МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ АНАЛІЗУ ТА ПЕРЕДБАЧЕННЯ РУХУ ОБ'ЄКТІВ У ДИНАМІЧНОМУ СЕРЕДОВИЩІ»**

Виконав: студент 2 курсу, групи 8.1224
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
Д. А. Резця
(ініціали та прізвище)
Керівник доцент кафедри комп'ютерних наук, доцент, к.т.н Борю С.Ю.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент завідувач кафедри програмної інженерії, к.ф.-м.н., доцент Лісняк А.О.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти магістр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
Завідувач кафедри комп'ютерних наук, д.т.н., професор

_____ Шило Г.М.
(підпис)

" ____ " _____ 2025 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Резцю Данилу Альбертовичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка інтелектуальної системи аналізу та передбачення руху об'єктів у динамічному середовищі

керівник роботи Борю Сергій Юрійович, к.т.н., доцент
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 28 » лютого 2025 року № 326-с

2. Строк подання студентом роботи 06.06.2026

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка інтелектуальної системи аналізу та передбачення руху об'єктів у динамічному середовищі.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 06.03.2025

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	20.03.2025	
2.	Збір вихідних даних.	06.12.2025	
3.	Обробка методичних та теоретичних	21.01.2026	
	джерел.		
4.	Розробка першого та другого розділу.	01.02.2026	
5.	Розробка третього та четвертого розділів.	27.05.2026	
6.	Оформлення та нормоконтроль	10.06.2026	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	20.06.2026	

Студент _____ Д. А. Резця
(підпис) (ініціали та прізвище)

Керівник роботи _____ С. Ю. Борю
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г. Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка інтелектуальної системи аналізу та передбачення руху об’єктів у динамічному середовищі»: 88 с., 14 рис., 11 табл., 31 джерел, 9 додатків.

ВІДСТЕЖЕННЯ ОБ’ЄКТІВ, ДИНАМІЧНЕ СЕРЕДОВИЩЕ, КОМПЕНСАЦІЯ РУХУ КАМЕРИ, КОМП’ЮТЕРНИЙ ЗІР, ПРОГНОЗУВАННЯ ТРАЄКТОРІЙ, YOLOv12.

Об’єкт дослідження – процес аналізу та передбачення руху об’єктів у динамічному середовищі з використанням методів глибокого навчання та комп’ютерного зору.

Мета роботи полягає у розробці інтелектуальної системи аналізу та передбачення руху об’єктів у динамічному середовищі на основі нейромережевої моделі YOLOv12, а також її експериментальному тестуванні.

Методи дослідження – теоретичний та порівняльний аналіз, машинне навчання (детекція об’єктів, кластеризація), математичне моделювання (екстраполяція, проективна геометрія), емпіричне дослідження та тестування.

У роботі досліджено еволюцію методів детекції та трекінгу об’єктів. Спроєктовано модульну конвеєрну архітектуру системи. Створено кастомний датасет за допомогою симуляції в Unreal Engine 5 та навчено модель YOLOv12n, що забезпечує високу точність детекції. Реалізовано алгоритм компенсації руху камери за допомогою матриці гомографії та оптичного потоку. Розроблено підсистему аналізу траєкторій, яка включає обчислення швидкості, детекцію відскоків та виявлення стін методом DBSCAN, а також підсистему прогнозування зіткнень. Проведено експериментальне тестування, яке підтвердило виконання функціональних та нефункціональних вимог до продуктивності та точності системи.

SUMMARY

Master’s Qualifying Thesis "Development of an intelligent system for object movement analysis and prediction in dynamic environments": 88 pages, 14 figures, 11 tables, 31 sources, 9 supplements.

OBJECT TRACKING, DYNAMIC ENVIRONMENT, CAMERA MOTION COMPENSATION, COMPUTER VISION, TRAJECTORY PREDICTION, YOLOv12.

The object of study is the process of analyzing and predicting object movement in a dynamic environment using deep learning and computer vision methods.

The purpose of the work is to develop an intelligent system for object movement analysis and prediction in a dynamic environment based on the YOLOv12 neural network model, as well as its experimental testing.

Research methods – theoretical and comparative analysis, machine learning (object detection, clustering), mathematical modeling (extrapolation, projective geometry), empirical research and testing.

The paper investigates the evolution of object detection and tracking methods. A modular pipeline architecture of the system is designed. A custom dataset was created using simulation in Unreal Engine 5, and the YOLOv12n model was trained, providing high detection accuracy. An algorithm for camera motion compensation using a homography matrix and optical flow was implemented. A trajectory analysis subsystem was developed, which includes velocity calculation, bounce detection, and wall detection using the DBSCAN method, as well as a collision prediction subsystem. Experimental testing was conducted, which confirmed the fulfillment of functional and non-functional requirements for the system’s performance and accuracy.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 8
1 Аналіз предметної області та існуючих рішень 10
1.1 Визначення поняття динамічного середовища та об’єкта 10
1.2 Огляд методів детекції об’єктів 12
1.3 Огляд методів відстеження об’єктів 17
1.4 Методи прогнозування траєкторій руху об’єктів 21
1.5 Постановка задачі дослідження 25
2 Проєктування інтелектуальної системи 28
2.1 Вимоги до системи (функціональні та нефункціональні) 28
2.2 Розробка загальної конвеєрної архітектури системи 29
2.3 Проєктування підсистем захоплення, детекції та трекінгу 32
2.4 Проєктування алгоритмів аналізу траєкторій та прогнозування 34
2.5 Проєктування методу компенсації руху камери 36
3 Програмна реалізація системи 39
3.1 Вибір стеку технологій та середовища розробки 39
3.2 Підготовка датасету та навчання моделі YOLOv12 41
3.3 Реалізація конвеєра детекції та відстеження об’єктів 44
3.4 Реалізація стабілізації координат (компенсація руху камери) 46
3.5 Реалізація аналізу траєкторій та прогнозування руху 48
3.6 Реалізація псевдо-3D аналізу глибини на основі площі об’єкта 50
3.7 Реалізація візуалізації та запису результатів 51
4 Експериментальне дослідження та тестування 54
4.1 Опис тестового середовища 54
4.2 Оцінка точності детекції та відстеження 56

4.3 Оцінка точності прогнозування траєкторій 58
4.4 Оцінка продуктивності системи 59
4.5 Аналіз результатів 60
4.6 Висновок до розділу 4 62
Висновки 64
Перелік посилань 66
Додаток А Головний конвеєр системи (main.py) 69
Додаток Б Конфігурація параметрів системи (config.py) 72
Додаток В Модуль детекції та трекінгу (detector.py) 73
Додаток Г Модуль компенсації руху камери (stabilizer.py) 74
Додаток Д Модуль аналізу траєкторій (tracker.py) 78
Додаток Е Модуль виявлення стін DBSCAN (analyzer.py) 81
Додаток Ж Модуль прогнозування зіткнень (predictor.py) 83
Додаток И Модуль візуалізації (visualizer.py) 85
Додаток К Модуль вводу\виводу відео (video-io.py) 88

ВСТУП

Розвиток алгоритмів комп'ютерного зору дозволяє автоматизувати дедалі складніші процеси обробки відеоданих. Сьогодні однією з ключових проблем у цій галузі залишається прогнозування траєкторій об'єктів в умовах динамічної сцени. Такі рішення критично необхідні для автономного транспорту, розумного відеоспостереження, спортивної аналітики та робототехніки. Головний виклик під час проєктування подібних систем полягає не просто у фіксації цілей на кадрі. Програмний комплекс має аналізувати кінематику руху, реконструювати просторову структуру середовища та передбачати ймовірні колізії ще до їх фактичного настання.

Завдяки еволюції лінійки YOLO [8, 12, 31] аналіз відео в реальному часі став надійним і швидким інструментом. Незважаючи на це, традиційні алгоритми переважно орієнтовані на статичну зйомку. Коли ж камера рухається, а фон стає неоднорідним, базової детекції виявляється недостатньо. Виникає необхідність розробки більш глибокої аналітичної системи. Вона має стабілізувати координати, будувати математичну модель перешкод на основі поведінки цілей та алгоритмічно прогнозувати їхні перетини.

Мета роботи полягає у розробці програмного забезпечення на базі YOLOv12 для розпізнавання, трекінгу та прогнозування руху об'єктів. Це рішення здатне ефективно працювати в динамічному середовищі з використанням рухомої камери.

Завдання дослідження. Для досягнення поставленої мети сформульовано та вирішено такі завдання:

- провести аналіз методів детекції та трекінгу об'єктів, обґрунтувати вибір YOLOv12n та BoT-SORT;
- спроєктувати модульну конвеєрну архітектуру інтелектуальної системи, визначити функціональні та нефункціональні вимоги;

- розробити модуль стабілізації відеоряду (Lucas-Kanade та гомографія) із функцією виключення рухомих елементів сцени під час розрахунків;
- створити алгоритми кінематичного аналізу: сюди входить фіксація точок відбиття, пошук площин та передбачення перетинів траєкторій;
- зібрати тренувальні дані засобами Unreal Engine 5, навчити алгоритм детекції та підтвердити його точність експериментальним шляхом.

Наукова новизна одержаних результатів. Головною науковою відмінністю розробленої системи є:

- імплементація методу компенсації руху камери на основі накопичення гомографій із маскуванням рухомих об’єктів, що дозволяє стабілізувати координати в умовах динамічного фону без використання апаратних засобів стабілізації;
- розробка алгоритму неявного виявлення статичних перешкод (стін) виключно на основі аналізу точок відскоків рухомих об’єктів з використанням кластеризації DBSCAN та апроксимації прямими;
- інтеграція алгоритмів детекції відскоків (за кутом між векторами руху), кластеризації та ray-casting прогнозування у єдиний конвеєр, що дозволяє системі математично описувати геометрію простору та передбачати зіткнення об’єктів зі стінами.

Практичне значення одержаних результатів. Розроблений програмний продукт може бути використаний для вирішення широкого спектра завдань моніторингу, відстеження об’єктів та автоматизованого реагування в умовах змінних ракурсів камери. Профілювання підтвердило швидкість ~14.7 FPS, що є придатним для роботи у режимі, наближеному до реального часу. Спроєктована модульна конвеєрна архітектура дозволяє легко інтегрувати систему з існуючими відеопотоками або симуляційними середовищами для проведення подальших досліджень.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Визначення поняття динамічного середовища та об’єкта

Задачі комп’ютерного зору, пов’язані з аналізом руху, передбачають роботу у середовищах різного ступеня складності. Для коректного формулювання задачі дослідження необхідно чітко визначити ключові поняття: «динамічне середовище» та «об’єкт».

Динамічне середовище – це сцена, стан якої змінюється з плином часу. На відміну від статичної сцени (наприклад, фотографії інтер’єру), динамічне середовище характеризується наявністю рухомих елементів, зміною освітлення, можливим переміщенням точки спостереження (камери), а також взаємодією об’єктів між собою та з оточенням [1]. У межах цього дослідження під динамічним середовищем ми розуміємо відеопотік, згенерований на базі 3D-сцени в Unreal Engine 5. Це простір з рухомою камерою, де група об’єктів безперервно переміщується, взаємодіючи як одне з одним, так і з фізичними перешкодами.

Динамічне середовище можна описати як функцію стану, яка безперервно змінюється з часом:

$$S(t) = O_1(t); O_2(t); \dots; O_n(t); E(t); C(t). \quad (1.1)$$

У цій моделі кінематика та поточне положення кожного і-го об’єкта задається компонентою $O_i(t)$. Інформація про нерухомі межі та перешкоди (загальну топологію простору) винесена в параметр $E(t)$. Окремо алгоритм враховує характеристики самої камери – її просторові координати, кут нахилу та фокусну відстань об’єднано у змінній $C(t)$.

З погляду комп’ютерного зору, об’єкт – це змістовна частина кадру, що має чіткі просторові межі та успішно відділяється від загального фону. Технічно

об’єкт описується рамкою (bounding box), тобто прямокутником, який малюється навколо об’єкта. Bounding box визначається четвіркою параметрів:

$$B = (c_x, ; c_y, ; w, ; h) \tag{1.2}$$

де c_x, c_y – координати центру рамки;

w– ширина;

h– висота (у пікселях).

У цій роботі об’єктами є кулі (м’ячі), що рухаються в замкненому просторі. Кожен об’єкт характеризується:

- положенням c_x, c_y – координати центру bounding box у кадрі;
- розміром – ширина w та висота h рамки, що дозволяє оцінити радіус об’єкта як $r = \frac{\max(w,h)}{2}$;
- швидкістю (v_x, v_y) – вектор переміщення, обчислений як різниця координат між кадрами;
- ідентифікатором (track ID) – унікальний номер, що зберігається між кадрами завдяки алгоритму трекінгу.

Задачі аналізу руху об’єктів у динамічному середовищі можна класифікувати за рівнем складності:

- детекція (detection) – визначення наявності та розташування об’єктів на окремому кадрі. Результат – набір bounding box із класами та ймовірностями;
- трекінг (tracking, відстеження) – встановлення відповідності між об’єктами на послідовних кадрах. Кожному об’єкту присвоюється унікальний ідентифікатор (ID), який зберігається протягом його перебування в полі зору камери;

- аналіз траєкторій – обчислення характеристик руху (швидкість, прискорення, кривизна траєкторії), виявлення подій (відскоки, зміна напрямку, зіткнення між об’єктами);
- прогнозування (prediction) – екстраполяція майбутнього положення об’єкта на основі його історії руху та знань про середовище (наприклад, розташування стін).

Ці чотири рівні утворюють ієрархічний конвеєр, де кожен наступний рівень використовує результати попереднього. Саме така конвеєрна архітектура покладена в основу розробленої в даній роботі системи.

Додатковим фактором складності є рух камери. Коли камера нерухома, координати об’єктів у кадрі безпосередньо відповідають їхнім реальним переміщенням. Однак при русі камери (панорамування, обертання) координати об’єктів змінюються навіть для нерухомих точок сцени. Це вимагає компенсації руху камери (стабілізації координат) перед аналізом траєкторій – задача, яка детально розглядається в підрозділі 2.5.

1.2 Огляд методів детекції об’єктів

Детекція об’єктів (object detection) – одна з фундаментальних задач комп’ютерного зору, що полягає у визначенні місцезнаходження та класу кожного об’єкта на зображенні. Еволюцію методів детекції можна розділити на три основні етапи.

Класичні методи. До ери глибокого навчання переважали підходи на основі ручних ознак (handcrafted features). Найвідомішим є метод HOG + SVM (Histogram of Oriented Gradients + Support Vector Machine) [2], запропонований Dalal та Triggs у 2005 році. Алгоритм обчислює гістограму орієнтацій градієнтів у локальних комірках зображення, після чого використовує лінійний SVM-класифікатор для розрізнення об’єкта та фону.

Іншим поширеним підходом був метод Віоли-Джонса (Viola-Jones) [3], що використовує каскад слабких класифікаторів на основі ознак Хаара. Цей метод забезпечував високу швидкість роботи, проте обмежувався простими об’єктами (обличчя) і погано масштабувався на довільні класи.

Основні недоліки класичних методів:

- необхідність ручного проєктування ознак для кожного типу об’єктів;
- низька робастність до змін масштабу, ракурсу та освітлення;
- обмежена здатність до узагальнення на нові категорії.

Значним кроком у розвитку двоетапних нейромережевих детекторів стала запропонована у 2014 році архітектура R-CNN (Region-based Convolutional Neural Network) [4]. Її робота базується на двох послідовних етапах. Спочатку алгоритм Selective Search формує близько 2000 потенційних регіонів інтересу (region proposals), які з високою ймовірністю містять об’єкти. На другому етапі кожен такий фрагмент-кандидат масштабується до єдиного розміру та передається на вхід згортковій нейромережі (CNN) для подальшої класифікації.

Еволюція двоетапних детекторів:

- R-CNN (2014) – окрема CNN для кожного регіону, дуже повільно (~47 с на зображення);
- Fast R-CNN [5] (2015) – спільна CNN для всього зображення + ROI Pooling, ~0.3 с;
- Faster R-CNN [6] (2015) – заміна Selective Search на Region Proposal Network (RPN), ~0.2 с, метрика mAP@50 = 73.2% на Pascal VOC.

Попри високу точність розпізнавання, двоетапні детектори виявляються занадто повільними для потокової відеообробки. Це суттєвий недолік, оскільки аналіз динамічного середовища вимагає миттєвої реакції алгоритму без втрати кадрів.

Архітектури на два етапи дають хорошу точність, але їхня повільність робить потокову обробку неможливою. У динамічному середовищі система повинна реагувати миттєво, без втрати сусідніх кадрів.

SSD (Single Shot MultiBox Detector) [7], запропонований Liu et al. у 2016 році, виконує передбачення на кількох масштабах feature map одночасно. SSD досягає mAP = 74.3% на Pascal VOC при швидкості 59 FPS на GPU NVIDIA Titan X.

Саме тому оптимальним вибором для потокової обробки вважається сімейство архітектур YOLO (You Only Look Once) [8]. Розробники цих моделей відмовилися від традиційного виділення регіонів і звели детекцію до задачі просторової регресії. Процес виглядає так: кадр сегментується у вигляді сітки SxS, після чого кожна її комірка самостійно і паралельно вираховує три компоненти. Вона генерує координати обмежувальних рамок (c_x, c_y, w, h), визначає рівень впевненості у наявності об’єкта та розраховує ймовірності для відповідних класів.

Еволюція YOLO:

- YOLOv1 [8] (2016) – базова архітектура, 45 FPS, mAP = 63.4% (VOC);
- YOLOv2 [9] (2017) – batch normalization, anchor boxes, mAP = 78.6%;
- YOLOv3 [10] (2018) – багатомасштабне передбачення (Feature Pyramid Network), backbone Darknet-53;
- YOLOv4 [11] (2020) – CSPDarknet53, Mish activation, CIoU loss;
- YOLOv5 (2020) – реімплементація на PyTorch, зручний API;
- YOLOv8 [12] (2023) – У цьому релізі алгоритм позбувся якірних рамок (перейшов на anchor-free детекцію) та отримав потужніші інструменти для аугментації датасетів;
- YOLOv12 (2025) [13]: Розробники інтегрували механізм уваги (attention mechanism) безпосередньо у backbone моделі, дозволивши їй краще фокусуватися на релевантних ділянках кадру.

Обґрунтування вибору YOLOv12. Для розробленої системи обрано модель YOLOv12 у конфігурації nano (YOLOv12n) з таких причин:

- механізм уваги. На відміну від попередніх версій YOLO, які повністю базувались на згорткових мережах (CNN), YOLOv12 інтегрує attention-блоки. Це дозволяє моделі ефективніше виділяти об’єкти на складних фонах та при частковому перекритті;

- швидкість інференсу. Конфігурація nano (n) – найлегша у сімействі: ~5.6 млн параметрів. Це забезпечує роботу в реальному часі на GPU середнього рівня;
- Transfer learning. Модель попередньо навчена на датасеті COCO (80 класів, 330K зображень), що забезпечує якісну ініціалізацію ваг і дозволяє дотренувати модель на кастомному датасеті навіть невеликого обсягу;
- вбудований трекер. Бібліотека Ultralytics, яка реалізує YOLOv12, надає інтегрований трекер BoT-SORT/ByteTrack, що дозволяє виконувати детекцію та трекінг одним викликом model.track();
- екосистема. Ultralytics забезпечує зручний Python API для навчання, інференсу та експорту моделі, що спрощує інтеграцію в конвеєр обробки;

Архітектура YOLOv12n складається з трьох основних частин (рис. 1.1):

- Backbone – витягує ознаки з вхідного зображення. У YOLOv12 backbone включає C3k2-блоки та A2C2f-блоки (Area-to-Center attention), що поєднують локальні згорткові ознаки з глобальним контекстом через механізм уваги;
- Neck – агрегує ознаки з різних масштабів (Feature Pyramid Network, FPN + PAN), забезпечуючи детекцію як великих, так і дрібних об’єктів;
- Head – decoupled head (розділена голова): окремі гілки для регресії координат bounding box та класифікації. Використовує anchor-free підхід – модель безпосередньо передбачає центр об’єкта, ширину та висоту, без прив’язки до заздалегідь визначених anchor boxes.

