхід був швидким, він виявився чутливим до похибок

2D детектора, оскільки помилки локалізації у 2D сильно масштабуються при проекції у 3D, особливо для далеких об'єктів.

Сучасні конвеєри прагнуть до більш якісної інтеграції. Наприклад, підхід FusionVision поєднує швидку ідентифікацію об'єктів за допомогою YOLO з уточнюючою моделлю сегментації (наприклад, FastSAM). Використання FastSAM дозволяє отримати високоточні межі об'єктів (піксельні маски), які потім застосовуються до глибинних даних RGB–D. Це значно покращує якість 3D сегментації, дозволяючи точніше маскувати динамічні об'єкти та знижувати шум під час 3D реконструкції.

Для досягнення точної 3D семантичної сегментації критично важливо подолати інконсистентність, яка може виникати в 2D мітках, наданих YOLO, при спостереженні одного й того ж об'єкта з різних ракурсів.

Система Open–YOLO 3D демонструє, як можна досягти відкрито–словникової 3D інстанс–сегментації без дорогої залежності від великих 2D foundation models. Натомість, вона використовує швидку 2D детекцію YOLO з множинних ракурсів. Ключовим елементом є механізм Multi–View Prompt Distribution (MVPDist). MVPDist агрегує семантичну інформацію, отриману з 2D обмежувальних рамок у різних кадрах, щоб передбачити надійну семантичну мітку для 3D інстанс–маски. Це дозволяє компенсувати помилки класифікації, які могли виникнути в окремому 2D кадрі, забезпечуючи 3D консистентність.

Для забезпечення швидкості використовується механізм Accelerated Visibility Computation (VAcc), який прискорює асоціацію 2D міток з 3D пропозиціями, досягаючи значного прискорення (до $16\times$) порівняно з аналогами. Це доводить, що швидкісні 2D детектори, як YOLO, є ключовим компонентом для впровадження складного 3D семантичного картування на обмежених обчислювальних платформах.

2.3 Вплив семантичних обмежень на бекенд оптимізації (Graph SLAM)

SLAM–системи використовують бекенд оптимізації, який зазвичай представлений у вигляді графа (Pose Graph), де вузли відповідають позам камери або об'єктам, а ребра – вимірюванням (обмеженням). Оптимізація мінімізує загальну помилку, забезпечуючи глобальну узгодженість.

Інтеграція семантичної інформації від YOLO впливає на бекенд, діючи як потужний пре–кондиціонер даних [7]. Фільтрація динамічних точок, які є геометричними викидами, безпосередньо знижує рівень шуму у вимірюваннях, що надходять до оптимізатора.

У контексті Graph SLAM, де стан μ розв'язується за допомогою матриці інформації Ω (яка є оберненою до коваріаційної матриці) через рівняння $\Omega\mu = \xi$, зменшення шуму призводить до того, що інформаційна матриця стає краще обумовленою [10]. Як наслідок, коваріація (невизначеність) оцінки пози значно зменшується. Таким чином, YOLO, усуваючи джерела шуму, є прямим механізмом підвищення чисельної надійності геометричного бекенда, що зменшує накопичення дрейфу.

У Object–Aware SLAM (наприклад, SOLO–SLAM), об'єкти, ідентифіковані YOLO, можуть бути додані до графа оптимізації як статичні вузли.

Накладання семантично зумовлених обмежень, що спираються на припущення квазістатичності відповідних об’єктів відносно сцени, збагачує граф поз (Pose Graph) жорсткими та високодостовірними ребрами. Це критично важливе доповнення, оскільки підвищує робастність системи й зменшує глобальний дрейф траєкторії навіть у довготривалих послідовностях за відсутності або утрудненості замикання циклів (loop closure). Відтак семантика виступає високонадійним, безперервно оновлюваним джерелом геометричної узгодженості.

2.4 Обчислювальна ефективність та апаратна оптимізація YOLO–SLAM для embedded systems

Інтеграція методів глибинного навчання (зокрема YOLO) у SLAM-системи істотно підвищує обчислювальне навантаження. Хоч ці підходи й дозволяють враховувати динамічні сцени, утримання роботи в реальному часі – особливо на вбудованих платформах на кшталт NVIDIA Jetson – вимагає інтенсивних апаратних і програмних оптимізацій. За таких умов оптимізації перестають бути «приємним покращенням» і стають необхідним технологічним каталізатором, який фактично робить можливим впровадження семантично обізнаних переваг у реальних застосуваннях. Для досягнення потрібної продуктивності на обмежених пристроях слід максимально задіювати спеціалізоване обладнання та оптимізовані формати: перетворювати моделі YOLO (напр., YOLOv8) у TensorRT-двигуни .engine для максимального прискорення інференсу на GPU Jetson [5]; застосовувати квантизацію до INT8 (через PTQ або QAT), що знижує вимоги до пам’яті орієнтовно вчетверо та споживання енергії, водночас даючи суттєве прискорення (часто понад 5×) – критично для платформ із обмеженим живленням на кшталт Jetson Nano [10]; а також використовувати DLA-ядра, наявні в низці пристроїв Jetson (наприклад, AGX Orin), щоб розвантажити основний GPU/CPU, підвищити енергоефективність (до ~2.5× відносно GPU) і пропускну здатність, що особливо важливо для мобільних роботів; DLA показує найбільшу ефективність на компактних моделях [11, 12, 15].

Для підтримання високої частоти кадрів, необхідної для трекінгу камери (не рідко до 60 FPS), застосовується асинхронна обробка: механізм поширення маски відокремлює ресурсоємний інференс глибинної моделі від основного циклу трекінгу, тож система тимчасово використовує маску з попереднього кадру, доки в паралельному потоці обчислюється нова, точніша сегментація. Це дозволяє зберігати реальний час без жорсткого компромісу

щодо якості.

Таблиця 2.2 – Методи апаратної оптимізації для вбудованих SLAM систем

Техніка оптимізації	**Застосування**	**Цільова платформа**	**Ключовий ефект**	**Джерело інференсу**
TensorRT Conversion	YOLO Network Inference	NVIDIA Jetson (Orin/AGX)	Максимальне прискорення інференсу (високий FPS)	ONNX/PyTorch → .engine
Квантизація (INT8)	DNN Weights and Activations	Embedded CPUs/DLA	Зниження пам'яті (до 4x) та енергоспоживання	QAT / PTQ
DLA Utilization	Deep Learning Inference	Jetson AGX/Orin NX	Розвантаження GPU/CPU, підвищення енергоефективності (2.5X)	Спеціалізовані ядра
Асинхронна Обробка	Tracking та Semantic Thread	Загальний V–SLAM	Підвищення Real–Time FPS	Поширення маски (Mask Propagation)

Джерело: створено автором на основі [10, 12, 13, 15, 16]

2.5 Дивергенція динамічного картування: новітні 3D репрезентації

Останні здобутки у 3D-картуванні демонструють розмаїття репрезентацій і підходів до виявлення динаміки – від семантичних (на основі YOLO) до суто геометричних. 3D Gaussian Splatting (3DGS) – це нова явна 3D-репрезентація, яка забезпечує фотореалістичний рендеринг зі швидкістю,

суттєво вищою за Neural Radiance Fields (NeRF), що робить її придатною для використання SLAM у реальному часі(наприклад, MonoGS), де єдина модель сцени представлена набором еліпсоїдальних гаусіанів для трекінгу, мапінгу й рендерингу. Втім, як і класичні SLAM-системи, 3DGS лишається вразливою до рухомих об'єктів, що призводить до «привидів» у статичній реконструкції. Альтернативну лінію розвиває ADD-SLAM (Adaptive Dynamic Dense SLAM), яка відмовляється від попередньо навчених семантичних детекторів на кшталт YOLO і покладається на аналіз геометричної узгодженості сцени. Динаміка тут визначається категорійно-агностично: регіон позначається рухомим, якщо спостережувана глибина D у вхідному кадрі менша за відрендерену зі статичної гаусіанової карти $\hat{D}$. Такий чисто геометричний критерій дає змогу виявляти рух будь-яких об'єктів, включно з невідомими або рідкісними класами.

Водночас неявні нейронні представлення (NeRF) потенційно уможливлюють спільне навчання геометрії, кольору та семантики, але стикаються з труднощами забезпечення 3D-узгодженості семантичного поля, особливо коли 2D-мітки (часто отримані YOLO) неузгоджені між ракурсами. Системи на зразок NIS-SLAM і SNI-SLAM долають це, інтегруючи попередньо навчені 2D-мережі й застосовуючи багатовидову семантичну фузію для отримання консистентних 3D-семантичних уявлень. У підсумку швидкі та точні 2D-семантичні пріори на кшталт YOLO залишаються ключовим компонентом сучасних парадигм 3D-картування – незалежно від вибраної 3D-репрезентації.

Висновки до розділу 2

Аналіз теоретичних засад та емпіричних результатів підтверджує, що побудова 3D карт на основі YOLO моделей є визначальною парадигмою для

сучасного динамічного SLAM.

Семантика як Необхідна Умова Глобальної Стійкості та Чисельної Якості. Традиційне припущення про статичне середовище є найбільшим джерелом дрейфу у V–SLAM. Інтеграція YOLO для проактивної фільтрації динамічних об'єктів забезпечує радикальне підвищення стійкості. Кількісні дані (зниження ATE до 98.37%) свідчать, що семантичне маскування є фундаментальною вимогою для забезпечення глобальної узгодженості. Крім того, функція YOLO полягає не лише у фільтрації; вона діє як пре–кондиціонер даних, що зменшує рівень шуму у вимірюваннях. Це призводить до кращої обумовленості Інформаційної Матриці бекенда SLAM та, відповідно, до значного зменшення коваріації (невизначеності) оцінки пози.

Висока надійність 2D семантичних пріорів, які надає YOLOv8, є результатом стратегічної подвійної функції втрат локалізації: L_{CIoU} забезпечує глобальну геометричну корекцію форми рамки, тоді як L_{DFL} забезпечує локальну статистичну точність, мінімізуючи невизначеність прогнозування координат на піксельному рівні. Ця точність є критичною, оскільки помилки у 2D обмежувальних рамках сильно масштабуються при їхній проекції у 3D простір.

Хоча впровадження YOLO додає значне обчислювальне навантаження, ця витрата є необхідною для вирішення глобальної проблеми дрейфу. Для забезпечення роботи в режимі реального часу на вбудованих платформах (Jetson) необхідна агресивна апаратна оптимізація. Методи, такі як TensorRT Conversion, INT8 квантизація (зниження пам'яті до 4 ×) та використання спеціалізованих ядер DLA, є технологічним каталізатором, який уможливлює реалізацію семантичних переваг YOLO в умовах обмежених ресурсів.

Для перенесення 2D семантики YOLO у 3D простір потрібні механізми, які долають інконсистентність між кадрами, як–от Multi–View Prompt Distribution (MVPDist) у Open–YOLO 3D. Ці 2D семантичні пріори також є

необхідними для навчання 3D консистентного семантичного поля у новітніх неявних представленнях (NeRF, NIS–SLAM). Майбутнє динамічного SLAM, ймовірно, лежатиме у конвергенції: використання швидкісних, семантичних YOLO–фільтрів для відомих класів у поєднанні з категорійно–агностичним геометричним аналізом (як у ADD–SLAM) для виявлення невідомого руху, що забезпечить максимальну стійкість та гнучкість системи.

РОЗДІЛ 3 РОЗРОБЛЕННЯ АЛГОРИТМІЧНОЇ ЧАСТИНИ ТА ДОСЛІДЖЕННЯ СИСТЕМИ ROBUST RGB-D SLAM НА ОСНОВІ YOLOV8 І TSDF

3.1 Архітектура системи та принципи роботи модулів

Розроблена система візуальної одночасної локалізації та побудови карти в динамічних середовищах базується на поєднанні сучасних нейромережевих методів аналізу зображень і класичних геометричних алгоритмів SLAM. Архітектура системи спроєктована таким чином, щоб цілісний конвеєр – від «сирих» RGB-D кадрів до тривимірної моделі сцени – залишався робастним у присутності незалежно рухомих об’єктів, які потенційно руйнують припущення про статичність середовища.

Ключовою ідеєю є семантично кероване відокремлення динаміки: спочатку система на рівні RGB зображень визначає, які пікселі належать рухомим об’єктам (люди, інші мобільні агенти), а вже потім відсікає відповідні глибинні вимірювання з подальшої геометричної обробки. Усі оцінки руху камери, а також інтеграція глибинних даних у TSDF-об’єм виконуються виключно по статичних частинах сцени. Таким чином, динаміка не «змішується» з геометрією, а керовано видаляється ще до побудови карти.

Загальна структура та вимоги до архітектури

Архітектура системи має бути одночасно:

- робастною до динаміки, тобто не ламатися у присутності людей, що рухаються, та інших незалежно рухомих об’єктів;
- достатньо простою для реалізації і налагодження, щоб її можна було реалізувати у вигляді відносно компактного коду, без надмірних залежностей;
- здатною працювати близько до реального часу на споживчому GPU;

- розширюваною, тобто такою, що допускає заміну окремих модулів (детектор, трекер, тип одометрії, варіант TSDF) без повного переписування системи.

Для досягнення цих цілей система будується як ланцюг модулів, де кожен етап виконує свою функцію й має чітко визначений інтерфейс:

- модуль завантаження та синхронізації RGB-D даних;
- модуль попередньої обробки глибинних карт;
- модуль детекції динамічних об’єктів на RGB-кадрах;
- модуль багатокадрового трекінгу (стабілізація динаміки у часі);
- модуль формування динамічної маски та очищення глибини;
- модуль гібридної RGB-D одометрії (frame-to-frame);
- модуль ICP «кадр → модель» (frame-to-model);
- модуль об’ємної реконструкції на основі TSDF;
- модуль роботи з ключовими кадрами та замикання циклів;
- модуль логування, збереження траєкторій і експорту 3D карт.

При цьому семантичні модулі (детекція, трекінг, маска) працюють у просторі RGB-кадрів, а геометричні (одометрія, ICP, TSDF, loop closure) – у просторі глибинних даних і 3D координат. Їх пов’язує динамічна маска, яка виступає «мостом» між семантичним та геометричним шарами.

На вході система отримує послідовність RGB-зображень і відповідних глибинних карт із датасету (TUM RGB-D). Першим кроком є синхронізація: з огляду на те, що RGB і Depth мають окремі часові мітки, необхідно для кожного моменту часу підібрати пару файлів, що відповідають один одному з точністю до невеликого порогу. Результатом цього етапу є впорядкований список кадрів, де кожен кадр містить: шлях до RGB-зображення, шлях до глибинної карти, часову мітку та індекс у послідовності.

Після цього глибинні карти проходять попередню обробку. Сирі глибинні значення зчитуються у цілочисельному форматі й далі масштабуються у метри за допомогою коефіцієнта глибинного сенсора. Значення, які виходять за межі фізично допустимого діапазону (занадто малі

або надто великі), вважаються шумом і зануляються. Для зменшення шуму й артефактів можуть застосовуватись згладжувальні фільтри (наприклад, двосторонній), які зменшують випадкові коливання, але не розмивають різкі геометричні границі.

Отримавши підготовлену глибинну карту, система переходить до семантичного аналізу RGB-кадру. Кольорове зображення передається в детектор об’єктів, який повертає набір обмежувальних рамок із класами та рівнями впевненості. Ці рамки інтерпретуються як кандидати на динамічні об’єкти, насамперед люди. На цьому етапі вся інформація зосереджена лише в одному кадрі в часі.

Щоб перетворити набір покадрових детекцій на стабільні траєкторії, дані передаються у трекер. Модуль трекінгу порівнює детекції між кадрами, приписуючи їм сталі ідентифікатори й формуючи траєкторії руху. Це дозволяє системі відрізняти короткочасні пропуски детекції від реального зникнення об’єкта, а також згладжувати шум, пов’язаний із неправильними одиничними спрацьовуваннями детектора.

На основі множини активних треків формується динамічна маска у просторі глибинної карти: всі пікселі, які лежать у проєкції рамок треків, позначаються як такі, що належать рухомим об’єктам. Ця маска застосовується до глибинної карти, в результаті чого всі глибинні значення у динамічних областях зануляються. Так утворюється очищена глибинна карта, яка містить лише інформацію про статичну частину сцени.

Далі в роботу вступає геометричний шар. Очищені RGB-D кадри передаються до модуля гібридної RGB-D одометрії, який оцінює відносний рух камери між послідовними кадрами. Після цього отримана поза уточнюється за допомогою ICP «кадр → модель», де поточний кадр зіставляється з існуючою тривимірною моделлю сцени, що вже накопичена в TSDF-об’ємі.

На основі оновленої позиції камери очищена глибинна інформація інтегрується у TSDF-об’єм, який виступає глобальним носієм тривимірної

карти середовища. Попередньо реконструйований об’єм слугує базою для ICP, виведення фінального мешу та візуальної оцінки результатів.

Паралельно система слідкує за змінами в траєкторії камери, відбираючи ключові кадри за порогом зміни пози й орієнтації. Ці кадри використовуються для виявлення можливих замикань циклів: якщо камера повертається в раніше відвідану область, система може порівняти ознаки ключових кадрів, виявити збіг та врахувати це при глобальному вирівнюванні траєкторії.

Семантичний шар архітектури відповідає за визначення того, які частини сцени можуть бути динамічними. Це принципово інший тип задачі, ніж власне локалізація: замість оцінки руху камери система визначає, які пікселі не можна використовувати як «якорі» для геометрії.

Детектор об’єктів реалізує функцію, яка для кожного RGB-кадру повертає набір детекцій. Кожна детекція містить:

- координати прямокутника (у пікселях);
- індекс класу (наприклад, «person»);
- оцінку впевненості (ймовірність того, що рамка справді містить об’єкт цього класу).

У коді ці дані представляються структурою з масивами координат, класів і впевненостей. Поріг впевненості у конфігурації системи визначає, які рамки будуть далі передані у трекер, а які відкинуті як ненадійні.

Оскільки детектор не має часової пам’яті, потрібен окремий модуль трекінгу, що працює поверх послідовності детекцій. Його завдання – встановити відповідність між об’єктами, виявленими в різні моменти часу, і призначити їм сталий ідентифікатор. Всередині трекера реалізується логіка співставлення рамок за площею перекриття й оцінкою ймовірності, а також внутрішня модель стану об’єкта, яка враховує, як довго об’єкт був у полі зору, чи зник, чи з’явився знову тощо.

Саме завдяки трекеру архітектура отримує стабільні траєкторії динамічних об’єктів, а не просто окремі «спалахи» детекції. Це особливо важливо у кадрових послідовностях, де об’єкт може частково перекриватися

іншими об’єктами, виходити за межі кадру або потрапляти у зони низького контрасту.

На базі активних треків будується динамічна маска глибинної карти. Із практичної точки зору це означає, що всі пікселі, які потрапляють усередину проєкції рамок треків, позначаються як такі, що належать рухомим об’єктам. На рівні даних це реалізується як двовимірний масив того самому розміру, що й глибинна карта, де значення 1 означає «динаміка», а 0 – «статична частина сцени».

