Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту
(повна назва)
Кафедра Інформатики
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

МЕТОДИ ОПТИМІЗАЦІЇ АЛГОРИТМІВ ТА СТРУКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ ВИКОРИСТАННЯ РЕСУРСІВ І ПОРТАТИВНОСТІ
(тема)

Виконав:
здобувач 4 року навчання,
групи ІТІНФ-22-3
Заіка Д. В.
(прізвище, ініціали)

Спеціальність 122 Комп'ютерні науки
(код і повна назва спеціальності)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва освітньої програми)

Керівник ст. викл. Кобилін І. О.
(посада, прізвище, ініціали)

Допускається до захисту

Завідувач кафедри інформатики _____ Кобилін О. А.
(підпис) (прізвище, ініціали)

2026 р.

Харківський національний університет радіоелектроніки

Факультет Інформаційно-аналітичних технологій та менеджменту

Кафедра Інформатики

Рівень вищої освіти перший (бакалаврський)

Спеціальність 122 Комп'ютерні науки
(код і повна назва)

Тип програми освітньо-професійна

Освітня програма Інформатика
(повна назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«____»_____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

здобувачеві Заіці Данилу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Методи оптимізації алгоритмів та структури програмного забезпечення для підвищення ефективності використання ресурсів і портативності

затверджена наказом університету від 26 травня 2026 року № 435Ст
2. Термін подання здобувачем роботи до екзаменаційної комісії 22 червня 2026 р.
3. Вихідні дані до роботи науково-методична та науково-технічна література, матеріали конференцій, дані інтернет-мережі.

4. Перелік питань, що потрібно опрацювати в роботі
1. Огляд актуальної ситуації оптимізації програмних застосунків
2. Моделювання структури ефективного програмного застосунку
3. Реалізація ефективного програмного застосунку

5. Перелік графічного матеріалу із зазначенням креслеників, схем, плакатів, комп'ютерних ілюстрацій (п.5 включається до завдання за рішенням випускової кафедри) діаграми моделювання застосунку, демонстраційні зображення роботи програми

6. Консультанти розділів роботи (п.6 включається до завдання 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

№ з/п	Назва етапів роботи	Строк / терміни виконання етапів роботи	Примітка
1	Отримання завдання на кваліфікаційну роботу	13.04.2026	
2	Аналіз завдання, підбір літератури	14.04.26-15.04.26	
3	Аналіз літератури з проблеми, що вирішується	16.04.26-19.04.26	
4	Аналіз існуючих методів оптимізації	20.04.26-22.04.26	
5	Формування кроків вирішення проблеми	23.04.26-05.05.26	
6	Програмна реалізація	26.04.26-20.05.26	
7	Аналіз отриманих результатів	21.05.26-31.05.26	
8	Оформлення пояснювальної записки	01.06.26-03.06.26	
9	Перевірка на нормоконтроль	10.06.26-11.06.26	
10	Перевірка на плагіат	12.06.26-14.06.26	
11	Рецензування	15.06.26-17.06.26	
12	Підготовка презентації та доповіді	18.06.26-22.06.26	
13	Занесення роботи в електронний архів	02.07.26	
14	Попередній захист кваліфікаційної роботи	02.07.26	

Дата видачі завдання 13 квітня 2026 р.

Здобувач _____
(підпис)

Керівник роботи _____ _____ст. викл. Кобилін І. О._____
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ/ABSTRACT

Пояснювальна записка до кваліфікаційної роботи: 70 с., 2 табл., 10 рис., 30 джерел.

ОПТИМІЗАЦІЯ, ПРОЦЕДУРНА ГЕНЕРАЦІЯ, ГРАФІЧНИЙ ЗАСТОСУНОК, ОБРОБКА ЗОБРАЖЕНЬ, SFML, C++.

Об’єктом роботи є процес обробки растрових зображень.

Метою роботи є розроблення ефективного програмного застосунку, із урахуванням принципів оптимізації алгоритмів використання обчислювальних ресурсів.

Під час роботи використано: інструменти розробки програмного забезпечення із графічним інтерфейсом (C++, Visual Studio, SFML).

Практична цінність застосунку полягає в демонстрації можливості створення ефективних алгоритмів обробки растрових зображень, що надають можливість процедурно створювати нові графічні об’єкти на основі наданих.

Розроблено застосунок, що забезпечує графічну зміни віхдних зображень та збереження нових.

Область застосування: навчальні програмні засоби, ігрова індустрія, системи редагування графіки.

OPTIMIZATION, PROCEDURAL GENERATION, GRAPHICS APPLICATION, IMAGE PROCESSING, SFML, C++.

The object of the work is the process of processing raster images.

The purpose of the work is to develop an effective software application, taking into account the principles of optimization of algorithms for the use of computing resources.

During the work, the following software development tools with a graphical interface were used: (C++, Visual Studio, SFML).

The practical value of the application lies in demonstrating the possibility of creating effective algorithms for processing raster images, which provide the ability to procedurally create new graphic objects based on the provided ones.

An application has been developed that provides graphical modification of output images and saving new ones.

Scope of application: educational software, gaming industry, graphics editing systems.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ШІ – штучний інтелект

API – Application Programming Interface (набір правил і протоколів)

BMP – Bitmap Image (формат зображення)

Bottleneck – вузьке місце

CI – Continuous Integration (безперервна інтеграція)

CD – Continuous Deployment (езперервне розгортання)

CUDA – Compute Unified Device Architecture (програмно-апаратна платформа)

DevOps – Development Operations (методологія розробки)

In-place – алгоритм, що працює без додаткової пам’яті

JPEG – Joint Photographic Experts Group (формат зображення)

Memoization – метод кешування результатів обчислень

MSVC – Microsoft Visual C++ (компілятор для Microsoft Visual Studio)

OpenCL – Open Computing Language (середовище для написання програм)

Pipeline – конвеєр обробки інструкцій процесора

PNG – Portable Network Graphics (формат зображення)

SFML – Simple and Fast Multimedia Library (бібліотека для розробки мультимедійних застосунків

SIMD – Single Instruction, Multiple Data (технологія паралельних обчислень)

ВСТУП

Програмне забезпечення є ключовим чинником забезпечення ефективного функціонування сучасних інформаційних систем. Зі зростанням складності програмних продуктів підвищуються вимоги до їхньої продуктивності, ефективності використання ресурсів та здатності працювати на різних апаратних і програмних платформах.

Однією з центральних проблем сучасної програмної інженерії є оптимізація алгоритмів і структури програмного забезпечення. Продуктивність систем значною мірою залежить від якості реалізації програмного коду. Неефективні алгоритми, невдало обрані структури даних або нераціональна організація доступу до пам’яті можуть призводити до втрат продуктивності та надмірного споживання ресурсів.

Сучасні підходи до оптимізації охоплюють не лише алгоритмічний рівень, а й архітектурні рішення, організацію структури програмного забезпечення, використання можливостей компіляторів та особливостей апаратного забезпечення.

Актуальність роботи полягає у зростанні вимог до ефективності програмного забезпечення в умовах обмежених ресурсів мобільних пристроїв, вбудованих систем, кросплатформених застосунків, а також у зв’язку зі збільшенням вартості на обчислювальні пристрої через розвиток штучного інтелекту. У таких умовах важливо забезпечити швидкодію та мінімізувати використання оперативної пам’яті, а також підтримувати роботу на різних операційних системах.

1 ТЕОРЕТИЧНІ ТА ПРАКТИЧНІ АСПЕКТИ ОПТИМІЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Актуальність підвищення ефективності використання обчислювальних ресурсів

1.1.1 Зростання вимог до продуктивності програмного забезпечення

Сучасний розвиток програмного забезпечення характеризується постійним зростанням вимог до продуктивності, що зумовлено як збільшенням обсягів оброблюваних даних, так і ускладненням алгоритмів. Програмні системи, які раніше могли ефективно працювати з невеликими наборами даних, сьогодні повинні обробляти значно більші масиви інформації в режимі, наближеному до реального часу.

Особливо це стосується інтерактивних застосунків, зокрема систем обробки зображень, де користувач очікує миттєвої реакції на будь-які зміни. У таких умовах навіть незначні затримки виконання операцій стають помітними та негативно впливають на загальний користувацький досвід. Це формує вимогу до максимально ефективного використання процесорного часу та оперативної пам’яті.

Розвиток апаратного забезпечення, включаючи багатоядерні процесори та великі обсяги оперативної пам’яті, не повністю компенсує зростання складності програмних задач. Причиною цього є те, що масштабування обчислювальних можливостей часто відстає від темпів зростання обсягів даних та вимог до якості обробки. У результаті ефективність програмного забезпечення все більше залежить від якості його реалізації.

Сучасні програмні системи часто працюють у середовищах із обмеженими ресурсами або повинні забезпечувати стабільну продуктивність на широкому спектрі пристроїв. Це додатково підсилює необхідність оптимізації.

1.1.2 Економічні фактори зростання вартості обчислювальних ресурсів

Додатковим чинником актуальності є зростання вартості обчислювальних ресурсів, зокрема оперативної пам’яті та твердотілих накопичувачів, що зумовлено підвищеним попитом з боку великих технологічних компаній та загальною тенденцією до збільшення обсягів даних. У сучасних умовах навіть локальні програмні застосунки змушені враховувати економічну ефективність використання ресурсів, оскільки апаратне масштабування не завжди є доцільним або доступним.

У результаті зростає значення програмної оптимізації як альтернативи апаратному розширенню. Раціональне використання пам’яті та процесорного часу дозволяє зменшити вимоги до системи без втрати функціональності, що є особливо важливим для інтерактивних застосунків та програм, орієнтованих на масове використання. Це також впливає на зниження енергоспоживання, що є додатковим фактором у мобільних та вбудованих системах.

Неефективне використання ресурсів у масштабованих системах може призводити до значних сукупних витрат при великій кількості користувачів. Тому навіть незначне покращення продуктивності або зменшення споживання пам’яті на рівні одного застосунку має кумулятивний ефект у масштабах системи або інфраструктури.

1.1.3 Вплив кросплатформеності на ефективність оптимізації

Необхідність забезпечення кросплатформеної портативності ускладнює процес оптимізації, оскільки рішення, які є ефективними для однієї апаратної або програмної архітектури, можуть демонструвати нижчу продуктивність на іншій [1, 2]. Це пов’язано з відмінностями у роботі процесорів, підсистем пам’яті, а також реалізаціях компіляторів і стандартних бібліотек.

У таких умовах розробник змушений шукати компроміс між максимальною продуктивністю та універсальністю коду [2–4]. Надмірно спеціалізовані оптимізації можуть давати виграш лише на окремих платформах, але при цьому погіршувати переносимість або навіть стабільність роботи програми в інших середовищах. Це особливо критично для програмного забезпечення, яке повинно працювати на різних операційних системах або апаратних конфігураціях.

Різні платформи можуть по-різному інтерпретувати однакові конструкції мови програмування з точки зору оптимізацій компілятора [1, 5]. Це означає, що фактична продуктивність алгоритмів може змінюватися без змін у вихідному коді, що ускладнює прогнозування поведінки системи.

Кросплатформеність виступає не лише вимогою сумісності, але й додатковим обмеженням у процесі оптимізації, змушуючи орієнтуватися на більш універсальні та стабільні рішення.

1.1.4 Проблема надлишкового споживання ресурсів у сучасному програмному забезпеченні

Сучасне програмне забезпечення все частіше характеризується тенденцією до збільшення споживання обчислювальних ресурсів, зокрема оперативної пам’яті, дискового простору та мережевого трафіку. Це явище отримало неформальну назву «software bloat» і є наслідком поєднання кількох факторів:

– зростання продуктивності апаратного забезпечення;

– пріоритет швидкості розробки над оптимізацією;

– використання високорівневих абстракцій і фреймворків.

У багатьох сучасних системах оптимізація на рівні розміру даних або ефективності зберігання відходить на другий план, оскільки розробники орієнтуються на скорочення часу виходу продукту на ринок. У результаті це

призводить до ситуацій, коли програмні продукти займають значні обсяги дискового простору та потребують суттєвих ресурсів навіть для виконання базових операцій.

Надмірне споживання ресурсів у сучасному програмному забезпеченні є суттєвою проблемою, що негативно впливає на користувацький досвід, продуктивність системи та загальну ефективність використання обчислювальних ресурсів.

1.2 Основні фактори впливу на продуктивність програмного забезпечення

Продуктивність програмного забезпечення залежить від взаємодії алгоритмічних, архітектурних і апаратних факторів.

До алгоритмічних факторів належать вибір алгоритмів і структур даних, які визначають асимптотичну складність обчислень і обсяг необхідних ресурсів. Невдалий вибір алгоритму може призвести до значного зростання часу виконання навіть при помірних обсягах даних.

Архітектурні фактори охоплюють організацію програмного коду: ефективність управління пам’яттю, структуру циклів, характер доступу до даних та використання паралельних обчислень.

Апаратні фактори зумовлені особливостями сучасних процесорів, зокрема конвеєризацією інструкцій, багаторівневою кеш-пам’яттю, механізмами передбачення розгалужень і підтримкою багатопоточності [1, 6, 7].

Важливим аспектом є просторово-часовий компроміс: збільшення використання пам’яті часто дозволяє суттєво скоротити час виконання, тоді як жорстка економія пам’яті підвищує обчислювальні витрати [8, 9].

Комплексне врахування зазначених факторів є необхідною умовою досягнення високої продуктивності програмного забезпечення та зумовлює

потребу в застосуванні різноманітних методів оптимізації на різних рівнях абстракції.

1.3 Еволюція підходів до оптимізації програмного забезпечення

У процесі розвитку програмної інженерії підходи до оптимізації програмного забезпечення зазнали суттєвих змін. Якщо раніше основна увага приділялася низькорівневій оптимізації коду та ефективному використанню обмежених ресурсів, то сучасні системи орієнтовані на масштабованість, швидкість розробки та економічну ефективність (табл 1.1).

Таблиця 1.1 – Зміна підходів до оптимізації

Аспект	**2000–2015**	**2015–2020**	**2020–2026**
Основна мета	Максимальна продуктивність	Баланс між швидкістю і розробкою	Бізнес-цінність
Підхід	Оптимізувати все	Оптимізувати критичні моменти	Оптимізувати тільки bottleneck-и
Хто оптимізує	Розробник вручну	Розробник + інструменти	Інструменти, платформи, ШІ
Рівень оптимізації	Низький рівень (алгоритми, кеші, SIMD)	Змішаний	Високий рівень (архітектура, cloud, data pipelines)
Пріоритети	Performance	Time-to-market	Time-to-market + scalability

Продовження таблиці 1.1

Аспект	**2000–2015**	**2015–2020**	**2020–2026**
Мови/стек	C/C++, Java	Python, JS	High-level + ШІ-інструменти
Автоматизація	Мінімальна	CI/CD, DevOps	Розробка із ШІ
Як знаходять проблеми	Досвід	Профайлери	Observability, telemetry, ШІ

Аналіз наведених даних свідчить про зміщення акцентів оптимізації з низькорівневих аспектів до архітектурних і системних характеристик. Сучасні підходи орієнтовані на виявлення вузьких місць і їх цільову оптимізацію, а також на використання автоматизованих інструментів [5, 10].

