НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

До захисту допущено:
В. о. завідувача кафедри
_____ Михайло НОВОТАРСЬКИЙ
(підпис)

“__” _____2025 р.

Дипломний проєкт

на здобуття ступеня бакалавра

за освітньо-професійною програмою “Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

на тему: Система слідкування за рухомим об’єктом

Виконав : студент __4__ курсу, групи ІМ-11
(шифр групи)

_____Чирков Максим Костянтинович_____ _____
(прізвище, ім’я, по батькові) (підпис)

Керівник _____професор, д.т.н. Сергієнко А. М._____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант (нормоконтроль) ас. Пономаренко Артем Миколайович_____
(назва розділу) (посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент __асистент каф. СП і СКС, д.філ. Олексій Молчанов_____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

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

Київ – 2025 р.

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

Рівень вищої освіти – перший (бакалавр)

Освітньо-професійна програма

“Інженерія програмного забезпечення комп’ютерних систем”

спеціальності 121 “Інженерія програмного забезпечення”

ЗАТВЕРДЖУЮ
В. о. завідувача кафедри
Михайло НОВОТАРСЬКИЙ
(підпис)
“__” _____2025 р.

ЗАВДАННЯ
на бакалаврський дипломний проєкт студента

Чиркова Максима Костянтиновича

1. Тема проєкту Система слідкування за рухомим об’єктом
керівник проєкту Сергієнко Анатолій Михайлович, професор, д.т.н.,
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від 23 травня 2025 року №1705-с

2. Термін здачі студентом закінченого проєкту *6 червня 2025 р.*

3. Вихідні дані до проєкту *технічна документація, теоретичні дані.*

4. Зміст розрахунково-пояснювальної записки (перелік питань, які розробляються)

Розділ 1. Огляд існуючих рішень.

Розділ 2. Огляд технологій для розробки системи.

Розділ 3. Деталі розробки системи.

Розділ 4. Дослідження та аналіз розробленої системи.

5. Перелік графічного матеріалу (з точним позначенням обов'язкових креслень) структурна схема системи, функціональна схема (діаграма класів), алгоритм дій програмного забезпечення.

6. Консультанта проєкту, з вказівкою розділів проєкту, які до них вносяться

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

7. Дата видачі завдання 8 січня 2025 р.

Календарний план

№ п/п	Найменування етапів дипломного проєкту	Терміни виконання етапів проєкту	Примітки
1.	*Затвердження теми проєкту*	*10.12.2024-15.12.2024*	
2.	*Вивчення та аналіз завдання*	*15.12.2024-15.03.2025*	
3.	*Розробка архітектури та загальної структури системи*	*15.03.2025-25.03.2025*	
4.	*Розробка структур окремих підсистем*	*25.03.2025-5.04.2025*	
5.	*Програмна реалізація системи*	*5.04.2025-15.04.2025*	
6.	*Оформлення пояснювальної записки*	*15.04.2025-20.05.2025*	
7.	*Захист програмного продукту*	*18.05.2025*	
8.	*Передзахист*	*03.06.2025*	
9.	*Захист*	*17.06.2025*	

Студент-дипломник _____ Максим ЧИРКОВ
(підпис)

Керівник проєкту _____ Анатолій СЕРГІЄНКО
(підпис)

АНОТАЦІЯ

У даній роботі було досліджено методи слідкування за рухомими об’єктами у відеопотоці та проаналізовано їхні сильні та слабкі сторони. На основі проведеного аналізу було розроблено програмний застосунок на мові програмування C++, у якому реалізовано власну версію алгоритму SURF для виявлення та відстеження об'єктів. Слідкування інваріантне до переміщення, повороту та зміни масштабу і враховує вимоги до точності та швидкості. Було проведено тестування створеної системи та виконано базову оптимізацію для підвищення її продуктивності.

Ключові слова: комп’ютерний зір, слідкування за об’єктом, SURF, C++, відеопотік.

ANNOTATION

This project explores methods of tracking moving objects in a video stream and analyzes their strengths and weaknesses. Based on the conducted analysis, a software application was developed in C++, featuring a custom implementation of the SURF algorithm for object detection and tracking. The tracking is invariant to transition, rotation and scale changes, and takes into account the requirements for accuracy and speed. The developed system was tested, and basic optimization was performed to improve its performance.

Key words: computer vision, object tracking, SURF, C++, video stream.

справки	Формат	*Значення*	*Найменування*	Кіл. листів	№ екземпля	*Додаток*
			Документація загальна			
			Знову розроблена			
	А4	***ІАЛЦ.467200.002 ТЗ***	Система слідкування за	4		
			рухомим об’єктом			
			Технічне завдання			
	А4	***ІАЛЦ.467200.003 ПЗ***	Система слідкування за	78		
			рухомим об’єктом			
			Пояснювальна записка			
	А4	***ІАЛЦ.467200.004 Д1***	Система слідкування за	1		
			рухомим об’єктом			
			Структурна схема			
			системи			
	А4	***ІАЛЦ.4672008.005 Д2***	Система слідкування за	1		
			рухомим об’єктом			
			Функціональна схема			
			(діаграма класів)			
	А4	***ІАЛЦ.467200.006 Д3***	Система слідкування за	1		
			рухомим об’єктом			
			Алгоритм дій програмного			
			забезпечення			
	А4	***ІАЛЦ.467200.007 Д4***	Система слідкування за	21		
			рухомим об’єктом			
			Текст програмного коду			

					ІАЛЦ.467200.001 ОА	
Зм	*Лист*	*№ докум.*	*Підп*	*Дата*		
Розроб		Чирков М. К.			*Система слідкування за рухомим об’єктом* Опис альбому	Літ. / Аркуш 1 / Аркушів 1
Перев		Сергієнко А. М.				КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-11

ТЕХНІЧНЕ ЗАВДАННЯ
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система слідкування за рухомим об'єктом*»

Київ – 2025

ЗМІСТ

					ІАЛЦ.467200.002 ТЗ		
		№ докум.	Підпис	Дата			
Розробив		Чирков М. К.			**Система слідкування за рухомим об'єктом** Технічне завдання	Літ.	Аркуш 1 / Аркушів 4
Перевірив		Сергієнко А. М.					
						КПІ ім. Ігоря Сікорського, ФІОТ, ІМ-11	
Н. Контр.		Пономаренко А. М.					
Затвердив							

1 НАЙМЕНУВАННЯ ТА ОБЛАСТЬ ЗАСТОСУВАННЯ

Дане технічне завдання спрямоване на розробку системи, що виконує виявлення та слідкування за об’єктом у відеопотоці в реальному часі з використанням власної реалізації алгоритму SURF.

Сфера застосування розробки охоплює задачі комп’ютерного зору, що потребують швидкої та стабільної роботи на пристроях з обмеженими обчислювальними можливостями. Система може бути інтегрована в робототехнічні платформи для навігації та взаємодії з рухомими об’єктами, у системи відеоспостереження та автоматизованого моніторингу, в безпілотні апарати, інтелектуальні транспортні системи для аналізу дорожньої ситуації, а також у навчальні та дослідницькі проєкти.

2 ПІДСТАВИ ДЛЯ РОЗРОБКИ

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

3 МЕТА ТА ПРИЗНАЧЕННЯ РОЗРОБКИ

Метою та призначенням даної роботи є створення системи для виявлення та слідкування за рухомим об’єктом у відеопотоці в реальному часі з високою точністю і продуктивністю.

4 ДЖЕРЕЛА РОЗРОБКИ

Джерелом розробки даного дипломного проекту є офіційні документації, публікації та статті в мережі Інтернет на дану тему, науково-технічна література.

5 ТЕХНІЧНІ ВИМОГИ

5.1. Вимоги до розробленого продукту

Розроблена система має виконувати такі вимоги:

- Баланс між швидкодією та точністю.
- Надати можливість користувачам передавати початкову точку, що є центром об’єкта.
- Надати можливість користувачам можливість власноруч конфігурувати специфічні параметри та налаштування алгоритму.
- Надати вичерпну та зрозумілу документацію для розробленого продукту.

5.2. Вимоги до програмного забезпечення

- ОС Windows, Mac чи Linux.
- Довільний вибір мови програмування (бажано C++).
- Формат вихідних даних:
    - множина дескрипторів знайдених характерних точок у заданому околі;
    - кадри з відміченими точками та обчисленим центром об’єкта (центр мас, де масою є інтенсивність характерних точок);

5.3. Вимоги до апаратної частини

- Raspberry Pi 5 або 4 із встановленою операційною системою на основі Linux.
- RAM не менше ніж 4 ГБ.

6 ЕТАПИ РОЗРОБКИ

Назва етапів виконання	Термін виконання
Затвердження теми роботи	*10.12.2020-15.12.2020*
Вивчення та аналіз завдання	*15.12.2020-15.03.2021*
Розробка архітектури та загальної структури системи	*15.03.2021-25.03.2021*
Розробка структур окремих частин системи	*25.03.2021-5.04.2021*
Програмна реалізація системи	*5.04.2021-10.04.2021*
Виправлення помилок	*10.04.2021-15.04.2021*
Оформлення пояснювальної записки	*15.04.2021-20.05.2021*

ПОЯСНЮВАЛЬНА ЗАПИСКА
ДО ДИПЛОМНОГО ПРОЄКТУ

на тему: «*Система слідкування за рухомим об'єктом*»

Київ – 2025

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

SURF	(Speeded-Up Robust Features) Прискорені стійкі ознаки
SIFT	(Scale-Invariant Feature Transform) Алгоритм на основі методу масштабно-незалежного перетворення ознак
BRIEF	(Binary Robust Independent Elementary Features) Двійкові стійкі незалежні елементарні ознаки
FPS	(Frames Per Second) кадрів на секунду
ORB	(Oriented FAST and Rotated BRIEF) Орієнтований FAST та обернений BRIEF
YOLO	(You Only Look Once) Одноетапний алгоритм виявлення об’єктів
SSD	(Single Shot MultiBox Detector) Одноетапний детектор із множиною вікон
CNN	(Convolutional Neural Network) Згорткова нейронна мережа
GOTURN	(Generic Object Tracking Using Regression Networks) Універсальне відстеження об’єктів за допомогою регресійних мереж
GPU	(Graphics Processing Unit) Графічний процесор
KLT	(Kanade–Lucas–Tomasi) Метод відстеження характерних ознак Канаде—Лукаса—Томасі
ROI	(Region of Interest) Область інтересу
IDE	(Integrated Development Environment) Інтегроване середовище розробки

ВСТУП

У сучасному світі системи комп’ютерного зору відіграють важливу роль у багатьох сферах — від промислової автоматизації до побутових пристроїв. Одним із ключових напрямів у цій галузі є розробка рішень для виявлення та слідкування за об’єктами в реальному часі. Такі системи знаходять широке застосування у відеоспостереженні, автономній навігації, транспортних системах, робототехніці, безпілотних апаратах, а також у наукових та навчальних цілях.

Слідкування за рухомим об’єктом є складним завданням, що вимагає точного аналізу зображення, стійкості до змін освітлення, масштабу, поворотів та часткових перекриттів об’єкта. Особливо актуальним є забезпечення ефективної роботи таких систем в умовах обмежених обчислювальних ресурсів, де використання важких бібліотек або глибоких нейронних мереж є недоцільним або неможливим.

Метою даної роботи є створення програмного застосунку для слідкування за рухомим об’єктом у відеопотоці з використанням власної реалізації алгоритму SURF. У процесі розробки було проаналізовано існуючі методи, обрано оптимальні підходи, реалізовано та протестовано систему, що демонструє стійку роботу в режимі реального часу. Робота охоплює етапи дослідження, розробки, оптимізації та аналізу ефективності створеного рішення.

РОЗДІЛ 1 СИСТЕМИ СЛІДКУВАННЯ ЗА ОБ'ЄКТАМИ

1.1 Задача слідкування за об'єктами у відеопотоці

1.1.1 Формулювання задачі

Слідкування за об'єктами (object tracking) є однією з ключових задач комп'ютерного зору, яка полягає у визначенні положення об'єкта на послідовності кадрів відеопотоку після його початкового виявлення [1]. Основною метою є збереження ідентичності об'єкта впродовж часу, незважаючи на можливі зміни його положення, масштабу, орієнтації, форми чи умов освітлення [1, 2].

Формально задача може бути сформульована так: маючи відеопотік, що складається з послідовності кадрів I_t, та область об'єкта, задану на початковому кадрі (наприклад, у вигляді прямокутного обмежувального вікна), необхідно автоматично визначати положення об'єкта на кожному наступному кадрі. При цьому важливо забезпечити стійкість до перешкод, таких як шум, перекриття об'єкта іншими об'єктами, незначні деформації, а також втрати фокуса [1].

1.1.2 Вимоги до систем відстеження

Системи відстеження об'єктів у відеопотоці повинні відповідати низці вимог, які визначають їхню практичну цінність у реальних умовах [1, 6]. Ці вимоги можуть варіюватися в залежності від сфери застосування, однак основні з них залишаються незмінними.

Точність локалізації. Трекер повинен точно визначати положення об'єкта на кожному кадрі. Висока точність є критичною для задач, що передбачають

подальшу взаємодію з об’єктом, наприклад, у робототехніці чи системах автоматичного керування.

Стійкість до змін зовнішніх умов. Система має бути інваріантною до змін масштабу, поворотів, освітлення, часткових перекриттів, шуму та деформацій. Це забезпечує можливість її застосування в динамічних середовищах із непередбачуваними умовами зйомки.

Робота в реальному часі. У багатьох практичних задачах критично важливою є здатність системи працювати з мінімальною затримкою. Це вимагає ефективного використання обчислювальних ресурсів, особливо в системах на базі вбудованих пристроїв або мікрокомп’ютерів.

Стабільність та безперервність трекінгу. Система має підтримувати ідентичність об’єкта протягом часу без частих “втрат” або переключень на інші об’єкти. Це важливо, зокрема, при відстеженні в натовпі або в присутності схожих об’єктів.

Автоматичність та мінімальна участь користувача. Сучасні системи мають функціонувати автономно, з мінімальною або повністю відсутньою необхідністю ручного втручання під час відстеження.

Масштабованість та узагальнюваність. Система має бути здатною адаптуватися до нових сцен, об’єктів та умов без потреби в перенавчанні або модифікаціях алгоритму.

Система відстеження повинна працювати в реальному часі або близькому до реального часу режимі, що накладає обмеження на обчислювальну складність використовуваних алгоритмів, особливо у випадках застосування на пристроях з обмеженими ресурсами [3].

За складністю задача відстеження розглядається як проміжна між задачами детекції та розпізнавання об’єктів. Якщо детектор спрацьовує незалежно на кожному кадрі, то трекер підтримує безперервність ідентифікації одного й того

ж об’єкта протягом серії кадрів, що робить її важливою частиною систем відеоаналітики, автономної навігації, відеоспостереження тощо.

1.1.3 Особливості вирішення задачі стеження

Серед типових викликів, що виникають в задачах стеження за об’єктом можна назвати наступні:

1. Зміни масштабу та орієнтації об’єкта.

Об’єкт може наближатися до камери або віддалятися, змінюючи свій розмір на зображенні. Також він може обертатися або нахилятися, що вимагає інваріантності алгоритму до геометричних трансформацій [1, 2].

2. Часткові або повні перекриття.

Часто трапляється ситуація, коли об’єкт тимчасово перекривається іншими об’єктами в сцені. Це може спричинити втрату трекінгу або помилкову ідентифікацію іншого об’єкта.

3. Варіації освітлення.

Зміна умов освітлення, відблиски або тіні впливають на зовнішній вигляд об’єкта, що ускладнює його виявлення. Деякі алгоритми чутливі до навіть незначних змін яскравості чи контрасту.

4. Швидкий або нерівномірний рух.

Якщо об’єкт рухається занадто швидко або змінює траєкторію руху, трекер може не встигати коректно оновлювати його положення. Також можлива поява розмиття в кадрі, що ускладнює аналіз.

5. Фонові завади та схожі об’єкти.

У складних сценах, де багато елементів або об’єкти мають схожий вигляд, трекеру складно зберегти унікальність обраного об’єкта. Це може призводити до помилок ідентифікації.

6. Нестача обчислювальних ресурсів.

У системах, що працюють на пристроях з обмеженими обчислювальними можливостями (наприклад, вбудовані платформи), постає проблема оптимізації обраного методу для забезпечення реального часу.

Кожен з цих викликів відіграє ключову роль у сучасних алгоритмах, створюючи додаткові умови, що повинні бути враховані на етапі розробки. Більшість алгоритмів вирішують кілька з перелічених потенційних проблем, але не всі в однаковій мірі , що створює додаткове поле для досліджень відповідності між сферою застосування та конкретним способом слідкувати за об’єктом.

1.2 Підходи для реалізації систем слідкування

Підходи до реалізації систем слідкування за об’єктами умовно поділяються на класичні, що базуються на методах обробки зображень, та ті, які використовують методи машинного і глибокого навчання, причому кожен з них має свої переваги та обмеження залежно від умов застосування.

1.2.1 Класичні алгоритми відстеження

Класичні алгоритми відстеження об'єктів базуються переважно на методах обробки зображень і не потребують попереднього навчання. Їхньою основною перевагою є швидкодія та можливість реалізації на пристроях з обмеженими ресурсами. Такі підходи зазвичай працюють з описовими ознаками об’єкта або його геометричними характеристиками.

1. Алгоритми, засновані на визначенні ознак (feature-based methods).

До цього типу належать методи, які відстежують ключові точки або дескриптори в послідовності кадрів. Найвідомішими є алгоритми SURF (Speeded-Up Robust Features), SIFT (Scale-Invariant Feature Transform), BRIEF, ORB [2, 6, 10]. Вони дозволяють відстежувати об’єкти, стійкі до змін масштабу, повороту та часткового перекриття.

2. Алгоритми, що базуються на шаблонах (template matching).

У таких методах відстеження використовується фіксований зразок або шаблон об'єкта, який порівнюється з частинами нового кадру. Одним з популярних представників цього підходу є метод нормалізованої крос-кореляції. Основним недоліком шаблонного підходу є його чутливість до змін зовнішнього вигляду об'єкта.

3. Методи на основі оптичного потоку (optical flow).

Алгоритми оптичного потоку, наприклад, метод Лукаса-Канаде (Lucas-Kanade), дозволяють оцінити рух об’єкта між кадрами, відстежуючи зміну інтенсивності пікселів. Такі підходи добре працюють для плавного руху, але можуть бути нестійкими при різких змінах або шумах [7].

4. Методи фону та різниці кадрів.

У цих підходах використовується аналіз змін у відеопотоці, щоб виявити об’єкти, які переміщуються на фоні. Вони є простими у реалізації, але зазвичай неефективні в складних сценах або при зміні освітлення.

Класичні методи актуальні в системах, що потребують швидкості, простоти реалізації та працюють на обмежених ресурсах. Проте вони іноді поступаються сучасним методам за точністю та стійкістю до складних умов.

Однак для випадків з обмеженими ресурсами класичні алгоритми, засновані на визначені ознак, ідеально підходять, адже витримують баланс між швидкодією, не потребують багато обчислювальних ресурсів та витримують точність спостереження на високому рівні.

1.2.2 Методи на основі глибокого навчання

Методи глибокого навчання стали основою для багатьох сучасних систем відстеження об'єктів завдяки високій точності, адаптивності до складних умов та здатності узагальнювати інформацію з відеопотоку. Ці підходи здатні автоматично виділяти релевантні ознаки, що дозволяє досягати значних результатів у відстеженні об’єктів, навіть у разі часткового перекриття, зміни масштабу чи освітлення.

1. Відстеження за допомогою нейронних мереж.

Багато систем комбінують детекцію та трекінг, наприклад, за допомогою таких моделей як YOLO, SSD або Faster R-CNN. Вони виконують обчислення для кожного кадру окремо, а далі поєднують інформацію для формування стабільної траєкторії об'єкта.