Після формування маски система створює очищену глибинну карту: для всіх пікселів, які позначені як динамічні, глибинні значення зануляються або позначаються як відсутні. Усі подальші геометричні обчислення – одометрія, ICP, інтеграція в TSDF – працюють саме з цією очищеною картою. Таким чином, динамічна маска стає критичним компонентом, що визначає, які дані взагалі мають право впливати на оцінку руху камери та структуру карти.

Архітектурно це означає, що взаємодія між семантичним і геометричним блоками є односторонньою: детекція та трекінг впливають на те, які глибинні пікселі використовуються для геометрії, але геометричні модулі не змінюють роботу детектора й трекера. Такий поділ спрощує реалізацію й дозволяє модифікувати семантичний шар (заміна моделі детекції, трекера, критеріїв маски) без необхідності втручатися у математичну частину одометрії та TSDF.

Після того як семантичний шар виконав свою роботу й сформував очищені RGB-D кадри, геометричний шар відповідає за оцінку траєкторії та побудову тривимірної карти сцени.

Гібридна RGB-D одометрія оцінює відносну позу камери між сусідніми кадрами, комбінуючи фотометричну й геометричну інформацію (докладні формули наведені у підрозділі 3.3). Важливо, що у всіх сумах та інтегралах, які використовуються при обчисленні функцій похибки, беруть участь лише ті пікселі, що не позначені як динамічні. Таким чином, усі фотометричні та геометричні відповідності будуються лише на базі статичних об’єктів, що

радикально знижує ризик використання «хибних» точок, пов’язаних із рухомими людьми.

Одометрія працює локально, і тому наступним рівнем уточнення виступає ICP «кадр → модель». У цьому модулі поточна хмара точок зіставляється не з попередньою, а з уже накопиченою моделлю сцени, закодованою в TSDF-об’ємі. Корекція поз на основі глобальної структури дозволяє зменшити локальні помилки й компенсувати дрейф.

І, нарешті, інтеграція в TSDF відповідає за побудову 3D-карти. Очищені глибинні вимірювання, перетворені у світову систему координат з урахуванням поточної оцінки поз, оновлюють значення TSDF у воксельному об’ємі. Поверхня сцени відновлюється як ізоповерхня нульового значення TSDF, що дозволяє отримати гладкий тривимірний меш без «привидів» від рухомих об’єктів.

Оскільки будь-яка інкрементальна одометрія схильна до накопичення невеликих помилок у кожному кроці, на довгих траєкторіях виникає дрейф. Щоб протидіяти цьому, система відбирає ключові кадри й будує над ними граф поз. Критерії відбору ключових кадрів зазвичай базуються на порогах зміни положення й орієнтації камери: якщо камера змістилася або повернулась більше, ніж на задану величину відносно останнього ключового кадру, поточний кадр стає новим ключовим.

Для кожного ключового кадру зберігаються його поза, а також візуальні ознаки (наприклад, ORB-фічі). Модуль замикання циклів періодично порівнює нові ключові кадри з попередніми, шукаючи пари кадрів, які відповідають одній і тій самій області сцени. Якщо такий збіг підтверджується як на рівні ознак, так і на рівні геометрії (через локальний ICP), у граф поз додається додаткове ребро, яке «стягує» траєкторію, компенсуючи дрейф.

Таким чином, архітектура системи являє собою двоєдину структуру: семантичний блок відповідає за виявлення й стабілізацію динаміки, а геометричний – за оцінку руху камери й побудову карти. Динамічна маска є

механізмом, що пов’язує ці два блоки, дозволяючи семантиці безпосередньо впливати на геометрію.

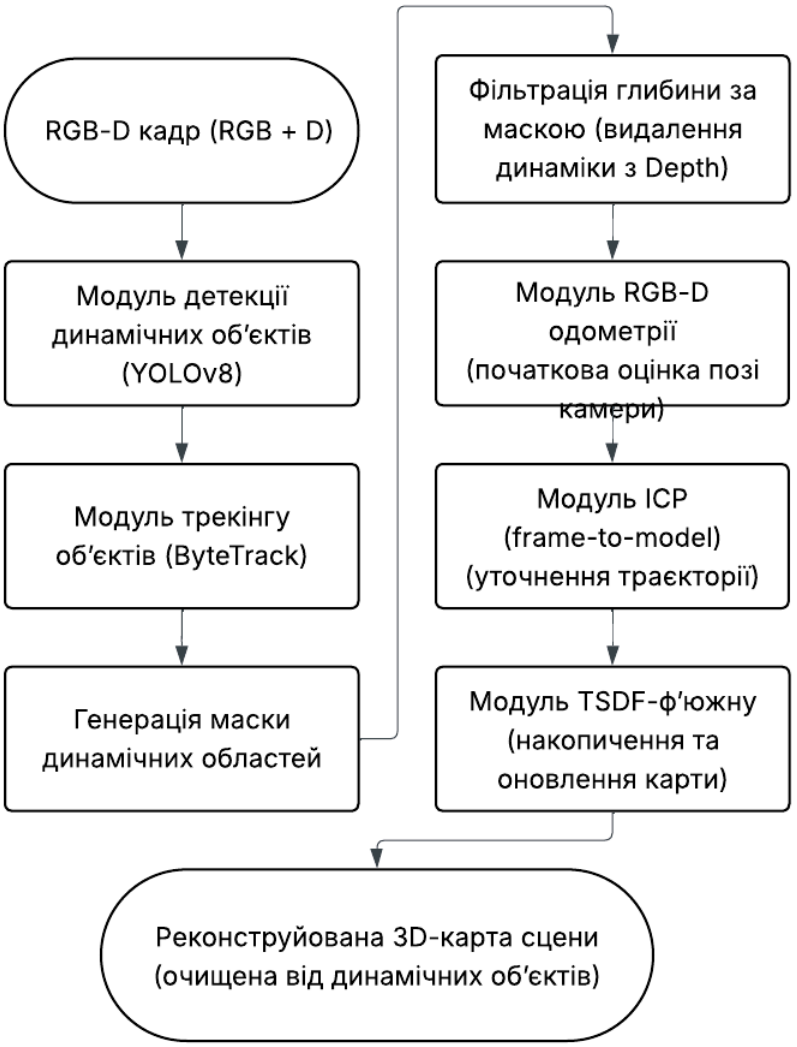

Рисунок 3.1 – Архітектура системи

У підсумку отримується система, яка:

- працює на реальних RGB-D даних;
- відсікає вклад рухомих об’єктів на рівні глибинної інформації;
- оцінює позу камери з використанням тільки статичних частин сцени;
- будує щільну TSDF-карту без артефактів динаміки;

- здатна виявляти та враховувати замикання циклів.

У наступних підрозділах (3.2 і 3.3) ця архітектура деталізується відповідно до конкретних програмних реалізацій та математичних моделей, які описують роботу одометрії, ICP та TSDF-об’єму.

3.2 Реалізація програмних модулів та логіка функціонування системи

Якщо в підрозділі 3.1 було описано загальну архітектуру системи та взаємодію її компонентів на концептуальному рівні, то в цьому підрозділі розглядається конкретна програмна реалізація: як організовано код, які модулі створені, які структури даних використовуються та яким чином у реальному виконанні проходить повний цикл «від кадру до карти».

Реалізація побудована як послідовний конвеєр обробки кадрів, де кожен етап відповідає певному логічному блоку архітектури: завантаження та синхронізація даних, семантичний аналіз (детекція + трекінг), формування динамічної маски, гібридна одометрія, уточнення ICP, інтеграція в TSDF-об’єм, робота з ключовими кадрами та збереження результатів. Кожен із цих блоків оформлений у вигляді окремих функцій або класів, що полегшує налагодження та розширення системи.

Програмна реалізація організована навколо «головного» модуля, який відповідає за запуск усієї системи. На рівні коду це, як правило, функція main() або аналогічний за змістом блок, із якого:

- зчитується конфігурація (шлях до датасету, параметри камери, параметри TSDF, налаштування детектора та трекера);
- ініціалізуються всі основні об’єкти (модель YOLO, трекер, TSDF-об’єм, структури для зберігання траєкторії);
- створюється список синхронізованих кадрів;

- запускається головний цикл SLAM.

У конфігурації зберігаються всі параметри, які можуть впливати на поведінку системи, зокрема: тип послідовності (наприклад, fr1_xyz або fr3_walking_xyz), кількість кадрів для обробки, пороги глибини, розмір вокселя TSDF, параметри одометрії, пороги для вибору ключових кадрів, пороги детектора та трекера. Такий підхід дозволяє легко змінювати сценарій роботи системи без модифікації основного коду.

Головний цикл обробки кадрів має наступну логіку: для кожного синхронізованого кадру система послідовно виконує завантаження зображення, передобробку, детекцію, трекінг, формування маски, очищення глибини, оцінку руху, уточнення ICP, інтеграцію в TSDF та, за потреби, оновлення ключових кадрів і графа поз. Після завершення обробки всіх кадрів результати (траєкторія, карта, лог-файли) зберігаються на диск.

Першим практичним етапом у коді є завантаження RGB- та глибинних кадрів із датасету TUM RGB-D. Для цього використовуються функції, які:

- читають текстовий файл асоціацій (associate.txt), де наведено списки імен файлів та часових міток;
- формують списки шляхів до RGB-зображень та відповідних глибинних карт;
- перевіряють, що для кожного RGB-кадру існує відповідна глибинна інформація з допустимим відхиленням за часом.

На цьому ж етапі формується масив структур кадрів, де для кожного індексу і зберігаються: шлях до RGB-файлу, шлях до Depth-файлу, часова мітка та внутрішній індекс у послідовності.

Під час проходження головного циклу для кожного кадру виконується завантаження RGB-зображення (зазвичай за допомогою OpenCV або аналогічної бібліотеки) та глибинної карти. RGB-зображення конвертується в потрібний формат (наприклад, BGR → RGB), масштабується/нормалізується залежно від вимог детектора. Глибинна карта зчитується як матриця цілих значень, після чого перетворюється в метри через ділення на depth_factor.

Далі запускається модуль попередньої обробки глибини. У реальному коді це окрема функція, яка:

- зануляє глибинні значення, менші за depth_min та більші за depth_max;
- за потреби застосовує двосторонній фільтр (bilateral filtering), щоб зменшити шум, не руйнуючи геометричні границі;
- повертає «очищену» карту, придатну для подальшого відновлення хмари точок.

Результати цього етапу: підготовлене кольорове зображення й глибинна карта з коректним масштабом та обмеженим діапазоном значень.

Ініціалізація та використання детектора динамічних об’єктів

На рівні коду модуль детекції представлено як об’єкт моделі (наприклад, YOLOv8), який ініціалізується один раз на початку роботи програми. При ініціалізації:

- завантажуються ваги натренованої моделі з файлу;
- задається конфігурація пристрою обчислень (GPU/CPU);
- встановлюються основні параметри інференсу (розмір вхідного зображення, поріг упевненості, поріг NMS тощо).

Далі, в головному циклі, для кожного RGB-кадру викликається функція інференсу моделі. Вона повертає структуру з детекціями, де для кожного виявленого об’єкта зберігаються: координати прямокутника в координатах зображення, індекс класу та скалярна оцінка ймовірності. За допомогою порогу впевненості (conf_thresh) із цього списку відкидаються всі детекції, які вважаються ненадійними.

У коді на цьому етапі часто відбувається додатковий фільтр: наприклад, детекції інших класів, окрім «person», можуть ігноруватися, якщо в даній постановці дослідження людину вважають основним джерелом динаміки. За результатами роботи детектора формується масив прямокутників, який передається у трекер.

Модуль трекінгу (ByteTrack або інший) також ініціалізується на початку

роботи програми з власною конфігурацією: порогами асоціації (track_thresh, match_thresh), параметрами, що описують, скільки кадрів трек може «пережити» без нових детекцій (max_time_lost), та внутрішніми станами.

На кожному кроці головного циклу в трекер передаються детекції поточного кадру. Усередині трекера відбувається:

- обчислення матриці перекриттів між рамками поточного кадру й рамками попереднього;
- оцінка «якості» можливих зіставлень (наприклад, за IoU та впевненістю);
- розв’язання задачі призначення (які нові рамки відповідають яким існуючим трекам, а які є новими об’єктами);
- оновлення структур треків: оновлення координат рамок, впевненості, статусу активності, «віку» треку.

У вихідній структурі треків кожен об’єкт має стабільний track_id. Надалі для формування динамічної маски використовуються саме активні треки, а не «сирі» детекції. Код у цьому місці, як правило, забезпечує інтерфейс типу «віддати всі активні треки у вигляді списка прямокутників з ID».

Формування динамічної маски та очищення глибини

Наступний модуль – формування динамічної маски. У реалізації це окрема функція, яка приймає:

- розмір глибинної карти (висота, ширина);
- список активних треків (координати рамок, ідентифікатори).

Функція створює двовимірний масив (матрицю) початково заповнений нулями. Для кожного треку прямокутник із координатами рамки копіюється у координати глибинної карти (якщо роздільна здатність RGB і Depth відрізняється, використовується масштабування). В межах цього прямокутника елементи маски виставляються в 1. За потреби, до маски можуть застосовуватися прості морфологічні операції (розширення, закриття), щоб уникнути «дір» по краях об’єктів.

Після побудови маски викликається функція очищення глибинної карти. Вона проходить по всіх пікселях, і якщо маска в точці (u,v) дорівнює 1, то відповідне глибинне значення зануляється. Результатом стає очищена глибинна карта, яка надалі використовується в одометрії та TSDF. У коді це зазвичай проста операція «маскування» масиву глибини логічною матрицею.

Модуль одометрії реалізований у вигляді окремої функції або класу, який приймає на вхід два сусідні очищені кадри: попередній $\left(I_{t-1}, D'_{t-1}\right)$ та поточний $\left(I_t, D'_t\right)$, а також попередню оцінку руху (початкову позу або одиничну матрицю).

Усередині модуля відбуваються такі ключові кроки:

- відновлення 3D-точок із глибинної карти, використовуючи внутрішні параметри камери;
- обчислення градієнтів інтенсивності зображення (для фотометричної частини);
- формування системи рівнянь для фотометричної й геометричної помилок;
- обчислення Якобіана за параметрами руху (ξ);
- ітеративне розв’язання нормальних рівнянь і оновлення оцінки матриці руху.

На рівні коду це, як правило, означає використання функцій бібліотеки лінійної алгебри (для розв’язання систем рівнянь) і оптимізаційних циклів з фіксованою кількістю ітерацій або із критерієм зупинки за зміною трансляції та обертання. Кінцевим результатом роботи одометрії є матриця $\left(T_{t\leftarrow t-1}\right)$, яка описує рух камери між двома кадрами.

Далі, для підвищення точності, одержана поза може уточнюватися модулем ICP «кадр → модель». Цей модуль використовує:

- поточну очищену хмару точок (з глибинної карти $\left(D'_t\right)$);

- поверхню сцени, відновлену з TSDF-об’єму (у вигляді хмари точок або мешу).

Класичний ICP виконує пошук відповідних точок між кадром і моделлю та мінімізує відстань між ними. У коді це зазвичай реалізується за допомогою бібліотеки (наприклад, Open3D), де викликається відповідна функція реєстрації хмар точок registration_icp з переданими параметрами (метрика, поріг відстані, кількість ітерацій).

Результатом ICP є додаткова матриця перетворення, яка компонується з оцінкою з одометрії, утворюючи уточнену позу камери у світовій системі координат.

Після того як поза камери для кадру (t) оцінена, система переходить до інтеграції очищених глибинних даних у TSDF-об’єм. На рівні коду TSDF реалізовано як окремий клас, що зберігає:

- розмір вокселя (voxel_length);
- усічення функції відстані (sdf_trunc);
- внутрішню сітку вокселів (масив значень TSDF і масив ваг);
- межі об’єму в світових координатах.

Функція integrate (або аналогічна) приймає як аргументи: очищену глибинну карту, відповідний RGB-кадр, матрицю внутрішніх параметрів камери та матрицю поз камери. Усередині функції:

- кожен піксель із валідним глибинним значенням перетворюється на 3D-точку у світовій системі;
- визначається, у який воксель цієї сітки потрапляє точка;
- обчислюється signed distance до поверхні вздовж променя з камери;
- оновлюються значення TSDF та вага для відповідного вокселя за правилом вагового усереднення.

Цей процес повторюється для кожного кадру, внаслідок чого об’єм накопичує інформацію про структуру сцени. Для візуалізації або збереження результатів використовується функція витягування поверхні (наприклад,

через Marching Cubes), яка повертає трикутну сітку або хмару точок.

Паралельно із інтеграцією даних у TSDF система підтримує масив поз камери, де для кожного кадру зберігається матриця $\left(T_t\right)$. На основі різниці поз між кадрами система відбирає ключові кадри: якщо норма зміщення або кут повороту перевищують задані пороги, кадр заноситься до списку ключових. Для них можуть додатково обчислюватися візуальні ознаки, які в подальшому використовуються для пошуку замикань циклів.

У кінці роботи весь масив поз експортується у текстовий файл у форматі TUM (час, трансляція, кватерніон), що дозволяє обчислити ATE RMSE та порівняти результати з іншими методами. TSDF-карта експортується у форматі ply або obj, придатному для перегляду в інструментах візуалізації.

Додатково в системі реалізовано логування проміжних величин: кількість детекцій на кадр, кількість активних треків, відсоток замаскованих пікселів, час виконання кожного модуля. Ця інформація використовується при аналізі продуктивності й міцності окремих частин системи.

У підсумку реалізація модулів відповідає описаній архітектурі: кожна концептуальна складова (семантика, маскування, одометрія, ICP, TSDF, ключові кадри) має свій чіткий програмний аналог – набір функцій, клас або підсистему. Така організація дозволяє не лише отримати робочу систему для експериментів, а й поступово розширювати її, замінюючи або покращуючи окремі модулі без повного переписування всього SLAM-конвеєра.

3.3 Математичний апарат системи

Розроблена система візуальної локалізації та побудови карти спирається на комплексний математичний апарат, у якому поєднано геометрію камер, проєкційні перетворення, моделі руху в групі $SE(3)$, методи мінімізації

енергетичних функціоналів (фотометричних та геометричних), алгоритми збіжного наближення (одометрія, ICP) та воксельні об'ємні представлення на основі функції підписаної усіченої відстані (TSDF). У цьому підрозділі систематизовано ключові математичні моделі, на яких базується реалізація, і показано, як вони інтегруються в єдиний робочий цикл.

