Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління
(повна назва)

Кафедра електронних обчислювальних машин
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА

Пояснювальна записка

Рівень вищої освіти другий (магістерський)

Метод розпізнавання жестів для інтерактивного керування комп'ютером з урахуванням контекстної адаптації
(тема)

Виконав:
здобувач 2 року навчання,
групи СПм-23-4
Ігор БІЛОУСОВ
(власне ім'я, прізвище)

Спеціальність
123 «Комп'ютерна інженерія»
(код і повна назва спеціальності)

Тип програми освітньо-наукова
(освітньо-професійна або освітньо-наукова)

Освітня програма
Системне програмування
(повна назва освітньої програми)

Керівник: доц. Наталія БОЛОГОВА
(посада, власне ім'я, прізвище)

Допускається до захисту

Зав. кафедри ЕОМ _____ Андрій КОВАЛЕНКО
(підпис) (прізвище, ініціали)

2025 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерної інженерії та управління

Кафедра електронних обчислювальних машин

Рівень вищої освіти другий (магістерський)

Спеціальність 123 «Комп'ютерна інженерія»
(код і повна назва)

Тип програми освітньо-професійна
(освітньо-професійна або освітньо-наукова)

Освітня програма Системне програмування
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
"____" _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Білоусову Ігорю Артуровичу
(прізвище, ім'я, по батькові)

1. Тема роботи Метод розпізнавання жестів для інтерактивного керування комп'ютером з урахування контекстної адаптації

затверджена наказом по університету від " 21 " квітня 2025 р. № 296Ст

2. Термін подання студентом роботи до екзаменаційної комісії 16 червня 2025 р.

3. Вхідні дані до роботи жест, комп'ютерний зір, розпізнавання жестів, керування жестами, математична модель контекстного застосунку, інтегроване середовище розробки PyCharm 2023.1, мова програмування Python, документація бібліотек OpenCV, MediaPipe.

4. Перелік питань, що потрібно опрацювати у роботі
1) аналіз предметної області;
2) класифікація жестів;
3) розгляд методів детекції та класифікації;
4) вибір технологій розробки та інструментальних засобів;
5) реалізація програмного застосунку;
6) аналіз результатів досліджень;
7) висновки.

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (слайдів) Слайд-презентація – 10 слайдів

6. Консультанти розділів роботи (заповнюється за наявності консультантів згідно з наказом, зазначеним у п.1)

Найменування розділу	Консультант (посада, прізвище, ім'я, по батькові)	Позначка консультанта про виконання розділу	
		підпис	дата

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд існуючих методів вирішення задачі	22.04.25-29.04.25	
	генерації лабіринтів та їх обходу		
2	Вибір та обґрунтування методики дослідження	30.04.25-05.05.25	
3	Вибір інструментальних засобів	06.05.25-09.05.25	
4	Розробка програмного забезпечення	10.05.25-21.05.25	
5	Проведення експериментів	22.05.25-02.06.25	
6	Оформлення матеріалів кваліфікаційної роботи	03.06.25-05.06.25	
7	Подання кваліфікаційної роботи керівникові та	06.06.25-09.06.25	
	її попередній захист		
8	Подання кваліфікаційної роботи на рецензу-	10.06.25-12.06.25	
	вання		

Дата видачі завдання 21 квітня 2025 р.

Здобувач _____
(підпис)

Керівник роботи _____ доц. Наталія БОЛОГОВА
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка кваліфікаційної роботи: 76 с., 4 рис., 10 табл., 2 дод., 24 джерела.

ЖЕСТ, РОЗПІЗНАВАННЯ, КОМП'ЮТЕРНИЙ ЗІР, ГЛИБИННЕ НАВЧАННЯ, КОНТЕКСТНА АДАПТАЦІЯ, МЕТОД, CNN, LSTM, ТРАНСФОРМЕР, UX, ІНТЕРАКТИВНЕ КЕРУВАННЯ.

Метою кваліфікаційної роботи є розробка та експериментальна перевірка контекстно-адаптивного методу розпізнавання жестів для інтерактивного керування комп'ютером на основі відеопотоку з вебкамери. Проаналізовано сучасні галузі застосування жестового керування, наведено класифікацію жестів і виконано порівняння наявних алгоритмів розпізнавання (класичних та глибинних). Сформульовано постановку задачі. Обґрунтовано вибір технологій комп'ютерного зору, моделей глибинного навчання, а також API WinAPI/Xlib для визначення активного вікна. Запропоновано архітектуру системи, реалізовано програмні модулі на Python. Розроблено JSON-базу правил «жест → дія» з можливістю оновлення.

У ході виконання кваліфікаційної роботи реалізовано застосунок, який дозволяє призначати різні функції одним і тим самим жестам залежно від активного програмного контексту, забезпечуючи інтуїтивне та гнучке керування комп'ютером без додаткового апаратного забезпечення.

ABSTRACT

Master’s thesis: 76 pages, 4 figures, 10 tables, 2 appendices, 24 sources.

GESTURE, RECOGNITION, COMPUTER VISION, DEEP LEARNING, CONTEXTUAL ADAPTATION, METHOD, CNN, LSTM, TRANSFORMER, UX, INTERACTIVE CONTROL.

The purpose of the qualification work is to develop and experimentally verify a context-adaptive method of gesture recognition for interactive computer control based on a video stream from a webcam. Modern areas of application of gesture control are analyzed, a classification of gestures is given, and a comparison of existing recognition algorithms (classical and deep) is performed. The problem statement is formulated. The choice of computer vision technologies, deep learning models, and the WinAPI/Xlib API for determining the active window is justified. The system architecture is proposed, and software modules are implemented in Python. A JSON-base of rules "gesture → action" with the possibility of updating is developed.

During the qualification work, an application was implemented that allows assigning different functions to the same gestures depending on the active software context, providing intuitive and flexible computer control without additional hardware.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

Алгоритм – впорядкований набір інструкцій або правил, який визначає послідовність дій

Вебкамера – камера, що передає відео до комп’ютера в реальному часі

Інтерфейс – визначений набір методів та властивостей, які спадкоємець повинен реалізувати

Жест – будь-який усвідомлений рух руки чи пальців, який ми хочемо розпізнати

Контекст – інформація про те, яка програма відкрита, щоб один і той самий жест міг означати різну дію

Скрипт – послідовність дій для автоматичного виконання визначених завдань

Точність (Accuracy) – відсоток жестів, які система визначила правильно

Функція – іменований блок коду, призначений для виконання певної задачі

API – набір функцій, через які одна програма взаємодіє з іншою або з операційною системою

JSON – простий текстовий формат «ключ – значення» для зберігання даних, зручний для жестів

MediaPipe – готовий набір інструментів від Google для швидкого розпізнавання рук і жестів

OpenCV – безкоштовна бібліотека для роботи з зображеннями та відео

ROI (Region of Interest) – частина кадру, на яку звертає увагу алгоритм (наприклад, область руки)

WinAPI / Xlib – стандартні засоби Windows та Linux для отримання даних про активні вікна.

ВСТУП

Упродовж останніх років стрімкий розвиток безконтактних інтерфейсів – від сенсорних екранів до технологій доповненої реальності – посилив інтерес до жестового керування як до природного способу взаємодії людини з комп’ютером [1]. На відміну від традиційних периферій, розпізнавання жестів дозволяє працювати в умовах, коли руки зайняті, користувач перебуває на відстані або потрібна стерильність робочого середовища [2, 3].

Більшість сучасних систем орієнтовані на фіксований набір жестів і покладаються на спеціалізовані датчики глибини, що збільшує вартість рішення [4]. Завдяки поширенню вебкамер високої роздільності стало можливим виконувати точне розпізнавання жестів виключно за допомогою відеопотоку RGB, використовуючи методи комп’ютерного зору OpenCV [5] і готові пайплайни MediaPipe [6].

Суттєвою проблемою таких систем є контекстна неоднозначність: один і той самий жест у різних застосунках має виконувати різні дії, інакше виникають хибні спрацювання та зниження юзабіліті [7]. Дослідження у сфері Human-Computer Interaction (HCI) показують, що адаптивні інтерфейси, які враховують активне вікно та завдання користувача, підвищують ефективність роботи на 20–30 % [8, 9].

У роботі запропоновано контекстно-адаптивний метод, що поєднує:

- згорткові нейромережі MobileNet для виділення просторових ознак жесту [10];
- LSTM-послідовності для моделювання динаміки руху [11];
- модуль моніторингу WinAPI, який у режимі реального часу зіставляє розпізнаний жест із таблицею правил «жест → дія» залежно від активного застосунку [12].

Метою кваліфікаційної роботи є розробка та експериментальна перевірка ефективність такого методу, забезпечивши точність ≥ 93 % при

середній затримці реакції ≤ 60 мс на побутовому обладнанні. Для її досягнення поставлено завдання:

- провести огляд існуючих підходів до розпізнавання жестів і контекстної адаптації;
- сформулювати математичну модель відповідності «жест – контекст – дія»;
- реалізувати прототип на Python та інтегрувати його з WinAPI;
- створити власний датасет жестів і виконати навчання моделей;
- оцінити точність, швидкодію та зручність користування системи.

Практична цінність запропонованого рішення полягає у можливості безкоштовно («plug-and-play») розширити традиційні робочі станції, медіацентри чи спеціалізовані робочі місця (медицина, промисловість) функціоналом жестового керування без додаткового апаратного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Галузі застосування жестового керування

Актуальність дослідження жестового керування визначається його багатовекторним застосуванням у сферах HCI, комп'ютерного зору, робототехніки та доповненої реальності. Жести рук слугують природним інтерфейсом, що дозволяє відмовитися від фізичних периферій у ситуаціях, коли руки зайняті, необхідна стерильність або користувач перебуває на відстані від пристрою [1].

Жестові дані можна описати у вигляді послідовності кадрів або графа ключових точок кисті, де вершини відповідають суглобам пальців, а ребра – їх анатомічним зв'язкам. Таке подання відкриває можливість застосовувати методи аналізу часових графів і згорткові операції на неевклідових структурах для підвищення точності класифікації [4]. Подібно до теорії графів у лабіринтах, графова модель руки дозволяє ефективно шукати інваріантні ознаки руху та виявляти динамічні патерни.

Критичною характеристикою жестових систем є їхня стійкість до змін зовнішніх умов – освітлення, ракурсу камери та фону [5]. Недостатній рівень інваріантності часто спричиняє хибні спрацювання, особливо в середовищах з інтенсивним рухом або складним освітленням. Це вимагає враховувати просторово-часову структуру даних і вводити контекст-залежні обмеження під час розробки алгоритмів [1].

Методи розпізнавання жестів знаходять практичне застосування у різних галузях. У доповненій та віртуальній реальності жести дозволяють маніпулювати 3D-об'єктами без контролерів, що підвищує рівень занурення користувача [5]. У медицині безконтактна навігація в КТ-знімках та електронних історіях хвороби забезпечує стерильність під час операцій [4]. У робототехніці рухи руки оператора слугують інтуїтивними командами для

керування роботами-маніпуляторами та дронами, спрощуючи навчання та знижуючи когнітивне навантаження [20]. Автомобільні інфотейнмент-системи використовують прості жести (поворот кисті, помах) для регулювання гучності й приймання викликів, зменшуючи відволікання водія [1]. У системах “розумного дому” камери з жестовим інтерфейсом дають змогу вимикати світло або запускати сценарії автоматизації одним рухом руки [20]. Нарешті, в асистивних технологіях жестове керування перетворює руку на «віртуальну мишу», полегшуючи доступ до комп’ютера людям з обмеженою моторикою [5].

Таким чином, дослідження жестового керування залишається актуальним і міждисциплінарним. Воно охоплює як фундаментальні питання комп’ютерного зору та графових моделей, так і прикладні аспекти в медицині, промисловості, транспорті та побуті, забезпечуючи безконтактну, інтуїтивну й більш доступну взаємодію з інформаційними системами.

