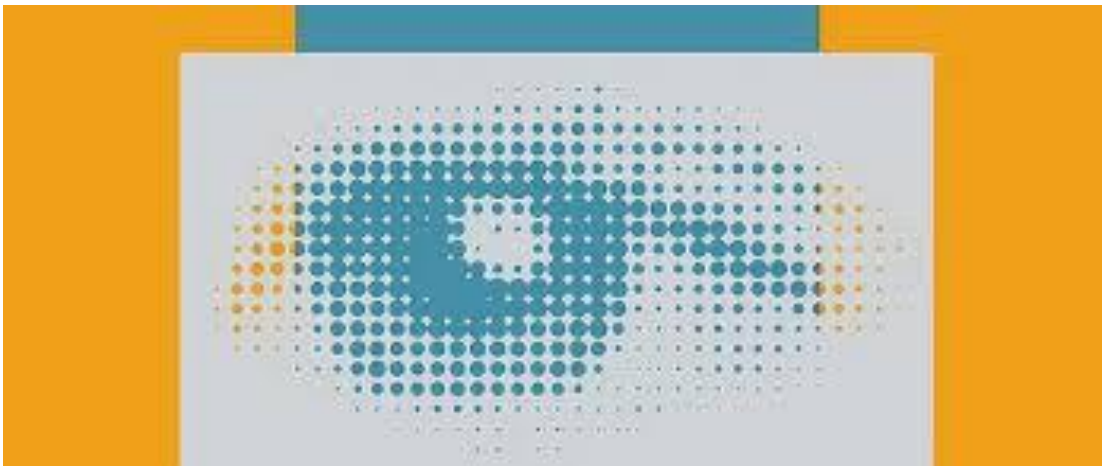
МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЧЕРНІГІВСЬКА ПОЛІТЕХНІКА»

Розпізнавання образів та обробка зображень

МЕТОДИЧНІ ВКАЗІВКИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ
ДЛЯ СТУДЕНТІВ СПЕЦІАЛЬНОСТІ 121 «ІНЖЕНЕРІЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ»



Обговорено та рекомендовано на
засіданні кафедри електроніки,
автоматики, робототехніки
та мехатроніки
Протокол № 2 від 30.09.2021 р.

ЧЕРНІГІВ – 2021

Розпізнавання образів та обробка зображень. Методичні вказівки до виконання лабораторних робіт для студентів спеціальності 121 «Інженерія програмного забезпечення». – Чернігів: НУ «Чернігівська політехніка», 2021. – 146 с.

Укладач: Войтенко Володимир Павлович, канд. техн. наук, доц.

Відповідальний за випуск – Денисов Юрій Олександрович, докт. техн. наук, проф., завідувач кафедри електроніки, автоматики, робототехніки та мехатроніки

Рецензент: Денисов Юрій Олександрович, докт. техн. наук, проф., завідувач кафедри електроніки, автоматики, робототехніки та мехатроніки національного університету «Чернігівська політехніка»

Зміст

Перелік умовних скорочень

ПЗ – програмне забезпечення.
API – Application Programming Interface (інтерфейс прикладного програмування).
BMP – Bitmap Picture (формат зберігання растрових зображень).
DNG – Digital Negative Specification (відкритий формат для RAW файлів зображень).
GIF – Graphics Interchange Format (формат для обміну зображеннями).
IDE – Integrated Development Environment (інтегроване середовище розробки.
IPT – Image Processing Toolbox (інструментарій MATLAB для обробки зображень).
JPEG – Joint Photographic Experts Group (назва розробника та один з популярних графічних форматів).
MIME – Multipurpose Internet Mail Extensions (багатоцільові розширення для інтернет–пошти).
PNG – Portable Network Graphics (растровий формат зберігання графічної інформації).
ROI – Region Of Interest (регіон інтересу).
TIFF – Tagged Image File Format (формат зберігання растрових графічних зображень).

Вступ

Доповнена реальність, автономне водіння, відеоспостереження. Численні додатки використовують технології комп'ютерного зору і аналізу зображень, допомагаючи розуміти світ через почуття зору [1].

Успішне засвоєння дисципліни «Розпізнавання образів та обробка зображень» в циклі професійної підготовки бакалаврів спеціальності 121 «Інженерія програмного забезпечення» може стати підґрунтям для подальшої успішної діяльності у таких напрямах суспільного інтересу, як національна, корпоративна та особиста безпека, медицина, інтертейнмент тощо. Суто теоретичне вивчення дисципліни не в змозі дати розуміння складних алгоритмів, застосовуваних при розпізнаванні образів, тому навчальним планом передбачені лабораторні заняття, під час яких студент може ознайомитися з особливостями методів, що використовуються в системах розпізнавання образів, та набути певних практичних навичок із їхнього застосування.

Поява потужних і доступних обчислювальних пристроїв і візуальних датчиків спростила створення складних додатків для обробки зображень. Доступно безліч програмних інструментів і бібліотек, які керують зображеннями і відео, але для всіх, хто хоче розробляти програми на основі інтелектуального бачення, бібліотека з відкритим вихідним кодом OpenCV (Open Source Computer Vision [2]) є одним з кращих для використання інструментів. Дані методичні вказівки призначені для самостійної підготовки до виконання лабораторних робіт і присвячені вивченню та практичному дослідженню характерних особливостей бібліотеки OpenCV, яка містить більше 500 оптимізованих алгоритмів для аналізу зображень і відео. З часу своєї появи в 1999 році спільнота дослідників і розробників комп'ютерного зору прийняла OpenCV в якості основного інструменту розробки.

Також завдання до лабораторних робіт можна виконати, використовуючи й інші програмні середовища, наприклад, [3].

Підготовка до виконання завдань потребує обов’язкової самостійної роботи з рекомендованою літературою та лекційним матеріалом. Контрольні запитання дозволяють оцінити ступінь опанування теоретичного матеріалу і готовність до виконання практичних завдань, які потрібно вирішити в ході роботи, користуючись наданими рекомендаціями.

Результати робіт оформлюються у вигляді звітів на аркушах формату *A4*. Звіт повинен містити таку інформацію:

1) короткі відомості про об’єкт вивчення (*15…20* рядків), в тому числі, – зміст завдання, застосовувані фреймворки та налаштування тощо;

2) відповіді на контрольні запитання;

3) коментований текст програми (скрипта або скетча);

4) висновки щодо отриманих результатів.

Пункти 1 та 2 оформлюються і показуються викладачеві на початку заняття, що є умовою допуску до виконання роботи в навчальній лабора-

торії. Весь звіт надається безпосередньо на захист роботи.

1 Лабораторна робота №1. Початок роботи з бібліотекою OpenCV

Мета роботи: встановити бібліотеку OpenCV та практично дослідити процес створення простих додатків, які можуть завантажувати, відображати та зберігати зображеная; ознайомитися із структурами даних cv::Mat; навчитися визначати регіони інтересу.

1.1 Вибір, встановлення та налаштування середовища розробки C++

Для того, щоб запустити приклади, представлені в цих рекомендаціях, і експериментувати з ними, потрібно гарне інтегроване середовище розробки (Integrated Development Environment, IDE) C++. Найбільш популярними рішеннями є Microsoft Visual Studio і Qt. Але часи змінюються, конкуренція на ринку загострюється, а тому кожен розробник додатків для обробки зображень мусить цікавитися та перевіряти можливості й інших мов програмування та пакетів програм. Наприклад, є позитивні результати експериментів у середовищі розробки SublimeText 3 для Python та ін.

У даних методичних вказівках і в навчальній лабораторії ми використовуватимемо органічне для операційної системи Windows середовище Visual Studio. Для його розгортання на власному комп'ютері спочатку треба завантажити інсталятор з сайту Microsoft. На час піідготовки даних методичних вказівок ми скористалися сторінкою Visual Studio 2019, Версія 16.8 [4]. Для навчальних цілей обмежимося варіантом інтегрованого середовища розробки Community, яке безкоштовне для студентів і окремих користувачів. Після завантаження інсталера, його треба запустити, для початку обмежившися наступним вибором:

Desktop development with C++Build modern C++ apps for Windows using tools of your choice, including MSVC, Clang, CMake, or MSBuild.

За бажанням попутно можна додати робочу мову IDE, обравши її на вкладці Language packs. Потрібний обсяг завантаження становить близько 2,06 Гб, а ємність сховища для зберігання розгорнутого середовища складає приблизно 7,76 Гб. Натискання на кнопку Install розпочне завантаження та розгортання IDE, що займе певний час.

Після завантаження та встановлення запускаємо Visual Studio та створюємо новий консольний застосунок на C++:

Create a new project/Console App/C++

Усі експериментальні проєкти в навчальній лабораторії слід розташовувати в каталозі, де, по-перше, вони не будуть заважати іншим, а по-друге, – не знищуватимуться під час регламентного обслуговування:

D:\Students\4_kypc\PI-2x1

Вибір

Place solution and project in the same directory

робить результати роботи протягом семестру більш компактними.

Переконайтеся у працездатності проєкту (CTRL+F5).

1.2 Загальна характеристика та налаштування бібліотеки OpenCV

В першій лабораторній роботі розглянемо основні елементи OpenCV і покажемо, як виконати найбільш фундаментальні завдання обробки зображень: читання, відображення та збереження зображень. Однак до початку роботи з OpenCV потрібно встановити бібліотеку. Всі програми комп'ютерного зору включатимуть обробку зображень. OpenCV пропонує потужну структуру даних для обробки зображень і матриць з корисними атрибутами і методами. Бібліотека також включає в себе вдосконалену модель управління пам'яттю, що значно полегшує розробку додатків. Задача лабораторної роботи – навчитися використовувати важливу структуру даних cv::Mat.

1.2.1 Завантаження та налаштування бібліотеки OpenCV

OpenCV – це бібліотека з відкритим кодом для розробки додатків комп'ютерного зору, які можуть працювати на декількох платформах, таких як Windows, Linux, Mac, Android та iOS. Бібліотеку можна використовувати як в академічних, так і в комерційних цілях під ліцензією BSD, що дозволяє вільно застовувати, поширювати і адаптувати програмне забезпечення (ПЗ). Розглянемо, як встановити бібліотеку на власному комп'ютері.

На офіційному веб-сайті OpenCV за адресою http://opencv.org/ [2] можна знайти останню версію бібліотеки, онлайнову документацію, що описує інтерфейс прикладного програмування (Application Programming Interface, API), та багато інших корисних ресурсів щодо OpenCV.

На веб-сайті OpenCV знайдіть останні доступні завантаження та виберіть потрібну платформу (Windows, Linux, Mac або iOS). Після завантаження пакету OpenCV [5] запустіть авторозархівуємий WinZip-файл і виберіть потрібне місце для встановлення бібліотеки. Створений каталог `opencv` доцільно перейменувати таким чином, щоб показати, яку версію ви використовуєте (наприклад, в лабораторії кінцевий каталог– це `D:\opencv-4.5.1`). Даний каталог буде містити набір файлів і каталогів, які складають бібліотеку. Зокрема, ви знайдете каталог `sources`, який містить всі вхідні файли (оскільки це – бібліотека з відкритим кодом).

Щоб завершити встановлення бібліотеки та мати її готовою до використання, потрібно зробити важливий крок: згенерувати двійкові файли бібліотеки для вибраного середовища. Це саме та точка, де треба прийняти рішення про цільову платформу, яку бажано використовувати для створення власних програм OpenCV, а саме: тип операційної системи та компілятор.

IDE, яке ви будете використовувати у розробці проектів, також допоможе зробити цей вибір. У випадку навчальної лабораторії потрібно зайти до `Панель керування\Система\Додаткові налаштування`

системи\Змінні оточення, вибрати системну змінну Path, редагувати її та створити шлях

D:\opencv-4.5.1\build\x64\vc14\bin

Далі повернімося до раніше створеного проєкту Visual Studio (підрозділ 1.1) та зробимо декілька налаштувань опцій проєкту, тобто скористаємося меню Проєкт (Project) /Властивості:Ім'я проєкту.

1. У підрозділі C/C++/Загальне (General), вкажемо пункт Додаткові каталоги файлів, що включаються (Addition include directories) з раніше розпакованого OpenCV:

D:\opencv-4.5.1\build\include

2. Далі вказуємо шлях до додаткових бібліотек компонувальника у пункті опцій проєкту Компонувальник (Linker) / Загальне (General)/ Додаткові каталоги бібліотек (Additional library directories) шлях:

D:\opencv-4.5.1\build\x64\vc14\lib

3. Переходимо до підрозділу опцій проєкту Компонувальник (Linker) / Введення (Input) та вказуємо для пункту Додаткові залежності (Additional dependencies) ім'я бібліотеки: opencv_world451d.lib.

4. У розділі Каталоги VC++ (VC++ Directories) опцій проєкту (вікно Властивості конфігурації (Configuration Properties)) у групі Загальне (General) задаємо додаткові Каталоги, що включаються (Include directories):

D:\opencv-4.5.1\build\x64\vc14\bin

Після вибору типу проєкту Debug та платформи x64 треба перезавантажити Visual Studio.

Для перевірки корректності підключення OpenCV необхідно завантажити тестове зображення до відповідного каталогу та відкомпілювати наступний код:

```
#include <opencv2/opencv.hpp>
#include <iostream>

using namespace cv;
using namespace std;

int main()
{
    Mat img = imread("D:\\opencv-4.5.1\\test_image.png");
    namedWindow("image", WINDOW_NORMAL);
    imshow("image", img);
    waitKey(0);
    return 0;
}
```

В результаті виконання програми на екрані відеомонітору маємо побачити чорне вікно консолі, а також – тестове зображення.