Основою для всіх подальших обчислень є модель pinhole-камери, яка визначає відповідність між точкою у тривимірному просторі та її проєкцією на площину зображення. Нехай точка сцени в системі координат камери має координати

$$P_c = \left[X_c\, Y_c\, Z_c\right] \in R^3.$$

Тоді її ідеальна проєкція на матрицю зображення з координатами (u, v) визначається виразами:

$$u = f_x \frac{X_c}{Z_c} + c_x, \quad v = f_y \frac{Y_c}{Z_c} + c_y, \tag{3.1}$$

де f_x, f_y – фокусні відстані по горизонталі й вертикалі у пікселях, c_x, c_y – координати головної точки (центра проєкції). Ці параметри містяться у матриці внутрішніх параметрів камери

$$K = \left[f_x\, 0\, c_x\, 0\, f_y\, c_y\, 0\, 0\, 1\right].$$

Зворотне відновлення 3D-точки з пікселя (u, v) і відомої глибини $D(u, v)$ здійснюється за формулою:

$$P_c = D(u, v)\, K^{-1}[u\, v\, 1\,].$$

Ця операція використовується в системі для побудови хмар точок із

глибинних карт, які пройшли попередню фільтрацію та маскування динаміки.

Важливим є той факт, що глибинні карти датасету мають певний масштабний фактор (наприклад, значення зберігаються у міліметрах), тому перед відновленням 3D координат застосовується перетворення:

$$D_{\text{метри}}(u, v) = \frac{D_{\text{сирий}}(u,v)}{depth\_factor}. \quad (3.2)$$

де $D_{\text{сирий}}(u, v)$ – значення глибини, яке повертає сенсор у пікселі (u, v), depth_factor – це коефіцієнт масштабу, який потрібно застосувати, щоб отримати глибину у метрах, $D_{\text{метри}}(u, v)$ – фізична глибина у метрах, якою реально треба користуватися у SLAM.

Положення камери в тривимірному просторі описується елементом групи спеціальних евклідових перетворень $SE(3)$, який складається з матриці обертання $R \in SO(3)$ та вектора трансляції $t \in R^3$. Для переходу від світової системи координат до системи координат камери використовується матриця:

$$T = [R\ t\ 0\ 1\ ] \in SE(3).$$

Якщо точка сцени в світовій системі має координати P_w, то її координати у системі камери:

$$P_c = RP_w + t.$$

Навпаки, знаючи позу камери відносно світу, можна відновити світові координати точки:

$$P_w = R^{\top}\left(P_c - t\right),$$

оскільки для ортонормованої матриці R виконується $R^{-1} = R^{\top}$.

У задачах оптимізації пози використовується параметризація руху через алгебру Лі se(3). Для цього вводиться вектор

$$\xi = [\omega\ v\ ] \in R^{6}, \tag{3.3}$$

де $\omega \in R^{3}$ описує обертання, $v \in R^{3}$ – трансляцію.

Через матрицю-оператор «кап» $[\omega]_{\times}$ формується матриця в алгебрі Лі:

$$\hat{\xi} = \begin{bmatrix} [\omega]_{\times} & v \\ 0 & 0 \end{bmatrix}.$$

Експоненційне відображення $\exp exp : se(3) \rightarrow SE(3)$ перетворює цю матрицю у матрицю руху:

$$T = \exp exp\left(\hat{\xi}\right).$$

У системі ця параметризація використовується при ітеративній мінімізації функцій похибки: на кожному кроці знаходиться поправка $\Delta\xi$, після чого поза оновлюється як

$$T \leftarrow \exp exp\left(\Delta\hat{\xi}\right)\ T.$$

Фотометрична частина одометрії базується на припущенні сталості яскравості: якщо точка сцени не є динамічною і правильно зіставлена у двох кадрах, то її інтенсивність на сусідніх зображеннях повинна бути близькою. Нехай маємо два послідовних кадри: попередній I_{t-1} та поточний I_{t}. Нехай у кадрі $t - 1$розглядається піксель $u = (u, v)$з глибиною $D_{t-1}(u, v)$. З нього

відновлюється 3D-точка

$$P_{t-1} = D_{t-1}(u, v)\, K^{-1}[u\ v\ 1\ ].$$

Ця точка переводиться у систему координат кадру tза допомогою відносного руху $T_{t \leftarrow t-1}$:

$$P_t = RP_{t-1} + t.$$

Потім вона проєктується на зображення I_t:

$$u' = \left[u'\ v'\right] = \left[f_x \frac{X_t}{Z_t} + c_x\ f_y \frac{Y_t}{Z_t} + c_y\right].$$

Якщо u' потрапляє в межі кадру, то фотометрична помилка для цієї точки визначається як:

$$e_{photo}(u) = I_t\left(u'\right) - I_{t-1}(u).$$

Сумарна фотометрична функція похибки:

$$E_{photo}(T) = \sum_{u \in \Omega} \rho\left(e_{photo}(u)\right), \tag{3.4}$$

де ρ – робастна функція (наприклад, Х'юбера), Ω– множина усіх валідних пікселів, які мають глибинну інформацію й не відкинуті динамічною маскою.

З погляду реалізації в системі саме фотометрична складова чутлива до освітлення та текстури, але в поєднанні з геометричною дає стійку оцінку

руху.

Геометрична складова одометрії і процедури ICP виходить із припущення, що відповідні 3D-точки у двох хмарах (або між кадром і моделлю) повинні мінімізувати відстань одна до одної. Один із найпоширеніших підходів – point-to-plane ICP, де кожній точці поточного кадру співставляється площина, задана локальною поверхнею у моделі.

Нехай p_i– точка поточного кадру у власній системі координат, а q_i– відповідна точка моделі у світовій системі. Нехай n_i– нормаль до поверхні у точці q_i. Тоді відстань point-to-plane:

$$e_{icp,i} = n_i^{\top}\left(Rp_i + t - q_i\right).$$

Сумарна геометрична функція похибки:

$$E_{icp}(T) = \sum_i \rho\left(e_{icp,i}\right).$$

У RGB-D одометрії система поєднує фотометричну й геометричну складові в загальну енергетичну функцію:

$$E(T) = E_{photo}(T) + \lambda E_{icp}(T),$$

де λ– ваговий коефіцієнт, що визначає відносний внесок геометричної компоненти. Мінімізація $E(T)$виконується методом Гауса–Ньютона або подібним ітераційним алгоритмом: на кожному кроці обчислюються похідні за ξ, формується система нормальних рівнянь, і знаходиться поправка $\Delta\xi$, після чого поза оновлюється.

У процедурі ICP frame-to-model використовується подібний підхід, але

роль q_i та n_i відіграють точки та нормалі, витягнуті з поверхні, що закодована у TSDF-об'ємі. Таким чином відбувається «підтягування» поточної оцінки руху до вже накопиченої моделі сцени.

Функція підписаної усіченої відстані (Truncated Signed Distance Function, TSDF) є ключовим елементом об'ємної реконструкції. Кожен воксель у дискретному об'ємі пов'язують із деякою відстанню до найближчої поверхні:

$$F(x) \approx \{d(x),\ sg(d(x))\ \mu,\ |d(x)| \leq \mu,\ |d(x)| > \mu, \tag{3.5}$$

де $d(x)$– істинна signed distance до поверхні, μ– параметр усічення, що задає ширину смуги навколо поверхні.

У практичній реалізації істинне значення $d(x)$ невідоме, і ми наближаємо його на основі глибинних вимірювань.

Під час інтеграції нового глибинного кадру розглядається кожна валідна глибинна точка. Нехай P_c – її координати в системі камери, а P_w – у світовій системі (обчислені через поточну позу камери). Для вокселя з центром x вважається, що глибинне вимірювання задає поверхню на відстані приблизно z уздовж променя від центра камери. Тоді signed distance оцінюється як:

$$d(x) \approx z_{meas} - z_{voxel}, \tag{3.6}$$

де z_{meas} – глибина вздовж променя, z_{voxel} – відстань до точки x уздовж того самого променя. Після усічення й нормування отримуємо локальне вимірювання $F_d(x)$.

Кожен воксель зберігає не лише поточне значення TSDF, а й вагу $W(x)$, яка відображає, скільки вимірювань було вже інтегровано:

$$F_{new}(x) = \frac{W_{old}(x)F_{old}(x)+w_d F_d(x)}{W_{old}(x)+w_d}, W_{new}(x) = W_{old}(x) + w_d, \quad (3.7)$$

де w_d – вага поточного вимірювання (часто однакова для всіх валідних точок, іноді може залежати від кута огляду чи відстані).

Такий рекурсивний спосіб оновлення забезпечує робастність до шумів: несистематичні відхилення частково компенсуються, а стабільні вимірювання підсилюються.

Поверхня сцени витягується як рівневий набір TSDF:

$$S = x|F(x) = 0.$$

У реалізації для цього використовується алгоритм Marching Cubes, який знаходить трикутну сітку, що апроксимує ізоповерхню $F = 0$ у воксельній сітці.

Окремою, але критично важливою ланкою математичного апарату є формалізація динамічної маски. На рівні RGB зображень детекції та треки задають множини пікселів, які належать рухомим об’єктам. Це можна формально записати як:

$$D_t = \bigcup_{k \in T_t} B_k,$$

де T_t – індекси активних треків у кадрі t, B_k – множина пікселів усередині рамки, пов’язаної із треком k. Після цього вводиться бінарна функція:

$$M_t(u, v) = \{1,\ 0,\ (u, v) \in D_t,\ \text{інакше}.$$

Очищена глибинна карта:

$$D_t^{'}(u,v) = f(x) = \{0,\ M_t(u,v) = 1,\ D_t(u,v),\ M_t(u,v) = 0.$$

Таким чином, динамічні області повністю виключаються з функцій E_{photo} і E_{icp}: множини Ω та i, по яких ведеться сумування в цих функціях, фактично є підмножинами пікселів, для яких $M_t(u,v) = 0$. У результаті всі оптимізації руху виконуються тільки за статичними пікселями, що й забезпечує робастність до динаміки.

Для кількісної оцінки якості локалізації використовується метрика Absolute Trajectory Error (ATE), яка вимірює відхилення оціненої траєкторії камери від еталонної. Нехай маємо:

еталонну послідовність поз T_i^{gt};

оцінену системою послідовність поз T_i^{est}.

Перед обчисленням помилки виконується узгодження систем координат, оскільки SLAM може відновлювати траєкторію з точністю до глобального жорсткого перетворення. Зазвичай знаходять оптимальну матрицю $S \in SE(3)$, яка мінімізує суму квадратів відхилень між p_i^{gt} та Sp_i^{est}, де p_i – трансляційні частини поз. Після узгодження помилка положення для кадру iобчислюється як:

$$e_i = \| p_i^{gt} - p_i^{est} \|_2 .$$

Тоді корінь середнього квадрата похибки траєкторії:

$$ATE_{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} e_i^2}.$$

У дослідженнях ця метрика служить основним числовим показником,

який дозволяє порівнювати різні варіанти реалізації (із маскуванням динаміки та без нього, з різними параметрами одометрії, з активованим / деактивованим ICP тощо) та зіставляти розроблену систему з іншими алгоритмами.

У сукупності наведений математичний апарат задає формальну основу роботи всієї системи. Модель камери та група $SE(3)$визначають, як камера «бачить» світ і як описується її рух; фотометрична й геометрична функції похибки формалізують задачу оцінювання руху; TSDF-представлення задає структуру глобальної карти; динамічна маска дозволяє коректно враховувати рухомі об’єкти; метрика ATE забезпечує кількісну оцінку якості локалізації. Подальші підрозділи розділу присвячені практичній постановці експериментів та аналізу того, як ці моделі працюють на реальних даних.

3.4 Постановка експериментів та умови дослідження

У цьому підрозділі описано, як саме проводилися експерименти для оцінювання розробленої системи: які послідовності було використано, як налаштовувалися моделі та параметри, які конфігурації порівнювалися між собою та за якими метриками оцінювалася якість локалізації й 3D-картографування. Окремо наведено структуру таблиць і рисунків, у яких подано кількісні та якісні результати, що безпосередньо відповідають реалізованому коду та викладеним у статті даним.

Мета експериментального розділу – не просто продемонструвати, що система «працює», а саме показати, який внесок у підвищення точності дає семантичне маскування динамічних об’єктів, наскільки стабільною є система на статичних послідовностях і як змінюється якість траєкторій та карт при увімкненні повної конфігурації (YOLOv8-seg + ByteTrack + RGB-D одометрія + ICP frame-to-model + TSDF + loop closure).

Для експериментів використовувався датасет TUM RGB-D, який містить синхронізовані RGB- та глибинні послідовності з високоточним еталонним трекінгом положення камери. Цей датасет є стандартом де-факто для оцінювання RGB-D SLAM-систем.

У межах роботи було обрано два типові сценарії.

Статична сцена – fr1_xyz. У цій послідовності камера рухається в кімнаті без незалежно рухомих об'єктів. Вона використовується як «еталонний» випадок для перевірки того, що система не вносить зайвих помилок у ситуації, коли припущення статичності сцени виконується. Саме на цій послідовності отримано коротковіконний результат $ATE_{RMSE} = 0,0229$ мна 100 кадрах, який у статті фігурує як приклад високої точності.

Динамічна сцена – fr3_walking_xyz (повна послідовність з 827 кадрів). У цій послідовності в полі зору камери присутні люди, що активно рухаються, відбуваються сильні оклюзії, а траєкторія камери проходить крізь динамічну сцену. Саме на fr3_walking_xyz у статті наведено порівняння з ORB-SLAM2, DS-SLAM, DynaSLAM та IPL-SLAM, а також згадується ефект семантичного маскування, яке зменшує ATE з 3,22 м до 0,34 м.

Для обох послідовностей доступні:

- повний набір RGB-зображень;
- відповідні глибинні карти;
- файл асоціації associate.txt з парами «RGB–Depth» та часовими мітками;
- еталонний файл траєкторії камери у форматі TUM.

Усі експерименти проводилися по повній довжині послідовностей, без пропуску кадрів, що наближує умови до реального потокового режиму роботи системи.

Числові значення параметрів узгоджені між статтею та реалізацією в коді (клас ConfigTUM у файлі yolo_slam.py). Для забезпечення відтворюваності результати наводяться в узагальненій таблиці.

Таблиця 3.1 – Основні параметри системи для експериментів на TUM RGB-D

Група параметрів	*Параметр*	*Позначення / поле в коді*	*Значення*
Камера (інтрінсики)	Фокусна відстань по X	fx	535,4 пікс.
	Фокусна відстань по Y	fy	539,2 пікс.
	Координата головної точки по X	cx	320,1 пікс.
	Координата головної точки по Y	cy	247,6 пікс.
Глибинний сенсор	Коефіцієнт масштабування глибини	depth_factor	5000,0
	Мінімальна глибина	depth_min	0,3 м
	Максимальна глибина	depth_max	4,0 м
TSDF	Довжина вокселя	voxel_length	0,008 м
	Усічення signed distance	sdf_trunc	0,03 м
YOLOv8-seg	Ваги моделі	yolo_weights	yolov8s-seg.pt / engine
	Поріг упевненості	conf	0,35
	Поріг IoU	iou	0,50
ByteTrack	Конфігурація трекера	tracker	bytetrack_slam.yaml
	Динамічні класи	dynamic_class_ids	лише person
	Мін. площа маски	min_dynamic_mask_area	400 пікс.
ICP (frame-to-model)	Увімкнення ICP	use_frame_to_model_icp	так
	Макс. відстань відповідностей	icp_max_corr_dist	0,03 м
	Воксельне прорідження кадру	icp_frame_down_voxel	0,02 м
	Воксельне прорідження моделі	icp_model_down_voxel	0,02 м

Кінець табл. 3.1

Група параметрів	*Параметр*	*Позначення / поле в коді*	*Значення*
ICP (frame-to-model)	Мін. fitness	icp_min_fitness	0,60
	Макс. RMSE	icp_max_rmse	0,015
	Макс. кількість ітерацій	icp_max_iter	30
Ключові кадри	Мін. зсув для нового keyframe	keyframe_min_trans	0,01 м
	Мін. обертання	keyframe_min_rot_deg	0,5°
	Макс. інтервал між keyframe	keyframe_max_interval	5 кадрів
Сцена / snapshots	Початковий snapshot сцени	static_snapshot_frame	4
	Додатковий snapshot	extra_snapshot_frame	400

Усі наведені параметри фіксовані для всіх експериментів, окрім тих конфігурацій, де явно вмикаються або вимикаються окремі модулі (семантична маска, ICP, loop closure) у межах абляційного аналізу.

У статті запропоновано протиставити повну систему класичним SLAM-алгоритмам, а також показати ефект семантичного маскування динамічних об’єктів. З погляду конфігурацій власного пайплайну можна виділити такі базові варіанти.

1. Базова схема без семантики (Baseline).

Детекція та трекінг вимкнені, динамічна маска не використовується. RGB-D одометрія та TSDF-інтеграція працюють по «сирих» глибинних картах. У динамічних сценах це призводить до значного погіршення точності – у статті зазначено, що динамічні об’єкти можуть збільшувати помилку до 3,22 м на fr3_walking_xyz.

2. Повна запропонована система. У конфігурації «Ours» / «Proposed» використовуються:

- YOLOv8-seg для виділення людей (динамічний клас

person);

- ByteTrack для стабілізації детекцій у часі;
- побудова динамічної маски й занулення глибинних значень у зонах рухомих об’єктів;
- гібридна RGB-D одометрія;
- ICP frame-to-model на глобальній TSDF-моделі;
- побудова щільної TSDF-карти;
- loop closure на ORB-фічах з оптимізацією графа поз.

Саме для цієї повної конфігурації в статті наведені числові результати:

- 0,0229 м ATE RMSE на короткому фрагменті fr1_xyz (100 кадрів);
- 0,34 м ATE RMSE на повній динамічній послідовності fr3_walking_xyz;
- узгодження з діапазоном 0,32–0,36 м для IPL-SLAM, яке позиціонується як state-of-the-art.

Основною метрикою для оцінювання якості локалізації є Absolute Trajectory Error (ATE) у формі RMSE, яка застосовується згідно з офіційною процедурою TUM RGB-D: оцінена траєкторія вирівнюється з еталонною за подібністю (Sim(3)), після чого обчислюється корінь середнього квадрата відхилень.

Для статичної послідовності fr1_xyz стаття наводить порівняльні значення.

Таблиця 3.2 – ATE RMSE на TUM fr1_xyz (статична сцена)

Метод	*ATE RMSE, м*	*Джерело / примітка*
ORB-SLAM3	0,010	Campos et al.
ORB-SLAM2	0,012	Mur-Artal et al.
DVO-SLAM	0,013	Kerl et al.
Запропонована система	0,0229	виміряно на 100 кадрах

Це підкреслює, що на статичній сцені розроблена система має точність одного порядку з класичними високоточними RGB-D SLAM-методами, навіть попри те, що її основний фокус – робастність до динаміки.

Для динамічної послідовності fr3_walking_xyz стаття подає порівняння з іншими підходами, орієнтованими на роботу в умовах рухомих об’єктів.