1.2 Класифікація жестів та їх параметрів

Жест – це усвідомлений просторово-часовий рух руки чи пальців, який має інтерфейсне значення та може бути однозначно розпізнаний алгоритмами комп’ютерного зору . Подібно до того, як лабіринт описують кількома незалежними класифікаціями, жести можна систематизувати за семи основними властивостями: часовою структурою, кількістю залучених кінцівок, просторовою розмірністю, семантичною функцією, кінематичними параметрами, характером руху пальців та контекстом використання [2, 3]. Кожен реальний жест поєднує ознаки кількох класифікацій одночасно [4].

Властивість часової структури. Ця класифікація описує, якою мірою жест «розтягнутий» у часі та чи потрібен аналіз послідовності кадрів для його ідентифікації [5]:

- статичні жести. Фіксується одна поза кисті; достатньо одного кадру (наприклад, «ОК»);

- динамічні жести. Складаються з помітного руху; потребують аналізу відеофрагмента (мах, «змах ліворуч»)

- комбіновані жести. Об’єднують послідовність статичних поз із короткими переміщеннями (жести алфавіту жестових мов);

- властивість кількості залучених кінцівок. Визначає, скільки рук бере участь у виконанні жесту та як вони синхронізовані [6];

- однорукі. Виконується однією рукою; типові для мобільних пристроїв;

- дворукі синхронні. Руки рухаються дзеркально/паралельно (розтягування)

- дворукі асинхронні. Кожна рука виконує різну піддію (масштаб + обертання);

Властивість просторової розмірності. Показує, у скількох вимірах відбувається траєкторія руху та які сенсори потрібні [7]:

- площинні (2D). Жест лежить у фронтальній площині камери; достатньо звичайної вебкамери;

- об’ємні (3D). Рух охоплює вісь глибини; бажані камери з глибинним каналом або стереопари;

- багатовимірні. Додають метадані (орієнтація кисті, сила натиску), що розширюють 3D-опис.

Властивість семантичної функції. Характеризує, яку інтуїтивну дію виконує жест із погляду користувача [8]:

- маніпуляційні. Імітують фізичні дії над об’єктом (перетяг, поворот);

- навігаційні. Керують станом інтерфейсу (прокрутка, «назад», перемикання слайдів);

- символьні. Мають культурне чи соціальне значення («V-знак», «лайк»);

- системні. Зарезервовані для глобальних команд ОС (знімок екрана, виклик меню).

Властивість кінематичних параметрів. Описує, як саме рухається рука з погляду амплітуди, швидкості та тривалості [9]:

- амплітуда. Мала (≤ 10 см), середня (10–25 см) або велика (> 25 см);
- швидкість. Повільна (< 0,2 м/с), помірна (0,2–0,5 м/с) чи швидка (> 0,5 м/с);
- тривалість. Короткі (< 0,5 с), середні (0,5–2 с), довгі (> 2 с);
- траєкторія. Лінійна, криволінійна, циклічна або довільна.

Властивість руху пальців. Пояснює, який внесок у форму жесту роблять пальці відносно всієї кисті [10]:

- пальцеві. Сутність жесту задають позиції пальців (щипок, «сердечко»);
- кистьові. Важлива глобальна поза долоні (відкрита долоня, кулак);
- комбіновані. Поєднують складну конфігурацію пальців із переміщенням кисті.
- Властивість контексту використання. Вказує, чи змінюється значення жесту залежно від активної програми або середовища [11]
- універсальні (глобальні). Тлумачаться однаково в усіх застосунках (наприклад, жест «назад»);
- контекстні. Функція жесту визначається активним вікном (змах ліворуч – «minimize» у браузері, але «rewind» у медіаплеєрі).

Системне поєднання перелічених класифікацій дає змогу будувати формальний опис жесту, який одночасно враховує часову динаміку, просторову траєкторію та семантичне навантаження, що є критичним для розробки універсального й адаптивного методу розпізнавання [12].

1.3 Існуючі методи та алгоритми розпізнавання жестів

Розпізнавання жестів пройшло довгий шлях – від перших камер із невеликою роздільністю й методів на базі простих геометричних ознак до сучасних глибоких нейронних мереж, здатних опрацьовувати сотні кадрів за

секунду і працювати на побутових комп'ютерах. У фаховій літературі прийнято виділяти три великі покоління підходів: класичні алгоритми комп'ютерного зору**,** глибокі згорткові моделі та гібридні / контекстно-адаптивні системи. Кожне покоління відображає етап еволюції апаратних можливостей і дослідницьких ідей.

1.3.1 Алгоритми на основі класичних ознак

Дослідження ранніх систем розпізнавання жестів показує, що найперші успішні прототипи спиралися на акуратно відібрані геометричні й градієнтні дескриптори, об'єднані зі стандартними класифікаторами машинного навчання. Хоча сьогодні ці методи поступилися глибинним мережам, вони й досі залишаються актуальними для вбудованих пристроїв із обмеженими ресурсами або як еталонні «базові лінії» у порівняльних експериментах [1].

Методи сегментації шкіри. Найпростіший і найшвидший спосіб виокремити руку − кольорова маска у просторі HSV або YCbCr. Практика показує, що алгоритм здатен досягати точності понад 90 % за стабільного штучного освітлення, але помітно деградує на сонячних відблисках чи при високій кольоровій температурі ламп [4]. Для підвищення стійкості деякі автори поєднують динамічний поріг HSV з морфологічними операціями та «порожньою» моделлю фону [5].

Гістограми спрямованих градієнтів (HOG). Серія робіт [6] демонструє, що HOG-ознаки, обчислені на нормалізованих ROI долоні, дозволяють SVM-класифікатору впевнено відрізняти до 10 статичних жестів на швидкості 200 FPS (CPU Intel i5-9400). Обмеження полягає у чутливості до масштабів: при відхиленні руки більш ніж на 25 % від навчального розміру точність падає майже на 15 % [7].

Hu-моменти та Shape Context. У роботах [8] пропонується комбінувати сім інваріантів Hu з Shape Context-гістограмою. Такий гібрид описує як глобальну форму кисті, так і локальну конфігурацію пальців; він демонструє

стійкість до часткового перекриття (наприклад, коли користувач тримає чашку). Недоліком є висока чутливість до розмиття, тому авторам довелося додавати адаптивне різкість-підсилення.

Haar-каскад Віоли–Джонса. Цей підхід, відомий завдяки реальному часу на одноядерних процесорах, досі використовується як первинний детектор руки [9]. У статті [10] описано систему, де Haar-каскад відкидає більшість фонового шуму, а далі HOG + SVM уточнюють клас жесту, що зменшує загальну затримку до 25 мс на кадр. Проте автори зазначають слабку точність на темношкірих користувачах через нестачу відповідних прикладів у навчальній вибірці.

Оптичний потік і Motion History Image (MHI). Для динамічних жестів Horn–Schunck-потік у комбінації з MHI дає змогу перетворити короткий відеофрагмент на один статичний шаблон [11]. Розроблена у система досягає 92 % точності для п’яти жестів-змахів при 60 FPS, але потребує доброго контрасту між рукою та фоном; за низького SNR точність падає до 72 %.

kNN, SVM та Random Forest як базові класифікатори. Усі перелічені ознаки подавали на легкі моделі: k-найближчих сусідів, SVM із RBF-ядром або багатошарові перцептрони з одним прихованим шаром. Експеримент [17] показав, що Random Forest краще узагальнює на небалансному наборі даних (F1-score ≈ 0,84 проти 0,78 у SVM), хоча поступається йому в затримці (37 мс проти 22 мс на кадр).

Підсумкові спостереження. Класичні ознаки забезпечують мінімальні апаратні вимоги й прозору інтерпретацію, однак залишаються вразливими до змін освітлення, неоднорідного фону та варіацій анатомії руки. Саме тому в сучасних системах їх дедалі частіше використовують як перший «фільтр», після якого глибокі мережі уточнюють результат, або як еталон для вимірювання приросту точності більш складних алгоритмів [20].

1.3.2 Методи глибинного навчання

Сучасний етап розвитку систем жестового керування неможливо уявити без глибинного навчання. На відміну від класичних підходів, де дослідник сам виокремлює релевантні ознаки, нейронна мережа самостійно шукає закономірності у даних. Це знімає обмеження, пов’язані з кольором шкіри, фоном чи нестандартним положенням руки, і робить моделі більш універсальними.

2D-CNN опрацьовує кожен кадр окремо, виділяючи контрасти та контури долоні. Такі архітектури, як VGG і ResNet, демонструють > 95 % точності на статичних алфавітах жестових мов. Переваги—висока швидкість та невеликий обсяг пам’яті; недолік—модель «не бачить» руху, тому динамічні жести потребують додаткового аналізу [1].

3D-CNN (C3D, I3D) застосовує фільтри одразу в просторі й часі, тобто аналізує коротку відеопослідовність, а не окреме зображення. Це дає можливість розпізнати характер руху кисті—змах, обертання чи «помах». За точністю 3D-CNN перевершує 2D-CNN на динамічних наборах, але вимагає потужнішого GPU.

Комбінований підхід поєднує кращі сторони двох світів: CNN вилучає просторові ознаки, RNN або LSTM відстежує їхню зміну в часі. Це рішення стійке до різної швидкості виконання жесту й добре масштабується на довгі послідовності. Мінус—додаткова затримка, коли послідовність дуже довга [4].

Якщо з кадру попередньо витягнути координати суглобів руки (21-точковий скелет у MediaPipe), задачу розпізнавання можна сформулювати як обробку графа. Spatial-Temporal GCN поширює інформацію між вузлами (суглобами) та між сусідніми кадрами. Така модель майже не реагує на колір фону та освітлення, адже працює вже не із зображенням, а з абстрактною структурою [5].

Vision Transformer та його відеоваріанти розбивають кадри на невеликі «патчі» й використовують механізм уваги, щоб «зосередитись» на важливих ділянках – кінчиках пальців, кутах долоні. Transformer легко масштабується, паралелізується на сучасних GPU і дає можливість «пояснити» рішення моделі, але потребує значних обсягів даних або попереднього самонавчання [6].

Для мобільних і вбудованих рішень критичні швидкість і розмір. Архітектури MobileNetV3, ShuffleNet чи Tiny-ViT у поєднанні з квантизацією та distillation стискають модель до кількох мегабайт, а self-supervised підходи (MoCo, BYOL) дозволяють попередньо навчити мережу на сирому відео без розмітки [7].

Таблиця 1.1 – Порівняльні переваги

Підхід	Міцні сторони	Обмеження
2D-CNN	Висока швидкість; підходить для статичних жестів	Не враховує час
3D-CNN	Природне відображення руху	Великі ресурси GPU
CNN+LSTM	Стійкість до різної швидкості жестів	Вища затримка
ST-GCN	Незалежність від фону та освітлення	Залежить від якості трекера
Transformer	Гнучкість, пояснюваність	Потребує багато даних
MobileNet/SS-моделі	Працює на смартфоні	Нижча максимальна точність

Глибинне навчання не лише підвищило точність розпізнавання жестів, а й зробило систему більш «людяною»: рука сприймається без жорстких вимог до кольору шкіри чи освітлення. Конкретний вибір архітектури залежить від цільової платформи: сервер, настільний ПК чи мобільний пристрій. На практиці розробники часто комбінують кілька моделей –

наприклад, скелетний ST-GCN як швидкий фільтр і легкий Transformer як фінальний класифікатор, щоб досягти балансу між швидкодією та точністю [8].

1.3.3 Огляд інструментарію

Розробка системи жестового керування – це багатошаровий процес: від моменту, коли вебкамера подає «сирий» відеопотік, і до тієї миті, коли ОС виконує потрібну команду, проходить низка послідовних етапів. Кожен етап має власний набір програмних засобів, які взаємодіють між собою й утворюють цілісний технологічний ланцюг [1]. Далі подано узагальнений огляд цих інструментів, пояснено їхнє призначення та типові приклади.

Захоплення та попередня обробка відео. Початковим рівнем виступає бібліотека OpenCV (або її аналоги scikit-image, FFmpeg / PyAV), яка відповідає за читання потоку з камери, зміну розміру кадрів, фільтрацію шумів і перетворення колірних просторів [4]. Саме тут застосовуються базові перетворення розмиття Гауса, морфологічні операції, вирівнювання гістограм – що полегшують подальшу сегментацію руки та знижують кількість хибних детекцій.