Зверніть увагу, що пакет бібліотеки постачається з попередньо скомпільованими *двійковими файлами*, які можна використовувати безпосередньо, якщо вони відповідають вашій ситуації (перевірте каталог build, розташований поряд з каталогом sources). Якщо один із попередньо скомпільованих бінарних файлів задовольняє вашим вимогам, все готове до роботи. Однак є одне важливе зауваження. Починаючи з версії 3, OpenCV

було розділено на дві основні компоненти. Перша з них є головним сховищем джерел OpenCV, що містить зрілі алгоритми. Це те, яке ви завантажили. Також існує окреме сховище внесків, яке містить алгоритм комп'ютерного зору, доданий учасниками OpenCV. Якщо ви плануєте використовувати лише основні функції OpenCV, пакет `contrib` не потрібен. Але, якщо ви захочете працювати з найсучаснішими алгоритмами, цей додатковий модуль може знадобитися. В інших лабораторних роботах показано, як використовувати декілька таких просунутих алгоритмів. Тому нам потрібні модулі внесків, і слід перейти на https://github.com/opencv/opencv_contrib та завантажити додаткові модулі OpenCV (завантажте ZIP–файл). Додаткові модулі слід знайти за адресою https://opencv_contrib–master/modules і розпакувати до будь–якого каталогу. Для простоти можна перейменувати цей каталог як `contrib` і скопіювати його безпосередньо в каталог `sources` основного пакета. Можна також вибрати окремі додаткові модулі і зберегти лише їх, однак, напевно, буде легше просто зберегти все.

Тепер можна продовжити встановлення. Для побудови бінарних файлів OpenCV рекомендується використовувати інструмент **CMake**, доступний за адресою http://cmake.org**.** CMake – це ще одне програмне забезпечення з відкритим вихідним кодом, призначене для керування процесом компіляції програмної системи з використанням незалежних від платформи конфігураційних файлів. Воно генерує необхідний `makefile` або файли `solution`, необхідні для компіляції програмної бібліотеки у вашому середовищі. Тому потрібно завантажити та встановити CMake. Окремо буде розглянутий додатковий пакет ПЗ **Visualization Toolkit** (**VTK**), який можна встановити перед компіляцією бібліотеки.

Можна запустити `cmake` у командному рядку, проте легше використовувати `CMake` з його графічним інтерфейсом (**cmake–gui**). В останньому випадку все, що потрібно зробити, це вказати папку, що містить джерело бібліотеки OpenCV і ту, яка буде містити двійкові файли. Тепер натисніть кнопку **Configure** і виберіть раніше обраний компілятор (рисунок 1.1).

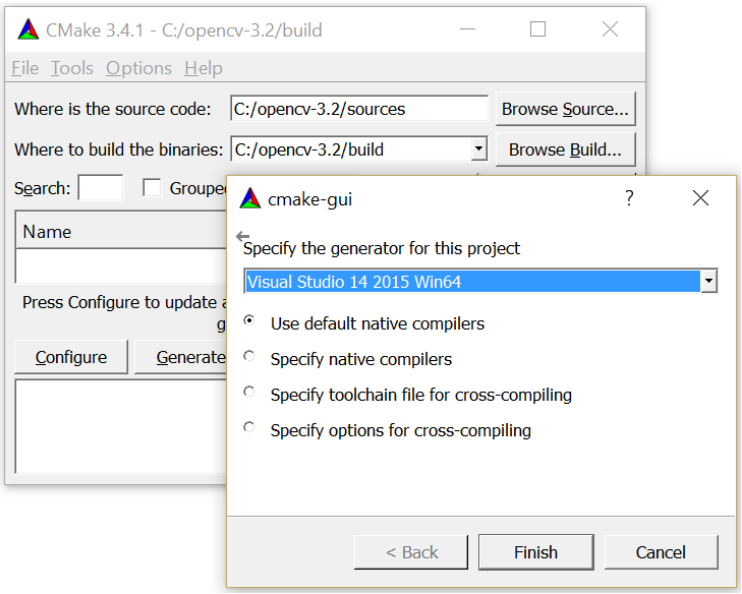


Рисунок 1.1 – Вибір раніше обраного компілятора

Після завершення цієї початкової конфігурації CMake надасть декілька параметрів конфігурації. Треба вирішити, наприклад, чи хочете ви мати встановлену документацію, або бажаєте встановити додаткові бібліотеки. Якщо ви не знаєте, що робите, можливо, краще залишити параметри за замовчуванням, як вони є. Однак, оскільки ми хочемо включити додаткові модулі, маємо вказати каталог, де їх можна знайти (рисунок 1.2).

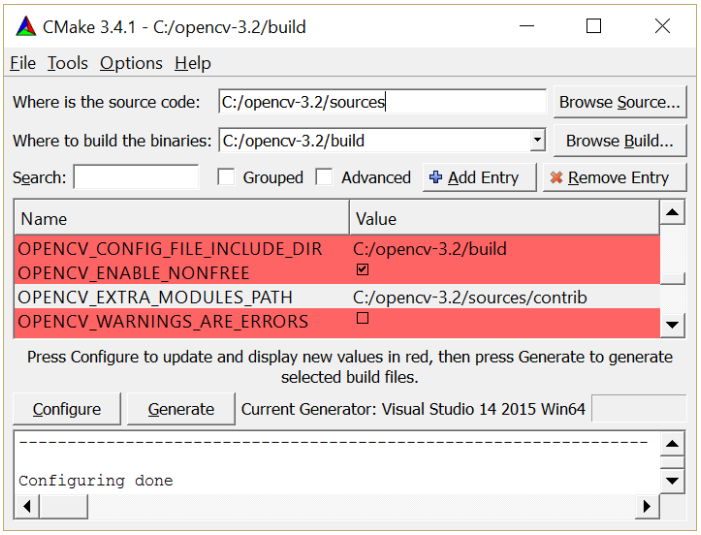


Рисунок 1.2 – Включення додаткових модулів

Після того, як шлях до додаткового модуля буде вказано, натисніть кнопку **Configure** знову. Тепер ви готові генерувати файли проекту, натискаючи кнопку **Generate**. Ці файли дозволять скомпілювати бібліотеку. Це останній крок процесу встановлення, який зробить бібліотеку готовою до використання у вашому середовищі розробки. Наприклад, якщо обрано MS Visual Studio, то все, що потрібно зробити, це відкрити файл рішення (`solution`) верхнього рівня, який створив CMake (файл `OpenCV.sln`). Потім виберіть **INSTALL** project (у розділі **CMakeTargets**) і введіть ко-

манду **Build** (скористайтеся правою кнопкою миші).

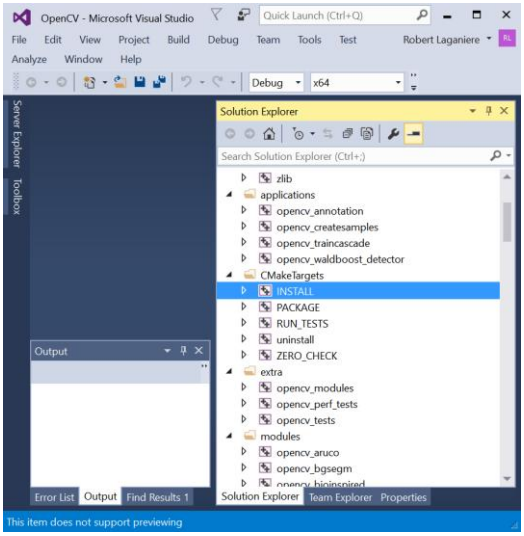


Рисунок 1.3 – Інсталяція проекту

Щоб отримати збірки **Release** і **Debug**, доведеться повторювати процес компіляції двічі, по одному для кожної конфігурації. Якщо все правильно, буде створений каталог `install` (у `build`). Цей каталог міститиме всі двійкові (`binary`) файли бібліотеки OpenCV, які будуть зв'язані з вашим додатком, а також динамічні бібліотечні файли, які виконуючі файли мають викликати під час виконання. Переконайтеся, що встановлено змінну середовища `PATH` (через панель керування **Control Panel**), щоб операційна система змогла знайти файли `.dll` під час запуску програм (наприклад, `C:\opencv-3.2\build\install\x64\vc14\bin`). Також треба визначити змінну середовища `OPENCV_DIR`, що вказує на каталог `INSTALL`. Таким чином, CMake зможе знайти бібліотеку при налаштуванні майбутніх проектів.

У середовищі Linux можна використати Cmake для створення необхідних `Makefiles`; потім завершіть інсталяцію, виконавши команду `sudo make install`. Крім того, також можна використати пакувальний інструмент `apt-get`, який взмозі автоматично виконувати повну інсталяцію бібліотеки. Для Mac OS слід використовувати менеджер пакетів `Homebrew`. Після установки треба просто набрати `brew install opencv3 --withcontrib` для того, щоб мати повну бібліотеку (запустіть `brew info opencv3`, щоб переглянути всі можливі варіанти).

OpenCV – це бібліотека, яка постійно розвивається. Версія 3 пропонує багато нових функціональних можливостей. Перехід до повного C++ API, який був ініційований у версії 2, зараз майже завершений, і реалізовані більш однорідні інтерфейси. Однією з найважливіших змін, внесених у цю нову версію, є реструктуризація модулів бібліотеки з метою полегшення її розподілу. Зокрема, створено окреме сховище, що містить останні алгоритми. Це сховище `contrib` також містить невільні алгоритми, які є

предметом певних ліцензій. Ідея полягає в тому, щоб OpenCV міг запропонувати найсучасніші функціональні можливості, якими розробники та дослідники хотіли б поділитися до того, поки не зможуть запропонувати дуже стабільне і добре підтримуване ядро API. Основні модулі, отже, ті, які отримуєте при завантаженні бібліотеки за адресою http://opencv.org/. Додаткові модулі мають бути завантажені безпосередньо з сховища розробки, розміщеного на GitHub (https://github.com/opencv/). Оскільки ці додаткові модулі постійно розвиваються, слід очікувати більш часті зміни алгоритмів, які вони містять.

Бібліотека OpenCV розділена на кілька модулів. Наприклад, модуль `opencv_core` містить основні функціональні можливості бібліотеки; модуль `opencv_imgproc` включає в себе основні функції обробки зображень; модуль `opencv_highgui` пропонує функції читання і запису зображень і відео, а також деякі функції інтерфейсу користувача і так далі. Для використання конкретного модуля необхідно включити відповідний файл заголовка верхнього рівня. Наприклад, більшість програм, які використовують OpenCV, починаються з таких декларацій:

```
#include <opencv2/core.hpp>
#include <opencv2/imgproc.hpp>
#include <opencv2/highgui.hpp>
```

В силу освоєння OpenCV виявлятимуться все більше і більше функціональних можливостей, доступних у його численних модулях. Веб–сайт OpenCV за адресою http://opencv.org/ містить докладні інструкції щодо встановлення бібліотеки, а також повну онлайнову документацію, що включає кілька посібників з різних компонентів бібліотеки.

1.2.2 Інструментарій візуалізації та модуль `cv::viz`

У деяких додатках комп'ютерне бачення використовується для реконструкції 3D-інформації сцени з зображень. При роботі з 3D-даними часто буває корисно візуалізувати результати в деякому 3D-віртуальному світі. Модуль cv::viz пропонує багато корисних функцій, які дозволяють візуалізувати об'єкти сцени і камери в 3D. Однак цей модуль побудований поверх іншої бібліотеки з відкритим кодом: VTK. Тому, якщо ви хочете використовувати модуль cv::viz, потрібно встановити на своєму комп'ютері VTK перед компілюванням OpenCV.

VTK доступний за адресою http://www.vtk.org/. Все, що потрібно зробити, це завантажити бібліотеку та використовувати CMake, щоб створити двійкові файли для вашого середовища розробки. Крім того, треба визначити змінну середовища `VTK_DIR`, вказуючи на каталог, що містить вбудовані файли. Також у налаштуваннях, запропонованих під час інсталяції OpenCV з CMake, переконайтеся, що опція `WITH_VTK` позначена.

1.3 Завантаження, відображення та збереження зображень

Настав час запустити свій перший додаток OpenCV. Оскільки OpenCV призначений для обробки зображень, роботу розпочнемо з найбільш фундаментальних операцій, необхідних при розробці програм обробки зображень. Це завантаження вхідного зображення з файлу, відображення зображення у вікні, застосування функції обробки та збереження вихідного зображення.

Використовуючи встановлену на комп'ютері IDE (наприклад, MS Visual Studio або Qt), створіть нову консольну програму з головною функцією, яка готова до заповнення.

Перше, що потрібно зробити, це включити файли заголовків, оголосивши класи і функції, які ви хочете використовувати. Тут ми просто хочемо відобразити зображення, тому потрібні файли заголовка `core`, який оголошує структуру даних зображень, а також `highgui`, який містить всі функції графічного інтерфейсу:

```
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
```

Наша основна функція починається з оголошення змінної, яка буде зберігати зображення. У OpenCV це робиться шляхом визначення об'єкта класу `cv::Mat`:

```
cv::Mat image; // Створити порожнє зображення
```

Таке визначення створює зображення розміром 0x0. Це можна підтвердити, звернувшись до атрибутів розміру `cv::Mat`:

```
  std::cout << "This image is " << image.rows << " x
"
            << image.cols << std::endl;
```

Далі простий виклик функції читання зчитує зображення з файлу, декодує його і виділяє пам'ять:

```
image= cv::imread("puppy.bmp");
// Зчитування вхідного зображення
```

Тепер ви готові використовувати це зображення. Проте спочатку слід перевірити, чи правильно прочитано зображення (помилка виникне, якщо файл не знайдений, пошкоджений або не в розпізнаваному форматі). Правомірність зображення перевіряється за допомогою наступного коду:

```
if (image.empty()) { // Обробка помилок
// зображення не створено...
// можливо відобразити повідомлення про помилку
// і вийти з програми
...
}
```

Порожній метод (`empty`) повертає `true`, якщо не було виділено даних зображення. Перше, що можна зробити з цим зображенням, це відобразити його. Це можна зробити за допомогою функцій модуля `highgui`.