2. CNN і сіамські мережі.

Моделі, як-от GOTURN або SiamFC, використовують згорткові мережі для порівняння зовнішнього вигляду об'єкта між кадрами. Це забезпечує швидке реагування на зміни сцени без необхідності повторної детекції на кожному кроці [4].

3. Моделі з увагою та трансформерами.

Сучасні підходи, зокрема TrackFormer або TransTrack, застосовують механізм attention для відстеження зв’язків між об'єктами у часі, що особливо ефективно в складних умовах, як-от багатолюдні сцени або динамічне тло.

Недоліки методів на основі глибокого навчання:

- Висока обчислювальна складність — для ефективної роботи більшості моделей потрібні потужні GPU або спеціалізовані апаратні рішення.
- Необхідність великих обсягів розмічених даних для навчання.

- Висока енергоспоживаність, що ускладнює використання на енергонезалежних або вбудованих пристроях.
- Погана переносимість без донавчання — моделі часто втрачають ефективність при зміні умов середовища або категорій об'єктів.
- Ускладнене налагодження та відсутність інтерпретованості результатів — важко зрозуміти, чому модель прийняла певне рішення, особливо у випадках помилок.
- Великий обсяг коду та залежностей, що ускладнює інтеграцію в невеликі застосунки або критичні системи з жорсткими обмеженнями.

Попри всі недоліки даний метод трекінгу користується популярністю і є центром багатьох сучасних академічних досліджень.

1.2.3 Гібридні системи

Гібридні системи поєднують класичні методи відстеження з підходами глибокого навчання, що дозволяє досягти кращого балансу між точністю та швидкодією. Однак така інтеграція ускладнює реалізацію, вимагає більше ресурсів та ретельного налаштування компонентів. У цій роботі було вирішено зосередитись на легшій та більш контрольованій власній реалізації класичного методу, що краще підходить для роботи на обмежених пристроях.

1.3 Порівняння популярних алгоритмів і технологій

Для побудови ефективної системи відстеження об'єктів у відеопотоці важливо розуміти особливості основних алгоритмів, які використовуються в цій галузі. У цьому підрозділі розглянуто найпоширеніші методи виявлення та

трекінгу об’єктів, що застосовуються в системах комп’ютерного зору, а також проаналізовано їх ключові характеристики, переваги та обмеження.

Таблиця 1.1 – Переваги та недоліки класичних алгоритмів відстеження

Алгоритм	Переваги	Недоліки
SIFT	1. Висока точність і стійкість до змін масштабу. 2. Добре працює з афінними перетвореннями. 3. Повністю відкритий та вільний для використання.	1. Порівняно повільний, особливо на обмежених обчислювальних пристроях. 2. Вимагає багато ресурсів пам’яті.
SURF	1. Значно швидший за SIFT. 2. Стійкий до змін масштабу, освітлення та обертання 3. Добре підходить для реального часу та вбудованих систем. 4. Має просту та ефективну реалізацію.	1. Патентований, тому може бути використаних лише в дослідницьких некомерційних цілях. 2. На деяких сценах точність поступається SIFT. 3. Повільніше за ORB на деяких пристроях без апаратного прискорення.
ORB	1. Дуже швидкий і легкий 2. Підходить для пристроїв з обмеженими ресурсами. 3. Повністю відкритий та вільний для використання.	1. Менш точний, особливо при зміні масштабу та освітлення. 2. Не такий стійкий як SIFT та SURF у складних умовах.
KLT	1. Підходить для трекінгу в послідовностях зображень (відео). 2. Ефективний при невеликих зсувах між кадрами. 3. Порівняно швидкий як для того, що не є дескриптором ознак.	1. Не стійкий до масштабування чи великих обертань. 2. Потребує гарної ініціалізації початкових ключових точок.

Як видно з таблиці 1.1, кожен з алгоритмів має свої переваги та недоліки. Серед розглянутих алгоритмів саме SURF виявився найкращим вибором для реалізації системи на Raspberry Pi, оскільки поєднує високу точність виявлення ключових точок зі швидкодією, що є критично важливим для пристроїв із

обмеженими обчислювальними ресурсами. На відміну від SIFT, він є значно швидшим, а у порівнянні з ORB — забезпечує вищу стійкість до змін масштабу та освітлення.

У межах системи недоліки даного алгоритму були усунуті завдяки ретельно підібраним коефіцієнтам для збільшення точності та обмеженням області обробки для збільшення швидкодії.

ВИСНОВОК ДО РОЗДІЛУ 1

У першому розділі було здійснено огляд існуючих підходів до реалізації систем слідкування за об’єктами у відеопотоці. На початку було сформульовано саму задачу відстеження, а також визначено ключові вимоги до таких систем: точність, стійкість до змін зовнішніх умов (шум, зміна масштабу, часткове перекриття, варіації освітлення), а також здатність працювати в реальному часі.

Далі були розглянуті основні класи підходів до відстеження: класичні методи, які базуються на аналізі ознак, шаблонів або оптичного потоку; сучасні алгоритми, побудовані на основі глибоких нейронних мереж; а також гібридні системи, які поєднують переваги обох підходів для досягнення кращої узагальненості та швидкодії.

Особливу увагу було приділено порівнянню популярних алгоритмів виявлення ознак — SIFT, SURF, ORB, KLT. Їх було оцінено за критеріями точності, стійкості до різних трансформацій та обчислювальної ефективності. На основі аналізу було зроблено висновок, що алгоритм SURF є найбільш збалансованим за співвідношенням швидкодії та надійності, що й обумовило його вибір як бази для реалізації власного детектора ключових точок у наступних розділах.

Таким чином, цей розділ сформував теоретичну основу для подальшої розробки системи, обґрунтував вибір конкретного підходу та визначив напрямок реалізації на основі перевірених і ефективних методів.

РОЗДІЛ 2 АЛГОРИТМ SURF І ЙОГО РЕАЛІЗАЦІЯ

2.1 Опис алгоритму SURF

2.1.1 Основні положення

Система складається з двох основних частин: виявлення об’єкта та подальшого спостереження за ним. Основою для цих процесів є алгоритм виявлення характерних точок SURF [2], який є прискореною модифікацією відомого методу SIFT.

На відміну від інших методів, SURF демонструє високу точність та надійність при виявленні й ідентифікації ключових точок у зображеннях, що піддаються масштабуванню, обертанню або мають неоднорідну яскравість.

Ідея SURF полягає в формуванні стосу зображень $H(x, y, \sigma)$ з відгуків детектору Гессе. Ці відгуки виражають міру неоднорідності зображення при різних значеннях σ, що є параметром фільтру Гауса і визначає масштаб аналізу. Головною метою є знайти піксель (x, y, σ) у цьому стосі з максимальною амплітудою, що потенційно є характерною точкою, і проаналізувати її окіл.

Основною перевагою SURF є використання прямокутних фільтрів, які апроксимують другі похідні кривої Гауса від різних осей зображення при заданому коефіцієнті розмиття σ, та можуть бути обчислені з використанням інтегрального зображення. Це дозволяє виконувати згортку не через обчислення зваженої суми пікселів (як у класичних фільтрах), а за допомогою кількох операцій додавання і віднімання, що значно прискорює обробку.

Розрахунок прямокутних фільтрів значно спрощується і пришвидшується завдяки створенню інтегрального зображення $I(x, y)$. Це дозволяє всього завдяки чотирьом операціям отримати суму пікселів в будь-якому

прямокутнику незалежно від його розмірів. Дана апроксимація завдяки прямокутним фільтрам і інтегральному зображенню – це основна відмінність від алгоритму SIFT, що значно пришвидшує виконування обчислень.

Після виявлення характерної точки для неї формується вектор-дескриптор, який складається з величини $H(x, y, \sigma)$, її координат і характеристики неоднорідності її околу радіусом 10σ пікселів. Характеристика неоднорідності будується на основі вектора з 64 значень, що складаються з локальних градієнтів яскравості по вертикалі і горизонталі та їх відповідних абсолютних значень в кожній з 16 областей, на які ділиться окіл точки розміром $20\sigma \times 20\sigma$. Такий вектор добре описує структуру зображення в околі точки і зберігає інформацію про міру і напрям зміни яскравості .

На етапі виявлення об’єкта формуються так звані еталонні точки. Згодом на етапі слідкування вимірюється евклідова відстань між дескрипторами даної точки і еталонної точки. Згодом при зміні зображення еталонні точки можуть змінюватись для забезпечення слідкування при еволюції об’єкта та його оточення.

Знайдений об’єкт характеризується центром мас, що вираховується на основі інтенсивності найвизначніших його характерних точок.

2.1.2 Побудова інтегрального зображення

Інтегральне зображення – це допоміжне представлення зображення, в якому кожне значення сітки з координатами (x, y) - це сума значень прямокутної області, що формується початком координат і координатами поточного значення. Нехай зображення вхідного кадру це $X(x, y)$. Тоді інтегральне зображення:

$$I(x,y) = \sum_{i=0}^{x}\sum_{j=0}^{y} X(x,y) \quad (1)$$

Маючи сітку з такими значеннями, обрахунок суми прямокутної області з вершинами $a, b, c\ d$ зводиться до:

$$S = I(d) - I(c) - I(b) + I(a) \quad (2)$$

Завдяки цьому обчислення суми значень в будь-якій прямокутній області зводиться до операції зі складністю *O(1)* (замість *O(n)*). На рис.2.1 зображена випадково згенерована ілюстрація та її інтегральне зображення.

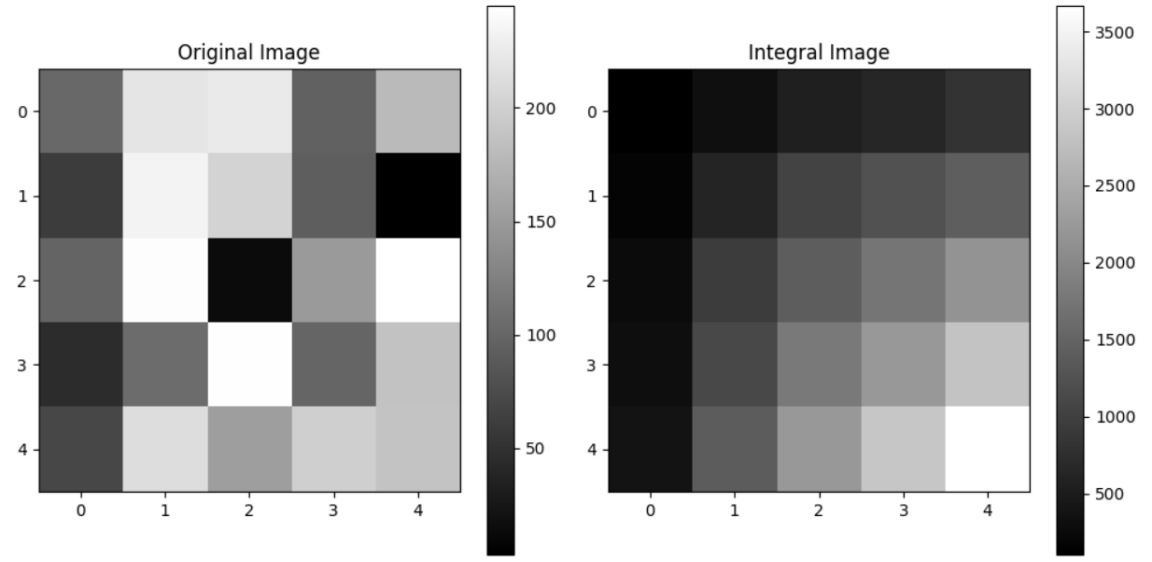


Рисунок 2.1 – Приклад побудови інтегрального зображення

2.1.3 Розрахунок значень параметру масштабу

У методі SURF аналіз зображення відбувається не лише в площині, а й у масштабному просторі задля забезпечення покриття різних рівнів деталізації виявлених точок. Кожному рівню масштабу відповідає значення σ, яке визначає ступінь згладжування. Метод SURF передбачає базовий масштаб $\sigma_0 = 1.2$, що відповідає найменшому фільтру. З передбачуваної поведінки системи можна зробити висновок, що еталонні точки потрібно оновлювати при збільшенні

масштабу не більше ніж вдвічі. Від розмірів фільтру залежить ступінь розмиття зображення – масштаб зображення, а фільтр прямопропорційно залежить від величини σ. Тому при збільшенні σ у 2 рази масштаб пошуку точки збільшується також удвічі. Для коректної апроксимації методу SURF використовується коефіцієнт 1.54. Отже, максимальне значення σ:

$$\sigma_{max} = 1.2 \times 2 \times 1.54 \approx 3.7$$

Таким чином максимальне ефективне розмиття, що дозволяє ефективно порівнювати дескриптори становить приблизно 3.7. Якщо значення розмиття більше, то дескриптори повинні оновлюватись. Щоб рівномірно покрити діапазон масштабів він 1.2 до ≈3.7 було використано крок $\Delta\sigma = 0.8$.

Отже, значення σ, що були обрані для системи:

$$\sigma = 1.2, 2.0, 2.8, 3.6$$

Відповідають чотирьом шарам, що покривають потрібний діапазон масштабів.

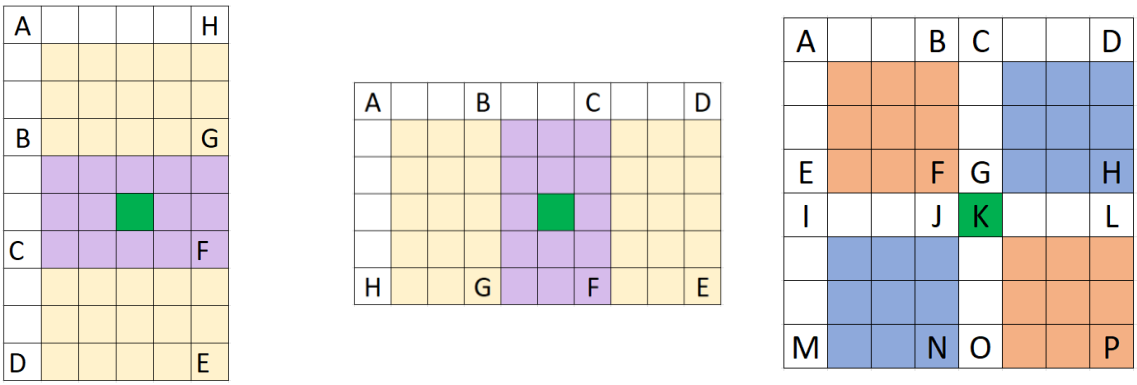
2.1.4 Виявлення характерних точок через відгук детектора Гессе

На етапі виявлення характерних точок алгоритм SURF використовує відгук детектора Гессе, який зазвичай є детермінантом матриці, що складається зі згорток з другим порядком похідних від фільтра Гауса у відповідних напрямах [2].

$$H(x, y, \sigma) = \begin{bmatrix} L_{xx}(x, y, \sigma) & L_{xy}(x, y, \sigma) \\ L_{xy}(x, y, \sigma) & L_{yy}(x, y, \sigma) \end{bmatrix} \quad (3)$$

Однак, в алгоритмі SURF згортка реалізується не класичними похідними Гауса, а їхніми апроксимаціями за допомогою прямокутних фільтрів і інтегрального зображення. Це допомагає досягнути значного прискорення.

На рис. 2.2 зображені апертури фільтрів для $\sigma = 1.2$, що покриваються зафарбованими прямокутниками в областях, над якими ми проводимо операції. В даному випадку D_{xx} – це фільтр неоднорідності по горизонталі, D_{yy} – по вертикалі, а D_{xy} по діагоналі. При чому значення пікселів у жовтих прямокутниках додаються, а в пурпурних – віднімаються подвоєними з загальної суми для $D_{xx}(x,y)$ і $D_{yy}(x,y)$, а в $D_{xy}(x,y)$ – віднімаються без подвоєння. Зелений квадрат позначає піксель з координатами (x,y) для заданого σ.



а) б) в)

Рисунок 2.2 – Ядра фільтрів D_{yy}(А), D_{xx}(Б), D_{xy}(В)

Завдяки раніше підготовленому інтегральному зображенню $I(x,y)$, розрахунок сум зводиться до:

$$D_{xx}(x,y) = I(G) - I(H) - I(B) + I(A) + I(E) - I(F) - I(D) + I(C) - 2(I(F) - I(G) - I(C) + I(B)) \quad (4)$$

$$D_{yy}(x,y) = I(G) - I(B) - I(H) + I(A) + I(E) - I(D) - I(F) + I(C) - 2(I(F) - I(C) - I(G) + I(B)) \quad (5)$$

$$D_{xy}(x,y) = I(F) - I(E) - I(B) + I(A) + I(P) - I(O) - I(L) + I(K) - (I(N) - I(M) - I(J) + I(I)) - (I(H) - I(G) - I(D) + I(C)) \quad (6)$$

Оскільки замість зменшення розміру зображення, як у класичних пірамідах, у SURF збільшується розмір фільтра, що дозволяє ефективно аналізувати зображення на різних масштабах. Тобто для різних σ використовуються різні розміри фільтру.

- $\sigma = 1.2$ – фільтр 9×9
- $\sigma = 2.0$ – фільтр 15×15
- $\sigma = 2.8$ – фільтр 21×21
- $\sigma = 3.6$ – фільтр 27×27

Тобто можна вирахувати розмір фільтру як $\sigma \times 7.5$. Для обчислення координат A, B,…, P до координат поточного пікселя додається $(\Delta x, \Delta y)$, що можуть бути вирахувані відповідно до розміру фільтра.

Матриця з відгуків фільтрів наближається до справжньої матриці Гессе [2]. Детермінант розраховується для кожного пікселя і значення σ як

$$H(x, y, \sigma) = \frac{D_{xx}D_{yy} - \omega^2 {D_{xy}}^2}{S}, \quad (7)$$

де D_{xx}, D_{yy}, D_{xy} – фільтри неоднорідності по горизонталі, вертикалі та діагоналі відповідно;

$\omega^2 \approx 0.9$;

S – площа квадрата апертури, для $\sigma = 1.2$ $S = 5^2 = 25$. Для інших значень розмиття ця площа складає 81, 169 та 289 відповідно.

Для виявлення об’єкту і подальшого стеження на наступних кадрах потрібно знайти достатню для цього кількість характерних точок. Для цього з усього кадру можна виокремити фрейм, який буде оброблятись. Аналіз подібної системи показав, що найчастіше об’єкт знаходиться в межах області 32 на 32 пікселі або ж він займає більшу область, але цих меж достатньо для виявлення достатньої кількості точок, щоб забезпечити стеження. Однак при збільшенні масштабу варто також збільшувати область обробки. Таким чином

найкращим варіантом може бути вибір фрейму 96×96 і його розбиття на 9 вікон по 34×34 пікселів розміром. Саме такі розміри пояснюються тим, що вікна мають перетинатись, щоб гарантувати правильну обробку точок на біля меж вікон. Оскільки вхідними даними системи є координати потенційного центру об'єкту, то починати обхід цих вікон варто з центрального. Згодом, якщо достатня точок була не знайдена, треба розглядати сусідні.

На рис. 2.3 зображено порядок обходу вікон відповідно до заданих умов, що враховує найбільш ймовірні місця виявлення характерних точок. Таким чином значно збільшується швидкодія системи, бо специфіка знаходження точок вимагає спочатку виконання обрахунків на визначеній області, а потім порівняння цих точок між собою. Тому поступовий обхід по меншим областям одночасно зменшує кількість ітерацій і дозволяє зупинитись у випадку, якщо достатня кількість точок була знайдена, не перевіряючи весь фрейм повністю.