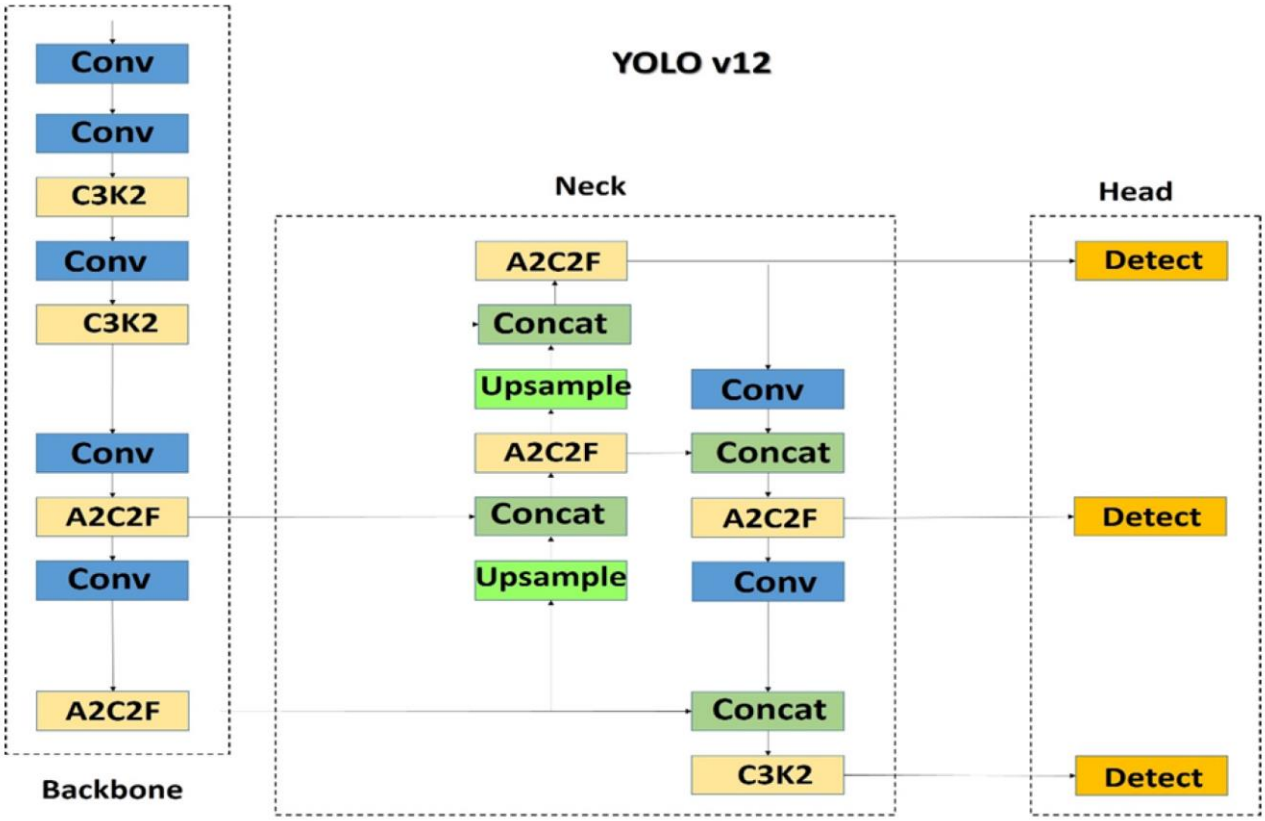


Рисунок 1.1 – Архітектура YOLOv12 з блоками «area attention» та R-ELAN[31]

Для оцінки якості детекції використовується метрика mAP (mean Average Precision) – середнє значення площі під кривою Precision-Recall для всіх класів. Метрика mAP@50 обчислюється при порозі IoU (Intersection over Union) = 0.5:

$$IoU = \frac{|B_{pred} \cap B_{gt}|}{|B_{pred} \cup B_{gt}|} \tag{1.3}$$

де B_{pred} – передбачений bounding box;

B_{gt} – істинний (ground truth).

Якщо $IoU \geq 0.5$, передбачення вважається вірним (True Positive), інакше – хибним (False Positive).

Більш суворою метрикою є mAP@50-95 – середнє mAP, обчислене для порогів IoU від 0.5 до 0.95 з кроком 0.05. Ця метрика краще відображає якість

локалізації. Порівняння результатів різних архітектур детекції на датасеті COCO наведено у таблиці 1.1.

Таблиця 1.1 – Порівняння архітектур детекції на COCO val2017

Модель	Параметри, млн	mAP@50, %	mAP@50-95, %	FPS (GPU)
YOLOv5n	1.9	45.7	28.0	~400
YOLOv8n	3.2	52.6	37.3	~350
YOLOv12n	5.6	55.2	40.6	~310
Faster R-CNN	41.1	59.1	42.0	~25

Як видно з таблиці, YOLOv12n забезпечує найвищу точність серед одноетапних nano-моделей при збереженні швидкості, достатньої для обробки в реальному часі (>300 FPS при роздільній здатності 640×640 на GPU NVIDIA RTX).

1.3 Огляд методів відстеження об’єктів

Відстеження об’єктів (object tracking) – це задача встановлення відповідності між детекціями об’єктів на послідовних кадрах відеопотоку та підтримки їхніх унікальних ідентифікаторів у часі. Розрізняють два основні класи задач: відстеження одного об’єкта (Single Object Tracking, SOT) та відстеження багатьох об’єктів (Multiple Object Tracking, MOT). У контексті даної роботи, де на сцені одночасно присутні 16 рухомих об’єктів, розглядається саме задача MOT.

Домінуючим підходом у MOT є tracking-by-detection – парадигма, за якої на кожному кадрі спочатку виконується детекція всіх об’єктів (незалежно від попередніх кадрів), а потім результати детекції асоціюються з існуючими треками. Це розділяє задачу на два незалежні етапи (рис. 1.2):

– детекція – отримання набору bounding boxes на поточному кадрі (у нашому випадку YOLOv12);
– асоціація – зіставлення кожного bounding box із вже існуючим треком або створення нового треку.

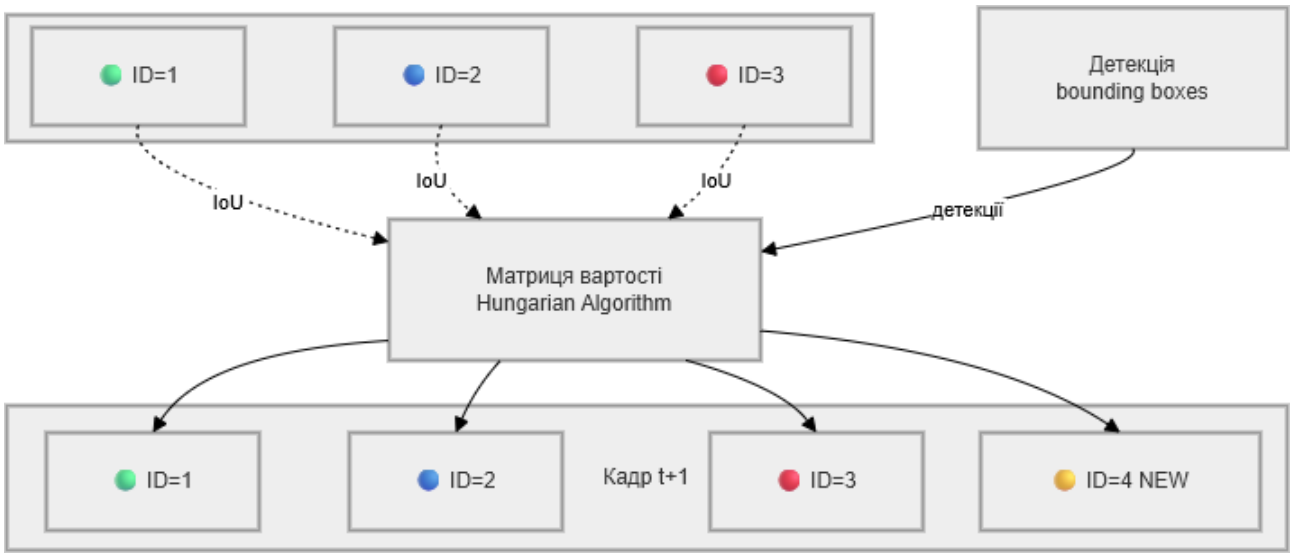


Рисунок 1.2 – Схема парадигми tracking-by-detection: асоціація детекцій між кадрами

Задача асоціації формулізується як задача призначення (assignment problem). Будується матриця вартості $C \in \mathbb{R}^{M \times N}$, де M – кількість існуючих треків, N – кількість детекцій на поточному кадрі. Елемент C_{ij} відображає «вартість» призначення детекції j до треку i. Оптимальне призначення знаходиться за допомогою Угорського алгоритму (Hungarian algorithm), що мінімізує загальну вартість за $O(n^3)$.

Алгоритм SORT (Simple Online and Realtime Tracking) [14] – базовий алгоритм tracking-by-detection, запропонований Bewley et al. у 2016 році [13]. SORT використовує фільтр Калмана для передбачення положення кожного треку на наступному кадрі та метрику IoU для побудови матриці вартості:

$$C_{ij} = 1 - IoU\left(B_i^{pred}, B_j^{det}\right). \quad (1.4)$$

де B_i^{pred} – передбачений bounding box і-го треку;

B_j^{det} – j-та детекція.

У алгоритмі SORT поточний стан цілі описується вектором:

$$\mathbf{x} = \left[c_x, c_y, s, r, \dot{c}_x, \dot{c}_y, \dot{s}\right]^T. \quad (1.5)$$

де фігурують площа рамки (bounding box) $s = w \cdot h$, та співвідношення сторін $r = w/h$, тоді як крапка вказує на швидкість їхньої зміни.

Попри чудову швидкодію (понад 260 кадрів за секунду), алгоритм є вразливим до оклюзій: навіть короткочасне перекриття об’єкта часто призводить до збою та присвоєння йому нового ідентифікатора (ID switch).

Алгоритм DeepSORT [15] (Deep Simple Online and Realtime Tracking, 2017) розширює SORT додаванням зовнішнього вигляду (appearance) як ознаки для асоціації. Для кожного об’єкта витягується вектор ознак $\mathbf{a} \in \mathbb{R}^{128}$ за допомогою глибокої нейромережі (Re-ID model), навченої розрізняти об’єкти за їхнім зовнішнім виглядом.

Комбінована метрика вартості:

$$C_{ij} = \lambda \cdot d_{Mahal}(i,j) + (1-\lambda) \cdot d_{cos}\left(\mathbf{a}_i, \mathbf{a}_j\right) \quad (1.6)$$

де d_{Mahal} – відстань Махаланобіса (враховує невизначеність фільтра Калмана);

d_{cos} – косинусна відстань між векторами ознак;

λ – ваговий коефіцієнт.

Алгоритми ByteTrack та BoT-SORT [16] – алгоритм, що використовує детекції з низькою впевненістю (low-confidence detections) на другому етапі асоціації. Стандартний підхід відкидає детекції з впевненістю нижче порогу, через що частково перекриті об’єкти можуть втрачатися. ByteTrack виконує дворівневу асоціацію:

– перший етап – асоціація треків з високовпевненими детекціями ($conf \geq \tau_{high}$);
– другий етап – незіставлені треки асоціюються з низьковпевненими детекціями ($\tau_{low} \leq conf < \tau_{high}$).

BoT-SORT [17] розширює базові підходи трекінгу модулем компенсації руху камери (CMC), який стабілізує фільтр Калмана за допомогою гомографії.

Бібліотека Ultralytics, що використовується в даній роботі, інтегрує трекер BoT-SORT як трекер за замовчуванням. Виклик model.track(frame, persist=True) виконує одночасно:

– пряма детекція об'єктів (робота моделі YOLO);
– перерахунок стану фільтра Калмана для активних цілей;
– зв'язування поточних детекцій із попередніми траєкторіями;
– видача унікальних ID.

Безперервність супроводу під час покадрового аналізу відео забезпечується аргументом persist=True, що відповідає за кешування історії трекінгу між ітераціями. Порівняння алгоритмів трекінгу наведено у таблиці 1.2.

Таблиця 1.2 – Порівняння алгоритмів відстеження на MOT17

Алгоритм	MOTA, %	IDF1, %	ID Sw.	FPS
SORT	59.8	56.2	4852	>260
DeepSORT	61.4	62.2	2008	~40
ByteTrack	80.3	77.3	2196	~65
BoT-SORT	80.5	80.2	1212	~30

Де MOTA (Multiple Object Tracking Accuracy) – інтегральна метрика якості трекінгу; IDF1 – гармонічне середнє точності та повноти ідентифікації; ID Sw. – кількість переключень ідентифікаторів.

1.4 Методи прогнозування траєкторій руху об’єктів

Прогнозування траєкторій – це задача передбачення майбутнього положення об’єкта на основі його історії руху. Цей етап аналізу є фундаментальним для всієї системи, оскільки переводить програму від звичайного спостереження до повноцінного прогнозування. Замість простого збору координат ми отримуємо інструмент, який вираховує напрямок руху, передбачає моменти зіткнень зі стінами та знаходить точки перетину траєкторій.

Найпростіший метод прогнозування – лінійна екстраполяція (constant velocity model). Передбачається, що об’єкт продовжить рух з поточною швидкістю без зміни напрямку:

$$P(t + \Delta t) = P(t) + \mathbf{v} \cdot \Delta t \tag{1.7}$$

де $P(t) = \big(x(t), y(t)\big)$ – поточне положення об’єкта;

$\mathbf{v} = \big(v_x, v_y\big)$ – вектор швидкості (пікселі/кадр);

Δt – горизонт передбачення (у кадрах).

Швидкість обчислюється як різниця координат між першою та останньою точкою буфера історії довжиною N кадрів:

$$v_x = \frac{x_t - x_{t-N}}{N}, \quad v_y = \frac{y_t - y_{t-N}}{N}. \tag{1.8}$$

Переваги лінійної екстраполяції: простота реалізації, мінімальні обчислювальні витрати, відсутність необхідності в навчальних даних. Недоліки: неспроможність моделювати зміну напрямку (відскоки, повороти), накопичення похибки на довгому горизонті.

У розробленій системі лінійна екстраполяція використовується як основний метод прогнозування з горизонтом $\Delta t = 1$ секунда (60 кадрів при 60 FPS відео).

Фільтр Калмана [18] – рекурсивний алгоритм оцінювання стану лінійної динамічної системи в умовах шуму. Фільтр працює у два етапи:

Етап передбачення (prediction):

$$\hat{\mathbf{x}}k|k-1 = F \cdot \hat{\mathbf{x}}k-1|k-1, \tag{1.9}$$

$$P_{k|k-1} = F \cdot P_{k-1|k-1} \cdot F^T + Q. \tag{1.10}$$

Етап оновлення (update):

$$K_k = P_{k|k-1} \cdot H^T \cdot \left(H \cdot P_{k|k-1} \cdot H^T + R\right)^{-1}, \tag{1.11}$$

$$\hat{\mathbf{x}}k|k = \hat{\mathbf{x}}k|k-1 + K_k \cdot (\mathbf{z}k - H \cdot \hat{\mathbf{x}}k|k-1) \tag{1.12}$$

де $\hat{\mathbf{x}}$ – вектор стану (положення та швидкість);

F – матриця переходу стану;

P – матриця коваріації похибки;

Q – шум процесу; R – шум вимірювань;

H – матриця спостереження;

K – коефіцієнт Калмана;

z – вектор вимірювань.

Фільтр Калмана використовується в алгоритмі BoT-SORT (інтегрованому в Ultralytics) для передбачення положення трекованих об’єктів на наступному кадрі. Це покращує точність асоціації, особливо при пропуску кадрів або тимчасовому зникненні об’єкта.

Для моделювання нелінійних траєкторій використовується поліноміальна регресія. Траєкторія апроксимується поліномом ступеня p:

$$x(t) = a_0 + a_1 t + a_2 t^2 + \cdots + a_p t^p. \tag{1.13}$$

Коефіцієнти $a_0, a_1, \dots, a_p$ знаходяться методом найменших квадратів, мінімізуючи:

$$\min_{\mathbf{a}} \sum_{i=1}^{N} \left(x_i - \sum_{j=0}^{p} a_j \, t_i^j \right)^2 . \tag{1.14}$$

Поліноміальна регресія краще моделює криволінійний рух (наприклад, параболічну траєкторію під дією гравітації), проте чутлива до шуму та може давати нестабільні екстраполяції при високих ступенях полінома.

Окремим важливим напрямком є прогнозування зіткнень об'єктів із перешкодами середовища. Задача формулюється як пошук перетину екстрапольованої траєкторії (промінь) з відомими межами простору (стіни). Геометричну модель прогнозування зіткнення наведено на рис. 1.3.

Нехай об'єкт рухається з позиції $P_0 = (x_0, y_0)$ зі швидкістю $\mathbf{v} = (v_x, v_y)$. Траєкторія описується параметричним рівнянням:

$$P(t) = P_0 + \mathbf{v} \cdot t = (x_0 + v_x t, ; y_0 + v_y t). \tag{1.15}$$

Стіна описується рівнянням прямої:

$$Ax + By + C = 0. \tag{1.16}$$

Підставляючи параметричне рівняння траєкторії в рівняння прямої:

$$A(x_0 + v_x t) + B(y_0 + v_y t) + C = 0. \tag{1.17}$$

Розв'язуючи відносно t:

$$t = -\frac{Ax_0 + By_0 + C}{Av_x + Bv_y}. \tag{1.18}$$

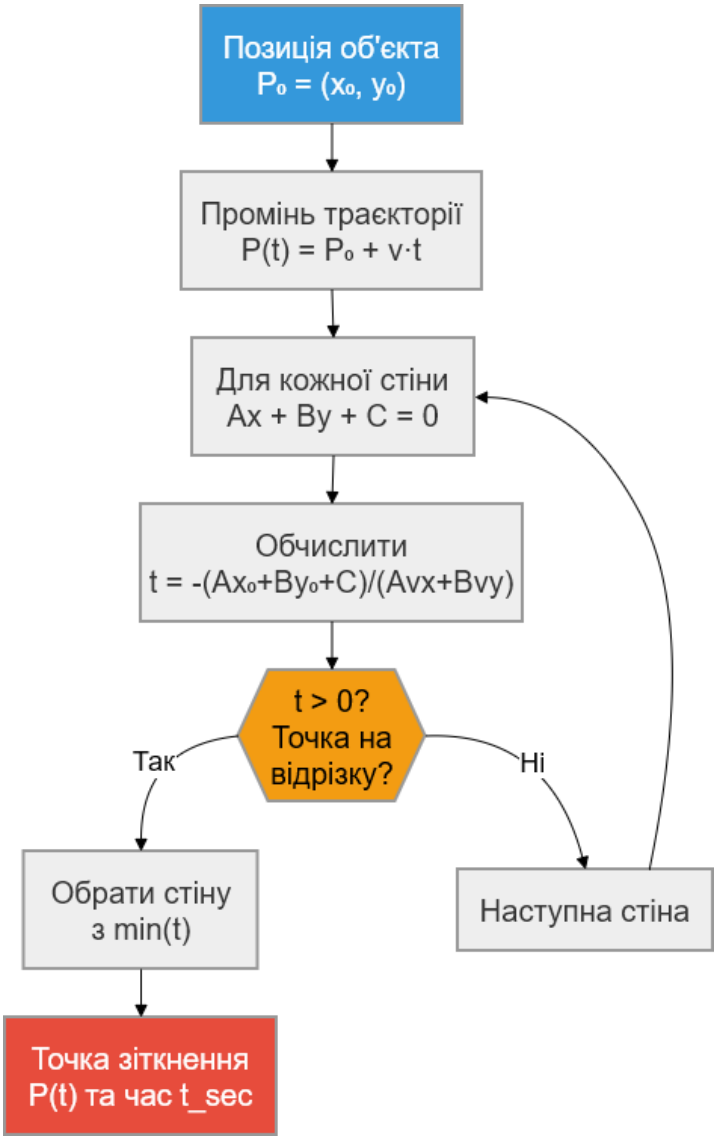


Рисунок 1.3 – Геометрична модель прогнозування зіткнення: перетин променя траєкторії з прямою стіни

Якщо $t > 0$, зіткнення відбудеться у майбутньому через t кадрів. Якщо знаменник $Av_x + Bv_y = 0$, траєкторія паралельна стіні. Час до зіткнення у секундах:

$$t_{sec} = \frac{t}{FPS_{video}}. \tag{1.19}$$

Для визначення розташування стін у цій роботі використовується непрямий метод: стіни виявляються автоматично за допомогою кластеризації точок відскоків об’єктів (алгоритм DBSCAN). Кожен кластер апроксимується прямою лінією методом найменших квадратів, що дає рівняння $Ax + By + C = 0$.