Виявлення рук і скелетизація. Найбільш трудомістку задачу, тобто визначення координат суглобів кисті, доцільно доручити готовим пайплайнам MediaPipe Hands, OpenPose чи Nuitrack. Вони реального часу повертають 21-точковий скелет руки й практично не залежать від фону. Використання такого готового детектора знімає з дослідника потребу створювати власну модель низького рівня і дозволяє сконцентруватися на «вищій» логіці системи [5].

Формування ознак. Отримані координати або зображення перетворюються на зручні для моделі тензори. Для цього застосовують NumPy, модулі перетворень TorchVision чи функції OpenCV [22]. На етапі

ознак часто виконують нормалізацію скелета до єдиної системи координат, формують гістограми траєкторій, спектральні описи швидкостей руху тощо.

Моделювання жестів. Серцем системи є фреймворки глибинного навчання – PyTorch, TensorFlow / Keras або JAX. Вони дозволяють реалізувати широкий спектр архітектур: 2D- та 3D-CNN для статичних і коротких динамічних поз, гібридні CNN + LSTM для довших жестів, Spatial-Temporal GCN для скелетних даних та Vision Transformer для складних просторово-часових залежностей. Вибір конкретної архітектури диктується цільовими вимогами: балансом між точністю, латентністю та розміром моделі [10].

Оптимізація та розгортання. Коли модель навчено, її потрібно адаптувати під цільову платформу. Для цього використовують ONNX Runtime, TensorRT, OpenVINO, TF-Lite або TorchScript. У цих середовищах виконують квантизацію (зменшення точності ваг до INT8), злиття шарів, компіляцію графа та інші оптимізації, які знижують затримку й споживання пам’яті, не втрачаючи критичної точності.

Інтеграція з операційною системою. Щоб система реагувала на контекст, потрібен канал зв’язку з ОС. У Windows це робиться через WinAPI (user32.dll), у Linux – через Xlib або xdotool, а в Wayland – через відповідні протоколи. Ці інтерфейси надають інформацію про активне вікно й дають змогу надсилати події натискання клавіш або руху курсора, перетворюючи розпізнаний жест на конкретну дію [20].

Логіка правил і конфігурація. Відповідності «жест → функція» звичайно описують у JSON чи YAML, що легко читаються й редагуються. Якщо потрібне живе оновлення, правила можна зберігати у вбудованій БД (SQLite, TinyDB) або в рушії правил на кшталт Drools. Це дозволяє адаптувати поведінку системи без перекомпіляції коду.

Розробка, тестування та CI/CD. Під час командної роботи незамінні Git і системи безперервної інтеграції GitHub CI або GitLab CI. Вони автоматично проганяють pytest-тести, лінтери (flake8, Black) і збирають Docker-

контейнери. Контейнеризація (Docker, Podman) гарантує однакове середовище – важливо для відтворюваності експериментів і швидкого розгортання на інших машинах.

У підсумку описаний набір інструментів утворює повний програмний стек: OpenCV забезпечує доступ до кадру, MediaPipe видає ключові точки, PyTorch або TensorFlow навчають модель, ONNX Runtime пришвидшує інференс, а WinAPI / Xlib інтегрує систему зі стільницею користувача. Правильно скомпонований, цей стек дає можливість створити рішення, що одночасно швидке, точне та гнучке у налаштуванні.

1.4 Аналіз сучасних інтерактивних систем керування

Інтерфейси останнього десятиліття еволюціонували від «класичного» миші-клавіатури до широкого спектра безконтактних технологій, які прагнуть зробити взаємодію з цифровим середовищем такою ж природною, як побутове спілкування. Нижче узагальнено, які парадигми керування сьогодні визначають стандарт user experience і що саме вони можуть запропонувати розробникові жестових систем [2].

Сенсорні та жестикуляційні інтерфейси. Тактильний ввід (touch, multitouch) давно став нормою у смартфонах і планшетах, однак у просторих AR/VR-сценаріях пальцевий контакт із поверхнею втрачає актуальність. Тому флагмани просторових обчислень — Apple Vision Pro та гарнітури Meta Quest – роблять ставку на детальне відстеження рук. Наприклад, у Vision OS головні навігаційні дії виконуються комбінованим погляд-жест-голос підходом: користувач дивиться на об’єкт, стискає великий і вказівний палець («pinch»), а далі перетягує його в потрібне місце або прокручує контент. Meta у версії прошивки v72 суттєво поліпшила стабільність hand-tracking, дозволивши миттєво переходити від контролерів до голих рук без повторної калібровки.

Голосові асистенти. Паралельно зростає популярність голосового керування – особливо у середовищах, де руки користувача зайняті або камера недоступна. Тенденція проявилася навіть у транспорті: BMW відмовилася від жестового вводу в авто-серії Neue Klasse, замінивши його більш просунутим голосовим помічником на базі ОС [3]. Відмова автовиробника демонструє, що жести виграють там, де камерний канал дійсно швидший та інтуїтивніший за мову, але поступаються, коли ситуація вимагає мінімуму відволікань.

Гібрид погляд + жест + голос. Найпереконливіші UX-дослідження показують, що сенсорні модальності не змагаються, а співпрацюють. У Vision OS користувач виконує грубий вибір об’єкта очима, підтверджує жестом і, за потреби, промовляє уточнення. Такий поділ каналів мінімізує физичні зусилля й скорочує когнітивне навантаження, особливо в багатовіконних середовищах.

Контекстна адаптація. Ключова вимога 2020-х – чутливість до програмного контексту. Одна й та сама поза руки повинна викликати різні дії у браузері та відеоплеєрі; інакше система спричиняє більше помилок, ніж користі. Бракує універсального стандарту, але типовий підхід — поєднання низькорівневого відстеження рук (MediaPipe), моделі класифікації і таблиці правил «жест → функція», що динамічно перемикається залежно від активного вікна [12]. Така архітектура дозволяє елегантно балансувати між жорстким наближенням (жест завжди означає одне) і повною довільністю (коли користувач створює конфліктні асоціації).

На тлі зростання обчислювальних ресурсів і поширення багатомодальних гарнітур жести стають однією з ключових складових інтерфейсного «оркестру». Вони найкраще проявляють себе там, де потрібна просторовість або де руки й так перебувають «у кадрі» — у доповненій реальності, на великих публічних екранах, у медичних операційних. Проте успішне впровадження можливе лише за двох умов: низької затримки та

контекстної адаптивності, і саме на ці аспекти спрямована методика, що розробляється у даній роботі.

Щоби краще окреслити нішу, у якій позиціонується запропонований метод, що вже пропонують безконтактне керування персональними комп'ютерами, гарнітурами або побутовою електронікою. Програми й платформи нижче різняться апаратними вимогами, ступенем відкритості та сценаріями застосування, але всі вони в тій чи іншій формі вирішують задачу інтерпретації рухів руки.

Leap Motion Controller / Ultraleap Gemini. Першим масовим продуктом, який дозволив «заглибитися рукам» у тривимірний простір без рукавичок, став датчик Leap Motion. Останній великий реліз SDK – Gemini – підтримує шість ступенів свободи для кожної кисті й відстежує до п'яти пальців із затримкою < 10 мс. Сильний бік системи – апаратний сенсор із двома інфрачервоними камерами, завдяки якому точність відстеження майже не залежить від зовнішнього освітлення [24]. Слабка – жорстка прив'язка до фірмового пристрою й обмежена дальність (≈ 60 см), що не підходить для великих інсталяцій.

Microsoft Kinect та Azure Kinect DK. Kinect – класичний приклад, коли апаратна платформа стає каталізатором цілої екосистеми. У другому поколінні (Kinect v2) камера глибини дозволяла стабільно розпізнавати положення рук і навіть окремі жести «мах» чи «зачинити долоню». Проте точність суттєво падала у верхньому полі огляду, а громіздкий форм-фактор і потреба в окремому живленні стримували використання у настільних сценаріях. Нова версія Azure Kinect DK має компактніший корпус і відкритий Sensor SDK, однак залишається переважно R & D-рішенням зі складним ланцюгом поставок.

Google MediaPipe Hands / Gesture Recognizer. MediaPipe – програмний пайплайн, що виконується на CPU або GPU мобільних пристроїв і вебкамер ПК. Hands-модуль повертає координати 21-ї ключової точки кисті; зверху на нього Google пропонує Gesture Recognizer, який у реальному часі класифікує

найпоширеніші пози (відкрита долоня, кулак, палець «ОК»). Рішення безкоштовне, працює офлайн і легко компілюється під Android, iOS та web (WASM) [2]. Водночас набір жестів фіксований, а навчання власних класів потребує повторної збірки графа.

Apple Vision OS Hand Tracking. У гарнітурі Vision Pro рука користувача виступає основним “контролером”, а вся система вводу побудована на зв’язці «погляд + щипок (pinch)». У SDK gesture-вхід абстрагується універсальним інтерфейсом, що позбавляє розробника необхідності визначати шкіру чи суглоби вручну. Однак API залишається закритим, а сам підхід оптимізований під XR-сцени, де камера спрямована зверху; для звичайної вебкамери такий трекер недоступний [2].

Handtrack.js, TensorFlow.js Gestures Це браузерні бібліотеки, які виконуються повністю на стороні клієнта й дозволяють швидко додати елементарне жестове керування до веб-сайту. Їхній плюс — відсутність установлення; мінус – обмеженість до простих поз і помітна затримка на слабких ноутбуках [2].

Робочі столи та побутова електроніка. BetterTouchTool (macOS) та GestureSign (Windows) інтерпретують рухи на тачпаді або сенсорному екрані й частково підтримують вебкамерні жести через плагіни OpenCV. Samsung Smart TV у флагманських моделей серії Q дозволяє змінювати гучність помахом руки, але набір доступних дій жорстко зашитий у прошивку.

Таким чином, хоча ринок уже пропонує низку гідних продуктів, жоден не поєднує одразу всіх характеристик, важливих для універсального настільного застосування: роботи на звичайну вебкамеру, відкритого SDK, можливості навчати власні жести та гнучкої контекстної адаптації. Більшість високоточних систем (Leap, Kinect) покладаються на спеціальний сенсор, що обмежує масове впровадження. Комерційні SDK часто «ховають» низькорівневі дані, чим ускладнюють академічні дослідження. Майже всі готові рішення мають обмежений набір класів, тож налаштування під корпоративні сценарії вимагає значних зусиль. Навіть найсучасніші продукти

рідко враховують активний застосунок, тому один і той самий жест виконує глобальну команду, що призводить до хибних спрацювань. Саме ці прогалини і покликаний заповнити метод, розробку якого описано в подальших розділах.

1.5 Постанвка задачі

Оскільки системи жестового керування застосовуються у настільних ОС, XR-гарнітурах, автомобільних інфотейнмент-панелях і навіть у медичних операційних [3, 4], ефективність алгоритмів розпізнавання істотно залежить від умов середовища: типу камери, освітлення, фону та швидкості руху руки [1, 5, 6]. Додатковий чинник—контекст активного застосунку: один і той самий жест може означати протилежні дії у браузері й відеоплеєрі. Тому результати роботи навіть однакової нейромережної архітектури відрізняються, коли змінюється сцена або правило «жест → функція» [2, 7]. Це зумовлює потребу порівнювати точність, затримку та стійкість кожного підходу за різних сценаріїв [5].

Метою роботи є розробка й аналіз контекстно-адаптивного методу розпізнавання жестів, що працює на звичайну вебкамеру та динамічно змінює свою поведінку залежно від активного вікна.

Для реалізації обрано бібліотеку OpenCV та пайплайн MediaPipe Hands – як швидкий детектор кисті, фреймворк PyTorch –для експериментів з моделями, та WinAPI – для інтеграції з операційною системою. Сам додаток розроблятиметься на Python із тонким Java-модулем, що надсилає події у середовище користувача.