8	4	9
2	1	3
6	5	7

Рисунок 2.3 – Порядок обходу вікон у фреймі

Перевірка того, чи є піксель $B(x, y, \sigma)$ центром характерної точки відбувається шляхом порівняння його з 8 сусідніми значеннями на даному шарі та на 9 + 9 значеннях на двох сусідніх шарах. Тобто в сумі порівняння відбувається з 26 значеннями і якщо дане значення це локальним максимум

серед цих значень, значить піксель з координатами (x, y) на даному шарі σ є центром характерної точки.

Оскільки значення потрібно порівнювати з сусідніми масштабами, то характерні точки можуть бути знайдені на масштабах $\sigma = 2, 2.8$. Але якщо зображення має маленькі розміри і роздільна здатність камери на хорошому рівні, то можна також розглядати точки на шарі $\sigma = 1.2$, порівнюючи її з одним сусіднім шаром – 17 значеннями, але треба враховувати нижчий рівень довіри до таких точок.

2.1.5 Обчислення дескриптора

Для кожного центрального пікселя $H(x, y, \sigma)$ характерної точки розраховується вектор-дескриптор, що виражає домінантну орієнтацію в околі цієї точки.

Навколо кожної з характерних точок формується квадрат розміром 20σ, яка розділяється на 16 вікон, кожен з яких має розмір $5\sigma \times 5\sigma$.

У кожному з 16 вікон розміщується рівномірно розподілені 25 точок. У кожній з них обчислюються згортки з вейвлетами Хаара для горизонтального d_{ix} та вертикального d_{iy} напрямків. Такі згортки обраховуються так само як на етапі пошуку характерних точок, тобто віднімаються суми значень в двох сусідніх прямокутних областях. При цьому розмір цих вейвлетів 2σ, тобто 2, 4, 6 для різних масштабів. Завдяки підготовленому інтегральному зображенню ця операція сильно пришвидшується [13].

Для спрощення вважаємо, що зображення не підлягає поворотам на кут більше 15°. Інакше необхідно додатково розраховувати провідний напрямок характерної точки, повертати зображення на відповідний кут навколо центрального пікселя і формувати інтегральне зображення.

Для того, щоб збільшити точність локалізації та стійкість до геометричних змін, значення d_x та d_y зважуються за допомогою ядра фільтра Гауса з параметром $\sigma = 3.3 \cdot s$. Значення ваг обчислюються залежно від відстані точки до центру субрегіону.

$$\omega(x, y) = \exp\left(-\frac{x^2 + y^2}{2\sigma^2}\right) \quad (8)$$

Після зважування для кожного субрегіону обчислюються [2]:

$$\sum dx\,, \sum dy\,, \sum |dx|\,, \sum |dy|, \quad (9)$$

де dx, dy – компоненти градієнта яскравості у напрямках x та y;

$|dx|$ та $|dy|$ – абсолютні значення цих компонент.

Оскільки таких субрегіонів 16, то всього вектор складається з 64 значень.

$$D = \{v_1, v_2, v_3, \dots, v_{64}\} \in \mathbb{R}^{64} \quad (10)$$

Для забезпечення незалежності від абсолютної яскравості та контрасту вектор-дескриптор нормується за евклідовою нормою:

$$V \to \frac{V}{\|V\|}, \qquad \|V\| = \sqrt{\sum_{k=1}^{64} v_k^2} \quad (11)$$

Такі нормалізовані вектори далі порівнюються між собою. В даному випадку порівнюються дескриптори еталонних точок з поточними, щоб знайти відповідності. Для цього використовується евклідова відстань між цими дескрипторами:

$$D(V_a, V_b) = \sqrt{\sum_{i=1}^{64} (v_i^{(a)} - v_i^{(b)})^2} \quad (12)$$

Якщо відстань D є менша за встановлену межу, такі точки вважаються відповідними.

2.1.6 Обробка результатів пошуку та формування опису об'єкта

Об'єкт описується множиною характерних точок $\{V_i\}$ для кожного i-го кадру. Кожен елемент цієї множини включає координати точок (x, y), масштаб σ, значення $H(x, y, \sigma)$ та вектор-дескриптор. Оскільки об'єкт з часом може переміщуватись по кадру, то фрейм, що обробляється, також має переміщуватись. Під час стеження, для того, щоб визначити наступний центр об'єкта, навколо якого будується ROI для обробки, з попереднього кадру ми зберігаємо центр мас об'єкта. Координати такого центру залежать від останніх характерних точок, дескриптори яких відповідають еталонним, та їхніх відгуків Гессе $H(x, y, \sigma)$. Отже, координати центру мас обчислюються за формулами:

$$x_c = \frac{\sum_i x_i \cdot h_i}{\sum_i h_i}, \qquad y_c = \frac{\sum_i y_i \cdot h_i}{\sum_i h_i}, \tag{13}$$

де h_i – відгук Гессе i-ї характерної точки;

(x_i, y_i) – координати i-ї точки.

Ці координати передаються в наступний кадр і виступають центром об'єкта на ньому для пошуку нових точок.

Якщо точок, у яких дескриптори відповідають еталонним, немає, то центр мас не має вираховуватись і об'єкт вважається втраченим. Якщо кількість таких точок більше за 80%, то еталонні точки потрібно оновити поточними задля забезпечення надійнішого стеження при геометричних змінах об'єкту.

2.2 OpenCV як допоміжний інструмент

2.2.1 Використання OpenCV для захоплення відео та обробки кадрів

OpenCV є потужною та поширеною бібліотекою для задач комп'ютерного зору. У межах даного проєкту вона використовується виключно як допоміжний інструмент для базових операцій, пов'язаних із роботою з відеопотоком і зображеннями. Основна логіка системи, зокрема алгоритм відстеження, реалізована власноруч на мові C++[8] без залучення сторонніх модулів OpenCV.

Зокрема, OpenCV застосовується для:

- Захоплення відео з камери або з відеофайлу;
- Зчитування та обробки окремих кадрів з потоку у зручному форматі;
- Конвертації зображень у відтінки сірого для попередньої обробки;
- Візуалізації результатів — відображення вікон із позначеними об'єктами, ключовими точками, межами тощо.

Ці функції дають змогу значно скоротити час на реалізацію рутинних операцій і зосередитися на розробці та оптимізації власного алгоритму відстеження. OpenCV підтримує широкий набір форматів відео та апаратних пристроїв захоплення, що забезпечує сумісність із більшістю камер без потреби у реалізації низькорівневого доступу.

Таким чином, OpenCV у цьому проєкті не є ядром системи, а виступає технічним посередником, який забезпечує доступ до зображень і спрощує їх обробку у форматі, зручному для подальшого аналізу [5, 9, 10].

2.2.2 Обмежене використання зовнішніх залежностей

Однією з важливих архітектурних рішень під час реалізації системи стало свідоме обмеження використання сторонніх бібліотек та залежностей. У проєкті за межами базових функцій OpenCV, що використовуються для захоплення відео та обробки кадрів, не залучаються жодні зовнішні фреймворки чи алгоритмічні модулі. Такий підхід має низку суттєвих переваг, особливо в контексті розробки застосунку для пристроїв із обмеженими ресурсами, як-от вбудовані системи або одноплатні комп'ютери.

1. Повний контроль над алгоритмом. Реалізація ключових компонентів системи вручну дозволяє глибше зрозуміти їхню внутрішню логіку та поведінку, що спрощує налагодження, оптимізацію та адаптацію алгоритму під специфічні умови використання. Це особливо важливо у задачах реального часу, де навіть незначне затримування може критично вплинути на ефективність системи.
2. Зниження обсягу споживаних ресурсів. Зовнішні бібліотеки часто включають у себе широкий набір функцій, багато з яких не використовуються у конкретному застосунку, проте все одно впливають на розмір програми, споживання пам'яті та час завантаження. Використання власного коду дає змогу зберігати легкість та ефективність виконання.
3. Підвищення портативності та простоти розгортання. Мінімальна кількість залежностей спрощує перенесення застосунку на інші пристрої та платформи, зменшує ймовірність конфліктів версій бібліотек або проблем сумісності. Це особливо актуально для вбудованих систем, де встановлення додаткових залежностей може бути складним або взагалі неможливим.

4. Безпека та передбачуваність. Чим більше стороннього коду використовується в проєкті, тим складніше гарантувати його безпеку, стабільність і відсутність неочікуваної поведінки. Власна реалізація забезпечує більший рівень довіри до кожного етапу обробки даних.
5. Незалежність від змін у зовнішніх проєктах. У разі оновлення або припинення підтримки сторонніх бібліотек розробники можуть зіткнутися з необхідністю переписування значної частини системи. Власна реалізація гарантує стабільність і довготривалу підтримку.

Таким чином, обмеження зовнішніх залежностей не лише підвищує ефективність системи, але й робить її гнучкішою, надійнішою та придатнішою для використання в критичних або ресурсно обмежених середовищах. Такий підхід також дозволяє краще масштабувати рішення та інтегрувати його в інші програмні або апаратні платформи.

2.3 Середовище та інструменти розробки

2.3.1 IDE та редактори коду

Для розробки програмного забезпечення було обрано Visual Studio Code — сучасне легке середовище розробки з відкритим вихідним кодом, яке підтримує широкий спектр мов програмування, зокрема й C++ [11]. Цей редактор забезпечує зручний інтерфейс користувача, високу швидкодію та багатий набір розширень, що значно полегшують процес написання, відлагодження й аналізу коду.

Серед основних переваг Visual Studio Code варто виділити:

- Інтеграція з компіляторами та дебагерами, що дозволяє зручно запускати, тестувати та налагоджувати програму безпосередньо в середовищі;
- Підтримка розширень для форматування коду, автодоповнення, статичного аналізу та інтеграції з Git;
- Легка вага і висока швидкодія, що важливо при роботі на слабших або обмежених за ресурсами пристроях.

У процесі розробки активно використовувалися такі розширення, як *C/C++*, *CMake Tools*, *CodeLLDB* та *clangd*, які забезпечували повноцінне середовище для роботи з мовою C++, зокрема підтримку синтаксису, підсвічування помилок, автодоповнення коду та інтелектуального рефакторингу.

Завдяки відкритості, гнучкості та активній підтримці спільноти, Visual Studio Code було ефективним і надійним інструментом у розробці системи відстеження об’єктів.

2.3.2 Системи збирання (CMake)

Для організації процесу збирання проєкту було використано CMake — одну з найпоширеніших кросплатформених систем керування збіркою [12]. CMake дозволяє гнучко описувати структуру проєкту, автоматизувати процес компіляції та спростити налаштування залежностей між файлами й модулями.

Основні причини вибору CMake:

- Підтримка багатоплатформності: CMake дозволяє створювати проєкти, що легко збираються як у Linux, так і в Windows чи macOS, що особливо важливо для коду, який може бути розгорнутий на різних пристроях.
- Сумісність із різними компіляторами: З CMake можна використовувати як GCC/Clang, так і MSVC, що надає розробнику більшу гнучкість.

- Інтеграція з IDE: Середовища розробки, такі як Visual Studio Code, CLion та Visual Studio, мають повноцінну підтримку CMake, що дозволяє зручно керувати конфігурацією проєкту та здійснювати збірку безпосередньо з IDE.
- Простота підтримки великих проєктів: Завдяки можливості структурувати проєкт на модулі та підкаталоги, CMake чудово підходить для підтримки масштабованих систем.

У межах цієї роботи CMake використовувався для:

- Опису конфігурацій збирання (Debug/Release);
- Зв’язування з бібліотекою OpenCV;
- Підключення зовнішніх заголовків та джерельних файлів;
- Генерації платформозалежних файлів для збірки.

Крім того, конфігураційні файли CMake (CMakeLists.txt) були написані з урахуванням максимальної простоти, що полегшує перенесення проєкту на інші платформи або середовища з обмеженими ресурсами.

2.4 Врахування апаратних обмежень

Оскільки система створювалася з розрахунком на запуск на пристроях із низькою продуктивністю, ключову роль у процесі розробки відігравало врахування обмежень доступних ресурсів. Обмежений обсяг оперативної пам’яті, невелика кількість ядер процесора, а також обмежена тактова частота вимагали ретельного підходу до архітектури, алгоритмів та використання зовнішніх бібліотек. Основна мета полягала у досягненні реального часу виконання задачі відстеження об’єкта без перевантаження системи. Це було досягнуто шляхом використання максимальної кількості методів прискорення роботи системи та зменшенням об’єму даних, що обробляються.

Основною стратегією оптимізації стало написання критичних частин коду вручну, без використання "важких" сторонніх рішень. Зокрема, було реалізовано власну, спрощену версію алгоритму SURF, яка базується на принципі використання інтегрального зображення для швидкого обчислення гесової відповіді. Завдяки цьому вдалось значно зменшити кількість операцій при побудові карти ключових точок.

Також застосовувалися такі методи оптимізації:

- Зниження роздільної здатності кадрів для попереднього аналізу, з подальшим уточненням на повнорозмірному зображенні за потреби;
- Паралельна обробка за допомогою багатопоточності (std::thread) для розділення завдань: захоплення відео, обробка та вивід результатів;
- Попереднє виділення пам’яті для буферів зображень і результатів, що зменшило накладні витрати на алокацію;
- Компільні оптимізації (використання -O2/-O3, -march=native, -ffast-math) для підвищення швидкодії без зміни логіки;
- Мінімізація копіювань об'єктів через передачу за посиланням або move-семантику.

Завдяки сукупності цих підходів, система здатна функціонувати з низьким рівнем споживання ресурсів, що робить її придатною до впровадження на вбудованих пристроях.

ВИСНОВОК ДО РОЗДІЛУ 2

У цьому розділі було детально розглянуто алгоритм SURF. Було охоплено ключові аспекти, які потрібні для реалізації даного методу:

- Інтегральне зображення. Було введено поняття інтегрального зображення як методу попередньої обробки, що забезпечує значне пришвидшення обчислення сум інтенсивностей у прямокутних областях зображення. Інтегральне зображення дозволяє обчислювати середні значення пікселів для прямокутних фільтрів за постійну кількість операцій, що сприяє ефективному виявленню характерних точок.
- Розрахунок значень параметрів масштабу. Було обґрунтовано вибір кількості та діапазону значень параметрів масштабу σ.
- Виявлення ключових точок. Розглянуто метод виявлення характерних точок на основі детермінанту матриці Гессе, апроксимованої із використанням прямокутних фільтрів. Тут були вказані розміри фрейму та субрегіонів для знаходження точок і розглянуто метод знаходження центру характерної точки.
- Обчислення вектора-дескриптора. Було детально розглянуто процес побудови дескрипторів для характерних точок, зібрання їх у вектор, його нормалізацію та порівняння таких дескрипторів між собою для виявлення відповідностей.
- Обробка результатів пошуку. Було досліджено параметри, завдяки яким описується об’єкт, над яким відбувається стеження, як ці параметри використовуються між кадрами для забезпечення неперервності трекінгу.

Важливу роль у розробці відіграла бібліотека OpenCV, яка, хоча і використовувалася обмежено, забезпечила простий та швидкий доступ до функціоналу захоплення відео, базової обробки кадрів, а також виводу візуалізації результатів. Важливою перевагою стало уникнення надмірної залежності від зовнішніх бібліотек, що позитивно вплинуло на портативність і продуктивність застосунку.

Розглянуто також аспекти налагодження, компіляції, профілювання та використання системи збирання CMake, яка забезпечила гнучку конфігурацію проєкту та полегшила підтримку і розширення коду.

Значна увага була приділена оптимізації під обмежені ресурси. Система розроблялася з урахуванням роботи на пристроях з обмеженими обчислювальними можливостями, що вимагало ретельного аналізу алгоритмів, зменшення складності обчислень, запровадження багатопоточності, а також оптимізації використання пам’яті. У результаті вдалося досягти прийнятного рівня продуктивності навіть на слабких апаратних платформах, що підтверджує ефективність обраного технічного підходу.

Таким чином, на основі аналізу технологій та інструментів, було реалізовано рішення, здатне працювати в реальному часі в умовах обмежених ресурсів. Це створює передумови для успішного впровадження системи в реальні задачі з автоматичного відстеження об’єктів.

РОЗДІЛ 3 ДЕТАЛІ РОЗРОБКИ СИСТЕМИ

3.1 Загальна архітектура системи

Система слідкування за рухомим об’єктом побудована на основі модульного підходу, що забезпечує гнучкість та легкість у підтримці. Кожен модуль реалізований у вигляді окремого класу, що забезпечує читабельність коду та передбачувану структуру. Це дозволяє легко замінювати або вдосконалювати окремі компоненти без впливу на інші частини системи.

3.1.1 Опис основних компонентів системи

Система складається з кількох основних компонентів, кожен з яких виконує свою специфічну функцію. Серед них:

- Головний модуль – координує роботу всіх інших компонентів. У головній функції main() здійснюється ініціалізація системи, вибір точки для слідкування, запуск обробки кадрів та вивід результатів продуктивності. Взаємодія між компонентами забезпечується через передачу даних, таких як кадри, ключові точки та дескриптори.
- Модуль зчитування відеопотоку (VideoReader) – відповідає за отримання кадрів із відеофайлу або камери. Використовує багатопотоковий підхід для зчитування кадрів у фоновому режимі, що дозволяє уникнути затримок під час обробки. Кадри зберігаються у черзі, доступ до якої синхронізується за допомогою м’ютексів.
- Модуль обробки кадрів (FrameProcessor) – виконує основні етапи обробки кадрів, включаючи детекцію ключових точок, побудову дескрипторів та слідкування за об’єктом. Цей модуль використовує

алгоритми для обчислення інтегрального зображення, детермінанти матриці Гессе та побудови дескрипторів. Він також відповідає за оновлення позиції об’єкта на основі знайдених ключових точок.

- Модуль для роботи з інтегральним зображенням (IntegralUtils) – забезпечує функції для формування інтегрального зображення та швидкого обчислення сум у прямокутних областях.
- Модуль детекції характерних точок (Detector) – відповідає за пошук ключових точок на зображенні, які мають локальні максимуми у просторі координат і масштабів.
- Модуль побудови дескрипторів (Descriptor) – відповідає за створення векторів-дескрипторів для ключових точок, що дозволяє їх порівнювати між кадрами.
- Модуль візуалізації (Visualizer) – забезпечує графічне відображення результатів обробки кадрів. Включає функції для нанесення ключових точок, центру мас та інших маркерів на зображення. Це дозволяє користувачеві візуально оцінити якість роботи системи.
- Модуль вимірювання продуктивності (Timer) – реалізує таймер для оцінки часу виконання окремих частин алгоритму. Використовується для аналізу швидкодії системи та оптимізації її роботи.

3.1.2 Взаємодія компонентів

Головний потік виконання системи складається з 3 етапів:

- VideoReader передає кадри до FrameProcessor, який обробляє їх для виявлення ключових точок та оновлення позиції об’єкта.
- FrameProcessor використовує функції з модулів Detector, Descriptor та IntegralUtils для виконання алгоритмічних операцій.

- Результати обробки кадрів передаються до Visualizer, який відображає їх на екрані.

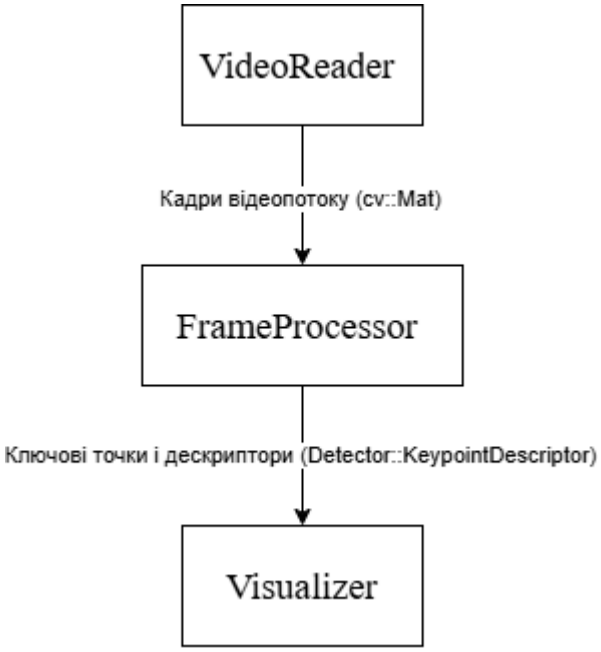


Рисунок 3.1 – Головний потік виконання програми

3.2 Модуль обробки відеопотоку

Для забезпечення обробки відеопотоку в реальному часі в рамках розробленої системи було реалізовано багатопоточну архітектуру з розділенням завдань на окремі етапи. Основна мета полягає в тому, щоб уникнути затримок при зчитуванні кадрів з відеофайлу та забезпечити стабільну подачу зображень на вхід наступного модуля.