Почніть з оголошення вікна, у якому ви хочете відобразити зображення, а потім вкажіть зображення, яке буде показано у цьому спеціальному вікні:

```
// визначаємо вікно (необов'язково)
cv::namedWindow("Original Image");
// показуємо зображення
cv::imshow("Original Image", image);
```

Як ви бачите, вікно ідентифікується ім'ям. Можна повторно використовувати це вікно для відображення іншого зображення пізніше, або створити кілька вікон з різними іменами. Після запуску цієї програми ви побачите вікно зображення.

Тепер непогано застосувати певну обробку зображення. OpenCV пропонує широкий вибір функцій обробки, і деякі з них розглядаються далі. Почнемо з дуже простої, яка повертає зображення горизонтально. Кілька перетворень зображень у OpenCV можуть бути виконані на місці, тобто перетворення застосовується безпосередньо до вхідного зображенні (нове зображення не створюється). Це стосується методу повертання. Проте завжди можна створити іншу матрицю для утримання результату виведення:

```
cv::Mat result; // Створюємо ще одне порожнє зображення
cv::flip(image,result,1); // позитивн. для горизонталі
                          // 0 для вертикалі,
                          // негативне для обох
```

Результат відображається в іншому вікні:

```
  cv::namedWindow("Output Image"); // the output window
  cv::imshow("Output Image", result);
```

Оскільки вікно консолі завершиться, коли досягне кінця функції `main`, додамо додаткову функцію `highgui`, щоб дочекатися натискання користувача на клавіши перш, ніж завершити програму:

```
cv::waitKey(0); // 0 для невизначеного очікування натискання клавіші
// чекаємо на позитивне значення
for
// задану кількість мілісекунд
```

Тепер можна побачити, що вихідне зображення відображається в окремому вікні.

Нарешті збережемо оброблене зображення на диску. Це робиться за допомогою наступної функції `highgui`:

```
cv::imwrite("output.bmp", result); // зберегти результат
```

Розширення файлу визначає, який кодек буде використано для збереження зображення. Інші популярні підтримувані формати зображень: JPG, TIFF і PNG.

Всі класи і функції в C++ API OpenCV визначаються в просторі імен `cv`. Існує два способи доступу до них. По-перше, передувати визначенню функції `main` наступною заявою:

```
using namespace cv;
```

Альтернативно, префікс всіх імен класу OpenCV і функцій зі специфікацією простору імен, тобто `cv::`, як будемо робити далі. Використання цього префікса полегшує ідентифікацію класів і функцій OpenCV у вашому коді.

Модуль `highgui` містить набір функцій, які дозволяють легко візуалізувати і взаємодіяти з вашими зображеннями. Коли завантажується зображення з функцією `imread`, його також можна прочитати як зображення сірого кольору. Це дуже вигідно, оскільки кілька алгоритмів комп'ютерного зору вимагають зображення сірого рівня. Перетворюючи вхідне кольорове зображення на льоту, під час його читання заощадить час і мінімізує використання пам'яті. Це можна зробити так:

```
// прочитати вхідне зображення в градаціях сірого
image= cv::imread("puppy.bmp", cv::IMREAD_GRAYSCALE);
```

Це призведе до створення зображення з беззнакових байтів (`unsigned char` у C++), яке OpenCV позначає з позначкою `CV_8U`. Альтернативно, іноді необхідно читати зображення як триканальне кольорове зображення, навіть якщо воно було збережене, як зображення сірого рівня. Це може бути досягнуто викликом функції `imread` з позитивним другим аргументом:

```
// зчитати вхідне зображення як 3-канальне кольорове
image= cv::imread("puppy.bmp", cv::IMREAD_COLOR);
```

На цей раз буде створено зображення, зроблене з 3 байтів на піксель і позначене у OpenCV, як `CV_8UC3`. Звичайно, якщо вхідне зображення було збережено, як зображення сірого рівня, всі три канали будуть містити однакове значення. Нарешті, якщо ви хочете прочитати зображення у форматі, в якому воно було збережене, просто введіть від'ємне значення, як другий аргумент. Кількість каналів на зображенні можна перевірити, використовуючи метод `channels`:

```
std::cout << "Це зображення має "
<< image.channels() << " канал(ів)";
```

Зверніть увагу, коли відкриваєте зображення з `imread` без вказівки повного шляху (як це тут зроблено). У такому випадку використовується стандартний каталог. Коли програма запускається з консолі, цей каталог, очевидно, є каталогом поточної консолі. Однак, якщо додаток запускається безпосередньо з IDE, типовим каталогом найчастіше буде той, який містить файл проекту. Отже, переконайтеся, що вхідний файл зображення розташований у правильному каталозі.

Коли ви використовуєте `imshow` для відображення зображення, що складається з цілих чисел (позначених як `CV_16U` для цілих 16-бітних беззнакових чисел або як `CV_32S` для 32-бітних цілих чисел), значення пікселів цього зображення спочатку будуть розділені на 256. Це робиться в спробі зробити його показовим з 256 сірими відтінками. Аналогічно, зображення, що складається з плаваючих точок, відображатиметься, вважаючи діапазон можливих значень між 0,0 (відображається як чорний) і 1,0 (відображається як білий). Значення за межами цього визначеного діапазону відображаються білим кольором (для значень вище 1,0) або чорним (для значень нижче 0,0).

Модуль `highgui` дуже корисний для побудови швидких прототипних додатків. Коли ви готові створити завершену версію програми, ймовірно, захочеться використати модуль графічного інтерфейсу, який пропонує ваше IDE, щоб створити програму з більш професійним виглядом.

Наша програма використовувала як вхідні, так і вихідні зображення. В якості вправи треба переписати цю просту програму таким чином, щоб вона скористалася обробкою функції на місці, тобто, не оголошуючи вихідне зображення, а записуючи його поверх:

```
cv::flip(image,image,1); // Обробка на місці
```

Модуль `highgui` містить багатий набір функцій, які допоможуть взаємодіяти з зображеннями. Використовуючи їх, ваші програми можуть реагувати на мишу або клавіатуру. Також можна малювати фігури та писати текст на зображеннях.

1.3.1 Натискання на зображення

Можна запрограмувати мишу на виконання певних операцій, коли вона знаходиться над одним з створених вікон зображення. Це робиться шляхом визначення відповідної функції зворотного виклику (**callback**). Функція зворотного виклику є функцією, яку ви явно не викликаєте, але яка викликається програмою у відповідь на певні події (тут, події, які стосуються миші, взаємодіючої з вікном зображення). Щоб розпізнати програми, функції зворотного виклику повинні мати певний підпис і бути зареєстровані. У випадку обробника подій миші функція зворотного виклику повинна мати наступний підпис:

```
void onMouse( int event, int x, int y, int flags,
void* param);
```

Перший параметр – це ціле число, яке використовується, щоб вказати, який тип події миші запустив виклик функції зворотного виклику. Інші два параметри – це просто піксельні координати розташування миші під час події. Прапори використовуються для визначення того, яка кнопка була натиснута під час спрацьовування миші. Нарешті, останній параметр використовується для передачі додаткового параметра функції у вигляді покажчика на будь-який об'єкт. Цю функцію зворотного виклику можна зареєструвати в додатку за допомогою наступного виклику:

```
cv::setMouseCallback("Original Image", onMouse,
reinterpret_cast<void*>(&image));
```

У цьому прикладі функція `onMouse` пов'язана з вікном зображення, що називається **Original Image**, а адреса відображеного зображення передається як додатковий параметр функції. Тепер, якщо ми визначимо функцію зворотного виклику `onMouse`, як продемонстровано в наступному коді, то при кожному натисканні миші значення відповідного пікселя буде показано на консолі (тут ми припускаємо, що це зображення сірого рівня):

```
  void onMouse( int event, int x, int y, int flags,
void* param) {
  cv::Mat *im= reinterpret_cast<cv::Mat*>(param);
  switch (event) { // Диспетчимо подію
 case cv::EVENT_LBUTTONDOWN: // Натиснуто ліву кноп-
ку миші
  event
  // Відображати значення пікселів на (x,y)
  std::cout << "at (" << x << "," << y << ") value
is:
              "
  << static_cast<int>(
  im->at<uchar>(cv::Point(x,y))) <<
  std::endl;
  break;
  }
 }
```

Зауважимо, що для того, щоб отримати значення пікселя при (x, y), було використано метод `at` об'єкта `cv::Mat`. Інші можливі події, які можуть бути отримані функцією зворотного виклику миші, включають `cv::EVENT_MOUSEMOVE`, `cv::EVENT_LBUTTONUP`, `cv::EVENT_RBUTTONDOWN` та `cv::EVENT_RBUTTONUP`.

1.3.2 Створення зображень

OpenCV пропонує кілька функцій для малювання фігур і написання тексту на зображеннях. Прикладами основних функцій малювання фігур є `circle` (коло), `ellipse` (еліпс), `line` (лінія) і `rectangle` (прямокутник). Нижче наведено приклад використання функції `circle`:

```
cv::circle(image, // Зображення призначення
cv::Point(155,110), // координати центру
65, // радіус
0, // Колір (тут чорний)
3); // Товщина
```

Структура `cv::Point` часто використовується в методах і функціях OpenCV для визначення координат пікселів. Зауважимо, що тут припущено, що малюнок виконується на зображенні сірого рівня; тому колір зада-

ється одним цілим числом. Можна задати значення кольору у випадку кольорових зображень, які використовують структуру `cv::Scalar`. Також можна писати текст на зображенні. Це можна зробити так:

```
cv::putText(image, // Зображення призначення
"This is a dog.", // Текст
cv::Point(40,200), // Положення тексту
cv::FONT_HERSHEY_PLAIN, // Тип шрифту
2.0, // Масштаб шрифту
255, // Колір тексту (тут білий)
white)
2); // Товщина тексту
```

Завдання: Викликати ці дві функцій на власному тестовому зображенні та зробити знімок екрану.

Зверніть увагу, що для роботи цих прикладів потрібно включити заголовок верхнього рівня `opencv2/imgproc.hpp`.

Клас `cv::Mat` – це структура даних, яка використовується для зберігання зображень (і, очевидно, інших даних матриці). Ця структура даних лежить в основі всіх класів і функцій OpenCV. Далі дано докладне пояснення цієї структури даних.

1.3.3 Дослідження структури даних cv::Mat

У попередньому розділі ми попередньо ознайомилися із структурою даних `cv::Mat`, яка є ключовим компонентом бібліотеки. Вона використовується для маніпулювання зображеннями та матрицями (насправді зображення є матрицею з обчислювальної та математичної точки зору). Оскільки ми будемо широко використовувати цю структуру даних у процесах розробки додатків, необхідно обов'язково ознайомитися з нею. Примітно, що в цьому розділі ви дізнаєтеся, що ця структура даних включає елегантний механізм керування пам'яттю.

Давайте напишемо наступну тестову програму, яка дозволить перевірити різні властивості структури даних `cv::Mat`:

```
#include <iostream>
#include <opencv2/core.hpp>
#include <opencv2/highgui.hpp>
// Тестова функція, яка створює зображення
cv::Mat function() {
// Створюємо зображення
cv::Mat ima(500,500,CV_8U,50);
// Поверемо це
return ima;
}
int main() {
// Створити нове зображен.: 240 рядків x 320 стовпців
cv::Mat image1(240,320,CV_8U,100);
```

```
cv::imshow("Image", image1); // Показати зображення
cv::waitKey(0); // Дочекатися натискання клавіші
// Повторно виділити нове зображення
image1.create(200,200,CV_8U);
    image1= 200;
cv::imshow("Image", image1); // Показати зображення
cv::waitKey(0); // Дочекатися натискання клавіші
// Створюємо червоне кольорове зображення
// Порядок каналів – BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
          // або:
// cv::Mat image2(cv::Size(320,240),CV_8UC3);
// image2= cv::Scalar(0,0,255);
cv::imshow("Image", image2); // Показати зображення
cv::waitKey(0); // Дочекатися натискання клавіші
// Прочитати зображення
cv::Mat image3= cv::imread("puppy.bmp");
// Всі ці зображення вказують на той самий блок даних
cv::Mat image4(image3);
image1= image3;
// Ці зображення є новими копіями вихідного
image3.copyTo(image2);
cv::Mat image5= image3.clone();
// Трансформувати зображення для тестування
cv::flip(image3,image3,1);
// Перевірити, на які зображення вплинула обробка
cv::imshow("Image 3", image3);
cv::imshow("Image 1", image1);
cv::imshow("Image 2", image2);
cv::imshow("Image 4", image4);
cv::imshow("Image 5", image5);
cv::waitKey(0); // Дочекатися натискання клавіші
// Отримати сіре зображення з функції
cv::Mat gray= function();
cv::imshow("Image", gray); // Показати зображення
cv::waitKey(0); // Дочекатися натискання клавіші
// Прочитати зображення в сірій шкалі
image1=                          cv::imread("puppy.bmp",
CV_LOAD_IMAGE_GRAYSCALE);
image1.convertTo(image2,CV_32F,1/255.0,0.0);
cv::imshow("Image", image2); // Показати зображення
cv::waitKey(0); // Дочекатися натискання клавіші
return 0;
}
```

Запустіть цю програму і подивіться на зображення, які вона створює.