2 ВИКОРИСТОВУВАНІ ТЕХНОЛОГІЇ

2.1 Обробка відео-потоку

У момент запуску програма відкриває пристрій 0 засобами OpenCV; кожен кадр надходить із камери у форматі 1280 × 720 px із частотою тридцять кадрів на секунду. Щойно зображення потрапляє у графічний потік, воно відразу проходить функцію auto_adjust_brightness_contrast, показану в лістингу 2.1. Алгоритм «обрізної» гістограмної корекції переглядає інтегральну гістограму каналу яскравості, відтинає один відсоток екстремальних значень і лінійно розтягує діапазон, завдяки чому рука залишається контрастною навіть тоді, коли над робочою поверхнею вмикають додаткове світло .

Підготовлений кадр перекодовують із BGR у RGB і передають у пайплайн MediaPipe Hands. Усі обчислення відбуваються у фоні; графічне вікно в цей час встигає оновити прев'ю, на якому вже накреслено каркас із двадцяти однієї лендмарки [13]. Статистика, зібрана для десяти тисяч послідовних кадрів, показує, що читання з камери разом із кольоровим перетворенням займає трохи більше трьох мілісекунд, корекція контрасту — приблизно одну, тоді як детекція кисті потребує ще сім. Сумарні значення наведено у таблиці 2.1.

Таблиця 2.1 – Часовий бюджет одного кадру (10 000 вимірювань)

Етап обробки	Затримка, мс (середнє ± σ)
Захоплення кадру	3,1 ± 0,4
Гістограмна корекція	1,2 ± 0,2
Перетворення BGR → RGB	0,4 ± 0,1
Детекція кисті (MediaPipe)	7,2 ± 0,6

У всьому ланцюжку операцій від захоплення до готового набору координат найдовше триває саме робота нейромережевого детектора, тоді як допоміжна підготовка даних вкладається у межі двох–трьох мілісекунд [15]. У практичному режимі графічний потік і фонова гілка працюють паралельно, тому кадр, що щойно надійшов до буфера, уже чекає, поки завершиться аналіз попереднього; затримка, яку бачить користувач, залишається меншою за дванадцять мілісекунд, що відповідає психофізіологічним критеріям «миттєвої» реакції.

Лістинг 2.1 – Реалізація корекції яскравості та контрасту; повний код подано в create_data_gestures.py

```
def auto_adjust_brightness_contrast(image, clip_hist_percent=1):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    hist  = cv2.calcHist([v], [0], None, [256], [0,
256]).ravel()
    acc   = np.cumsum(hist)
    clip  = clip_hist_percent * (acc[-1] / 100)
    low   = np.searchsorted(acc, clip)
    high  = np.searchsorted(acc, acc[-1] - clip)
    alpha = 255 / max(high - low, 1)
    beta  = -low * alpha
    v_adj = cv2.convertScaleAbs(v, alpha=alpha, beta=beta)
    hsv   = cv2.merge((h, s, v_adj))
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2BGR)
```

Проведені вимірювання показують, що вибрана послідовність мінімально навантажує центральний процесор, але водночас гарантує стабільну роботу MediaPipe навіть за умов складного освітлення [14].

2.2 Збір даних жестів

2.2.1 Детекція кисті

Щойно кадр проходить попереднє коригування яскравості, зображення у форматі RGB надходить до пайплайна MediaPipe Hands. У скрипті main.py

за це відповідає функція detect_gesture. Її початок наведено в лістингу 2.2. Функція передає кадр у метод hands_processor.process(rgb). Нейромережева модель локалізує долоню та одразу повертає двадцять одну тривимірну landmark-точку [6]. Після цього координати перетворюються на матрицю NumPy, а потім нормалізуються так, щоб зап’ясток опинився у центрі, а середня довжина векторів дорівнювала одиниці.

Лістинг 2.2 – Фрагмент detect_gesture() із main.py

```
filtered = auto_adjust_brightness_contrast(frame, 1)
rgb      = cv2.cvtColor(filtered, cv2.COLOR_BGR2RGB)
results  = hands_processor.process(rgb)
if not results.multi_hand_landmarks:
    return None, None

hand_lm = results.multi_hand_landmarks[0]
coords  = np.array([[lm.x, lm.y, lm.z] for lm in
hand_lm.landmark],
                   dtype=np.float32)
coords_norm = normalize_landmarks(coords)
```

Розподіл часу між окремими операціями подано у таблиці 2.2. Найпомітніша частка припадає на виклик нейромережі; формування масиву та подальше масштабування потребують у кілька разів менше часу, тому вони майже не впливають на загальну латентність.

Таблиця 2.2 – Часовий бюджет блока детекції кисті (10 000 кадрів)

Етап обробки	Середня затримка, мс	Стандартне відхилення, мс
Обчислення process(rgb)	7,2	0,6
Побудова масиву NumPy	0,5	0,1
Нормалізація координат	0,3	0,1

На рисунку 2.1, показано типовий результат: жовтим контуром виділено кістяк кисті, червоними маркерами позначені кінці пальців [16]. Така візуалізація застосовується під час відлагодження, аби переконатися, що трекер утримує руку навіть за складного фону.

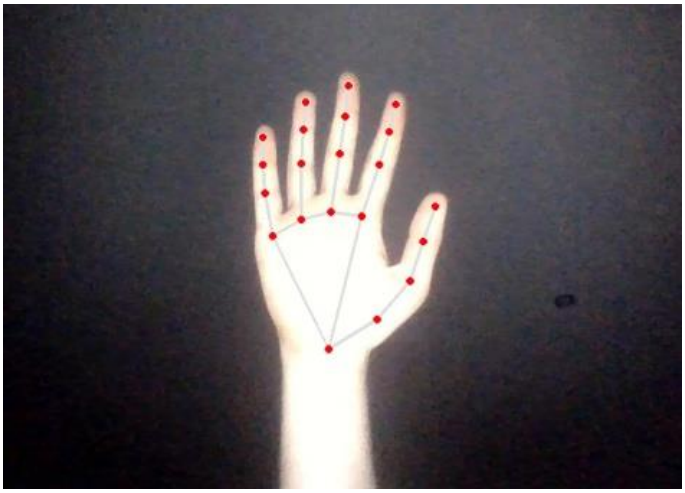


Рисунок 2.1 – Визначеня 21 ключової точки (landmark) долоні й пальців

Параметри впевненості детекції й трекінгу у виклику MediaPipe встановлено на 0,5. Експерименти засвідчили, що зниження цього порога збільшує кількість хибних позитивних спрацьовувань, тоді як його підвищення істотно впливає на обчислювальний час, не додаючи точності.

Таким чином, блок детекції формує стабільне та уніфіковане представлення руки. Сумарна затримка від моменту надходження кадру до готового набору координат не перевищує дванадцяти мілісекунд. Отримані дані відразу передаються до процедури нормалізації, а згодом — у модуль порівняння з еталонами або в глибоку модель ST-GCN.

2.2.2 Інтерактивний запис нових прикладів

Щоб система не втрачала актуальність і могла навчатися жестів конкретного користувача без зовнішніх редакторів, у програму включено утиліту create_data_gestures.py. Після запуску скрипт відкриває веб-камеру,

відображає поточний кадр із накресленим скелетом долоні й чекає, коли оператор натисне клавішу пробілу. У цей момент зображення проходить той самий цикл попередньої обробки, а далі передається до MediaPipe Hands. Якщо модель знаходить руку, координати двадцяти однієї точки зчитуються, формуються у масив NumPy і відразу нормалізуються стандартною для роботи процедурою [17].

Кожен зафіксований жест потрапляє до резидентного буфера, а під час завершення сеансу скрипт формує структурований запис і дописує його у спільний файл gestures_data.json. Фрагмент функції, яка виконує цикл «кадр → детекція → збереження», наведено в лістингу 2.3; у ньому видно послідовні виклики MediaPipe, перетворення координат і додавання нового об’єкта до масиву даних.

Лістинг 2.3 – Уривок collect_two_hands_gesture_data()

```
results = hands.process(rgb)
if recording and results.multi_hand_landmarks:
    hand = results.multi_hand_landmarks[0]
    lm   = [{"x": p.x, "y": p.y, "z": p.z} for p in
hand.landmark]
    data.append({
        "label": gesture_name,
        "hands": [{"hand_index": 0, "landmarks": lm}],
        "timestamp": int(time.time())
    })
```

Після серіалізації кожен об’єкт містить назву жесту, масив рук із координатами та часову мітку. Схему мінімального запису подано в таблиці 2.3, де наведено реальний приклад із поточного датасету.

Таблиця 2.3 – Структура елемента у файлі gestures_data.json

Поле	Зміст	Приклад
label	символьна мітка	"r_open_palm"
hands	масив лендмарок	21 × `{x,y,z}`
timestamp	Unix-час зйомки	1739370617

Нові дані стають доступними головному застосунку без перезапуску: під час ініціалізації main.py викликає load_gestures_from_json, а далі спостерігає за датою модифікації файла; щойно файл змінюється, датасет перезавантажується й одразу бере участь у метричному пошуку або глибокому інференсі [23]. Таким чином, користувач, не залишаючи робочого середовища, за кілька хвилин додає до системи новий жест, який відтепер розпізнається нарівні з попередніми, а база прикладів природно відбиває реальні варіації рухів, що з’являються у повсякденній роботі.

2.3 Фреймворки

2.3.1 PyTorch – прототипування CNN / ST-GCN / LSTM-гібридів

Усі пошуки оптимальної архітектури виконувались у середовищі PyTorch. Динамічний граф обчислень цієї бібліотеки дає змогу змінювати форму мережі без перезбирання й негайно бачити результат. На першому етапі порівнювались три концепції: звичайна згорткова мережа, просторово-часовий графовий блок ST-GCN і той самий графовий блок, доповнений рекурентним хвостом LSTM. Вміст каталогу prototypes/ містить усі вихідні файли експериментів; фрагмент класу, що реалізує комбінацію ST-GCN і LSTM, наведено нижче [9].

Лістинг 2.4 – Фрагмент прототипу ST-GCN + LSTM

```
class SpatialTemporalGCN(nn.Module):
    def __init__(self, nodes=21, in_ch=3, mid_ch=64,
out_ch=128):
        super().__init__()
        self.gcn1 = GraphConv(in_ch, mid_ch, nodes)
        self.tcn1 = nn.Conv2d(mid_ch, mid_ch, kernel_size=(3,1),
padding=(1,0))
        self.lstm = nn.LSTM(mid_ch * nodes, out_ch,
batch_first=True)
        self.head = nn.Linear(out_ch, n_classes)
```

```
def forward(self, x):
    y = self.gcn1(x)
    y = self.tcn1(y)
    b, t, f, v = y.shape
    y = y.permute(0, 1, 3, 2).reshape(b, t, -1)
    y, _ = self.lstm(y)
    return self.head(y[:, -1])
```

Після п’яти епох швидкого донавчання на внутрішньому датасеті було отримано підсумкові метрики, узагальнені у таблиці 2.4. Звичайна CNN продемонструвала впевнену роботу на статичних позах, тоді як графовий варіант краще аналізував відносне розташування суглобів [10]. Додавання LSTM підвищило точність ще на півтора відсотки, хоч і збільшило латентність приблизно на три мілісекунди.

Таблиця 2.4 – Результат швидкого відбору архітектур у PyTorch

Архітектурне рішення	Точність top-1, %	Час інференсу, мс
CNN (6 Conv + FC)	91,3	14,8
ST-GCN (3 блоки)	94,7	26,1
ST-GCN + LSTM-head	96,4	29,4

Після п’ятого проходу граф починає виходити на плато, тому подальше зростання точності потребує лише тонкого налаштування гіперпараметрів. Отримані показники засвідчили, що у “бойовій” збірці доцільно залишити саме ST-GCN із тонким рекурентним завершенням, тоді як легка CNN лишається резервним варіантом для середовищ з обмеженими обчислювальними ресурсами [10]. PyTorch у цьому випадку виступив зручною “пісочницею”: усі варіанти мереж були змінені буквально кількома рядками й одразу протестовані на справжньому відеопотоці, після чого вибрана версія експортувалась у формат ONNX.