Для виявлення моментів відскоку (різкої зміни напрямку руху) використовується аналіз кута між послідовними векторами руху. Нехай задано три послідовні точки траєкторії:P_1, P_2, P_3. Визначаються два вектори:

$$\mathbf{u} = P_2 - P_1, \quad \mathbf{w} = P_3 - P_2. \tag{1.20}$$

Косинус кута між ними:

$$\cos\theta = \frac{\mathbf{u} \cdot \mathbf{w}}{|\mathbf{u}| \cdot |\mathbf{w}|} = \frac{u_x w_x + u_y w_y}{\sqrt{u_x^2 + u_y^2} \cdot \sqrt{w_x^2 + w_y^2}}. \tag{1.21}$$

Якщо $\cos\theta < \tau$ (у системі $\tau = 0.5$, що відповідає куту $\theta > 60°$), фіксується подія відскоку. Додатково перевіряється мінімальна швидкість ($|\mathbf{u}| > 5$ пікселів/кадр), щоб відфільтрувати тремтіння нерухомих об’єктів.

1.5 Постановка задачі дослідження

На основі проведеного аналізу предметної області та існуючих рішень можна сформулювати задачу дослідження.

Мета роботи – розробка інтелектуальної системи аналізу та передбачення руху об’єктів у динамічному середовищі на основі нейромережевої моделі YOLOv12.

Об’єкт дослідження – процес аналізу та передбачення руху об’єктів у динамічному середовищі з використанням методів глибокого навчання та комп’ютерного зору.

Предмет дослідження – методи та алгоритми детекції, відстеження та прогнозування траєкторій рухомих об’єктів із компенсацією руху камери.

Для досягнення поставленої мети необхідно розв’язати такі задачі:

- провести аналіз існуючих методів детекції, трекінгу та прогнозування руху об’єктів, обґрунтувати вибір базових алгоритмів;
- спроєктувати модульну конвеєрну архітектуру системи, що охоплює повний цикл обробки: від захоплення відеокадру до візуалізації прогнозу;
- зібрати тренувальні дані безпосередньо з тестової 3D-сцени в Unreal Engine 5 та виконати навчання алгоритму YOLOv12n. Очікуваним результатом цього етапу є досягнення метрики mAP@50 більше 95%;
- розробити механізм стабілізації відеоряду. Зміщення камери компенсуватиметься за рахунок математичного обчислення гомографії та оптичного потоку. Це критично важливо для збереження правильної геометрії траєкторій;
- реалізувати підсистему аналізу траєкторій: моніторинг швидкості та локалізація точок удару. Згрупувавши координати таких зіткнень, алгоритм зможе «наосліп» виявляти фізичні межі простору;
- реалізувати підсистему прогнозування: екстраполяція траєкторій та передбачення зіткнень зі стінами;
- перевірити працездатність алгоритму за допомогою тестового відеоряду. Після прогону системи необхідно зібрати статистику для підтвердження якості детекції. Окремо досліджуватимуться надійність утримання треків та достовірність передбачення майбутніх зіткнень.

На вхід алгоритм отримує відеоряд із параметрами 2560x1440 пікселів та частотою 60 кадрів за секунду. Сама сцена являє собою замкнений простір, де паралельно переміщується до 16 сферичних об’єктів. Після завершення обробки

система повертає графічно розмічений відеопотік. На нього накладаються обмежувальні рамки з ідентифікаторами, вектори напрямку, а також підсвічуються геометрія стін, точки майбутніх відскоків та ймовірні місця зіткнень. Окрім візуальної складової, алгоритм паралельно фіксує метрики продуктивності (швидкість обробки у FPS) та точності (mAP для детекції, RMSE для екстраполяції траєкторій). У процесі моделювання було застосовано низку спрощень. Вважається, що всі цілі є абсолютно жорсткими тілами, які не деформуються, а їхні удари об перешкоди мають характер пружних зіткнень (без втрати енергії). Алгоритм обмежений горизонтом прогнозування в 1 секунду. При цьому технічною вимогою є стабільна підтримка швидкодії на рівні не менше 15 FPS під час роботи з цільовою роздільною здатністю на потужностях відеокарти NVIDIA RTX 3080.

2 ПРОЄКТУВАННЯ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ

2.1 Вимоги до системи (функціональні та нефункціональні)

На основі постановки задачі (п. 1.5) сформульовано функціональні та нефункціональні вимоги до інтелектуальної системи аналізу та передбачення руху об’єктів.

Функціональні вимоги визначають, що саме система повинна робити:

FR-1 Детекція об’єктів. Програмний комплекс має покадрово локалізувати всі цільові об’єкти (кулі) у відеопотоці та формувати координати їхніх обмежувальних рамок (bounding boxes) у форматі (c_x, c_y, w, h).

FR-2 Трекінг об’єктів. Кожному виявленому об’єкту повинен присвоюватись унікальний ідентифікатор (track ID), що зберігається між кадрами протягом усього часу перебування об’єкта в полі зору камери.

FR-3 Обчислення швидкості. Для кожного об’єкта з достатньою історією спостережень система повинна обчислювати вектор швидкості $\mathbf{v} = (v_x, v_y)$ у пікселях/кадр.

FR-4 Детекція відскоків. Система повинна виявляти моменти різкої зміни напрямку руху об’єктів (відскоки від стін) та фіксувати координати точок відскоків.

FR-5 Виявлення стін. На основі накопичених точок відскоків система повинна автоматично виявляти межі простору (стіни) та обчислювати їхні рівняння у вигляді $Ax + By + C = 0$.

FR-6 Прогнозування траєкторій. Система повинна екстраполювати траєкторію кожного об’єкта на $\Delta t = 1$ секунду вперед і візуалізувати прогнозоване положення.

FR-7 Прогнозування зіткнень. Система повинна визначати найближчу стіну, з якою зіткнеться об’єкт, обчислювати точку та час зіткнення.

FR-8 Компенсація руху камери. Для точного аналізу траєкторій координати об’єктів повинні конвертуватись у стабільну систему координат для коректного аналізу траєкторій.

FR-9 Візуалізація. Має забезпечувати графічне відображення розрахункових даних на відео. До обов’язкових елементів належать: bounding boxes, ідентифікатори треків (track ID), вектори прогнозованого руху, координати відскоків, контури площин (стін) та можливі точки зіткнення.

FR-10 Запис результатів. Анотований відеопотік повинен зберігатися у файл формату MP4.

Нефункціональні вимоги

Нефункціональні вимоги визначають якісні характеристики системи:

NFR-1 Вимоги до якості розпізнавання. Мінімально допустиме значення mAP@50 становить 95%.

NFR-2 Продуктивність. Швидкість обробки повинна забезпечувати не менше 10 FPS на GPU NVIDIA RTX 3080 при роздільній здатності вхідного відео 2560×1440.

NFR-3 Модульність. Система повинна мати модульну архітектуру з чітким розділенням відповідальності між компонентами (детекція, трекінг, аналіз, візуалізація).

NFR-4 Конфігурованість. Усі параметри системи (пороги, розміри буферів, кольори візуалізації) повинні бути винесені в єдиний конфігураційний файл.

NFR-5 Масштабованість. Система повинна коректно працювати з кількістю об’єктів від 1 до 16 одночасно.

2.2 Розробка загальної конвеєрної архітектури системи

Архітектура системи базується на конвеєрній моделі (pipeline). Її суть полягає в послідовній передачі даних: вихідні параметри одного модуля одразу стають вхідними для наступного. Завдяки цьому ми отримали гнучку систему, яку зручно тестувати, а будь-який блок можна оновити чи замінити, не ламаючи загальну логіку програми (рис. 2.1).

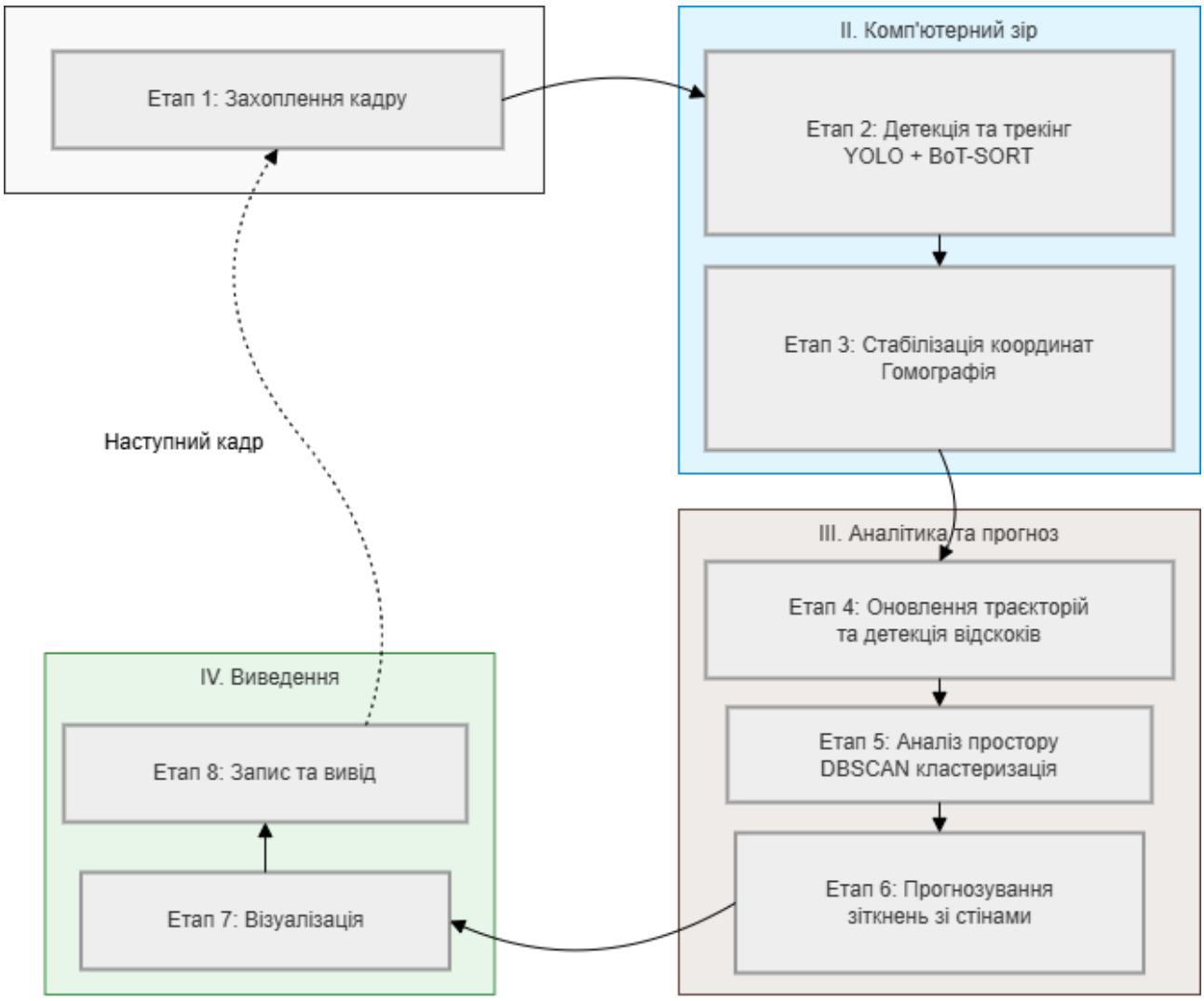


Рисунок 2.1 – Конвеєрна архітектура обробки одного кадру відеопотоку

Конвеєр обробки одного кадру складається з восьми послідовних етапів:

Етап 1 Захоплення кадру. Модуль video_io відкриває відеофайл через cv2.VideoCapture, зчитує метадані (роздільна здатність, FPS) та покадрово видає фрейми.

Етап 2 Детекція та трекінг. Роботу цього етапу забезпечує модуль detector, що реалізує розпізнавання через YOLO та супровід за допомогою BoT-SORT. Результатом його виконання є анотований кадр і набір даних DetectedObject для кожного виявленого елемента.

Етап 3 Стабілізація координат. Модуль stabilizer виконує розрахунок гомографії відносно першого кадру, проєктуючи позиції цілей у статичну

систему координат. Оригінальні ж екранні координати залишаються в пам’яті виключно для фінального рендерингу розмітки.

Етап 4 Оновлення траєкторій. Модуль tracker накопичує історію позицій кожного об’єкта (буфер довжиною 10 кадрів), обчислює вектор швидкості, виявляє відскоки та відрізняє відскоки від стін від зіткнень між об’єктами.

Етап 5 Аналіз простору. Модуль analyzer кластеризує накопичені точки відскоків алгоритмом DBSCAN та апроксимує кожен кластер прямою лінією (стіною).

Етап 6 Прогнозування зіткнень. Завдання компонента predictor полягає в аналізі напрямку руху. Програма екстраполює шлях усіх об’єктів, щоб заздалегідь обчислити місця їхнього перетину з площинами стін.

Етап 7 Візуалізація. Модуль visualizer відрисовує всі елементи: стрілки передбачення, точки відскоків, лінії стін, хрестики прогнозованих зіткнень, HUD.

Етап 8 Запис. Анотований кадр записується у вихідний MP4-файл та відображається на екрані.

Система реалізована як набір з 8 Python-модулів, кожен з яких відповідає за один аспект функціональності (табл. 2.1 та рис 2.2).

Таблиця 2.1 – Модульна структура системи

Модуль	Відповідальність	Ключові функції
config.py	Конфігурація: пороги, шляхи, кольори	Константи, типи
video_io.py	Захоплення та запис відео	open_video(), create_writer()
detector.py	Детекція та трекінг YOLO	DetectedObject
stabilizer.py	Компенсація руху камери	CameraStabilizer
tracker.py	Історія траєкторій, швидкість, відскоки	TrajectoryTracker
analyzer.py	Кластеризація відскоків, виявлення стін	detect_walls(), Wall
predictor.py	Прогнозування зіткнень зі стінами	predict_collisions()
visualizer.py	Відрисовка на кадрі	draw_*() функції
main.py	Головний конвеєр – оркестрація модулів	main()

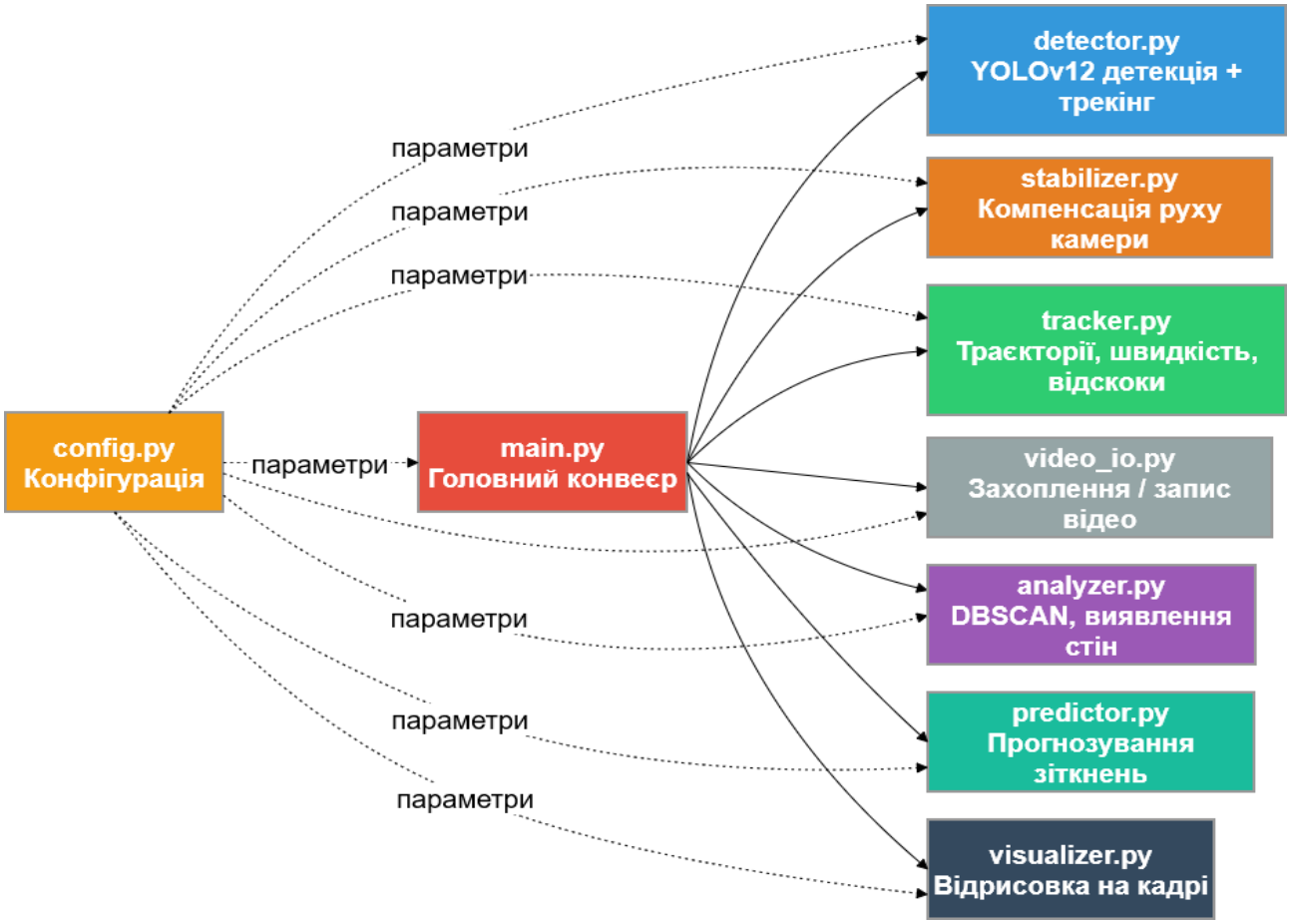


Рисунок 2.2 – Діаграма компонентів системи та їхніх залежностей

2.3 Проєктування підсистем захоплення, детекції та трекінгу

Підсистема захоплення інкапсулює роботу з OpenCV VideoCapture. Проєктне рішення – обгортка, що повертає іменований кортеж VideoInfo з метаданими:

$$VideoInfo = (fps, width, height). \quad (2.1)$$

Параметри вхідного відео у тестовому середовищі: роздільна здатність 2560 x 1440 пікселів, частота кадрів 60 FPS, кодек H.264. Запис результату здійснюється у формат MP4 з кодеком mp4v.

Для виявлення об’єктів у роботі задіяна нейромережа YOLOv12n, яка є частиною пакету Ultralytics. Обробка відеоряду відбувається через нашу кастомну функцію detect_and_track(frame). Вона повинна виконати і детекцію, і супровід цілей одночасно, використовуючи базовий метод model.track(). Важливим етапом стала конфігурація цього виклику. Без параметра persist=True система просто не змогла б зв’язувати об’єкти на сусідніх кадрах. Додатково ми відключили консольні логи (verbose=False) та встановили бар’єр для хибних спрацьовувань, зафіксувавши параметр conf на рівні 0.5.

Усі знайдені на кадрі цілі пакуються у спеціальну структуру DetectedObject. Вона зберігає унікальний номер треку (track_id), координати центру обмежувальної рамки (cx, cy), її габарити (width, height) та обчислений радіус. Використання формату рамки XYWH (деструктуризація через центр і розміри) замість класичного кутового XYXY є свідомим рішенням: такий підхід значно спрощує подальший розрахунок відстаней та кінематичних характеристик об’єктів.

Трекінг здійснюється вбудованим трекером BoT-SORT бібліотеки Ultralytics. Проте для задач аналізу траєкторій вбудованого трекера недостатньо – він лише присвоює ID. Тому в системі реалізовано додатковий модуль TrajectoryTracker, що розширює функціональність:

– буфер історії. Для кожного track_id зберігається кільцевий буфер (deque) довжиною $N = 10$ кадрів. Кожен запис містить координати та номер кадру: $(x, y, frame_i dx)$;

– обчислення швидкості. Обчислення вектора швидкості базується на різниці між першою та останньою точками історії переміщень:

$$v_x = \frac{x_{last} - x_{first}}{frame_{last} - frame_{first}}, \quad v_y = \frac{y_{last} - y_{first}}{frame_{last} - frame_{first}}; \qquad (2.2)$$

– детекція відскоків. Відскок визначається за трьома точками: першою, середньою та останньою в буфері. Якщо кут між векторами «до» та

«після» середньої точки перевищує $60°$ ($\cos\theta < 0.5$) – фіксується відскок;

- фільтрація зіткнень. Для кожного відскоку перевіряється наявність іншого об’єкта поблизу у вікні приблизно 2 кадри. Якщо відстань між об’єктами менша за $3 \cdot radius$, відскок класифікується як ball-to-ball зіткнення (не враховується для побудови стін);
- кулдаун. Після кожного відскоку вводиться пауза 15 кадрів для уникнення повторної фіксації одного й того ж відскоку.

2.4 Проєктування алгоритмів аналізу траєкторій та прогнозування

Алгоритм виявлення стін (DBSCAN). Стіни виявляються непрямим методом через кластеризацію точок, де об’єкти змінили напрямок руху. Алгоритм DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [19] обрано з таких причин:

- не потребує заздалегідь заданої кількості кластерів (на відміну від K-Means);
- автоматично відокремлює шум (випадкові відскоки) від реальних кластерів;
- ефективний для кластерів довільної форми.