1.4 Оцінка практичних прикладів оптимізації програмних систем

1.4.1 Проєкт .kkrieger як приклад оптимізованого застосунку

Одним із ефективних підходів до зменшення обсягу споживаних ресурсів є використання процедурної генерації, яка полягає у створенні даних алгоритмічним шляхом під час виконання програми замість їх попереднього зберігання. Такий підхід дозволяє суттєво зменшити розмір програмного забезпечення та обсяг дискового простору, необхідного для його зберігання, оскільки замість великих масивів готового контенту використовуються алгоритми його побудови [11–15].

Яскравим практичним прикладом застосування процедурної генерації є проєкт Kkrieger. У цьому проєкті майже весь ігровий контент, включно з текстурами, геометрією рівнів та аудіоефектами, не зберігається у традиційному вигляді, а генерується алгоритмічно під час запуску програми. Завдяки такому підходу розмір виконуваного файлу був зменшений до кількох десятків кілобайт, що є одним із найбільш відомих прикладів

радикальної оптимізації використання дискового простору в історії розробки ігор.

Таким чином, Kkrieger демонструє потенціал процедурної генерації як методу оптимізації, що дозволяє значно мінімізувати обсяг статичних даних у програмному забезпеченні шляхом їх заміни на алгоритмічні правила формування контенту.

1.4.2 Гра Helldivers 2 як приклад неефективної оптимізації застосунку

Гра Helldivers 2 може бути розглянута як приклад застосунку, у якому спостерігаються проблеми з ефективністю використання дискового простору та організацією ігрових даних. Однією з ключових проблем, яка обговорювалася в спільноті користувачів, є значний обсяг встановлених файлів, що частково пов’язаний із дублюванням ігрових ресурсів.

У процесі розробки та оптимізації завантаження даних застосовуються техніки попередньої упаковки та дублювання файлів у різних архівах. Такий підхід використовується для прискорення доступу до ресурсів, оскільки дозволяє розміщувати потрібні дані ближче до точок їх використання в ігровому рушії та зменшувати затримки під час потокового завантаження.

Однак така стратегія призводила до суттєвого збільшення обсягу встановлених файлів, який у певний період міг сягати понад 100 ГБ. Згодом, після оптимізацій з боку розробників, розмір інсталяції було значно зменшено – приблизно до 23 ГБ, що свідчить про успішну роботу над оптимізацією структури ресурсів та усунення надлишкового дублювання.

Важливо зазначити, що попри збільшення розміру інсталяції, практичний виграш у продуктивності завантаження був відносно незначним. За оцінками користувачів і тестуваннями, різниця у часі завантаження рівнів або ресурсів становить лише кілька секунд у порівнянні з потенційно більш компактною структурою даних. Це свідчить про те, що обраний підхід не

забезпечує пропорційного покращення швидкодії відносно значного збільшення використання дискового простору.

1.5 Огляд існуючих інструментаріїв

1.5.1 Вибір мови програмування

Вибір мови програмування є одним із ключових стратегічних рішень при розробці програмного забезпечення, оскільки він безпосередньо впливає на продуктивність системи, складність реалізації, можливості оптимізації та подальший супровід проєкту. На практиці це рішення визначає компроміс між швидкодією виконання програмного коду, зручністю розробки та рівнем контролю над апаратними ресурсами.

Низькорівневі мови програмування, такі як C та C++, надають розробнику максимальний контроль над пам’яттю, процесором і структурою даних [3, 4, 16]. Це дозволяє реалізовувати високоефективні алгоритми з мінімальними накладними витратами. Завдяки можливості прямої роботи з пам’яттю, вказівниками та оптимізації на рівні компілятора, такі мови є оптимальним вибором для задач, де критично важлива продуктивність, зокрема у системному програмуванні, обробці зображень, ігрових рушіях та високонавантажених сервісах [1, 5, 17–19].

Разом із тим, використання низькорівневих мов пов’язане з підвищеною складністю розробки. Розробник повинен самостійно керувати виділенням та звільненням пам’яті, контролювати коректність доступу до даних та уникати типових помилок, таких як витоки пам’яті або некоректні вказівники. Це підвищує вимоги до кваліфікації розробника та збільшує час розробки.

На противагу цьому, високорівневі мови програмування, такі як Python, Java або JavaScript, орієнтовані на підвищення продуктивності розробки та зменшення кількості помилок [6, 20]. Вони надають зручні абстракції,

автоматичне керування пам’яттю та велику кількість готових бібліотек. Це дозволяє значно скоротити час створення програмного забезпечення та зосередитися на логіці задачі, а не на технічних деталях реалізації.

Однак такі переваги досягаються за рахунок додаткових накладних витрат під час виконання. Інтерпретація або віртуальні машини, що використовуються у високорівневих мовах, можуть знижувати продуктивність, особливо у задачах, пов’язаних із великими обсягами даних або інтенсивними обчисленнями.

Важливим аспектом є також екосистема мови програмування. Наявність розвиненої стандартної бібліотеки, підтримка спільноти, інструменти розробки та документація значною мірою впливають на ефективність розробки. У багатьох випадках саме доступність якісних бібліотек визначає доцільність вибору тієї чи іншої мови.

Крім того, необхідно враховувати вимоги до портативності програмного забезпечення. Деякі мови забезпечують кращу кросплатформеність завдяки використанню віртуальних машин або стандартизованих бібліотек, тоді як інші потребують додаткової адаптації під конкретні платформи.

У контексті задач, пов’язаних з обробкою растрових зображень, перевага зазвичай надається мовам, що забезпечують високий рівень контролю над пам’яттю та швидкодію виконання [21–23]. Це обумовлено необхідністю обробки великих масивів даних та виконання значної кількості однотипних операцій.

1.5.2 Використання бібліотек та інструментів

У сучасній розробці програмного забезпечення широке застосування знаходить використання сторонніх бібліотек та спеціалізованих інструментів, які дозволяють значно підвищити ефективність розробки та оптимізувати

продуктивність програмних систем. Замість реалізації базових компонентів з нуля, розробники можуть використовувати вже оптимізовані та протестовані рішення, що суттєво скорочує час розробки та зменшує ймовірність помилок.

Бібліотеки, як правило, містять реалізації алгоритмів і структур даних, які були оптимізовані з урахуванням особливостей сучасного апаратного забезпечення. Це означає, що їх використання дозволяє досягти високого рівня продуктивності без необхідності детального опрацювання низькорівневих аспектів. Особливо це актуально для задач, пов’язаних із обробкою зображень, графікою, математичними обчисленнями та роботою з великими обсягами даних.

Окрім функціональних бібліотек, важливу роль відіграють інструменти аналізу продуктивності, зокрема профайлери. Вони дозволяють визначити, які частини програми споживають найбільше часу або ресурсів, і тим самим виявити вузькі місця [5, 24]. Це дає змогу зосередити зусилля оптимізації саме на критичних ділянках коду, що є більш ефективним, ніж спроби оптимізувати програму в цілому.

Сучасні інструменти профілювання можуть надавати детальну інформацію про:

– час виконання функцій;

– кількість викликів;

– використання пам’яті;

– кеш промахи;

– паралельність виконання.

Також важливими є інструменти статичного та динамічного аналізу коду, які дозволяють виявляти потенційні помилки ще на етапі розробки. Це сприяє підвищенню якості програмного забезпечення та зменшенню витрат на тестування.

Застосування бібліотек і інструментів також позитивно впливає на портативність програмного забезпечення. Кросплатформені бібліотеки забезпечують єдиний інтерфейс для роботи з різними операційними

системами, що дозволяє уникнути необхідності реалізації платформозалежного коду. Це значно спрощує процес розробки та супроводу програмних систем.

Разом із тим, використання сторонніх компонентів має і певні обмеження. Зокрема, надмірна залежність від бібліотек може ускладнювати контроль над продуктивністю та збільшувати розмір програмного забезпечення. Крім того, деякі бібліотеки можуть містити зайвий функціонал, що не використовується у конкретному проєкті, але впливає на споживання ресурсів.

Тому важливо здійснювати обґрунтований вибір бібліотек, враховуючи їх продуктивність, розмір, активність підтримки та відповідність вимогам проєкту. У деяких випадках доцільним є поєднання використання сторонніх бібліотек із власною реалізацією критичних компонентів.

1.5.3 Вибір середовища програмування

Вибір середовища програмування є важливим етапом у процесі розробки програмного забезпечення, оскільки він безпосередньо впливає на продуктивність розробника, швидкість реалізації функціоналу, зручність налагодження та загальну якість коду. Сучасні середовища розробки представляють собою комплексні системи, які об’єднують редактор коду, компілятор або інтерпретатор, засоби відлагодження, профілювання та інтеграції з системами контролю версій [25].

Використання інтегрованого середовища розробки дозволяє значно підвищити ефективність роботи програміста за рахунок автоматизації рутинних операцій. Зокрема, функції автодоповнення коду, підсвічування синтаксису, автоматичне форматування та рефакторинг сприяють зменшенню кількості помилок і прискорюють процес написання програмного

коду. Це особливо важливо у великих проєктах, де обсяг коду є значним, а підтримка його читабельності критично необхідна [5, 20].

Окрему роль відіграють вбудовані засоби налагодження, які дозволяють виконувати покрокове виконання програми, відстежувати значення змінних та аналізувати потік виконання. Це значно спрощує процес пошуку логічних помилок, які складно виявити лише шляхом аналізу вихідного коду. У поєднанні з інструментами профілювання це дає можливість не лише виправляти помилки, але й оптимізувати продуктивність застосунку.

Сучасні середовища розробки також часто містять інтегровані засоби аналізу продуктивності, які дозволяють виявляти вузькі місця у виконанні програми. Такі інструменти можуть аналізувати використання процесора, пам’яті, частоту викликів функцій, а також поведінку кешу. Це дозволяє більш точно визначати ділянки коду, які потребують оптимізації, що є важливим у контексті розробки високопродуктивних застосунків [1, 5].

Серед найбільш поширених середовищ програмування можна виділити Visual Studio, CLion та Visual Studio Code, які підтримують різні мови програмування та платформи. Їх вибір залежить від специфіки проєкту, використовуваних технологій та вимог до продуктивності розробки. Наприклад, для системного програмування та задач, пов’язаних з високою продуктивністю, часто використовуються середовища з глибокою інтеграцією з компіляторами та профайлерами.

Важливим аспектом є також інтеграція середовища розробки з іншими інструментами, такими як системи контролю версій, системи автоматичної збірки та тестування. Це дозволяє створювати безперервний процес розробки, що підвищує стабільність програмного забезпечення та зменшує кількість помилок на етапі релізу.

Середовище програмування є невід’ємною частиною інструментального забезпечення розробника, яке суттєво впливає як на

швидкість створення програмного забезпечення, так і на якість кінцевого продукту.

1.6 Постановка задачі

Процедурне створення графічних зображень є актуальним завданням у контексті розвитку ігрової індустрії. Тому ставиться завдання створити графічний застосунок, що дозволить ефективно створювати нові зображення майже на будь якій операційній системі.

Об’єктом роботи є процес обробки растрових зображень.

Метою роботи є розроблення ефективного програмного застосунку, із урахуванням принципів оптимізації алгоритмів використання обчислювальних ресурсів.

Для досягнення мети необхідно виконати такі завдання:

– реалізувати базові операції модифікації зображення, такі як зміна кольорових характеристик та масштабування;

– створити інтерактивне керування параметрами обробки через графічний інтерфейс із миттєвим відображенням результатів;

– зберігати нове зображення у файловій системі.

2 АНАЛІЗ ВИМОГ ТА ПРЕДМЕТНОЇ ОБЛАСТІ

2.1 Аналіз предметної області

2.1.1 Особливості обробки растрових зображень

`

Обробка растрових зображень є важливою складовою сучасних програмних систем, що працюють із графічною інформацією. На відміну від векторної графіки, растрові зображення представляють собою дискретну структуру, де кожен елемент (піксель) має фіксовані координати та значення кольору. Це визначає як гнучкість їх використання, так і обмеження, пов’язані з продуктивністю обробки.

З технічної точки зору, растрове зображення зберігається у вигляді лінійного масиву байтів, організованого у row-major порядку. Така організація означає, що елементи одного рядка розташовані послідовно в пам’яті. Це дозволяє ефективно використовувати кеш-пам’ять процесора при послідовному обході даних [21, 22, 26].

Особливістю задач обробки зображень є їхня обчислювальна інтенсивність. Кожна операція, навіть найпростіша, виконується над усіма пікселями. Наприклад, при роздільній здатності 1920 на 1080 необхідно обробити понад 2 мільйони елементів. Якщо врахувати, що кожен піксель містить кілька компонент кольору, кількість елементарних операцій зростає в декілька разів. Цю залежність можна представити у вигляді формули:

$$N = W \cdot H \cdot C, \tag{2.1}$$

де N – кількість операцій;

W – ширина зображення;

H – висота зображення;

C – кількість каналів у пікселі над якими проводяться операції.

У межах даного застосунку має бути реалізовано обмежений, але показовий набір операцій:

– зміна кольору зображення;

– відображення результату в реальному часі;

– обрізання області зображення;

– збереження результату у файл.

На відміну від складних графічних систем, де використовуються фільтри або трансформації, у даній роботі акцент буде зроблено на базових операціях, що дозволяє більш детально дослідити їх ефективність.

2.1.2 Класифікація операцій обробки зображень

Операції обробки растрових зображень можна класифікувати за характером впливу на піксельні дані та ступенем зміни структури зображення. Такий поділ дозволяє систематизувати алгоритми та визначити їх обчислювальну складність, вимоги до пам’яті та особливості реалізації.

Однією з базових категорій є точкові операції, які передбачають незалежну обробку кожного пікселя без врахування його сусідів. До таких операцій належать зміна яскравості, контрасту, інверсія кольорів та інші перетворення, що виконуються над окремими піксельними значеннями. Їх основною особливістю є простота реалізації.

Наступною категорією є локальні (сусідні) операції, у яких значення кожного пікселя обчислюється на основі його оточення. Такі операції використовуються у фільтрах згладжування, підвищення різкості, виявлення контурів та інших задачах, де важлива просторова взаємодія пікселів. Їх реалізація є більш обчислювально складною порівняно з точковими операціями через необхідність додаткових звернень до пам’яті.

Окрему групу становлять геометричні операції, які змінюють просторову структуру зображення. До них відносяться масштабування,

поворот, трансформації координат та обрізання. Такі операції можуть вимагати створення нового буфера зображення, оскільки змінюють розташування пікселів у просторі.