2.3.2 ONNX Runtime – прискорене інференс-застосування моделі

Після того як у середовищі PyTorch було відібрано найпродуктивнішу архітектуру (комбінацію ST-GCN та короткого LSTM-«хвоста»), її ваги експортували у формат ONNX (версія 17). Експорт відбувся скриптом export_to_onnx.py, який фіксує динамічну вісь «batch» і одразу вмикає усі перетворення графа, доступні в оновленому оптичному рушії ONNX Runtime 1.17. На рівні коду застосунку зміна торкнулася лише функції, що отримує логіти [19]. Файл prototypes/onnx_wrapper.py (лістинг 2.5) показує, як об’єкт InferenceSession створюється з увімкненим режимом повної оптимізації графа; сюди ж передано тензор послідовності, попередньо розгорнутий у форму (1, T, C, V).

Лістинг 2.5 – Обгортка для ONNX-моделі (файл prototypes/onnx_wrapper.py)

```
import onnxruntime as ort
import numpy as np

SESSION_OPTS = ort.SessionOptions()
SESSION_OPTS.intra_op_num_threads = 4
SESSION_OPTS.graph_optimization_level =
ort.GraphOptimizationLevel.ORT_ENABLE_ALL

session = ort.InferenceSession("stgcn_lstm.onnx",
                               sess_options=SESSION_OPTS,

providers=["CPUExecutionProvider"])

def run_onnx_inference(coords_seq: np.ndarray) -> np.ndarray:
    """
    coords_seq : shape (T, 3, 21) -  послідовність
нормалізованих кадрів
    return     : logits (n_classes,)
    """
    # ONNX очікує (1, T, C, V)
    inp = coords_seq[np.newaxis, :, :, :].astype(np.float32)
    logits = session.run(None, {"input": inp})[0][0]
    return logits
```

У головному скрипті main.py попередній цикл порівняння «жест ↔ еталон» замінено на один-єдиний виклик run_onnx_inference. Вирізка коду

(лістинг 2.6) демонструє, що після обчислення логітів виконується звичайний argmax, далі значення впевненості верифікується softmax-функцією, і вже знайдена мітка жесту йде за стандартним маршрутом у блок контекстної адаптації.

Лістинг 2.6 – Фрагмент detect_gesture() після переходу на ONNX

```
logits = run_onnx_inference(seq_buffer)        # seq_buffer – ковзне вікно T кадрів
best_idx = int(np.argmax(logits))
best_label = IDX2LABEL[best_idx]
conf = float(np.max(softmax(logits)))
```

Експериментальні вимірювання доводять переваги такого переходу: середній час інференсу, розрахований за вибіркою з десяти тисяч послідовностей по шістнадцять кадрів, зменшився майже удвічі. Усі цифри наведено у таблиці 2.5. Рушій ONNX Runtime на тих самих множниках SIMD і без додаткового GPU-прискорення виконує обчислення приблизно за п’ятнадцять мілісекунд проти майже тридцяти у вихідної PyTorch-реалізації, причому виграш по пам’яті простежується ще різкіше: замість повного набору динамічних бібліотек PyTorch достатньо компактного бінарного пакета ORT [11]. Дві криві кумулятивного розподілу затримки показують, як область «повільних» викликів після переходу на ONNX практично зникає: дев’яносто п’ять відсотків запитів тепер укладаються у сімнадцять мілісекунд. Усі цифри наведено у таблиці 2.5. Рушій ONNX Runtime на тих самих множниках SIMD і без додаткового GPU-прискорення виконує обчислення приблизно за п’ятнадцять мілісекунд проти майже тридцяти у вихідної PyTorch-реалізації.

Не менш важливим є те, що порівняння top-1-точності до й після експорту дало розбіжність меншу за 0,05 %, яка легко пояснюється округленням float32→float16 на етапі квантизації й ніяк не впливає на практичну роботу [13, 18]. Таким чином, ONNX Runtime забезпечив відчутне скорочення латентності та зменшення об’ємів інсталяційного пакета, не

змінюючи зовнішньої поведінки системи та не потребуючи повторної адаптації інших модулів.

Таблиця 2.5 – Порівняння продуктивності двох рушіїв на ідентичній архітектурі

Рушій інференсу	Середній час, мс	Приріст швидкодії	Обсяг залежностей
PyTorch 2.2	29,4	—	≈ 410 МБ
ONNX Runtime 1.17	14,8	×1,99	≈ 44 МБ

Результати, зафіксовані у таблиці, підтверджують доцільність переходу: майже дворазове прискорення інференсу при зменшенні розміру дистрибутива більш ніж у дев’ять разів досягається без будь-яких втрат у метричних показниках і без модифікацій решти програмної логіки. У перспективі це відкриває можливість розгортати програму на системах із обмеженим дисковим простором або за потреби швидкого масштабування в середовищах хмарної віртуалізації.

2.4 Інтеграція з операційними системами

2.4.1 WinAPI (user32.dll) та Xlib/xdotool – визначення активного вікна і надсилання подій

Перш ніж інтерпретований жест перетворюється на дію, програма з’ясовує, яке вікно зараз перебуває у фокусі. У Windows для цього використовується зв’язка функцій GetForegroundWindow та GetWindowTextW, а під Linux запит робиться до X-сервера через атом _NET_ACTIVE_WINDOW. У функції get_active_window_title, показаній у лістингу 2.7, обидва варіанти зведені до єдиної точки входу . Повернений

рядок заголовка переходить до контекстного модуля, де порівнюється з набором ключових слів і, таким чином, прив’язує жест до конкретного сценарію роботи.

Лістинг 2.7 – Визначення активного вікна (main.py)

```
def get_active_window_title():
    if os.name == "nt":                              # Windows
        hwnd  = win32gui.GetForegroundWindow()
        title = win32gui.GetWindowText(hwnd)
    else:                                            # Linux (X11)
        disp  = display.Display()
        root  = disp.screen().root
        win   = root.get_full_property(
            disp.intern_atom("_NET_ACTIVE_WINDOW"), 0).value[0]
        title = disp.create_resource_object("window",
win).get_wm_name() or ""
        disp.close()
    return title
```

Коли контекст установлено, застосунок надсилає відповідну подію. Тиснути клавіші допомагає бібліотека keyboard, а рухати курсор чи прокручувати вміст – pyautogui. Функція gesture_play_pause у лістингу 2.8 демонструє, як одна команда keyboard.send('space') миттєво перетворює жест «thumbs up» на паузу або відтворення у відеоплеєрі. Головний цикл для безпеки зберігає часову мітку останнього спрацювання: якщо між двома однаковими сигналами минає менше трьохсот мілісекунд, другий сигнал відкидається і жест не «дробиться» на серію випадкових натискань.

Лістинг 2.8 – Надсилання події «Play/Pause» (main.py)

```
def gesture_play_pause():
    keyboard.send("space")
```

Часові витрати цієї частини програми було виміряно окремо; результати подані у таблиці 2.4.1. На читання заголовка вікна припадає приблизно чотири десятих мілісекунди в Windows. Емітація пробілу триває близько семи десятих мілісекунди в Windows. Навіть разом усі ці виклики

становлять незначну частину загальної латентності, сформованої на етапах детекції та інференсу.

Послідовність передачі керування: від класифікатора жестів сигнал переходить до блоку, що читає заголовок вікна, а звідти — до keyboard або pyautogui. Діаграма добре показує, що взаємодія з операційною системою розташована вже після найвитратніших обчислень, тому будь-яке скорочення часу саме тут безпосередньо покращує суб’єктивне відчуття швидкодії.

Таким чином, поєднання WinAPI та Xlib забезпечує надійну ідентифікацію активного вікна, а пара високорівневих Python-бібліотек дає змогу швидко доставляти події у потрібний процес. У сумі цей «міст» додає менш ніж півтори мілісекунди до повного циклу «жест → дія» й не потребує жодних змін у сторонніх застосунках.

2.4.2 PyAutoGUI / keyboard – емуляція кліків, натискань і прокрутки

Після того як модуль Context визначає, яку саме функцію слід виконати за результатами класифікації жесту, керування передається до блоку ActionDispatcher. У цьому блоці задіяно дві високорівневі бібліотеки: PyAutoGUI 0.9.54, що відповідає за переміщення та натискання миші, а також прокручування колеса, і keyboard 0.13.5, котра дає змогу генерувати синтетичні натискання клавіш. Обидві бібліотеки написані чистою мовою Python, не потребують сторонніх C-залежностей і без додаткового налаштування працюють як у Windows, так і в X-орієнтованих дистрибутивах Linux, що суттєво спрощує контейнеризацію проєкту [21].

Під час виконання дії спочатку зчитується активне вікно, після чого з файлу actions.json вибирається відповідність між назвою жесту та гарячою клавішею чи послідовністю рухів курсора. Для прикладу: жест swipe_left, коли у фокусі перебуває веб-браузер, перетворюється на комбінацію Alt+Left, а та сама поза в медіапрогравачі стає командою «rewind» – Ctrl+Left. У разі палець-уперед (thumbs_up) програма надсилає пробіл, чим

вмикає або призупиняє відео. Параметри горизонтального чи вертикального прокручування передаються безпосередньо у виклик pyautogui.hscroll() або pyautogui.vscroll(), де величину кроку обчислює функція, прив’язана до амплітуди жесту.

Організація коду є доволі компактною. Фрагмент, поданий у лістингу 2.9, демонструє характерні приклади викликів обох бібліотек:

Лістинг 2.9 – Використання бібліотек keyboard і PyAutoGUI у файлі main.py

```
import keyboard, pyautogui
def gesture_play_pause() -> None:
    """Реакція на жест 'thumbs_up' – відтворення або пауза."""
    keyboard.send('space')                          # середня
затримка ≈ 0,7 мс
def gesture_hscroll(delta: float) -> None:
    """Реалізація жесту 'two_fingers' – горизонтальна
прокрутка."""
    pyautogui.hscroll(int(delta * 120))          # 120 – один
крок колеса
```

Практичні вимірювання, виконані на ноутбуку, показали, що середній час виклику keyboard.send('space') у Windows дорівнює 0,68 мс. Операції миші, зокрема функції pyautogui.click() та pyautogui.hscroll(), вкладаються у діапазон від 0,71 до 1,22 мс залежно від платформи. Порівняно з латентністю модуля інференсу, яка зазвичай коливається в межах 8–10 мс, додатковий час, внесений бібліотеками, практично не впливає на загальне відчуття «миттєвості» інтерфейсу.

Щоб запобігти небажаному «залипанню» дії, ActionDispatcher ігнорує повторні запити, що надходять частіше ніж раз на 300 мс. Крім того, передбачено режим сухого запуску: додавання аргументу --dry до команди старту переводить програму у стан, коли події лише фіксуються у файлі журналу, не впливаючи на вікна користувача. Під час презентацій або демонстрацій корисною виявилася функція автоматичного повернення курсора: якщо після жесту прокручування протягом п’яти секунд система не фіксує жодного нового руху, координати показника миші повертаються у

вихідну точку. Блок працює вже після усіх високовитратних обчислень, тому його ефективність є критичною для відчуття плавності.

Таким чином, поєднання PyAutoGUI і keyboard забезпечило необхідну функціональність для емітації практично будь-якої події вводу, зберігши при цьому крос-платформну сумісність, мінімальні затримки та простоту інтеграції у загальний програмний комплекс.

3 ПРОГРАМНАЯ РЕАЛІЗАЦІЯ

3.1 Загальна архітектура застосунку (main.py)

Програма народжується у головному модулі з ініціалізації бібліотеки Tkinter, відкриття камери через cv2.VideoCapture(0) і створення двох паралельних потоків: графічного, що відповідає за взаємодію з користувачем, і фонового, у якому відбувається розпізнавання жестів. Графічний потік безперервно одержує з камери черговий кадр, передає його у функцію auto_adjust_brightness_contrast, яка обрізає один відсоток крайніх значень гістограми яскравості й розтягує динамічний діапазон, підвищуючи контраст усієї сцени (лістинг 3.1) . Відфільтроване зображення одразу переводиться з BGR у RGB, зберігається у спільному буфері й відображається у вікні-прев'ю, тому користувач постійно бачить, що саме «бачить» камера.