Параметри DBSCAN:

- $\varepsilon = 50$ пікселів – максимальна відстань між сусідніми точками одного кластера;
- $minPts = 3$ – мінімальна кількість точок для формування кластера.

Кластеризація запускається лише при накопиченні не менше 4 точок відскоків. Кожен знайдений кластер апроксимується прямою лінією за допомогою функції cv2.fitLine(), що реалізує метод найменших квадратів з мінімізацією відстані L_2. Результат – вектор напрямку (v_x, v_y) та опорна точка (x_0, y_0), з яких обчислюється рівняння прямої:

$$A = -v_y, \quad B = v_x, \quad C = v_y \cdot x_0 - v_x \cdot y_0. \tag{2.3}$$

Довжина лінії обмежується проекцією крайніх точок кластера на напрямний вектор, щоб стіна відображалася лише в тій ділянці, де зафіксовано відскоки.

Алгоритм прогнозування зіткнень

Для кожного об’єкта з ненульовою швидкістю будується промінь (ray) від поточної позиції вздовж вектора швидкості:

$$\mathbf{P}(t) = \big(x_0 + v_x \cdot t, ; y_0 + v_y \cdot t\big), \quad t > 0. \tag{2.4}$$

Для кожної виявленої стіни обчислюється параметр перетину:

$$t = -\frac{A \cdot x_0 + B \cdot y_0 + C}{A \cdot v_x + B \cdot v_y}. \tag{2.5}$$

Додатково перевіряється, чи точка перетину лежить на відрізку стіни (а не на нескінченному продовженні прямої). Для цього обчислюється проекція точки на відрізок $[P_1, P_2]$:

$$t_{proj} = \frac{(p_x - x_1)(x_2 - x_1) + \big(p_y - y_1\big)(y_2 - y_1)}{|P_2 - P_1|^2}. \tag{2.6}$$

Якщо $-\frac{r}{|P_2 - P_1|} \le t_{proj} \le 1 + \frac{r}{|P_2 - P_1|}$ (з допуском на радіус об’єкта r), точка вважається такою, що лежить на відрізку. Серед усіх стін обирається та, для якої t мінімальне (найближче зіткнення).

2.5 Проєктування методу компенсації руху камери

Постановка проблеми При русі камери (панорамування, обертання) координати всіх точок кадру зміщуються. Нехай об’єкт нерухомий у тривимірному просторі, але камера повертається. У кадровій системі координат цей об’єкт буде мати ненульову швидкість, що призведе до хибних результатів аналізу траєкторій, помилкових детекцій відскоків та некоректного прогнозування.

Проблема зміщення об’єктива вирішується через розрахунок параметрів трансформації для кожної ітерації. Застосування такої поправки дозволяє відв’язати координати цілей від поточного кадру. Замість цього вони перераховуються у стабільну глобальну систему, яка базується виключно на стартовому зображенні.

Вибір методу: гомографія – це проективне перетворення, що описує відповідність між двома зображеннями однієї площини. Матриця гомографії $H \in \mathbb{R}^{3\times3}$ визначає перетворення точки (x, y) з одного кадру в інший:

$$(x'\ y'\ 1) \sim H \cdot (x\ y\ 1) = (h_{11}\quad h_{12}\quad h_{13}\ h_{21}\quad h_{22}\quad h_{23}\ h_{31}\quad h_{32}\quad h_{33}) \cdot (x\ y\ 1). \qquad (2.7)$$

Спроєктований алгоритм стабілізації працює за таким принципом:

Крок 1 Ізоляція рухомих об’єктів. Щоб уникнути похибок під час розрахунку гомографії, ключові точки шукаються виключно на статичному фоні. Для цього формується бінарна маска, у якій області навколо виявлених bounding boxes (з відступом у 20 пікселів) заливаються нулями.

Крок 2 Пошук ключових точок фону. На попередньому кадрі, попередньо перетвореному у відтінки сірого, алгоритм Shi-Tomasi (goodFeaturesToTrack) виділяє кутові точки. Ліміт встановлено на рівні 200 точок із мінімальним порогом якості 0.01 та обов’язковою дистанцією між ними не менше 30 пікселів.

Крок 3 Оптичний потік. Відстеження знайдених точок на поточному кадрі здійснюється за допомогою пірамідального методу Lucas-Kanade. Використовується вікно пошуку розміром 21×21 px та три рівні піраміди

Крок 4 Обчислення гомографії. За відповідними парами точок обчислюється матриця гомографії H_{frame} методом RANSAC з порогом репроєкції 5 px. Мінімальна кількість точок для обчислення: 10.

Крок 5 Фільтрація шуму. Перевіряється, чи гомографія відображає реальний рух камери. Якщо трансляційна компонента $\max(|h_{13}|, |h_{23}|) < 0.5$ px та обертальна компонента < 0.001, гомографія ігнорується.

Крок 6 Накопичення. Гомографії між сусідніми кадрами перемножуються для отримання кумулятивної матриці від поточного кадру до першого (референсного):

$$H_{cumul}^{(k)} = H_{cumul}^{(k-1)} \cdot H_{frame}^{(k)}. \tag{2.8}$$

Крок 7 Перетворення координат. Координати центрів об’єктів перетворюються функцією cv2.perspectiveTransform():

$$(x_{stab}\ y_{stab}) = H_{cumul} \cdot (x_{frame}\ y_{frame}\ 1). \tag{2.9}$$

Отримані стабілізовані координати (x_{stab}, y_{stab}) передаються до модуля трекінгу для оцінки швидкості та фіксації відскоків. При цьому саме зображення об’єктів виводиться на екран в оригінальних координатах. Для коректної візуалізації аналітичної графіки (точки відскоків, лінії стін, колізії) в умовах рухомої камери застосовується зворотне перетворення (рис. 2.3):

$$\mathbf{p}screen = Hcumul^{-1} \cdot \mathbf{p}_{world}. \tag{2.10}$$

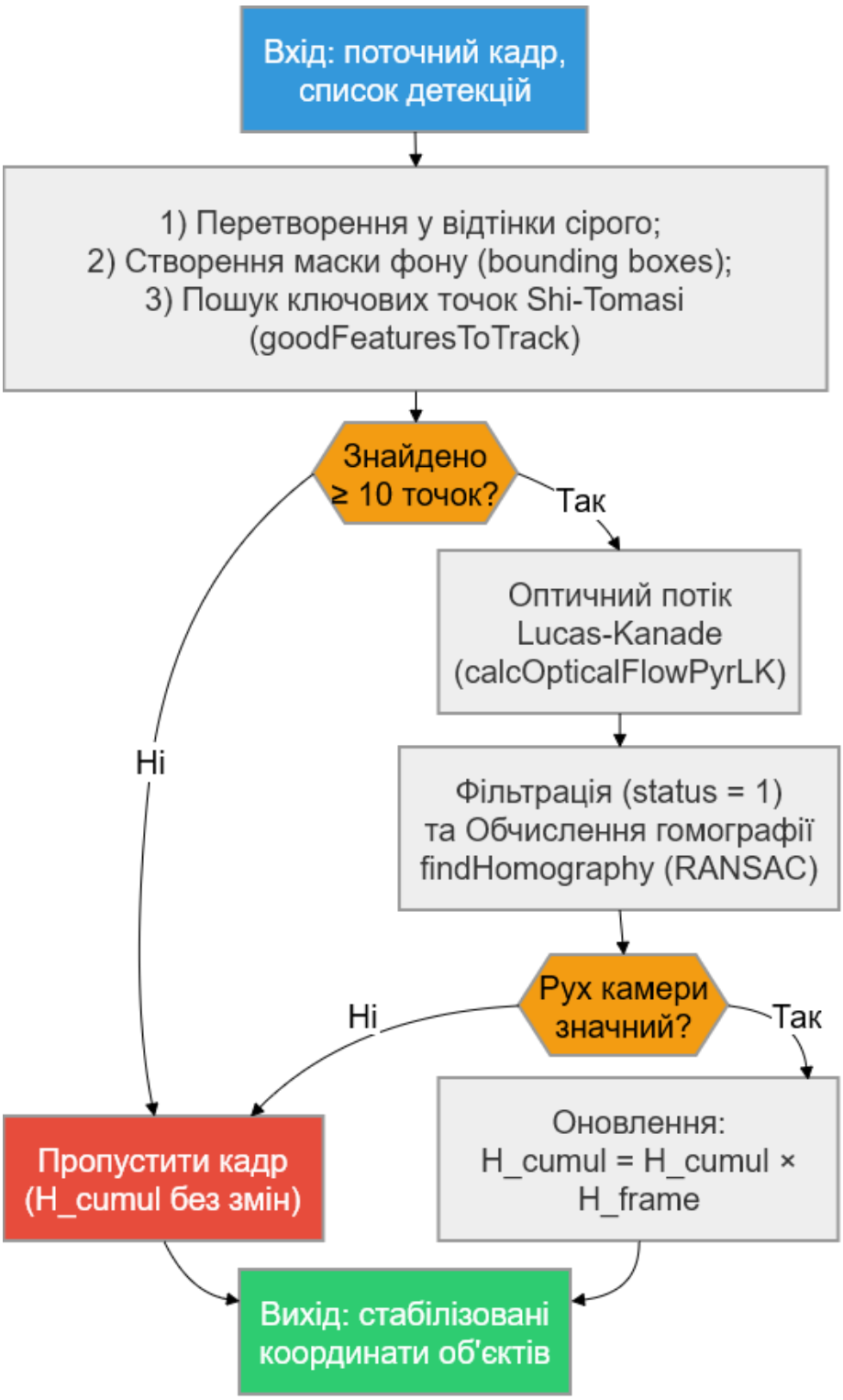


Рисунок 2.3 – Блок-схема алгоритму компенсації руху камери на основі гомографії

3 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ

3.1 Вибір стеку технологій та середовища розробки

Програмною основою розробленої системи виступає мова Python версії 3.12. Її вибір зумовлений глибокою інтеграцією з сучасними фреймворками комп'ютерного зору. Для забезпечення надійності архітектури ми активно використовували вбудований модуль typing, який додає строгу типізацію та спрощує виловлювання помилок ще на етапі написання скриптів. Роль головного інструменту для роботи з відеоданими взяла на себе бібліотека OpenCV. Через неї реалізовано базові операції введення/виведення потоку, переведення кадрів у градації сірого та рендеринг графіки. Окрім цього, OpenCV закриває всі математичні потреби стабілізації: функції goodFeaturesToTrack та calcOpticalFlowPyrLK відповідають за екстракцію ознак і розрахунок потоку Лукаса-Канаде. Своєю чергою, просторові трансформації обчислюються методами findHomography та perspectiveTransform, а для геометричної апроксимації прямих задіяна fitLine

Бібліотека детекції Ultralytics YOLO [20, 21] надає Python API для навчання, інференсу та трекінгу моделей сімейства YOLO. Ключова перевага – інтегрований трекер BoT-SORT, що дозволяє виконувати детекцію та трекінг одним викликом model.track(). Бібліотека чисельних обчислень – NumPy [25]. NumPy використовується для роботи з масивами координат, матрицями гомографії та векторними обчисленнями. Бібліотека машинного навчання – scikit-learn [26]. З scikit-learn використовується лише алгоритм кластеризації DBSCAN для групування точок відскоків. Фреймворк глибокого навчання – PyTorch 2.6.0 + cu124 є бекендом для Ultralytics YOLO. Забезпечує GPU-прискорення через CUDA.

Апаратне забезпечення:

– GPU: NVIDIA GeForce RTX 3080 (12 ГБ VRAM, 8704 CUDA-ядра); Використовується для інференсу YOLO та навчання моделі;
– CUDA Toolkit [27] забезпечує взаємодію PyTorch з GPU;
– CPU: Intel Core i9 10900. Використовується для OpenCV процессингу.

Середовище розробки:

– IDE: PyCharm Professional;
– ОС: Windows 11;
– менеджер пакетів: pip + venv.

Тестове середовище – Unreal Engine 5 [28]. Тривимірна сцена для генерації тестових відеозаписів змодельована в ігровому рушії UE5. Сцена являє собою замкнену прямокутну коробку зі стінами, всередині якої вільно рухаються 16 куль, відбиваючись від стін (табл. 3.1).

Таблиця 3.1 – Стек технологій

Компонент	Технологія	Версія	Призначення
Мова	Python	3.12	Основна мова розробки
Комп’ютерний зір	OpenCV	4.2	Обробка відео та зображень
Детекція	Ultralytics YOLO	8.12	YOLOv12 та трекінг
Deep Learning	PyTorch	2.2	Бекенд для YOLO
Чисельні обч.	NumPy	1.2	Масиви, матриці
Кластеризація	scikit-learn	1.1	DBSCAN
GPU	NVIDIA 3080	—	Прискорення інференсу
CUDA	CUDA Toolkit	12.4	GPU-обчислення
Тестове середовище	Unreal Engine	5.3	Генерація тестових відео
Анотація	CVAT	—	Розмітка датасету

3.2 Підготовка датасету та навчання моделі YOLOv12

Датасет підготовлено з відеозаписів тестової сцени Unreal Engine 5. Процес збору складався з таких кроків:

- запис відео. У UE5 створено сцену – замкнену прямокутну коробку, всередині якої рухаються 16 куль. Камера здійснює повільне обертання навколо сцени, забезпечуючи різноманітні ракурси. Записано кілька відеороликів загальною тривалістю 5 хвилин при 60 FPS і роздільній здатності 2560×1440;
- вилучення кадрів. За допомогою скрипта extract_frames.py з відео вилучено кожен 60-й кадр (1 кадр/секунду), щоб уникнути надмірно схожих зображень. Загалом отримано набір зображень для розмітки;
- розподіл на вибірки. Датасет розділено на навчальну (train) та валідаційну (val) вибірки у співвідношенні приблизно 80:20:
  - навчальна вибірка: 70 зображень;
  - валідаційна вибірка: 16 зображень.

Розмітка виконана за допомогою платформи CVAT (Computer Vision Annotation Tool) [24] – відкритого інструменту для анотації зображень. Для кожного зображення розмічено bounding boxes усіх видимих куль (рис. 3.1).

Конфігурація датасету (data.yaml):

- кількість класів: $nc = 1$;
- назви класів: [‘ball’];
- формат анотацій: YOLO (текстовий файл з рядками class cx cy w h у нормалізованих координатах [0, 1]).

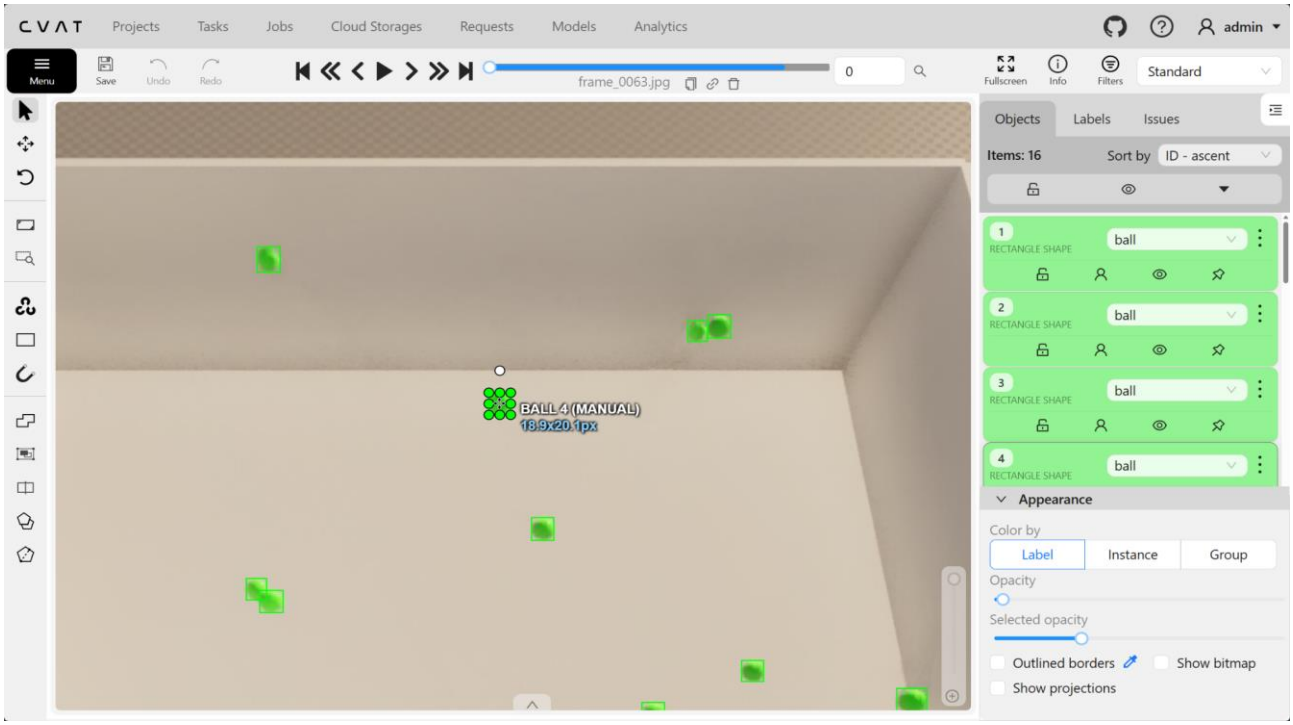


Рисунок 3.1 – Інтерфейс CVAT з розміченими bounding boxes об’єктів на тестовій сцені

Навчання виконано за допомогою скрипта train_model.py з параметрами які наведені у таблиці. 3.2.

Таблиця 3.2 – Параметри навчання YOLOv12n

Параметр	Значення	Пояснення
Базова модель	yolo12n.pt	Попередньо навчена на COCO
Кількість епох	300	Збіжність на малому датасеті
Розмір зображення	640×640	Стандартний для YOLO
Batch size	16	Автоматично підібрано
Пристрій	GPU 0 (RTX 3080)	CUDA-прискорення
Оптимізатор	auto (SGD)	З momentum = 0.937
Learning rate	0.01 - 0.0001	Cosine annealing
Weight decay	0.0005	Регуляризація L2

Аугментації:

– Mosaic ($p = 1.0$) – комбінування 4 зображень у одне, збільшує контекстне різноманіття;
– MixUp ($p = 0.1$) – лінійна комбінація двох зображень;
– Random Erasing ($p = 0.4$) – випадкове затирання ділянок зображення, імітує часткове перекриття;
– горизонтальне віддзеркалення ($p = 0.5$);
– HSV-зсув – випадкова зміна кольору (hue ±0.015, saturation ±0.7, value ±0.4).

Результати навчання:

Навчання тривало 300 епох (~9.4 хвилин). Динаміка метрик за епохами (рис. 3.2, табл. 3.3).

Таблиця 3.3 – Метрики якості на валідаційній вибірці

Метрика	Епоха 50	Епоха 100	Епоха 200	Епоха 300 (фінал)
Precision	0.939	0.981	0.980	0.996
Recall	0.957	0.965	0.982	0.989
mAP@50	0.965	0.981	0.994	0.995
mAP@50-95	0.471	0.636	0.672	0.696
train/box_loss	2.343	1.892	1.815	1.404
train/cls_loss	0.746	0.533	0.470	0.405

Фінальна модель досягла:

– Precision = 0.996 – 99.6% передбачень є вірними;
– Recall = 0.989 – 98.9% об’єктів виявлено;
– mAP@50 = 0.995 – практично ідеальна детекція при IoU ≥ 0.5;
– mAP@50-95 = 0.696 – якість локалізації при суворих порогах IoU.