Також виділяють глобальні операції, які враховують інформацію про все зображення. Прикладами є побудова гістограми, нормалізація інтенсивності та корекція динамічного діапазону. Вони потребують попереднього аналізу всього масиву пікселів перед виконанням основного перетворення.

2.1.3 Формати зберігання растрових зображень

Формат зберігання растрових зображень безпосередньо впливає на спосіб їх обробки, швидкодію алгоритмів та вимоги до обчислювальних ресурсів. Оскільки растрове зображення являє собою масив пікселів, різні формати визначають як спосіб кодування цих даних, так і можливість їх ефективного декодування для подальшої обробки.

У сучасних програмних системах найбільш поширеними є формати BMP, PNG та JPEG, які суттєво відрізняються за принципами зберігання інформації. Формат BMP є найбільш простим з точки зору структури, оскільки зазвичай зберігає піксельні дані у майже незжатому вигляді. Це забезпечує швидкий доступ до пікселів, однак призводить до значного збільшення обсягу файлів.

Формат PNG використовує безвтратне стиснення, що дозволяє зменшити розмір файлів без втрати якості зображення. При цьому дані перед обробкою повинні бути декодовані, що створює додаткові обчислювальні витрати. Незважаючи на це, PNG є зручним у задачах, де важлива точність представлення кольорів та відсутність артефактів стиснення.

Формат JPEG, у свою чергу, базується на алгоритмах із втратами якості, що дозволяє досягати значного зменшення розміру файлів. Однак під

час декодування відбувається відновлення наближеного представлення зображення, що може призводити до появи артефактів, особливо на контрастних межах. Це може впливати на результати подальших операцій обробки пікселів, оскільки початкові дані вже є апроксимацією оригіналу.

З точки зору продуктивності обробки ключовим є не лише формат зберігання, але й етап декодування. Перед виконанням операцій над зображенням воно має бути приведене до уніфікованого внутрішнього представлення у вигляді суцільного масиву пікселів у пам’яті [1, 7]. Саме це представлення використовується у подальших алгоритмах обробки, незалежно від початкового формату файлу.

Таким чином, вибір формату зберігання впливає як на швидкість завантаження та підготовки зображення до обробки, так і на якість вхідних даних, що в подальшому визначає коректність результатів алгоритмів.

2.1.4 Вимоги до ефективності використання ресурсів

При розробці інтерактивних графічних застосунків особливу увагу приділяють ефективності використання ресурсів, оскільки це безпосередньо впливає на якість досвіду користувача.

Одним із ключових параметрів є час виконання операцій. Оскільки користувач взаємодіє із зображенням у режимі реального часу, будь-яка затримка може бути помітною. Практично це означає, що обробка повинна виконуватися в межах кількох десятків мілісекунд.

Іншим важливим аспектом є використання оперативної пам’яті. Зображення займають значний обсяг пам’яті, тому неефективне управління нею може призвести до перевантаження системи. Особливо це актуально при роботі з кількома зображеннями одночасно.

Крім того, необхідно враховувати, що програма може запускатися на різних пристроях із різними характеристиками. Це означає, що алгоритми

повинні бути достатньо універсальними і не покладатися на специфічні апаратні можливості.

Також важливим фактором також стабільність роботи. Програма повинна коректно обробляти всі дії користувача, не допускати помилок і забезпечувати передбачувану поведінку.

Таким чином, ефективність у даному контексті визначається не лише швидкістю виконання, але й збалансованим використанням усіх доступних ресурсів.

2.1.5 Постановка задачі оптимізації

На основі проведеного аналізу можна сформулювати задачу оптимізації як комплексну проблему, що охоплює як алгоритмічні, так і архітектурні аспекти.

Основна мета полягає у забезпеченні максимально швидкої обробки зображень при мінімальному використанні пам’яті та збереженні простоти реалізації. Це передбачає пошук оптимальних рішень на рівні організації циклів, структури даних та взаємодії компонентів системи.

До ключових напрямків оптимізації належать:

– скорочення кількості елементарних операцій;

– оптимізація доступу до пам’яті;

– уникнення зайвих копіювань даних;

– раціональний вибір структур даних;

– забезпечення незалежності від конкретної платформи.

Особливістю задачі є те, що оптимізація не повинна ускладнювати програму. Надмірно складні рішення можуть ускладнити підтримку коду та знизити його надійність.

У результаті необхідно досягти компромісу між продуктивністю, простотою реалізації та портативністю.

2.2 Теоретичні основи оптимізації алгоритмів

2.2.1 Аналіз складності алгоритмів

Аналіз складності алгоритмів є фундаментальним інструментом теоретичної інформатики, що дозволяє оцінювати ефективність алгоритмічних рішень незалежно від конкретної апаратної платформи та особливостей реалізації.

Розрізняють часову складність (кількість елементарних операцій) і просторову складність (обсяг додаткової пам’яті). Для опису асимптотичної поведінки алгоритмів при великих обсягах вхідних даних найчастіше застосовується нотація «велике O» [8, 9].

Використання нотації O(f(n)) дає змогу порівнювати алгоритми: наприклад, алгоритм зі складністю O(n) демонструє кращу масштабованість порівняно з O(n log n) при обробці великих масивів даних.

$$O(n) < O(n\log n) < O(n^2), \tag{2.2}$$

де $O(n)$ – лінійна складність алгоритму;

$O(n\log n)$ – лінійно-логарифмічна складність алгоритму;

$O(n^2)$ – квадратична складність алгоритму.

Важливе значення має також розгляд найкращого, середнього та найгіршого випадків виконання. У практичних системах найгірший випадок часто визначає гарантовані характеристики продуктивності та стабільність роботи під максимальним навантаженням.

Таким чином, аналіз складності є ключовим етапом на стадії проєктування, який дозволяє обґрунтовано відбирати ефективні алгоритмічні рішення.

2.2.2 Вибір алгоритмів для конкретних задач

Вибір алгоритму в практичній розробці є інженерним компромісом, що виходить за межі формальної асимптотичної оцінки.

Основними факторами впливу виступають:

– характер і статистичний розподіл вхідних даних;

– обсяг даних та частота виконання операцій;

– обмеження середовища виконання (обсяг доступної пам’яті, вимоги до затримок, можливості апаратного паралелізму).

Для задач з невеликими обсягами даних або рідкісними операціями доцільніше використовувати простіші алгоритми з низькими константними накладними витратами. У високонавантажених і масштабованих системах пріоритет віддається рішенням з оптимальною асимптотичною ефективністю.

Отже, обґрунтований вибір алгоритму вимагає глибокого аналізу реальних умов експлуатації програмного забезпечення.

2.2.3 Компроміс між швидкодією та використанням пам’яті

У процесі оптимізації програмного забезпечення часто виникає необхідність свідомого балансування між часом виконання та обсягом споживаної пам’яті (space-time trade-off).

Типовими прикладами такого компромісу є:

– застосування мемоїзації (memoization) та динамічного програмування;

– вибір між in-place алгоритмами та алгоритмами, що використовують додаткові структури даних;

– попереднє обчислення та зберігання проміжних результатів.

У серверних і вебсистемах пріоритет зазвичай надається швидкодії, тоді як у вбудованих системах критичним є мінімальне споживання ресурсів.

Оптимальний баланс завжди залежить від контексту задачі, архітектури системи та апаратних обмежень.

2.3 Обчислювальні ресурси

2.3.1 Оперативна пам’ять та організація доступу до даних

Ефективне використання оперативної пам’яті та раціональна організація доступу до даних є ключовими факторами продуктивності програмних систем. Оскільки в системі обробки зображень самі зображення представлені у вигляді великих масивів пікселів, неефективне управління пам’яттю або хаотичний доступ до даних може суттєво знижувати швидкодію алгоритмів.

У процесі проєктування системи варто враховувати необхідність мінімізації навантаження на підсистему управління пам’яттю. Це досягається за рахунок зменшення кількості динамічних виділень та повторного використання вже існуючих буферів для зберігання зображень. Такий підхід дозволяє знизити витрати на алокацію та зменшити ризик фрагментації пам’яті при тривалому виконанні програми.

Окрему увагу приділено контролю тимчасових структур даних. Надмірне створення проміжних об’єктів у процесі обробки може призводити до додаткових витрат пам’яті та збільшення навантаження на систему управління пам’яттю, тому їх використання мінімізується.

Додатковим аспектом оптимізації є зменшення фрагментації оперативної пам’яті, яка виникає при частому виділенні та звільненні блоків різного розміру. Фрагментація призводить до неефективного використання доступного обсягу пам’яті та може негативно впливати на стабільність роботи системи при тривалому виконанні. Для зменшення цього ефекту використовується підхід повторного використання вже виділених структур замість постійного створення нових.

Важливим елементом оптимізації є попереднє резервування пам’яті для структур даних, розмір яких може змінюватися під час виконання програми. Це дозволяє уникнути багаторазових перевиділень пам’яті та зменшує кількість копіювань даних при розширенні контейнерів або буферів зображень.

У випадках інтенсивної роботи з однотипними об’єктами може застосовуватися концепція пулів пам’яті, яка полягає у попередньому виділенні великого блоку пам’яті з подальшим його розподілом під потреби програми. Це зменшує кількість звернень до стандартного алокатора та підвищує ефективність повторного використання пам’яті.

Варто враховувати баланс між використанням стекової та динамічної пам’яті. Стекова пам’ять забезпечує швидший доступ і автоматичне звільнення ресурсів, однак має обмежений обсяг і не придатна для зберігання великих структур, таких як зображення. Динамічна пам’ять є більш гнучкою, але потребує додаткового контролю з боку програми. Поєднання цих підходів дозволяє забезпечити ефективне використання оперативної пам’яті, зменшити накладні витрати та підвищити стабільність роботи при обробці зображень.

2.3.2 Центральний процесор

Центральний процесор є одним із ключових компонентів обчислювальної системи, від якого безпосередньо залежить продуктивність програмного забезпечення. Оптимізація виконання програм на цьому рівні спрямована на ефективне використання внутрішніх механізмів процесора, таких як конвеєризація, організація доступу до пам’яті, передбачення розгалужень і багатоядерна обробка. Ці підходи дозволяють підвищити швидкодію без зміни загальної логіки алгоритмів.

Конвеєризація є однією з ключових архітектурних особливостей сучасних процесорів, що забезпечує підвищення пропускної здатності за рахунок паралельного виконання різних стадій обробки інструкцій. Типовий конвеєр включає такі етапи:

– вибірка інструкцій;

– декодування;

– виконання, доступу до пам’яті;

– запис результату.

Завдяки цьому процесор може одночасно обробляти кілька інструкцій, що перебувають на різних етапах виконання, значно підвищуючи загальну ефективність обчислень. Ефективність конвеєра залежить від регулярності потоку інструкцій і мінімізації конфліктів, які включають структурні конфлікти, конфлікти за даними та конфлікти керування. Для зменшення їх впливу застосовуються передбачення розгалужень, перепланування інструкцій та оптимізації на рівні компілятора.

Важливим фактором продуктивності є також принцип локальності даних, який включає часову та просторову локальність. Він визначає ефективність взаємодії програми з ієрархією пам’яті та безпосередньо впливає на кількість промахів кешу. Висока локальність дозволяє частіше використовувати швидку кеш-пам’ять процесора та зменшує потребу звернення до повільнішої оперативної пам’яті. Оптимізація цього аспекту досягається через використання компактних структур даних, таких як масиви замість зв’язаних списків, а також через впорядкування циклів і даних для забезпечення послідовного доступу. Низька локальність доступу часто є причиною суттєвого зниження продуктивності навіть у випадках ефективних алгоритмів.

Передбачення розгалужень дозволяє процесору завчасно визначати напрямок виконання умовних переходів. Це дає змогу уникати простоїв конвеєра та підтримувати його безперервну роботу. Точність передбачення суттєво впливає на загальну швидкодію системи, оскільки помилкові

передбачення призводять до скидання виконаних інструкцій і повторного заповнення конвеєра. З точки зору розробника програмного забезпечення важливо мінімізувати непередбачувані розгалуження у критичних ділянках коду, особливо в межах циклів, використовуючи більш стабільні структури керування виконанням.

Багатопоточність і паралельні обчислення дозволяють ефективно використовувати багатоядерні процесори. Найкраще масштабуються задачі, які допускають незалежне виконання підзадач [13, 27, 28]. Основними обмеженнями цього підходу є накладні витрати на створення та синхронізацію потоків, а також конкуренція за спільні ресурси, зокрема пам’ять і кеш. Ефективність паралельних обчислень значною мірою залежить від правильного проєктування алгоритмів і врахування апаратних особливостей системи.

2.3.3 CPU та GPU обробка

Важливим аспектом оптимізації є вибір між використанням центрального процесора (CPU) та графічного процесора (GPU). Обидва підходи мають свої переваги та обмеження, тому ефективність системи значною мірою залежить від правильного розподілу обчислювального навантаження.

CPU традиційно використовується для виконання послідовних або слабопаралельних обчислень. Він забезпечує високу гнучкість алгоритмів, зручність реалізації та простоту відлагодження. У контексті обробки зображень CPU є ефективним для виконання операцій з невеликою обчислювальною складністю, таких як точкові перетворення, обрізання зображень або керування структурою даних. Додатковою перевагою є відсутність накладних витрат на передачу даних між різними обчислювальними пристроями.

GPU, у свою чергу, оптимізований для масово-паралельних обчислень. Завдяки великій кількості обчислювальних ядер він здатний виконувати однакові операції над великою кількістю пікселів одночасно. Це робить його особливо ефективним для задач фільтрації, згладжування, трансформацій зображень та інших операцій, які мають високий рівень паралелізму [11].

Однак використання GPU пов’язане з певними обмеженнями. Основним з них є затримки передачі даних між оперативною пам’яттю та відеопам’яттю. У випадках, коли обсяг обчислень є відносно невеликим, витрати на копіювання даних можуть перевищувати виграш від паралельного виконання. Крім того, реалізація GPU-алгоритмів є складнішою з точки зору програмування та вимагає використання спеціалізованих технологій, таких як CUDA або OpenCL.

У розробленому застосунку доцільно застосовувати CPU-орієнтований підхід для операцій з невисокою обчислювальною складністю та мінімальним обсягом даних. Це дозволяє уникнути додаткових накладних витрат і забезпечити стабільну продуктивність на широкому спектрі пристроїв. GPU-обробка може розглядатися як потенційне розширення системи для більш складних операцій, однак не є обов’язковою для базового функціоналу.

2.4 Підходи до оптимізації

2.4.1 Проблеми передчасної оптимізації

Передчасна оптимізація є однією з проблем у процесі розробки програмного забезпечення, яка полягає у спробах покращити продуктивність коду до завершення його базової реалізації та перевірки коректності. Такий підхід може створювати ілюзію більш ефективного рішення, однак на практиці часто призводить до ускладнення структури програми без відчутного приросту швидкодії.