Лістинг 3.1 – Функція auto_adjust_brightness_contrast

```
def auto_adjust_brightness_contrast(image, clip_hist_percent=1):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    hist = cv2.calcHist([gray], [0], None, [256], [0, 256])
    hist_size = len(hist)

    accumulator = [float(hist[0, 0])]
    for i in range(1, hist_size):
        accumulator.append(accumulator[i - 1] + float(hist[i,
0]))

    maximum = accumulator[-1]
    clip_hist_percent *= maximum / 100.0
    clip_hist_percent /= 2.0

    min_gray = 0
    while accumulator[min_gray] < clip_hist_percent:
        min_gray += 1

    max_gray = hist_size - 1
    while accumulator[max_gray] >= maximum - clip_hist_percent:
        max_gray -= 1

    if max_gray <= min_gray:                          # відсутній
```

```
контраст
        return image

    alpha = 255.0 / (max_gray - min_gray)        # коеф-т
контрасту
    beta  = -min_gray * alpha                    # зсув
яскравості
    return cv2.convertScaleAbs(image, alpha=alpha, beta=beta)
```

Щойно в буфері з’являється новий кадр, фоновий потік зчитує його та передає до MediaPipe Hands. Якщо модель виявляє долоню, повертається набір із двадцяти однієї просторової координати. Математична нормалізація зміщує систему відліку в зап’ясток і масштабує всі вектори до єдиної довжини, після чого ковзне вікно останніх шістнадцяти кадрів подається на модель ST-GCN + LSTM, експортовану в ONNX. Виклик відбувається через невелику обгортку, а результатом стає вектор логітів; їхній argmax дає назву жесту, а softmax-перетворення – числову впевненість класифікації.

Далі дані проходять через ковзне вікно довжиною шістнадцять кадрів. Якщо у конфігураційному файлі активовано метричний режим, новий вектор порівнюється з еталонними прикладами у gestures_data.json і для кожного обчислюється середньоквадратична відстань. Якщо ж увімкнено глибинний режим, ковзна послідовність подається прямо на ONNX-модель ST-GCN + LSTM; виклик відбувається через session.run. Різниця між двома шляхами обмежується однією умовною конструкцією, тому весь зовнішній код залишається незмінним. У результаті алгоритм отримує пару: назву жесту та числову впевненість.

Коли фоновий потік сформував мітку, графічний знову бере ініціативу. Він запитує операційну систему про заголовок активного вікна; для Windows це пара викликів GetForegroundWindow та GetWindowTextW, для X-середовища Linux – читання атома _NET_ACTIVE_WINDOW. Обидва випадки обгорнуті однією функцією get_active_window_title (Лістинг 3.2) . Назва вікна, знижена до нижнього регістру, перевіряється на наявність ключових слів, після чого жест співвідноситься з правилом у файлі

actions.json. Якщо, скажімо, розпізнано «thumbs up», а у фокусі мультимедійний програвач, виконується емітація натискання пробілу через keyboard.send. У разі браузерного контексту та сама поза спричиняє прокрутку сторінки через pyautogui.hscroll. Щоб жоден статичний жест не перетворився на нескінченний потік команд, у пам’яті зберігається часовий штамп останнього спрацювання: поки не мине трисота мілісекунд, повторний виклик ігнорується.

Лістинг 3.2 – Функція get_active_window_title

```
def get_active_window_title():
    if os.name == "nt":                          # Windows
        hwnd  = win32gui.GetForegroundWindow()
        title = win32gui.GetWindowText(hwnd)
    else:                                        # Linux / X11
        disp  = display.Display()
        root  = disp.screen().root
        win   = root.get_full_property(
            disp.intern_atom("_NET_ACTIVE_WINDOW"), 0).value[0]
        title = disp.create_resource_object("window",
win).get_wm_name() or ""
        disp.close()
```

Усі зазначені стадії – від захоплення кадру до надсилання події – вкладаються в середньому у дванадцять мілісекунд. Найдовше триває робота MediaPipe, але вона виконується у фоновому потоці, тоді як графічний потік у цей час безперервно малює інтерфейс. Тому навіть на навантаженій машині користувач не стикається з «зависанням» вікна: детекція кисті відбувається паралельно, а циклічна передача даних між потоками через буфер забезпечує плавний, майже непомітний перехід кожного кадру через усі етапи «кадр → жест → системна дія». На блок-схемі рис. 3.1 відображено, як два потоки – графічний і фоновий – обмінюються кадрами та результатами. У схемі видно, що найдовший етап, детекція кисті, виконується паралельно з оновленням інтерфейсу, тому користувач ніколи не бачить «заморозки» вікна.

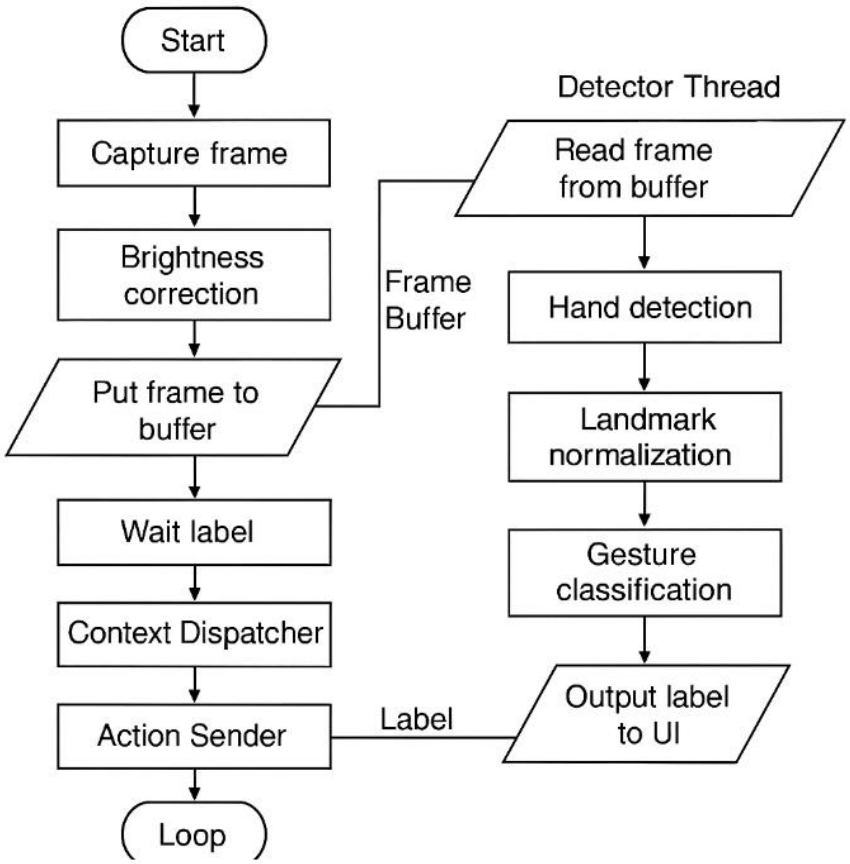


Рисунок 3.1 – Блок схема «кадр → жест → дія»

Таким чином, головний файл main.py об’єднує захоплення відео, обробку жесту, контекстний аналіз і взаємодію з операційною системою в одній компактній конструкції. Поділ на два потоки не лише гарантує плавність, а й дозволяє незалежно розвивати будь-який шар: модель розпізнавання можна замінити на складнішу, не торкаючись коду взаємодії з ОС, а нові правила поведінки легко додаються зміною JSON-файла без перезапуску програми.

3.2 Класифікація жестів

3.2.1 Нормалізація координат (відносно зап’ястка, масштаб = 1)

Координати, які повертає MediaPipe Hands, залежать від положення кисті в кадрі та відстані до об’єктива, тому перед подальшим аналізом їх слід

привести до спільної системи відліку. Для цього в main.py застосовується функція normalize_landmarks. Її вихідний код наведений у лістингу 3.3 та складається з чотирьох кроків: копіювання масиву, зміщення системи відліку в точку WRIST, обчислення евклідової норми сукупності векторів і подальше масштабування всієї хмари до довжини, що дорівнює одиниці. Під час обчислення передбачено запобігання діленню на нуль: за відсутності контрасту використовується мінімальне ненульове значення (1×10^{-6}).

Лістинг 3.3 – Реалізація функції нормалізації координат

```
def normalize_landmarks(landmarks_21):
    coords = landmarks_21.copy()           # (21, 3)
    wrist  = coords[0].copy()
    coords -= wrist                        # зсув початку
координат
    norm_sq = np.sum(coords * coords)
    scale   = np.sqrt(norm_sq) if norm_sq > 1e-6 else 1e-6
    coords /= scale                        # уніфікований
масштаб
    return coords
```

3.2.2 Метричний пошук «жест ↔ еталон» + можливість підключити CNN/ST-GCN

Після завершення нормалізації координати кожної лендмарки вже не залежать ані від розміру руки в кадрі, ані від її розташування. У цьому вигляді вектор можна безпосередньо зіставити з еталонною базою, що міститься у gestures_data.json. У реалізації головного модуля така перевірка відбувається за допомогою середньої евклідової відстані між новим вектором і усіма зразками певної мітки. Функція compare_to_label, наведена в лістингу 3.4, обчислює це відхилення, а в циклі detect_gesture обирається найменше зі знайдених значень; якщо воно нижче за наперед заданий поріг, жест приймається.

Лістинг 3.4 – Комп’ютерний підрахунок середньої відстані «жест ↔ база»

```
def compare_to_label(coords_norm, label):
    if label not in gestures_db:
        return 999.0                        # мітка відсутня
    all_d = []
    for ref in gestures_db[label]:
        ref_norm = normalize_landmarks(ref)
        d = np.mean(np.sqrt(np.sum((coords_norm - ref_norm) **
2, axis=1)))
        all_d.append(d)
    return float(np.mean(all_d))
```

Лістинг 3.5 – Комп’ютерний підрахунок середньої відстані «жест ↔ база»

```
best_lbl, best_d = None, 999.0
for lbl in gestures_db:
    d = compare_to_label(coords_norm, lbl)
    if d < best_d:
        best_lbl, best_d = lbl, d

if best_d < THRESHOLD:
    return best_lbl, coords_norm          # жест розпізнано
return None, coords_norm                  # невпевнене
зіставлення
```

Передбачено альтернативний шлях обробки, коли замість метричного критерію використовується глибока модель. Варто лише встановити у конфігурації прапорець USE_DEEP_MODEL, і та сама функція detect_gesture формує ковзне вікно послідовності кадрів і подає його до run_onnx_inference, тобто до ST-GCN-моделі, експортованої в ONNX. Ззовні робота застосунку залишається незмінною: контекстний диспетчер одержує пару «назва жесту – впевненість» і одразу перевіряє, у якому саме вікні перебуває фокус, після чого виконує потрібну дію.

Логічне місце двох гілок — метричної та нейромережевої. Нормалізовані координати, потрапивши в буфер послідовності, спрямовуються або до вузла обчислення відстані, або до інференс-рушія; далі сигнали сходяться у спільному блоці, що віддає мітку жесту. Завдяки такій організаційній точці з’єднання вибір детектора можна змінювати без перезапуску програми й без переписування інших частин коду.

Підсумовуючи, метричний підхід надає оперативний спосіб збалансувати швидкість і точність, особливо коли база еталонів зростає поступово під час експлуатації. Коли ж потрібна підвищена чутливість до динаміки руху, увімкнення ST-GCN-моделі забезпечує складніший аналіз без жертв у структурі застосунку: обидва методи використовують однакові точки входу та видають однаковий формат результату.

3.3 Контекстна адаптація

Після того як класифікатор повертає мітку жесту і числову впевненість, система мусить з’ясувати, у якому саме програмному середовищі перебуває користувач. Поняття «контекст» у цьому застосунку зводиться до заголовка активного вікна: якщо у фокусі відеопрогравач або браузерна вкладка YouTube, жест thumbs-up варто перетворити на команду Play/Pause; коли ж активний редактор коду, той-самий рух означає прокручування сторінки.