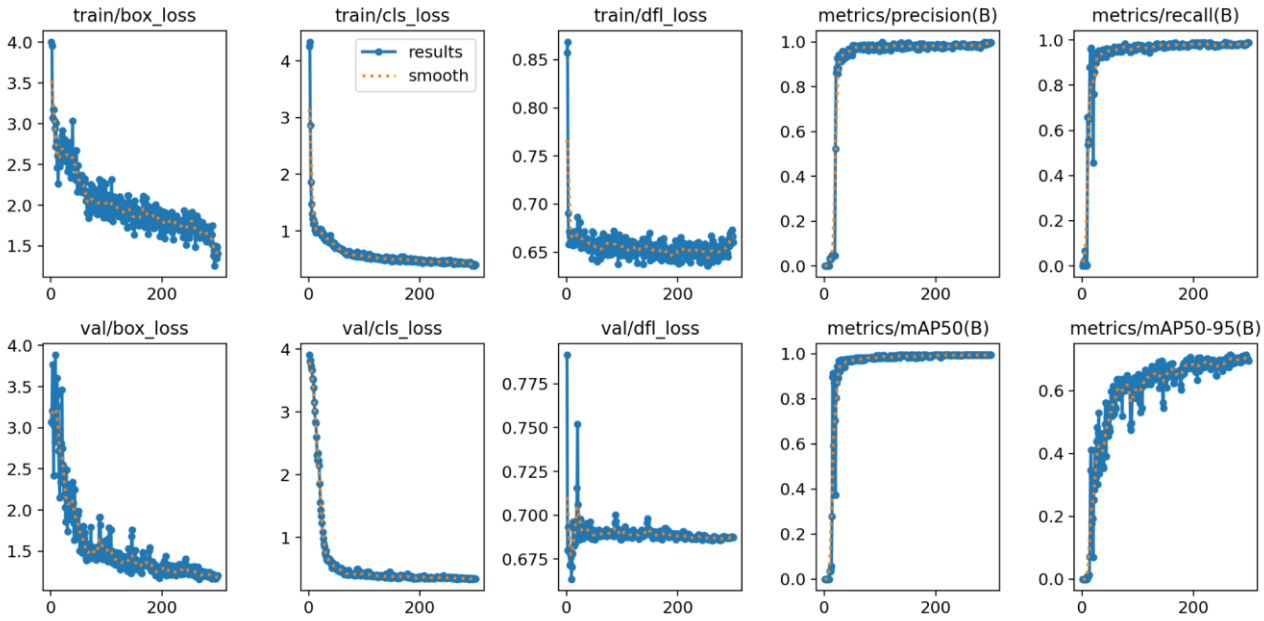


Рисунок 3.2 – Динаміка метрик якості та функцій втрат під час навчання YOLOv12n протягом 300 епох

Висока точність при невеликому наборі даних (86 зображень) пояснюється:

– ефективним transfer learning з моделі, навченої на COCO;
– відносною простотою задачі (1 клас, однорідний фон, чіткі контури об’єктів);
– агресивними аугментаціями (mosaic, erasing), що штучно збільшують різноманітність.

3.3 Реалізація конвеєра детекції та відстеження об’єктів

Модуль detector.py реалізує клас Detector – обгортку над Ultralytics YOLO. Основний метод detect_and_track() приймає кадр у форматі BGR (numpy array) та повертає анотований кадр і список детекцій (рис. 3.3).

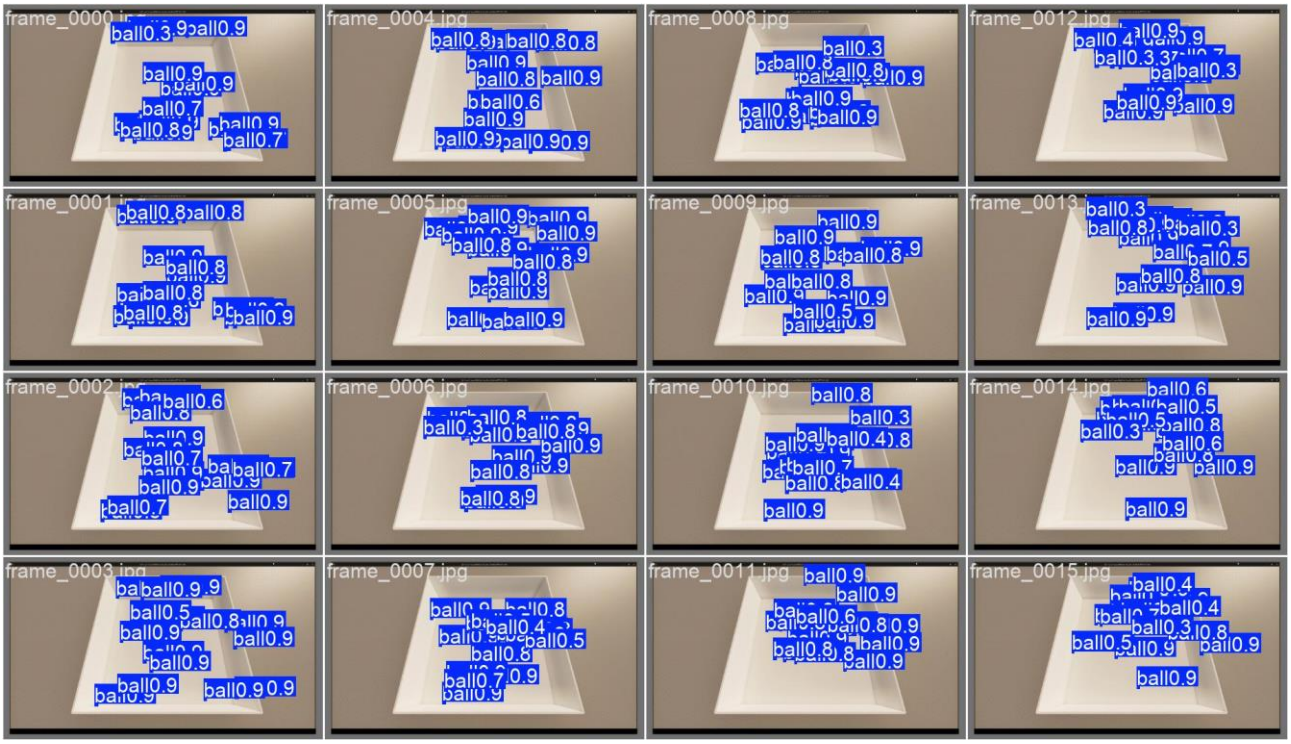


Рисунок 3.3 – Результат детекції та трекінгу

Послідовність обробки одного кадру:

а) інференс. Кадр подається в model.track() з порогом впевненості 0.5. Модель YOLOv12n обробляє зображення за один прохід, повертаючи результат у форматі Ultralytics Results;

б) анотація. Метод results[0].plot() відрисовує bounding boxes на кадрі. Параметри conf=False, labels=False вимикають відображення впевненості та класів (оскільки клас один);

в) вилучення даних. З об’єкта boxes отримуються:

1) boxes.xywh – координати у форматі (центр X, центр Y, ширина, висота);

2) boxes.id – ідентифікатори трекінгу.

г) створення об’єктів. Для кожної детекції створюється dataclass DetectedObject з обчисленим радіусом $r = \frac{\max(w,h)}{2}$.

Координацію всіх підсистем забезпечує головний модуль (main.py). В основі його роботи лежить безперервний цикл, який відповідає за покадрове зчитування відеопотоку, послідовну передачу даних до аналітичних блоків та

збереження фінального результату. Для оцінки продуктивності системи розрахунок кадрової частоти (FPS) виконується за такою формулою:

$$FPS_{process} = \frac{1}{\Delta t_{frame}}, \tag{3.1}$$

де Δt_{frame} – час обробки одного кадру у секундах.

3.4 Реалізація стабілізації координат (компенсація руху камери)

Модуль stabilizer.py реалізує клас CameraStabilizer, що інкапсулює весь алгоритм компенсації руху камери, спроєктований у п. 2.5.

При створенні екземпляра ініціалізуються:

- кумулятивна матриця гомографії $H_{cumul} = I_{3\times3}$ (одинична матриця);
- параметри Shi-Tomasi та Lucas-Kanade (з конфігурації).

Метод update(frame, detections) виконує повний цикл обчислення гомографії для одного кадру:

- перетворення кадру у відтінки сірого;
- створення маски фону: для кожної детекції область $[cx - w/2 - pad, ; cy - h/2 - pad] \times [cx + w/2 + pad, ; cy + h/2 + pad]$ заповнюється нулями (де pad = 20px);
- пошук ключових точок на попередньому кадрі з маскою;
- відстеження точок оптичним потоком;
- обчислення гомографії методом RANSAC;
- перевірка значимості руху (фільтрація мікрошуму);
- оновлення кумулятивної матриці: $H_{cumul} \leftarrow H_{cumul} \cdot H_{frame}$.

Ключовий фрагмент обчислення гомографії між двома кадрами наведено нижче на рисунку 3.4:

```
def _compute_frame_homography(self, prev_gray, curr_gray, mask):
    # Ключові точки на фоні (з маскою, що виключає об’єкти)
    prev_pts = cv2.goodFeaturesToTrack(
        prev_gray, mask=mask, **self._feature_params
    )
    if prev_pts is None or len(prev_pts) < 10:
        return None

    # Оптичний потік Lucas-Kanade
    curr_pts, status, _ = cv2.calcOpticalFlowPyrLK(
        prev_gray, curr_gray, prev_pts, None, **self._lk_params
    )

    # Фільтрація – лише успішно відстежені точки
    good_prev = prev_pts[status.ravel() == 1]
    good_curr = curr_pts[status.ravel() == 1]

    # Гомографія методом RANSAC
    H, inliers = cv2.findHomography(
        good_prev, good_curr, cv2.RANSAC, ransacReprojThreshold=3.0
    )
    return H
```

Метод stabilize_detections() перетворює координати центрів усіх детекцій з кадрової системи у стабільну:

- збирає координати (cx, cy) у масив форми $[N, 1,2]$;
- застосовує cv2.perspectiveTransform(centers, H);
- створює нові об’єкти DetectedObject зі стабілізованими координатами (розміри bbox залишаються незмінними).

Метод to_screen_space(). Зворотне перетворення: переводить точки зі стабілізованих координат у екранні для відрисовки. Обчислює H^{-1} через np.linalg.inv(H) та застосовує perspectiveTransform.

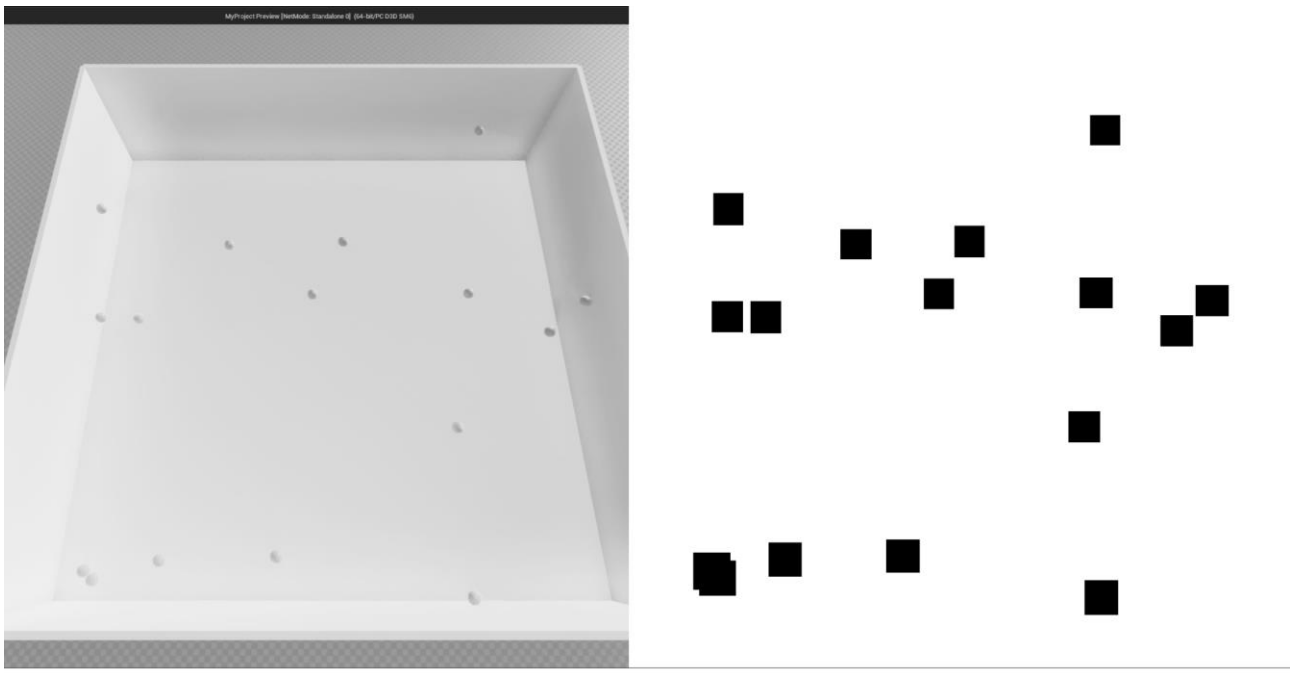


Рисунок 3.4 – Маска фону для обчислення гомографії: (а) оригінальний кадр; (б) двійкова маска з виключеними областями об’єктів

3.5 Реалізація аналізу траєкторій та прогнозування руху

Клас TrajectoryTracker (tracker.py) реалізує п’ять функцій аналізу руху: Накопичення історії. Для кожного track_id використовується collections.deque(maxlen=10), що автоматично витісняє найстаріші записи при переповненні. Кожен запис – кортеж $(x, y, frame_i dx)$.

Обчислення швидкості. Метод _compute_velocity() обчислює середню швидкість за весь буфер:

$$v_x = \frac{x_N - x_1}{\Delta frames}, \quad v_y = \frac{y_N - y_1}{\Delta frames}. \tag{3.2}$$

Детекція відскоків. Метод _detect_bounce() аналізує три точки буфера (першу, середню, останню). Обчислює косинус кута між векторами руху «до» та

«після» середньої точки. Перевіряє мінімальну швидкість $|\mathbf{v}| > 5$ px/кадр та стан кулдауну. Детекція відскоку за кутом між векторами руху (tracker.py, фрагмент):

```
def _detect_bounce(self, history, cooldown):
    p_start = history[0]
    p_mid = history[len(history) // 2]
    p_end = history[-1]

    # Вектори до і після середньої точки
    vx_before = p_mid[0] - p_start[0]
    vy_before = p_mid[1] - p_start[1]
    vx_after = p_end[0] - p_mid[0]
    vy_after = p_end[1] - p_mid[1]

    mag_before = math.hypot(vx_before, vy_before)
    mag_after = math.hypot(vx_after, vy_after)

    if mag_before <= MIN_SPEED_FOR_BOUNCE or mag_after <= MIN_SPEED_FOR_BOUNCE:
        return False, None

    # Косинус кута між двома векторами руху
    dot = vx_before * vx_after + vy_before * vy_after
    cos_angle = dot / (mag_before * mag_after)

    if cos_angle < BOUNCE_ANGLE_THRESHOLD:  # < 0.3
        return True, (int(p_mid[0]), int(p_mid[1]))
    return False, None
```

Фільтрація ball-to-ball зіткнень. Метод _is_ball_collision() перебирає історії всіх інших об’єктів і перевіряє, чи був інший об’єкт ближче ніж $3r$ у вікні ± 2 кадри від моменту відскоку.

Оркестрація. Метод update() для кожної детекції: додає точку в історію, перевіряє відскок, обчислює швидкість, повертає пару (об’єкт, швидкість).

Виявлення стін (analyzer.py)

Функція detect_walls() реалізує кластеризацію:

- перевірка мінімальної кількості точок (≥ 4);
- перетворення списку точок у масив NumPy;
- виклик DBSCAN(eps=50, min_samples=3).fit(X);
- для кожного кластера (мітка $\neq -1$): виклик _fit_wall_line();
- _fit_wall_line() використовує cv2.fitLine() з метрикою L_2, обчислює коефіцієнти A, B, C та обмежує довжину відрізка.

Результат – список об'єктів Wall з атрибутами: координати кінців відрізка, коефіцієнти рівняння, масив точок кластера.

Прогнозування зіткнень (predictor.py)

Функція predict_collisions() для кожного об'єкта з ненульовою швидкістю:

- обчислює горизонт: $t_{max} = FPS \cdot 1.0 = 60$ кадрів;
- для кожної стіни обчислює $t = -\frac{Ax_0 + By_0 + C}{Av_x + Bv_y}$;
- перевіряє: $t > 0$ (зіткнення попереду), $t < t_{max}$ (в межах горизонту);
- перевіряє, чи точка перетину лежить на відрізку стіни (з допуском на радіус);
- обирає стіну з мінімальним t (найближче зіткнення);
- повертає CollisionPrediction з точкою, часом та рівнянням стіни.

3.6 Реалізація псевдо-3D аналізу глибини на основі площі об'єкта

Для оцінки відносної відстані об'єкта від камери використовується зміна площі bounding box. Принцип базується на проективній геометрії: чим ближче об'єкт до камери, тим більшим він виглядає на зображенні.

Площа bounding box пропорційна квадрату відносної відстані:

$$S = w \cdot h \propto \frac{1}{d^2}, \tag{3.3}$$

де d – відстань від об’єкта до камери.

Отже, зміна площі між кадрами дозволяє оцінити напрямок руху вздовж осі Z (до камери / від камери):

$$\Delta d \propto \frac{1}{\sqrt{S_t}} - \frac{1}{\sqrt{S_{t-1}}}. \quad (3.4)$$

У реалізованій системі площа bbox кожного об’єкта ($w \times h$) доступна через поля width та height структури DetectedObject. Радіус $r = \frac{\max(w,h)}{2}$ використовується для:

- масштабування кола передбачення (розмір кола дорівнює радіусу об’єкта);
- визначення допуску при перевірці точки перетину на відрізку стіни;
- фільтрації ball-to-ball зіткнень (поріг $3r$).

Зміна радіусу між кадрами дозволяє якісно оцінити рух об’єкта вздовж осі глибини, що доповнює інформацію про 2D-переміщення в площині кадру.

3.7 Реалізація візуалізації та запису результатів

Модуль visualizer.py відповідає за відрисовку всіх аналітичних елементів на кадрі. Кожна функція приймає кадр (numpy array) та малює на ньому in-place за допомогою примітивів OpenCV (рис. 3.5).

Функція draw_prediction_arrow() відрисовує вектор передбаченого руху:

- від поточного центру об’єкта до прогнозованого положення через 1 секунду;
- обчислення:$\left(x_{future}, y_{future}\right) = \left(c_x + v_x \cdot FPS \cdot \Delta t, ; c_y + v_y \cdot FPS \cdot \Delta t\right)$;
- зелена стрілка (arrowedLine) + жовте коло розміром із об’єкт на кінці.

Функція draw_hud() виводить метрики у лівому верхньому куті:

– Video FPS (частота оригінального відео) – жовтий;
– Process FPS (швидкість обробки) – зелений;
– Frame (номер поточного кадру) – білий.

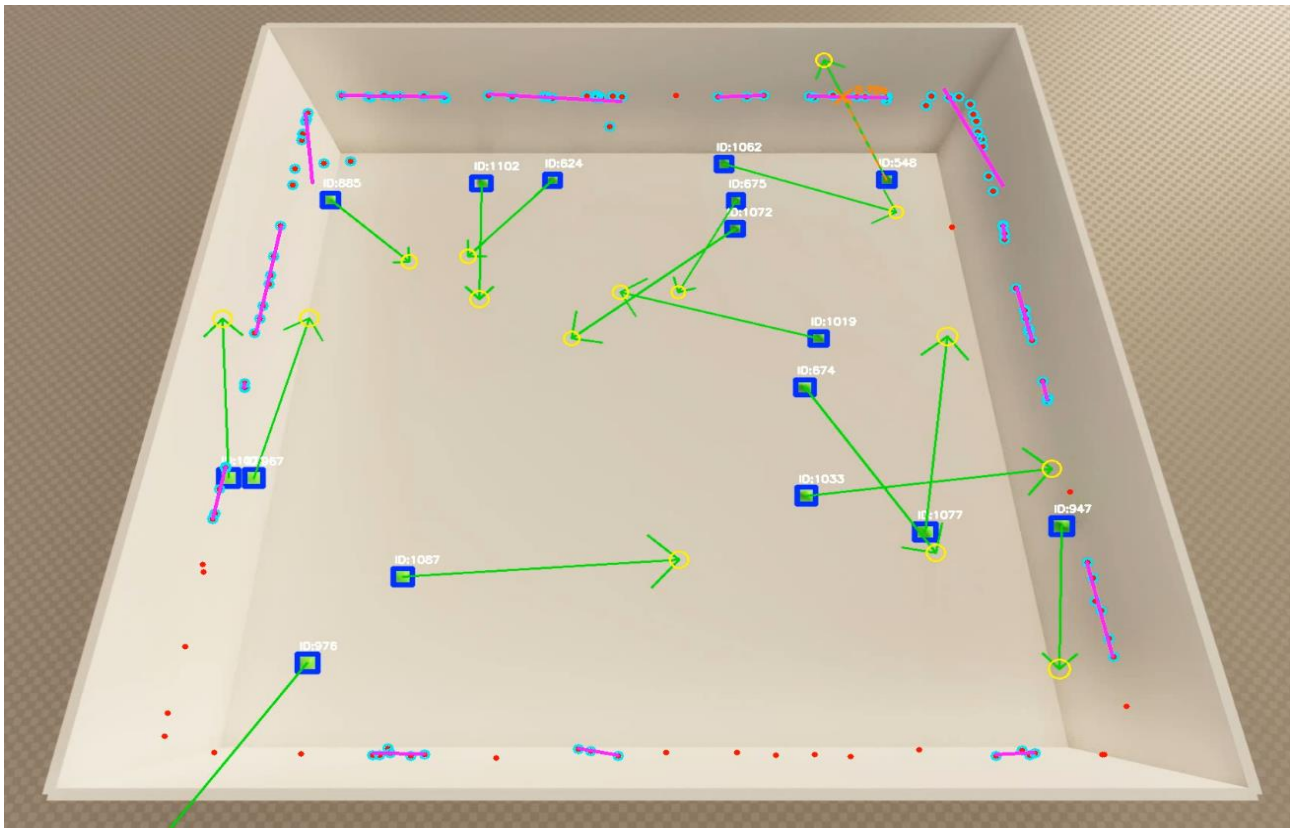


Рисунок 3.5 – Візуалізація стрілок передбачення руху, точок відскоків та виявлених стін

Точки відскоків. Функція draw_bounce_points() малює червоні кола радіусом 4 px у кожній зафіксованій точці відскоку.