Структура даних `cv::Mat` складається з двох частин: заголовка і блоку даних. Заголовок містить всю інформацію, пов'язану з матрицею (розмір, кількість каналів, тип даних тощо). Попередній приклад показує, як отримати доступ до деяких атрибутів цієї структури, що містяться в його заголовку (наприклад, за допомогою `cols`, `rows` або `channels`). Блок даних містить всі піксельні значення зображення. Заголовок містить змінну покажчика, яка вказує на цей блок даних; це атрибут `data`. Важливою властивістю структури даних `cv::Mat` є той факт, що блок пам'яті копіюється тільки при явному запиті. Дійсно, більшість операцій просто скопіюють заголовок `cv::Mat` таким чином, що кілька об'єктів будуть вказувати на один і той же блок даних. Ця модель управління пам'яттю робить програми більш ефективними, уникаючи витоку пам'яті, але її наслідки слід розуміти.

За замовчуванням об'єкти `cv::Mat` мають нульовий розмір при створенні, але також можна вказати початковий розмір наступним чином:

```
// Створити нове зображення 240 рядків x 320 стовпців
cv::Mat image1(240,320,CV_8U,100);
```

У цьому випадку також потрібно вказати тип кожного елемента матриці. Тут `CV_8U` відповідає 1-байтовим зображенням (відтінки сірого). Літера `U` означає, що елементи не мають знаку. Можна також оголосити знакові числа за допомогою `S`. Для кольорового зображення потрібно вказати три канали (`CV_8UC3`). Можна також оголосити цілі числа (знакові або беззнакові) розміром `16` і `32` (наприклад, `CV_16SC3`). Також є доступ до 32-бітових і 64-бітових чисел з плаваючою точкою (наприклад, `CV_32F`). Кожен елемент зображення (або матриця) може складатися з більш, ніж одного значення (наприклад, трьох каналів кольорового зображення). Тому OpenCV ввів просту структуру даних, яка використовується, коли піксельні значення передаються функціям. Це структура `cv::Scalar`, яка зазвичай використовується для зберігання одного або трьох значень. Наприклад, щоб створити кольорове зображення, ініційоване червоними пікселями, напишемо наступний код:

```
// Створюємо червоне кольорове зображення
// Порядок каналів - BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
```

Аналогічно, ініціалізація зображення зі шкалою сірого також могла бути зроблена з використанням цієї структури, написавши `cv::Scalar (100)`.

Розмір зображення часто потрібно передати до функцій. Ми вже згадували, що атрибути `cols` і `rows` можна використовувати для отримання розмірностей екземпляра `cv::Mat`. Інформація про розмір також може бути надана через структуру `cv::Size`, яка просто містить висоту і ширину матриці. Метод `size()` дозволяє отримати поточний розмір матриці. Це

формат, який використовується у багатьох методах, де повинен бути вказаний розмір матриці.

Наприклад, зображення можна створити таким чином:

```
// Створити неініціалізоване кольорове зображення
cv::Mat image2(cv::Size(320,240),CV_8UC3);
```

Блок даних зображення завжди може бути виділений або перерозподілений за допомогою методу `create`. Коли зображення вже було виділено раніше, спочатку видаляється його старий вміст. З міркувань ефективності, якщо новий запропонований розмір і тип співпадає з вже існуючим розміром і типом, то новий розподіл пам'яті не виконується:

```
// Повторно виділити нове зображення
// (тільки, якщо розмір або тип відрізняються)
image1.create(200,200,CV_8U);
```

Коли немає більше посилань на заданий об'єкт `cv::Mat`, виділена пам'ять автоматично звільняється. Це дуже зручно, оскільки дозволяє уникнути проблем з витоками загальної пам'яті, часто пов'язаних з динамічним розподілом пам'яті в C++. Цей ключовий механізм у OpenCV (введений у версії 2) виконується за допомогою здатності класу `cv::Mat` реалізувати підрахунок посилань і пусте копіювання.

Тому, коли зображення призначається іншому, дані зображення (тобто пікселі) не копіюються, а обидва зображення вказують на той самий блок пам'яті. Це також стосується зображень, які передаються або повертаються значенням. Лічильник посилань зберігається так, що пам'ять буде звільнена тільки тоді, коли всі посилання на зображення зруйновані або призначені іншому зображенню:

```
// Всі ці зображення вказують на той самий блок даних
cv::Mat image4(image3);
image1= image3;
```

Будь–яке перетворення, застосоване до одного з попередніх зображень, також вплине на інші зображення. Якщо ви хочете створити глибоку копію вмісту зображення, скористайтеся методом `copyTo`. У цьому випадку на зображенні призначення викликається метод `create`. Іншим методом, який створює копію зображення, є метод `clone`, який створює нове ідентичне зображення наступним чином:

```
// Ці зображення є новими копіями вхідного
image3.copyTo(image2);
cv::Mat image5= image3.clone();
```

У прикладі цього розділу ми застосували перетворення до `image3`. Інші зображення також містять це зображення; деякі з них мають однакові дані зображення, а інші зберігають копію цього зображення. Перевірте відображені зображення і з'ясуйте, на які зображення впливає трансформація `image3`.

Якщо потрібно скопіювати зображення в інше зображення, яке не обов'язково має мати той самий тип даних, скористайтеся методом convertTo:

```
// Перетворення у зображення з плаваючою точкою [0,1]
image1.convertTo(image2,CV_32F,1/255.0,0.0);
```

У цьому прикладі вхідне зображення копіюється у зображення з плаваючою точкою. Метод включає два необов'язкових параметра: коефіцієнт масштабування і зміщення. Зауважте, що обидва зображення повинні мати однакову кількість каналів.

Модель розподілу для об'єктів cv::Mat також дозволяє безпечно записувати функції (або методи класу), які повертають зображення:

```
cv::Mat function() {
// Створити зображення
cv::Mat ima(240,320,CV_8U,cv::Scalar(100));
// Повернути його
return ima;
}
```

Також можна визвати цю функцію з основної функції main наступним чином:

```
// Отримуємо зображення з рівнями сірого
cv::Mat gray= function();
```

Якщо ми це зробимо, змінна gray зберігатиме зображення, створене функцією, без додаткового виділення пам'яті. Дійсно, тільки порожня копія зображення буде передана з поверненого екземпляра cv::Mat до сірого зображення. Коли локальна змінна ima виходить за межі досяжності, ця змінна звільняється. Однак, оскільки пов'язаний лічильник посилань вказує, що його внутрішні дані зображення посилаються на інший екземпляр (тобто, змінна gray), його блок пам'яті не звільняється. Варто відзначити, що у випадку класів слід бути обережним і не повертати атрибути класу зображень. Ось приклад реалізації, схильної до помилок:

```
class Test {
// Атрибут зображення
cv::Mat ima;
public:
// конструктор, що створює зображення сірого рівня
Test() : ima(240,320,CV_8U,cv::Scalar(100)) {}
// Метод повертає атрибут класу, не гарна ідея...
cv::Mat method() { return ima; }
};
```

Тут, якщо функція викликає метод цього класу, вона отримує порожню копію атрибутів зображення. Якщо ця копія буде змінена пізніше, атрибут class також буде змінено, що може вплинути на подальшу поведінку класу (і навпаки). Це є порушенням важливого принципу інкапсуляції в об'єктно–орієнтованому програмуванні.

Щоб уникнути таких помилок, слід повернути клон атрибута.

Маніпулючи класом `cv::Mat`, можна виявити, що OpenCV також включає кілька інших пов'язаних класів, з якими важливо ознайомитися.

1.3.4 Вхідні та вихідні масиви

Якщо подивитися на документацію OpenCV, можна побачити, що багато методів і функцій приймають параметри типу `cv::InputArray` як вхідні дані. Цей тип є простим класом проксі, введеним для узагальнення концепції масивів у OpenCV і таким чином уникнення дублювання декількох версій одного і того ж методу або функції з різними типами вхідних параметрів. В основному це означає, що можна подати як аргумент об'єкт `cv::Mat` або інші сумісні типи. Оскільки він оголошений як вхідний масив, є гарантія, що наша структура даних не буде змінена функцією. Цікаво знати, що `cv::InputArray` також може бути побудований з популярного класу `std::vector`. Це означає, що такі об'єкти можна використовувати як вхідні параметри для методів і функцій OpenCV (однак, ніколи не використовуйте цей клас усередині своїх класів і функцій). Інші сумісні типи `cv::Scalar` і `cv::Vec`; остання структура буде представлена далі. Також існує проксі клас `cv::OutputArray`, який використовується для позначення параметрів, які відповідають зображенню, яке повертається функцією або методом.

1.3.5 Маніпулювання малими матрицями

Під час написання своїх програм доводиться маніпулювати малими матрицями. Тоді можна використовувати шаблонний клас `cv::Matx` і його підкласи. Наприклад, наступний код оголошує матрицю чисел з плаваючою точкою 3x3 з подвійною точністю і 3–елементний вектор. Потім ці дві групи розмножуються:

```
// Матриця 3x3 double
cv::Matx33d matrix(3.0, 2.0, 1.0,
2.0, 1.0, 3.0,
1.0, 2.0, 3.0);
// Матриця 3x1 (вектор)
cv::Matx31d vector(5.0, 1.0, 3.0);
// Перемноження
cv::Matx31d result = matrix*vector;
```

Як видно, до цих матриць можуть бути застосовані звичайні математичні оператори.

Повну документацію OpenCV можна знайти за адресою http://docs.opencv.org/

1.4 Визначення області інтересу

Іноді функція обробки повинна застосовуватися лише до частини зображення. OpenCV включає елегантний і простий механізм для визначення

субрегіону в зображенні і маніпулювання ним у вигляді звичайного зображення. Розглянемо, як визначати область інтересу всередині зображення.

Припустимо, треба скопіювати невелике зображення у більше. Наприклад, скажімо, потрібно вставити в тестове зображення логотип. Для цього можна визначити регіон інтересу (**Region Of Interest, ROI**), поверх якої можна застосувати операцію копіювання. Позиція ROI визначатиме, де на зображенні буде вставлено логотип.

Перший крок полягає у визначенні ROI, після чого єю можна маніпулювати як звичайним екземпляром `cv::Mat`. Ключ полягає в тому, що ROI дійсно є об'єктом `cv::Mat`, який вказує на той самий буфер даних, що й батьківське зображення, і має заголовок, який визначає координати ROI.

Після цього вставлення логотипу здійснюється наступним чином:

```
// Визначимо ROI на зображенні внизу праворуч
cv::Mat imageROI(image,
cv::Rect(image.cols-logo.cols, // Координати ROI
image.rows-logo.rows,logo.cols,logo.rows));// Розмір
// Всавляємо logo
    logo.copyTo(imageROI);
```

Тут `image` є цільовим зображенням, а `logo` – зображення логотипу (меншого розміру). Отримайте наступне зображення шляхом виконання попереднього коду.

Один із способів визначення ROI полягає у використанні екземпляра `cv::Rect`. Як видно з назви, він описує прямокутну область, у якій вказано положення лівого верхнього кута (перші два параметри конструктора) і розмір прямокутника (ширина і висота наведені в двох останніх параметрах). У нашому прикладі ми використовували розмір зображення та розмір логотипу, щоб визначити місце, де логотип буде охоплювати нижній правий кут зображення. Очевидно, що ROI завжди повинна бути повністю всередині батьківського зображення.

ROI можна також описати, використовуючи діапазони рядків і стовпців. Діапазон – це безперервна послідовність від початкового індексу до кінцевого (без урахування обох). Структура `cv::Range` використовується для представлення цієї концепції. Отже, ROI може бути визначений з двох діапазонів; у нашому прикладі ROI можна було б визначити наступним чином:

```
imageROI= image(cv::Range(image.rows-
logo.rows,image.rows),cv::Range(image.colslogo.
cols,image.cols));
```

У цьому випадку оператор `()` функції `cv::Mat` повертає інший екземпляр `cv::Mat`, який потім може бути використаний у наступних викликах. Будь–яке перетворення ROI вплине на вхідне зображення у відповідній області, оскільки зображення та ROI мають однакові дані зображення. Оскільки визначення ROI не включає копіювання даних, воно виконується за постійний проміжок часу, незалежно від розміру ROI.

Якщо треба визначити ROI, що складається з деяких рядків зображення, можна використовувати наступний виклик:

```
cv::Mat imageROI= image.rowRange(start,end);
```

Аналогічно для ROI, що складається з деяких стовпців зображення, можна використовувати наступне:

```
cv::Mat imageROI= image.colRange(start,end);
```

Методи і функції OpenCV включають багато додаткових параметрів, які не обговорюються в цих вказівках. Якщо ви бажаєте скористатися функцією вперше, завжди слід поглянути на документацію, щоб дізнатися більше про можливі варіанти, які пропонує функція. Одним з найбільш поширених варіантів є можливість визначення масок зображень.