Основною небезпекою передчасної оптимізації є зміна початкової логіки програми ще до того, як вона повністю сформована та протестована. Це може ускладнювати подальшу розробку, оскільки код стає менш інтуїтивно зрозумілим, а його поведінка – менш передбачуваною. У результаті збільшується складність супроводу та внесення змін у систему [9, 29].

Ще одним негативним наслідком є підвищення залежності коду від конкретних припущень щодо його роботи. Заздалегідь оптимізовані рішення часто базуються на очікуваному характері навантаження або використання, які можуть не відповідати реальним умовам експлуатації. Це призводить до того, що такі оптимізації або не дають очікуваного ефекту, або навіть погіршують загальну поведінку системи.

Також варто враховувати, що надмірна складність коду, яка виникає внаслідок передчасних покращень, ускладнює його тестування та перевірку коректності. Окрім цього, значно погіршується читабельність програмного коду: логіка стає менш прозорою, з’являються додаткові умови, оптимізаційні хитрощі та неочевидні залежності між частинами програми. Це ускладнює розуміння коду іншими розробниками та підвищує ризик помилок при його подальшому супроводі.

Раціональний підхід передбачає початкове створення простої та зрозумілої реалізації, яка повністю виконує поставлені функціональні вимоги. Лише після цього доцільно розглядати можливі напрями оптимізації, коли вже відома реальна поведінка системи у практичних умовах використання.

Обмеження передчасної оптимізації дозволяє зберегти баланс між складністю програмного забезпечення та його розвитком, забезпечуючи більш контрольований і передбачуваний процес розробки.

2.4.2 Профілювання

Профілювання є одним із базових інструментів аналізу продуктивності програмного забезпечення, який дозволяє отримати об’єктивні дані про фактичну поведінку програми під час виконання. Його основна мета полягає у визначенні найбільш ресурсомістких ділянок коду, що є потенційними кандидатами для подальшої оптимізації.

На відміну від припущень щодо “повільних” частин програми, профілювання ґрунтується на вимірюваннях реального часу виконання, кількості викликів функцій, завантаженості процесора та інших метрик. Це дозволяє уникнути помилкових оптимізацій, коли зусилля спрямовуються на ділянки коду, які фактично не мають суттєвого впливу на загальну продуктивність [30].

У контексті обробки зображень профілювання є особливо важливим, оскільки значна частина обчислювального навантаження зосереджена у вузькому наборі операцій над пікселями. Саме тому навіть незначні зміни в реалізації циклів або доступі до даних можуть суттєво впливати на загальний час виконання [9, 21]. Профілювання дозволяє виявити такі гарячі точки та оцінити їх внесок у загальну продуктивність системи.

Додатковою перевагою профілювання є можливість оцінки ефективності різних підходів реалізації однієї й тієї самої функціональності. Порівняння результатів до і після змін дозволяє кількісно оцінити ефект оптимізації, а не покладатися лише на теоретичні припущення.

Профілювання є необхідним етапом процесу оптимізації, який забезпечує перехід від інтуїтивних оцінок до даних, заснованих на вимірюваннях, і дозволяє приймати обґрунтовані рішення щодо подальшого вдосконалення програмного забезпечення.

2.4.3 Поступовий підхід до оптимізації

Поступовий підхід до оптимізації передбачає поетапне вдосконалення програмного забезпечення після отримання коректної базової реалізації. Його основна ідея полягає в тому, що спочатку система повинна повністю виконувати всі функціональні вимоги, а лише після цього можуть застосовуватися методи підвищення продуктивності.

Такий підхід дозволяє уникнути ситуацій, коли передчасні спроби оптимізації ускладнюють архітектуру програми або вводять додаткові помилки. Наявність стабільної та перевіреної базової версії забезпечує контрольований процес змін, у якому кожне покращення можна оцінити окремо.

У практичному застосуванні поступова оптимізація зазвичай реалізується через ітераційний цикл: аналіз поведінки програми, визначення вузьких місць, внесення змін та повторне вимірювання результатів. Такий цикл повторюється доти, доки не буде досягнуто необхідного рівня продуктивності або поки подальші покращення не стануть недоцільними з точки зору витрат часу та складності.

Важливим аспектом є те, що оптимізація виконується локально, тобто впливає лише на конкретні ділянки коду, які були визначені як проблемні. Це дозволяє зберегти загальну структуру програми та мінімізувати ризик виникнення побічних ефектів.

Окремо слід зазначити, що поступовий підхід забезпечує кращу контрольованість змін. Кожне покращення може бути протестоване та виміряне окремо, що дозволяє чітко відстежувати вплив конкретних рішень на продуктивність системи.

Отже, поступова оптимізація є практичним і безпечним методом вдосконалення програмного забезпечення, який дозволяє досягати підвищення ефективності без втрати стабільності та читабельності коду.

2.4.4 Врахування особливостей компіляції

Ефективність виконання програмного забезпечення значною мірою залежить не лише від алгоритмічних рішень, але й від того, як вихідний код обробляється компілятором. Сучасні компілятори виконують велику кількість автоматичних оптимізацій, які можуть суттєво впливати на кінцеву продуктивність програми [5]. У зв’язку з цим важливо враховувати особливості компіляції вже на етапі написання коду.

Одним із ключових факторів є можливість автоматичного інлайнінгу функцій. Невеликі та часто використовувані функції можуть бути вбудовані безпосередньо у місця виклику, що дозволяє уникнути накладних витрат на виклик функції. Проте надмірно складна структура функцій або непрозорі залежності можуть обмежувати застосування такої оптимізації.

Важливу роль відіграє також оптимізація циклів, яка виконується компілятором автоматично за умови, що структура циклу є достатньо простою та передбачуваною. Зайві розгалуження всередині циклів, непрозорі умови завершення або складні залежності між змінними можуть ускладнювати або повністю блокувати такі оптимізації.

Окремо слід враховувати вплив рівня оптимізації компілятора, який задається параметрами збірки. Різні режими компіляції можуть змінювати баланс між швидкістю виконання та часом компіляції, а також впливати на використання пам’яті та розмір кінцевого виконуваного файлу. Це означає, що результати роботи програми можуть відрізнятися залежно від конфігурації збірки.

Ще одним важливим аспектом є дотримання принципів, які сприяють передбачуваності коду для компілятора. Чим простішою є структура даних і логіка обчислень, тим більше можливостей має компілятор для застосування низькорівневих оптимізацій, таких як векторизація або перестановка інструкцій.

Врахування особливостей компіляції дозволяє підвищити ефективність програмного забезпечення без зміни його функціональної логіки, забезпечуючи кращу взаємодію між кодом розробника та механізмами автоматичної оптимізації.

2.5 Забезпечення портативності програмного забезпечення

2.5.1 Використання кросплатформених технологій

При розробці програмного забезпечення важливим є забезпечення його незалежності від конкретної операційної системи. Це досягається шляхом використання кросплатформених технологій, які забезпечують єдиний інтерфейс доступу до системних ресурсів [2, 10].

До основних вимог, яким повинні відповідати такі технології, належать:

– платформна незалежність (можливість компіляції та виконання програми на різних операційних системах без змін у вихідному коді);

– уніфікований API (наявність стандартного набору функцій для роботи з графікою, введенням та виведенням);

– ефективність виконання (мінімальні накладні витрати при роботі з пам’яттю та графічними даними);

– простота інтеграції (можливість швидкого включення в проєкт без складної конфігурації);

– активна підтримка та документація (наявність спільноти, прикладів використання та оновлень).

Особливу роль відіграє підтримка роботи з графічними буферами та подіями, оскільки інтерактивні застосунки потребують швидкої обробки введення користувача та оперативного оновлення зображення.

Крім того, важливим є використання мов програмування та стандартів, які забезпечують переносимість коду між різними компіляторами. У цьому

контексті перевагу мають стандартизовані можливості мови, що не залежать від конкретної платформи або середовища виконання.

Отже, вибір кросплатформених технологій має ґрунтуватися не лише на їх функціональних можливостях, але й на здатності забезпечувати стабільну, ефективну та переносиму роботу програмного застосунку.

2.5.2 Обмеження та особливості портативності

Попри те, що сучасні програмні рішення часто розробляються з урахуванням кросплатформеності, досягнення однакової продуктивності на різних пристроях залишається складним завданням. Формальна сумісність не гарантує однакової швидкодії, оскільки реальні умови виконання можуть суттєво відрізнятися залежно від апаратної конфігурації та особливостей конкретної платформи.

На продуктивність програмного забезпечення впливають такі ключові фактори, як тактова частота процесора, кількість ядер та ефективність їх використання, а також ієрархія кеш-пам’яті. Зокрема, розмір кешу та швидкість доступу до нього можуть істотно змінювати поведінку алгоритмів, особливо тих, що активно працюють із великими обсягами даних. Не менш важливим є показник пропускної здатності оперативної пам’яті, який визначає швидкість обміну даними між процесором і пам’яттю.

Додатково слід враховувати відмінності в архітектурах процесорів, які можуть по-різному обробляти інструкції, оптимізації компілятора та навіть порядок виконання операцій. Це означає, що один і той самий алгоритм може демонструвати різну ефективність залежно від платформи, навіть за ідентичних вхідних даних.

У зв’язку з цим важливо розробляти алгоритми з урахуванням адаптивності – тобто здатності ефективно працювати в широкому діапазоні апаратних умов. Це може передбачати використання параметризованих

підходів, масштабування обчислень, а також уникнення надмірно специфічних оптимізацій, які працюють добре лише на окремих конфігураціях.

Отже портативність слід розглядати не лише як здатність запуску програми на різних платформах, а як комплексну характеристику, що включає стабільність продуктивності, передбачуваність поведінки та ефективне використання доступних ресурсів у різних середовищах виконання.

2.6 Проєктування об’єктно-орієнтованої моделі

2.6.1 Основні класи та їх взаємодія

Реалізація програмного застосунку базується на об’єктно-орієнтованому підході, який дозволяє структуризувати код та розподілити відповідальність між окремими компонентами системи.

Ключовим класом є main, який відповідає за ініціалізацію програми, запуск основного циклу виконання та координацію взаємодії між іншими компонентами. Він виступає центральним елементом системи та забезпечує її цілісну роботу.

Клас ImageProcessor реалізує алгоритми обробки зображення. Він виконує операції над зображеннями, змінюючи їхній стан відповідно до запитів користувача. Важливою особливістю є те, що цей клас не залежить від інтерфейсу користувача, що забезпечує гнучкість системи.

Клас GUI відповідає за взаємодію з користувачем, обробку подій і відображення результатів. Він приймає дії користувача, інтерпретує їх і передає відповідні команди до інших компонентів.

Взаємодія між класами організована таким чином, щоб мінімізувати залежності та забезпечити чітке розділення логіки. Це дозволяє спростити тестування, модифікацію та розширення програмного забезпечення [4, 16].

2.6.2 Діаграма варіантів використання

З точки зору користувача, функціональність програмного застосунку представлена набором базових сценаріїв, які відображають основні можливості системи.

Основним актором у системі є користувач, який взаємодіє з програмою через графічний інтерфейс. До ключових варіантів використання належать зміна кольору зображення, обрізання вибраної області, відновлення зображення та збереження отриманого результату (рис. 2.1).

Операція зміни кольору реалізується як миттєва трансформація значень пікселів із подальшим відображенням результату. Обрізання дозволяє виділити частину зображення та працювати лише з нею. Відновлення дозволить повернути зображення до початкового стану.

Завершальним етапом є збереження результату, що передбачає запис зміненого зображення у файл.

Кожен сценарій ініціюється користувачем через графічний інтерфейс, після чого система виконує відповідну послідовність внутрішніх операцій.

Такий набір функцій є мінімально достатнім для демонстрації основних принципів обробки зображень та дозволяє зосередитися на продуктивності й архітектурі.

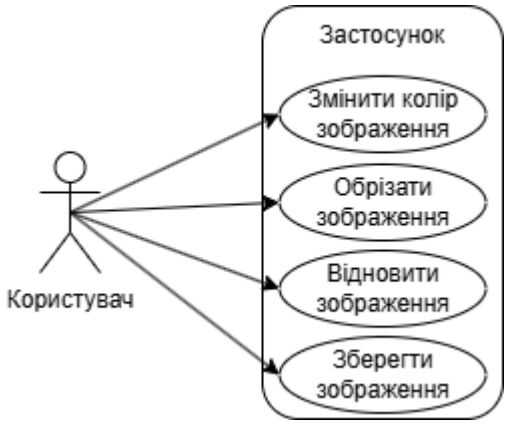

Рисунок 2.1 – Діаграма варіантів використання застосунку

2.6.3 Діаграма послідовності виконання операцій

Для детального розуміння взаємодії між компонентами системи доцільно розглянути послідовність виконання операцій (рис. 2.2).

Як показано на рисунку 2.2, процес починається з дії користувача, який обирає відповідну функцію через інтерфейс. Отримана подія передається до модуля обробки подій, де вона інтерпретується як команда.

Далі управління передається до компонента, відповідального за обробку зображення. Він виконує необхідні обчислення, змінюючи значення пікселів у відповідному об’єкті даних.

Після завершення обробки оновлені дані передаються до модуля відображення, який відповідає за візуалізацію результату. Зображення оновлюється у вікні програми, що дозволяє користувачу миттєво побачити результат виконаної операції.

Робота застосунку базуватиметься на класичній подієвій моделі, яка широко використовується в інтерактивних графічних системах. Цикл повторюватиметься безперервно до завершення роботи програми. Така модель дозволить забезпечити інтерактивність без блокування інтерфейсу, оскільки кожна ітерація циклу виконується за обмежений проміжок часу.

Важливим аспектом є асинхронний характер обробки подій. Це означає, що система реагуватиме лише на зміну стану, а не виконуватиме постійне активне опитування, що знижуватиме навантаження на процесор.

Такий підхід забезпечує послідовну та зрозумілу взаємодію між компонентами системи і дозволяє ефективно організувати процес обробки даних.