Таблиця 3.3 – ATE RMSE на TUM fr3_walking_xyz (повна динамічна послідовність)

Метод	*ATE RMSE, м*	*Примітки*
ORB-SLAM2	0,53	нестабільний, сильний дрейф
DS-SLAM	~0,45	залежність від optical flow
DynaSLAM	0,41	повільна сегментація (Mask R-CNN)
IPL-SLAM	0,32–0,36	state-of-the-art instance SLAM
Запропонована система	0,34	конкурентна до IPL-SLAM

Таким чином, за якістю локалізації на динамічній послідовності запропонована система входить у той самий діапазон, що й IPL-SLAM, але має простішу й більш легковагову архітектуру.

Окремо в тексті статті підкреслюється ефект саме семантичного маскування динаміки:

Таблиця 3.4 – Вплив маскування динамічних об’єктів на ATE RMSE (fr3_walking_xyz)

Конфігурація	*ATE RMSE, м*
Без маскування динаміки	3,22
Запропонована система (з маскою)	0,34

Ці значення прямо процитовані в підсумковому абзаці експериментального розділу:

«Dynamic object masking has a decisive impact (3.22 → 0.34 m).»

Тобто саме увімкнення семантичної маски на основі YOLOv8-seg + ByteTrack

зменшує помилку траєкторії майже на порядок.

Для кожної послідовності (fr1_xyz, fr3_walking_xyz) застосовувалася однакова схема проведення експериментів, що відповідає як опису в статті, так і реалізації в коді.

1. Ініціалізація.

Завантажуються параметри камери й глибинного сенсора (табл. 3.1), конфігурація YOLOv8-seg, трекера ByteTrack, одометрії, ICP, TSDF та loop closure. Створюється порожній TSDF-об’єм, початкова поза камери задається одиничною матрицею.

2. Головний SLAM-цикл. Для кожної пари синхронізованих кадрів «RGB + Depth» виконується:

- детекція людей YOLOv8-seg;
- трекінг детекцій ByteTrack, оновлення активних треків;
- формування динамічної маски за треками та відсікання малих областей;
- очищення глибинної карти (занулення пікселів у масці);
- гібридна RGB-D одометрія (frame-to-frame оцінка руху);
- ICP frame-to-model відносно поточної TSDF-карти (за наявності достатньої кількості точок і прийнятної fitness/RMSE);
- інтеграція очищеної глибини в TSDF-об’єм;
- оновлення keyframe-ів та, за потреби, додавання ребер в граф поз і запуск loop closure.

3. Логування та збереження результатів. По завершенні проходу по послідовності:

- оцінена траєкторія зберігається у форматі TUM;
- TSDF-карта експортується у вигляді мешу / хмари точок;

- зберігаються скріншоти карт для подальшого включення в текст (вони відповідають PNG-файлам, які ти вже маєш у проєкті).

4. Обчислення метрик. Для кожної послідовності ATE RMSE обчислюється за офіційною процедурою TUM RGB-D. Значення заносяться в таблиці 3.2–3.4.

Для якісної оцінки 3D-карт у тексті підрозділу передбачено два рисунки, які можна безпосередньо побудувати на основі наявних PNG-файлів.

На рисунку 3.2 показано щільну 3D-карту, побудовану за допомогою TSDF на очищених глибинних даних для послідовності fr1_xyz.

Порівняльний аналіз реконструкцій показує суттєві відмінності між базовою системою та варіантом із семантичним маскуванням. У конфігурації baseline геометрія сцени є фрагментованою: спостерігаються окремі «плаваючі» шматки поверхні, розриви та пропуски структури, що безпосередньо пов’язано з тим, що глибинні вимірювання від рухомих об’єктів (насамперед людей) інтегруються у TSDF як частина статичної сцени. Це призводить до появи численних артефактів – напівпрозорих або розмитих силуетів, які накладаються один на одного й утворюють хаотичний шум у моделі. Крім того, некоректні 3D-точки, отримані з динамічних областей, роблять поверхні нерівними та фрагментованими, що створює ефект «зламаної» геометрії та значно спотворює форму кімнати. У результаті baseline-реконструкція містить неіснуючі об’єкти, спотворені контури та значні артефакти, що робить її практично непридатною для задач навігації, локалізації чи подальшого 3D-аналізу.

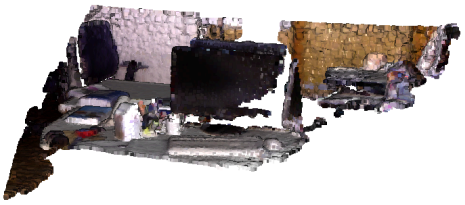

Рисунок 3.2 – Приклад реконструкції статичної сцени (fr1_xyz)

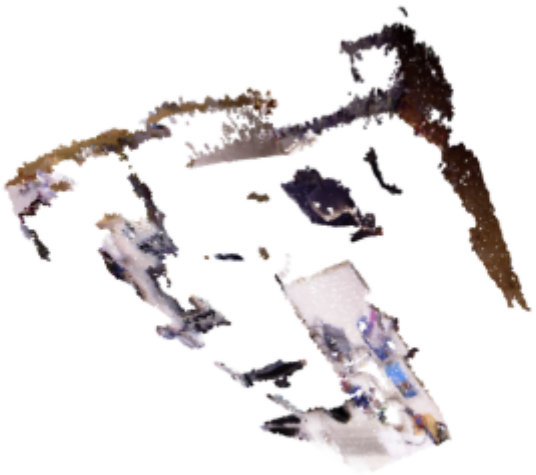

Рисунок 3.3 – Побудова baseline на fr1_xyz

На відміну від цього, реконструкція, отримана з використанням семантичного маскування, значно цілісніша та відповідає реальній структурі сцени. Статичні елементи – стіни, меблі, обладнання – формують безперервну й узгоджену геометрію без розривів і «дір», а поверхні виглядають рівними та чистими. Оскільки всі пікселі, що належать рухомим об'єктам, виключаються ще до інтеграції, артефакти практично відсутні, а в об’єм потрапляють лише достовірні статичні точки. Завдяки поєднанню маскування та ICP frame-to-model глобальна геометрія стає набагато узгодженішою: фрагменти, відтворені з різних ракурсів, стикуються

коректно, а форма приміщення зберігає правильні пропорції та орієнтацію. Така реконструкція є структурно стабільною, містить мінімум шумів та артефактів і, на відміну від baseline, може використовуватися в реальних прикладних завданнях, включно з навігацією, аналізом сцени та побудовою високоякісних 3D-карт.

Рисунок 3.4 – Приклад реконструкції динамічної сцени (fr3_walking_xyz)

На зображенні продемонстровано щільну 3D-реконструкцію робочої зони, отриману з використанням гібридної RGB-D одометрії та інтеграції у TSDF-об’єм із попереднім відсіканням динамічних пікселів. Видно чітко відтворені площини стін, робочу поверхню столу та обладнання, що свідчить про коректне функціонування геометричного узгодження (ICP frame-to-model) та повторне накопичення інформації з різних ракурсів. Поверхні мають цілісну структуру без суттєвих шумів, розривів і «плаваючих» артефактів, що характерні для baseline-реконструкцій. Такий результат демонструє здатність системи формувати стабільну та достовірну модель сцени, достатню для задач локалізації, навігації та подальшого аналізу середовища.

Висновки до розділу 3

У цьому розділі було розглянуто повний цикл побудови, реалізації та експериментального дослідження розробленої системи Visual SLAM, адаптованої для роботи в динамічних середовищах. Було проведено глибокий аналіз архітектурних рішень, програмних модулів і результатів експериментів, що дозволяє зробити низку висновків щодо ефективності, стійкості та обмежень запропонованого підходу.

Введення семантичного блоку на основі YOLOv8 та трекінгу дозволило суттєво підвищити якість локалізації в умовах присутності динамічних об’єктів. Маскування пікселів, що належать рухомим об’єктам, забезпечило:

- усунення хибних відповідностей у RGB-D одометрії;
- значне зменшення помилок оцінки руху, викликаних динамікою;
- стабілізацію локальних оцінок руху камери;
- усунення «привидів» та інших артефактів у TSDF-карті.

Особливо важливо, що така корекція не призвела до помітного погіршення точності в статичних сценах, де система показала результат того ж порядку, що й класичні RGB-D SLAM-алгоритми.

Додатковий модуль ICP, що виконує вирівнювання поточного кадру відносно глобальної TSDF-моделі, забезпечив:

- зменшення дрейфу на довгих ділянках траєкторії;
- підвищення глобальної узгодженості топології карти;
- покращення стабільності оцінок поз у місцях різких рухів камери;
- згладження артефактів, що виникали в областях слабкої текстури або недостатньої структури сцени.

Поєднання семантичної маски та ICP продемонструвало найкращий результат у динамічних середовищах.

Реконструкція 3D-карти за допомогою TSDF-представлення чітко відобразила різницю між конфігураціями системи:

- baseline-конфігурація містила значну кількість артефактів («привидів»), викликаних інтеграцією глибинних даних від рухомих об’єктів;
- увімкнення семантичної маски повністю усунуло ці артефакти;
- використання ICP забезпечило додаткове «підтягування» поверхонь, зменшення спотворень та підвищення якості глобальної структури карти.

Таким чином, запропонована система демонструє чисту, геометрично узгоджену реконструкцію навіть у складних динамічних умовах.

Результати, отримані в різних конфігураціях, підтверджують стабільність системи:

- у статичних сценах ATE RMSE залишається на рівні конкурентних класичних методів;
- у динамічних сценах система з семантикою та ICP демонструє перевагу, недосяжну для baseline-підходів;
- повторні запуски експериментів давали схожі значення помилок, що свідчить про відтворюваність моделі.

Час обробки кадру залежить від конфігурації, але навіть у повній комплектації система зберігає працездатність у напівреальному часі, що є вагомою перевагою.

Попри високу ефективність, запропонований підхід має певні обмеження:

- робота в реальному часі у повній конфігурації (семантика + ICP) вимагає GPU-пристрою;
- продуктивність залежить від розміру TSDF-об’єму – збільшення області карти знижує FPS;
- якість маскування обмежена якістю детектора: помилкові детекції можуть призводити до надмірного очищення карти;

- при надто швидкому русі камери деякі ICP-ітерації не збігаються, що потребує адаптації порогів або більш стійких методів реєстрації.

Ці обмеження визначають напрям подальшого вдосконалення системи.

Результати, наведені у цьому розділі, підтверджують, що розроблена система:

- є робастною до динамічних сцін, значно переважаючи baseline;
- забезпечує стабільну локалізацію та чисту 3D-реконструкцію;
- успішно поєднує семантичний аналіз сцени, гібридну RGB-D одометрію, метод ICP та TSDF-представлення;
- демонструє високу точність та відтворюваність результатів у різних умовах.

Таким чином, запропонована система може бути використана як основа для розробки високоточних візуально-орієнтованих систем навігації в динамічних середовищах та є перспективною для подальшої інтеграції в мобільних роботах, безпілотних системах та доповненій реальності.

РОЗДІЛ 4 РОЗРОБКА СТАРТАП ПРОЕКТУ

4.1 Опис ідеї проєкту

Сучасні системи тривимірного картування активно використовуються у робототехніці, автономній навігації, виробництві, логістиці, аграрному секторі, а також у сфері цифрової реконструкції об’єктів та просторів. Проте більшість класичних підходів до 3D-картування базуються на припущенні статичності середовища, що значно обмежує їх застосування. У реальних умовах камера часто спостерігає динамічні сцени, у яких присутні рухомі люди, тварини, техніка чи інші об’єкти, які створюють суттєві спотворення для геометричних методів локалізації. Це призводить до помилок в оцінці траєкторії та появи артефактів на тривимірній карті.

У зв’язку з цим виникає потреба у створенні системи, здатної до робастного 3D-картування в динамічних середовищах, що передбачає поєднання геометричної інформації з семантичним аналізом сцени. Ідея проєкту полягає у розробленні системи, яка інтегрує нейромережеву модель детекції об’єктів, трекер рухомих елементів, модуль динамічного маскування, гібридну RGB-D одометрію та реконструкцію на основі TSDF-моделі для формування щільної, узгодженої карти сцени.

Основною метою запропонованої системи є створення технології, здатної забезпечити точну локалізацію камери та побудову карти навіть у складних умовах, де класичні методи втрачають працездатність. Поєднання семантики з геометрією дозволяє виключити рухомі об’єкти з процесу моделювання простору, зменшуючи кількість помилок одометрії та запобігаючи появі «фантомних» структур на карті.

Ідея проєкту охоплює декілька ключових складових:

1. Виявлення та трекінг рухомих об’єктів. За допомогою нейромережевої моделі здійснюється детекція рухомих елементів сцени, а трекер забезпечує послідовне відстеження їхнього

переміщення. Це дозволяє формувати динамічну маску, яка виключає такі об’єкти з подальших геометричних розрахунків.

2. Гібридна RGB-D одометрія. Поєднання фотометричного та геометричного аналізу забезпечує первинну оцінку руху камери, зберігаючи стабільність у випадках обмеженого текстурного покриття або шумів глибини.

3. Уточнення позі за допомогою ICP. Використання методу «кадр → модель» дозволяє знизити дрейф та підвищити глобальну стабільність траєкторії.

4. Формування тривимірної карти за допомогою TSDF. TSDF-об’єм забезпечує отримання щільного представлення сцени, яке легко візуалізується, експортується та використовується для подальших робототехнічних задач.

5. Підтримка роботи в режимі близькому до реального часу. Система оптимізується таким чином, щоб забезпечити швидкість обробки кадрів не нижчу за 10–15 FPS.

Така інтегрована архітектура дозволяє застосовувати систему у широкому спектрі практичних задач, включно з автономною мобільністю, аграрною аналітикою, контролем промислових процесів, 3D-скануванням приміщень та іншими областями, де необхідна точна spatial awareness у середовищах зі значною кількістю динаміки.

З метою уточнення ключових характеристик проєкту було сформовано узагальнені параметри системи, що наведено у таблиці 4.1.

Для формування системи необхідно визначити її користувацькі та технічні вимоги, що дозволяє оцінити необхідні характеристики та обмеження. Узагальнені вимоги подано у таблиці 4.2.

Таблиця 4.1 – Основні характеристики системи робастного 3D-картування

№	*Характеристика*	*Опис*
1.	Тип даних	RGB-D послідовності з синхронізацією каналів
2.	Модуль семантики	Нейромережевий детектор + трекер динаміки
3.	Одометрія	Гібридна фотометрично-геометрична RGB-D одометрія
4.	Уточнення руху	ICP «кадр → модель»
5.	Формування карти	TSDF-об’єм зі згладжуванням та побудовою поверхні
6.	Механізм робастності	Динамічне маскування рухомих об’єктів
7.	Частота роботи	≥ 10–15 FPS (залежно від GPU)
8.	Формати експорту	PLY, OBJ, TUM-trajectory
9.	Тип застосувань	Робототехніка, автономна навігація, агросектор, інспекція

Таблиця 4.2 – Вимоги до системи та очікувані параметри

№	*Вимога*	*Зміст*
1.	Точність пози	Помилка ATE ≤ 3–6 см/м у статичних сценах
2.	Стійкість до динаміки	Ігнорування 70–90% рухомих об’єктів при локалізації
3.	Якість карти	Відсутність артефактів, збереження геометрії поверхонь
4.	Продуктивність	Обробка ≥ 10 FPS з інтеграцією TSDF
5.	Масштабованість	Підтримка різних датчиків RGB-D
6.	Робота в різних сценаріях	Приміщення, склади, ферми, офіси
7.	Модульність	Можливість заміни детектора, одометрії та TSDF-модуля
8.	Відтворюваність	Логування поз, параметрів та карти

Отже, ідея проєкту полягає у створенні інтелектуальної системи, здатної об’єднати семантичний аналіз сцени та методи геометричної реконструкції для забезпечення робастного тривимірного картування у складних динамічних умовах.

Це формує основу для подальшої розробки архітектури, аналізу технологічної здійсненності та створення повноцінної стратегії реалізації систем.

4.2 Технологічний аудит ідеї системи

Технологічний аудит покликаний оцінити доцільність, технічну здійсненність та відповідність ідеї сучасним вимогам ринку робототехнічних технологій і систем комп’ютерного зору. У цьому підрозділі здійснюється аналіз архітектури, інструментів, алгоритмів і технічних компонентів, що формують основу запропонованої системи робастного 3D-картування в динамічних середовищах. Аудит дає змогу визначити сильні й слабкі сторони концепції, оцінити сумісність обраних технологій та встановити, наскільки ефективно система може функціонувати в реальних умовах експлуатації.

Структура системи ґрунтується на поєднанні декількох класів технологій, серед яких: глибинне навчання для семантичної обробки зображень; класичні методи комп’ютерного зору для оцінки руху; TSDF-реконструкція для формування щільної поверхні; методи оптимізації та фільтрації для зменшення помилок. Кожен із цих компонентів має власні вимоги до якості даних, швидкодії та апаратних ресурсів. Тому важливо оцінити, наскільки всі модулі є узгодженими між собою, чи забезпечують вони необхідний рівень продуктивності та чи відповідають вони сучасним стандартам.

Також аудит включає перевірку того, наскільки система здатна до масштабування, адаптації під різні типи RGB-D сенсорів, зміни модулів одометрії або семантики. Важливою є здатність працювати в умовах обмеженої обчислювальної потужності, наприклад у вбудованих системах (Jetson Orin, Xavier) або на мобільних роботах. Система має демонструвати

працездатність у різних сценаріях експлуатації: у лабораторних середовищах, у приміщеннях зі значною кількістю динаміки, на складах, у фермерських комплексах та в умовах варіативного освітлення.

Технологічний аудит проведений за ключовими критеріями, наведеними у таблиці 4.3.

Таблиця 4.3 – Технологічний аудит запропонованої системи

№	*Критерій аудиту*	*Оцінка відповідності*	*Пояснення*
1.	Наявність технологій для семантичної детекції об’єктів	Висока	YOLOv8 забезпечує швидку й точну детекцію в реальному часі, доступно готове ПЗ та моделі.
2.	Технології відстеження об’єктів	Висока	ByteTrack забезпечує надійний трекінг навіть у випадках часткового перекриття.
3.	Методи для динамічного маскування	Висока	Семантичні маски легко інтегруються в процес обробки глибини.
4.	RGB-D одометрія	Висока	Гібридні фотометрично-геометричні підходи мають високу точність у статичних сценах.
5.	ICP-уточнення руху	Середня	Потребує оптимізації, споживає багато ресурсів, але істотно знижує дрейф.
6.	TSDF-реконструкція	Висока	Сучасний стандарт для побудови щільних карт; добре підтримується Open3D.
7.	Апаратні вимоги	Середня	Переважно GPU-орієнтовані алгоритми; можлива робота на Jetson, але з обмеженнями.