1.4.1 Використання масок зображень

Деякі операції OpenCV дозволяють визначити маску, яка обмежить застосовність даної функції або методу, який зазвичай повинен працювати на всіх пікселях зображення. Маска – це 8-бітове зображення, яке має бути ненульовим у всіх місцях, де потрібно застосувати операцію. У місцях пікселів, які відповідають нульовим значенням маски, зображення не є недоторканим. Наприклад, метод `copyTo` можна назвати маскою. Ми можемо тут використовувати його, щоб скопіювати лише білу частину попереднього логотипу, а саме:

```
// Визначимо ROI на зображенні внизу праворуч
imageROI= image(cv::Rect(image.cols-logo.cols,
image.rows-logo.rows,
logo.cols,logo.rows));
// Використовуємо логотип як маску (має бути сірим)
cv::Mat mask(logo);
// Вставка копії тільки в місцях ненульової маски
logo.copyTo(imageROI,mask);
```

Якщо фон логотипу чорний (отже, має значення 0), його легко використовувати і як копіюване зображення, і як маску. Більшість операцій на основі пікселів OpenCV дають можливість використовувати маски.

Особливим випадком є методи `rowRange` і `colRange`, в яких початкові та кінцеві індекси однакові для визначення одиничного або однобічного ROI.

1.5 Контрольні запитання

1. До яких найбільш популярних середовищ розробки ПЗ можна підключити бібліотеку OpenCV? Що саме та чому обрано особисто Вами?
2. Де саме можна знайти останню версію бібліотеки OpenCV? Що ще містить цей ресурс?
3. Що потрібно зробити, аби завершити встановлення бібліотеки OpenCV та мати її готовою до використання?
4. На які дві основні компоненти розділено OpenCV версії 3?

5. Для чого призначені і навіщо завантажжувати додаткові модулі OpenCV?
6. Дайте стислу характеристику програмного забезпечення CMake.
7. На які модулі розділена бібліотека OpenCV? Дайте їм стислу характеристику.
8. Для чого використовується та які особливості встановлення модуля cv::viz?
9. Яким чином можна повернути зображення горизонтально?
10. У якому просторі імен визначаються всі класи і функції в C++ API OpenCV?
11. У яких числових форматах може зберігатися інформація про пікселі зображення, і яким чином позначаються ці формати в OpenCV?
12. Що таке функції зворотного виклику і з якою метою їх можна використовувати в OpenCV?
13. Дайте характеристику структурі даних `cv::Mat`(використання, властивості, склад).
14. Дайте характеристику терміну «ROI».
15. Для чого використовується та які особливості мають маски зображень?

1.6 Хід роботи

1.6.1 Підготовчі стадії

Поза навчальною лабораторією треба зробити наступне:

1. Користуючись підрозділами 1.1 – 1.3, а також рекомендованою літературою, ознайомитися із бібліотеку OpenCV, складом та особливостями її функцій.

2. Оформити першу частину звіту з виконання лабораторної роботи, в якій зазначити прізвища студентів бригади, мету лабораторних досліджень та дати письмові відповіді на контрольні питання (п. 1.4).

1.6.2 Завдання до роботи

Для того, щоб встановити бібліотеку OpenCV, освоїти базові можливості з завантаження, відображення та збереження зображень, визначити регіон інтересу, необхідно виконати наступну послідовність дій.

1. Налаштувати середовище розробки C++, зокрема – *Microsoft Visual Studio* або *Qt*.
2. Керуючись рекомендацією пункту 1.1.1, встановити бібліотеку *OpenCV* на власному комп'ютері.
3. Спираючись на дані підрозділу 1.2, створіть нову консольну програму та нове порожнє зображення для роботи з *OpenCV*.
4. Підготуйте тестове кольорове зображення у форматі зберігання растрових зображень `bmp` (*Bitmap Picture*). Наприклад, файл `Image_1-1.bmp` розміром 320x200 пікселів для подальшої обробки засобами *OpenCV* під час виконання завдань. Головна вимога до зображення – його *унікальність*,

прив'язка до конкретного студента. Це може бути, наприклад, фотографія виконавця, або його близьких, або домашніх улюбленців тощо. Під час захисту це дозволить підтвердити факт виконання лабораторної роботи та ідентифікувати автора звіту.

5. Зчитайте тестове зображення з файлу, перевірте, чи правильно воно прочитане (обовязково організуйте обробку помилок), відобразіть зображення, поверніть його в різних напрямках, відображаючи в інших вікнах.
6. Збережіть оброблені зображення на диску.
7. Прочитайте тестове зображення в градаціях сірого.
8. Прочитайте тестове зображення як 3-канальне кольорове та перевірте кількість каналів на ньому.
9. Поверніть тестове зображення горизонтально, скориставшися обробкою функції на місці.
10. Розробіть програму, яка при кожному натисканні миші буде виводити значення відповідного пікселя сірого зображення на консоль.
11. Розробіть програму, яка засобами OpenCV створює фігури (кола, еліпси, лінії та прямокутники) в різних кольорах.
12. Поверх тестового зображення введіть невеликий текст-підпис.
13. Напишіть тестову програму для перевірки різних властивостей структури даних `cv::Mat` (п. 1.2.3) і проаналізуйте зображення, які вона створює.
14. За допомогою графічного редактора створіть або підберіть готовий графічний файл логотипу у растровому форматі зберігання графічної інформації `png` (Portable Network Graphics) (наприклад, `Logo_1-1.png`) розміром приблизно 50x50 пікселів.
15. Вставте логотип у тестове зображення.
16. З використанням маски скопіювати лише білу частину логотипу у тестове зображення. Отримані нові зображення треба відобразити поруч з вхідним так само, як і в попередніх пунктах.

1.7 Вимоги до звіту по роботі

Звіт про виконання роботи повинен містити:

1. Прізвище та групу студента, а також назву, мету та завдання лабораторної роботи.
2. Відповіді на контрольні питання.
3. Таблицю з переліком категорій функцій, які підтримуються бібліотекою *OpenCV*.
4. Тексти та скрін-шоти роботи програм перевірених прикладів з коментарями.
5. Висновки, в яких треба зазначити, чи досягнута мета даної лабораторної роботи. Що саме конкретно дозволяє (перерахувати по пунктах) робити з зображеннями бібліотека *OpenCV*?
6. Електронну версію звіту з лабораторної роботи у форматі 2020_Прізвище-ЛР1.doc або *.docx разом з додатками (програмними

проектами в архіві) треба завантажити у відповідному розділі сторінки дисципліни в системі дистанційного навчання “*Moodle*”.

1.8 Орієнтовні варіанти завдань до захисту лабораторної роботи

Завдання за варіантами видаються викладачем після виконання студентом підготовчої стадії (п. 1.5.1), типових завдань (п. 1.5.2), здавання звіту та його перевірки.

2 Лабораторна робота №2. Маніпуляції з пікселями

Мета роботи: експериментально дослідити можливості доступу до значень пікселів, навчитися сканувати зображення за допомогою покажчиків та ітераторів, писати ефективні цикли сканування зображень, робити сканування зображення з сусіднім доступом, виконувати просту арифметику з зображенням та перепризначати зображення

2.1 Загальна інформація про маніпуляції з пікселями

Для того, щоб розробляти програми для комп'ютерного зору, треба мати можливість отримувати доступ до вмісту зображення і, в кінцевому підсумку, – змінювати або створювати зображення. У цій лабораторній роботі ми навчимося маніпулювати елементами зображення (також відомими, як пікселі). Ми дізнаємося, як сканувати зображення і обробляти кожен з його пікселів. Також дізнаєтмося, як це зробити ефективно, оскільки зображення навіть скромних розмірів можуть містити сотні тисяч пікселів.

Принципово те, що зображення є матрицею числових значень. Ось чому, як ми дізналися в лабораторній роботі 1, OpenCV маніпулює ними за допомогою структури даних `cv::Mat`. Кожен елемент матриці являє собою один піксель. Для зображення градацій сірого (чорно-біле зображення) пікселі є 8-бітовими значеннями без знака (тобто типу `unsigned char`), де `0` відповідає чорному, а `255` – білому.

У випадку кольорових зображень для відтворення різних видимих кольорів потрібні три первинні значення кольору. Це є наслідком того, що візуальна система людини є трихроматичною; три типи конусних клітин на сітківці ока передають інформацію про колір в наш мозок. Звідси випливає, що для кольорового зображення до *кожного пікселя* мають бути прив'язані *три* значення. У фотозйомці та цифрових зображеннях найчастіше використовуються первинні канали кольору – червоний, зелений і синій. Елемент матриці у цьому випадку складається з *триплета* 8-бітових значень. 8-бітові канали, як правило, є достатніми, хоча існують спеціалізовані програми, де потрібні 16-бітові канали (наприклад, медичні зображення).

Як ми бачили в попередній лабораторній роботі, OpenCV дозволяє створювати матриці (або зображення) з піксельними значеннями інших типів, наприклад, `CV_32U` або `CV_32S`, а також з плаваючою точкою `CV_32F`. Це дуже корисно для зберігання, наприклад, проміжних значень у деяких завданнях обробки зображень. Більшість операцій можна застосовувати на матрицях будь-якого типу; проте інші вимагають певного типу або працюють тільки з заданим числом каналів. Тому для того, щоб уникнути поширених помилок у програмуванні, важливо розуміти передумови функції.

У якості прикладу в цій лабораторній роботі потрібно використовувати власні кольорові зображення форматом jpg щонайменше 320x240.

Для того, щоб отримати доступ до кожного окремого елемента матриці, потрібно просто вказати його рядки та номери стовпців. Це дозволить

повернути відповідний елемент, який може бути єдиним числовим значенням або вектором значень у випадку багатоканального зображення.

2.1.1 Доступ до значень пікселів

Щоб проілюструвати прямий доступ до значень пікселів, ми створимо просту функцію, яка додає до зображення шум «соль» та «перець». Як випливає з назви, шум «соль» та «перець» – це особливий тип шуму, в якому деякі випадково вибрані пікселі замінюються білим або чорним пікселем. Цей тип шуму може виникати при несправній комунікації, коли значення деяких пікселів втрачається під час передачі. У нашому випадку просто випадковим чином виділимо кілька пікселів і присвоїмо їм білий колір.

Створимо функцію, яка отримує вхідне зображення, яке буде змінено нашою функцією. Другий параметр – це кількість пікселів, які ми хочемо зробити білими:

```
void salt(cv::Mat image, int n) {
// Генератор випадкових чисел C++11
std::default_random_engine generator;
std::uniform_int_distribution<int>
                      randomRow(0, image.rows - 1);
std::uniform_int_distribution<int>
                      randomCol(0, image.cols - 1);


int i,j;
for (int k=0; k<n; k++) {

// Координати випадкового зображення
i= randomCol(generator);
j= randomRow(generator);

if (image.type() == CV_8UC1) { // Сіре зображення
// одноканальне 8-бітове зображення
image.at<uchar>(j,i)= 255;

} else if (image.type() == CV_8UC3) {
// Кольорове зображення
// 3-канальне зображення
image.at<cv::Vec3b>(j,i)[0]=  255;
image.at<cv::Vec3b>(j,i)[1]=  255;
image.at<cv::Vec3b>(j,i)[2]=  255;
          }
     }
```

```
}
```

Наведена функція складається з єдиного циклу, який присвоює `n` разів величину `255` випадково вибраним пікселям. Тут стовпчик пікселів `i` і рядок `j` вибираються за допомогою генератора випадкових чисел. Зауважимо, що за допомогою методу `type` ми розрізняємо випадки сірих і кольорових зображень. У випадку зображення у градаціях сірого число `255` присвоюється одному 8-бітовому значенню. Щоб отримати білий піксель на кольоровому зображенні, потрібно призначити `255` трьом первинним кольоровим каналам.

Викличемо цю функцію, передавши їй раніше відкрите зображення:

```
// Відкрити зображення
cv::Mat image= cv::imread("test2_1.jpg",1);

// Викликаемо функцію, щоб додати шум
salt(image,3000);

// Відображення результату
cv::namedWindow("Image");
cv::imshow("Image",image);
```

Клас `cv::Mat` включає кілька методів доступу до різних атрибутів зображення. Загальнодоступні змінні члена класу `cols` і `rows` дають кількість стовпців і рядків у зображенні. Для елемента доступу `cv::Mat` має метод `at(int y, int x)`, в якому `x` – номер стовпця, а `y` – номер рядка. Тим не менш, тип, повернутий методом, повинен бути відомий під час компіляції, і, оскільки `cv::Mat` може утримувати елементи будь-якого типу, програміст має вказати тип повернення, який очікується.

Тому метод `at` був реалізований як шаблонний метод. Отже, коли ви його називаєте, треба вказати тип елемента зображення таким чином:

```
image.at<uchar>(j,i)= 255;
```

На програміста полягає відповідальність за те, аби переконатися, що вказаний тип відповідає типу, який міститься в матриці. Метод `at` не виконує жодного перетворення типу.

У кольорових зображеннях кожен піксель пов'язаний з трьома компонентами: червоним, зеленим і синім каналами. Тому клас `cv::Mat`, який містить кольорове зображення, поверне вектор з трьох 8-бітових значень. OpenCV визначив тип для таких коротких векторів, і він називається `cv::Vec3b`. Це вектор з трьох непідписаних символів. Ось чому доступ до елементів колірного пікселя записується наступним чином:

```
image.at<cv::Vec3b>(j,i)[channel]=value;
```

Індекс `channel` позначає один з трьох каналів кольору. OpenCV зберігає значення каналу в порядку синього, зеленого і червоного (отже, синій – це канал 0). Також можна безпосередньо використовувати структуру коротких векторних даних і записувати:

```
image.at<cv::Vec3b>(j, i) = cv::Vec3b(255, 255, 255);
```

Подібні типи векторів також існують для 2-елементних і 4-елементних векторів (`cv::Vec2b` і `cv::Vec4b`), а також для інших типів елементів. Наприклад, для 2-елементного вектора з плаваючою крапкою остання буква назви типу буде замінена на `f`, тобто `cv::Vec2f`. У випадку короткого цілого числа остання буква замінюється на `s`. Ця ж буква використовується і для цілого числа, а `d` – для вектора з плаваючою точкою з подвійною точністю. Всі ці типи визначаються за допомогою класу шаблонів `cv::Vec <T, N>`, де `T` – тип, а `N` – кількість елементів вектора.