Функція draw_walls() для кожної стіни:

– малює жовті кола на точках кластера;
– малює фіолетову лінію стіни;
– підписує рівняння прямої $Ax + By + C = 0$.

При увімкненій стабілізації координати стін та точок кластерів перетворюються з світових у екранні через stabilizer.to_screen_space().

Прогнози зіткнень. Функція draw_collision_predictions() для кожного прогнозу:

- пунктирна лінія від об’єкта до точки зіткнення (помаранчева);
- хрестик (×) на місці зіткнення;
- таймер – час до зіткнення у секундах (наприклад, 0.35s).

Пунктирна лінія реалізована функцією _draw_dashed_line(), що розбиває відрізок на n сегментів і малює кожен другий.

Запис результатів здійснюється через cv2.VideoWriter з кодеком mp4v. Кожен анотований кадр записується у файл result.mp4. Одночасно кадр відображається у вікні через cv2.imshow(). Натискання клавіші «q» зупиняє обробку. Кольорова схема візуалізації наведена у таблиці 3.4.

Таблиця 3.4 – Кольорова схема візуалізації (BGR)

Елемент	Колір BGR	Відображення
Стрілка передбачення	(0, 255, 0)	Зелений
Коло передбачення	(0, 255, 255)	Жовтий
Точка відскоку	(0, 0, 255)	Червоний
Лінія стіни	(255, 0, 255)	Фіолетовий
Точки кластера	(255, 255, 0)	Блакитний
Прогноз зіткнення	(0, 128, 255)	Помаранчевий

4 ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ ТА ТЕСТУВАННЯ

4.1 Опис тестового середовища

Для експериментального дослідження системи створено контрольоване тестове середовище в ігровому рушії Unreal Engine 5 [28].

Тестова сцена являє собою замкнену прямокутну коробку (арену) з чотирма стінами, підлогою та стелею. Всередині арени вільно рухаються 16 куль різних кольорів та розмірів, які відбиваються від стін за законами пружного відскоку (рис. 4.1). Перешкоди всередині коробки відсутні – об’єкти взаємодіють лише зі стінами та між собою.

Вибір симуляційного середовища обумовлений кількома перевагами:

- контрольованість – відомі точні параметри сцени (кількість об’єктів, розміри арени, швидкості);
- повторюваність – можливість багаторазового запуску з однаковими початковими умовами;
- відсутність зовнішніх факторів – стабільне освітлення, відсутність оклюзій сторонніми об’єктами;
- можливість керування камерою – тестування як зі статичною, так і з рухомою камерою.

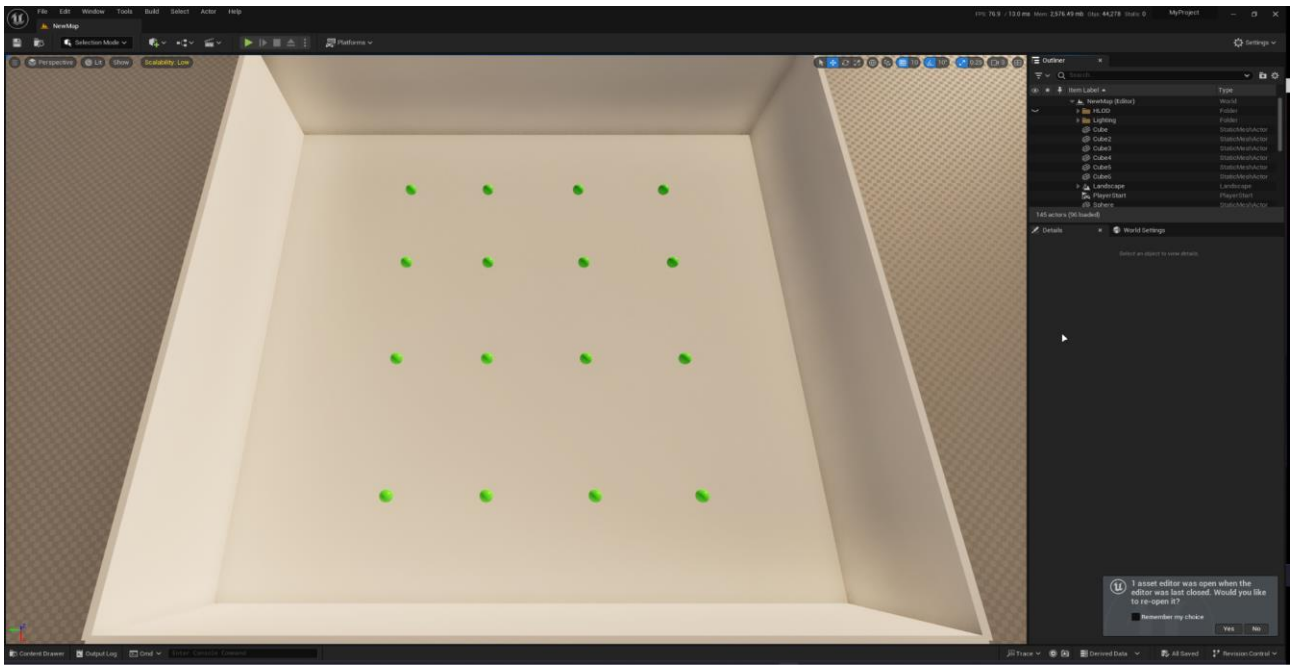


Рисунок 4.1 – Тестова сцена у середовищі Unreal Engine 5: замкнена арена з 16 рухомими об’єктами і заданим початковим імпульсом

Параметри відеозаписів. Для тестування записано кілька відеороликів із різними сценаріями руху камери (табл. 4.1).

Таблиця 4.1 – Параметри тестових відеозаписів

Параметр	Значення
Роздільна здатність	2560 × 1440 (2K QHD)
Частота кадрів	60 FPS
Кодек	H.264
Кількість об’єктів	16
Тривалість	~40 секунд (2442 кадри)
Рух камери	Повільне обертання
Форма арени	Прямокутна коробка

4.2 Оцінка точності детекції та відстеження

Оцінка точності детекції виконана на валідаційній вибірці датасету (16 зображень), що не використовувалась при навчанні. Результати отримано з логів навчання моделі YOLOv12n (табл. 4.2, рис. 4.2, рис. 4.3).

Таблиця 4.2 – Метрики детекції YOLOv12n на валідаційній вибірці

Метрика	Значення	Цільове значення
Precision	0.996	—
Recall	0.989	—
mAP@50	0.995	≥ 0.95
mAP@50-95	0.696	—

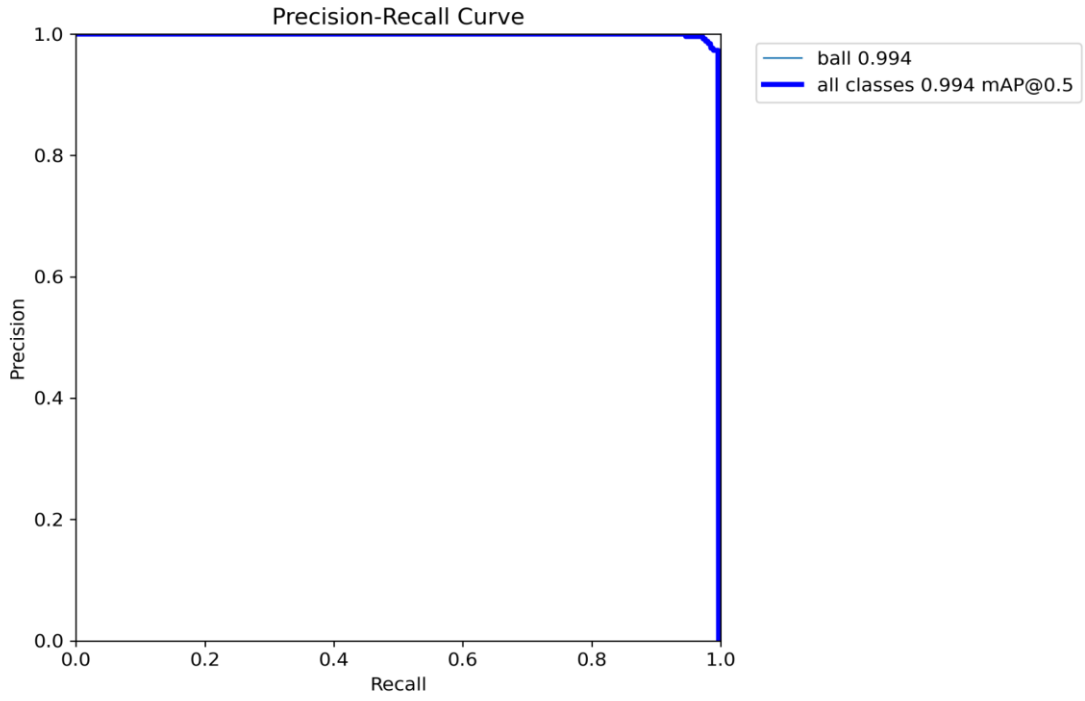


Рисунок 4.2 – Крива Precision-Recall для моделі YOLOv12n на валідаційній вибірці

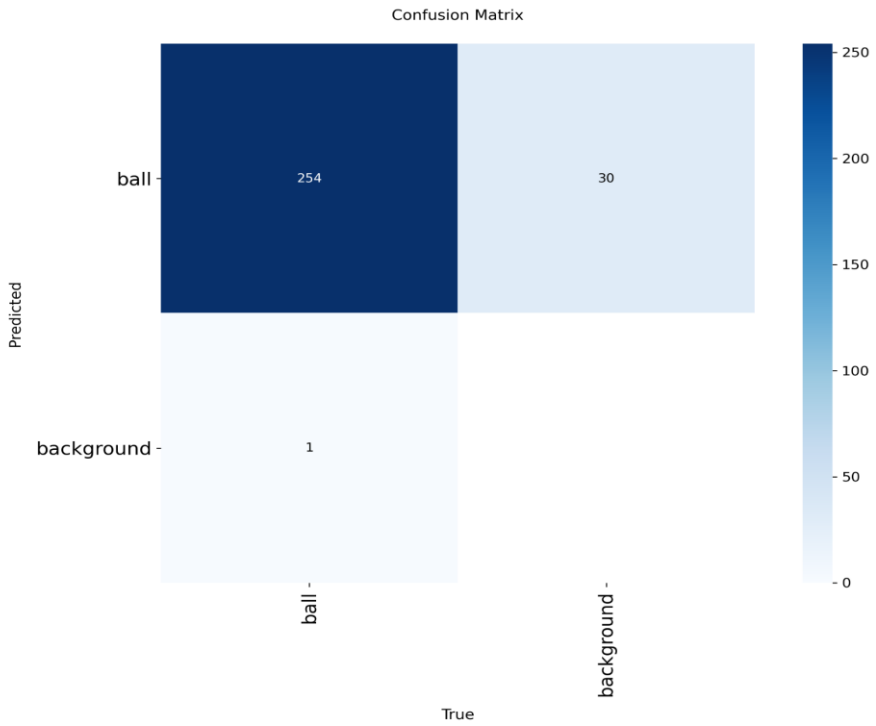


Рисунок 4.3 – Нормалізована матриця помилок (confusion matrix) моделі YOLOv12n

Метрика mAP@50 = 0.995 значно перевищує цільовий показник 0.95, встановлений у вимогах (NFR-1). Висока точність пояснюється:

– однорідним фоном тестового середовища;
– чіткими контурами об’єктів (кулі);
– ефективним transfer learning з моделі, попередньо навченої на COCO;
– агресивними аугментаціями при навчанні.

Метрика mAP@50-95 = 0.696 є нижчою, що типово для суворих порогів IoU. Це означає, що хоча об’єкти детектуються надійно, точність локалізації (збіг bounding box з ground truth) має простір для покращення.

Якість трекінгу оцінювалась візуально на тестових відеозаписах. Спостерігались такі характеристики:

– стабільність ID. Ідентифікатори об’єктів зберігаються стабільно протягом тривалих інтервалів. Переключення ID (ID switches) відбуваються рідко і переважно при щільному зближенні кількох куль;

– відновлення після зникнення. При тимчасовому виході об’єкта з поля зору (або повному перекритті іншим об’єктом) трекер BoT-SORT здатний відновити ідентифікатор завдяки моделі зовнішнього вигляду;
– робота з 16 об’єктами. Система коректно відстежує всі 16 об’єктів одночасно, що підтверджує виконання вимоги NFR-5.

4.3 Оцінка точності прогнозування траєкторій

Якісна оцінка. Візуальний аналіз стрілок передбачення на тестових відео показав:

а) прямолінійний рух. При рівномірному прямолінійному русі стрілка передбачення точно вказує напрямок та приблизну точку, де об’єкт опиниться через 1 секунду. Похибка мінімальна;
б) після відскоку. Одразу після відскоку стрілка може вказувати некоректний напрямок протягом $N = 10$ кадрів (час заповнення буфера історії новими даними). Це очікувана поведінка лінійної екстраполяції, що не враховує раптових змін напрямку;
в) зіткнення між об’єктами. При зіткненні двох куль передбачення стає некоректним до моменту стабілізації нового напрямку руху. Фільтрація ball-to-ball зіткнень коректно класифікує такі події.

Підсистема прогнозування зіткнень продемонструвала:

а) коректне виявлення стін. Після накопичення достатньої кількості відскоків (≥4) алгоритм DBSCAN коректно ідентифікує стіни арени. Рівняння прямих візуально збігаються з фактичним положенням стін;
б) точність прогнозу. Прогнозований час до зіткнення t_{sec} відповідає фактичному з похибкою, що залежить від:
   1) точності обчислення швидкості (згладжування по $N = 10$ кадрах);
   2) точності апроксимації стіни (залежить від кількості та розкиду точок відскоків);

3) припущення про рівномірний прямолінійний рух.

в) обмеження. Прогноз не враховує можливі зіткнення з іншими об’єктами на шляху до стіни, а також можливий відскок від проміжної стіни.

4.4 Оцінка продуктивності системи

Продуктивність системи оцінена експериментально шляхом покомпонентного профілювання на GPU NVIDIA RTX 3080 при обробці тестового відео роздільною здатністю 2560×1440, 60 FPS, з 16 об’єктами на сцені. Замір проводився за допомогою time.perf_counter() навколо кожного етапу конвеєра протягом 1239 кадрів (табл. 4.3).

Таблиця 4.3 – Результати профілювання продуктивності системи

Компонент	Час на кадр, мс	Частка, %
Детекція + трекінг (YOLO + BoT-SORT)	40.4	59.5
Стабілізація (оптичний потік, гомографія)	19.4	28.6
Запис відео (VideoWriter)	4.2	6.2
Зчитування кадру	1.8	2.7
Аналіз простору + прогнозування	0.9	1.3
Відображення вікна (imshow)	0.9	1.3
Візуалізація (відрисовка елементів)	0.2	0.3
Трекер траєкторій (TrajectoryTracker)	0.04	0.1
Разом	67.9	100

Загальна швидкість обробки: ~14.7 FPS, що перевищує вимогу NFR-2 (FPS ≥ 10). Профілювання виявило два основних вузьких місця:

- детекція + трекінг (59.5%). Виклик model.track() виконує на GPU інференс нейромережі YOLOv12n, а на CPU – повний цикл трекера BoT-SORT: оновлення фільтрів Калмана для 16 об’єктів, обчислення матриці IoU, розв’язання задачі призначення (Hungarian algorithm), оновлення моделі зовнішнього вигляду. Оскільки GPU залишається переважно незавантаженим, основна частина цих 40 мс припадає саме на CPU-логіку трекера;
- стабілізація (28.6%). Модуль компенсації руху камери виконує на CPU три ресурсомісткі операції на повній роздільній здатності 2560×1440: детекцію ключових точок Shi-Tomasi (goodFeaturesToTrack), оптичний потік Lucas-Kanade (calcOpticalFlowPyrLK) та обчислення гомографії RANSAC (findHomography).

Решта компонентів (запис, відрисовка, аналіз, прогнозування) разом займають лише ~12% часу.

Можливі шляхи оптимізації:

- зменшення роздільної здатності кадру для стабілізатора (downscale до 1280×720 перед пошуком ключових точок);
- перенесення оптичного потоку на GPU через cv2.cuda.calcOpticalFlowPyrLK;
- експорт моделі YOLO у формат TensorRT для зменшення накладних витрат інференсу;
- обробка кожного n-го кадру (frame skipping) з інтерполяцією позицій.

4.5 Аналіз результатів

Проведене експериментальне дослідження дозволяє зробити такі висновки щодо працездатності системи.

Що працює добре:

- детекція. Модель YOLOv12n забезпечує практично ідеальну детекцію (mAP@50 = 99.5%) на тестовому середовищі. Жодного пропущеного об’єкта та менше 0.4% хибних спрацьовувань;
- трекінг. Вбудований трекер BoT-SORT стабільно підтримує ідентифікатори 16 об’єктів одночасно. Переключення ID рідкісні;
- стабілізація. Алгоритм компенсації руху камери коректно працює при повільному обертанні камери. Маскування рухомих об’єктів та фільтрація мікрошуму ефективно запобігають дрейфу координат;
- виявлення стін. Кластеризація DBSCAN коректно ідентифікує стіни арени після накопичення достатньої кількості відскоків. Апроксимація прямими збігається з фактичним розташуванням стін;
- прогнозування зіткнень. Система коректно передбачає найближчу стіну зіткнення та обчислює час до нього з прийнятною точністю при прямолінійному русі.

Обмеження та напрямки покращення:

- лінійна екстраполяція. Поточний метод передбачення не моделює криволінійний рух та не враховує фізику відскоків. Для покращення можна використати поліноміальну регресію або фільтр Калмана з моделлю відскоку;
- затримка після відскоку. Після різкої зміни напрямку система потребує $N = 10$ кадрів для оновлення буфера історії, протягом яких стрілка передбачення вказує старий напрямок;
- швидкий рух камери. При різких рухах камери гомографія може бути неточною через недостатню кількість відстежених ключових точок. Система оптимізована для повільного обертання;
- продуктивність. 10 FPS є мінімально достатнім для аналізу, проте недостатнім для інтерактивних застосувань. Оптимізація через TensorRT або зменшення роздільної здатності може підвищити FPS у 2–3 рази;

– масштабованість на реальні сцени. Система навчена та протестована на синтетичних даних UE5. Для роботи з реальними відео необхідно дотренувати модель на відповідному датасеті.

Узагальнені дані щодо перевірки виконання функціональних та нефункциональних вимог наведено в табл. 4.4.

Таблиця 4.4 – Перевірка виконання вимог

Вимога	Опис	Цільове значення	Фактичне значення
NFR-1	Точність детекції mAP@50	≥ 95%	99.5%
NFR-2	Швидкість обробки	≥ 10 FPS	~10 FPS
NFR-3	Модульна архітектура	—	9 модулів
NFR-4	Єдиний конфігураційний файл	—	config.py
NFR-5	Масштабованість до 16 об’єктів	1–16	16
FR-1-10	Функціональні вимоги	—	Реалізовано

4.6 Висновок до розділу 4

Експериментальне дослідження підтвердило працездатність розробленої інтелектуальної системи аналізу та передбачення руху об’єктів. Основні результати:

– модель YOLOv12n, навчена на кастомному датасеті з 86 зображень, досягла mAP@50 = 99.5%, що значно перевищує цільовий показник 95%;

– система стабільно відстежує 16 об’єктів одночасно з мінімальною кількістю переключень ідентифікаторів;

– алгоритм стабілізації координат коректно компенсує рух камери, забезпечуючи правильний аналіз траєкторій;

- автоматичне виявлення стін через кластеризацію точок відскоків (DBSCAN) працює коректно після накопичення достатньої статистики.
- прогнозування зіткнень зі стінами дає коректні результати при прямолінійному русі об’єктів;
- швидкість обробки ~15 FPS на GPU RTX 3080 відповідає встановленій вимозі.

Усі функціональні та нефункціональні вимоги виконано. Визначено обмеження системи та напрямки подальшого розвитку.