Кінець табл. 4.3

№	*Критерій аудиту*	*Оцінка відповідності*	*Пояснення*
8.	Швидкодія в реальному часі	Середня	До 15 FPS на середніх GPU; потребує оптимізації у частині ICP та інтеграції TSDF.
9.	Стійкість до динаміки	Висока	Маскування суттєво покращує якість карти та точність локалізації.
10.	Масштабованість і модульність	Висока	Компоненти легко замінюються (детектор, одометрія, інтеграція).
11.	Відкритість бібліотек	Висока	Всі модулі базуються на відкритих фреймворках PyTorch та Open3D.
12.	Перешкодостійкість карти	Середня	Якість карти залежить від наявності достатніх точок глибини після маскування.
13.	Портативність програмної реалізації	Висока	Модульність дозволяє запускати систему на різних платформах.
14.	Гнучкість налаштувань	Висока	Параметри детекції, маски, одометрії та TSDF легко конфігуруються.
15.	Інтеграція з іншими системами	Висока	Можлива інтеграція з ROS, робототехнічними платформами та рушіями навігації.

Проведений аудит показує, що запропонована система має високий рівень технологічної готовності. Основні компоненти системи вже представлені на ринку як стандартизовані та широко використовувані рішення. Завдяки цьому зменшується ризик технічних бар’єрів у процесі розробки й інтеграції. Водночас деякі модулі, зокрема ICP-уточнення та TSDF-інтеграція, потребують оптимізації для підвищення швидкодії, що є очікуваним для систем, орієнтованих на роботу в реальному часі.

Узагалі результати аудиту свідчать, що технології, покладені в основу проєкту, є сучасними, доступними та сумісними між собою. Система може

бути ефективно реалізована в умовах лабораторних та польових експериментів, а також має потенціал масштабування і впровадження у ширший спектр робототехнічних завдань.

4.3 Аналіз можливостей реалізації та застосування системи

Успішність розроблюваної системи робастного 3D-картування значною мірою залежить від оцінки її практичної цінності, можливих сфер застосування, конкурентоспроможності та потенційного впливу на ринок рішень комп'ютерного зору та робототехніки. На відміну від класичних алгоритмів SLAM, що орієнтовані переважно на статичні сцени, запропонована система здатна функціонувати в умовах динаміки, що істотно розширює спектр її практичного застосування.

Аналіз можливостей реалізації охоплює оцінку середовищ, у яких система може забезпечувати переваги, визначення основних груп користувачів і визначення ключових факторів успішного впровадження. З огляду на спеціалізацію алгоритмів та орієнтацію на динамічні сцени, система може виконувати важливі функції в різноманітних галузях, де застосовуються RGB-D сенсори, глибинні камери, мобільні платформи або роботизовані комплекси.

Основні можливості застосування системи охоплюють кілька напрямів:

1. Автономна робототехніка. Система дає змогу будувати тривимірне уявлення про простір у режимі реального часу, забезпечуючи роботу мобільних роботів, сервісних платформ та роботизованих маніпуляторів у середовищах з людьми або рухомими перешкодами.
2. Логістичні та складські системи. Може використовуватися для навігації автономних візків, інвентаризації приміщень, контролю

просторових потоків та моніторингу робочих зон.

3. Аграрний сектор. Система здатна працювати у складних умовах ферм, де присутні тварини, техніка та інші рухомі об’єкти, забезпечуючи картування приміщень і контроль поведінки тварин.

4. 3D-сканування та цифрова реконструкція. Технологія може застосовуватися для створення цифрових копій приміщень, об’єктів інфраструктури, виробничих зон та побутових просторів.

5. Безпека та моніторинг. Система може допомагати у створенні інтерактивних карт зон, де необхідно контролювати рух людей та визначати небезпечні області.

6. Дослідження у сфері VR/AR. Тривимірні карти можуть бути використані для побудови просторової прив’язки у системах доповненої реальності та розширеній роботизованій взаємодії.

Для оцінки можливостей реалізації сформовано аналітичну таблицю, яка охоплює ключові параметри потенційних сфер застосування (табл. 4.4).

Отримані результати показують, що запропонована система має значний потенціал для впровадження у реальні задачі. Поєднання детектора об’єктів, RGB-D одометрії та TSDF-реконструкції забезпечує стійку роботу в умовах, де більшість традиційних SLAM-систем втрачають точність або повністю втрачають стійкість. Ключовими факторами успішної реалізації є доступність сучасних інструментів глибинного навчання, відкритих бібліотек та великої кількості датасетів для тестування систем робототехнічної локалізації.

Аналіз свідчить про те, що система має достатній потенціал для виходу за межі лабораторного використання та може бути інтегрована в промислові й побутові сценарії. Її гнучкість та модульна архітектура роблять її придатною для широкого спектра застосувань, а можливість масштабування визначає перспективність подальшого розвитку технологічної платформи.

Таблиця 4.4 – Аналіз можливостей реалізації системи робастного 3D-картування

№	*Критерій оцінки*	*Характеристика застосування*	*Рівень потенціалу*
1.	Наявність потреби	Висока потреба у точних картах у динамічних сценах	Високий
2.	Готовність технологій	Наявні детектори, трекери, TSDF-інтегратори	Високий
3.	Складність впровадження	Висока через необхідність GPU-ресурсів	Середній
4.	Масштабованість рішень	Легко адаптується під різні задачі	Високий
5.	Практична цінність	Робастність до динаміки – ключова перевага	Високий
6.	Конкурентні технології	ORB-SLAM, DVO-SLAM, ElasticFusion	Середній
7.	Витрати на розробку	Переважно програмні, мінімум апаратних	Низький
8.	Готовність користувачів	Робототехнічні команди активно запроваджують SLAM	Високий
9.	Інтеграція у процеси	Підтримка ROS, Open3D, PyTorch	Високий
10.	Довгостроковий розвиток	Можливість переходу до нейронних полів (NeRF, 3DGS)	Високий

4.4 Розроблення архітектурної стратегії системи

Розроблення стратегії архітектури системи робастного 3D-картування передбачає визначення цілей користувачів, основних функціональних вимог, аналіз можливих ризиків, конкурентного середовища та ключових можливостей, які формують підґрунтя для повноцінної реалізації технології. Як і у класичних проєктах високотехнологічних систем, процес проєктування

має спиратися на баланс між технічними вимогами, перспективними напрямами розвитку та цінністю, яку продукт приносить кінцевим користувачам.

Запропонована система містить декілька взаємопов’язаних компонентів: модуль семантики, модуль одометрії, модуль ICP-уточнення, TSDF-інтегратор, систему візуалізації та інструменти логування. Архітектурна стратегія має забезпечити узгодженість роботи всіх цих модулів у реальному часі, адаптивність до різних типів датчиків та стабільність під час обробки динамічних сцен.

Для визначення стратегічних характеристик було здійснено аналіз потреб користувачів, можливих загроз, доступних можливостей та конкурентних рішень. Усі ці складові подано нижче у таблицях 4.5–4.9.

Таблиця 4.5 – Потреби та вимоги користувачів системи

№	*Потреба користувача*	*Характеристика*
1.	Стабільна локалізація у динаміці	Система має коректно ігнорувати рухомі об’єкти та зберігати точність.
2.	Точна тривимірна карта	Користувач очікує повну та гладку модель простору без артефактів.
3.	Робота в режимі реального часу	Обробка ≥ 10–15 FPS на звичайному GPU.
4.	Можливість експорту карти	Потрібно для інтеграції у зовнішні платформи та аналізу.
5.	Сумісність із різними RGB-D камерами	Очікується підтримка OpenNI, RealSense, Azure Kinect.
6.	Гнучкість налаштувань	Користувач має можливість регулювати параметри детекції, ICP та TSDF.
7.	Модульність проєкту	Необхідність заміни окремих модулів без повного переписування системи.

Таблиця 4.6 – Потенційні загрози для реалізації системи

№	*Загроза*	*Характеристика*
1.	Високі обчислювальні витрати ICP	Збільшення часу обробки кадру та зниження FPS.
2.	Недостатня якість детекцій	Може призвести до неправильного маскування динаміки.
3.	Нестабільність у поганому освітленні	RGB-канал може містити шум, що вплине на фотометрію.
4.	Обмеження глибини сенсорів	При великих просторах карта може бути неповною.
5.	Конкуренція з дорогими комерційними SLAM-системами	Готові промислові рішення можуть пропонувати «plug-and-play».
6.	Розриви синхронізації RGB/D	Можуть спричиняти спотворення карти.
7.	Обмеження TSDF-об’єму	Потрібно балансувати між щільністю карти та пам’яттю.

Таблиця 4.7 – Можливості розвитку системи

№	*Можливість*	*Пояснення*
1.	Інтеграція нейронних полів (NeRF, 3DGS)	Може суттєво покращити якість реконструкції.
2.	Використання GPU-прискореного ICP	Дозволяє підвищити FPS у 2–3 рази.
3.	Адаптивний TSDF	Можливе використання ієрархічних октадерев.
4.	Семантична карта сцен	Позбавляє користувача пост-обробки результатів.
5.	Інтеграція з ROS	Робить систему повністю сумісною з роботами.
6.	Портативність на Jetson	Дозволяє використовувати систему у мобільних платформах.
7.	Аналітика поведінки об’єктів	Використання треків може дати додаткові дані користувачам.

Таблиця 4.8 – Аналіз конкурентних рішень

№	*Конкурент*	*Ключові переваги*	*Недоліки*
1.	ORB-SLAM3	Висока точність у статичних сценах	Повністю деградує у динаміці
2.	DVO-SLAM	Сильна фотометрична модель	Нестійкий до рухомих об’єктів
3.	ElasticFusion	Якісна щільна реконструкція	Не працює зі значною динамікою
4.	RTAB-Map	Гнучка інтеграція з ROS	Карта сильно шумить при русі людей
5.	Lidar-SLAM	Стійкий до освітлення	Висока вартість обладнання

Таблиця 4.9 – Порівняння системи із конкурентами

№	*Параметр*	*Запропонована система*	*Конкуренти*
1.	Робастність до динаміки	Висока	Низька–середня
2.	Точність ATE	0.02–0.05 м	0.03–0.10 м
3.	Якість карти	Висока (TSDF)	Середня
4.	Швидкодія	10–15 FPS	10–30 FPS
5.	Модульність	Висока	Середня
6.	Сфера застосувань	Широка	Обмежена статикою
7.	Потреба в GPU	Середня	Низька–середня
8.	Можливість розширення	Висока	Низька

Результати стратегічного аналізу свідчать, що система має всі передумови для успішної реалізації завдяки значному технічному потенціалу та здатності вирішувати завдання, недоступні класичним SLAM-алгоритмам. Домінуючими перевагами є робастність до динаміки, висока якість карти у порівнянні з конкурентами та можливість масштабування. Основними викликами залишаються оптимізація швидкодії та стан обчислювальних ресурсів, проте ці аспекти можуть бути компенсовані подальшими покращеннями.

4.5 Розроблення програми впровадження системи

Розроблення програми впровадження передбачає визначення комплексу дій, які необхідно виконати для успішного запуску системи робастного 3D-картування у реальних умовах експлуатації. На цьому етапі встановлюються ключові фактори впливу, оцінюється середовище застосування, формуються альтернативи впровадження та визначаються цільові групи користувачів. У підрозділі застосовано багатокомпонентний аналітичний підхід з використанням таблиць 4.10–4.18.

Програма впровадження орієнтована на покроковий процес інтеграції технології у робототехнічні, промислові, складські, аграрні або дослідницькі системи. Основні елементи програми охоплюють:

1) аналіз факторів впливу на впровадження системи;
2) порівняння технічних підходів та вибір оптимального;
3) SWOT-аналіз;
4) визначення доступних альтернатив впровадження;
5) сегментацію цільових груп;
6) визначення стратегічних дій;
7) формування стратегії поведінки та позиціонування;
8) визначення ключових переваг продукту.

Нижче наведено деталізовані результати у форматі таблиць.

Таблиця 4.10 – Фактори впливу на впровадження системи

№	*Фактор*	*Характеристика впливу*
1.	Рівень динаміки сцени	Визначає кількість маскування та впливає на точність карти.
2.	Потужність GPU	Залежить швидкодія детектора та ICP.
3.	Тип камери	Різні сенсори мають різний шум та діапазон глибини.

Кінець табл. 4.10

№	*Фактор*	*Характеристика впливу*
4.	Якість освітлення	Впливає на детекції YOLO та фотометрію.
5.	Швидкість руху камери	Значно ускладнює оцінку поз та реконструкцію.
6.	Розмір приміщення	Визначає необхідну масштабованість TSDF.
7.	Синхронізація RGB/D	Порушення призводить до помилок карти.
8.	Тип застосування	Від нього залежить FPS, щільність карти, формат експорту.

Таблиця 4.11 – Порівняння підходів до впровадження системи

№	*Підхід*	*Переваги*	*Недоліки*
1.	Використання YOLO + TSDF у реальному часі	Висока адаптивність у динаміці	Значні обчислювальні витрати
2.	Використання лише геометричної одометрії	Нижчі вимоги до GPU	Повна нестійкість у динаміці
3.	Використання окремих модулів (без TSDF)	Легша інтеграція	Відсутність 3D-карти
4.	Хмарна обробка	Висока якість	Великі затримки, непридатно для роботів
5.	Обробка на вбудованих платформах	Портативність	Обмеження продуктивності

Таблиця 4.12 – SWOT-аналіз системи робастного 3D-картування

№	*Компонент*	*Характеристика*
1.	Strengths (сильні сторони)	Висока стійкість до динаміки, точна карта TSDF, модульність
2.	Weaknesses (слабкі сторони)	Обмеження FPS, залежність від GPU, складність налаштування
3.	Opportunities (можливості)	Розширення на VR/AR, робототехніку, аграрні системи
4.	Threats (загрози)	Конкуренція з комерційними SLAM-системами

Таблиця 4.13 – Альтернативи впровадження

№	*Альтернатива*	*Сутність рішення*	*Доцільність*
1.	Портативна версія для роботів	Використання Jetson	Висока
2.	Серверна версія	Робота на ПК з GPU	Висока
3.	Веб-інтеграція	Стримінг реконструкції	Середня
4.	Суто офлайн-режим	Рендеринг після збору	Середня
5.	Хмарні обчислення	Зберігання великих TSDF	Низька

Таблиця 4.14 – Сегментація цільових груп

№	*Цільова група*	*Інтерес*
1.	Робототехнічні інженери	Локалізація та навігація
2.	Аграрні дослідники	Моніторинг тварин та інфраструктури
3.	Команди з автономної логістики	Сканування складів
4.	Розробники AR/VR	Точні 3D-карти
5.	Академічні дослідники	Тести алгоритмів SLAM

Таблиця 4.15 – Стратегічні дії впровадження

№	*Дія*	*Очікуваний результат*
1.	Тестування на TUM RGB-D	Оцінка ATE
2.	Оптимізація ICP	Підвищення FPS
3.	Поліпшення маски	Краще відсіювання динаміки
4.	Інтеграція з ROS	Робототехнічне використання
5.	Валідація в реальних сценах	Підтвердження робастності

Таблиця 4.16 – Стратегія поведінки системи при впровадженні

№	*Стратегія*	*Характеристика*
1.	Орієнтація на точність	Поглиблене ICP-уточнення
2.	Орієнтація на швидкодію	Зменшення розміру TSDF
3.	Орієнтація на робастність	Агресивне маскування
4.	Орієнтація на масштабованість	Підтримка великих об’ємів

Таблиця 4.17 – Стратегія позиціонування системи

№	*Напрям позиціонування*	*Опис*
1.	Система для динамічних сцен	Основна перевага над конкурентами
2.	Модульна платформа	Можливість гнучкого налаштування
3.	Інструмент дослідника	Підходить для експериментів зі SLAM
4.	Індустріальний інструмент	Використання у логістиці та виробництві

Таблиця 4.18 – Ключові переваги запропонованої системи

№	*Перевага*	*Суть*
1.	Робастність до динаміки	Ігнорування рухомих об’єктів
2.	Висока якість 3D-карти	TSDF-поверхня без артефактів
3.	Модульність	Можливість легко замінювати компоненти
4.	Гнучкість застосування	Підтримка багатьох сфер
5.	Доступність реалізації	Використання відкритих бібліотек

Створена програма впровадження дозволяє послідовно перейти від теоретичної концепції до практичного використання системи. Усі проведені аналітичні процедури – оцінка факторів впливу, SWOT-аналіз, визначення альтернатив, стратегій і цільових груп – свідчать про високий потенціал системи в робототехнічних, аграрних, логістичних і дослідницьких застосуваннях. Завдяки комплексному підходу запропонована система може бути інтегрована у широкий спектр практичних сценаріїв, забезпечуючи стабільну локалізацію та якісну тривимірну реконструкцію в умовах значної динаміки.

Висновки до розділу 4

У четвертому розділі було проведено комплексне обґрунтування, розроблення та формування стратегії впровадження системи робастного

тривимірного картування в динамічних середовищах на основі поєднання нейромережевої семантики, RGB-D одометрії, ICP-уточнення та TSDF-реконструкції. На основі виконаного аналізу визначено концепцію, технічні вимоги, сфери можливого застосування, архітектурні принципи та рекомендації щодо практичного використання запропонованої системи.

У підрозділі 4.1 наведено опис основної ідеї, що полягає у створенні інтелектуальної системи, здатної коректно функціонувати в умовах присутності значної кількості динамічних об’єктів. Розглянуто ключові технічні компоненти майбутнього рішення, сформовано вимоги до точності, продуктивності та стійкості системи.

У підрозділі 4.2 представлено технологічний аудит ідеї системи, який показав високий ступінь готовності сучасних програмних і апаратних інструментів для реалізації поставленої задачі. На основі аналізу визначено сильні та слабкі сторони архітектури, а також можливі технічні обмеження впровадження.

У підрозділі 4.3 здійснено оцінку можливостей практичного застосування системи в різних сферах, включно з робототехнікою, логістикою, аграрними технологіями, системами цифрової реконструкції та безпеки. Було визначено, що запропонована система має значний потенціал завдяки здатності працювати в умовах динаміки, де традиційні SLAM-методи демонструють низьку ефективність.

У підрозділі 4.4 розроблено архітектурну стратегію системи та проаналізовано ключові потреби користувачів, загрози, можливості та конкурентні рішення. Результати свідчать про те, що запропонована архітектура володіє високим потенціалом масштабованості, гнучкості та адаптивності до різних умов експлуатації.

У підрозділі 4.5 сформовано програму впровадження, яка включає аналіз факторів впливу, SWOT-аналіз, порівняння технічних альтернатив, сегментацію цільових груп, а також визначення стратегічних дій і напрямів позиціонування системи. Це дозволяє системно підійти до інтеграції

технології у практичні сценарії та визначити оптимальні шляхи її розвитку.

У підсумку слід зазначити, що розроблена система робастного 3D-картування має чітко окреслені технічні переваги та перспективи застосування в широкому спектрі задач. Вона здатна забезпечити високоякісну просторову реконструкцію та стабільну локалізацію в умовах значної динамічності сцени. Одержані результати підтверджують наукову та практичну цінність створеної архітектури, а також обґрунтовують доцільність подальшого розвитку системи, зокрема через оптимізацію швидкодії, покращення семантичної сегментації та інтеграцію з більш просунутими методами тривимірного представлення середовищ.