Останнє зауваження. Може здивувати той факт, що наша функція модифікації зображень використовує параметр передачі за значенням. Це працює тому, що під час копіювання зображення вони мають однакові дані.

Отже, не обов'язково передавати зображення за допомогою посилання, коли треба змінити їхній вміст. До речі, параметри переходу за значенням часто спрощують оптимізацію коду для компілятора.

2.1.2 Клас шаблону cv::Mat

Використання методу `at` класу `cv::Mat` іноді може бути громіздким, оскільки повернений тип має бути вказаний, як аргумент шаблону в кожному виклику. У тих випадках, коли тип матриці відомий, можна використовувати клас `cv::Mat_`, який є шаблоном підкласу `cv::Mat`. Цей клас визначає кілька додаткових методів, але не має нових атрибутів даних, так що покажчики або посилання на один клас можуть бути безпосередньо перетворені в інший клас.

Серед додаткових методів є оператор `()`, який дозволяє прямий доступ до елементів матриці. Тому, якщо зображення є змінною `cv::Mat`, що відповідає матриці `uchar`, можна написати наступний код:

```
// Використовуємо зображення з шаблоном
Mat_ cv::Mat_<uchar> img(image);
img(50,100)= 0; // доступ до рядка 50 і стовпця 100
```

Оскільки тип елементів `cv::Mat_` оголошується при створенні змінної, метод оператора `()` знає, який тип повертається під час компіля-

ції. За винятком того факту, що його коротше писати, використання методу оператора `()` забезпечує точно такий же результат, як і метод `at`.

2.2 Сканування зображення за допомогою покажчиків

У більшості завдань обробки зображень потрібно сканувати усі пікселі зображення, щоб виконати обчислення. Враховуючи велику кількість пікселів, які потрібно охопити, необхідно виконати це завдання ефективно. Цей і наступний приклади покажуть різні способи реалізації ефективних циклів сканування. Ці приклади використовують арифметику вказівника.

2.2.1 Зменшення кількості кольорів у зображенні

Проілюструємо процес сканування зображень, виконавши просте завдання: зменшення кількості кольорів у зображенні.

Кольорові зображення складаються з 3-канальних пікселів. Кожен з каналів відповідає значенню інтенсивності одного з трьох основних кольорів: червоного, зеленого та синього.

Оскільки кожне з цих значень є 8-бітним символьним беззнаковим, загальна кількість кольорів становить `256x256x256`, що складає більше 16 мільйонів кольорів. Отже, щоб зменшити складність аналізу, іноді корисно зменшити кількість кольорів у зображенні. Одним із способів досягнення цієї мети є простий поділ простору RGB на кубики рівних розмірів. Наприклад, якщо ми зменшимо кількість кольорів у кожному вимірі у 8 разів, то отримаємо загалом `32x32x32` кольори.

Кожному кольору на оригінальному зображенні потім присвоюється нове значення кольору, яке відповідає значенню в центрі куба, до якого належить піксель.

Тому основний алгоритм зменшення кількості кольорів простий. Якщо N – коефіцієнт зменшення, треба значення кожного пікселя поділити на N (тут береться ціле ділення, отже, залишок втрачається), потім помножити результат на N.

Це дасть нам *кратне* значення N трохи нижче значення вхідного пікселя. Додаємо $N/2$ і отримуємо центральне положення інтервалу між двома сусідніми кратними N. Якщо повторити цей процес для кожного 8-бітного значення каналу, то отримаємо всього `256/N x 256/N x 256/N` можливих значень кольорів.

Запис функції зменшення кольорів буде наступним:

```
void colorReduce(cv::Mat image, int div=64);
```

Користувач надає зображення та коефіцієнт зменшення на канал. Тут обробка виконується на місці, тобто значення пікселів вхідного зображення змінюються функцією. Обробка здійснюється просто шляхом створення

подвійного циклу, який переходить через всі значення пікселів наступним чином:

```
void colorReduce(cv::Mat image, int div=64) {
int nl= image.rows; // Кількість рядків
// Загальна кількість елементів на рядок
int nc= image.cols * image.channels();
for (int j=0; j<nl; j++)  {
// Отримуємо адресу даних рядка j
uchar* data = image.ptr<uchar>(j);

      for (int i=0; i<nc; i++)  {
      // Обробляємо кожен піксель ------------------
      data[i]= data[i]/div*div + div/2;
      // Завершення обробки пікселів ----------------
                                  } // Кінець рядка
                            }
}
```

Цю функцію можна перевірити за допомогою наступного фрагмента коду:

```
// Читаемо зображення
image = cv::imread("test2_1.jpg");
// Обробляемо зображення
colorReduce(image,64);
// Показуемо зображення
cv::namedWindow("Image");
cv::imshow("Image",image);
```

2.2.2 Додавання пікселів та ефективна ширина рідка зображення

У кольоровому зображенні перші три байти буфера даних зображень являють собою 3-канальні значення верхнього лівого пікселя, наступні три байти – значення другого пікселя першого рядка і так далі (пам'ятайте, що OpenCV використовує за замовчуванням порядок каналів BGR). Зображення шириною `W` і висотою `H` потребують блоку пам'яті з `WxHx3 uchar`. Однак для підвищення ефективності довжину рядка можна доповнити кількома *додатковими* пікселями. Річ у тім, що іноді обробка зображень більш ефективна, коли рядки кратні, наприклад, 8. Тоді вони краще узгоджуються з конфігурацією локальної пам'яті. Очевидно, що ці додаткові пікселі не відображаються або не зберігаються, а їх точні значення ігноруються.

OpenCV позначає довжину доповненого рядка як *ефективну* ширину. Очевидно, що, якщо зображення не було доповнене додатковими пікселями, ефективна ширина буде дорівнювати *реальній* ширині зображення. Ми вже дізналися, що атрибути `cols` і `rows` надають ширину та висоту зо-

браження. Аналогічно, атрибут кроку даних надає ефективну ширину в кількості байтів. Навіть, якщо зображення має тип, відмінний від `uchar`, дані кроку все одно дають нам кількість байтів поспіль.

Розмір елемента пікселя в байтах задається методом `elemSize` (наприклад, для триканальної короткої цілочисельної матриці (`CV_16SC3`), `elemSize` поверне `6`). Нагадаємо, що кількість каналів у зображенні задається методом `nchannels` (який буде `1` для зображення сірого рівня та `3` для кольорового зображення). Нарешті, метод `total` повертає загальну кількість пікселів (тобто записів матриці) в зображенні.

Кількість пікселів у кожному рядку задається наступним кодом:

```
int nc= image.cols * image.channels();
```

Щоб спростити арифметичні обчислення покажчиків, клас `cv::Mat` пропонує метод, який безпосередньо дає початкову адресу рядка зображень. Це метод шаблону `ptr`, який повертає адресу рядка з номером `j`:

```
uchar* data= image.ptr<uchar>(j);
```

Зауважимо, що в операторі обробки можна еквівалентно використовувати арифметику вказівника для переходу від стовпця до стовпця. Отже, можна було б написати наступний код:

```
*data++= *data/div*div + div2;
```

Функція зменшення кольорів, представлена раніше, є лише одним із способів виконання цього завдання. Можна також використовувати інші формули для зменшення кольорів. Більш загальна версія функції дозволяє визначати також різні вхідні та вихідні зображення. Сканування зображень також може бути більш ефективним, беручи до уваги безперервність даних зображення. Нарешті, також можна використовувати регулярну арифметику покажчика низького рівня для сканування буфера зображень. Всі ці елементи обговорюються далі.

2.2.3 Інші методи зменшення кольорів

У нашому прикладі зменшення кольору досягається шляхом використання переваг цілочисельного ділення, яке дає результат у вигляді найближчого нижнього цілого числа наступним чином:

```
data[i]= (data[i]/div)*div + div/2;
```

Редукований колір можна також обчислити за допомогою оператора модуля наступним чином:

```
data[i]= data[i] - data[i]%div + div/2;
```

Інший варіант – це використання побітових операторів. Дійсно, якщо обмежити коефіцієнт скорочення до ступеня 2, тобто `div = pow (2, n)`, то маскування перших `n` біт значення пікселя дасть найближчий менший кратний дільник `div`. Ця маска буде обчислена за допомогою простого зсуву бітів наступним чином:

```
// Маскування, що використовується для округлення значення пікселя
uchar mask= 0xFF<<n; // тобто для div=16, mask= 0xF0
```

Зменшення кількості кольорів буде надано наступним кодом:

```
*data &= mask; // маскування
*data++ += div>>1;  // add div/2;
```

(Тут також можна було використати побітове АБО, а не +).

Загалом, побітові операції можуть стати потужною альтернативою, коли ефективність коду – головна вимога.

2.2.4 Аргументи введення і виведення

У нашому прикладі зменшення кольору застосовується *безпосередньо* до вхідного зображення, що називається *перетворенням на місці*. Таким чином, ніякого додаткового зображення не потрібно для виведення результату, що може заощадити використання пам'яті. Однак у деяких додатках користувач може захотіти зберегти вихідне зображення недоторканим. Тоді доведеться створити копію зображення **до** виклику функції. Зверніть увагу, що найпростіший спосіб створити ідентичну глибоку копію зображення – викликати метод `clone ()`. Прикладом є наступний код:

```
// Читаемо зображення
image= cv::imread("test2_1.jpg");
// Клонуемо зображення
cv::Mat imageClone= image.clone();
// Обробка клона. Оригінальне зображення
// залишається недоторканим!
colorReduce(imageClone);
// Відображення результату
cv::namedWindow("Image Result");
cv::imshow("Image Result",imageClone);
```

Додаткових витрат можна уникнути, визначивши функцію, яка дає можливість використовувати або не використовувати обробку на місці. Підпис методу буде наступним:

```
void colorReduce(const cv::Mat &image,
                                    // вхідне зображення
                 cv::Mat &result, // вихідне зображення
                 int div=64);
```

Зверніть увагу, що вхідне зображення тепер передається з посиланням `const`, що означає, що це зображення не буде змінено функцією. Вихідне зображення передається в якості посилання так, що викликаюча функція буде бачити вихідний аргумент, модифікований цим викликом. Коли бажана обробка на місці, той же образ задається як вхідний, так і вихідний:

```
colorReduce(image,image);
```

Якщо ні, може бути наданий інший примірник `cv::Mat`:

```
cv::Mat result; colorReduce(image,result);
```

Ключ тут полягає в тому, щоб спочатку перевірити, чи має вихідне зображення виділений буфер даних з типом розміру і пікселя, які відповідають таким у вхідному зображенні. Ця перевірка інкапсульована всередині методу `create` `cv::Mat`, що дуже зручно. Цей метод слід використовувати, коли матрицю необхідно перерозподілити з новим розміром і типом. Якщо матриця вже має вказаний розмір і тип, то не виконується жодна операція, і метод просто повертається, не торкаючись примірника.

Тому наша функція повинна просто починатися з виклику `create`, який будує матрицю (*якщо необхідно*) того ж розміру і типу, що і вхідне зображення:

```
result.create(image.rows,image.cols,image.type());
```

Розподілений блок пам'яті має розмір `total () * elemSize ()`. Потім сканування виконується за допомогою двох покажчиків:

```
for (int j=0; j<nl; j++) {

// Отримати адреси рядків вводу та виводу j
const uchar* data_in= image.ptr<uchar>(j);
uchar* data_out= result.ptr<uchar>(j);

for (int i=0; i<nc*nchannels; i++) {
```

```
// Обробляємо кожен піксель ---------------------

data_out[i]= data_in[i]/div*div + div/2;

// Завершення обробки пікселів ----------------

     } // Кінець рядка
}
```

У тих випадках, коли одне і те ж зображення забезпечується як вхід і вихід, ця функція стає повністю еквівалентною першій версії, представленій у цьому прикладі. Якщо в якості вихідного зображення надається інше зображення, функція працюватиме коректно, незалежно від того, чи було виділено зображення до виклику функції.

Нарешті, зверніть увагу, що два параметри цієї нової функції могли бути оголошені як `cv::InputArray` і `cv::OutputArray`. Як обговорювалося в Лабораторній роботі №1, вони забезпечували б таку ж поведінку, але приносять додаткову гнучкість в термінах типу аргументів, які вони можуть прийняти.

2.3 Ефективне сканування безперервних зображень

Раніше ми пояснили, що з міркувань ефективності зображення може бути доповнено додатковими пікселями в кінці кожного рядка. Цікаво відзначити, що, навіть, коли пікселі не додаються, зображення також можна розглядати як довгий одновимірний масив пікселів `WxH`.