Клас VideoReader запускає окремий потік для зчитування відео, що дозволяє не блокувати головний потік, який відповідає за аналіз зображення. Кадри зберігаються у черзі фіксованого розміру (max_queue_size) у вигляді зображень у градаціях сірого.

Основна логіка зчитування реалізована у методі readerThread (рис. 3.2).

```cpp
void VideoReader::readerThread() {
    while (!done_reading_) {
        cv::Mat frame;
        if (!cap_.read(frame)) {
            break;
        }

        cv::Mat gray;
        cv::cvtColor(frame, gray, cv::COLOR_BGR2GRAY);

        std::unique_lock<std::mutex> lock(queue_mutex_);
        frame_available_.wait(lock, [this] { return frame_queue_.size() < max_queue_size_ || done_reading_; });
        if (done_reading_) {
            break;
        }
        frame_queue_.push(gray);
        lock.unlock();
        frame_available_.notify_one();
    }
    done_reading_ = true;
    frame_available_.notify_all();
}
```

Рисунок 3.2 – метод readerThread для зчитування кадрів

Така реалізація дозволяє зчитувати кадри з відеопотоку незалежно від того, скільки часу займає їх подальша обробка.

Кадри зчитуються з черги за допомогою методу getNextFrame (рис. 3.3), який призупиняє виконання головного потоку до появи нового кадру. Якщо всі кадри прочитано, метод повертає false, що сигналізує про завершення відеопотоку.

```cpp
bool VideoReader::getNextFrame(cv::Mat& frame) {
    std::unique_lock<std::mutex> lock(queue_mutex_);
    frame_available_.wait(lock, [this] { return !frame_queue_.empty() || done_reading_; });
    if (frame_queue_.empty() && done_reading_) {
        return false;
    }
    frame = frame_queue_.front();
    frame_queue_.pop();
    frame_available_.notify_one();
    return true;
}
```

Рисунок 3.3 - Метод, який витягує кадр з черги для подальшої обробки

У головному потоці кадри витягуються з черги та передаються на обробку в об’єкт класу FrameProcessor (рис. 3.4).

```
void FrameProcessor::processFrames(VideoReader& video_reader, FrameProcessor& processor, const char* window_name) {
    cv::Mat frame;

    int total_keypoints_detected = 0;
    int total_matches_found = 0;

    while (true) {
        if (!video_reader.getNextFrame(frame)) {
            break;
        }

        cv::Mat output;
        double detect_time;
        if (!processor.processFrame(frame, output, detect_time)) {
            break;
        }

        total_keypoints_detected += processor.getLastFrameKeypointsCount();
        total_matches_found += processor.getLastFrameMatchesCount();

        Visualizer::displayImage(output, window_name);
        int key = cv::waitKey(1);
        if (key == 27 || key == 'q') {
            break;
        }
    }

    if (processor.getTotalFrames() > 0) {
        std::cout << "=== Average Timings (s) ===\n";
        std::cout << "Frames processed: " << processor.getTotalFrames() << "\n";
        std::cout << "Average frame time: " << (processor.getTotalFrameTime() / processor.getTotalFrames()) * 1000 << " ms\n";
        std::cout << "=== Tracking Statistics ===\n";
        std::cout << "Avarage keypoints detected: " << total_keypoints_detected / processor.getTotalFrames() << "\n";
        std::cout << "Avarage matches found: " << total_matches_found / processor.getTotalFrames() << "\n";

        Visualizer::visualizeTrajectory(processor.getTrajectory(), frame.cols, frame.rows);
    } else {
        std::cout << "No frames processed.\n";
    }
}
```

Рисунок 3.4 – метод processFrames з викликом getNextFrame

Цей цикл забезпечує також покадрову обробку відеопотоку, включаючи виявлення об'єкта, його позиціювання та візуалізацію результатів.

Серед переваг такого підходу: зниження затримок завдяки незалежному зчитуванню та можливість обробляти кадри великої роздільної здатності з умови достатньої обчислювальної потужності.

3.3 Реалізація модуля детекції ключових точок

3.3.1 Виділення області інтересу (ROI)

Для локалізації ключових точок та подальшого аналізу було реалізовано динамічне виділення області інтересу навколо об’єкта, що відстежується (рис. 3.5). Це дозволяє суттєво зменшити обсяг обчислень та підвищити точність детекції завдяки фокусуванню на релевантній ділянці зображення.

Область інтересу визначається як квадрат фіксованого розміру, центрований навколо поточної координати центру об’єкта current_center_. Після визначення прямокутника ROI виконується перевірка його перетину з межами зображення. Якщо ROI повністю входить у межі зображення, з нього безпосередньо вирізається фрагмент. Якщо ж ні — створюється заповнена сірим фоном cv::Scalar(128) область, куди копіюється доступна частина зображення.

```
int roi_size = 152;

cv::Rect roi(current_center_.x - roi_size / 2, current_center_.y - roi_size / 2, roi_size, roi_size);
cv::Mat patch(roi_size, roi_size, gray.type(), cv::Scalar(128));
cv::Rect img_rect(0, 0, gray.cols, gray.rows);
cv::Rect roi_in_img = roi & img_rect;

if (roi_in_img.area() == roi_size * roi_size) {
    patch = gray(roi).clone();
}
else {
    patch.setTo(128);
    int dx = roi_in_img.x - roi.x;
    int dy = roi_in_img.y - roi.y;
    gray(roi_in_img).copyTo(patch(cv::Rect(dx, dy, roi_in_img.width, roi_in_img.height)));
}

cv::Point center_in_roi = current_center_ - roi.tl();
```

Рисунок 3.5 – Виділення ROI з обробкою граничних випадків

Також розраховується координата об’єкта у системі координат ROI center_in_roi, що потрібна для подальшої локальної детекції ознак.

Оскільки в середині ROI виділяється фрейм 96×96, то roi_size має бути не меншим за це значення. Однак важливо також врахувати, що найбільше значення $\sigma = 2.8$, для якого при створенні дескрипторів аналізується область $20\sigma = 56$, створює додаткову вимогу в кількості кадрів області інтересу. Тобто якщо точка знаходиться на краю фрейма, то потрібно враховувати ще $56 \div 2 = 28$ точок з усіх боків. Тобто roi_size має набувати значень не менше ніж $96 + 2 \cdot 28 = 152$.

Отриманий фрагмент передається до методу detectKeypointsWithDescriptors (рис. 3.6), де виконується виділення ключових точок з обчисленням дескрипторів у межах ROI.

```
auto patch_descriptors = Detector::detectKeypointsWithDescriptors(patch, center_in_roi, sigmas_, thresholds_);
```

Рисунок 3.6 – Виклик детектора для ROI

Після знаходження ключових точок та дескрипторів потрібно знову повернути масштаб всього зображення для коректного відображення точок користувачу.

```
for (auto& d : patch_descriptors) {
    d.x += roi.x;
    d.y += roi.y;
}
```

Рисунок 3.7 – Додавання початкових координат ROI до координат знайдених точок

3.3.2 Формування інтегрального зображення

Для формування інтегрального зображення і для дій над ним було створено окремий модуль IntegralUtils. Він складається всього з двох методів: computeIntegralImage для побудови зображення та boxIntegral для швидкого обрахунку суми прямокутної області. Дані методи представлені на рис. 3.8.

```
cv::Mat IntegralUtils::computeIntegralImage(const cv::Mat& img) {
    cv::Mat integral;
    cv::integral(img, integral, CV_32F);
    return integral;
}

float IntegralUtils::boxIntegral(const cv::Mat& integral, int top, int left, int bottom, int right) {
    float A = integral.at<float>(top, left);
    float B = integral.at<float>(top, right);
    float C = integral.at<float>(bottom, left);
    float D = integral.at<float>(bottom, right);

    return std::max(0.0f, D - B - C + A);
}
```

Рисунок 3.8 – Методи модуля IntegralUtils

Для методу computeIntegralImage було використано OpenCV через простоту використання і швидкодію. Побудова інтегрального зображення робиться один раз перед початком пошуку характерних точок та дескрипторів для всього ROI.

Метод boxIntegral використовується для розрахунку прямокутних фільтрів для виявлення характерних точок і для вейвлетів Хаара при формуванні дескрипторів.

3.3.3 Виділення області пошуку та розбиття на вікна

На основі вказаного центрального пікселя здійснюється формування робочого фрейму розміром 96×96 пікселів, який охоплює окіл навколо точки

інтересу. Для підвищення надійності виявлення ключових точок, даний фрейм додатково розбивається на 9 перекривних вікон розміром 34×34 пікселі, які охоплюють центральну, кутові та бокові області. Це дозволяє уникнути втрати точок, що розташовані поблизу меж основної області. Код, що реалізує дану операцію, знаходиться в методі divideFrameIntoWindows.

```
std::vector<cv::Rect> windows;
const int window_size = Config::WINDOW_SIZE;
const int frame_size = Config::FRAME_SIZE;
const int half_frame = frame_size / 2;

const int num_windows = 3;
const int overlap = (num_windows * window_size - frame_size) / (num_windows - 1);

int frame_top_left_x = center.x - half_frame;
int frame_top_left_y = center.y - half_frame;
```

Рисунок 3.9 – Створення ідентифікаторів для розбиття фрейму на вікна

```
// Offsets for 9 windows (dy, dx) in required order
const std::vector<std::pair<int, int>> offsets = {
    {0, 0},     // Center
    {0, -1},    // Left
    {0, 1},     // Right
    {-1, 0},    // Top
    {1, 0},     // Bottom
    {1, -1},    // Bottom-left
    {1, 1},     // Bottom-right
    {-1, -1},   // Top-left
    {-1, 1}     // Top-right
};
```

Рисунок 3.10 – Порядок обходу вікон

```
for (const auto& offset : offsets) {
    int dx = offset.second * (window_size - overlap);
    int dy = offset.first * (window_size - overlap);

    windows.emplace_back(
        frame_top_left_x + half_frame + dx - window_size / 2,
        frame_top_left_y + half_frame + dy - window_size / 2,
        window_size,
        window_size
    );
}

return windows;
```

Рисунок 3.11 – Розрахунок координат, збереження та повернення вікон

3.3.4 Обчислення відгуку Гессе

На даному етапі реалізовано метод для оцінки відгуку Гессе — значення детермінанта наближеної матриці другого порядку похідних по трьом осям: D_{xx}, D_{yy}, D_{xy}. Обчислення виконується на основі інтегрального зображення, що дозволяє швидко оцінювати суму пікселів у прямокутній області.

Всі обчислення відбуваються у методі detectKeypoints. В ньому відбувається почерговий обхід усіх вікон. Спочатку для кожного значення σ на кожному пікселі інтегрального зображення обчислюється відгук Гессе за допомогою методу computeHessianResponse.

```
float Detector::computeHessianResponse(const cv::Mat& ii, int y, int x, float sigma, float threshold) {
    int filter_size = static_cast<int>(std::round(7.5f * sigma));
    int half_size = filter_size / 2;
    int third_size = filter_size / 3;

    int top = y - half_size - 1;
    int bottom = y + half_size;
    int left = x - half_size;
    int right = x + half_size + 1;

    int mid_top = y - third_size / 2 - 1;
    int mid_bottom = y + third_size / 2;
    int mid_left = x - third_size / 2;
    int mid_right = x + third_size / 2 + 1;
```

Рисунок 3.12 – Підготовка ідентифікаторів для знаходження значень прямокутних фільтрів

```
float left_third = IntegralUtils::boxIntegral(ii, top, left, bottom, mid_left);
float horizontal_center_third = IntegralUtils::boxIntegral(ii, top, mid_left, bottom, mid_right);
float right_third = IntegralUtils::boxIntegral(ii, top, mid_right, bottom, right);

float top_third = IntegralUtils::boxIntegral(ii, top, left, mid_top, right);
float vertical_center_third = IntegralUtils::boxIntegral(ii, mid_top, left, mid_bottom, right);
float bottom_third = IntegralUtils::boxIntegral(ii, mid_bottom, left, bottom, right);

int top_square = y - half_size;
int right_square = x + half_size;

float top_left = IntegralUtils::boxIntegral(ii, top_square, left, y, x);
float top_right = IntegralUtils::boxIntegral(ii, top_square, x, y, right_square);
float bottom_left = IntegralUtils::boxIntegral(ii, y, left, bottom, x);
float bottom_right = IntegralUtils::boxIntegral(ii, y, x, bottom, right_square);
```

Рисунок 3.13 – Обчислення сум в межах апертур фільтрів

```
// Calculate Dxx, Dyy, and Dxy
float Dxx = left_third + right_third - 2 * horizontal_center_third;
float Dyy = top_third + bottom_third - 2 * vertical_center_third;
float Dxy = top_left + bottom_right - top_right - bottom_left;

float inv_area = 1.0f / (filter_size * filter_size);

float weight = Config::HESSIAN_RESPONSE_COEFFICIENT;
float determinant = (Dxx * Dyy - weight * Dxy * Dxy) * inv_area;

return determinant >= threshold ? determinant : 0.0f;
```

Рисунок 3.14 – Обчислення наближеного детермінанту матриці Гессе

Далі відповідно до алгоритму відбувається визначення характерних точок. Для цього потрібно порівняти кожне значення відгуку з сусідніми на тому самому шарі (рис. 3.15).

```
for (int y = 1; y < responses[i].rows - 1; ++y) {
    for (int x = 1; x < responses[i].cols - 1; ++x) {
        float r = responses[i].at<float>(y, x);

        bool is_local_max = true;
        for (int dy = -1; dy <= 1; ++dy) {
            for (int dx = -1; dx <= 1; ++dx) {
                if (dy == 0 && dx == 0) continue;
                if (r <= responses[i].at<float>(y + dy, x + dx)) {
                    is_local_max = false;
                    break;
                }
            }
            if (!is_local_max) break;
        }

        if (is_local_max && r > thresholds[i]) {
            mask.at<uchar>(y, x) = 255;
        }
    }
}
```

Рисунок 3.15 – Знаходження локальних максимумів на одному шарі

Далі відбувається порівняння зі значеннями на сусідніх шарах та збереження результатів (рис. 3.16).

```
if (i > 0) {
    mask &= responses[i] > responses[i - 1];
}
if (i < sigmas.size() - 1) {
    mask &= responses[i] > responses[i + 1];
}

for (int y = 0; y < mask.rows; ++y) {
    for (int x = 0; x < mask.cols; ++x) {
        if (mask.at<uchar>(y, x)) {
            float r = responses[i].at<float>(y, x);
            result.emplace_back(win.x + x, win.y + y, sigmas[i], r);
        }
    }
}
```

Рисунок 3.16 – Порівняння відгуків з сусідніми шарами та збереження результатів

Таким чином для поточного вікна було знайдено певну кількість точок, що можна вважати характерними. Якщо цих точок достатньо для слідкування за об’єктом, то потрібно завершити виконання методу та повернути результати для наступного модуля. Є також ймовірність, що точок у вікні знайдено більше, ніж ми очікуємо. В такому випадку при звичайному відкиданні зайвих є шанс відкинути точки з більшим значенням відгуку Гессе. Щоб уникнути цього, результат спочатку фільтрується за спаданням. Це продемонстровано в наступній ділянці коду на рис. 3.17.

```
if (result.size() > static_cast<size_t>(max_keypoints * 1.5)) {
    std::sort(result.begin(), result.end(), [](const Keypoint& a, const Keypoint& b) {
        return a.response > b.response;
    });
    result.resize(max_keypoints * 1.5);
    return result;
}
```

Рисунок 3.17 – Передчасне закінчення методу detectKeypoints при достатній кількості точок

Якщо після огляду всіх вікон потрібну кількість так і не було знайдено, то результат буде повернений для наступного кроку, але оскільки кількість точок буде меншою, то ймовірність подальшого спостереження також знижується.

В результаті виконання цього методу отримуються характерні точки з координатами, значеннями σ та відгуку Гессе.

3.4 Реалізація модуля побудови дескрипторів

Щоб описати локальну структуру навколо кожної знайденої характерної точки використовується дескриптор, що моделює просторовий розподіл

градієнтів в околі точки. Для його обчислення використовується метод computeDescriptor. Виклик методу для кожної точки на рис. 3.18.

```
std::vector<KeypointDescriptor> descriptors;

for (const auto& kp : kps) {
    auto desc = Descriptor::computeDescriptor(ii, kp.x, kp.y, kp.sigma);
    descriptors.emplace_back(kp.x, kp.y, kp.sigma, kp.response, desc);
}
```

Рисунок 3.18 – Виклик computeDescriptor для кожної точки

3.4.1 Обчислення вагів Гауса для субрегіону

На етапі обчислення вектора-дескриптора навколо точки створюється область, що ділиться на 16 субрегіонів розміром $20\sigma \div 4 = 5\sigma$. Кожен з таких субрегіонів містить 25 рівномірно розподілених точок. Для кожної з цих точок розраховується згортки з вейвлетами Хаара по горизонтальному і вертикальному напрямках, які потім зважуються ядром фільтра Гауса. Оскільки для кожного субрегіону на визначеному шарі ваги будуть набувати однакових значень, то головний метод computeDescriptor починається зі знаходження цих ваг у методі getGaussianWeights (рис. 3.20).

```
std::vector<float> Descriptor::computeDescriptor(const cv::Mat& integral, float x, float y, float scale) {
    const int sample_count = 5;
    const int grid_size = 4;
    const int region_size = static_cast<int>(Config::DESCRIPTOR_REGION_SIZE_MULTIPLIER * scale);
    const int half_region = region_size / 2;
    const int step = region_size / grid_size;
    const float sample_step = static_cast<float>(step) / sample_count;
    const float wavelet_size = 2.0f * scale;

    cv::Mat weights = Descriptor::getGaussianWeights(scale, grid_size, region_size, sample_count);
```

Рисунок 3.20 – Обрахунок допоміжних ідентифікаторів та виклик getGaussianWeights

Код методу getGaussianWeights представлено на рис. 3.21.

```
cv::Mat Descriptor::getGaussianWeights(float scale, int grid_size, int region_size, int sample_count) {
    int step = region_size / grid_size;
    float sigma = Config::GUASSIAN_WEIGHTS_SIGMA_MULTIPLIER * scale;
    float sample_step = static_cast<float>(step) / sample_count;

    cv::Mat weights(sample_count, sample_count, CV_32F);

    for (int i = 0; i < sample_count; ++i) {
        for (int j = 0; j < sample_count; ++j) {
            float x = (j - (sample_count - 1) / 2.0f) * sample_step;
            float y = (i - (sample_count - 1) / 2.0f) * sample_step;
            float r2 = x * x + y * y;

            weights.at<float>(i, j) = std::exp(-(r2) / (2.0f * sigma * sigma));
        }
    }

    return weights;
}
```

Рисунок 3.21 – Обчислення ваг Гессе

Цей метод створює матрицю вагів розміром sample_count × sample_count, яка застосовується до кожного осередку в регіоні побудови дескриптора.