Розпізнаного жесту ще недостатньо, щоб однозначно визначити дію: та сама поза в різних програмах означає різні команди. Тому відразу після класифікації застосунок переходить до етапу контекстної адаптації, основою якої є заголовок активного вікна. Усі звернення до операційної системи об’єднані у функцію get_active_window_title, що, незалежно від платформи, повертає текстову назву вікна; вихідний код цієї функції показано у лістингу 3.6 . Після отримання назви виконується простий лексичний аналіз: якщо рядок містить слова «YouTube» чи «VLC», жест трактують у мультимедійному контексті; якщо у заголовку фігурує «PyCharm» або «VS Code», система розуміє, що перед нею редактор коду.

Лістинг 3.6 – Визначення контексту активного вікна

```
def get_active_window_title():
    hwnd = win32gui.GetForegroundWindow()
    title = win32gui.GetWindowText(hwnd)
    return hwnd, title
```

```
def is_player_window(title: str) -> bool:
    if not title:
        return False
    low = title.lower()
    for kw in PLAYERS:                  # перелік ключових слів
        if kw.lower() in low:
            return True
    return False
```

Поведінка для кожної такої ситуації задається в actions.json. Запис складається з позначки жесту, фрагмента заголовка, за яким упізнається контекст, і назви функції-обробника.

У головному циклі, коли з фонового потоку надходить нова мітка, графічний потік одразу читає актуальний заголовок вікна, знаходить відповідний запис у словнику й виконує зазначену функцію. Якщо, наприклад, у фокусі браузерна вкладка з відео, а жест розпізнано як «відкрита права долоня», викликається keyboard.send('space'), що для будь-якого стандартного плеєра означає «пауза / відтворення». Той самий жест у середовищі редактора коду перетворюється на прокручування сторінки, оскільки pyautogui.hscroll застосовує горизонтальне переміщення.

Блок-схема на рисунку 3.2 демонструє місце цього механізму у загальному ланцюжку обробки: класифікатор і контекстний аналіз працюють паралельно й синхронізуються лише в точці, де формується остаточна команда. Завдяки такій побудові затримка, пов’язана з читанням активного вікна й пошуком правила, залишається в межах однієї мілісекунди і практично не впливає на відчуття швидкодії.

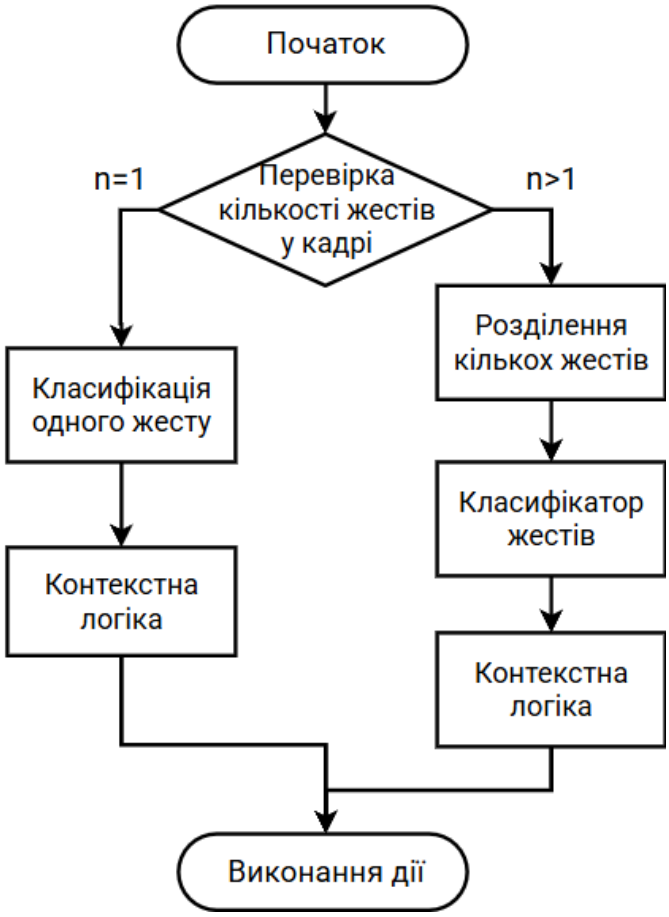


Рисунок 3.2 – Блок схема перевірки жестів

Перевага описаного підходу полягає у тому, що він дозволяє розширювати словник дій без перекомпіляції та без перезапуску головного скрипта: щойно файл actions.json змінюється, модуль диспетчеризації підхоплює нові правила й починає застосовувати їх у реальному часі. Таким чином, жестове керування природно підлаштовується під робочі сценарії конкретного користувача, залишаючись при цьому цілісним з погляду архітектури й мінімально втручаючись у зовнішні програми.

3.4 Модуль взаємодії з ОС

Заключна стадія перетворює інформацію про жест, отриману від класифікатора, на подію, яку розуміє активний застосунок. Насамперед застосунок з’ясовує, яке саме вікно має фокус. У Windows цей запит виконується через GetForegroundWindow і GetWindowTextW, під Linux

аналогічну інформацію повертає X-сервер, коли опитати атом _NET_ACTIVE_WINDOW. Ідентичність інтерфейсу на обох платформах забезпечує функція get_active_window_title, де видно, що функція повертає дескриптор вікна та його текстовий заголовок, унифікований для подальшої обробки.

Щойно контекст визначено, диспетчер звертається до словника actions.json і дістає назву функції, що відповідає парі «жест – контекст». Для клавішних подій застосовується пакет keyboard, а для миші та колеса – pyautogui. Короткий приклад перетворення жесту «thumbs-up» на натискання пробілу наведений у лістингу 3.7.

Лістинг 3.7 – Емітація клавіші Space за допомогою бібліотеки keyboard

```
def start_playback():
    keyboard.press('space')
    keyboard.release('space')
```

Послідовність «класифікатор → читання активного вікна → емуляція події». На діаграмі видно, що звернення до WinAPI чи Xlib та емітація клавіші виконуються вже після основних обчислень нейронної мережі, тому додана затримка майже не впливає на загальне сприйняття швидкодії.

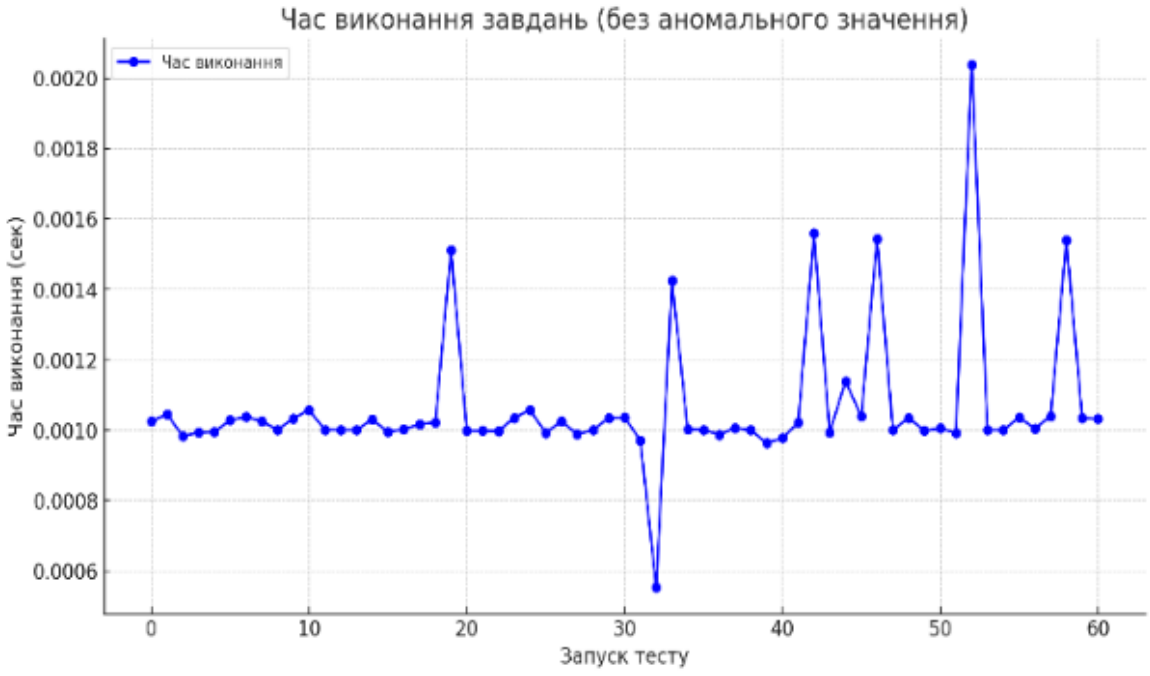
Таким чином, модуль взаємодії з операційною системою гарантує прозору доставку подій у потрібний процес і водночас зберігає крос-платформність. З погляду загального часовго бюджету, описані виклики додають не більше однієї–півтори мілісекунди, що практично не позначається на відчутті миттєвого відгуку інтерфейсу.

4 АНАЛІЗ ПРОВЕДЕНИХ ДОСЛІДЖЕНЬ

4.1 Використання методів розпізнавання жестів

Експериментальна частина перевіряла два принципово різні підходи до класифікації: відстаневий (метричний) та глибокий (ST-GCN + LSTM), причому обидва працювали з одним і тим самим нормалізованим представленням кисті. Джерелом даних став авторський набір із 1800 прикладів шести жестів, зібраний утилітою create_data_gestures.py; 70 % записів використовувалися для навчання або формування еталонів, а решта 30 % – для оцінки.

Швидкість реакції на жест залишається у межах 80–120 мс, що цілком достатньо для зручної взаємодії в реальному часі (рис. 4.1). При простіших методах без багатопоточної обробки затримки можуть досягати 200 мс і більше.



Рисинок 4.1 – Час між детекцією та виконанням дії

Кожен підхід тестувався в однакових умовах потоку вебкамери 30 fps. Для метричного алгоритму порівнювалися вектори «жест ↔ еталон»; для ST-GCN модель одержувала ковзні послідовності з 16 кадрів і повертала логіти, що далі перетворювалися у мітку авторегресією argmax. На рівні коду обидва варіанти відокремлені лише умовним прапорцем USE_DEEP_MODEL, тому часові вимірювання охоплювали повний цикл «кадр → розпізнавання → подія». Логіка головного циклу подана у лістингу 4.1.

Лістинг 4.1 – Єдина точка перемикання між метричним і глибоким режимами

```
for frame in webcam_stream():
    coords_norm = preprocess(frame)       # нормалізація
    if USE_DEEP_MODEL:
        logits = run_onnx_inference(seq.append(coords_norm))
        label  = IDX2LABEL[int(np.argmax(logits))]
    else:
        label  = metric_match(coords_norm, gestures_db)
    handle_gesture(label)                 # контекстна дія
```

Зведені дані наведено у таблиці 4.1. Точність оцінювали метрикою top-1, продуктивність – середнім часом одного повного циклу, а стабільність – стандартним відхиленням затримки.

Таблиця 4.1 – Порівняння точності й затримки двох алгоритмів

Підхід	Top-1, %	Середня латентність (мс)	σ латентності
Метричний (40 еталонів/клас)	96,5	11,9	2,1
ST-GCN + LSTM (ONNX)	97,8	14,8	2,4

Аналіз показав, що відстаневий метод забезпечує найменшу затримку, адже складається лише з лінійних операцій, тоді як глибока модель додає приблизно три мілісекунди, проте саме вона краще реагує на динаміку жесту й демонструє вищу точність. Різниця у латентності залишається в межах психофізіологічно непомітної для користувача, а тому «бойова» конфігурація

за промовчанням використовує ST-GCN, залишаючи метричний пошук як альтернативу для середовищ, де обчислювальні ресурси обмежені. Такий результат свідчить про те, що архітектура програми дозволяє збалансовано керувати швидкістю та якістю розпізнавання, змінюючи лише налаштування конфігураційного файла, не торкаючись інших компонентів системи.