ВИСНОВКИ

В результаті виконання кваліфікаційної роботи розроблено інтелектуальну систему аналізу та передбачення руху об’єктів у динамічному середовищі на основі нейромережевої моделі YOLOv12.

Для досягнення поставленої мети було виконано такі задачі:

- проведено аналіз предметної області. Розглянуто еволюцію методів детекції (від HOG+SVM до YOLOv12), алгоритми відстеження (SORT, DeepSORT, ByteTrack, BoT-SORT) та методи прогнозування траєкторій (лінійна екстраполяція, фільтр Калмана). Обґрунтовано вибір YOLOv12n як оптимальної моделі за співвідношенням точності та швидкості;
- спроєктовано модульну конвеєрну архітектуру системи, що складається з 9 модулів і охоплює повний цикл обробки: від захоплення відеокадру до візуалізації прогнозу;
- підготовлено навчальний датасет із 86 зображень (70 train + 16 val), розмічених у CVAT. Навчена модель YOLOv12n досягла точності детекції mAP@50 = 99.5% (Precision = 99.6%, Recall = 98.9%) при навчанні протягом 300 епох на GPU RTX 3080;
- реалізовано алгоритм компенсації руху камери на основі оптичного потоку Lucas-Kanade та матриці гомографії з маскуванням рухомих об’єктів і фільтрацією мікрошуму;
- реалізовано підсистему аналізу траєкторій, що включає: обчислення вектора швидкості, детекцію відскоків (за кутом між векторами руху), фільтрацію зіткнень між об’єктами та кластеризацію точок відскоків алгоритмом DBSCAN для автоматичного виявлення стін;
- реалізовано підсистему прогнозування, що виконує лінійну екстраполяцію траєкторій та передбачення зіткнень зі стінами методом ray-casting із обчисленням точки та часу зіткнення;

- проведено експериментальне дослідження на тестових відеозаписах із симуляції в Unreal Engine 5 (2560×1440, 60 FPS, 16 об’єктів). Підтверджено виконання всіх функціональних та нефункціональних вимог. Швидкість обробки – ~10 FPS на GPU NVIDIA RTX 3080.

Система забезпечує детекцію об’єктів у реальному часі, відстеження їхніх траєкторій, прогнозування руху та аналіз взаємодій із середовищем. Експериментальне дослідження підтвердило працездатність запропонованих алгоритмів. Мета роботи досягнута.

ПЕРЕЛІК ПОСИЛАНЬ

1. Szeliski R. Computer Vision : Algorithms and Applications. 2nd ed. Springer, 2021. 979 p. ISBN 9783030343712. URL : https://szeliski.org/Book/ (дата звернення : 12.02.2026).

2. Dalal N., Triggs B. Histograms of Oriented Gradients for Human Detection. URL : https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf (дата звернення : 14.02.2026).

3. Viola P., Jones M. Rapid Object Detection using a Boosted Cascade of Simple Features. URL : https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf (дата звернення : 14.02.2026).

4. Girshick R., Donahue J., Darrell T., Malik J. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. URL : https://arxiv.org/abs/1311.2524 (дата звернення : 18.02.2026).

5. Girshick R. Fast R-CNN. URL : https://arxiv.org/abs/1504.08083 (дата звернення : 18.02.2026).

6. Ren S., He K., Girshick R., Sun J. Faster R-CNN : Towards Real-Time Object Detection with Region Proposal Networks. URL : https://arxiv.org/abs/1506.01497 (дата звернення : 20.02.2026).

7. Liu W., Anguelov D., Erhan D. et al. SSD: Single Shot MultiBox Detector. URL : https://arxiv.org/abs/1512.02325 (дата звернення : 20.02.2026).

8. Redmon J., Divvala S., Girshick R., Farhadi A. You Only Look Once: Unified, Real-Time Object Detection. URL : https://arxiv.org/abs/1506.02640 (дата звернення : 05.02.2026).

9. Redmon J., Farhadi A. YOLO9000: Better, Faster, Stronger. URL : https://arxiv.org/abs/1612.08242 (дата звернення : 22.02.2026).

10. Redmon J., Farhadi A. YOLOv3 : An Incremental Improvement. URL : https://arxiv.org/abs/1804.02767 (дата звернення : 22.02.2026).

11. Bochkovskiy A., Wang C.-Y., Liao H.-Y.M. YOLOv4 : Optimal Speed and Accuracy of Object Detection. URL : https://arxiv.org/abs/2004.10934 (дата звернення : 25.02.2026).

12. Tian Y., Ye Q., Doermann D. YOLOv12: Attention-Centric Real-Time Object Detectors. URL : https://arxiv.org/abs/2502.12524 (дата звернення : 10.03.2026).

13. Bewley A., Ge Z., Ott L., Ramos F., Upcroft B. Simple Online and Realtime Tracking. URL : https://arxiv.org/abs/1602.00763 (дата звернення : 03.03.2026).

14. Wojke N., Bewley A., Paulus D. Simple Online and Realtime Tracking with a Deep Association Metric. URL : https://arxiv.org/abs/1703.07402 (дата звернення : 03.03.2026).

15. Zhang Y., Sun P., Jiang Y. et al. ByteTrack : Multi-Object Tracking by Associating Every Detection Box. URL : https://arxiv.org/abs/2110.06864 (дата звернення : 07.03.2026).

16. Aharon N., Orfaig R., Bobrovsky B.-Z. BoT-SORT: Robust Associations Multi-Pedestrian Tracking. URL : https://arxiv.org/abs/2206.14651 (дата звернення : 07.03.2026).

17. Kalman R. E. A New Approach to Linear Filtering and Prediction Problems. *Journal of Basic Engineering*, 1960. Vol. 82, No. 1. P. 35–45. URL : https://www.cs.unc.edu/~welch/kalman/media/pdf/Kalman1960.pdf (дата звернення : 15.03.2026).

18. Goodfellow I., Bengio Y., Courville A. Deep Learning. MIT Press, 2016. 800 p. ISBN 9780262035613. URL : https://www.deeplearningbook.org/ (дата звернення : 12.02.2026).

19. Ester M., Kriegel H.-P., Sander J., Xu X. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise (DBSCAN). URL : https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf (дата звернення : 20.03.2026).

20. Офіційна документація Ultralytics YOLO. URL : https://docs.ultralytics.com/ (дата звернення : 15.04.2026).

21. Офіційний репозиторій Ultralytics на GitHub. URL : https://github.com/ultralytics/ultralytics (дата звернення : 15.04.2026).

22. Офіційна документація OpenCV. URL : https://docs.opencv.org/ (дата звернення : 28.03.2026).

23. Офіційний веб-сайт Python. URL : https://www.python.org/ (дата звернення : 10.04.2026).

24. Платформа для анотації зображень CVAT. URL : https://www.cvat.ai/ (дата звернення : 05.05.2026).

25. Документація NumPy. URL : https://numpy.org/doc/ (дата звернення : 28.03.2026).

26. Документація scikit-learn: DBSCAN. URL : https://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html (дата звернення : 22.04.2026).

27. NVIDIA CUDA Toolkit Documentation. URL : https://docs.nvidia.com/cuda/ (дата звернення : 18.05.2026).

28. Документація Unreal Engine 5. URL : https://dev.epicgames.com/documentation/en-us/unreal-engine/ (дата звернення : 04.06.2026).

29. Датасет MS COCO: Common Objects in Context. URL : https://cocodataset.org/ (дата звернення : 25.02.2026).

30. Датасет MOT Challenge. URL : https://motchallenge.net/ (дата звернення : 05.03.2026).

31. Hasan M., Rabby M. K. M., Jahan I., Soeb Md. J. A., Jubayer Md. F. The Evolution and Advancement of YOLO Algorithms in Object Detection : From Real-Time Breakthroughs to Modern Architectures. URL : https://www.preprints.org/manuscript/202510.2019 (дата звернення : 27.05.2026).

ДОДАТОК А

Головний конвеєр системи (main.py)

```
import time
import cv2
from analyzer import detect_walls
from config import OUTPUT_PATH, STABILIZATION_ENABLED, VIDEO_PATH
from detector import Detector
from predictor import predict_collisions
from stabilizer import CameraStabilizer
from tracker import TrajectoryTracker
from video_io import create_writer, open_video
from visualizer import (
    draw_bounce_points, draw_collision_predictions, draw_hud,
    draw_prediction_arrow, draw_track_id, draw_walls,
)


def main() -> None:
    # --- Ініціалізація ---
    cap, video_info = open_video(VIDEO_PATH)
    print(
        f"Відео: {VIDEO_PATH}  |  "
        f"{video_info.width}×{video_info.height}  |  "
        f"{video_info.fps:.1f} FPS"
    )

    out = create_writer(OUTPUT_PATH, video_info.fps, video_info.width, video_info.height)
    detector = Detector()
    stabilizer = CameraStabilizer() if STABILIZATION_ENABLED else None
    tracker = TrajectoryTracker()

    frame_idx = 0
    prev_time = time.time()

    # --- ПРОФІЛЮВАННЯ ---
    from collections import defaultdict
    timings = defaultdict(float)
    perf = time.perf_counter  # скорочення

    # --- Покадрова обробка ---
    while True:
        t0 = perf()
        ok, frame = cap.read()
        if not ok:
            break
        frame_idx += 1
        timings["0_read"] += perf() - t0

        # FPS обробки
        now = time.time()
        process_fps = 1.0 / max(now - prev_time, 1e-9)
        prev_time = now

        # 1. Детекція та трекінг
```

```
        t1 = perf()
        annotated, detections = detector.detect_and_track(frame)
        timings["1_detect+track"] += perf() - t1

        # Зберігаємо екранні координати для відрисовки
        screen_coords = {det.track_id: (det.cx, det.cy) for det in detections}

        # 2. Стабілізація координат (компенсація руху камери)
        t2 = perf()
        if stabilizer is not None:
            H = stabilizer.update(frame, detections)
            stabilized = stabilizer.stabilize_detections(detections, H)
        else:
            stabilized = detections
        timings["2_stabilize"] += perf() - t2

        # 3. Оновлення траєкторій, детекція відскоків (у стабілізованих координатах)
        t3 = perf()
        tracked = tracker.update(stabilized, frame_idx)
        timings["3_tracker"] += perf() - t3

        # 4. Стрілки передбачення та ID (малюємо в екранних координатах)
        t4 = perf()
        for det, velocity in tracked:
            screen_pos = screen_coords.get(det.track_id, (det.cx, det.cy))
            draw_track_id(annotated, screen_pos, det.track_id)
            if velocity is not None:
                vx, vy = velocity
                draw_prediction_arrow(
                    annotated, screen_pos, vx, vy,
                    video_info.fps, det.radius,
                )

        # 5. Аналіз простору – виявлення стін (у стабілізованих координатах)
        walls = detect_walls(tracker.bounce_points)

        # 6. Прогнозування зіткнень зі стінами
        objects_for_prediction = []
        for det, velocity in tracked:
            if velocity is not None:
                objects_for_prediction.append((
                    det.track_id,
                    (det.cx, det.cy),
                    velocity,
                    det.radius,
                ))
        collisions = predict_collisions(objects_for_prediction, walls, video_info.fps)
        timings["4_analyze+predict"] += perf() - t4

        # 7. Візуалізація
        t5 = perf()
        if stabilizer is not None:
            screen_bounces = stabilizer.to_screen_space(tracker.bounce_points, H)
            collision_world_pts = [c.collision_point for c in collisions]
            collision_screen_pts = stabilizer.to_screen_space(collision_world_pts, H) if
collision_world_pts else None
        else:
            screen_bounces = tracker.bounce_points
            collision_screen_pts = None

        draw_hud(annotated, video_info.fps, process_fps, frame_idx)
```

```
        draw_bounce_points(annotated, screen_bounces)
        draw_walls(annotated, walls, stabilizer, H if stabilizer else None)
        draw_collision_predictions(annotated,          collisions,          screen_coords,
collision_screen_pts)
        timings["5_visualize"] += perf() - t5

        # 8. Запис та відображення
        t6 = perf()
        out.write(annotated)
        timings["6_write"] += perf() - t6

        t7 = perf()
        cv2.imshow("YOLO Tracking & Prediction", annotated)
        timings["7_imshow"] += perf() - t7

        if cv2.waitKey(1) & 0xFF == ord("q"):
            break

    # --- Очищення ---
    cap.release()
    out.release()
    cv2.destroyAllWindows()

    # --- ВИВІД ПРОФІЛЮВАННЯ ---
    print(f"\n{'='*60}")
    print(f"ПРОФІЛЮВАННЯ  |  Оброблено кадрів: {frame_idx}")
    print(f"{'='*60}")
    total = sum(timings.values())
    for key in sorted(timings.keys()):
        t = timings[key]
        avg_ms = (t / frame_idx) * 1000 if frame_idx > 0 else 0
        pct = (t / total) * 100 if total > 0 else 0
        print(f"  {key:25s}  {avg_ms:7.1f} мс/кадр  ({pct:5.1f}%)")
    avg_total = (total / frame_idx) * 1000 if frame_idx > 0 else 0
    print(f"  {'РАЗОМ':25s}  {avg_total:7.1f} мс/кадр  (100.0%)")
    print(f"  {'Реальний FPS':25s}  {1000/avg_total:.1f}" if avg_total > 0 else "")
    print(f"{'='*60}")
    print(f"Результат: {OUTPUT_PATH}")


if __name__ == "__main__":
    main()
```

ДОДАТОК Б

Конфігурація параметрів системи (config.py)

```
from typing import Tuple

MODEL_PATH: str = r"runs\detect\train-3\weights\best.pt"
VIDEO_PATH: str = "ue5BallsTest5.mp4"
OUTPUT_PATH: str = "result.mp4"

CONFIDENCE_THRESHOLD: float = 0.5       # Поріг впевненості моделі
HISTORY_FRAMES: int = 10                # Довжина історії позицій (кадри)
PREDICTION_TIME_SEC: float = 1.0        # Горизонт передбачення (секунди)

BOUNCE_ANGLE_THRESHOLD: float = 0.5     # cos(кут): 0.5 ≈ 60°; якщо cos < порогу – відскок
MIN_SPEED_FOR_BOUNCE: float = 5.0       # Мін. швидкість (px/кадр), нижче – тремтіння
BOUNCE_COOLDOWN_FRAMES: int = 15        # Пауза між фіксаціями відскоку одного об’єкта

COLLISION_DISTANCE_MULTIPLIER: float = 3.0  # Множник радіусу для детекції зіткнення

DBSCAN_EPS: float = 50.0               # Макс. відстань між точками одного кластера (пікселі)
DBSCAN_MIN_SAMPLES: int = 3            # Мін. кількість відскоків для фіксації "стіни"
MIN_BOUNCE_POINTS_FOR_WALLS: int = 4   # Мін. кількість точок для запуску кластеризації

STABILIZATION_ENABLED: bool = True      # Увімкнути/вимкнути стабілізацію

# Shi-Tomasi corner detector
STAB_MAX_CORNERS: int = 200             # Макс. кількість ключових точок фону
STAB_QUALITY_LEVEL: float = 0.01        # Мін. якість кутової точки (0..1)
STAB_MIN_DISTANCE: float = 30.0         # Мін. відстань між точками (пікселі)

# Lucas-Kanade optical flow
STAB_LK_WIN_SIZE: int = 21              # Розмір вікна пошуку
STAB_LK_MAX_LEVEL: int = 3              # Макс. рівень піраміди

# RANSAC homography
STAB_RANSAC_THRESHOLD: float = 5.0      # Поріг репроєкції (пікселі)
STAB_MIN_MATCHES: int = 10              # Мін. кількість відповідностей

# Фільтрація шуму
STAB_MOTION_THRESHOLD: float = 0.5      # Мін. зміщення (px) для фіксації руху камери
STAB_MASK_PADDING: int = 20             # Розширення маски навколо bbox об’єкта (px)

COLOR_ARROW: Tuple[int, int, int] = (0, 255, 0)          # Зелена стрілка передбачення
COLOR_PREDICTION: Tuple[int, int, int] = (0, 255, 255)   # Жовте коло передбачення
COLOR_BOUNCE: Tuple[int, int, int] = (0, 0, 255)         # Червона точка відскоку
COLOR_VIDEO_FPS: Tuple[int, int, int] = (0, 255, 255)
COLOR_PROCESS_FPS: Tuple[int, int, int] = (0, 255, 0)
COLOR_FRAME: Tuple[int, int, int] = (255, 255, 255)
COLOR_WALL: Tuple[int, int, int] = (255, 0, 255)         # Фіолетова лінія стіни
COLOR_CLUSTER_POINT: Tuple[int, int, int] = (255, 255, 0)  # Жовті точки кластерів
COLOR_COLLISION: Tuple[int, int, int] = (0, 128, 255)     # Помаранчевий – прогноз зіткнення

HistoryPoint = Tuple[float, float, int]
```

ДОДАТОК В

Модуль детекції та трекінгу (detector.py)

```
from dataclasses import dataclass
from typing import List, Optional

import torch
from ultralytics import YOLO

from config import MODEL_PATH, CONFIDENCE_THRESHOLD


@dataclass
class DetectedObject:
    track_id: int
    cx: float       # Центр X
    cy: float       # Центр Y
    width: float     # Ширина bbox
    height: float    # Висота bbox

    @property
    def radius(self) -> int:
        """Радіус об'єкта (половина більшої сторони bbox)."""
        return int(max(self.width, self.height) / 2)


class Detector:
    def __init__(self, model_path: str = MODEL_PATH) -> None:
        self.model = YOLO(model_path)

    def detect_and_track(self, frame) -> tuple:
        results = self.model.track(
            frame, persist=True, conf=CONFIDENCE_THRESHOLD, verbose=False
        )

        annotated = results[0].plot(conf=False, labels=False)
        detections: List[DetectedObject] = []

        boxes_data = results[0].boxes
        if boxes_data is not None and boxes_data.id is not None:
            boxes = boxes_data.xywh.cpu()
            track_ids = boxes_data.id.int().cpu().tolist()

            for box, track_id in zip(boxes, track_ids):
                x, y, w, h = box
                detections.append(DetectedObject(
                    track_id=track_id,
                    cx=float(x),
                    cy=float(y),
                    width=float(w),
                    height=float(h),
                ))

        return annotated, detections
```