Зручний метод `cv::Mat` може сказати нам, чи було зображення змінено. Це метод `isContinuous`, який повертає `true`, якщо зображення не містить доданих пікселів. Зауважимо, що ми могли б також перевірити безперервність матриці, написавши наступний тест:

```
// Перевірити, чи розмір рядка (у байтах) дорівнює
// кількості стовпців пікселів у байтах
image.step == image.cols*image.elemSize();
```

Щоб бути повним, цей тест також повинен перевірити, чи матриця має тільки один рядок. У цьому випадку вона є постійною за визначенням. Тим не менш, завжди використовуйте метод `isContinuous` для перевірки стану безперервності. У деяких специфічних алгоритмах обробки можна скористатися безперервністю зображення, обробляючи його в одному (більш довгому) циклі. Нашу функцію обробки запишемо наступним чином:

```
void colorReduce(cv::Mat image, int div=64) {
```

```
int nl= image.rows; // Кількість рядків
// Загальна кількість елементів на рядок
int nc= image.cols * image.channels();

if (image.isContinuous()){
// То пікселі не додаються
nc= nc*nl;
nl= 1;    // тепер це 1D масив
}

int n= staic_cast<int>(
log(static_cast<double>(div))/log(2.0) + 0.5);
// Маскування, яке використовується для округлення
значення пікселя
uchar mask= 0xFF<<n; // Тобто для div=16 mask= 0xF0
uchar div2 = div >> 1; // div2 = div/2

// Цей цикл виконується тільки один раз
// у випадку безперервних зображень
for (int j=0; j<nl; j++) {

uchar* data= image.ptr<uchar>(j);
for (int i=0; i<nc; i++) {
*data &= mask;
*data++ += div2;
          } // Кінець рядка
     }
}
```

Тепер, коли тест безперервності говорить нам, що зображення не містить доданих пікселів, ми виключаємо зовнішній цикл, встановлюючи ширину до `1`, а висоту – `WxH`. Також існує спосіб зміни формату, який можна тут використати. У цьому випадку пишемо наступне:

```
if (image.isContinuous())
{
// Не розміщено пікселів
image.reshape(1,    // Нове число каналів
              1);   // Нове число рядків
}

int nl= image.rows; // Кількість рядків
int nc= image.cols * image.channels();
```

Метод `reshape ()` змінює розміри матриці без необхідності копіювання або перерозподілу пам'яті. Перший параметр – це нове число каналів, а другий – нове число рядків. Кількість стовпців повторно коригується.

У цих реалізаціях внутрішній цикл обробляє всі пікселі зображення послідовно.

2.4 Низькорівнева арифметика покажчиків

У класі `cv::Mat` дані зображення містяться в блоці пам'яті беззнакових байтів. Адреса першого елемента цього блоку пам'яті задається атрибутом `data`, який повертає беззнаковий покажчик `char`. Отже, щоб почати цикл з початку зображення, можна написати наступний код:

```
uchar *data = image.data;
```

А переміщення від одного рядка до іншого можна зробити шляхом переміщення покажчика рядка за допомогою ефективної ширини наступним чином:

```
data+= image.step;   // наступний рядок
```

Атрибут `step` дає загальну кількість байтів (у тому числі – додані пікселі) у рядку. Загалом, можна отримати адресу пікселя в рядку `j` і стовпець `i` наступним чином:

```
// Адреса пікселя в (j, i), тобто &image.at (j, i)
data= image.data+j*image.step+i*image.elemSize();
```

Однак, навіть якщо це буде працювати в даному прикладі, продовжувати цей шлях не рекомендується.

2.5 Сканування зображення за допомогою ітераторів

У об'єктно-орієнтованому програмуванні цикл над збіркою даних зазвичай здійснюється з використанням ітераторів. Ітератори – це спеціалізовані класи, які побудовані для перегляду кожного елемента колекції, приховуючи, як ітерація кожного елемента спеціально виконується для даної колекції. Це застосування принципу приховування інформації робить сканування колекції простішим, безпечнішим і схожим за формою незалежно від того, який тип колекції використовується. **Стандартна бібліотека шаблонів** (STL) має клас ітераторів, асоційований з кожним з класів колекції. OpenCV пропонує клас ітератора `cv::Mat`, сумісний зі стандартними ітераторами, знайденими в C++ STL.

Об'єкт ітератора для екземпляра `cv::Mat` можна отримати, спочатку створивши об'єкт `cv::MatIterator_`. Як і у випадку з `cv::Mat_`, підкреслення вказує, що це шаблонний підклас.

Дійсно, оскільки ітератори зображень використовуються для доступу до елементів зображення, тип повернення повинен бути відомий під час компіляції. Так, ітератор для кольорового зображення оголошується наступним чином:

```
cv::MatIterator_<cv::Vec3b> it;
```

Альтернативно також можна використовувати тип ітератора, визначений всередині `Mat_`. Шаблон класу наступний:

```
cv::Mat_<cv::Vec3b>::iterat it;
```

Потім ви перекриваєте пікселі за допомогою звичайних методів ітератора `start` і `end`, за винятком того, що вони, знову ж таки, є методами шаблону. Отже, наша функція зменшення кольору тепер записується наступним чином:

```
void colorReduce(cv::Mat image, int div=64) {
// div має бути ступенем 2
int n= staic_cast<int>(
log(static_cast<double>(div))/log(2.0) + 0.5);
// Маскування, що використовується для округлення
значення пікселя
uchar mask= 0xFF<<n; // тобто для div=16, mask= 0xF0
uchar div2 = div >> 1; // div2 = div/2
// Отримуемо ітератори
cv::Mat_<cv::Vec3b>::iterator it=image.begin <cv::Vec3b> ();
cv::Mat_<cv::Vec3b>::iterator itend=image.end <cv::Vec3b> ();
// Сканування всіх пікселів
for ( ; it!= itend; ++it) {

      (*it)[0]&= mask;
      (*it)[0]+= div2;
      (*it)[1]&= mask;
      (*it)[1]+= div2;
      (*it)[2]&= mask;
      (*it)[2]+= div2;
      }
}
```

Пам'ятайте, що тут ітератор повертає примірник `cv::Vec3b`, оскільки ми обробляємо кольорове зображення. Доступ до кожного елементу каналу кольорів здійснюється за допомогою оператора розіменування `[]`. Зауважте, що також можна покластися на перевантажені оператори `cv::Vec3b` і просто написати:

```
*it= *it/div*div+offset;
```

Це дозволить застосувати операції на кожному елементі короткого вектора.

Робота з ітераторами завжди йде за однією схемою, незалежно від того, який тип сканування відбувається.

По-перше, ми створюємо об'єкт ітератора, використовуючи відповідний спеціалізований клас, який у нашому прикладі є `cv::Mat_ <cv::Vec3b> :: iterator` (або `cv::MatIterator_ <cv:: Vec3b>`).

Потім отримуємо ітератор, ініціалізований у початковому положенні (у нашому прикладі – лівий верхній кут зображення). Це робиться за допомогою методу `begin`. З екземпляром `cv::Mat` кольорового зображення ми отримуємо його як `image.begin<cv::Vec3b>()`. Також можна використовувати арифметику на ітераторі. Наприклад, якщо ви хочете почати з другого рядку зображення, можна ініціалізувати ітератор `cv::Mat` на `image.begin <cv::Vec3b> () + image.cols`. Кінцеве положення вашої колекції отримується аналогічно, але з використанням методу `end`. Однак, отриманий таким чином ітератор знаходиться за межами колекції. Ось чому ітераційний процес повинен зупинитися, коли він досягне кінцевої позиції. Також, якщо треба зупинитись до останнього рядка, можна використовувати арифметику на цьому ітераторі. Остаточна ітерація припиниться, коли ітератор досягне `image.end <cv::Vec3b> () - image.cols`.

Після ініціалізації ітератора створюємо цикл, який переходить до всіх елементів, доки не буде досягнуто кінця. Типовий цикл `while` буде виглядати наступним чином:

```
while (it!= itend) {

// Оброблемо кожен піксель ---------------------
...
// Завершення обробки пікселів ----------------
++it;
}
```

Оператор ++ – саме той, який буде використовуватися для переходу до наступного елемента. Також можна вказати більший розмір кроку. Наприклад, `it + = 10` оброблятиме кожні `10` пікселів зображення.

Нарешті, всередині циклу обробки ми використовуємо оператор розіменування `*` для того, щоб отримати доступ до поточного елементу, використовуючи який, можна прочитати (наприклад, `element = * it;`) або записати (наприклад, `* it = element;`). Зауважимо, що також можна створювати константні ітератори, які використовуються, якщо ми отримуємо посилання на `const cv::Mat` або якщо треба позначити, що

поточний цикл не змінює примірник `cv::Mat`. Вони оголошуються наступним чином:

```
cv::MatConstIterator_<cv::Vec3b> it;
```

Або можуть бути заявлені наступним чином:

```
cv::Mat_<cv::Vec3b>::const_iterator it;
```

2.6 Написання ефективних циклів сканування зображень

У попередніх прикладах ми представили різні способи сканування зображення для обробки його пікселів. У цьому прикладі ми порівняємо ефективність різних підходів.

Коли створюємо функцію обробки зображень, часто викликає занепокоєння обчислювальна ефективність коду. Перевірка ефективності потрібна для виявлення вузького місця в обробці, яке може сповільнити програму.

Однак важливо відзначити, що, якщо це не потрібно, оптимізація не повинна здійснюватися ціною зниження *ясності* коду. Простий код дійсно, завжди легше налагоджувати і підтримувати. Тільки частини коду, які є критичними для ефективності програми, мають бути сильно оптимізовані.

Для того, щоб виміряти час виконання функції або частини коду, існує дуже зручна функція OpenCV, звана `cv::getTickCount ()`. Ця функція дає кількість тактових циклів, що сталися з часу останнього запуску комп'ютера. Оскільки ми хочемо оцінити час виконання, ідея полягає в тому, щоб отримати це число тактових циклів до і після виконання деякого коду. Щоб отримати час виконання в секундах, використаємо інший метод, `cv::getTickFrequency ()`. Це дає кількість циклів за секунду, якщо процесор має фіксовану частоту (що *не обов'язково* стосується нових процесорів!). Звичайний шаблон, який слід використовувати для отримання часу обчислення заданої функції (або частини коду), буде таким:

```
const int64 start = cv::getTickCount();
colorReduce(image); // виклик функції
// минулий час у секундах
double duration = (cv::getTickCount()-start)/
                                  cv::getTickFrequency();
```

Далі описаний час виконання різних реалізацій функції `colorReduce`. Абсолютні числа виконання можуть відрізнятися від однієї машини до іншої (тут використовували машину з частотою 2,40 ГГц, обладнану 64-розрядним процесором Intel Core i7). Досить цікаво подивитися на відносну різницю. Ці результати також залежать від конкретного компілятора, який використовується для створення виконуваного файлу. Дані те-

сти інформують про середній час зменшення кольорів тестового зображення з роздільною здатністю `320x240` пікселів.

Тестування проведено на трьох різних конфігураціях:

1. Машина 2,5 ГГц, обладнана 64-розрядним процесором Intel i5 і компілятором Visual Studio 14 2015 під Windows 10.
2. Машина потужністю 3,6 ГГц, 64-розрядний Intel i7 і gcc 4.9.2 під Ubuntu Linux.
3. 2011 MacBook Pro 2.3 GHz Intel i5 та clang++ 7.0.2.

По-перше, показане порівняння трьох способів обчислення зменшення кольору методом *Сканування зображення за допомогою покажчиків*.

	Конфігурація 1	**Конфігурація 2**	**Конфігурація 3**
Ціле ділення	0.867 ms	0.586 ms	1.119 ms
Оператор модуля	0.774 ms	0.527 ms	1.106 ms
Побітний оператор	0.015 ms	0.013 ms	0.066 ms

Цікаво, що формула, яка використовує побітовий оператор, *набагато швидше*, ніж інші. Два інших способи мають подібний час роботи. Ось чому важливо витрачати час на визначення найбільш ефективного способу обчислення результату в циклі зображення, оскільки чистий вплив може бути дуже значним.

У циклі слід уникати повторюваних обчислень значень на користь попереднього обчислення. Наприклад, було б поганою ідеєю написати внутрішню функцію зменшення кольору наступним чином:

```
for (int i=0; i<image.cols * image.channels(); i++) {
*data &= mask;
*data++ += div/2;
```

Дійсно, в цьому коді цикл повинен обчислювати загальну кількість елементів у рядку, і результат `div/2` знову і знову. Кращий код – наступний:

```
int nc= image.cols * image.channels();
uchar div2= div>>1;

for (int i=0; i<nc; i++) {
*(data+i) &= mask;
*(data+i) += div2;
}
```

У середньому код з повторними обчисленнями в 10 разів повільніше, ніж більш оптимальне рішення. Зауважимо, однак, що деякі компілятори могли б оптимізувати ці види циклів і отримати ефективний код.

Версія функції зменшення кольору, яка використовує ітератори (і побітові оператори), як показано в прикладі *Сканування зображення з ітераторами*, дає більш повільні результати при 0,480 мс, 0,320 мс і 0,655 мс для розглянутих трьох конфігурацій. Основна мета ітераторів – спростити процес сканування зображень і зробити його менш схильним до помилок.

Для повноти ми також реалізували версію функції, яка використовує метод at для доступу до пікселів. Головний цикл цієї реалізації буде просто читатися наступним чином:

```
for (int j=0; j<nl; j++) {
     for (int i=0; i<nc; i++) {

image.at<cv::Vec3b>(j,i)[0]=
     image.at<cv::Vec3b>(j,i)[0]/div*div + div/2;
image.at<cv::Vec3b>(j,i)[1]=
     image.at<cv::Vec3b>(j,i)[1]/div*div + div/2;
image.at<cv::Vec3b>(j,i)[2]=
     image.at<cv::Vec3b>(j,i)[2]/div*div + div/2;
     } Кінець рядка
}
```

Ця реалізація має більш повільний час виконання 0,925 мс, 0,580 мс і 1,128 мс. Цей метод повинен використовуватися тільки для довільного доступу до пікселів зображення, але ніколи – при скануванні зображення.