3.4.2 Обчислення дескриптора ключової точки

Дескриптор будується як вектор довжиною 64:

- Регіон навколо точки масштабується відповідно до scale,
- Розбивається на 4 на 4 сітку,
- У кожному блоці обчислюються 4 компоненти:
  - Сума похідних по x
  - Сума похідних по y
  - Сума абсолютних значень похідних по x
  - Сума абсолютних значень похідних по y

Таким чином $4 \times 4 \times 4 = 64$ значення. Основний метод computeDescriptor складається з обходу по сітці 4×4, в середині кожного блока обхід по 25 точкам, для кожної точки обрахунок наближеної похідної по d_x та d_y, зважування ядром фільтру Гауса та нормалізації вектору.

```
for (int i = 0; i < grid_size; ++i) {
    for (int j = 0; j < grid_size; ++j) {
        float dx_sum = 0.0f;
        float dy_sum = 0.0f;
        float dx_sum_abs = 0.0f;
        float dy_sum_abs = 0.0f;
```

Рисунок 3.22 – Ітерація по кожному блоку в сітці 4×4

```
for (int u = 0; u < sample_count; ++u) {
    for (int v = 0; v < sample_count; ++v) {
        float sample_x = x - half_region + j * step + (v + 0.5f) * sample_step;
        float sample_y = y - half_region + i * step + (u + 0.5f) * sample_step;

        int top = static_cast<int>(sample_y - wavelet_size / 2);
        int bottom = static_cast<int>(sample_y + wavelet_size / 2);
        int left = static_cast<int>(sample_x - wavelet_size / 2);
        int right = static_cast<int>(sample_x + wavelet_size / 2);

        float dx = IntegralUtils::boxIntegral(integral, top, static_cast<int>(sample_x), bottom, right) -
                   IntegralUtils::boxIntegral(integral, top, left, bottom, static_cast<int>(sample_x));

        float dy = IntegralUtils::boxIntegral(integral, static_cast<int>(sample_y), left, bottom, right) -
                   IntegralUtils::boxIntegral(integral, top, left, static_cast<int>(sample_y), right);

        float weight = weights.at<float>(u, v);
        dx_sum += dx * weight;
        dy_sum += dy * weight;
        dx_sum_abs += std::abs(dx) * weight;
        dy_sum_abs += std::abs(dy) * weight;
    }
}
```

Рисунок 3.23 – Обхід 25 точок, обрахунок d_x та d_y, масштабування Гаусовими вагами

```
            int idx = 4 * (i * grid_size + j);

            descriptor[idx] = dx_sum;
            descriptor[idx + 1] = dy_sum;
            descriptor[idx + 2] = dx_sum_abs;
            descriptor[idx + 3] = dy_sum_abs;
        }
    }

    float norm = 0.0f;
    for (float val : descriptor) norm += val * val;
    norm = std::sqrt(norm);
    if (norm > 0.0f) {
        for (float& val : descriptor) val /= norm;
    }

    return descriptor;
}
```

Рисунок 3.24 – Збереження значень у вектор, нормалізація дескриптора

3.4.3 Фільтрація подібних до еталонних точок

Для покращення стабільності відстеження об'єкта та зменшення кількості хибних відповідностей реалізовано метод filterSimilarKeypoints (рис. 3.25). Він виконує пошук подібних дескрипторів між двома наборами ключових точок — поточним (current) та еталонним (reference). Метод порівнює кожен дескриптор із поточного набору з усіма дескрипторами з еталонного і зберігає ті, які мають евклідову відстань меншу за заданий поріг.

```cpp
std::vector<KeypointDescriptor> Descriptor::filterSimilarKeypoints(
        const std::vector<KeypointDescriptor>& current,
        const std::vector<KeypointDescriptor>& reference,
        float max_distance
    ) {
        std::vector<KeypointDescriptor> filtered;

        for (const auto& kp : current) {
            float best_dist_sq = std::numeric_limits<float>::max();
            for (const auto& ref_kp : reference) {
                if (kp.descriptor.size() != ref_kp.descriptor.size()) continue;

                float dist_sq = 0.0f;
                const float* a = kp.descriptor.data();
                const float* b = ref_kp.descriptor.data();
                for (size_t i = 0; i < kp.descriptor.size(); ++i) {
                    float diff = a[i] - b[i];
                    dist_sq += diff * diff;
                    if (dist_sq > max_distance * max_distance) break;
                }
                if (dist_sq < max_distance * max_distance) {
                    filtered.push_back(kp);
                    break;
                }
            }
        }

        return filtered;
}
```

Рисунок 3.25 – Фільтрація подібних до еталонних точок за дескрипторами

Алгоритм роботи:

- Для кожної точки з поточного набору обчислюється квадрат евклідової відстані до кожної точки з еталонного набору.
- Якщо знайдено дескриптор, що має відстань меншу за max_distance, точка вважається подібною та додається до результату.
- Вихідний вектор filtered містить усі знайдені подібні точки.

У класі FrameProcessor метод filterSimilarKeypoints використовується при обробці кожного нового кадру (рис. 3.26). Він фільтрує набір ключових точок, знайдених у регіоні інтересу, порівнюючи їх з попередньо збереженими (еталонними). Якщо кількість збігів перевищує 80% від кількості еталонних точок, вони використовуються як нові еталонні.

```
auto descriptors = Descriptor::filterSimilarKeypoints(patch_descriptors, reference_descriptors_, Config::MAX_DISTANCE_FOR_MATCHING);

if (descriptors.size() > reference_descriptors_.size() * Config::UPDATE_REFERENCE_POINTS_PERCENTAGE) {
    reference_descriptors_ = descriptors;
}
```

Рисунок 3.26 – Оновлення еталонних точок

Це дозволяє поступово оновлювати опорні дескриптори, адаптуючись до змін зовнішнього вигляду об'єкта.

3.4.4 Розрахунок центра мас ключових точок

Для стабільного відстеження об'єкта на послідовності кадрів використовується обчислення центра мас ключових точок, які залишилися після фільтрації за схожістю дескрипторів. Такий центр слугує як оновлене положення об’єкта у поточному кадрі.

Центр мас визначається як зважене середнє координат усіх ключових точок. У якості ваги використовується відповідь (response), яка відображає значущість точки.

Реалізація методу calculateMassCenter виглядає наступним чином (рис. 3.27).

```cpp
bool Descriptor::calculateMassCenter(const std::vector<KeypointDescriptor>& descriptors, cv::Point& center) {
    double total_weight = 0.0;
    double x_sum = 0.0, y_sum = 0.0;

    for (const auto& ds : descriptors) {
        double response = ds.response;
        total_weight += response;
        x_sum += ds.x * response;
        y_sum += ds.y * response;
    }

    if (total_weight == 0.0) {
        return false;
    }

    center.x = static_cast<int>(x_sum / total_weight);
    center.y = static_cast<int>(y_sum / total_weight);
    return true;
}
```

Рисунок 3.27 – Обчислення центру мас знайдених точок

3.5 Модуль оцінки швидкодії системи

Для оцінки ефективності реалізованого алгоритму та визначення його «вузьких місць» було створено спеціальний модуль вимірювання продуктивності. Основне завдання цього модуля — визначити загальний час обробки кадру, а також час, що витрачається на окремі етапи, такі як детекція ключових точок, обчислення дескрипторів тощо.

Для вимірювання часу використано клас Timer, який інкапсулює механізм високоточного хронометражу на основі std::chrono. Клас дозволяє запускати й зупиняти таймер, отримувати проміжні результати та за потреби автоматично виводити їх у консоль.

Реалізація теймера на рис. 3.28.

```cpp
Timer::Timer(const std::string& name)
    : name_(name), print_on_destruct_(false) {}

void Timer::start() {
    start_time_ = std::chrono::high_resolution_clock::now();
    elapsed_.reset();
}

double Timer::stop() {
    const auto end_time = std::chrono::high_resolution_clock::now();
    elapsed_ = std::chrono::duration<double>(end_time - start_time_).count();
    return *elapsed_;
}

double Timer::getElapsed() const {
    const auto now = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double>(now - start_time_).count();
}

void Timer::printElapsedOnDestruct(bool enable) {
    print_on_destruct_ = enable;
}

void Timer::printElapsed() const {
    if (elapsed_.has_value()) {
        if (!name_.empty())
            std::cout << "[" << name_ << "] Elapsed time: " << *elapsed_ << " seconds\n";
        else
            std::cout << "Elapsed time: " << *elapsed_ << " seconds\n";
    }
}

Timer::~Timer() {
    if (print_on_destruct_) {
        printElapsed();
    }
}
```

Рисунок 3.28 – Реалізація класу для вимірювання швидкодії

Таймер було інтегровано безпосередньо в метод processFrame класу FrameProcessor. Після запуску таймера (timer_.start()) відбувається повна обробка поточного кадру. Після завершення — вимірюється загальний час (timer_.stop()) та час, витрачений лише на детекцію (timer_.getElapsed()).

Після завершення обробки відео виводяться середні значення за всі оброблені кадри (рис. 3.29).

```
=== Average Timings (s) ===
Frames processed: 408
Average frame time: 0.00375542 s
Average detect time: 0.00321028 s
```

Рисунок 3.29 – Формат виводу результатів виміру

3.6 Модуль візуалізації результатів

Для зручного аналізу роботи алгоритму відстеження об’єкта в реальному часі було реалізовано окремий модуль візуалізації, що дозволяє відображати проміжні та кінцеві результати обробки кадрів.

Для кожної виявленої точки на зображенні будується коло, радіус якого залежить від масштабу σ (рис. 3.30).

```
void Visualizer::drawKeypoints(cv::Mat& img, const std::vector<KeypointDescriptor>& keypoints) {
    for (const auto& kp : keypoints) {
        cv::circle(img, cv::Point(kp.x, kp.y), static_cast<int>(kp.sigma * 5), cv::Scalar(128, 128, 128), 1);
    }
}
```

Рисунок 3.30 – Візуалізація ключових точок

Забезпечується відображення поточного кадру з результатами на екран із використанням OpenCV (рис. 3.31).

```
void Visualizer::displayImage(const cv::Mat& img, const std::string& windowName) {
    cv::imshow(windowName, img);
}
```

Рисунок 3.31 – Відображення кадру

Для позначення обраної точки користувача використовується хрест (рис. 3.32).

```
void Visualizer::drawCross(cv::Mat& img, const cv::Point& pt, const cv::Scalar& color, int size, int thickness) {
    cv::line(img, cv::Point(pt.x - size, pt.y), cv::Point(pt.x + size, pt.y), color, thickness);
    cv::line(img, cv::Point(pt.x, pt.y - size), cv::Point(pt.x, pt.y + size), color, thickness);
}
```

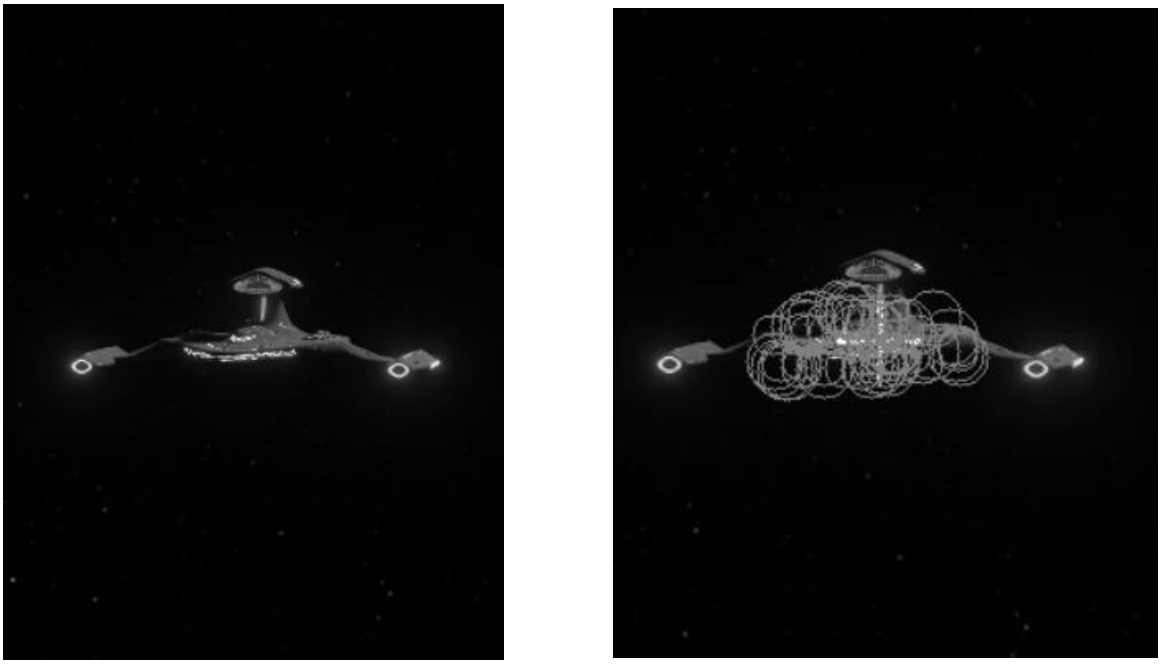
Рисунок 3.32 – Відображення Вибраної точки

При необхідності можливо відобразити вектор (рис. 3.33).

```
void Visualizer::drawVector(cv::Mat& img, const cv::Point& start, const cv::Point& end, const cv::Scalar& color, int thickness) {
    cv::arrowedLine(img, start, end, color, thickness);
}
```

Рисунок 3.33 – Відображення вектору

Приклад відображення результату знаходження характерних відфільтрованих характерних точок та центру мас на рис. 3.34.



а) б)

Рисунок 3.34 – Кадр до (а) та після (б) відображення ключових точок та центру мас

3.7 Труднощі в процесі розробки та їх подолання

У ході реалізації системи виникли деякі труднощі, що впливали на ефективність та точність програмного забезпечення.

3.7.1 Недостатня швидкодія початкової реалізації

На ранньому етапі розробки виявилося, що час обробки одного кадру перевищує допустимі межі для реального часу і становить понад 2 секунди. Основними причинами такої затримки були неефективні алгоритмічні рішення, обчислювально затратні операції над повним зображенням, а також відсутність оптимізації в роботі з ключовими точками.

Для підвищення швидкодії було реалізовано кілька заходів оптимізації:

- Обмеження області інтересу: Замість обробки всього зображення система фокусувалася на області навколо точки відстеження, що значно зменшило кількість пікселів для аналізу.
- Поділ кадру на вікна: Визначення кандидатів на ключові точки здійснювалося окремо в кожному вікні, що дозволило уникнути глобального пошуку максимуму по всьому зображенню.
- Паралельна обробка: Зчитування кадрів відбувається в паралельному потоці, щоб не затримувати виконання основного алгоритму.

Унаслідок оптимізації час обробки одного кадру було зменшено більш ніж у 100 разів, що дозволило досягти продуктивності, прийнятної для обробки відео в режимі реального часу.

3.7.2 Обмеження через патентування SURF-алгоритму

Однією з основних проблем при виборі алгоритму виявлення ключових точок стала неможливість використання стандартної реалізації алгоритму SURF, що входить до складу бібліотеки OpenCV. Алгоритм є запатентованим, тому його реалізація доступна лише в складі модуля xfeatures2d, який не постачається з основною дистрибуцією OpenCV і потребує окремого ліцензування.

Це обмеження унеможливило як використання самого алгоритму SURF у проєкті, так і пряме порівняння з його результатами.

В якості вирішення було прийнято рішення створити власну реалізацію SURF, засновану на описаних у літературі принципах: використанні інтегрального зображення, апроксимації гаусових фільтрів прямокутними фільтрами, багатомасштабному аналізі та визначенні детермінанти гессіану, створенні векторів-дескрипторів. Для верифікації коректності реалізації використовувалися результати з доступних відкритих джерел.

3.7.3 Складність налаштування масштабів і порогів детектора

Ще однією проблемою стала необхідність точного налаштування параметрів детектора ключових точок, зокрема масштабів (σ), відповідних порогів для визначення значущих максимумів у відгуках Гессе, відсотку відповідності дескрипторів та достатньої кількості точок на одному кадрі.

У теоретичних джерелах часто наводяться лише загальні рекомендації щодо вибору масштабів, але конкретні значення значною мірою залежать від характеристик вхідного відео (роздільна здатність, освітлення, швидкість руху об’єктів тощо).

Для досягнення задовільної точності та стабільності детектора була проведена серія емпіричних експериментів з підбору відповідних порогів для кожного рівня, кількості шуканих точок для одного кадру, бажаного відсотку відповідності дескрипторів. Також відповідно до логіки дії алгоритму було підібрано значень σ (1.2, 2.0, 2.8, 3.6). В результаті вдалося досягти балансу між кількістю виявлених точок, їх стабільністю та швидкістю обробки.

ВИСНОВОК ДО РОЗДІЛУ 3

У межах розділу було проведено аналіз архітектури розробленої системи спостереження за рухомим об’єктом. Детально розглянуто кожен із функціональних компонентів — від завантаження відео та попередньої обробки кадрів до побудови інтегрального зображення, реалізації багатомасштабного детектора на основі детермінанта матриці Гессе та візуалізації результатів. Наведено приклади коду для ключових модулів, що ілюструють як базові принципи роботи алгоритмів, так і специфічні аспекти реалізації.

Окрему увагу приділено практичним труднощам, що виникли під час розробки, зокрема проблемі недостатньої швидкодії, юридичним обмеженням через патентування алгоритму SURF та складності вибору параметрів детектора. Для кожної проблеми були запропоновані конкретні шляхи вирішення, які дозволили суттєво покращити ефективність системи та досягти високої продуктивності при роботі з відеоданими.

Таким чином, проведене дослідження та розробка системи засвідчили можливість побудови функціонального аналогу SURF-детектора на основі відкритих підходів з належною якістю та швидкодією, придатною для застосування в реальному часі. Отримані результати підтверджують доцільність обраної архітектури та застосованих методів оптимізації.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ ТА АНАЛІЗ РОЗРОБЛЕНОЇ СИСТЕМИ

Після завершення розробки програмної системи було проведено серію експериментів з метою перевірки її працездатності, ефективності та стійкості до різноманітних умов. У цьому розділі розглянуто методику тестування, характеристики тестового середовища, отримані результати, а також проведено їх аналіз. Основна увага приділяється оцінці швидкодії алгоритмів, точності локалізації ключових точок та стабільності відстеження в реальних відеоданих.

Отримані результати стали основою для формулювання висновків щодо доцільності застосування розробленого підходу у задачах комп'ютерного зору в режимі реального часу.

4.1 Мета та умови аналізу системи

Метою проведеного експериментального дослідження є оцінка ефективності розробленої системи виявлення та відстеження ключових точок у відеопотоці. Основними критеріями оцінювання є:

- Швидкодія — середній час обробки одного відеокадру, зокрема тривалість детектування ключових точок;
- Точність — стабільність локалізації ключових при зміні розміру об'єкта, переміщенні, повороті, при шумах, рухах камерою та інших можливих чинниках;

Для перевірки роботи системи використовувалися вибрані відеофрагменти, що містять помірні рухи камери, зміну масштабу, рух об’єкта відносно кадру, поворот.

4.1.1 Тестове середовище

Тестування розробленої системи проводилось на одноплатному комп’ютері Raspberry Pi 5 з обсягом оперативної пам’яті 8 ГБ. Обране апаратне забезпечення дозволяє оцінити ефективність реалізованих алгоритмів в умовах обмежених обчислювальних ресурсів, що є типовим сценарієм для вбудованих систем реального часу.

Операційна система — Ubuntu 24.04. Компіляцію виконано з використанням компілятора g++ 13 з увімкненою оптимізацією (-O3). Для відображення результатів використовувався HDMI-вихід, відеодані зчитувалися з локального накопичувача у форматі MP4.