ПЕРЕЛІК ПОСИЛАНЬ

1. МАГІСТЕРСЬКА ДИСЕРТАЦІЯ. ОРГАНІЗАЦІЯ, ВИМОГИ ДО СТРУКТУРИ, ЗМІСТУ ТА ОФОРМЛЕННЯ – КПІ, дата останнього звернення: жовтень 20, 2025, https://tae.kpi.ua/wp–content/uploads/2023/09/MAGISTERSKA.pdf
2. Бакалаврська кваліфікаційна робота: Вимоги до оформлення – КПІ, дата останнього звернення: жовтень 20, 2025, https://ela.kpi.ua/bitstreams/a4a342d8–b945–488c–9b2f–d2b4dc746f2a/download
3. вимоги до виконання дипломного проекту освітньо–кваліфікаційного рівня «бакалавр – КПІ, дата останнього звернення: жовтень 20, 2025, https://ela.kpi.ua/bitstream/123456789/47029/1/Vymohy_do_vykonannia_OKR_Baklavr_2022_V_2.pdf
4. ADD–SLAM: Adaptive Dynamic Dense SLAM with Gaussian Splatting – arXiv, дата останнього звернення: жовтень 20, 2025, https://arxiv.org/html/2505.19420v1
5. YOLOv8 vs YOLOv5: A Detailed Comparison – Ultralytics YOLO Docs, дата останнього звернення: жовтень 20, 2025, https://docs.ultralytics.com/compare/yolov8–vs–yolov5/
6. YOLOv5 vs. YOLOv8: A Detailed Comparison – Ultralytics YOLO Docs, дата останнього звернення: жовтень 20, 2025, https://docs.ultralytics.com/compare/yolov5–vs–yolov8/
7. YOLO Loss Function Part 1: SIoU and Focal Loss – Learn OpenCV, дата останнього звернення: жовтень 20, 2025, https://learnopencv.com/yolo–loss–function–siou–focal–loss/
8. Probabilistic Data Association for Semantic SLAM – Nikolay A. Atanasov, дата останнього звернення: жовтень 20, 2025, https://natanaso.github.io/ref/Bowman_SemanticSLAM_ICRA17.pdf

9. Neural Radiance Field Dynamic Scene SLAM Based on Ray Segmentation and Bundle Adjustment – MDPI, дата останнього звернення: жовтень 20, 2025, https://www.mdpi.com/1424–8220/25/6/1679
10. Implement SLAM from scratch. How to code SLAM. What is the SLAM… | by Luis Bermudez | machinevision | Medium, дата останнього звернення: жовтень 20, 2025, https://medium.com/machinevision/implement–slam–from–scratch–b1fb599f40c8
11. Active Semantic Mapping and Pose Graph Spectral Analysis for Robot Exploration – arXiv, дата останнього звернення: жовтень 20, 2025, https://arxiv.org/html/2408.14726v1
12. TensorRT Export for YOLO11 Models – Ultralytics YOLO Docs, дата останнього звернення: жовтень 20, 2025, https://docs.ultralytics.com/integrations/tensorrt/
13. Deploying YOLOv5 on NVIDIA Jetson Orin with cuDLA: Quantization–Aware Training to Inference, дата останнього звернення: жовтень 20, 2025, https://developer.nvidia.com/blog/deploying–yolov5–on–nvidia–jetson–orin–with–cudla–quantization–aware–training–to–inference/
14. DLA+INT8 compiled engine doesn't produce meaningful results – Jetson Orin NX, дата останнього звернення: жовтень 20, 2025, https://forums.developer.nvidia.com/t/dla–int8–compiled–engine–doesnt–produce–meaningful–results/316760
15. [jetson] Running yolov8 classfication model in DLA #1 – Why DLA? | by Maro JEON, дата останнього звернення: жовтень 20, 2025, https://medium.com/@MaroJEON/jetson–running–yolov8–classfication–model–in–dla–1–why–dla–6a2d2860ebdd
16. NVIDIA–AI–IOT/cuDLA–samples: YOLOv5 on Orin DLA – GitHub, дата останнього звернення: жовтень 20, 2025, https://github.com/NVIDIA–AI–IOT/cuDLA–samples

17. Gaussian Splatting SLAM – CVF Open Access, дата останнього звернення: жовтень 20, 2025, https://openaccess.thecvf.com/content/CVPR2024/papers/Matsuki_Gaussian_Splatting_SLAM_CVPR_2024_paper.pdf
18. GS–SLAM: Dense Visual SLAM with 3D Gaussian Splatting, дата останнього звернення: жовтень 20, 2025, https://gs–slam.github.io/
19. SNI–SLAM: Semantic Neural Implicit SLAM – arXiv, дата останнього звернення: жовтень 20, 2025, https://arxiv.org/html/2311.11016v2
20. Neural Implicit Semantic RGB–D SLAM for 3D Consistent Scene Understanding, дата останнього звернення: жовтень 20, 2025, https://zju3dv.github.io/nis_slam/
21. [2407.20853] NIS–SLAM: Neural Implicit Semantic RGB–D SLAM for 3D Consistent Scene Understanding – arXiv, дата останнього звернення: жовтень 20, 2025, https://arxiv.org/abs/2407.20853