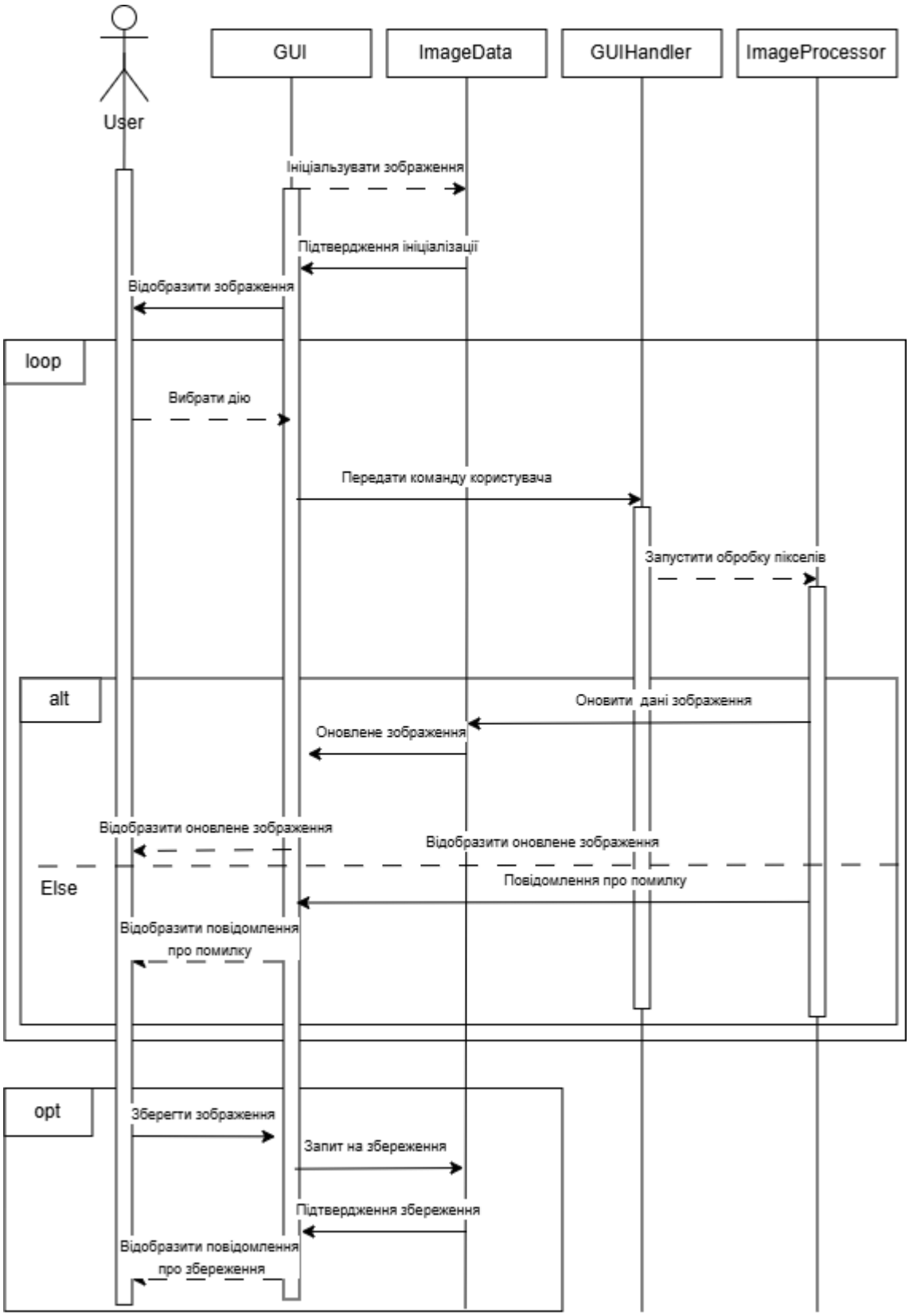

Рисунок 2.2 – Діаграма послідовності виконання операцій застосунку

3 ПРОЄКТУВАННЯ ТА РЕАЛІЗАЦІЯ ПРОГРАМНОГО ЗАСТОСУНКУ

3.1 Проєктування структури програмного забезпечення

3.1.1 Загальна архітектура програмного застосунку

Програмний застосунок будується із використанням модульного підходу, який передбачає поділ системи на окремі функціональні компоненти. Така організація дозволяє зменшити зв’язаність між частинами системи та забезпечити зручність подальшого розширення і супроводу [3, 4].

Архітектура системи включає такі основні компоненти:

– модуль обробки зображень;

– модуль графічного інтерфейсу користувача;

– центральний керуючий компонент (функція main), що реалізує цикл обробки подій;

– модуль роботи з файлами, реалізований засобами класу sf::Image.

Модуль обробки зображень відповідає за виконання операцій над пікселями, зокрема зміну кольору, перетворення у відтінки сірого, обрізання та змішування зображень. Він не залежить від інтерфейсу користувача, що забезпечує гнучкість і повторне використання.

Модуль графічного інтерфейсу користувача забезпечує взаємодію з користувачем, відображення елементів керування (кнопок, полів введення, палітри кольорів) та обробку подій миші і клавіатури.

Функція main виконує роль координатора: обробляє події, перевіряє стан елементів інтерфейсу (логічні прапорці) та викликає відповідні операції обробки зображень.

Такий розподіл відповідає принципу розділення відповідальностей і спрощує подальший розвиток програмного забезпечення.

3.1.2 Організація потоку виконання програми

Виконання програмного застосунку організовано за принципом циклу обробки подій, який є стандартним для інтерактивних графічних систем. Такий підхід забезпечує постійну взаємодію з користувачем і своєчасне оновлення відображення.

На початковому етапі відбувається ініціалізація необхідних ресурсів, включаючи створення графічного вікна та підготовку структур даних. Після цього виконується ініціалізація зображення, з яким буде працювати користувач.

Основний цикл програми складається з кількох етапів. Спочатку відбувається опитування та обробка подій, що надходять від користувача. Далі, залежно від отриманих команд, виконуються відповідні операції обробки зображення. Після завершення обчислень оновлюється стан даних, і результат передається до модуля відображення.

Завершальним етапом кожної ітерації є виведення оновленого зображення на екран. Цикл повторюється безперервно до завершення роботи програми.

Такий підхід дозволяє уникнути блокування інтерфейсу користувача та забезпечує плавну та передбачувану роботу застосунку навіть при виконанні обчислювально інтенсивних операцій.

3.2 Обґрунтування вибору інструментальних засобів реалізації

3.2.1 Обґрунтування вибору мови програмування C++

Вибір мови програмування для реалізації застосунку обробки зображень є критично важливим, оскільки він безпосередньо впливає на продуктивність, ефективність використання пам’яті та можливості оптимізації.

Мова C++ забезпечує високий рівень контролю над ресурсами системи, що є особливо важливим при роботі з великими обсягами даних, такими як масиви пікселів. На відміну від мов із автоматичним керуванням пам’яттю, C++ дозволяє явно керувати виділенням та звільненням пам’яті, що зменшує накладні витрати та підвищує передбачуваність виконання.

Ще однією перевагою є висока продуктивність виконання програм. Завдяки компіляції у машинний код та можливості оптимізації на рівні компілятора, C++ забезпечує швидке виконання циклів обробки, що є критичним для задач, пов’язаних із проходженням великих масивів даних.

Мова також надає ефективні засоби роботи з масивами та вказівниками, що дозволяє реалізувати обробку пікселів із мінімальними накладними витратами. Це особливо важливо для реалізації точкових операцій, де кожна зайва інструкція може впливати на загальний час виконання.

Крім того, стандарт C++ забезпечує переносимість коду між різними платформами та компіляторами, що відповідає вимогам кросплатформеності застосунку.

3.2.2 Обґрунтування вибору бібліотеки SFML

Для реалізації графічного інтерфейсу та взаємодії з користувачем необхідно використовувати бібліотеку, яка забезпечує зручні засоби роботи з вікнами, графікою та подіями.

Бібліотека SFML надає високорівневий інтерфейс для створення графічних застосунків і обробки подій, що дозволяє суттєво спростити процес розробки.

Однією з ключових переваг є простота використання. Бібліоткка є інтуїтивно зрозуміла, що дозволяє швидко реалізувати базову функціональність без значних витрат часу на налаштування.

SFML також забезпечує ефективну роботу з графічними буферами, що є важливим для відображення растрових зображень. Бібліотека дозволяє безпосередньо передавати масиви пікселів до графічної підсистеми, що мінімізує накладні витрати.

Додатковою перевагою є підтримка обробки подій, включаючи введення з клавіатури та миші, що є необхідним для інтерактивної взаємодії з користувачем.

Важливим фактором є також кросплатформеність бібліотеки, яка дозволяє використовувати її на різних операційних системах без зміни вихідного коду.

3.2.3 Обґрунтування вибору середовища Visual Studio

Для реалізації програмного застосунку використовується інтегроване середовище розробки Microsoft Visual Studio, яке є одним із найпоширеніших інструментів для створення програмного забезпечення мовою C++.

Visual Studio забезпечує повноцінний цикл розробки програмного продукту, включаючи написання коду, компіляцію, відлагодження та профілювання. Завдяки глибокій інтеграції з компілятором MSVC середовище дозволяє отримувати високопродуктивний машинний код, а також застосовувати оптимізації на етапі компіляції, що є важливим для задач обробки великих обсягів даних.

Вбудований відлагоджувач, дозволяє виконувати покрокове виконання програми, контролювати значення змінних та аналізувати потік виконання. Це значно спрощує процес пошуку та виправлення помилок у складних алгоритмах.

Окрему роль відіграють інструменти аналізу продуктивності, які дозволяють виявляти ділянки коду з найбільшими витратами часу та

ресурсів. Це дає можливість здійснювати цілеспрямовану оптимізацію найбільш критичних частин програми.

Visual Studio підтримує інтеграцію із системою контролю версій, що забезпечує зручне керування змінами у вихідному коді та сприяє організації процесу розробки. Також середовище підтримує розширення функціональності через додаткові компоненти та плагіни, що дозволяє адаптувати його під вимоги конкретного проєкту.

Microsoft Visual Studio забезпечує комплексний набір інструментів для ефективної розробки, тестування та оптимізації програмного забезпечення, що робить його доцільним вибором для реалізації даного проєкту.

3.2.4 Переваги та недоліки обраного стеку технологій

Використання поєднання мови C++ та бібліотеки SFML дозволяє досягти високої продуктивності та достатньої гнучкості при розробці застосунку, особливо в задачах, пов’язаних з обробкою графічних даних та великих масивів інформації.

До основних переваг такого підходу належать:

– висока швидкодія обробки даних завдяки компіляції коду в машинний рівень та мінімальним накладним витратам виконання;

– ефективне використання оперативної пам’яті за рахунок ручного керування ресурсами та відсутності надлишкових абстракцій;

– можливість реалізації оптимізованих алгоритмів з урахуванням особливостей апаратної архітектури;

– простота інтеграції графічного інтерфейсу завдяки використанню SFML, яка надає зручний API для роботи з вікнами, подіями та графічними буферами;

– кросплатформеність рішення, що дозволяє запускати застосунок на різних операційних системах без суттєвих змін у вихідному коді.

Разом із цим, існують і певні обмеження. Зокрема, бібліотека SFML не надає вбудованих засобів для складної обробки зображень та розширених графічних алгоритмів, що вимагає самостійної реалізації відповідних компонентів або використання додаткових бібліотек. Це збільшує обсяг коду та складність розробки, однак водночас забезпечує повний контроль над процесом обробки даних і дозволяє більш гнучко оптимізувати алгоритми під конкретні задачі.

Також мова C++ вимагає більш уважного підходу до керування пам’яттю та ресурсами системи. Неправильне використання динамічної пам’яті може призводити до її витоків або зниження продуктивності, що особливо критично у довготривалих або ресурсомістких застосунках. Крім того, висока гнучкість мови супроводжується більш складним процесом розробки порівняно з високорівневими мовами програмування, що збільшує вимоги до кваліфікації розробника.

Обраний стек технологій забезпечує баланс між продуктивністю та гнучкістю розробки, однак вимагає більш уважного підходу до проєктування архітектури та керування ресурсами.

3.2.5 Аналіз альтернативних рішень

Під час вибору інструментальних засобів реалізації програмного застосунку важливим етапом є аналіз альтернативних рішень, які можуть бути використані для досягнення подібних цілей. Такий аналіз дозволяє обґрунтувати доцільність обраного стеку технологій та оцінити його переваги у порівнянні з іншими підходами.

Серед альтернатив мов програмування для реалізації задач обробки зображень можна виділити як низькорівневі, так і високорівневі рішення.

Мови низького рівня, такі як C та C++, забезпечують високий рівень продуктивності та прямий доступ до апаратних ресурсів. Вони дозволяють

реалізовувати оптимізовані алгоритми обробки даних з мінімальними накладними витратами, що є критично важливим для ресурсомістких задач. Саме тому вони часто застосовуються у системному програмуванні, ігрових рушіях та графічних застосунках.

Натомість високорівневі мови програмування, такі як Python та Java, забезпечують вищий рівень абстракції та значно спрощують процес розробки. Вони мають розвинуті стандартні бібліотеки та активні екосистеми, що дозволяє швидко створювати прототипи та реалізовувати складні системи. Наприклад, Python широко використовується в задачах комп'ютерного зору завдяки бібліотекам OpenCV та NumPy. Однак такі мови зазвичай мають нижчу продуктивність у порівнянні з C++ через використання інтерпретатора або віртуальної машини.

Отже вибір мови програмування є компромісом між продуктивністю виконання та швидкістю розробки.

Окрім SFML, існує низка інших бібліотек для роботи з графікою та мультимедіа.

Однією з найбільш поширених є OpenGL – низькорівневий графічний API, який забезпечує максимальний контроль над рендерингом. Його використання дозволяє досягати високої продуктивності, однак вимагає значно більшої складності реалізації та глибокого розуміння графічного конвеєра.

Також широко використовується бібліотека SDL, яка надає базовий рівень абстракції для роботи з вікнами, введенням та графікою. SDL є більш низькорівневою порівняно з SFML і часто використовується як основа для ігрових рушіїв.

Іншим варіантом є Qt, який орієнтований переважно на створення графічних інтерфейсів користувача. Qt має потужні засоби для побудови UI, але є менш орієнтованим на високопродуктивну обробку графічних даних у реальному часі.

У порівнянні з ними, SFML займає проміжне положення, забезпечуючи баланс між простотою використання та достатнім рівнем продуктивності.

Для розробки програмного забезпечення на C++ можуть використовуватися різні середовища розробки.

Microsoft Visual Studio є одним із найпотужніших середовищ, яке забезпечує глибоку інтеграцію з компілятором MSVC, засобами відлагодження та профілювання. Воно особливо ефективне для розробки під Windows та великих проєктів.

CLion від JetBrains є кросплатформеним середовищем, яке базується на системі CMake і надає зручні інструменти рефакторингу та аналізу коду. Воно широко використовується у кросплатформеній розробці, однак може вимагати більш складної початкової конфігурації.

Visual Studio Code є більш легким редактором коду з можливістю розширення функціональності через плагіни. Він є гнучким рішенням, але потребує додаткового налаштування для повноцінної роботи як середовище розробки C++.

У порівнянні з альтернативами, Visual Studio було обрано як найбільш збалансоване рішення, що поєднує високу продуктивність, зручність використання та широкий набір вбудованих інструментів.

Аналіз альтернативних рішень показує, що обраний стек технологій забезпечує оптимальний баланс між продуктивністю, зручністю розробки та можливостями оптимізації. Альтернативні рішення можуть бути ефективними у певних сценаріях, однак вони або поступаються у швидкодії, або ускладнюють процес розробки, що робить їх менш доцільними для даної роботи.

3.3 Алгоритми обробки пікселів

3.3.1 Точкові операції

Точкові операції є найпростішим і водночас наймасовішим типом обробки зображень. Вони передбачають незалежну зміну значень пікселів, що дозволяє реалізувати їх за допомогою простих циклів.