4.1.2 Характеристики тестового відео

Для дослідження було використано коротке тестове відео у форматі MP4. Основні параметри відеофайлу наведено нижче:

- Тривалість: 17,66 секунди
- Роздільна здатність: 720 × 486 пікселів
- Кадрова частота: 25 кадрів на секунду

Це відео забезпечує достатню деталізацію та тривалість для тестування стабільності, швидкодії та точності роботи алгоритму в умовах реального потоку даних.

4.2 Методика випробувань

Для забезпечення об'єктивної оцінки ефективності розробленої системи було сформульовано методику випробувань, що охоплює як кількісні, так і якісні характеристики її роботи. Основна мета полягає у перевірці відповідності системи поставленим вимогам, а саме: швидкодії, точності локалізації ключових точок, стабільності супроводу та придатності до використання на обчислювально обмеженому обладнанні.

Методика включає вибір тестових сценаріїв, визначення метрик оцінювання, встановлення параметрів запуску системи та способів фіксації результатів. Усі випробування проводилися в однакових умовах для забезпечення достовірності отриманих результатів.

Одним із ключових аспектів дослідження є вивчення впливу параметрів конфігурації системи на якість та стабільність виявлення ключових точок і побудови відповідностей між кадрами. Усі основні параметри були винесені до окремого конфігураційного файлу, що забезпечує можливість централізованої модифікації умов випробувань. Це дозволяє легко адаптувати систему до різних відеоджерел та сценаріїв.

Основні параметри, що підлягали варіюванню під час експериментів:

- порогові значення для фільтра Гессе;
- допустима евклідова відстань для відповідностей;
- максимальна кількість ключових точок;
- пошук додаткових точок на масштабі $\sigma = 1.2$.

Методика проведення випробувань передбачала запуск системи на одному і тому самому відеофрагменті при зміні окремих параметрів.

Для кожного варіанту фіксувалися:

- кількість кадрів;

- кількість виявлених ключових точок;
- кількість знайдених відповідностей між кадрами;
- середній час обробки одного кадру;
- траєкторія руху центру мас;
- чи був об’єкт втрачений.

Для аналізу результатів використовувались середні значення виміряних величин.

Оскільки на етапі розробки було досліджено та емпірично підібрано коефіцієнти, які забезпечують ефективну роботу системи, то при зміні параметрів інші виставлялись у своє початкове значення, щоб оцінка будувалась на основі зміни лише одного з конфігураційних значень.

Завдяки зміні параметрів було продемонстровано залежність основних показників системи від конфігурації.

4.3 Результати та їх аналіз

У цьому підрозділі наведено узагальнені результати експериментального дослідження роботи розробленої системи. Аналіз базується на даних, отриманих під час серії випробувань з використанням відео файлу.

Результати представлені у вигляді числових показників середньої швидкодії, кількості виявлених та успішно відслідкованих ключових точок, а також візуалізацій.

4.3.1 Вплив порогового значення на ефективність системи

У цьому експерименті досліджено вплив порогового значення на швидкодію та стабільність системи. Було протестовано п’ять варіантів

параметра THRESHOLD: 100, 300, 500, 700, 1000. Зі зростанням порогу зменшується кількість слабких відповідей детермінанти, що прямо впливає на кількість виявлених ключових точок. Результати представлені в таблиці 4.1:

Таблиця 4.1 – Результати при зміні параметру порогового значення

THRESHOLD	**кількість кадрів**	**середній час обробки кадру**	**середня кількість знайдених точок**	**Середня кількість збігів**	**Чи був втрачений**
100	440	23	56	54	так
300	440	26.5	54	53	так
500	440	28	59	56	так
700	440	29	58	55	так
1000	440	29.5	57	55	ні

Графік залежності часу обробки від порогового значення продемонстрований на рис. 4.1.

Рисунок 4.1 – Залежність часу обробки він порогового значення

Зі зростанням порогу не спостерігається суттєвого зменшення кількості точок, проте лише при THRESHOLD = 1000 система змогла стабільно відстежити об’єкт протягом усього відео. Це свідчить про потребу усунення занадто слабких точок у початковій фільтрації хоч і зі збільшенням порогового значення збільшується час обробки кадру. Час відстеження поступово зростає зі збільшенням значення порогу, адже витрачається додатковий ресурс на пошук більшої кількості точок.

4.3.2 Вплив вибору масштабу для пошуку характерних точок

Наступне дослідження було присвячене впливу того, на якому масштабі виконується пошук точок для порівняння дескрипторів. Оскільки шар $\sigma = 1.2$ має всього один сусідній, то кожна значення детермінанту матриці Гессе порівнюється лише з 17 значеннями замість 26, тому точки на цьому масштабі є менш надійними. Результати показано нижче в таблиці 4.2:

Таблиця 4.2 – Результати при додаванні шару $\sigma = 1.2$

Пошук точок на шарі 1.2	**кількість кадрів**	**середній час обробки кадру**	**середня кількість знайдених точок**	**Середня кількість збігів**	**Чи був втрачений**
так	43	28.5	46	46	так
ні	440	30	59	56	ні

Хоча пошук на найменшому масштабі дещо пришвидшує обробку, система виявилась нестабільною: об’єкт втрачено вже після 43 кадрів. Оптимальним виявився вибір точок з більших масштабів ($\sigma = 2.0, 2.8$), які є стійкішими до шумів і змін. Зміна часу обробки кадру при цьому змінюється несуттєво.

4.3.3 Вплив максимальної допустимої відстані між дескрипторами

Цей параметр визначає чутливість алгоритму при зіставленні дескрипторів. Надто суворі обмеження призводять до втрати зв’язків між кадрами, тоді як занадто великі значення можуть додати хибні збіги.

Таблиця 4.3 – Результати при зміні відстані порівняння дескрипторів

Максимальна відстань порівняння дескрипторів	**кількість кадрів**	**середній час обробки кадру**	**середня кількість знайдених точок**	**Середня кількість збігів**	**Чи був втрачений**
0.4	16	25	59	7	так
0.6	20	29.5	57	20	так
0.8	440	30	59	56	ні
1	440	30	57	57	ні

Параметр за значеннями 0.8 та 1 забезпечив найкраще співвідношення — точність при збереженні максимальної кількості збігів. Менші значення надто обмежують кількість допустимих відповідностей, що призводить до втрати об’єкта. Час обробки кадру зі зміною параметра змінюється в межах похибки вимірювання.

4.3.4 Вплив обмеження максимальної кількості ключових точок

Для зменшення навантаження на систему передбачено обмеження максимальної кількості точок, які зберігаються для подальшого супроводу. Досліджено діапазон від 10 до 60 точок.

Чим більше ця кількість, тим надійнішим зазвичай є спостереження за об’єктом. Однак при занадто великих значеннях параметру час обробки кадру

збільшується, адже кожна точка окремо має бути оброблена і кількість вікон, потрібна для знаходження, також може збільшуватись.

Таблиця 4.4 – Результати при зміні максимальної кількості точок

Максимальна кількість точок	**кількість кадрів**	**середній час обробки кадру**	**середня кількість знайдених точок**	**Середня кількість збігів**	**Чи був втрачений**
10	43	12.2	8	8	так
20	43	20.5	16	15	так
30	43	23	24	23	так
40	43	26	31	30	так
50	43	28	38	37	так
60	440	30	57	55	ні

Графік залежності часу обробки одного кадру від максимальної кількості характерних точок продемонстровано на рис. 4.2.

Рисунок 4.2 – Залежність часу обробки від максимальної кількості точок

Хоча зменшення кількості ключових точок пришвидшує роботу, це відбувається за рахунок втрати стійкості. Лише при MAX_KEYPOINTS = 60 система змогла стабільно підтримувати відстеження протягом усього відео.

4.4 Візуальна оцінка роботи системи

Для візуального підтвердження працездатності розробленої системи відстеження об'єкта було збережено знімки екрану з різних етапів обробки відео.

Рисунок 4.3 – Перший кадр до обробки системою

Рисунок 4.4 – Перший кадр після знаходження точок та центру мас

Рисунок 4.5 – Збереження точок при переміщенні об’єкта

Рисунок 4.6 – Збереження точок при частковому перекритті об’єкта

Рисунок 4.7 – Збереження точок при зміні фону та зміні масштабу

Рисунок 4.8 – Безперервна траєкторія руху центру мас об’єкта

ВИСНОВОК ДО РОЗДІЛУ 4

У результаті проведеного експериментального дослідження було здійснено всебічну перевірку працездатності та ефективності розробленої системи виявлення та відстеження ключових точок у відеопотоці в умовах обмежених обчислювальних ресурсів.

Аналіз показав, що запропоноване рішення забезпечує стабільну роботу на реальних відеоданих із помірним рівнем змін положення, масштабу та руху об’єкта. У ході тестування були визначені оптимальні конфігурації параметрів, які дозволяють досягти балансу між точністю локалізації, кількістю виявлених відповідностей та швидкодією. Зокрема:

- Порогове значення відгуку матриці Гессе суттєво впливає на точність і стійкість трекінгу: при занадто низьких значеннях збільшується кількість слабких або хибних точок, що призводить до втрати об'єкта; при надто високих — зменшується загальна кількість знайдених ключових точок, що також погіршує стабільність.
- Використання першого масштабу ($\sigma = 1.2$) у процесі пошуку ключових точок може забезпечити кращу швидкодію, однак призводить до зменшення надійності трекінгу, оскільки ці точки більш чутливі до шумів і деформацій. Надійніший результат забезпечується при використанні вищих рівнів масштабу.
- Параметр максимальної допустимої відстані при порівнянні дескрипторів напряму впливає на кількість знайдених збігів. Зменшення цього значення значно знижує кількість валідних відповідностей, що може призвести до втрати об’єкта. Значення 0.8 та 1 виявилися оптимальним для забезпечення якості порівняння.

- Введення обмеження на максимальну кількість ключових точок дозволяє регулювати обчислювальне навантаження. Встановлення межі нижче 40 значно підвищує швидкодію, але при цьому порушує стабільність трекінгу. Повноцінне відстеження забезпечується за наявності не менше ніж 60 точок.

Результати дослідження також підтверджуються візуально — на знімках екрану видно чітке збереження траєкторії об’єкта в кадрі, за умови використання узгоджених параметрів.

Середній час обробки одного кадру на одноплатному комп’ютері Raspberry Pi 5 дозволяє здійснювати обробку відеопотоку в режимі реального часу. Це свідчить про високу ефективність реалізованого підходу навіть в умовах обмежених ресурсів.

Таким чином, проведене тестування підтвердило працездатність розробленої системи в умовах реального відеопотоку та дозволило визначити набір оптимальних параметрів, за яких забезпечується задовільний компроміс між продуктивністю, точністю та стійкістю до змін середовища.

ВИСНОВКИ

У результаті виконання дипломного проєкту було розроблено, реалізовано та досліджено ефективність високопродуктивної системи відстеження об'єктів у відеопотоці, побудованої на базі алгоритму SURF з урахуванням обмежень вбудованих систем.

У першому розділі проведено огляд існуючих методів слідкування за об’єктами. Детально проаналізовано класичні алгоритми та сучасні нейромережеві підходи. Було обґрунтовано вибір підходу, заснованого на визаченні ознак, для досягнення балансу між точністю та обчислювальною складністю, що особливо важливо для систем реального часу.

У другому розділі детально описано математичну суть алгоритму SURF та принцип його реалізації, включаючи побудову інтегрального зображення, виявлення характерних точок на основі детермінанту матриці Гессе, обчислення дескрипторів і механізму порівняння. Окремо розглянуто оптимізаційні аспекти та адаптацію до обмежень апаратної платформи, включаючи зменшення кількості обчислень і відмову від надмірних бібліотек.

У третьому розділі розглянуто архітектуру програмної системи, принципи взаємодії модулів, а також підходи до обробки відеопотоку в режимі реального часу. Значна увага приділена розробці власного оптимізованого детектора ключових точок, дескрипторів, модулю фільтрації та засобів візуалізації. Наведено приклади вирішення технічних труднощів, що виникали у процесі реалізації системи та шляхи їх вирішення.

У четвертому розділі проведено експериментальну перевірку розробленої системи, оцінено її ефективність за різних параметрів: порогових значень, масштабу виявлення ознак, максимальної кількості точок та допустимої

відстані між дескрипторами. Продемонстровано вплив кожного параметра на точність, швидкодію та стабільність трекінгу. Показано, що при оптимальних налаштуваннях середній час обробки кадру оптимальний для системи реального часу.

Таким чином, поставлені у роботі цілі досягнуто: реалізовано систему слідкування, що є ефективною, стабільною, належно оптимізованою та придатною до використання у вбудованих пристроях з обмеженими обчислювальними ресурсами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Yilmaz A., Javed O., Shah M. Object tracking: A survey // ACM Computing Surveys. — 2006. — Vol. 38, No. 4. — P. 13–es. — DOI: https://doi.org/10.1145/1177352.1177355.
2. Bay H., Tuytelaars T., Van Gool L. SURF: Speeded Up Robust Features // Computer Vision – ECCV 2006. — Springer, 2006. — P. 404–417. — DOI: https://doi.org/10.1007/11744023_32.
3. Bochkovskiy A., Wang C.-Y., Liao H.-Y. M. YOLOv4: Optimal Speed and Accuracy of Object Detection [Електронний ресурс] // arXiv:2004.10934. — 2020. — Режим доступу: https://arxiv.org/abs/2004.10934.
4. Bertinetto L., Valmadre J., Henriques J. F., Vedaldi A., Torr P. H. S. Fully-Convolutional Siamese Networks for Object Tracking // ECCV Workshops. — 2016. — P. 850–865. — DOI: https://doi.org/10.1007/978-3-319-48881-3_56.
5. Bradski G., Kaehler A. Learning OpenCV 4: Computer Vision with Python. — 2nd ed. — Sebastopol: O'Reilly Media, 2019. — 576 p.
6. Rublee E., Rabaud V., Konolige K., Bradski G. ORB: An Efficient Alternative to SIFT or SURF // Proceedings of the IEEE International Conference on Computer Vision (ICCV). — 2011. — P. 2564–2571. — DOI: https://doi.org/10.1109/ICCV.2011.6126544.
7. Lucas B. D., Kanade T. An Iterative Image Registration Technique with an Application to Stereo Vision // Proceedings of the 7th International Joint Conference on Artificial Intelligence (IJCAI). — 1981. — P. 674–679. — [Електронний ресурс]. — Режим доступу: https://www.ijcai.org/Proceedings/81-2/Papers/017.pdf. — Дата звернення: 20.05.2025.

8. Stroustrup B. The C++ Programming Language. — 4th ed. — Boston: Addison-Wesley, 2013. — 1376 p.
9. Using OpenCV with C++ [Електронний ресурс] // OpenCV Documentation. — Режим доступу: https://docs.opencv.org/master/d7/d9f/tutorial_linux_install.html. — Дата звернення: 20.05.2025.
10. OpenCV Documentation: API Reference and Tutorials [Електронний ресурс]. — Режим доступу: https://docs.opencv.org/master. — Дата звернення: 20.05.2025.
11. Visual Studio Code – Code Editing. Redefined [Електронний ресурс] / Microsoft. — Режим доступу: https://code.visualstudio.com. — Дата звернення: 20.05.2025.
12. CMake Documentation [Електронний ресурс]. — Режим доступу: https://cmake.org/documentation/. — Дата звернення: 20.05.2025.
13. Cheon, S. H., Eom, I. K., & Moon, Y. H. (2016). Fast descriptor extraction method for a SURF-based interest point. Electronics Letters, 52(4), 274-275. https://doi.org/10.1049/el.2015.3055

ДОДАТОК 1

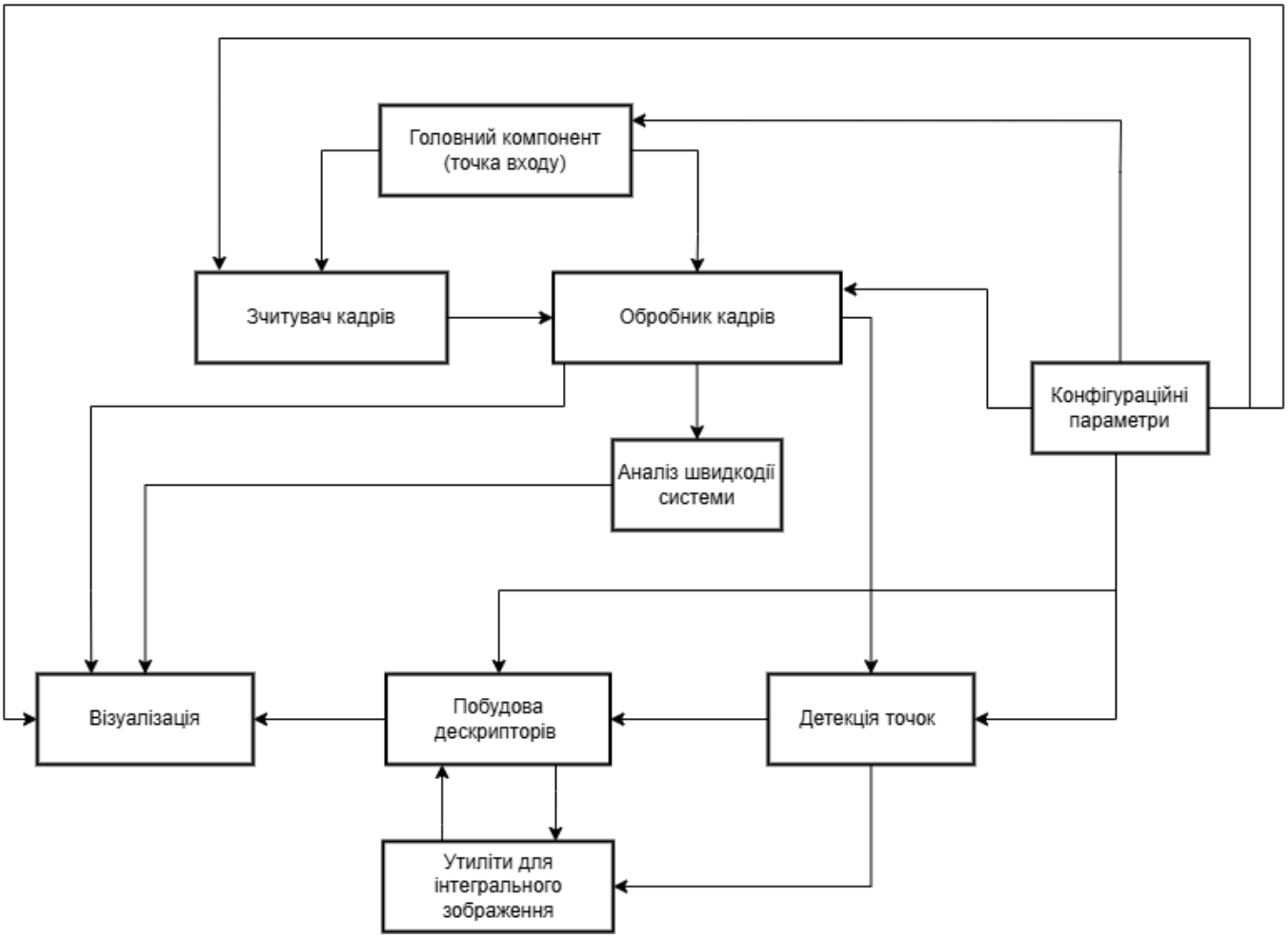
Система слідкування за рухомим об’єктом

Структурна схема системи

ІАЛЦ.467200.004 Д1

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.004 Д1		
		№ докум.	Підпис	Дата			
Розробив		Чирков М. К.			**Система слідкування за рухомим об'єктом** Структурна схема системи	Літ.	Аркуш 1 / Аркушів 1
Перевірив		Сергієнко А.М.					
						КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-11	
Н. Контр.		Пономаренко А. М.					
Затвердив							