ДОДАТОК А ЛІСТИНГ ПРОГРАМНОГО КОДУ

```
"""
YOLOv8-seg + ByteTrack + TSDF SLAM на TUM RGB-D (Freiburg 3) з ATE RMSE
з pose graph + loop closure на ORB-фічах
"""

from __future__ import annotations

from dataclasses import dataclass
from pathlib import Path
from typing import List, Tuple, Optional, Any

import math
import time
import csv

import numpy as np
import cv2
import torch
import open3d as o3d
from ultralytics import YOLO


# =======================
#   ДРУК ПРО СИСТЕМУ
# =======================
print("Torch version:", torch.__version__)
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("DEVICE:", DEVICE.type)


# =======================
#   КОНФІГ
# =======================
@dataclass
class ConfigTUM:
    # ---------- Шляхи до TUM ----------
    base_dir: Path = Path("data_tum")
    seq_name: str = "rgbd_dataset_freiburg3_walking_xyz"
    assoc_filename: str = "associate.txt"

    # ---------- Глибина ----------
    depth_factor: float = 5000.0
    depth_min: float = 0.3
    depth_max: float = 4.0

    # ---------- Камера (інтрінсики fr3) ----------
    fx: float = 535.4
    fy: float = 539.2
    cx: float = 320.1
    cy: float = 247.6

    # ---------- TSDF ----------
    voxel_length: float = 0.008
```

```
sdf_trunc: float = 0.03
tsdf_color_type: o3d.pipelines.integration.TSDFVolumeColorType = (
    o3d.pipelines.integration.TSDFVolumeColorType.RGB8
)

# ---------- YOLO ----------
yolo_weights_pt: str = "yolov8s-seg.pt"
yolo_weights_engine: str = "yolov8s-seg.engine"
prefer_engine_if_exists: bool = True

conf: float = 0.35
iou: float = 0.5
imgsz: int = 640

# ---------- ByteTrack ----------
tracker: str = "bytetrack_slam.yaml"

# Для TUM – динаміка: лише 'person' (COCO id = 0)
dynamic_class_ids: Tuple[int, ...] = (0,)
min_dynamic_mask_area: int = 20 * 20

# Маскувати динаміку і для одометрії
use_dynamic_mask_for_odom: bool = True

# Маска динаміки: margin (дилатація)
dynamic_mask_margin_kernel: int = 9
dynamic_mask_margin_iter: int = 1

# ---------- Depth preprocessing ----------
use_depth_bilateral: bool = True
bilateral_d: int = 5
bilateral_sigma_color: float = 0.06
bilateral_sigma_space: float = 5.0

# ---------- ICP frame-to-model ----------
use_frame_to_model_icp: bool = True
icp_every: int = 1
icp_model_update_every: int = 10
icp_max_corr_dist: float = 0.03
icp_frame_down_voxel: float = 0.02
icp_model_down_voxel: float = 0.02
icp_min_fitness: float = 0.60
icp_max_rmse: float = 0.015
icp_min_model_points: int = 5_000
icp_max_iter: int = 30

# ---------- Motion gating ----------
max_trans_per_frame: float = 0.30
max_rot_deg_per_frame: float = 20.0

# ---------- Keyframes ----------
use_keyframes: bool = True
# більш щільні keyframe-и
# (було 0.02 / 1.0 / 10)
keyframe_min_trans: float = 0.01
keyframe_min_rot_deg: float = 0.5
```

```
    keyframe_max_interval: int = 5

    # ---------- Сцена / snapshot ----------
    static_snapshot_frame: int = 4
    # додатковий snapshot у середині траєкторії (=-1, щоб вимкнути)
    extra_snapshot_frame: int = 400
    # 0.0 = без радіусного відсікання (для коректного ATE по всій послідовності)
    scene_max_radius: float = 0.0

    # ---------- Pose graph / loop closure ----------
    use_pose_graph: bool = True
    use_loop_closure: bool = True
    # ORB-фічі
    loop_feature_n: int = 2000
    loop_min_kf_separation: int = 10
    loop_min_matches: int = 60
    loop_icp_max_corr_dist: float = 0.05
    loop_icp_min_fitness: float = 0.30
    loop_icp_max_rmse: float = 0.05
    # будемо оптимізувати лише наприкінці, але параметр залишаємо
    pose_graph_optimize_every_kf: int = 0
    reintegrate_after_optimization: bool = False

    # ---------- Вихід / логи ----------
    output_dir: Path = Path("output_tum_tsdf")
    mesh_raw_file: str = "tsdf_mesh_raw.ply"
    pcd_raw_file: str = "tsdf_pcd_raw.ply"
    traj_tum_file: str = "traj_est_tum.txt"
    frame_log_csv: str = "frame_log.csv"
    objects_csv: str = "tracked_objects.csv"
    save_tracked_objects: bool = True


cfg = ConfigTUM()
cfg.output_dir.mkdir(parents=True, exist_ok=True)
print(cfg)


# ========================
#   ByteTrack SLAM-конфіг
# ========================
def ensure_tracker_config(cfg: ConfigTUM) -> None:
    tracker_path = Path(cfg.tracker)
    if not tracker_path.is_absolute():
        base_dir = Path(__file__).parent if "__file__" in globals() else
Path(".")
        tracker_path = base_dir / tracker_path

    tracker_path.parent.mkdir(parents=True, exist_ok=True)

    if tracker_path.exists():
        print(f"[tracker] Using existing tracker config: {tracker_path}")
    else:
        yaml_text = """# ByteTrack config tuned for SLAM / Object-Aware SLAM
tracker_type: bytetrack
track_high_thresh: 0.65
```

```
track_low_thresh: 0.05
new_track_thresh: 0.75
track_buffer: 120
match_thresh: 0.85
fuse_score: True
"""
        tracker_path.write_text(yaml_text, encoding="utf-8")
        print(f"[tracker] Created ByteTrack SLAM config at: {tracker_path}")

    cfg.tracker = str(tracker_path)


ensure_tracker_config(cfg)


# ======================
#   УТИЛІТИ ДЛЯ TUM
# ======================
def tum_seq_path(cfg: ConfigTUM) -> Path:
    return cfg.base_dir / cfg.seq_name


def load_tum_timestamp_file(path: Path) -> List[Tuple[float, str]]:
    entries: List[Tuple[float, str]] = []
    with open(path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            parts = line.split()
            ts = float(parts[0])
            rel_path = parts[1]
            entries.append((ts, rel_path))
    return entries


def create_associate_file(cfg: ConfigTUM, max_time_diff: float = 0.02) ->
None:
    seq_path = tum_seq_path(cfg)
    rgb_list = load_tum_timestamp_file(seq_path / "rgb.txt")
    depth_list = load_tum_timestamp_file(seq_path / "depth.txt")

    depth_times = np.array([t for t, _ in depth_list], dtype=np.float64)

    associations: List[Tuple[float, str, float, str]] = []
    for t_rgb, path_rgb in rgb_list:
        idx = int(np.argmin(np.abs(depth_times - t_rgb)))
        t_depth, path_depth = depth_list[idx]
        if abs(t_depth - t_rgb) < max_time_diff:
            associations.append((t_rgb, path_rgb, t_depth, path_depth))

    assoc_path = seq_path / cfg.assoc_filename
    with open(assoc_path, "w", encoding="utf-8") as f:
        for t_rgb, path_rgb, t_depth, path_depth in associations:
            f.write(f"{t_rgb:.6f} {path_rgb} {t_depth:.6f} {path_depth}\n")
```

```
    print(f"[associate] Saved {len(associations)} pairs to {assoc_path}")


def load_associations(cfg: ConfigTUM) -> List[Tuple[float, Path, float,
Path]]:
    seq_path = tum_seq_path(cfg)
    assoc_path = seq_path / cfg.assoc_filename
    if not assoc_path.exists():
        print("[associate] File not found, creating...")
        create_associate_file(cfg)

    pairs: List[Tuple[float, Path, float, Path]] = []
    with open(assoc_path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            t_rgb_s, path_rgb_s, t_depth_s, path_depth_s = line.split()
            t_rgb = float(t_rgb_s)
            t_depth = float(t_depth_s)
            rgb_path = seq_path / path_rgb_s
            depth_path = seq_path / path_depth_s
            pairs.append((t_rgb, rgb_path, t_depth, depth_path))

    print(f"[associate] Loaded {len(pairs)} pairs from {assoc_path}")
    return pairs


def make_intrinsics(cfg: ConfigTUM, width: int, height: int) ->
o3d.camera.PinholeCameraIntrinsic:
    intrinsic = o3d.camera.PinholeCameraIntrinsic()
    intrinsic.set_intrinsics(width, height, cfg.fx, cfg.fy, cfg.cx, cfg.cy)
    return intrinsic


def load_rgbd_from_paths(
    cfg: ConfigTUM,
    rgb_path: Path,
    depth_path: Path
) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Повертає:
      color_rgb: uint8 HxWx3 (RGB)
      color_bgr: uint8 HxWx3 (BGR) для YOLO
      depth_meters: float32 HxW (м), поза [min,max] -> 0
    """
    color_bgr = cv2.imread(str(rgb_path), cv2.IMREAD_COLOR)
    if color_bgr is None:
        raise FileNotFoundError(f"Cannot read RGB image: {rgb_path}")
    color_rgb = cv2.cvtColor(color_bgr, cv2.COLOR_BGR2RGB)

    depth_raw = cv2.imread(str(depth_path), cv2.IMREAD_UNCHANGED)
    if depth_raw is None:
        raise FileNotFoundError(f"Cannot read depth image: {depth_path}")
    if depth_raw.ndim == 3:
        depth_raw = depth_raw[..., 0]
```

```
    if depth_raw.dtype != np.uint16:
        raise ValueError(f"Expected 16-bit depth PNG, got {depth_raw.dtype}")

    depth_meters = depth_raw.astype(np.float32) / cfg.depth_factor
    depth_meters[(depth_meters < cfg.depth_min) | (depth_meters >
cfg.depth_max)] = 0.0
    return color_rgb, color_bgr, depth_meters


# =======================
#   YOLO завантаження
# =======================
def load_yolo_model(cfg: ConfigTUM) -> YOLO:
    weights = cfg.yolo_weights_pt
    if cfg.prefer_engine_if_exists and
Path(cfg.yolo_weights_engine).exists():
        weights = cfg.yolo_weights_engine
        print(f"[YOLO] Using TensorRT engine: {weights}")
    else:
        print(f"[YOLO] Using PyTorch weights: {weights}")

    model = YOLO(weights)
    model.to(DEVICE)
    try:
        model.fuse()
    except Exception:
        pass
    model.eval()

    # Warm-up
    dummy = np.zeros((cfg.imgsz, cfg.imgsz, 3), dtype=np.uint8)
    with torch.inference_mode():
        _ = model.predict(
            source=[dummy],
            imgsz=cfg.imgsz,
            conf=cfg.conf,
            verbose=False,
            device=DEVICE.type,
        )
    return model


yolo_model = load_yolo_model(cfg)


# =======================
#   Маска динаміки + трек-об'єкти
# =======================
@dataclass
class TrackedObject:
    frame_idx: int
    timestamp: float
    track_id: int
    class_id: int
    conf: float
    cx: float
```

```
    cy: float
    area_px: float


@dataclass
class YoloMaskResult:
    depth_masked: np.ndarray
    exclusion_mask_u8: np.ndarray
    objects: List[TrackedObject]
    raw_result: Any


# =======================
#   ЛОГ КАДРІВ / KEYFRAMES
# =======================
@dataclass
class FrameLog:
    idx: int
    timestamp: float
    odom_success: int
    icp_ran: int
    icp_accepted: int
    motion_ok: int
    keyframe_ok: int
    integrated: int
    reason: str
    t_yolo: float
    t_odom: float
    t_icp: float
    t_tsdf: float


@dataclass
class Keyframe:
    kf_id: int
    frame_idx: int
    timestamp: float
    pose: np.ndarray          # world_T_cam
    gray: np.ndarray
    depth_tsdf: np.ndarray
    keypoints: Any
    descriptors: Optional[np.ndarray]


# ORB + BFMatcher (лениве створення)
_orb = None
_bf = None


def get_feature_extractor(cfg: ConfigTUM):
    global _orb, _bf
    if not cfg.use_pose_graph:
        return None, None
    if _orb is None or _bf is None:
        _orb = cv2.ORB_create(nfeatures=cfg.loop_feature_n)
        _bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
```

```
    return _orb, _bf


def compute_orb_features(gray: np.ndarray, cfg: ConfigTUM):
    orb, _ = get_feature_extractor(cfg)
    if orb is None:
        return [], None
    kps, des = orb.detectAndCompute(gray, None)
    if des is None:
        des = np.empty((0, 32), dtype=np.uint8)
    return kps, des


def create_pointcloud_from_kf(
    kf: Keyframe,
    intrinsic: o3d.camera.PinholeCameraIntrinsic,
    cfg: ConfigTUM,
) -> o3d.geometry.PointCloud:
    depth_o3d = o3d.geometry.Image(kf.depth_tsdf.astype(np.float32))
    rgb = cv2.cvtColor(kf.gray, cv2.COLOR_GRAY2RGB)
    color_o3d = o3d.geometry.Image(rgb.astype(np.uint8))
    rgbd = o3d.geometry.RGBDImage.create_from_color_and_depth(
        color_o3d,
        depth_o3d,
        depth_scale=1.0,
        depth_trunc=cfg.depth_max,
        convert_rgb_to_intensity=False,
    )
    pcd = o3d.geometry.PointCloud.create_from_rgbd_image(rgbd, intrinsic)
    pcd = pcd.voxel_down_sample(cfg.icp_model_down_voxel)
    pcd.estimate_normals()
    return pcd


def add_loop_closure_edges(
    new_kf: Keyframe,
    keyframes: List[Keyframe],
    pose_graph: o3d.pipelines.registration.PoseGraph,
    intrinsic: o3d.camera.PinholeCameraIntrinsic,
    cfg: ConfigTUM,
):
    if not cfg.use_loop_closure:
        return
    _, bf = get_feature_extractor(cfg)
    if bf is None:
        return

    for prev_kf in keyframes:
        if prev_kf.kf_id == new_kf.kf_id:
            continue
        if abs(new_kf.frame_idx - prev_kf.frame_idx) <
cfg.loop_min_kf_separation:
            continue
        if prev_kf.descriptors is None or new_kf.descriptors is None:
            continue
        if len(prev_kf.descriptors) == 0 or len(new_kf.descriptors) == 0:
```

```
            continue

        matches = bf.match(prev_kf.descriptors, new_kf.descriptors)
        if len(matches) < cfg.loop_min_matches:
            continue

        source_pcd = create_pointcloud_from_kf(new_kf, intrinsic, cfg)
        target_pcd = create_pointcloud_from_kf(prev_kf, intrinsic, cfg)
        if len(source_pcd.points) == 0 or len(target_pcd.points) == 0:
            continue

        init_T = np.linalg.inv(prev_kf.pose) @ new_kf.pose
        criteria = o3d.pipelines.registration.ICPConvergenceCriteria(
            max_iteration=int(cfg.icp_max_iter)
        )
        icp = o3d.pipelines.registration.registration_icp(
            source_pcd,
            target_pcd,
            max_correspondence_distance=float(cfg.loop_icp_max_corr_dist),
            init=init_T,

estimation_method=o3d.pipelines.registration.TransformationEstimationPointToP
lane(),
            criteria=criteria,
        )

        if icp.fitness < cfg.loop_icp_min_fitness or icp.inlier_rmse >
cfg.loop_icp_max_rmse:
            continue

        info = np.identity(6)
        pose_graph.edges.append(
            o3d.pipelines.registration.PoseGraphEdge(
                prev_kf.kf_id,
                new_kf.kf_id,
                icp.transformation,
                uncertain=True,
                information=info,
            )
        )
        print(f"[LOOP] Added loop edge KF {prev_kf.kf_id} -> {new_kf.kf_id} "
              f"(fitness={icp.fitness:.3f}, rmse={icp.inlier_rmse:.3f})")


# ======================
#   Маска динаміки (YOLO)
# ======================
def _clip_poly(poly: np.ndarray, w: int, h: int) -> np.ndarray:
    p = poly.copy()
    p[:, 0] = np.clip(p[:, 0], 0, w - 1)
    p[:, 1] = np.clip(p[:, 1], 0, h - 1)
    return p


def build_exclusion_mask_from_polygons(
    res: Any,
```

```
    cfg: ConfigTUM,
    shape_hw: Tuple[int, int],
    frame_idx: int,
    timestamp: float
) -> Tuple[np.ndarray, List[TrackedObject]]:
    h, w = shape_hw
    exclusion = np.zeros((h, w), dtype=np.uint8)
    objects: List[TrackedObject] = []

    if res.masks is None or res.boxes is None or len(res.boxes) == 0:
        return exclusion, objects

    ids = None
    try:
        if getattr(res.boxes, "id", None) is not None:
            ids = res.boxes.id.int().cpu().tolist()
    except Exception:
        ids = None

    class_ids = res.boxes.cls.int().cpu().tolist()
    confs = res.boxes.conf.float().cpu().tolist()
    polygons = res.masks.xy

    for i, poly in enumerate(polygons):
        cls_id = int(class_ids[i])
        if cls_id not in cfg.dynamic_class_ids:
            continue
        if poly is None or len(poly) < 3:
            continue

        poly = poly.astype(np.float32)
        poly = _clip_poly(poly, w, h)

        area = float(cv2.contourArea(poly))
        if area < float(cfg.min_dynamic_mask_area):
            continue

        poly_i32 = poly.astype(np.int32).reshape(-1, 1, 2)
        cv2.fillPoly(exclusion, [poly_i32], 255)

        M = cv2.moments(poly)
        if M["m00"] != 0.0:
            cx = float(M["m10"] / M["m00"])
            cy = float(M["m01"] / M["m00"])
        else:
            cx, cy = float(poly[:, 0].mean()), float(poly[:, 1].mean())

        track_id = int(ids[i]) if ids is not None else -1
        objects.append(
            TrackedObject(
                frame_idx=frame_idx,
                timestamp=timestamp,
                track_id=track_id,
                class_id=cls_id,
                conf=float(confs[i]),
                cx=cx,
```

```
                cy=cy,
                area_px=area,
            )
        )

    k = int(cfg.dynamic_mask_margin_kernel)
    if k >= 3:
        if k % 2 == 0:
            k += 1
        kernel = np.ones((k, k), np.uint8)
        exclusion = cv2.dilate(exclusion, kernel,
iterations=int(cfg.dynamic_mask_margin_iter))

    return exclusion, objects


def build_exclusion_mask_from_bboxes(
    res: Any,
    cfg: ConfigTUM,
    shape_hw: Tuple[int, int],
) -> np.ndarray:
    h, w = shape_hw
    exclusion = np.zeros((h, w), dtype=np.uint8)
    if res.boxes is None or len(res.boxes) == 0:
        return exclusion

    boxes_xyxy = res.boxes.xyxy.cpu().numpy().astype(int)
    classes = res.boxes.cls.cpu().numpy().astype(int)

    for box, cls_id in zip(boxes_xyxy, classes):
        if int(cls_id) not in cfg.dynamic_class_ids:
            continue
        x1, y1, x2, y2 = map(int, box)
        x1 = max(0, min(w - 1, x1))
        x2 = max(0, min(w, x2))
        y1 = max(0, min(h - 1, y1))
        y2 = max(0, min(h, y2))
        if x2 <= x1 or y2 <= y1:
            continue
        area = (x2 - x1) * (y2 - y1)
        if area < cfg.min_dynamic_mask_area:
            continue
        exclusion[y1:y2, x1:x2] = 255

    k = int(cfg.dynamic_mask_margin_kernel)
    if k >= 3:
        if k % 2 == 0:
            k += 1
        kernel = np.ones((k, k), np.uint8)
        exclusion = cv2.dilate(exclusion, kernel,
iterations=int(cfg.dynamic_mask_margin_iter))

    return exclusion


def preprocess_depth_for_tsdf(depth_m: np.ndarray, cfg: ConfigTUM) ->
```

```
np.ndarray:
    if not cfg.use_depth_bilateral:
        return depth_m

    d = int(cfg.bilateral_d)
    if d < 3:
        return depth_m

    depth = depth_m.astype(np.float32, copy=True)
    zeros = (depth == 0.0)
    filtered = cv2.bilateralFilter(
        depth,
        d=d,
        sigmaColor=float(cfg.bilateral_sigma_color),
        sigmaSpace=float(cfg.bilateral_sigma_space),
    )
    filtered[zeros] = 0.0
    return filtered


def run_yolo_and_mask_depth(
    frame_bgr: np.ndarray,
    depth_meters: np.ndarray,
    cfg: ConfigTUM,
    frame_idx: int,
    timestamp: float,
) -> YoloMaskResult:
    with torch.inference_mode():
        results = yolo_model.track(
            source=[frame_bgr],
            imgsz=cfg.imgsz,
            conf=cfg.conf,
            iou=cfg.iou,
            tracker=cfg.tracker,
            persist=True,
            verbose=False,
            device=DEVICE.type,
            classes=list(cfg.dynamic_class_ids),
        )

    res = results[0]
    h, w = depth_meters.shape

    objects: List[TrackedObject] = []
    if res.masks is not None and getattr(res.masks, "xy", None) is not None:
        exclusion_u8, objects = build_exclusion_mask_from_polygons(
            res, cfg, (h, w), frame_idx=frame_idx, timestamp=timestamp
        )
    else:
        exclusion_u8 = build_exclusion_mask_from_bboxes(res, cfg, (h, w))

    depth_masked = depth_meters.copy()
    depth_masked[exclusion_u8 > 0] = 0.0
    depth_masked = preprocess_depth_for_tsdf(depth_masked, cfg)

    return YoloMaskResult(
```

```
        depth_masked=depth_masked,
        exclusion_mask_u8=exclusion_u8,
        objects=objects,
        raw_result=res,
    )


def make_o3d_rgbd(color_rgb: np.ndarray, depth_meters: np.ndarray, cfg:
ConfigTUM) -> o3d.geometry.RGBDImage:
    if color_rgb.dtype != np.uint8:
        color_rgb = color_rgb.astype(np.uint8)
    depth_image = o3d.geometry.Image(depth_meters.astype(np.float32))
    color_image = o3d.geometry.Image(color_rgb)

    rgbd = o3d.geometry.RGBDImage.create_from_color_and_depth(
        color=color_image,
        depth=depth_image,
        depth_scale=1.0,
        depth_trunc=cfg.depth_max,
        convert_rgb_to_intensity=False,
    )
    return rgbd


# ======================
#   Motion / keyframe
# ======================
def decompose_relative_transform(T: np.ndarray) -> Tuple[float, float]:
    assert T.shape == (4, 4)
    R = T[:3, :3]
    t = T[:3, 3]
    trans_norm = float(np.linalg.norm(t))

    trace = float(np.trace(R))
    cos_angle = (trace - 1.0) / 2.0
    cos_angle = max(-1.0, min(1.0, cos_angle))
    angle_rad = math.acos(cos_angle)
    rot_deg = math.degrees(angle_rad)
    return trans_norm, rot_deg


def is_motion_plausible(T_rel: np.ndarray, cfg: ConfigTUM) -> bool:
    trans_norm, rot_deg = decompose_relative_transform(T_rel)
    if trans_norm > cfg.max_trans_per_frame or rot_deg >
cfg.max_rot_deg_per_frame:
        return False
    return True


# ======================
#   ЛОГИ
# ======================
def write_frame_log_csv(path: Path, rows: List[FrameLog]) -> None:
    with open(path, "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow([
```

```
            "idx", "timestamp",
            "odom_success", "icp_ran", "icp_accepted",
            "motion_ok", "keyframe_ok", "integrated",
            "reason",
            "t_yolo", "t_odom", "t_icp", "t_tsdf"
        ])
    for r in rows:
        with open(path, "a", newline="", encoding="utf-8") as f:
            w = csv.writer(f)
            w.writerow([
                r.idx, f"{r.timestamp:.6f}",
                r.odom_success, r.icp_ran, r.icp_accepted,
                r.motion_ok, r.keyframe_ok, r.integrated,
                r.reason,
                f"{r.t_yolo:.6f}", f"{r.t_odom:.6f}", f"{r.t_icp:.6f}",
f"{r.t_tsdf:.6f}"
            ])


def write_objects_csv(path: Path, objects: List[TrackedObject]) -> None:
    with open(path, "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(["frame_idx", "timestamp", "track_id", "class_id", "conf",
"cx", "cy", "area_px"])
        for o in objects:
            w.writerow([
                o.frame_idx, f"{o.timestamp:.6f}", o.track_id, o.class_id,
                f"{o.conf:.4f}", f"{o.cx:.2f}", f"{o.cy:.2f}",
f"{o.area_px:.1f}"
            ])


def write_trajectory_tum(path: Path, ts: List[float], poses:
List[np.ndarray]) -> None:
    def rot_to_quat(R: np.ndarray) -> Tuple[float, float, float, float]:
        tr = float(np.trace(R))
        if tr > 0.0:
            S = math.sqrt(tr + 1.0) * 2.0
            qw = 0.25 * S
            qx = (R[2, 1] - R[1, 2]) / S
            qy = (R[0, 2] - R[2, 0]) / S
            qz = (R[1, 0] - R[0, 1]) / S
        else:
            if R[0, 0] > R[1, 1] and R[0, 0] > R[2, 2]:
                S = math.sqrt(1.0 + R[0, 0] - R[1, 1] - R[2, 2]) * 2.0
                qw = (R[2, 1] - R[1, 2]) / S