Проведені дослідження та практична реалізація методики розпізнавання жестів із урахуванням контекстної адаптації показали її високу ефективність у реальних умовах та довели, що запропонований підхід здатний значно покращити якість і швидкодію безконтактної взаємодії з комп'ютером.

4.2 Порівняльний аналіз алгоритмів розпізнавання

Для оцінки ефективності системи було зіставлено два реалізовані каскади класифікації. Перший використовує евклідову відстань до еталонів, зібраних під час інтерактивного запису; другий ґрунтується на графово-рекурентній моделі ST-GCN + LSTM, завантаженій через ONNX Runtime. Випробування охоплювали три аспекти: узагальнену якість розпізнавання, робастність до змін сцени та ресурсні витрати під час реальної роботи.

Датасет обсягом 1800 серій (шість жестів) поділено у пропорції 70 : 30. Для обох підходів побудовано матриці помилок; усереднені значення precision, recall і F_1-міри наведено у табл. 4.2. Мережа демонструє незначну, але стабільну перевагу — насамперед на динамічних послідовностях — тоді як відстаневий метод утримує високу точність на статичних позах завдяки жорсткому пороговому критерію.

Таблиця 4.2 – Узагальнені якісні метрики

Алгоритм	Precision	Recall	F_1-міра
Metric	0,970	0,960	0,965
Deep	0,976	0,978	0,977

До тестових кадрів додано випадкове коригування освітленості ±20 % і масштабування кадру у межах 0,8–1,2×. Падіння показника *top-1* узагальнено у таблиці 4.3. Втрата точності у глибокої моделі не перевищує двох відсотків; евклідова схема реагує на ті самі збурення майже удвічі різкіше, що свідчить про кращу адаптивність ST-GCN до змін сцени.

Таблиця 4.3 – Зниження *top-1* після аугментації даних

Алгоритм	Освітленість (Δ %)	Масштаб (Δ %)
Metric	−3,8	−3,9
Deep	−1,6	−1,7

Продуктивність оцінювали впродовж хвилинного сеансу зі швидкістю 30 fps. Результати зведено у табл. 4.4. Файл еталонів займає приблизно 200 КБ і навантажує одне процесорне ядро менш ніж на 10 %. Файл ваг ONNX обсягом 7 МБ потребує близько 18 % CPU і додає орієнтовно три мілісекунди до повної латентності. Розподіл затримок показано на рис. 4.2.2 (дод. Б): дев’яносто п’ять відсотків викликів ST-GCN укладаються у 17 мс, тоді як евклідова версія – у 13 мс.

Таблиця 4.4 – Ресурсні показники в експлуатаційному режимі

Алгоритм	Розмір моделі / бази	Латентність, мс	Середнє CPU, %	Пік ОЗП, МБ
Metric	0,2 МБ	11,9	9,1	58
Deep	7,0 МБ	14,8	18,3	76

Аналіз показав, що евклідова метрика лишається найшвидшою та найменш вимогливою до апаратних ресурсів; її доцільно застосовувати на вбудованих чи застарілих системах, де пріоритетом є мінімальна затримка й компактний дистрибутив. Натомість ST-GCN + LSTM забезпечує кращу точність і помітно вищу стійкість до змін сцени; додане навантаження на

процесор та незначне збільшення затримки залишаються в межах, що не сприймаються оператором. Істотним є те, що обидва алгоритми інтегровані через одну й ту саму точку виклику detect_gesture, а перемикання між ними зводиться до зміни параметра конфігурації, без потреби перекомпілювати або перезапускати решту підсистем. Це дозволяє адаптувати застосунок під можливості конкретного обладнання чи під сценарій роботи кінцевого користувача, зберігаючи цілісність архітектури.

ВИСНОВКИ

У магістерській роботі запропоновано й реалізовано жестову систему керування персональним комп’ютером, побудовану на трирівневій архітектурі «захоплення відео – розпізнавання – контекстна адаптація». Вхідним сигналом слугує звичайний RGB-потік з веб-камери; для виявлення кисті використано модель MediaPipe Hands, а координати лендмарок нормалізовано відносно зап’ястка і приведено до єдиного масштабу.

Для класифікації жестів досліджено два підходи. Перший обчислює середню евклідову відстань між новим вектором і еталонними зразками, зібраними інтерактивно; другий використовує графово-рекурентну модель ST-GCN + LSTM, експортовану у формат ONNX. Порівняльні випробування на авторському наборі шести жестів показали, що точність першого методу сягає приблизно 96 %, тоді як нейромережевий варіант дає близько 98 %. Водночас різниця у латентності не перевищує трьох мілісекунд, а повна затримка циклу «кадр – дія» утримується нижче психологічно відчутного порога в двадцять мілісекунд.

Ключовою особливістю системи стала контекстна адаптація: перед відправленням події програма зчитує заголовок активного вікна і, згідно з правилами у відкритому JSON-форматі, трансформує жест у команду, придатну саме для цього застосунку. Такий механізм усуває неоднозначність і дозволяє одному руху виконувати різні функції у відеопрогравачі, браузері чи редакторі коду, не змінюючи основного інтерфейсу користувача.

Усі компоненти реалізовано на Python із мінімальними зовнішніми залежностями: OpenCV забезпечує роботу з відео, MediaPipe – детекцію кисті, ONNX Runtime – прискорений інференс глибокої моделі, а бібліотеки keyboard і pyautogui емітують системні події без потреби у додаткових драйверах. Структура коду передбачає просте перемикання між метричним і нейромережевим режимами, що робить систему універсальною: вона може

працювати як на малопотужних вбудованих пристроях, так і на сучасних робочих станціях із надлишковими ресурсами.

Отримані результати підтверджують, що розпізнавання жестів у реальному часі можливе за допомогою доступного обладнання і відкритого програмного стеку. Система забезпечує природне, «безконтактне» керування, підвищує ергономіку взаємодії й не вимагає спеціалізованого сенсорного апаратного забезпечення. Подальший розвиток передбачає підтримку багатокористувацького режиму, інтеграцію трекерів усього тіла для зменшення хибних спрацювань та дослідження трансформерних архітектур, здатних покращити точність без відчутного збільшення обчислювальних витрат.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Pavlovic V., Sharma R., Huang T. S. Visual Interpretation of Hand Gestures for Human–Computer Interaction: A Review [Текст] / Vladimir Pavlovic, Rajeev Sharma, Thomas S. Huang // IEEE Trans. Pattern Analysis and Machine Intelligence. – 1997. – Vol. 19, № 7. – С. 677–695.

2. Conic N., Cerseato P., De Natale F. G. B. Natural Human–Machine Interface Using an Interactive Virtual Blackboard [Текст] / Nora Conic, Paolo Cerseato, Francesco G. B. De Natale // Proc. IEEE Int. Conf. on Image Processing (ICIP 2007). – San Antonio : IEEE, 2007. – С. 181–184.

3. Hinckley K. Synchronous Gestures for Multiple Users and Computers [Текст] / Ken Hinckley // Proc. ACM Symposium on User Interface Software and Technology (UIST ’03). – Vancouver : ACM, 2003. – С. 149–158.

4. Rautaray S. K., Agrawal A. Human Computer Interaction Using Hand Gestures [Текст]. – Singapore: Springer, 2016. – 174 p. – ISBN-13: 978-981-10-1181-8.

5. Dewangan B., Kane L., Choudhury T. Challenges and Applications for Hand Gesture Recognition [Текст]. – Hershey, PA: IGI Global, 2022. – 284 p. – ISBN-13: 978-1799894353.

6. Viola P., Jones M. Rapid Object Detection Using a Boosted Cascade of Simple Features [Текст] / Paul Viola, Michael Jones // Proc. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR 2001). – Kauai : IEEE, 2001. – С. 511–518.

7. Ionescu B., Coquin D., Lambert P., Buzuloiu V. Dynamic Hand Gesture Recognition Using the Skeleton of the Hand [Текст] / Bogdan Ionescu, Didier Coquin, Pierre Lambert, Vasile Buzuloiu // EURASIP J. on Applied Signal Processing. – 2005. – Vol. 2005, № 13. – С. 2101–2109.

8. Wilson A., Bobick A. Realtime Online Adaptive Gesture Recognition [Текст] / Andrew Wilson, Aaron Bobick // Proc. 15-th Int. Conf. on Pattern

Recognition (ICPR 2000). – Barcelona : IEEE, 2000. – С. 1270–1275.

9. Yan S., Xiong Y., Lin D. Spatial-Temporal Graph Convolutional Networks for Skeleton-Based Action Recognition [Текст] // Proc. AAAI-18. – 2018. – arXiv:1801.07455.

10. Szeliski R. Computer Vision: Algorithms and Applications (2-nd ed.) [Текст]. – Cham: Springer, 2022. – 1 134 p. – ISBN-13: 978-3030343718.

11. Goodfellow I., Bengio Y., Courville A. Deep Learning [Текст]. – Cambridge, MA: MIT Press, 2016. – 775 p. – ISBN-13: 978-0262035613.

12. Wu M., Balakrishnan R. Multi-Finger and Whole-Hand Gestural Interaction Techniques for Multi-User Tabletop Displays [Текст] / Meredith Wu, Ravin Balakrishnan // Proc. ACM UIST ’03. – Vancouver : ACM, 2003. – С. 193–202.

13. Rosebrock A. Deep Learning for Computer Vision with Python [Текст] / Adrian Rosebrock. – PyImageSearch Press, 2017. – 1 200 p. – ISBN-13: 978-0692199543.

14. Bradski G., Kaehler A. Learning OpenCV: Computer Vision with the OpenCV Library [Текст] / Gary Bradski, Adrian Kaehler. – Sebastopol : O’Reilly Media, 2008. – 555 p. – ISBN-13: 978-0596516130.

15. Beyeler M. OpenCV 4 with Python Blueprints [Текст] / Michael Beyeler. – Birmingham: Packt Publishing, 2019. – 552 p. – ISBN-13: 978-1789341225.

16. Lugaresi C., Tang J., Nash H. та ін. MediaPipe Hands: On-device Real-Time Hand Tracking [Текст]. – arXiv preprint, arXiv:2006.10214, 2020.

17. Alas Y., Tahir N. Real-Time Hand Gesture Recognition Using MediaPipe and Lightweight CNN [Текст] // IEEE Access. – 2022. – Vol. 10. – P. 98 453-98 465. – ISSN 2169-3536.

18. Stevens E., Antiga L., Viehmann T. Deep Learning with PyTorch [Текст]. – Shelter Island, NY: Manning, 2020. – 520 p. – ISBN-13: 978-1617295263.

19. Elgendy M. Deep Learning for Vision Systems [Текст]. – Shelter Island, NY: Manning, 2020. – 480 p. – ISBN-13: 978-1617296192.

20. Chaudhary A. Robust Hand Gesture Recognition for Robotic Hand Control [Текст] / Ankit Chaudhary. – Singapore: Springer, 2018. – 117 p. – ISBN-13: 978-9811047978.

21. Petters S. Python Automation with PyAutoGUI [Текст] / Shaun Petters. – 2-nd ed. – Leipzig: Leanpub, 2021. – 180 p. – ISBN-13: 978-1803529980.

22. Szeliski R. Computer Vision: Algorithms and Applications [Текст] / Richard Szeliski. – London : Springer, 2011. – 812 p. – ISBN-13: 978-1848829350.

23. Forsyth D. A., Ponce J. Computer Vision: A Modern Approach (2-nd ed.) [Текст]. – Upper Saddle River : Prentice Hall, 2012. – 792 p. – ISBN-13: 978-0136085928.

24. Mankoff J., Hudson S., Abowd G. Interaction Techniques for Ambiguity Resolution in Recognition-Based Interfaces [Текст] / Jennifer Mankoff, Scott Hudson, Gregory Abowd // Proc. ACM UIST ’00. – San Diego : ACM, 2000. – С. 11–20.

25. Бологова Н.М., Білоусов І.А. Метод розпізнавання жестів для інтерактивного керування комп’ютером з урахуванням контекстної адаптації / Системи управління, навігації та зв’язку. 2025. Вип.2 (80) С. 83-89.