ДОДАТОК 2

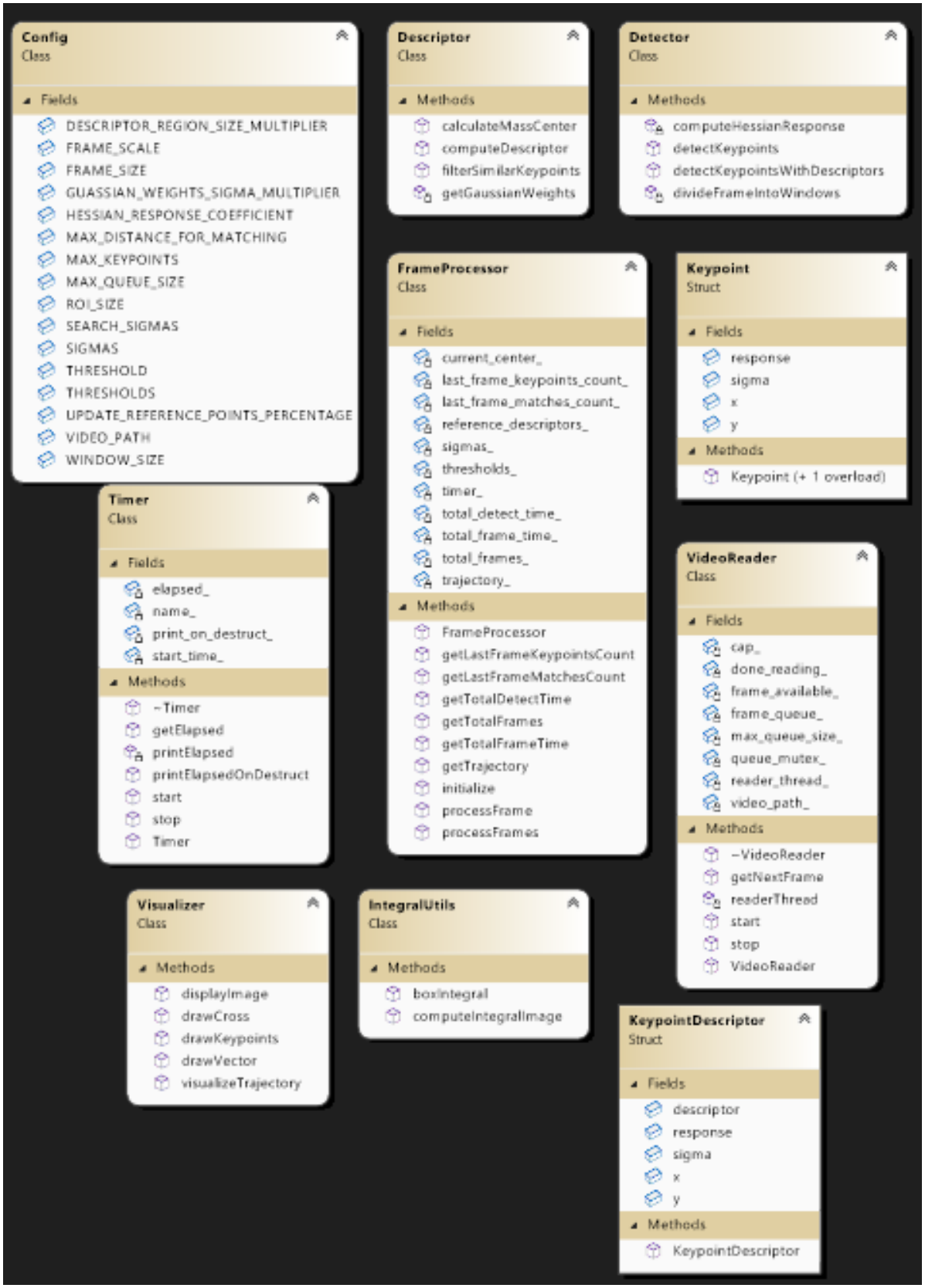
Система слідкування за рухомим об’єктом

Функціональна схема (діаграма класів)

ІАЛЦ.467200.005 Д2

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.005 Д2		
		№ докум.	Підпис	Дата			
Розробив		Чирков М. К.			Система слідкування за рухомим об'єктом	Літ.	Аркуш	Аркушів
Перевірив		Сергієнко А. М.					1	1
					Функціональна схема (діаграма класів)	КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-11		
Н. Контр.		Пономаренко А. М.						
Затвердив								

ДОДАТОК 3

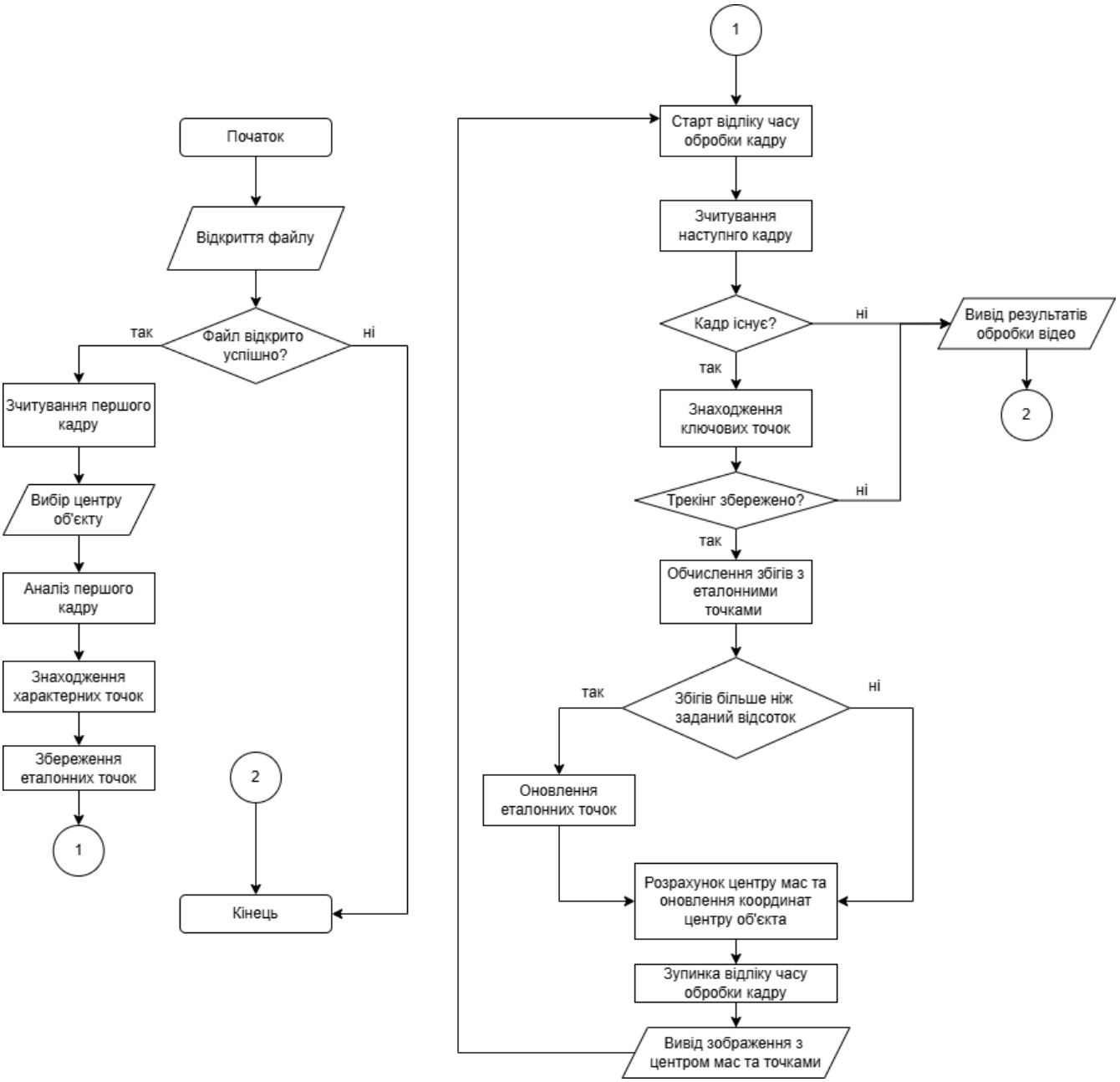
Система слідкування за рухомим об’єктом

Алгоритм дій програмного забезпечення

ІАЛЦ.467200.006 Д3

Аркушів 1

Київ 2025 р



					ІАЛЦ.467200.006 Д3	
		№ докум.	Підпис	Дата		
Розробив		Чирков М. К.			**Система слідкування за рухомим об'єктом** Алгоритм дій програмного забезпечення	Літ. / Аркуш 1 / Аркушів 1
Перевірив		Сергієнко А. М.				
Н. Контр.		Пономаренко А. М.				КПІ ім. Ігоря Сікорського, ФІОТ,ІМ-11
Затвердив						