Однак навіть у цьому випадку ефективність реалізації має суттєве значення. Основною проблемою є велика кількість ітерацій, що робить критичними навіть незначні накладні витрати.

Одним із ключових підходів є мінімізація складності внутрішнього циклу. Це означає, що всі допоміжні обчислення, які не залежать від конкретного пікселя, повинні виконуватися поза циклом. Такий підхід дозволяє зменшити загальний обсяг обчислень.

Ще одним важливим аспектом є уникнення розгалужень. Умовні оператори можуть призводити до порушення конвеєризації виконання інструкцій у процесорі, що негативно впливає на продуктивність.

Також доцільно використовувати прямий доступ до масиву пікселів без додаткових рівнів абстракції. Це дозволяє уникнути накладних витрат, пов’язаних із викликом функцій або перевірками.

У сукупності ці підходи дозволяють реалізувати обробку, яка є максимально близькою до апаратних можливостей системи.

3.3.2 Геометричні операції

Геометричні операції, зокрема обрізання, мають іншу природу, ніж точкові, оскільки змінюють структуру зображення. Це означає, що їх реалізація пов’язана з додатковими витратами пам’яті. Створення нових масивів є витратними операціями, тому вони мають виконуватися лише тоді, коли це дійсно необхідно.

Основною задачею є формування нового зображення на основі вибраної області. Для цього необхідно визначити координати області та виконати копіювання відповідних пікселів. Оптимізація цього процесу полягає у правильній організації копіювання. Найбільш ефективним є копіювання цілими рядками або їх частинами, що дозволяє зменшити кількість операцій і покращити використання кешу.

Важливо уникати надлишкових перевірок у циклах. Наприклад, перевірка меж області повинна виконуватися до початку обробки, а не під час кожної ітерації.

У результаті має бути досягнено прийнятний баланс між простотою реалізації та ефективністю виконання.

3.4 Структури даних

Вибір способу організації даних у задачах обробки зображень безпосередньо впливає як на швидкодію, так і на передбачуваність роботи алгоритмів. Оскільки зображення є щільною структурою з великою кількістю однотипних елементів, найбільш ефективним підходом є використання суцільного масиву байтів, який можна представити у вигляді формули:

$$\text{index} = y \cdot \text{width} + x, \tag{3.1}$$

де index – ідекс пікселя;

y – координата пікселя по осі y;

width – ширина зображення;

x – координата пікселя по осі x.

Такий формат зберігання забезпечує кілька важливих переваг. По-перше, дані розміщуються в пам’яті без розривів, що дозволяє процесору виконувати послідовне завантаження блоків у кеш. По-друге, адресація

кожного пікселя виконується через прості арифметичні обчислення, що зменшує накладні витрати на доступ.

Використання багаторівневих або вкладених структур, наприклад, масив масивів, у подібних задачах є менш ефективним, оскільки призводить до фрагментації пам’яті та збільшення кількості звернень до різних ділянок оперативної пам’яті. Це, у свою чергу, негативно впливає на кеш-локальність і стабільність часу виконання.

Окремо варто зазначити, що лінійне представлення даних спрощує інтеграцію з графічними API та бібліотеками, які також працюють із послідовними буферами пікселів. Це зменшує необхідність додаткових перетворень форматів і підвищує загальну ефективність системи.

3.5 Реалізація програмного забезпечення

3.5.1 Реалізація модуля роботи з растровими зображеннями

Модуль роботи з растровими зображеннями реалізовано як окремий компонент у вигляді класу ImageProcessor, який забезпечує виконання базових операцій обробки зображень.

Для зберігання піксельних даних використовується клас sf::Image бібліотеки SFML. Даний клас інкапсулює внутрішнє представлення зображення у вигляді масиву байтів у форматі RGBA та надає зручний інтерфейс для доступу до окремих пікселів.

Доступ до пікселів здійснюється за допомогою методів getPixel() та setPixel(), які приймають координати пікселя. Це дозволяє виконувати обробку зображення шляхом послідовного обходу всіх пікселів за допомогою вкладених циклів.

Модуль реалізує такі основні операції:

– перетворення зображення у відтінки сірого;

– обрізання зображення;

– зміна кольорових компонентів.

Кожна операція реалізована як окрема статична функція, що приймає вхідне зображення та повертає нове оброблене зображення. Такий підхід спрощує використання модуля та забезпечує відсутність побічних ефектів.

3.5.2 Реалізація модуля графічного інтерфейсу та обробки подій

Графічний інтерфейс реалізовано з використанням віконного підходу, який передбачає відображення зображення у графічному вікні та обробку подій користувача.

Основою є цикл обробки подій, у якому система отримує інформацію про дії користувача та відповідно реагує на них. Події включають натискання клавіш, рух миші та інші взаємодії.

Кожна подія інтерпретується та перетворюється у відповідну команду, яка передається до модуля обробки зображень. Такий підхід забезпечує чітке розділення між інтерфейсом та логікою обробки.

Відображення зображення виконується шляхом оновлення графічного буфера та його виведення у вікно програми.

3.5.3 Реалізація точкових операцій обробки пікселів

Точкові операції реалізуються шляхом послідовного обходу всіх пікселів зображення з використанням вкладених циклів.

Доступ до пікселів здійснюється через методи getPixel() та setPixel() класу sf::Image.

У процесі обробки для кожного пікселя виконуються арифметичні операції над його кольоровими компонентами. Наприклад, при зміні кольору виконується додавання зміщення до кожного каналу з подальшим обмеженням значень у діапазоні [0; 255].

Обчислення організовано таким чином, щоб мінімізувати складність операцій у внутрішньому циклі, що позитивно впливає на продуктивність.

3.5.4 Реалізація геометричних операцій

Операція обрізання реалізується шляхом створення нового зображення, яке містить лише вибрану область.

Перед виконанням обрізання здійснюється перевірка та корекція меж області. Це дозволяє уникнути виходу за межі зображення та забезпечує коректність виконання операції.

Після перевірки виконується копіювання відповідних пікселів із вихідного зображення у нове. Обробка здійснюється за допомогою вкладених циклів, що проходять по вибраній області.

Такий підхід забезпечує простоту реалізації та достатню ефективність для інтерактивного використання.

3.5.5 Інтеграція модулів та організація основного циклу

Інтеграція модулів здійснюється через центральний компонент, який координує їхню взаємодію.

Основний цикл програми забезпечує послідовне виконання обробки подій, обчислень та відображення результатів. Дані передаються між модулями через чітко визначені інтерфейси, що дозволяє уникнути зайвих залежностей.

Такий підхід забезпечує узгоджену роботу всіх компонентів системи та дозволяє легко розширювати функціональність у майбутньому.

3.6 Тестування та оцінка якості рішення

3.6.1 Методика тестування

Для оцінювання ефективності розробленого програмного застосунку було проведено серію тестувань, спрямованих на визначення продуктивності, використання пам’яті та стабільності роботи.

Тестування виконувалося на наборі растрових зображень різної роздільної здатності, що дозволило оцінити поведінку системи при різних обсягах даних. Основна увага приділялася операціям зміни кольору та обрізання, оскільки вони є базовими для реалізованого функціоналу.

В якості критеріїв оцінювання використовувалися:

– час виконання операцій обробки;

– обсяг використаної оперативної пам’яті;

– стабільність роботи при багаторазовому виконанні операцій.

Вимірювання часу виконання здійснювалося шляхом фіксації тривалості виконання відповідних функцій. Для підвищення точності результати усереднювалися на основі кількох запусків.

Такий підхід дозволяє отримати об’єктивну оцінку ефективності реалізованих алгоритмів.

3.6.2 Результати вимірювання продуктивності

Аналіз результатів тестування показав, що час виконання операцій прямо залежить від розміру зображення, що є очікуваним для алгоритмів, які виконують повний обхід масиву пікселів (табл. 3.1).

Операція зміни кольору демонструє лінійну залежність часу виконання від кількості пікселів. Це підтверджує ефективність реалізації, оскільки відсутні додаткові накладні витрати, що могли б призвести до нелінійного зростання часу.

Таблиця 3.1 – залежність швидкостей операцій від обсягу та типу

Операція	Розміри зображення	Середній час
Зміна кольору	640x640	8 мс
Зміна кольору	420x420	4 мс
Зміна кольору	200x200	<0 мс
Вирізання	630x630	2 мс
Вирізання	420x420	1 мс
Вирізання	200x200	<0 мс

Швидкість операції обрізання, залежить від розміру вибраної області . Оскільки обробляється лише частина зображення, час не залежить від загального розміру зображення.

Загалом результати свідчать про те, що реалізовані алгоритми забезпечують достатній рівень продуктивності для інтерактивного використання. Затримки при виконанні операцій залишаються в межах, що не впливають на сприйняття користувачем.

3.6.3 Використання пам’яті

Оцінка використання оперативної пам’яті проводилася під час активної роботи програмного застосунку, зокрема після виконання основних операцій обробки зображень. За результатами вимірювань встановлено, що загальний обсяг споживаної пам’яті становить приблизно 28 МБ.

Слід враховувати, що отримане значення включає не лише безпосередні витрати на зберігання зображень та робочих структур даних, але й додаткові накладні витрати, пов’язані з середовищем виконання. Зокрема, програма запускалася у режимі налагодження (debug), який передбачає використання додаткових службових механізмів, таких як

розширена інформація для відстеження виконання, перевірки пам’яті та інші інструменти діагностики.

У зв’язку з цим реальне споживання пам’яті у релізній версії програми буде нижчим, оскільки відсутні додаткові витрати, характерні для debug-режиму. Таким чином, зафіксоване значення можна розглядати як верхню оцінку використання ресурсів.

3.6.4 Оцінка портативності

Тестування програмного застосунку проводилося на інших операційних системах, що дозволило оцінити його переносимість

Для коректної роботи застосунку, було необхідно було скомпілювати його відповідним до системи компілятором, а також завантажити бібліотеку SFML, що є сумісною із обраною системою.

Результати показали, що після застосування відповідних змін, програма на linux функціонує коректно без необхідності внесення змін у код. Це підтверджує ефективність використання кросплатформених засобів.

Водночас було встановлено, що продуктивність може відрізнятися залежно від апаратного забезпечення. Основний вплив мають характеристики процесора та підсистеми пам’яті.

Незважаючи на це, у обох випадках забезпечується стабільна робота та достатній рівень швидкодії.

3.6.5 Аналіз стабільності та обробки помилок

Стабільність роботи застосунку перевірялася під час виконання основних операцій обробки зображень.

Особлива увага приділялася перевірці коректності параметрів операції обрізання. У випадку, якщо задана область виходить за межі зображення або має некоректні розміри, користувачу виводиться повідомлення про помилку.

Також реалізовано механізм візуального повідомлення про успішне збереження зображення.

Такий підхід забезпечує передбачувану поведінку системи та запобігає виникненню критичних помилок під час роботи.

3.7 Інструкція користувача

Для початку роботи із програмним застосунком необхідно запустити виконуваний файл програми. Після запуску відкривається головне вікно, у якому відображається інтерфейс користувача (рис. 3.1).

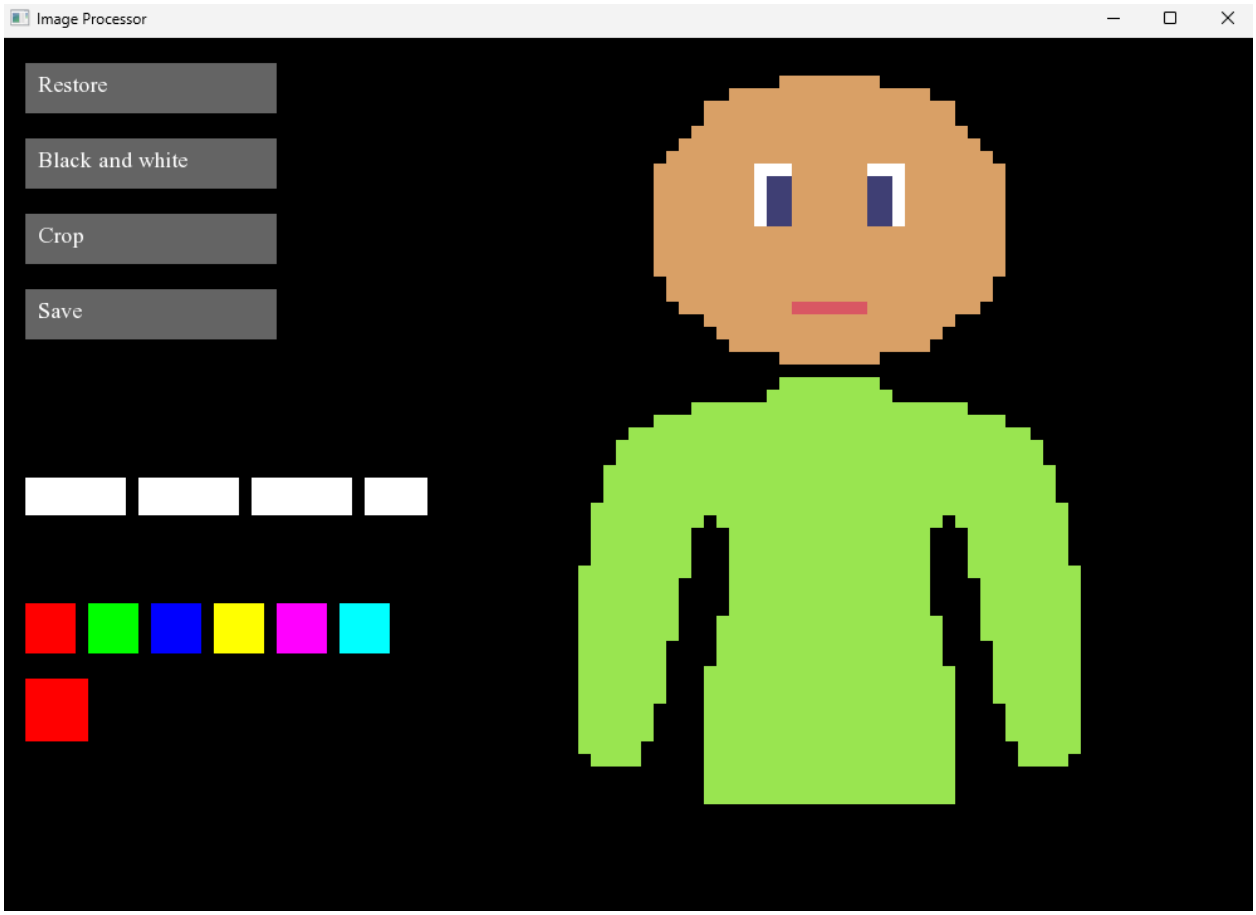

Рисунок 3.1 – Вікно програми після запуску

Для зміни кольору зображення необхідно двічі натиснути на бажаний варіант із запропонованих у кольоровій палітрі. Результат застосовується миттєво та відображається на екрані (рис. 3.2).