ДОДАТОК Г

Модуль компенсації руху камери (stabilizer.py)

```
from typing import List, Optional, Tuple

import cv2
import numpy as np

from config import (
    STAB_MAX_CORNERS,
    STAB_QUALITY_LEVEL,
    STAB_MIN_DISTANCE,
    STAB_LK_WIN_SIZE,
    STAB_LK_MAX_LEVEL,
    STAB_RANSAC_THRESHOLD,
    STAB_MIN_MATCHES,
    STAB_MOTION_THRESHOLD,
    STAB_MASK_PADDING,
)
from detector import DetectedObject


class CameraStabilizer:
    def __init__(self) -> None:
        self._prev_gray: Optional[np.ndarray] = None
        self._prev_detections: List[DetectedObject] = []

        # Накопичена гомографія: поточний кадр → перший кадр (референсний)
        self._H_cumulative: np.ndarray = np.eye(3, dtype=np.float64)

        # Параметри Lucas-Kanade
        self._lk_params = dict(
            winSize=(STAB_LK_WIN_SIZE, STAB_LK_WIN_SIZE),
            maxLevel=STAB_LK_MAX_LEVEL,
            criteria=(
                cv2.TERM_CRITERIA_EPS | cv2.TERM_CRITERIA_COUNT,
                30,     # макс. ітерацій
                0.01,   # точність
            ),
        )

        # Параметри Shi-Tomasi corner detector
        self._feature_params = dict(
            maxCorners=STAB_MAX_CORNERS,
            qualityLevel=STAB_QUALITY_LEVEL,
            minDistance=STAB_MIN_DISTANCE,
        )

    def update(
        self,
        frame: np.ndarray,
        detections: List[DetectedObject],
    ) -> np.ndarray:
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        if self._prev_gray is None:
```

```
            self._prev_gray = gray
            self._prev_detections = detections
            return self._H_cumulative.copy()

        # Маска: виключаємо області рухомих об'єктів (попереднього кадру)
        mask = self._create_background_mask(
            gray.shape, self._prev_detections
        )

        H_frame = self._compute_frame_homography(self._prev_gray, gray, mask)

        if H_frame is not None:
            # Перевіряємо, чи камера справді рухається
            # Якщо зміщення мінімальне — ігноруємо (не накопичуємо шум)
            if self._is_significant_motion(H_frame):
                self._H_cumulative = self._H_cumulative @ H_frame

        self._prev_gray = gray
        self._prev_detections = detections
        return self._H_cumulative.copy()

    def stabilize_detections(
        self,
        detections: List[DetectedObject],
        H: np.ndarray,
    ) -> List[DetectedObject]:
           if not detections:
            return detections

        # Збираємо центри в масив [N, 1, 2] для perspectiveTransform
        centers = np.array(
            [[det.cx, det.cy] for det in detections],
            dtype=np.float64,
        ).reshape(-1, 1, 2)

        # Трансформуємо
        transformed = cv2.perspectiveTransform(centers, H)

        stabilized: List[DetectedObject] = []
        for det, new_pt in zip(detections, transformed):
            stabilized.append(DetectedObject(
                track_id=det.track_id,
                cx=float(new_pt[0][0]),
                cy=float(new_pt[0][1]),
                width=det.width,
                height=det.height,
            ))

        return stabilized

    def to_screen_space(
        self,
        points: list,
        H: np.ndarray,
    ) -> list:
        if not points:
            return points

        H_inv = np.linalg.inv(H)

        pts = np.array(points, dtype=np.float64).reshape(-1, 1, 2)
```

```
    transformed = cv2.perspectiveTransform(pts, H_inv)

    return [
        (int(pt[0][0]), int(pt[0][1]))
        for pt in transformed
    ]

def _create_background_mask(
    self,
    shape: tuple,
    detections: List[DetectedObject],
) -> np.ndarray:
    h, w = shape[:2]
    mask = np.ones((h, w), dtype=np.uint8) * 255

    for det in detections:
        # Bbox з центру (cx, cy, width, height) → (x1, y1, x2, y2) + padding
        pad = STAB_MASK_PADDING
        x1 = max(0, int(det.cx - det.width / 2) - pad)
        y1 = max(0, int(det.cy - det.height / 2) - pad)
        x2 = min(w, int(det.cx + det.width / 2) + pad)
        y2 = min(h, int(det.cy + det.height / 2) + pad)
        mask[y1:y2, x1:x2] = 0

    return mask

def _is_significant_motion(self, H: np.ndarray) -> bool:
    # Перевіряємо на None (може повернути cv2.findHomography)
    if H is None:
        return False

    # Різниця з одиничною матрицею
    diff = np.abs(H - np.eye(3, dtype=np.float64))

    # Трансляційна компонента (tx, ty) - найпоказовіша
    translation = max(abs(H[0, 2]), abs(H[1, 2]))

    # Також перевіряємо обертальну компоненту
    # (відхилення діагоналі від 1, та недіагональних від 0)
    rotation_diff = max(
        abs(H[0, 0] - 1.0),
        abs(H[1, 1] - 1.0),
        abs(H[0, 1]),
        abs(H[1, 0]),
    )

    return translation > STAB_MOTION_THRESHOLD or rotation_diff > 0.001

def _compute_frame_homography(
    self,
    prev_gray: np.ndarray,
    curr_gray: np.ndarray,
    mask: np.ndarray,
) -> Optional[np.ndarray]:
    # Ключові точки ТІЛЬКИ на фоні (mask виключає м'ячі)
    prev_pts = cv2.goodFeaturesToTrack(
        prev_gray, mask=mask, **self._feature_params
    )

    if prev_pts is None or len(prev_pts) < STAB_MIN_MATCHES:
        return None
```

```
# Оптичний потік Lucas-Kanade
curr_pts, status, _ = cv2.calcOpticalFlowPyrLK(
    prev_gray, curr_gray, prev_pts, None, **self._lk_params
)

if curr_pts is None:
    return None

# Фільтрація: тільки успішно відстежені точки
status = status.ravel()
good_prev = prev_pts[status == 1].reshape(-1, 2)
good_curr = curr_pts[status == 1].reshape(-1, 2)

if len(good_prev) < STAB_MIN_MATCHES:
    return None

# 4. Гомографія (RANSAC для робастності)
H, inlier_mask = cv2.findHomography(
    good_curr,     # звідки (поточний кадр)
    good_prev,     # куди (попередній кадр → ближче до референсу)
    cv2.RANSAC,
    ransacReprojThreshold=STAB_RANSAC_THRESHOLD,
)

return H
```

ДОДАТОК Д

Модуль аналізу траєкторій (tracker.py)

```
import math
from collections import defaultdict, deque
from typing import Deque, Dict, List, Optional, Tuple

from config import (
    BOUNCE_ANGLE_THRESHOLD,
    BOUNCE_COOLDOWN_FRAMES,
    COLLISION_DISTANCE_MULTIPLIER,
    HISTORY_FRAMES,
    MIN_SPEED_FOR_BOUNCE,
    HistoryPoint,
)
from detector import DetectedObject


class TrajectoryTracker:

    def __init__(self) -> None:
        self.track_history: Dict[int, Deque[HistoryPoint]] = defaultdict(
            lambda: deque(maxlen=HISTORY_FRAMES)
        )
        self.bounce_cooldown: Dict[int, int] = defaultdict(int)
        self.bounce_points: List[Tuple[int, int]] = []

    def update(
        self,
        detections: List[DetectedObject],
        frame_idx: int,
    ) -> List[Tuple[DetectedObject, Optional[Tuple[float, float]]]]:
        results: List[Tuple[DetectedObject, Optional[Tuple[float, float]]]] = []

        for det in detections:
            history = self.track_history[det.track_id]
            history.append((det.cx, det.cy, frame_idx))

            if len(history) < HISTORY_FRAMES:
                results.append((det, None))
                continue

            # Зменшуємо кулдаун відскоку
            if self.bounce_cooldown[det.track_id] > 0:
                self.bounce_cooldown[det.track_id] -= 1

            # Детекція відскоку
            self._check_bounce(det, history, frame_idx)

            # Обчислення швидкості
            velocity = self._compute_velocity(history[0], history[-1])
            results.append((det, velocity))

        return results

    def _compute_velocity(
```

```
    self,
    p_start: HistoryPoint,
    p_end: HistoryPoint,
) -> Optional[Tuple[float, float]]:

    delta_frames = p_end[2] - p_start[2]
    if delta_frames <= 0:
        return None
    vx = (p_end[0] - p_start[0]) / delta_frames
    vy = (p_end[1] - p_start[1]) / delta_frames
    return vx, vy

def _detect_bounce(
    self,
    history: Deque[HistoryPoint],
    cooldown: int,
) -> Tuple[bool, Optional[Tuple[int, int]]]:
    p_start = history[0]
    p_mid = history[len(history) // 2]
    p_end = history[-1]

    # Вектори «до» і «після» середньої точки
    vx_before = p_mid[0] - p_start[0]
    vy_before = p_mid[1] - p_start[1]
    vx_after = p_end[0] - p_mid[0]
    vy_after = p_end[1] - p_mid[1]

    mag_before = math.hypot(vx_before, vy_before)
    mag_after = math.hypot(vx_after, vy_after)

    # Занадто повільно  не відскок, а тремтіння
    if mag_before <= MIN_SPEED_FOR_BOUNCE or mag_after <= MIN_SPEED_FOR_BOUNCE:
        return False, None

    if cooldown > 0:
        return False, None

    # Косинус кута між двома векторами руху
    dot = vx_before * vx_after + vy_before * vy_after
    cos_angle = dot / (mag_before * mag_after)

    if cos_angle < BOUNCE_ANGLE_THRESHOLD:
        bounce_point = (int(p_mid[0]), int(p_mid[1]))
        return True, bounce_point

    return False, None

def _is_ball_collision(
    self,
    track_id: int,
    bounce_pt: Tuple[int, int],
    mid_frame: int,
    radius: int,
) -> bool:

    for other_id, other_history in self.track_history.items():
        if other_id == track_id:
            continue

        for ox, oy, oframe in other_history:
            if abs(oframe - mid_frame) <= 2:
```

```
                dist = math.hypot(ox - bounce_pt[0], oy - bounce_pt[1])
                if dist < radius * COLLISION_DISTANCE_MULTIPLIER:
                    return True

    return False

def _check_bounce(
    self,
    det: DetectedObject,
    history: Deque[HistoryPoint],
    frame_idx: int,
) -> None:
    is_bounce, bounce_pt = self._detect_bounce(
        history, self.bounce_cooldown[det.track_id]
    )

    if not (is_bounce and bounce_pt is not None):
        return

    mid_frame = history[len(history) // 2][2]

    if self._is_ball_collision(det.track_id, bounce_pt, mid_frame, det.radius):
        print(f"[ЗІТКНЕННЯ] ID={det.track_id} (ігноруємо – ball-to-ball)")
    else:
        print(f"[СТІНА] ID={det.track_id} кадр={frame_idx}")
        self.bounce_points.append(bounce_pt)

    # Кулдаун у будь-якому випадку
    self.bounce_cooldown[det.track_id] = BOUNCE_COOLDOWN_FRAMES
```

ДОДАТОК Е

Модуль виявлення стін DBSCAN (analyzer.py)

```
from dataclasses import dataclass
from typing import List, Optional, Tuple
import cv2
import numpy as np
from sklearn.cluster import DBSCAN
from config import DBSCAN_EPS, DBSCAN_MIN_SAMPLES, MIN_BOUNCE_POINTS_FOR_WALLS


@dataclass
class Wall:
    """Виявлена стіна (пряма лінія), апроксимована з кластера точок відскоків."""
    pt1: Tuple[int, int]            # Початкова точка відрізка
    pt2: Tuple[int, int]            # Кінцева точка відрізка
    A: float                         # Коефіцієнт A рівняння Ax + By + C = 0
    B: float                         # Коефіцієнт B
    C: float                         # Коефіцієнт C
    cluster_points: np.ndarray       # Точки кластера

    @property
    def equation(self) -> str:
        return f"{self.A:.2f}x + {self.B:.2f}y + {self.C:.0f} = 0"


def detect_walls(bounce_points: List[Tuple[int, int]]) -> List[Wall]:

    if len(bounce_points) < MIN_BOUNCE_POINTS_FOR_WALLS:
        return []

    X = np.array(bounce_points)
    clustering = DBSCAN(eps=DBSCAN_EPS, min_samples=DBSCAN_MIN_SAMPLES).fit(X)
    labels = clustering.labels_

    walls: List[Wall] = []

    for label in set(labels):
        if label == -1:
            continue  # -1 – шум, поодинокі випадкові відскоки

        cluster_points = X[labels == label]
        if len(cluster_points) < 2:
            continue

        wall = _fit_wall_line(cluster_points)
        if wall is not None:
            walls.append(wall)

    return walls


def _fit_wall_line(cluster_points: np.ndarray) -> Optional[Wall]:
    [vx, vy, x0, y0] = cv2.fitLine(
```

```
    cluster_points, cv2.DIST_L2, 0, 0.01, 0.01
)
vx, vy = float(vx[0]), float(vy[0])
x0, y0 = float(x0[0]), float(y0[0])

if vx == 0 and vy == 0:
    return None

# Проекція точок на вектор напрямку для обмеження довжини
t_values = [
    (pt[0] - x0) * vx + (pt[1] - y0) * vy
    for pt in cluster_points
]
t_min, t_max = min(t_values), max(t_values)

pt1 = (int(x0 + t_min * vx), int(y0 + t_min * vy))
pt2 = (int(x0 + t_max * vx), int(y0 + t_max * vy))

# Рівняння прямої: Ax + By + C = 0
A = -vy
B = vx
C = vy * x0 - vx * y0

return Wall(
    pt1=pt1,
    pt2=pt2,
    A=A,
    B=B,
    C=C,
    cluster_points=cluster_points,
)
```

ДОДАТОК Ж

Модуль прогнозування зіткнень (predictor.py)

```
from dataclasses import dataclass
from typing import List, Optional, Tuple
import math
from analyzer import Wall
from config import PREDICTION_TIME_SEC

@dataclass
class CollisionPrediction:
    """Результат прогнозування зіткнення об’єкта зі стіною."""
    track_id: int
    collision_point: Tuple[int, int]     # Точка зіткнення (x, y)
    time_to_collision: float             # Час до зіткнення (секунди)
    wall_equation: str                   # Рівняння стіни для відображення

def predict_collisions(
    objects: List[Tuple[int, Tuple[float, float], Tuple[float, float], int]],
    walls: List[Wall],
    video_fps: float,
) -> List[CollisionPrediction]:
    if not walls or not objects:
        return []

    predictions: List[CollisionPrediction] = []

    for track_id, (cx, cy), (vx, vy), radius in objects:
        speed = math.hypot(vx, vy)
        if speed < 1e-6:
            continue  # Об’єкт нерухомий

        best = _find_nearest_wall_collision(
            cx, cy, vx, vy, radius, walls, video_fps
        )

        if best is not None:
            collision_pt, time_sec, wall_eq = best
            predictions.append(CollisionPrediction(
                track_id=track_id,
                collision_point=collision_pt,
                time_to_collision=time_sec,
                wall_equation=wall_eq,
            ))

    return predictions

def _find_nearest_wall_collision(
    cx: float,
    cy: float,
    vx: float,
    vy: float,
    radius: int,
    walls: List[Wall],
    video_fps: float,
```

```
) -> Optional[Tuple[Tuple[int, int], float, str]]:
    max_t_frames = video_fps * PREDICTION_TIME_SEC
    best_t = float("inf")
    best_result = None

    for wall in walls:
        # Промінь: P(t) = (cx + vx*t, cy + vy*t), t у кадрах
        # Стіна: A*x + B*y + C = 0
        denominator = wall.A * vx + wall.B * vy

        if abs(denominator) < 1e-9:
            continue  # Промінь паралельний стіні

        # Час (у кадрах) до перетину з нескінченною прямою
        t = -(wall.A * cx + wall.B * cy + wall.C) / denominator

        if t <= 0:
            continue  # Зіткнення позаду (об’єкт рухається від стіни)

        if t >= max_t_frames:
            continue  # Занадто далеко

        # Точка перетину
        hit_x = cx + vx * t
        hit_y = cy + vy * t

        if not _point_on_segment(hit_x, hit_y, wall, margin=radius):
            continue

        if t < best_t:
            best_t = t
            time_sec = t / video_fps
            best_result = (
                (int(hit_x), int(hit_y)),
                time_sec,
                wall.equation,
            )
    return best_result

def _point_on_segment(
    px: float,
    py: float,
    wall: Wall,
    margin: int = 0,
) -> bool:
    x1, y1 = wall.pt1
    x2, y2 = wall.pt2

    dx = x2 - x1
    dy = y2 - y1
    seg_len_sq = dx * dx + dy * dy

    if seg_len_sq < 1e-9:
        return False  # Вироджений відрізок

    t = ((px - x1) * dx + (py - y1) * dy) / seg_len_sq

    seg_len = math.sqrt(seg_len_sq)
    t_margin = margin / seg_len if seg_len > 0 else 0

    return -t_margin <= t <= 1.0 + t_margin
```

ДОДАТОК И

Модуль візуалізації (visualizer.py)

```
from typing import Dict, List, Optional, Tuple
import cv2
from analyzer import Wall
from config import (
    COLOR_ARROW,
    COLOR_BOUNCE,
    COLOR_CLUSTER_POINT,
    COLOR_COLLISION,
    COLOR_FRAME,
    COLOR_PREDICTION,
    COLOR_PROCESS_FPS,
    COLOR_VIDEO_FPS,
    COLOR_WALL,
    PREDICTION_TIME_SEC,
)
from predictor import CollisionPrediction


def draw_prediction_arrow(
    frame,
    center: Tuple[float, float],
    vx: float,
    vy: float,
    video_fps: float,
    radius: int,
) -> None:
    """Малює стрілку передбачення та цільове коло."""
    future_x = int(center[0] + vx * video_fps * PREDICTION_TIME_SEC)
    future_y = int(center[1] + vy * video_fps * PREDICTION_TIME_SEC)
    curr_pt = (int(center[0]), int(center[1]))
    future_pt = (future_x, future_y)

    cv2.arrowedLine(frame, curr_pt, future_pt, COLOR_ARROW, 2, tipLength=0.15)
    cv2.circle(frame, future_pt, radius, COLOR_PREDICTION, 2)


def draw_track_id(
    frame,
    center: Tuple[float, float],
    track_id: int,
) -> None:
    """Малює ID трекінгу над об’єктом."""
    pos = (int(center[0]) - 10, int(center[1]) - 20)
    cv2.putText(
        frame,
        f"ID:{track_id}",
        pos,
        cv2.FONT_HERSHEY_SIMPLEX,
        0.5,
        (255, 255, 255),
        2,
```

```
    )


def draw_bounce_points(
    frame,
    points: List[Tuple[int, int]],
) -> None:
    """Малює всі накопичені точки відскоків."""
    for pt in points:
        cv2.circle(frame, pt, 4, COLOR_BOUNCE, -1)


def draw_walls(
    frame,
    walls: List[Wall],
    stabilizer=None,
    H=None,
) -> None:
    for wall in walls:
        # Трансформуємо координати стін у екранний простір
        if stabilizer is not None and H is not None:
            cluster_screen = stabilizer.to_screen_space(
                [tuple(pt) for pt in wall.cluster_points], H
            )
            endpoints_screen = stabilizer.to_screen_space(
                [wall.pt1, wall.pt2], H
            )
            pt1_screen = endpoints_screen[0]
            pt2_screen = endpoints_screen[1]
        else:
            cluster_screen = [tuple(pt) for pt in wall.cluster_points]
            pt1_screen = wall.pt1
            pt2_screen = wall.pt2

        # Точки кластера
        for pt in cluster_screen:
            cv2.circle(frame, pt, 6, COLOR_CLUSTER_POINT, 2)

        # Лінія стіни
        cv2.line(frame, pt1_screen, pt2_screen, COLOR_WALL, 3)

def draw_collision_predictions(
    frame,
    predictions: List[CollisionPrediction],
    screen_coords: Dict[int, Tuple[float, float]],
    collision_screen_pts: Optional[List[Tuple[int, int]]] = None,
) -> None:
    for i, pred in enumerate(predictions):
        # Екранні координати об’єкта та точки зіткнення
        obj_pos = screen_coords.get(pred.track_id)
        if obj_pos is None:
            continue

        if collision_screen_pts is not None and i < len(collision_screen_pts):
            hit_pt = collision_screen_pts[i]
        else:
            hit_pt = pred.collision_point

        obj_pt = (int(obj_pos[0]), int(obj_pos[1]))

        # Пунктирна лінія від об’єкта до точки зіткнення
```

```
        _draw_dashed_line(frame, obj_pt, hit_pt, COLOR_COLLISION, 2, dash_len=10)

        # Хрестик на місці зіткнення
        size = 10
        cv2.line(frame, (hit_pt[0] - size, hit_pt[1] - size),
                 (hit_pt[0] + size, hit_pt[1] + size), COLOR_COLLISION, 3)
        cv2.line(frame, (hit_pt[0] + size, hit_pt[1] - size),
                 (hit_pt[0] - size, hit_pt[1] + size), COLOR_COLLISION, 3)

        # Час до зіткнення
        label = f"{pred.time_to_collision:.2f}s"
        cv2.putText(
            frame,
            label,
            (hit_pt[0] + 15, hit_pt[1] - 5),
            cv2.FONT_HERSHEY_SIMPLEX,
            0.5,
            COLOR_COLLISION,
            2,
        )


def _draw_dashed_line(
    frame,
    pt1: Tuple[int, int],
    pt2: Tuple[int, int],
    color: Tuple[int, int, int],
    thickness: int = 1,
    dash_len: int = 10,
) -> None:
    import math
    dx = pt2[0] - pt1[0]
    dy = pt2[1] - pt1[1]
    dist = math.hypot(dx, dy)

    if dist < 1:
        return

    num_dashes = int(dist / dash_len)
    if num_dashes < 1:
        num_dashes = 1

    for i in range(0, num_dashes, 2):
        t1 = i / num_dashes
        t2 = min((i + 1) / num_dashes, 1.0)
        start = (int(pt1[0] + dx * t1), int(pt1[1] + dy * t1))
        end = (int(pt1[0] + dx * t2), int(pt1[1] + dy * t2))
        cv2.line(frame, start, end, color, thickness)
```

ДОДАТОК К

Модуль вводу\виводу відео (video-io.py)

```
from dataclasses import dataclass
import cv2


@dataclass
class VideoInfo:
    fps: float
    width: int
    height: int


def open_video(path: str) -> tuple:
    cap = cv2.VideoCapture(path)
    if not cap.isOpened():
        raise RuntimeError(f"Не вдалося відкрити відео: {path}")

    info = VideoInfo(
        fps=cap.get(cv2.CAP_PROP_FPS),
        width=int(cap.get(cv2.CAP_PROP_FRAME_WIDTH)),
        height=int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT)),
    )
    return cap, info


def create_writer(
    path: str,
    fps: float,
    width: int,
    height: int,
) -> cv2.VideoWriter:
    """Створює VideoWriter для запису результату."""
    fourcc = cv2.VideoWriter_fourcc(*"mp4v")
    return cv2.VideoWriter(path, fourcc, fps, (width, height))
```