ДОДАТОК 4

Система слідкування за рухомим об’єктом

Текст програмного коду

ІАЛЦ.467200.007 Д4

Аркушів 21

Київ 2025 р

```
#include <opencv2/opencv.hpp>
#include <iostream>
#include <thread>
#include "Timer.h"
#include "Detector.h"
#include "IntegralUtils.h"
#include "Visualizer.h"
#include "Descriptor.h"
#include "VideoReader.h"
#include "FrameProcessor.h"
#include "Config.h"

cv::Point selected_point(-1, -1);
bool point_selected = false;

void onMouse(int event, int x, int y, int, void*) {
    if (event == cv::EVENT_LBUTTONDOWN && !point_selected) {
        selected_point = cv::Point(x, y);
        point_selected = true;
        std::cout << "Selected point: (" << x << ", " << y << ")\n";
    }
}

bool initializeTracking(VideoReader& video_reader, cv::Mat& first_frame, cv::Point& selected_point, const char*
window_name) {
    if (!video_reader.start()) {
        std::cerr << "Failed to open video.\n";
        return false;
    }

    if (!video_reader.getNextFrame(first_frame)) {
        std::cerr << "First frame is empty.\n";
        return false;
    }

    cv::namedWindow(window_name);
    cv::setMouseCallback(window_name, onMouse);

    while (true) {
```

```
        cv::Mat display;
        cv::cvtColor(first_frame, display, cv::COLOR_GRAY2BGR);
        if (point_selected)
            Visualizer::drawCross(display, selected_point, cv::Scalar(0, 255, 0), 5, 2);

        Visualizer::displayImage(display, window_name);
        if (point_selected) break;
        int key = cv::waitKey(30);
        if (key == 27) return false;
    }

    return true;
}

int main() {
    try {
        cv::setNumThreads(cv::getNumberOfCPUs());

        const char* window_name = "Tracking";
        VideoReader video_reader(Config::VIDEO_PATH, Config::MAX_QUEUE_SIZE);

        cv::Mat first_frame;
        if (!initializeTracking(video_reader, first_frame, selected_point, window_name)) {
            return 1;
        }

        std::vector<float> sigmas(Config::SIGMAS.begin(), Config::SIGMAS.end());
        std::vector<float> thresholds(Config::THRESHOLDS.begin(), Config::THRESHOLDS.end());

        FrameProcessor processor(sigmas, thresholds);
        processor.initialize(first_frame, selected_point);

        FrameProcessor::processFrames(video_reader, processor, window_name);

        video_reader.stop();
    }
    catch (cv::Exception& e) {
        std::cerr << "OpenCV Exception: " << e.what() << std::endl;
    }
    catch (std::exception& e) {
        std::cerr << "Standard Exception: " << e.what() << std::endl;
    }
    catch (...) {
        std::cerr << "Unknown exception!\n";
```

```
    }
    return 0;
}
```

```
#pragma once

#include <vector>
#include <opencv2/opencv.hpp>

struct KeypointDescriptor {
    int x;
    int y;
    float sigma;
    float response;
    std::vector<float> descriptor;
    KeypointDescriptor(int x_, int y_, float sigma_, float response_, const std::vector<float>& desc)
        : x(x_), y(y_), sigma(sigma_), response(response_), descriptor(desc) {}
};

class Descriptor {
public:
    static std::vector<float> computeDescriptor(const cv::Mat& integral, float x, float y, float scale);

    static std::vector<KeypointDescriptor> filterSimilarKeypoints(
        const std::vector<KeypointDescriptor>& current,
        const std::vector<KeypointDescriptor>& reference,
        float max_distance = 0.8f
    );

    static bool calculateMassCenter(const std::vector<KeypointDescriptor>& descriptors, cv::Point& center);

private:
    static cv::Mat getGaussianWeights(float scale, int grid_size, int region_size, int sample_count);
};
#include "Descriptor.h"
#include "IntegralUtils.h"
#include <cmath>
#include <iostream>
#include <opencv2/opencv.hpp>
#include "Config.h"

cv::Mat Descriptor::getGaussianWeights(float scale, int grid_size, int region_size, int sample_count) {
    int step = region_size / grid_size;
    float sigma = Config::GUASSIAN_WEIGHTS_SIGMA_MULTIPLIER * scale;
```

```
    float sample_step = static_cast<float>(step) / sample_count;

    cv::Mat weights(sample_count, sample_count, CV_32F);

    for (int i = 0; i < sample_count; ++i) {
        for (int j = 0; j < sample_count; ++j) {
            float x = (j - (sample_count - 1) / 2.0f) * sample_step;
            float y = (i - (sample_count - 1) / 2.0f) * sample_step;
            float r2 = x * x + y * y;

            weights.at<float>(i, j) = std::exp(-(r2) / (2.0f * sigma * sigma));
        }
    }

    return weights;
}

std::vector<float> Descriptor::computeDescriptor(const cv::Mat& integral, float x, float y, float scale) {
    const int sample_count = 5;
    const int grid_size = 4;
    const int region_size = static_cast<int>(Config::DESCRIPTOR_REGION_SIZE_MULTIPLIER * scale);
    const int half_region = region_size / 2;
    const int step = region_size / grid_size;
    const float sample_step = static_cast<float>(step) / sample_count;
    const float wavelet_size = 2.0f * scale;

    cv::Mat weights = Descriptor::getGaussianWeights(scale, grid_size, region_size, sample_count);

    std::vector<float> descriptor(grid_size * grid_size * 4, 0.0f);

    for (int i = 0; i < grid_size; ++i) {
        for (int j = 0; j < grid_size; ++j) {
            float dx_sum = 0.0f;
            float dy_sum = 0.0f;
            float dx_sum_abs = 0.0f;
            float dy_sum_abs = 0.0f;

            for (int u = 0; u < sample_count; ++u) {
                for (int v = 0; v < sample_count; ++v) {
                    float sample_x = x - half_region + j * step + (v + 0.5f) * sample_step;
                    float sample_y = y - half_region + i * step + (u + 0.5f) * sample_step;

                    int top = static_cast<int>(sample_y - wavelet_size / 2);
                    int bottom = static_cast<int>(sample_y + wavelet_size / 2);
```

```
                int left = static_cast<int>(sample_x - wavelet_size / 2);
                int right = static_cast<int>(sample_x + wavelet_size / 2);

                float dx = IntegralUtils::boxIntegral(integral, top, static_cast<int>(sample_x), bottom, right) -
                        IntegralUtils::boxIntegral(integral, top, left, bottom, static_cast<int>(sample_x));

                float dy = IntegralUtils::boxIntegral(integral, static_cast<int>(sample_y), left, bottom, right) -
                        IntegralUtils::boxIntegral(integral, top, left, static_cast<int>(sample_y), right);

                float weight = weights.at<float>(u, v);
                dx_sum += dx * weight;
                dy_sum += dy * weight;
                dx_sum_abs += std::abs(dx) * weight;
                dy_sum_abs += std::abs(dy) * weight;
            }
        }

        int idx = 4 * (i * grid_size + j);

        descriptor[idx] = dx_sum;
        descriptor[idx + 1] = dy_sum;
        descriptor[idx + 2] = dx_sum_abs;
        descriptor[idx + 3] = dy_sum_abs;
    }
}

float norm = 0.0f;
for (float val : descriptor) norm += val * val;
norm = std::sqrt(norm);
if (norm > 0.0f) {
    for (float& val : descriptor) val /= norm;
}

return descriptor;
}

std::vector<KeypointDescriptor> Descriptor::filterSimilarKeypoints(
    const std::vector<KeypointDescriptor>& current,
    const std::vector<KeypointDescriptor>& reference,
    float max_distance
) {
    std::vector<KeypointDescriptor> filtered;

    for (const auto& kp : current) {
```

```
            float best_dist_sq = std::numeric_limits<float>::max();
            for (const auto& ref_kp : reference) {
                if (kp.descriptor.size() != ref_kp.descriptor.size()) continue;

                float dist_sq = 0.0f;
                const float* a = kp.descriptor.data();
                const float* b = ref_kp.descriptor.data();
                for (size_t i = 0; i < kp.descriptor.size(); ++i) {
                    float diff = a[i] - b[i];
                    dist_sq += diff * diff;
                    if (dist_sq > max_distance * max_distance) break;
                }
                if (dist_sq < max_distance * max_distance) {
                    filtered.push_back(kp);
                    break;
                }
            }
        }

        return filtered;
}

bool Descriptor::calculateMassCenter(const std::vector<KeypointDescriptor>& descriptors, cv::Point& center) {
    double total_weight = 0.0;
    double x_sum = 0.0, y_sum = 0.0;

    for (const auto& ds : descriptors) {
        double response = ds.response;
        total_weight += response;
        x_sum += ds.x * response;
        y_sum += ds.y * response;
    }

    if (total_weight == 0.0) {
        return false;
    }

    center.x = static_cast<int>(x_sum / total_weight);
    center.y = static_cast<int>(y_sum / total_weight);
    return true;
}
#pragma once
#include <opencv2/core.hpp>
#include <vector>
```

```
#include "Descriptor.h"

struct Keypoint {
    int x;
    int y;
    float sigma;
    float response;
    Keypoint() : x(0), y(0), sigma(0.0f), response(0.0f) {}
    Keypoint(int x_, int y_, float sigma_, float response_) : x(x_), y(y_), sigma(sigma_), response(response_) {}
};

class Detector {
public:
    static std::vector<Keypoint> detectKeypoints(const cv::Mat& img, cv::Point center,
        const std::vector<float>& sigmas,
        const std::vector<float>& thresholds,
        int max_keypoints = 50);

    static std::vector<KeypointDescriptor> detectKeypointsWithDescriptors(const cv::Mat& img, cv::Point center,
        const std::vector<float>& sigmas,
        const std::vector<float>& thresholds);

private:
    static float computeHessianResponse(const cv::Mat& ii, int y, int x, float sigma, float threshold);

    static std::vector<cv::Rect> divideFrameIntoWindows(const cv::Point& center);
};

#include "Detector.h"
#include "IntegralUtils.h"
#include "Descriptor.h"
#include <opencv2/imgproc.hpp>
#include <cmath>
#include <algorithm>
#include "Timer.h"
#include <unordered_map>
#include <array>
#include "Config.h"

float Detector::computeHessianResponse(const cv::Mat& ii, int y, int x, float sigma, float threshold) {
    int filter_size = static_cast<int>(std::round(7.5f * sigma));
    int half_size = filter_size / 2;
    int third_size = filter_size / 3;
```

```
    int top = y - half_size - 1;
    int bottom = y + half_size;
    int left = x - half_size;
    int right = x + half_size + 1;

    int mid_top = y - third_size / 2 - 1;
    int mid_bottom = y + third_size / 2;
    int mid_left = x - third_size / 2;
    int mid_right = x + third_size / 2 + 1;

    float left_third = IntegralUtils::boxIntegral(ii, top, left, bottom, mid_left);
    float horizontal_center_third = IntegralUtils::boxIntegral(ii, top, mid_left, bottom, mid_right);
    float right_third = IntegralUtils::boxIntegral(ii, top, mid_right, bottom, right);

    float top_third = IntegralUtils::boxIntegral(ii, top, left, mid_top, right);
    float vertical_center_third = IntegralUtils::boxIntegral(ii, mid_top, left, mid_bottom, right);
    float bottom_third = IntegralUtils::boxIntegral(ii, mid_bottom, left, bottom, right);

    int top_square = y - half_size;
    int right_square = x + half_size;

    float top_left = IntegralUtils::boxIntegral(ii, top_square, left, y, x);
    float top_right = IntegralUtils::boxIntegral(ii, top_square, x, y, right_square);
    float bottom_left = IntegralUtils::boxIntegral(ii, y, left, bottom, x);
    float bottom_right = IntegralUtils::boxIntegral(ii, y, x, bottom, right_square);

    // Calculate Dxx, Dyy, and Dxy
    float Dxx = left_third + right_third - 2 * horizontal_center_third;
    float Dyy = top_third + bottom_third - 2 * vertical_center_third;
    float Dxy = top_left + bottom_right - top_right - bottom_left;

    float inv_area = 1.0f / (filter_size * filter_size);

    float weight = Config::HESSIAN_RESPONSE_COEFFICIENT;
    float determinant = (Dxx * Dyy - weight * Dxy * Dxy) * inv_area;

    return determinant >= threshold ? determinant : 0.0f;
}

std::vector<Keypoint> Detector::detectKeypoints(const cv::Mat& ii, cv::Point center,
    const std::vector<float>& sigmas,
    const std::vector<float>& thresholds,
    int max_keypoints) {
    std::vector<Keypoint> result;
```

```
std::vector<cv::Rect> windows = divideFrameIntoWindows(center);
cv::Mat kernel = cv::getStructuringElement(cv::MORPH_RECT, { 3, 3 });
std::vector<float> search_sigmas(Config::SEARCH_SIGMAS.begin(), Config::SEARCH_SIGMAS.end());

for (const auto& win : windows) {
    std::vector<cv::Mat> responses(sigmas.size());
    for (int i = 0; i < sigmas.size(); ++i) {
        responses[i] = cv::Mat::zeros(win.height, win.width, CV_32F);
        for (int y = 0; y < win.height; ++y) {
            for (int x = 0; x < win.width; ++x) {
                float r = computeHessianResponse(ii, win.y + y, win.x + x, sigmas[i], thresholds[i]);
                responses[i].at<float>(y, x) = r;
            }
        }
    }

    for (size_t i = 0; i < sigmas.size(); ++i) {
        if (std::find(search_sigmas.begin(), search_sigmas.end(), sigmas[i]) == search_sigmas.end()) {
            continue;
        }

        cv::Mat mask = cv::Mat::zeros(responses[i].size(), CV_8U);

        for (int y = 1; y < responses[i].rows - 1; ++y) {
            for (int x = 1; x < responses[i].cols - 1; ++x) {
                float r = responses[i].at<float>(y, x);

                bool is_local_max = true;
                for (int dy = -1; dy <= 1; ++dy) {
                    for (int dx = -1; dx <= 1; ++dx) {
                        if (dy == 0 && dx == 0) continue;
                        if (r <= responses[i].at<float>(y + dy, x + dx)) {
                            is_local_max = false;
                            break;
                        }
                    }
                    if (!is_local_max) break;
                }

                if (is_local_max && r > thresholds[i]) {
                    mask.at<uchar>(y, x) = 255;
                }
            }
        }
```

```
            if (i > 0) {
                mask &= responses[i] > responses[i - 1];
            }
            if (i < sigmas.size() - 1) {
                mask &= responses[i] > responses[i + 1];
            }

            for (int y = 0; y < mask.rows; ++y) {
                for (int x = 0; x < mask.cols; ++x) {
                    if (mask.at<uchar>(y, x)) {
                        float r = responses[i].at<float>(y, x);
                        result.emplace_back(win.x + x, win.y + y, sigmas[i], r);
                    }
                }
            }
        }

        if (result.size() > static_cast<size_t>(max_keypoints)) {
            std::sort(result.begin(), result.end(), [](const Keypoint& a, const Keypoint& b) {
                return a.response > b.response;
            });
            result.resize(max_keypoints);
            return result;
        }

    }

    return result;
}

std::vector<KeypointDescriptor> Detector::detectKeypointsWithDescriptors(const cv::Mat& img, cv::Point center,
    const std::vector<float>& sigmas,
    const std::vector<float>& thresholds) {
    cv::Mat ii = IntegralUtils::computeIntegralImage(img);

    auto kps = detectKeypoints(ii, center, sigmas, thresholds, Config::MAX_KEYPOINTS);

    std::vector<KeypointDescriptor> descriptors;

    for (const auto& kp : kps) {
        auto desc = Descriptor::computeDescriptor(ii, kp.x, kp.y, kp.sigma);
        descriptors.emplace_back(kp.x, kp.y, kp.sigma, kp.response, desc);
    }
```

```
    return descriptors;
}

std::vector<cv::Rect> Detector::divideFrameIntoWindows(const cv::Point& center) {
    std::vector<cv::Rect> windows;
    const int window_size = Config::WINDOW_SIZE;
    const int frame_size = Config::FRAME_SIZE;
    const int half_frame = frame_size / 2;

    const int num_windows = 3;
    const int overlap = (num_windows * window_size - frame_size) / (num_windows - 1);

    int frame_top_left_x = center.x - half_frame;
    int frame_top_left_y = center.y - half_frame;

    // Offsets for 9 windows (dy, dx) in required order
    const std::vector<std::pair<int, int>> offsets = {
        {0, 0},    // Center
        {0, -1},   // Left
        {0, 1},    // Right
        {-1, 0},   // Top
        {1, 0},    // Bottom
        {1, -1},   // Bottom-left
        {1, 1},    // Bottom-right
        {-1, -1},  // Top-left
        {-1, 1}    // Top-right
    };

    for (const auto& offset : offsets) {
        int dx = offset.second * (window_size - overlap);
        int dy = offset.first * (window_size - overlap);

        windows.emplace_back(
            frame_top_left_x + half_frame + dx - window_size / 2,
            frame_top_left_y + half_frame + dy - window_size / 2,
            window_size,
            window_size
        );
    }

    return windows;
}
```

```
#pragma once

#include <opencv2/opencv.hpp>
#include <iostream>
#include <thread>
#include "Timer.h"
#include "Detector.h"
#include "IntegralUtils.h"
#include "Visualizer.h"
#include "Descriptor.h"
#include "VideoReader.h"
#include "Config.h"

class FrameProcessor {
public:
    FrameProcessor(const std::vector<float>& sigmas, const std::vector<float>& thresholds);

    void initialize(const cv::Mat& first_frame, const cv::Point& selected_point);
    bool processFrame(const cv::Mat& frame, cv::Mat& output, double& detect_time);

    int getTotalFrames() const { return total_frames_; }
    double getTotalFrameTime() const { return total_frame_time_; }
    double getTotalDetectTime() const { return total_detect_time_; }

    int getLastFrameKeypointsCount() const { return last_frame_keypoints_count_; }
    int getLastFrameMatchesCount() const { return last_frame_matches_count_; }

    const std::vector<cv::Point>& getTrajectory() const { return trajectory_; }

    static void processFrames(VideoReader& video_reader, FrameProcessor& processor, const char* window_name);

private:
    std::vector<float> sigmas_;
    std::vector<float> thresholds_;
    std::vector<KeypointDescriptor> reference_descriptors_;
    cv::Point current_center_;
    Timer timer_;

    int total_frames_;
    double total_frame_time_;
    double total_detect_time_;

    int last_frame_keypoints_count_;
    int last_frame_matches_count_;
```

```
    std::vector<cv::Point> trajectory_;
};
#include "FrameProcessor.h"
#include <opencv2/opencv.hpp>
#include <iostream>
#include <thread>
#include "Timer.h"
#include "Detector.h"
#include "IntegralUtils.h"
#include "Visualizer.h"
#include "Descriptor.h"
#include "VideoReader.h"
#include "Config.h"

FrameProcessor::FrameProcessor(const std::vector<float>& sigmas, const std::vector<float>& thresholds)
    : sigmas_(sigmas), thresholds_(thresholds), total_frames_(0), total_frame_time_(0.0), total_detect_time_(0.0),
      last_frame_keypoints_count_(0), last_frame_matches_count_(0) {}

void FrameProcessor::initialize(const cv::Mat& first_frame, const cv::Point& selected_point) {
    current_center_ = selected_point;
        reference_descriptors_ = Detector::detectKeypointsWithDescriptors(first_frame, current_center_, sigmas_,
thresholds_);
}

void FrameProcessor::processFrames(VideoReader& video_reader, FrameProcessor& processor, const char*
window_name) {
    cv::Mat frame;

    int total_keypoints_detected = 0;
    int total_matches_found = 0;

    while (true) {
        if (!video_reader.getNextFrame(frame)) {
            break;
        }

        cv::Mat output;
        double detect_time;
        if (!processor.processFrame(frame, output, detect_time)) {
            break;
        }

        total_keypoints_detected += processor.getLastFrameKeypointsCount();
```

```
            total_matches_found += processor.getLastFrameMatchesCount();

            Visualizer::displayImage(output, window_name);
            int key = cv::waitKey(1);
            if (key == 27 || key == 'q') {
                break;
            }
        }

        if (processor.getTotalFrames() > 0) {
            std::cout << "=== Average Timings (s) ===\n";
            std::cout << "Frames processed: " << processor.getTotalFrames() << "\n";
             std::cout << "Average frame time: " << (processor.getTotalFrameTime() / processor.getTotalFrames()) * 1000
<< " ms\n";
            std::cout << "=== Tracking Statistics ===\n";
            std::cout << "Avarage keypoints detected: " << total_keypoints_detected / processor.getTotalFrames() << "\n";
            std::cout << "Avarage matches found: " << total_matches_found / processor.getTotalFrames() << "\n";

            Visualizer::visualizeTrajectory(processor.getTrajectory(), frame.cols, frame.rows);
        } else {
            std::cout << "No frames processed.\n";
        }
    }

    bool FrameProcessor::processFrame(const cv::Mat& frame, cv::Mat& output, double& detect_time) {
        timer_.start();

        cv::Point frame_center(frame.cols / 2, frame.rows / 2);
        cv::Mat gray = frame;

        int roi_size = Config::ROI_SIZE;

        cv::Rect roi(current_center_.x - roi_size / 2, current_center_.y - roi_size / 2, roi_size, roi_size);
        cv::Mat patch(roi_size, roi_size, gray.type(), cv::Scalar(128));
        cv::Rect img_rect(0, 0, gray.cols, gray.rows);
        cv::Rect roi_in_img = roi & img_rect;

        if (roi_in_img.area() == roi_size * roi_size) {
            patch = gray(roi).clone();
        } else {
            patch.setTo(128);
            int dx = roi_in_img.x - roi.x;
            int dy = roi_in_img.y - roi.y;
            gray(roi_in_img).copyTo(patch(cv::Rect(dx, dy, roi_in_img.width, roi_in_img.height)));
```

```
        }

        cv::Point center_in_roi = current_center_ - roi.tl();

        auto patch_descriptors = Detector::detectKeypointsWithDescriptors(patch, center_in_roi, sigmas_, thresholds_);

        last_frame_keypoints_count_ = static_cast<int>(patch_descriptors.size());

        if (patch_descriptors.empty()) {
            std::cout << "The object was lost!" << std::endl;
            return false;
        }

        for (auto& d : patch_descriptors) {
            d.x += roi.x;
            d.y += roi.y;
        }

            auto descriptors = Descriptor::filterSimilarKeypoints(patch_descriptors, reference_descriptors_,
Config::MAX_DISTANCE_FOR_MATCHING);

        last_frame_matches_count_ = static_cast<int>(descriptors.size());

        if (descriptors.empty()) {
            std::cout << "The object was lost!" << std::endl;
            return false;
        }

                    if (descriptors.size() > reference_descriptors_.size() *
Config::UPDATE_REFERENCE_POINTS_PERCENTAGE) {
            reference_descriptors_ = descriptors;
        }

        cv::Point new_center;
        if (Descriptor::calculateMassCenter(descriptors, new_center)) {
            current_center_ = new_center;
            trajectory_.push_back(current_center_);
            Visualizer::drawCross(gray, current_center_, cv::Scalar(255, 255, 255), 20);
        }

        cv::Point vector_to_center = frame_center - current_center_;
        // Visualizer::drawVector(gray, frame_center, current_center_, cv::Scalar(0, 0, 255), 2);

        detect_time = timer_.getElapsed();
```

```
    Visualizer::drawKeypoints(gray, descriptors);
    output = gray.clone();

    total_frames_++;
    total_frame_time_ += timer_.stop();
    total_detect_time_ += detect_time;

    return true;
}
#pragma once

#include <opencv2/opencv.hpp>

class IntegralUtils {
public:
    static cv::Mat computeIntegralImage(const cv::Mat& img);

    static float boxIntegral(const cv::Mat& integral, int top, int left, int bottom, int right);
};

#include "IntegralUtils.h"
#include <opencv2/imgproc.hpp>
#include <algorithm>

cv::Mat IntegralUtils::computeIntegralImage(const cv::Mat& img) {
    cv::Mat integral;
    cv::integral(img, integral, CV_32F);
    return integral;
}

float IntegralUtils::boxIntegral(const cv::Mat& integral, int top, int left, int bottom, int right) {
    float A = integral.at<float>(top, left);
    float B = integral.at<float>(top, right);
    float C = integral.at<float>(bottom, left);
    float D = integral.at<float>(bottom, right);

    return std::max(0.0f, D - B - C + A);
}

#pragma once

#include <chrono>
#include <string>
```

```
#include <optional>

class Timer {
public:
    explicit Timer(const std::string& name = "");

    void start();
    double stop();
    [[nodiscard]] double getElapsed() const;

    void printElapsedOnDestruct(bool enable = true);

    ~Timer();

private:
    std::string name_;
    std::chrono::high_resolution_clock::time_point start_time_;
    std::optional<double> elapsed_;
    bool print_on_destruct_;

    void printElapsed() const;
};

#include "Timer.h"
#include <iostream>

Timer::Timer(const std::string& name)
    : name_(name), print_on_destruct_(false) {}

void Timer::start() {
    start_time_ = std::chrono::high_resolution_clock::now();
    elapsed_.reset();
}

double Timer::stop() {
    const auto end_time = std::chrono::high_resolution_clock::now();
    elapsed_ = std::chrono::duration<double>(end_time - start_time_).count();
    return *elapsed_;
}

double Timer::getElapsed() const {
    const auto now = std::chrono::high_resolution_clock::now();
    return std::chrono::duration<double>(now - start_time_).count();
}
```

```
void Timer::printElapsedOnDestruct(bool enable) {
    print_on_destruct_ = enable;
}

void Timer::printElapsed() const {
    if (elapsed_.has_value()) {
        if (!name_.empty())
            std::cout << "[" << name_ << "] Elapsed time: " << *elapsed_ << " seconds\n";
        else
            std::cout << "Elapsed time: " << *elapsed_ << " seconds\n";
    }
}

Timer::~Timer() {
    if (print_on_destruct_) {
        printElapsed();
    }
}

#pragma once
#include <opencv2/opencv.hpp>
#include <queue>
#include <mutex>
#include <condition_variable>
#include <atomic>

class VideoReader {
public:
    explicit VideoReader(const std::string& video_path, int max_queue_size = 20);
    ~VideoReader();

    bool start();
    void stop();
    bool getNextFrame(cv::Mat& frame);

private:
    void readerThread();

    std::string video_path_;
    int max_queue_size_;
    std::queue<cv::Mat> frame_queue_;
    std::mutex queue_mutex_;
    std::condition_variable frame_available_;
```

```
    std::atomic<bool> done_reading_;
    std::thread reader_thread_;
    cv::VideoCapture cap_;
};
#include "VideoReader.h"
#include "Config.h"

VideoReader::VideoReader(const std::string& video_path, int max_queue_size)
    : video_path_(video_path), max_queue_size_(max_queue_size), done_reading_(false) {}

VideoReader::~VideoReader() {
    stop();
}

bool VideoReader::start() {
    cap_.open(video_path_);
    if (!cap_.isOpened()) {
        return false;
    }
    done_reading_ = false;
    reader_thread_ = std::thread(&VideoReader::readerThread, this);
    return true;
}

void VideoReader::stop() {
    done_reading_ = true;
    frame_available_.notify_all();
    if (reader_thread_.joinable()) {
        reader_thread_.join();
    }
    cap_.release();
}

bool VideoReader::getNextFrame(cv::Mat& frame) {
    std::unique_lock<std::mutex> lock(queue_mutex_);
    frame_available_.wait(lock, [this] { return !frame_queue_.empty() || done_reading_; });
    if (frame_queue_.empty() && done_reading_) {
        return false;
    }
    frame = frame_queue_.front();
    frame_queue_.pop();
    frame_available_.notify_one();
    return true;
}
```

```
void VideoReader::readerThread() {
    while (!done_reading_) {
        cv::Mat frame;
        if (!cap_.read(frame)) {
            break;
        }

                // cv::resize(frame, frame, cv::Size(frame.cols * Config::FRAME_SCALE, frame.rows * Config::FRAME_SCALE));

        cv::Mat gray;
        cv::cvtColor(frame, gray, cv::COLOR_BGR2GRAY);

        std::unique_lock<std::mutex> lock(queue_mutex_);
        frame_available_.wait(lock, [this] { return frame_queue_.size() < max_queue_size_ || done_reading_; });
        if (done_reading_) {
            break;
        }
        frame_queue_.push(gray);
        lock.unlock();
        frame_available_.notify_one();
    }
    done_reading_ = true;
    frame_available_.notify_all();
}
#pragma once

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "Descriptor.h"

class Visualizer {
public:
    static void drawKeypoints(cv::Mat& img, const std::vector<KeypointDescriptor>& keypoints);

     static void drawVector(cv::Mat& img, const cv::Point& start, const cv::Point& end, const cv::Scalar& color, int thickness = 2);

    static void displayImage(const cv::Mat& img, const std::string& windowName = "Image");

    static void drawCross(cv::Mat& img, const cv::Point& pt, const cv::Scalar& color, int size = 6, int thickness = 2);
```

```
    static void visualizeTrajectory(const std::vector<cv::Point>& trajectory, int width, int height);
};

#include <opencv2/opencv.hpp>
#include <vector>
#include <string>
#include "Descriptor.h"
#include "Visualizer.h"

void Visualizer::drawKeypoints(cv::Mat& img, const std::vector<KeypointDescriptor>& keypoints) {
    for (const auto& kp : keypoints) {
        cv::circle(img, cv::Point(kp.x, kp.y), static_cast<int>(kp.sigma * 5), cv::Scalar(128, 128, 128), 1);
    }
}

void Visualizer::displayImage(const cv::Mat& img, const std::string& windowName) {
    cv::imshow(windowName, img);
}

void Visualizer::drawVector(cv::Mat& img, const cv::Point& start, const cv::Point& end, const cv::Scalar& color, int
thickness) {
    cv::arrowedLine(img, start, end, color, thickness);
}

void Visualizer::drawCross(cv::Mat& img, const cv::Point& pt, const cv::Scalar& color, int size, int thickness) {
    cv::line(img, cv::Point(pt.x - size, pt.y), cv::Point(pt.x + size, pt.y), color, thickness);
    cv::line(img, cv::Point(pt.x, pt.y - size), cv::Point(pt.x, pt.y + size), color, thickness);
}

void Visualizer::visualizeTrajectory(const std::vector<cv::Point>& trajectory, int width, int height) {
    cv::Mat trajectory_image(height, width, CV_8UC3, cv::Scalar(255, 255, 255));

    for (size_t i = 1; i < trajectory.size(); ++i) {
        cv::line(trajectory_image, trajectory[i - 1], trajectory[i], cv::Scalar(0, 0, 255), 2);
    }

    cv::imshow("Trajectory Visualization", trajectory_image);
    cv::waitKey(0);
}
```