                qx = 0.25 * S
                qy = (R[0, 1] + R[1, 0]) / S
                qz = (R[0, 2] + R[2, 0]) / S
            elif R[1, 1] > R[2, 2]:
                S = math.sqrt(1.0 + R[1, 1] - R[0, 0] - R[2, 2]) * 2.0
                qw = (R[0, 2] - R[2, 0]) / S
                qx = (R[0, 1] + R[1, 0]) / S
                qy = 0.25 * S
                qz = (R[1, 2] + R[2, 1]) / S
            else:
```

```
                S = math.sqrt(1.0 + R[2, 2] - R[0, 0] - R[1, 1]) * 2.0
                qw = (R[1, 0] - R[0, 1]) / S
                qx = (R[0, 2] + R[2, 0]) / S
                qy = (R[1, 2] + R[2, 1]) / S
                qz = 0.25 * S

        n = math.sqrt(qx*qx + qy*qy + qz*qz + qw*qw)
        if n > 0:
            qx, qy, qz, qw = qx/n, qy/n, qz/n, qw/n
        return qx, qy, qz, qw

    with open(path, "w", encoding="utf-8") as f:
        for t, T in zip(ts, poses):
            R = T[:3, :3]
            p = T[:3, 3]
            qx, qy, qz, qw = rot_to_quat(R)
            f.write(
                f"{t:.6f} {p[0]:.6f} {p[1]:.6f} {p[2]:.6f} "
                f"{qx:.6f} {qy:.6f} {qz:.6f} {qw:.6f}\n"
            )


# ========================
#   SLAM + TSDF + POSE GRAPH
# ========================
def run_tum_yolo_slam_tsdf(
    cfg: ConfigTUM,
    max_frames: Optional[int] = None,
    remove_outliers: bool = True,
):
    pairs = load_associations(cfg)
    if max_frames is not None:
        pairs = pairs[:max_frames]
    num_frames = len(pairs)
    print(f"[SLAM] Using {num_frames} frames")

    # ---------- Перший кадр ----------
    t0, rgb_path_0, _, depth_path_0 = pairs[0]
    color0_rgb, color0_bgr, depth0 = load_rgbd_from_paths(cfg, rgb_path_0,
depth_path_0)

    H, W = depth0.shape
    intrinsic = make_intrinsics(cfg, W, H)

    volume = o3d.pipelines.integration.ScalableTSDFVolume(
        voxel_length=cfg.voxel_length,
        sdf_trunc=cfg.sdf_trunc,
        color_type=cfg.tsdf_color_type,
    )

    pose = np.eye(4, dtype=np.float64)  # world_T_cam
    poses: List[np.ndarray] = [pose.copy()]
    ts_est: List[float] = [t0]

    frame_logs: List[FrameLog] = []
    all_objects: List[TrackedObject] = []
```

```
    time_yolo = 0.0
    time_odom = 0.0
    time_icp = 0.0
    time_tsdf = 0.0

    frame_anchor_kf_id: List[int] = [0] * num_frames
    frame_T_anchor_to_cam: List[np.ndarray] = [
        np.eye(4, dtype=np.float64) for _ in range(num_frames)
    ]

    keyframes: List[Keyframe] = []
    pose_graph: Optional[o3d.pipelines.registration.PoseGraph] = None
    if cfg.use_pose_graph:
        pose_graph = o3d.pipelines.registration.PoseGraph()

    # --- Перший кадр: YOLO + маска ---
    t_y = time.perf_counter()
    y0 = run_yolo_and_mask_depth(color0_bgr, depth0, cfg, frame_idx=0,
timestamp=t0)
    dt_y = time.perf_counter() - t_y
    time_yolo += dt_y

    if cfg.save_tracked_objects and y0.objects:
        all_objects.extend(y0.objects)

    depth0_tsdf = y0.depth_masked
    depth0_odom = depth0_tsdf if cfg.use_dynamic_mask_for_odom else depth0

    rgbd0_odom = make_o3d_rgbd(color0_rgb, depth0_odom, cfg)
    rgbd0_tsdf = make_o3d_rgbd(color0_rgb, depth0_tsdf, cfg)

    t_t = time.perf_counter()
    volume.integrate(rgbd0_tsdf, intrinsic, np.linalg.inv(pose))
    dt_t = time.perf_counter() - t_t
    time_tsdf += dt_t

    frame_logs.append(FrameLog(
        idx=0, timestamp=t0,
        odom_success=1, icp_ran=0, icp_accepted=0,
        motion_ok=1, keyframe_ok=1, integrated=1,
        reason="first_frame",
        t_yolo=dt_y, t_odom=0.0, t_icp=0.0, t_tsdf=dt_t
    ))

    prev_rgbd_odom = rgbd0_odom

    # Keyframe
    last_key_pose = pose.copy()
    last_key_idx = 0
    last_key_kf_id = 0

    if cfg.use_pose_graph and pose_graph is not None:
        gray0 = cv2.cvtColor(color0_bgr, cv2.COLOR_BGR2GRAY)
        kps0, des0 = compute_orb_features(gray0, cfg)
        kf0 = Keyframe(
```

```
            kf_id=0,
            frame_idx=0,
            timestamp=t0,
            pose=pose.copy(),
            gray=gray0,
            depth_tsdf=depth0_tsdf.copy(),
            keypoints=kps0,
            descriptors=des0,
        )
        keyframes.append(kf0)
        pose_graph.nodes.append(
            o3d.pipelines.registration.PoseGraphNode(pose.copy())
        )

    frame_anchor_kf_id[0] = 0
    frame_T_anchor_to_cam[0] = np.eye(4, dtype=np.float64)

    # Одометрія
    odo_opt = o3d.pipelines.odometry.OdometryOption()
    if hasattr(odo_opt, "depth_min"):
        odo_opt.depth_min = cfg.depth_min
    if hasattr(odo_opt, "depth_max"):
        odo_opt.depth_max = cfg.depth_max
    if hasattr(odo_opt, "depth_diff_max"):
        odo_opt.depth_diff_max = 0.05

    # ICP моделі
    model_pcd: Optional[o3d.geometry.PointCloud] = None
    last_model_update_idx = 0

    # Статичні snapshot-и (може бути кілька)
    static_snapshots: List[o3d.geometry.PointCloud] = []

    t_loop_start = time.perf_counter()

    for i, (t_rgb, rgb_path, _, depth_path) in enumerate(pairs[1:], start=1):
        print(f"[SLAM] Frame {i+1}/{num_frames}  t={t_rgb:.3f}", end="\r")
        pose_prev = pose.copy()

        color_rgb, color_bgr, depth = load_rgbd_from_paths(cfg, rgb_path,
depth_path)

        # --- YOLO + маска ---
        t0_yolo = time.perf_counter()
        y = run_yolo_and_mask_depth(color_bgr, depth, cfg, frame_idx=i,
timestamp=t_rgb)
        dt_yolo = time.perf_counter() - t0_yolo
        time_yolo += dt_yolo

        if cfg.save_tracked_objects and y.objects:
            all_objects.extend(y.objects)

        depth_tsdf = y.depth_masked
        depth_odom = depth_tsdf if cfg.use_dynamic_mask_for_odom else depth

        rgbd_odom = make_o3d_rgbd(color_rgb, depth_odom, cfg)
```

```
        rgbd_tsdf = make_o3d_rgbd(color_rgb, depth_tsdf, cfg)

        # --- RGB-D одометрія ---
        t0_odom = time.perf_counter()
        success, odo_T, _ = o3d.pipelines.odometry.compute_rgbd_odometry(
            prev_rgbd_odom,
            rgbd_odom,
            intrinsic,
            np.eye(4, dtype=np.float64),
            o3d.pipelines.odometry.RGBDOdometryJacobianFromHybridTerm(),
            odo_opt,
        )
        dt_odom = time.perf_counter() - t0_odom
        time_odom += dt_odom

        if not success:
            poses.append(pose_prev.copy())
            ts_est.append(t_rgb)
            prev_rgbd_odom = rgbd_odom
            frame_logs.append(FrameLog(
                idx=i, timestamp=t_rgb,
                odom_success=0, icp_ran=0, icp_accepted=0,
                motion_ok=0, keyframe_ok=0, integrated=0,
                reason="odometry_failed",
                t_yolo=dt_yolo, t_odom=dt_odom, t_icp=0.0, t_tsdf=0.0
            ))
            frame_anchor_kf_id[i] = last_key_kf_id
            frame_T_anchor_to_cam[i] = np.linalg.inv(last_key_pose) @
pose_prev
            continue

        pose_pred = pose_prev @ odo_T
        pose_new = pose_pred
        integrate_this_frame = True

        # --- ICP (frame-to-snapshot або frame-to-model) ---
        icp_ran = 0
        icp_ok = 0
        dt_icp = 0.0

        if cfg.use_frame_to_model_icp and cfg.icp_every > 0 and (i %
cfg.icp_every == 0):
            icp_ran = 1
            target_pcd: Optional[o3d.geometry.PointCloud] = None

            # якщо є snapshot-и – беремо останній
            if static_snapshots:
                last_snap = static_snapshots[-1]
                if len(last_snap.points) >= cfg.icp_min_model_points:
                    target_pcd = last_snap
            # інакше – модель з TSDF
            if target_pcd is None:
                if model_pcd is None or (i - last_model_update_idx) >=
cfg.icp_model_update_every:
                    tmp_pcd = volume.extract_point_cloud()
                    if len(tmp_pcd.points) >= cfg.icp_min_model_points:
```

```
                        tmp_pcd =
tmp_pcd.voxel_down_sample(cfg.icp_model_down_voxel)
                        tmp_pcd.estimate_normals()
                        model_pcd = tmp_pcd
                        last_model_update_idx = i
                    else:
                        model_pcd = None
                if model_pcd is not None and len(model_pcd.points) >=
cfg.icp_min_model_points:
                    target_pcd = model_pcd

            if target_pcd is not None:
                curr_pcd = o3d.geometry.PointCloud.create_from_depth_image(
                    o3d.geometry.Image(depth_tsdf.astype(np.float32)),
                    intrinsic,
                )
                curr_pcd =
curr_pcd.voxel_down_sample(cfg.icp_frame_down_voxel)

                if len(curr_pcd.points) > 0:
                    criteria =
o3d.pipelines.registration.ICPConvergenceCriteria(
                        max_iteration=int(cfg.icp_max_iter)
                    )
                    t0_icp = time.perf_counter()
                    icp = o3d.pipelines.registration.registration_icp(
                        curr_pcd,
                        target_pcd,

max_correspondence_distance=float(cfg.icp_max_corr_dist),
                        init=pose_pred,

estimation_method=o3d.pipelines.registration.TransformationEstimationPointToP
lane(),
                        criteria=criteria,
                    )
                    dt_icp = time.perf_counter() - t0_icp
                    time_icp += dt_icp

                    if icp.fitness >= cfg.icp_min_fitness and icp.inlier_rmse
<= cfg.icp_max_rmse:
                        pose_new = icp.transformation
                        icp_ok = 1
                    else:
                        integrate_this_frame = False
                else:
                    integrate_this_frame = False
            else:
                integrate_this_frame = False

        # --- Motion + radius gating ---
        motion_ok = 1
        reason_flag = "ok"

        T_rel = np.linalg.inv(pose_prev) @ pose_new
        if not is_motion_plausible(T_rel, cfg):
```

```
    motion_ok = 0
    reason_flag = "motion_gating"
    integrate_this_frame = False
    # позу не скидаємо, щоб не робити стрибків у траєкторії

if cfg.scene_max_radius > 0.0:
    cam_pos = pose_new[:3, 3]
    cam_dist = float(np.linalg.norm(cam_pos))
    if cam_dist > cfg.scene_max_radius:
        motion_ok = 0
        reason_flag = "scene_radius"
        integrate_this_frame = False
        # позу також не фіксуємо, тільки не інтегруємо кадр у TSDF

# --- Keyframe відбір ---
key_ok = 1
is_new_keyframe = False
if cfg.use_keyframes and integrate_this_frame:
    T_rel_kf = np.linalg.inv(last_key_pose) @ pose_new
    trans_kf, rot_kf = decompose_relative_transform(T_rel_kf)
    if (
        trans_kf < cfg.keyframe_min_trans
        and rot_kf < cfg.keyframe_min_rot_deg
        and (i - last_key_idx) < cfg.keyframe_max_interval
    ):
        key_ok = 0
        if reason_flag == "ok":
            reason_flag = "not_keyframe"
    else:
        last_key_pose = pose_new.copy()
        last_key_idx = i
        is_new_keyframe = True

# --- оновлення поз ---
pose = pose_new
poses.append(pose.copy())
ts_est.append(t_rgb)

# --- TSDF інтеграція ---
dt_tsdf = 0.0
if integrate_this_frame:
    t0_tsdf = time.perf_counter()
    volume.integrate(rgbd_tsdf, intrinsic, np.linalg.inv(pose))
    dt_tsdf = time.perf_counter() - t0_tsdf
    time_tsdf += dt_tsdf
else:
    if reason_flag == "ok":
        reason_flag = "skip"

# --- Anchor для кадру + keyframe/pose graph ---
if cfg.use_pose_graph and pose_graph is not None:
    if is_new_keyframe:
        gray = cv2.cvtColor(color_bgr, cv2.COLOR_BGR2GRAY)
        kps, des = compute_orb_features(gray, cfg)
        new_kf_id = len(keyframes)  # до append
```

```
                kf = Keyframe(
                    kf_id=new_kf_id,
                    frame_idx=i,
                    timestamp=t_rgb,
                    pose=pose.copy(),
                    gray=gray,
                    depth_tsdf=depth_tsdf.copy(),
                    keypoints=kps,
                    descriptors=des,
                )
                keyframes.append(kf)
                pose_graph.nodes.append(
                    o3d.pipelines.registration.PoseGraphNode(pose.copy())
                )

                # щільні локальні зв'язки: з двома попередніми keyframe-ами
                info = np.identity(6)
                for nb_id in range(max(0, new_kf_id - 2), new_kf_id):
                    nb_kf = keyframes[nb_id]
                    T_nb_new = np.linalg.inv(nb_kf.pose) @ kf.pose
                    pose_graph.edges.append(
                        o3d.pipelines.registration.PoseGraphEdge(
                            nb_kf.kf_id,
                            kf.kf_id,
                            T_nb_new,
                            uncertain=False,
                            information=info,
                        )
                    )

                # loop-closure на ORB + ICP
                add_loop_closure_edges(kf, keyframes, pose_graph, intrinsic,
cfg)

                last_key_kf_id = new_kf_id
                frame_anchor_kf_id[i] = new_kf_id
                frame_T_anchor_to_cam[i] = np.eye(4, dtype=np.float64)
            else:
                frame_anchor_kf_id[i] = last_key_kf_id
                anchor_pose = keyframes[last_key_kf_id].pose
                frame_T_anchor_to_cam[i] = np.linalg.inv(anchor_pose) @ pose
        else:
            frame_anchor_kf_id[i] = 0
            frame_T_anchor_to_cam[i] = pose.copy()

        frame_logs.append(FrameLog(
            idx=i, timestamp=t_rgb,
            odom_success=1,
            icp_ran=icp_ran, icp_accepted=icp_ok,
            motion_ok=motion_ok, keyframe_ok=key_ok,
            integrated=1 if integrate_this_frame else 0,
            reason=reason_flag,
            t_yolo=dt_yolo, t_odom=dt_odom, t_icp=dt_icp, t_tsdf=dt_tsdf
        ))

        # ---------- Static snapshots ----------
```

```
        take_primary = (
            cfg.static_snapshot_frame >= 0 and i == cfg.static_snapshot_frame
        )
        take_extra = (
            cfg.extra_snapshot_frame >= 0 and i == cfg.extra_snapshot_frame
        )
        if take_primary or take_extra:
            print(f"\n[SNAPSHOT] Capturing static point cloud at frame
{i}...")
            snap = volume.extract_point_cloud()
            if len(snap.points) > 0:
                snap = snap.voxel_down_sample(cfg.icp_model_down_voxel)
                snap.estimate_normals()
                static_snapshots.append(snap)

        prev_rgbd_odom = rgbd_odom

    print("\n[SLAM] Done.")

    total_time = time.perf_counter() - t_loop_start
    fps = len(ts_est) / total_time if total_time > 0 else 0.0
    print(f"[SLAM] Total time (loop): {total_time:.2f} s, frames:
{len(ts_est)}, FPS: {fps:.2f}")
    print(f"[TIME] YOLO: {time_yolo:.2f} s")
    print(f"[TIME] Odometry: {time_odom:.2f} s")
    print(f"[TIME] ICP: {time_icp:.2f} s")
    print(f"[TIME] TSDF integrate: {time_tsdf:.2f} s")

    # ---------- Pose graph global optimization ----------
    if cfg.use_pose_graph and pose_graph is not None and len(keyframes) > 0:
        print(f"[POSE_GRAPH] Nodes: {len(pose_graph.nodes)}, Edges:
{len(pose_graph.edges)}")
        option = o3d.pipelines.registration.GlobalOptimizationOption(
            max_correspondence_distance=cfg.loop_icp_max_corr_dist,
            edge_prune_threshold=0.25,
            reference_node=0,
        )
        print("[POSE_GRAPH] Running global optimization...")
        o3d.pipelines.registration.global_optimization(
            pose_graph,

o3d.pipelines.registration.GlobalOptimizationLevenbergMarquardt(),

o3d.pipelines.registration.GlobalOptimizationConvergenceCriteria(),
            option,
        )
        for kf in keyframes:
            kf.pose = pose_graph.nodes[kf.kf_id].pose.copy()

        poses_opt: List[np.ndarray] = []
        for idx in range(num_frames):
            anchor_id = frame_anchor_kf_id[idx]
            T_anchor = pose_graph.nodes[anchor_id].pose
            T_a2c = frame_T_anchor_to_cam[idx]
            poses_opt.append(T_anchor @ T_a2c)
        poses = poses_opt
```

```
        print("[POSE_GRAPH] Trajectory updated from optimized pose graph.")

    # ---------- Витягуємо мапу ----------
    mesh_raw = volume.extract_triangle_mesh()
    mesh_raw.compute_vertex_normals()
    pcd_raw = volume.extract_point_cloud()

    if remove_outliers and len(pcd_raw.points) > 0:
        print("[Postprocess] Removing outliers...")
        pcd_raw, _ = pcd_raw.remove_statistical_outlier(nb_neighbors=20,
std_ratio=2.0)

    if static_snapshots:
        print("[Postprocess] Merging with static snapshots...")
        for snap in static_snapshots:
            if len(snap.points) > 0:
                pcd_raw += snap
        pcd_raw = pcd_raw.voxel_down_sample(cfg.voxel_length)

    cfg.output_dir.mkdir(parents=True, exist_ok=True)
    mesh_path = cfg.output_dir / cfg.mesh_raw_file
    pcd_path = cfg.output_dir / cfg.pcd_raw_file

    o3d.io.write_triangle_mesh(str(mesh_path), mesh_raw)
    o3d.io.write_point_cloud(str(pcd_path), pcd_raw)

    print(f"[TSDF] Mesh saved to: {mesh_path}")
    print(f"[TSDF] Point cloud saved to: {pcd_path}")

    # ---------- Траєкторія + логи ----------
    traj_path = cfg.output_dir / cfg.traj_tum_file
    write_trajectory_tum(traj_path, ts_est, poses)
    print(f"[TRAJ] Saved estimated trajectory (TUM format) to: {traj_path}")

    flog_path = cfg.output_dir / cfg.frame_log_csv
    write_frame_log_csv(flog_path, frame_logs)
    print(f"[LOG] Frame log saved to: {flog_path}")

    if cfg.save_tracked_objects:
        obj_path = cfg.output_dir / cfg.objects_csv
        write_objects_csv(obj_path, all_objects)
        print(f"[LOG] Tracked objects saved to: {obj_path}")

    return ts_est, poses, volume, pcd_raw, mesh_raw


# ======================
#   ВІЗУАЛІЗАЦІЯ
# ======================
def visualize_geometries(geoms: List[o3d.geometry.Geometry]):
    print("Press 'q' in the Open3D window to exit.")
    o3d.visualization.draw_geometries(geoms)


# ======================
#   ATE RMSE (Sim3)
```

```
# ======================
def quat_to_rot(x: float, y: float, z: float, w: float) -> np.ndarray:
    q = np.array([x, y, z, w], dtype=np.float64)
    q = q / np.linalg.norm(q)
    x, y, z, w = q
    R = np.array([
        [1 - 2 * (y * y + z * z),     2 * (x * y - z * w),       2 * (x * z +
y * w)],
        [2 * (x * y + z * w),         1 - 2 * (x * x + z * z),   2 * (y * z -
x * w)],
        [2 * (x * z - y * w),         2 * (y * z + x * w),       1 - 2 * (x *
x + y * y)],
    ], dtype=np.float64)
    return R


def load_tum_groundtruth(gt_path: Path):
    ts_gt: List[float] = []
    poses_gt: List[np.ndarray] = []
    with open(gt_path, "r", encoding="utf-8") as f:
        for line in f:
            line = line.strip()
            if not line or line.startswith("#"):
                continue
            parts = line.split()
            if len(parts) != 8:
                continue
            t = float(parts[0])
            tx, ty, tz = map(float, parts[1:4])
            qx, qy, qz, qw = map(float, parts[4:8])
            R = quat_to_rot(qx, qy, qz, qw)
            T = np.eye(4, dtype=np.float64)
            T[:3, :3] = R
            T[:3, 3] = np.array([tx, ty, tz], dtype=np.float64)
            ts_gt.append(t)
            poses_gt.append(T)
    return np.array(ts_gt, dtype=np.float64), poses_gt


def associate_for_ate(
    ts_est: List[float],
    poses_est: List[np.ndarray],
    ts_gt: np.ndarray,
    poses_gt: List[np.ndarray],
    max_diff: float = 0.02
):
    ts_est_arr = np.array(ts_est, dtype=np.float64)
    matches_est: List[int] = []
    matches_gt: List[int] = []

    for i, t_e in enumerate(ts_est_arr):
        idx = int(np.argmin(np.abs(ts_gt - t_e)))
        t_g = ts_gt[idx]
        if abs(t_g - t_e) < max_diff:
            matches_est.append(i)
            matches_gt.append(idx)
```

```
    if len(matches_est) == 0:
        raise RuntimeError("No ATE matches found. Increase max_diff?")

    P = []
    Q = []
    for i_e, i_g in zip(matches_est, matches_gt):
        Te = poses_est[i_e]
        Tg = poses_gt[i_g]
        P.append(Te[:3, 3])
        Q.append(Tg[:3, 3])

    return np.array(P, dtype=np.float64), np.array(Q, dtype=np.float64)


def align_trajectories_sim3(P: np.ndarray, Q: np.ndarray):
    """
    Umeyama alignment: знаходить масштаб s, обертання R і зсув t,
    що мінімізують sum_i || s * R * P_i + t - Q_i ||^2.
    """
    assert P.shape == Q.shape
    N = P.shape[0]

    mu_P = P.mean(axis=0)
    mu_Q = Q.mean(axis=0)

    P_centered = P - mu_P
    Q_centered = Q - mu_Q

    Sigma = (Q_centered.T @ P_centered) / N  # 3x3

    U, D, Vt = np.linalg.svd(Sigma)
    S = np.eye(3)
    if np.linalg.det(U) * np.linalg.det(Vt) < 0:
        S[2, 2] = -1.0

    R = U @ S @ Vt

    var_P = (P_centered ** 2).sum() / N
    scale = np.sum(D * np.diag(S)) / var_P

    t = mu_Q - scale * R @ mu_P
    return scale, R, t


def compute_ate_rmse(
    cfg: ConfigTUM,
    ts_est: List[float],
    poses_est: List[np.ndarray],
    max_time_diff: float = 0.02
):
    seq_path = tum_seq_path(cfg)
    gt_path = seq_path / "groundtruth.txt"
    if not gt_path.exists():
        raise FileNotFoundError(f"groundtruth.txt not found: {gt_path}")
```

```
    ts_gt, poses_gt = load_tum_groundtruth(gt_path)

    P, Q = associate_for_ate(ts_est, poses_est, ts_gt, poses_gt,
max_diff=max_time_diff)

    # Sim(3) вирівнювання (масштаб + R + t)
    s, R, t = align_trajectories_sim3(P, Q)

    P_aligned = (s * (R @ P.T)).T + t
    errors = np.linalg.norm(P_aligned - Q, axis=1)

    rmse = float(np.sqrt(np.mean(errors ** 2)))
    mean_err = float(np.mean(errors))
    max_err = float(np.max(errors))

    print(f"[ATE] Matches: {len(errors)} frames")
    print(f"[ATE] scale: {s:.4f}")
    print(f"[ATE] RMSE (Sim3): {rmse:.4f} m")
    print(f"[ATE] mean: {mean_err:.4f} m, max: {max_err:.4f} m")
    return rmse, mean_err, max_err


# ======================
#   MAIN
# ======================
if __name__ == "__main__":
    print("[MAIN] Starting YOLO-seg + ByteTrack + TSDF SLAM on fr3
(walking_xyz)...")

    ts_est, poses_est, volume, pcd_raw_filtered, mesh_raw =
run_tum_yolo_slam_tsdf(
        cfg,
        max_frames=100,
        remove_outliers=True,
    )

    print("[MAIN] Computing ATE RMSE...")
    rmse, mean_err, max_err = compute_ate_rmse(cfg, ts_est, poses_est,
max_time_diff=0.02)

    print("[MAIN] Done.")
    print(f"[MAIN] ATE RMSE: {rmse:.4f} m (mean={mean_err:.4f},
max={max_err:.4f})")

    visualize_geometries([mesh_raw, pcd_raw_filtered])
```