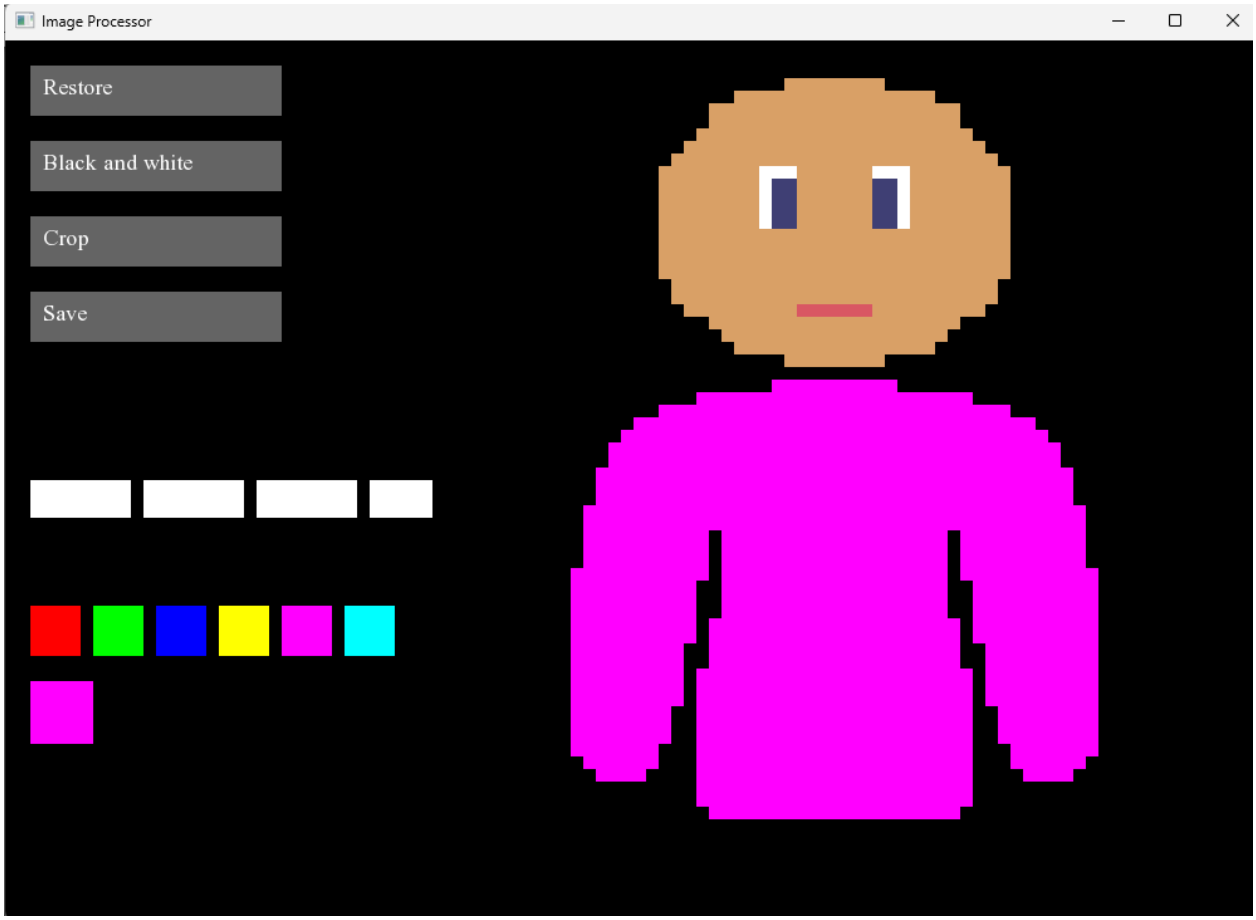

Рисунок 3.2 – Вибір рожевого коліру

Також є можливість змішати кольори, натиснувши двічі по різним кольорам у палітрі (3.3).

Обрізання зображення виконується шляхом задання області, що складається з координат x та y, і відстані по висоті та ширині та натискання кнопки «Crop». Після натискання на кнопку «Crop» формується нове зображення, що містить лише вибраний фрагмент (рис. 3.4).

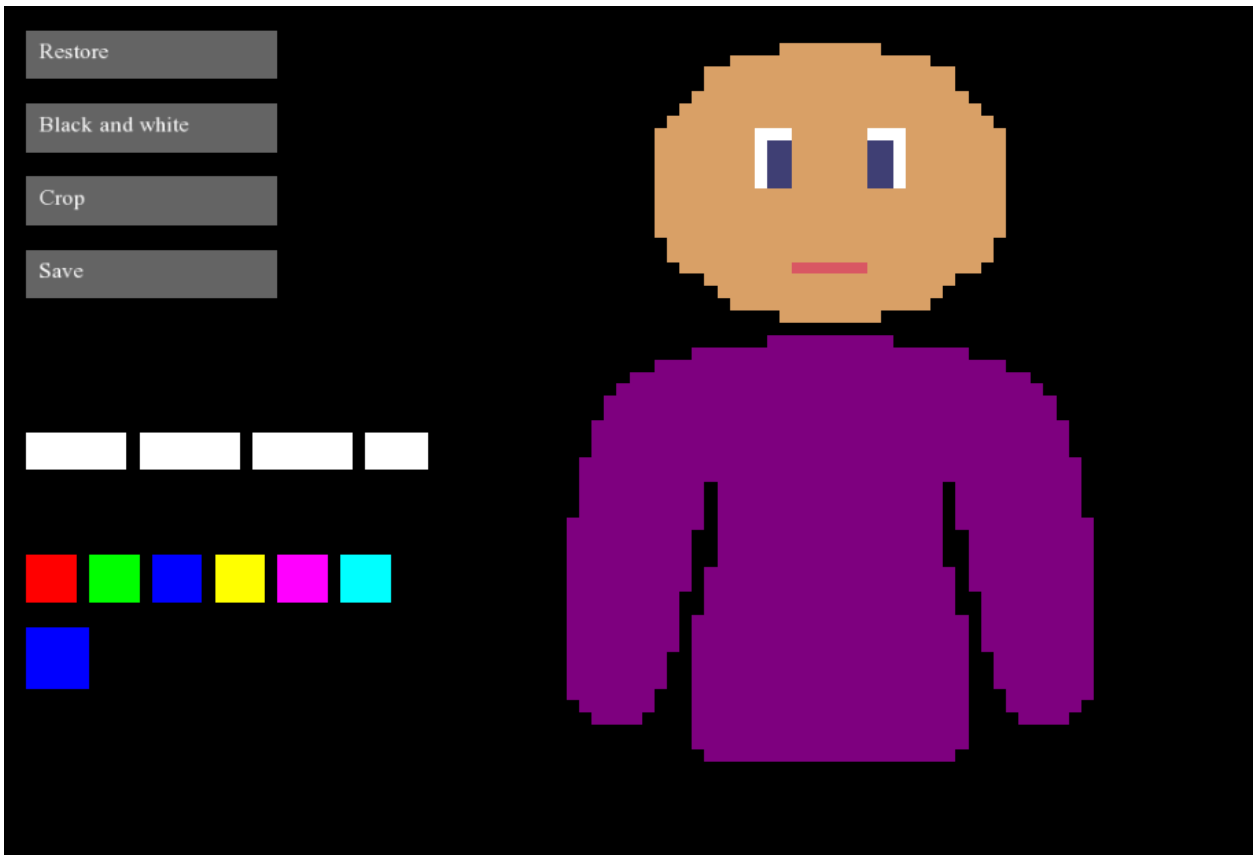

Рисунок 3.3 – Змішування червоного та синього кольорів

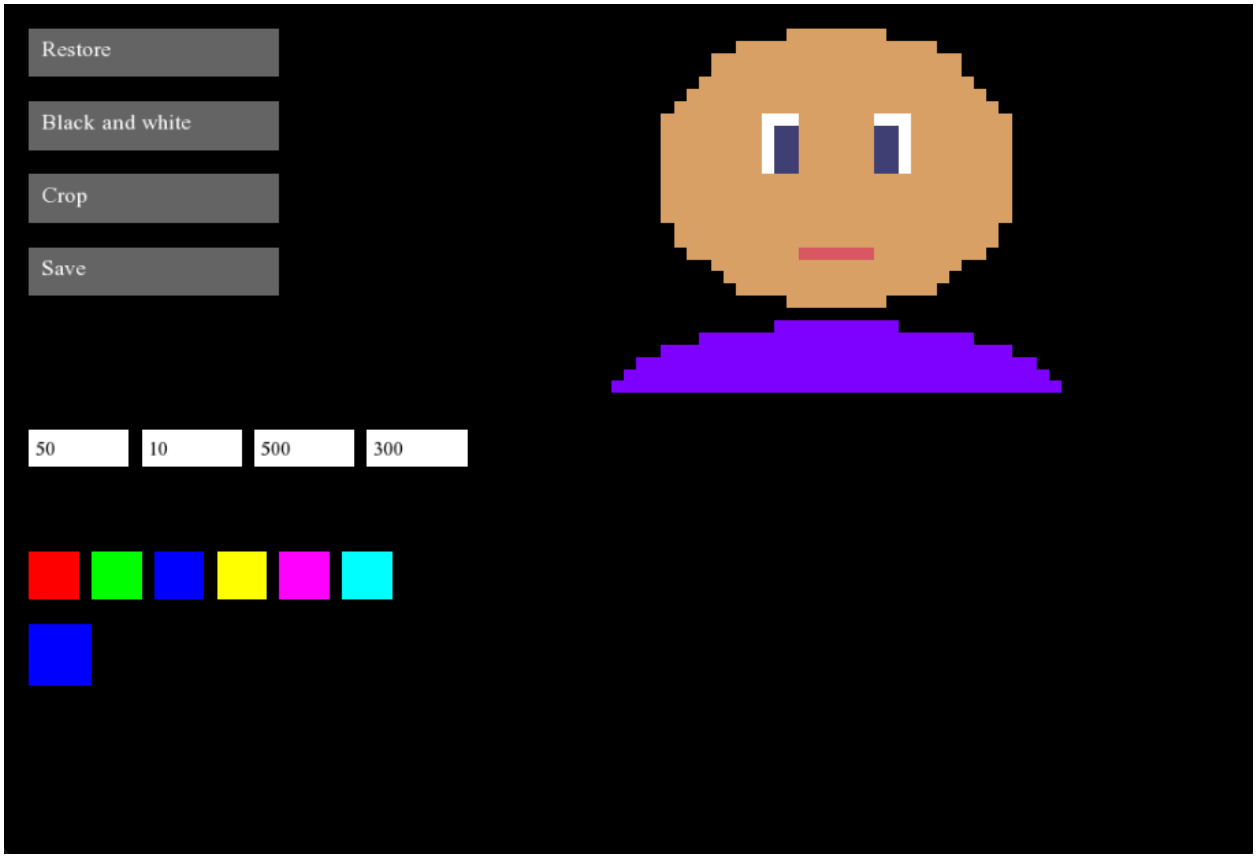

Рисунок 3.4 – Результат обрізання зображення за введеними координатами

У випадку некоректно введених координат, користувач отримає повідомлення про помилку яке автоматично зникає із часом (рис. 3.5)

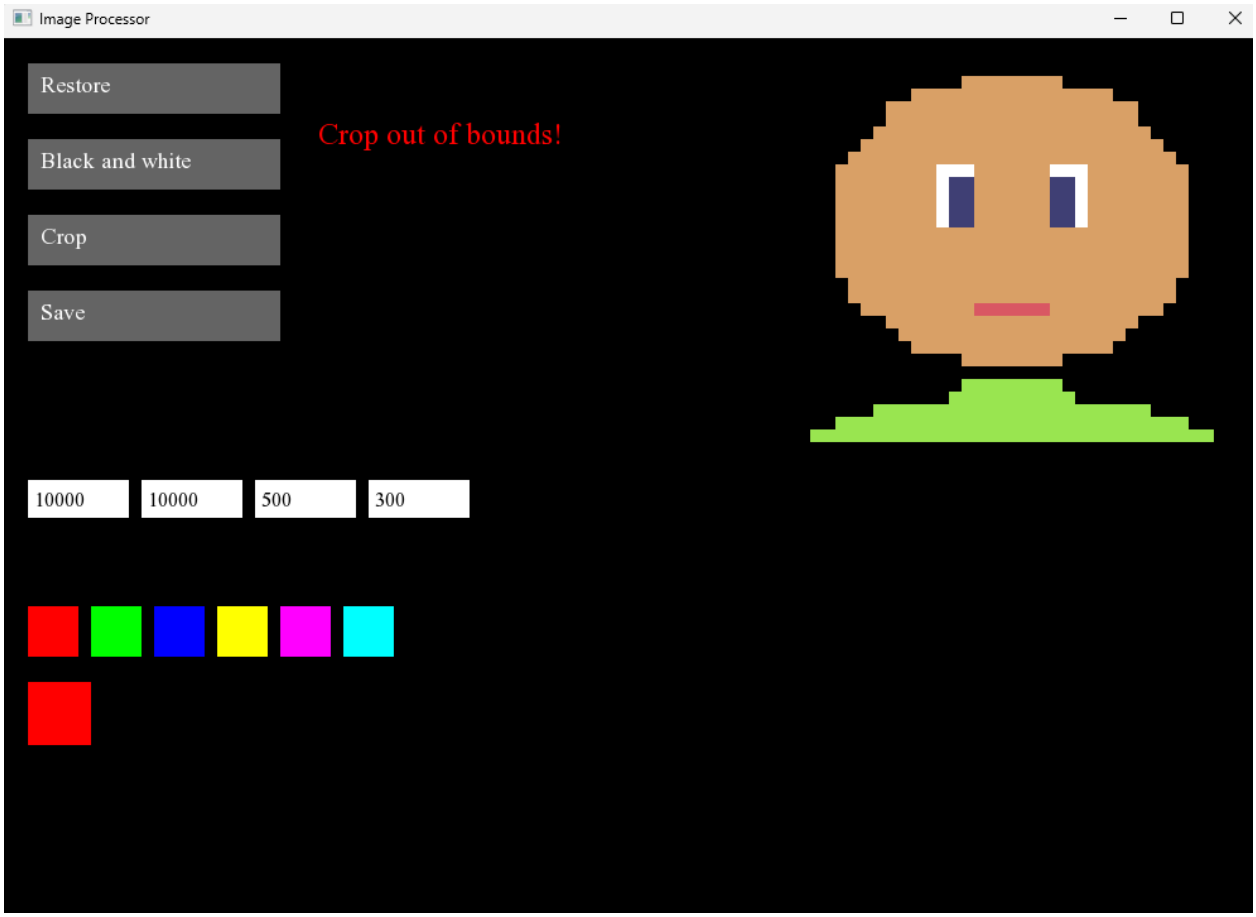

Рисунок 3.5 – Повідомлення про помилку при некоректних даних

Після обрізання зображення, користувач все ще може змінювати колір або здійснювати повторні обрізання.

Кнопка «Black and white» дозволяє перевести зображення у чорно-білий фомат (рис. 3.6).

Для збереження зображення необхідно натиснути на кнопку «Save». Після натискання, при успішному збережені користувач отримає відповідне повідомлення на екрані як автоматично зникає із часом (рис. 3.7).

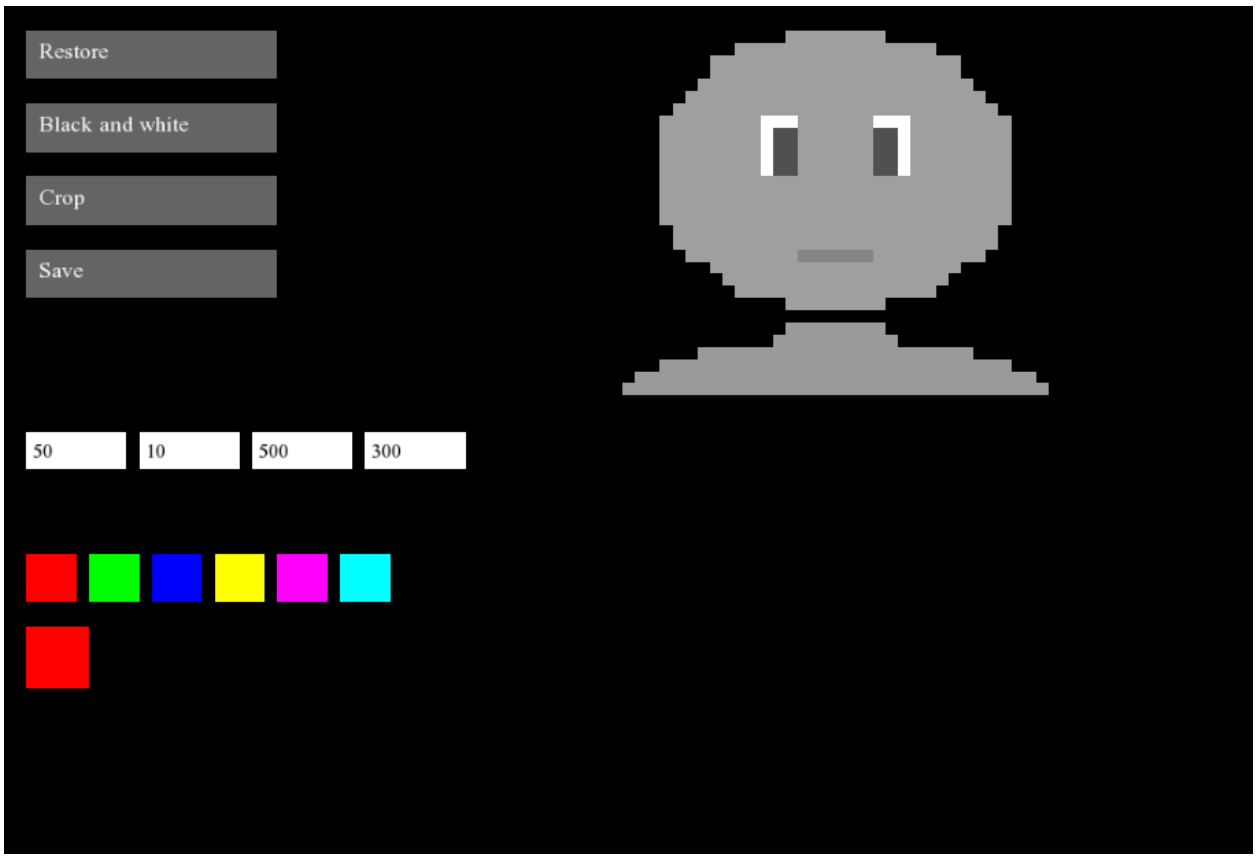

Рисунок 3.6 – Обрізане зображенняя переведено у чорно-білий формат

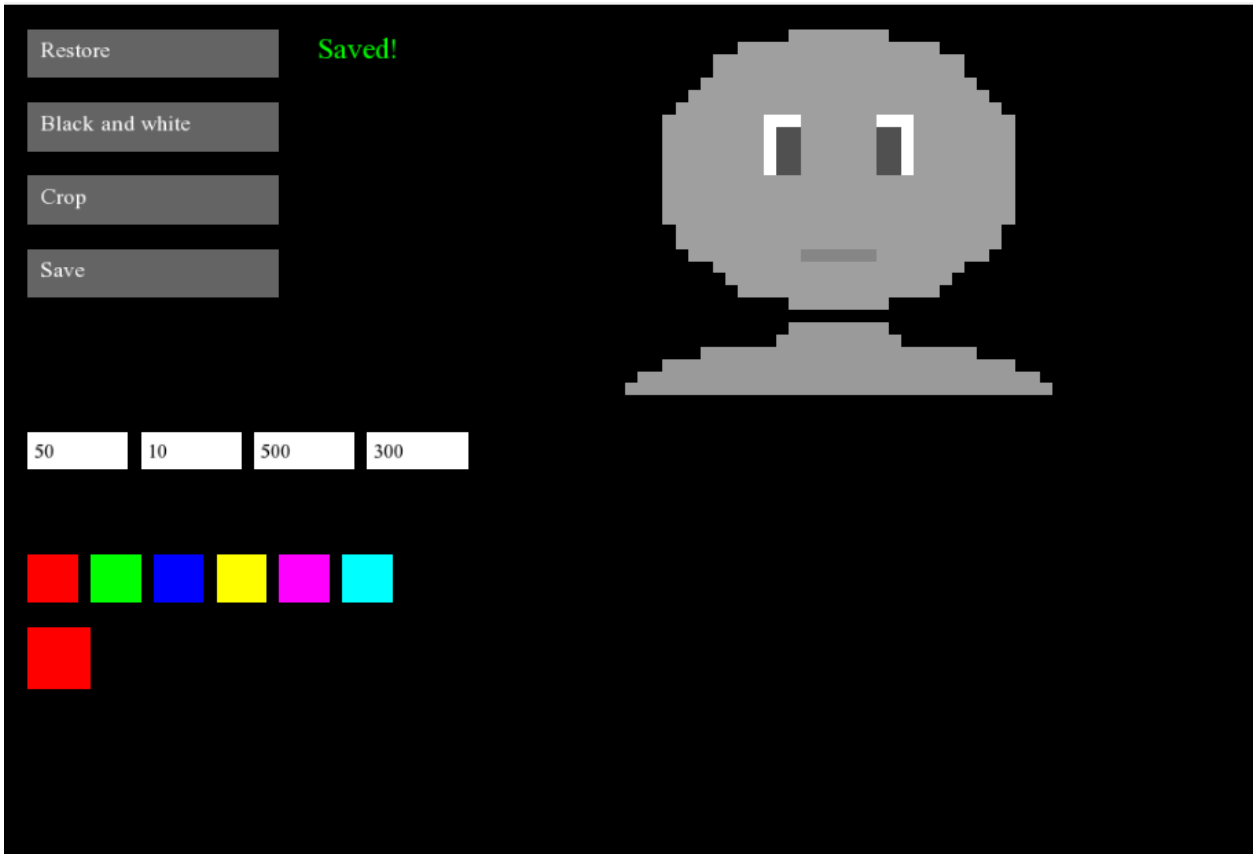

Рисунок 3.7 – Повідомлення про успішне збереження зображення

Для відновлення зображення у початковий стан необхідно натиснути на кнопку «Restore». Після натискання, зображення повертає свої оригінальний колір та розміри.

Зображення зберігається у папці програми із назвою «Output_X», де X виступає порядковим номером збереженого файлу (рис. 3.8).

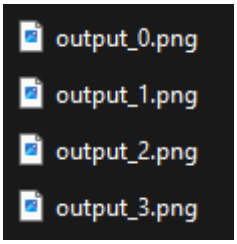

Рис. 3.8 – Автоматично прономеровані збережені зображення

3.8 Перспективи подальшого розвитку та удосконалення

Розроблений програмний застосунок може бути розширений та вдосконалений у кількох напрямках, що дозволить підвищити його функціональність та продуктивність.

Одним із перспективних напрямків є використання графічних процесорів для виконання обчислень. Перенесення обробки на GPU дозволяє значно підвищити продуктивність при роботі з великими зображеннями.

Також можливе розширення функціональності за рахунок додавання нових алгоритмів обробки, таких як фільтрація, масштабування або корекція яскравості.

Додатково може бути реалізована інтеграція з іншими бібліотеками, що надають розширені можливості роботи з зображеннями.

Іншим напрямком може стати переведення прогманого застосунку у функцію для гри, котра процедурно створюватиме нові зображення без необхідності їх зберігання.

ВИСНОВКИ

У рамках кваліфікаційної роботи було розроблено і реалізовано програмний застосунок для обробки зображень, що реалізує базові операції над пікселями, зокрема зміну кольору, перетворення у відтінки сірого та обрізання. Архітектура застосунку побудована на модульному принципі з чітким розподілом відповідальностей між компонентами, що забезпечило гнучкість, спрощення супроводу та можливість подальшого розширення функціональності без значних змін у структурі системи.

Було обґрунтовано вибір мови програмування C++ та використання бібліотеки SFML, які дозволили досягти високої продуктивності при роботі з піксельними даними та реалізувати ефективну взаємодію з користувачем через графічний інтерфейс. Особливу увагу приділено реалізації алгоритмів обробки зображень із урахуванням мінімізації накладних витрат у внутрішніх циклах, що позитивно вплинуло на швидкодію застосунку.

Проведене тестування підтвердило ефективність реалізованих рішень. Оцінка використання пам’яті показала раціональне використання ресурсів, а перевірка портативності підтвердила можливість коректної роботи застосунку на різних платформах без необхідності суттєвої модифікації коду.

Результати роботи апробовано у вигляді 3 тез доповідей під час Міжнародного молодіжного форуму «РАДІОЕЛЕКТРОНІКА І МОЛОДЬ У XXI СТОЛІТТІ» [14], II Всеукраїнської науково-технічної конференції «Виклики та рішення в програмній інженерії» [2], XI Міжнародної науково-технічної конференції "Інформаційно-комп'ютерні технології" [10].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

2. Zaika, D. V., & Kobylin, I. O. (2025) Code optimization for easy portability to different operating systems, pp. 453-456.

3. Alexandrescu, A. (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley Professional.

4. Meyers, S. (1995). *More Effective C++: 35 New Ways to Improve Your Programs and Designs (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional.

5. Tan, J., Jiao, S., Chabbi, M., & Liu, X. (2020, June). What every scientific programmer should know about compiler optimizations?. In *Proceedings of the 34th ACM International Conference on Supercomputing* (pp. 1-12).

6. Тіхвиський, О. В., & Каратанов, О. В. (2025). Огляд сучасних методів оптимізації продуктивності комп'ютерних програм. *Open Information and Computer Integrated Technologies*, (104), 133–144.

7. Drepper, U. (2007). What every programmer should know about memory. *Red Hat, Inc*, *11*(2007), 2007.

8. Rayward-Smith, V. J., Cormen, T. H., Leiserson, C. E., & Rivest, R. L. (1991). Introduction to Algorithms. *The Journal of the Operational Research Society*, *42*(9), 816.

9. Skiena, S. S. (2020). Introduction to algorithm design. In *The Algorithm Design Manual* (pp. 3-30). Cham: Springer International Publishing.

10. Zaika, D. V., & Kobylin, I. O. (2025) Optimization of code algorithms in video games. Current trends and efficiency issues, pp. 56-58.

11. *Chapter 13. Implementing the mental images Phenomena Renderer on the GPU*. (б. д.). NVIDIA Developer. URL: https://developer.nvidia.com/gpugems/gpugems2/part-ii-shading-lighting-and-shadows/chapter-13-implementing-mental-images (дата звернення 14.04.2026).

12. Bilocerkivska, V. A., & Kobylin, I. O. (2024) on the processes of computer forgery and generation of media content.

13. Šťava, O., Beneš, B., Měch, R., Aliaga, D. G., & Krištof, P. (2010). Inverse Procedural Modeling by Automatic Generation of L-systems. *Computer Graphics Forum*, *29*(2), 665–674.

14. Zaika, D. V., & Kobylin, I. O. (2026) Disk space optimization in video games. Current trends and efficiency issues.

15. Tvoroshenko, I., & Almakaieva, A. (2020). Application of procedural generation of game content using software algorithms.

16. Sutter, H. (2025). *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Pearson Education, Limited.

17. Gorokhovatskyi V., Chmutov Y., Tvoroshenko I., and Kobylin O. (2025) Reducing computational costs by compressing the structural description in image classification methods, Advanced Information Systems, vol. 9, no. 1, pp. 5–12.

18. Gorokhovatskyi V., Tvoroshenko I. (2024) An effective method for transforming an image description into a compact vector for classification. Information Technology and Implementation (Satellite): Conference Proceedings, November 21, 2024, Kyiv, Ukraine / Ministry of Education and Science of Ukraine, Taras Shevchenko National University of Kyiv and [etc]; Vitaliy Snytyuk (Editor). – Kyiv: Publishing House «Caravela», pp. 25-28.

19. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Al-Dhaifallah M. (2022) Classification of Images Based on a System of Hierarchical Features, Computers, Materials & Continua, 72(1), pp. 1785-1797.

20. Buhayevskyy, M. S, & Yakovleva, O. V. (2020) Improving the quality of project management based on the agile project management methodology

21. Pratt, W. K. (2007). Digital Image Processing, 4th Edition. *Journal of Electronic Imaging*, *16*(2), 029901.

22. Кобилін, О. А., & Творошенко, І. С. (2021) Методи цифрової обробки зображень: навч. посібник.

23. Luciva, D. V., & Lubchenko, V. A. (2020) Creating averaged face images using c++ and opencv, dlib libraries.

24. Karakonstantyn D., Tvoroshenko I. (2024) About the issue of optimization the performance of the server part of the information system, Abstracts of XII International Scientific and Practical Conference «Prospective directions of modern science and education in the world», (November 19 – 22, 2024). Rotterdam, Netherlands, pp. 364-366.

25. Daradkeh Y.I., Gorokhovatskyi V., Tvoroshenko I., and Zeghid M. (2022) Tools for Fast Metric Data Search in Structural Methods for Image Classification, IEEE Access, 10, pp. 124738-124746.

26. Grygoryeva, D. O. ., & Kobylin, I. O. (2024) classification processes of images of external parts of a car.

27. McKenney, P. E. (2015). *Is Parallel Programming Hard*. Blurb.

28. Reinders, J. (2007). *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly Media, Incorporated.

29. S., D., & Knuth, D. E. (1971). The Art of Computer Programming--Errata et Addenda. *Mathematics of Computation*, *25*(116), 937.

30. Кобилін, І. О., & Ніколайчук, А. І. (2024). Monitoring and diagnosing faults in online mode using time series data. Системи обробки інформації, (3 (178)), 27-32.