Крім того, більш короткий цикл з декількома операторами, як правило, більш ефективно виконується, ніж більш довгий цикл над однією операцією, навіть якщо загальна кількість оброблюваних елементів однакова. Аналогічно, якщо у нас є *N* різних обчислень, що застосовуються до пікселя, застосуйте їх усі в одному циклі, замість написання *N* послідовних циклів, по одному для кожного обчислення.

Також виконувся тест безперервності, який створює один цикл у випадку безперервних зображень замість звичайного подвійного циклу над рядками та стовпцями. Було отримане незначне скорочення часу роботи в середньому на 10%. Загалом, добре використовувати цю стратегію, оскільки вона може призвести до значного збільшення швидкості.

2.7 Сканування зображення з сусіднім доступом

При обробці зображень загальноприйнято мати функцію обробки, яка обчислює значення в кожному розташуванні пікселя на основі значення сусідніх пікселів. Коли ця околиця включає пікселі попереднього і наступного рядків, нам потрібно одночасно сканувати кілька рядків зображення.

Щоб проілюструвати рішення цієї задачі, ми застосуємо функцію обробки, яка загострює зображення. Вона заснована на операторі Лапласа (що буде розглянуто в лабораторній роботі *Фільтрація зображень*). Це дійсно добре відомий результат в обробці зображень: якщо відняти лапласіан від зображення, ребра зображення посилюються, тим самим даючи більш чітке зображення.

Це загострене значення обчислюється наступним чином:

```
sharpened_pixel= 5*current-left-right-up-down;
```

Тут `left` є піксель, який знаходиться безпосередньо з лівого боку поточного, а `up` – відповідний на попередньому рядку і так далі.

На цей раз обробка не може бути виконана на місці. Користувач має надати вихідне зображення. Сканування зображень здійснюється за допомогою трьох покажчиків: один для поточного рядка, другий для рядка вище, а третій для рядка нижче. Крім того, оскільки кожне обчислення пікселів вимагає доступу до сусідів, неможливо обчислити значення для пікселів першого і останнього рядка зображення, а також пікселів першого і останнього стовпців. Цикл можна записати наступним чином:

```
void sharpen(const cv::Mat &image, cv::Mat &result) {

// Виділити за необхідності
result.create(image.size(), image.type());
int nchannels = image.channels(); // отримати кіль-
кість каналів
// Для всіх рядків (крім першого та останнього)
 for (int j= 1; j<image.rows-1; j++) {
const uchar* previous= image.ptr<const uchar>(j-1);
// Попередній рядок
const uchar* current= image.ptr<const uchar>(j);
// Поточний рядок
const uchar* next= image.ptr<const uchar>(j+1);
// Наступний рядок
uchar* output= result.ptr<uchar>(j);
// Вихідний рядок

for (int i=nchannels; i<(image.cols-1)*nchannels;
i++)
          {
// Застосувати оператор загострення
*output++= cv::saturate_cast<uchar>(
          5*current[i]-current[i-nchannels]-
          current[i+nchannels]-previous[i]-next[i]);
```

```
            }
        }

        // Встановити необроблені пікселі на 0
        result.row(0).setTo(cv::Scalar(0));
        result.row(result.rows-1).setTo(cv::Scalar(0));
        result.col(0).setTo(cv::Scalar(0));
        result.col(result.cols-1).setTo(cv::Scalar(0));
}
```

Зверніть увагу на те, що ця функція написана так, щоб вона працювала як на сірих, так і на кольорових зображеннях. Якщо ми застосуємо цю функцію на чорно-білій версії тестового зображення, отримаємо на виході покращення різкості.

Щоб отримати доступ до сусідніх пікселів попереднього і наступного рядка, потрібно просто визначити додаткові покажчики, які спільно збільшуються. Потім ми отримаємо доступ до пікселів цих рядків у циклі сканування.

У обчисленні вихідного піксельного значення на результат операції викликається функція шаблону `cv::saturate_cast`. Це пояснюється тим, що часто математичний вираз, застосований до пікселів, призводить до результату, який виходить за межі дозволених значень пікселів (тобто нижче `0` або більше `255`). Рішення полягає в тому, щоб повернути значення назад в цей 8-бітний діапазон. Це робиться шляхом заміни негативних значень на `0`, а значень, більших за `255`, на `255`. Саме це робить функція `cv::saturate_cast <uchar>`. Крім того, якщо вхідним аргументом є число з плаваючою точкою, то результат округляється до найближчого цілого числа. Очевидно, цю функцію можна використовувати з іншими типами, щоб гарантувати, що результат залишиться в межах, визначених цим типом.

Граничні пікселі, які не можуть бути оброблені, оскільки їхня околиця не повністю визначена, повинні оброблятися окремо. Тут ми просто встановлюємо їх на `0`. В інших випадках для цих пікселів можна виконати спеціальне обчислення, але тут немає сенсу витрачати час на обробку цих кількох пікселів. У нашій функції ці межі пікселів встановлюються рівними `0`, використовуючи два спеціальних метода: `row` і `col`. Вони повертають спеціальний екземпляр `cv::Mat`, який складається з однорядкового регіону інтересу (або ROI одного стовпця), як зазначено в параметрі. Тут немає жодної копії; тому, якщо елементи цієї 1D-матриці змінені, вони також будуть змінені у вихідному зображенні. Це ми робимо, коли викликається метод `setTo`. Цей метод присвоює значення всім елементам матриці, як зазначено нижче:

```
result.row(0).setTo(cv::Scalar(0));
```

Попередня операція присвоює значення `0` всім пікселям першого рядка результату зображення. У випадку 3-канального кольорового зображення треба використовувати `cv::Scalar (a, b, c)`, щоб вказати три значення, які будуть призначені кожному каналу пікселя.

Коли обчислення виконується за сусідством пікселя, загальноприйнятим є представлення його матрицею ядра. Це ядро описує, як пікселі, що беруть участь у обчисленні, об'єднуються, щоб отримати бажаний результат. Для фільтра загострення, використовуваного в цьому прикладі, ядро буде таким:

0	-1	0
-1	5	-1
0	-1	0

Якщо не вказано інше, поточний піксель відповідає центру ядра. Значення в кожній комірці ядра являє собою коефіцієнт, який множить відповідний піксель. Результат застосування ядра до пікселя задається сумою всіх цих множень. Розмір ядра відповідає розміру околиці (тут `3x3`).

Використовуючи це представлення, можна побачити, що, як того вимагає фільтр заточування, чотири горизонтальні і вертикальні сусіди поточного пікселя множаться на `-1`, тоді як поточний піксель множиться на `5`. Застосування ядра до зображення більше, ніж звичайне представлення: це основа концепції **згортки** в обробці сигналів. Ядро визначає фільтр, який застосовується до зображення.

Оскільки фільтрація є звичайною операцією в обробці зображень, в OpenCV визначено спеціальну функцію, яка виконує це завдання: `cv::filter2D`. Для цього потрібно просто визначити ядро (у вигляді матриці). Потім функція викликається з зображенням і ядром, і повертає відфільтровані зображення. Використовуючи дану функцію, легко перевизначити нашу функцію загострення наступним чином:

```
void sharpen2D(const cv::Mat &image,cv::Mat &result)
{
// Побудувати ядро (всі записи ініціалізуються до 0)
cv::Mat kernel(3,3,CV_32F,cv::Scalar(0));
    // Присвоїти значення ядра
    kernel.at<float>(1,1)= 5.0;
    kernel.at<float>(0,1)= -1.0;
    kernel.at<float>(2,1)= -1.0;
    kernel.at<float>(1,0)= -1.0;
    kernel.at<float>(1,2)= -1.0;
```

```
// Фільтруємо зображення
cv::filter2D(image,result,image.depth(),kernel);
}
```

Ця реалізація дає точно такий же результат, як і попередня (і з тією ж ефективністю). Якщо ви вводите кольорове зображення, то те саме ядро буде застосовано до всіх трьох каналів. Зауважимо, що особливо корисно використовувати функцію `filter2D` з великим ядром, оскільки вона використовує, в даному випадку, більш ефективний алгоритм.

2.8 Виконання простої арифметики з зображенням

Зображення можна комбінувати різними способами. Оскільки вони є регулярними матрицями, їх можна додавати, віднімати, множити або ділити. OpenCV пропонує різні арифметичні оператори зображення, і їх використання обговорюється в цьому прикладі.

Давайте працювати з другим зображенням, яке буде поєднуватися з нашим вхідним зображенням за допомогою арифметичного оператора. В якості другого зображення можна взяти чорно-білу картинку з монотонною текстурою, що нагадує косий дощ або з орнаментом.

Додавання двох зображень корисно, коли треба створити деякі спеціальні ефекти або накласти інформацію на зображення. Ми робимо це, викликавши функцію `cv::add`, точніше тут функцію `cv::addWeighted`, оскільки ми хочемо, щоб зважена сума була такою:

```
cv::addWeighted(image1,0.7,image2,0.9,0.,result);
```

Всі двійкові арифметичні функції працюють однаково. Наведено два входи, а третій параметр визначає вихід. У деяких випадках можуть бути задані значення, які використовуються як скалярні множники в операції. Кожна з цих функцій має кілька особливостей; `cv::add` є гарним прикладом функції, яка доступна в багатьох формах:

```
// c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC);
// c[i]= a[i]+k;
cv::add(imageA,cv::Scalar(k),resultC);
// c[i]= k1*a[i]+k2*b[i]+k3;
cv::addWeighted(imageA,k1,imageB,k2,k3,resultC);
// c[i]= k*a[i]+b[i];
cv::scaleAdd(imageA,k,imageB,resultC);
```

Для деяких функцій можна також вказати маску:

```
// if (mask[i]) c[i]= a[i]+b[i];
```

```
cv::add(imageA,imageB,resultC,mask);
```

Якщо застосувати маску, операція виконується тільки на пікселях, для яких значення маски не є нульовим (маска повинна бути 1-канальною). Погляньте на різні форми функцій `cv::subtract`, `cv::absdiff`, `cv::multiply` та `cv::divide`. Бітові оператори (оператори, застосовані до кожного окремого біта двійкового представлення пікселів) також доступні: `cv::bitwise_and`, `cv::bitwise_or`, `cv::bitwise_xor` і `cv::bitwise_not`. Оператори `cv::min` та `cv::max`, які знаходять максимальне або мінімальне значення пікселя за елементом, також дуже корисні.

У всіх випадках завжди використовується функція `cv::saturate_cast`, щоб переконатися, що результати залишаються в межах визначеного діапазону значення пікселя (тобто, щоб уникнути переповнення або перевантаження).

Зображення повинні мати однаковий розмір і тип (вихідне зображення буде перерозподілено, якщо воно не відповідає розміру вхідного). Також, оскільки операція виконується поелементно, одне з вхідних зображень може використовуватися як вихідне.

Також доступні кілька операторів, які приймають одне зображення в якості вхідних даних: `cv::sqrt`, `cv::pow`, `cv::abs`, `cv::cuberoot`, `cv::exp`, і `cv::log`. Фактично для будь-якої операції, яку потрібно застосувати до пікселів зображення, існує функція OpenCV.

Також можна використовувати звичайний арифметичний оператор C++ на екземплярах `cv::Mat` або на окремих каналах екземплярів `cv::Mat`.

2.8.1 Перевантажені оператори зображення

Дуже зручно, що більшість арифметичних функцій мають перевантажений відповідний оператор у OpenCV. Отже, виклик `cv::addWeighted` може бути записаний наступним чином:

```
result= 0.7*image1+0.9*image2;
```

Попередній код є більш компактною формою, яка також легше читається. Ці два способи написання зваженої суми є еквівалентними. Зокрема, функція `cv::saturate_cast` однаково буде викликатися в обох випадках.

Більшість операторів C++ були перевантажені. Серед них побітові оператори `&`, `|`, `^` і `~`; функції `min`, `max` та `abs`. Оператори порівняння `<`, `<=`, `==`, `!=`, `>` `I>` `=` також були перевантажені, і вони повертають 8-бітове двійкове зображення. Ви також знайдете множення матриці `m1*m2` (де `m1` і `m2` є і випадками `cv::Mat`), `m1.inv ()` інверсія матриці, `m1.t ()` транспонування, `m1.determinant ()` детермінант, векторна норма `v1.norm ()`, перехресний добуток `v1.cross (v2)`, точ-

ковий добуток `v1.dot (v2)` тощо. Якщо це має сенс, ви також маєте відповідний зіставний оператор присвоєння (оператор `+=`, як приклад).

У підрозділі, присвяченому *ефективному скануванню зображень*, ми створили функцію зменшення кількості кольорів, яка була написана з використанням циклів, що сканують пікселі зображення для виконання деяких арифметичних операцій над ними. З того, що ми дізналися тут, цю функцію можна переписати, просто використовуючи арифметичні оператори над вхідним зображенням наступним чином:

```
image=(image&cv::Scalar(mask,mask,mask))
            +cv::Scalar(div/2,div/2,div/2);
```

Використання `cv::Scalar` пояснюється тим, що ми маніпулюємо кольоровим зображенням. Використання операторів зображення робить код простим, а працю програміста продуктивною. Отже, ми маємо розглянути їх використання в більшості ситуацій.

2.9 Розділення каналів зображення

Іноді потрібно самостійно обробляти різні канали зображення. Наприклад, можна виконати операцію лише на одному каналі зображення. Звичайно, досягти цього можна в циклі сканування зображень. Тим не менш, також можна використати функцію `cv::split`, яка буде копіювати три канали кольорового зображення в три окремі екземпляри `cv::Mat`. Припустимо, ми хочемо додати зображення дощу лише до синього каналу. Нижче наводиться, як це зробити:

```
// Створюємо вектор з 3 зображень
     std::vector<cv::Mat> planes;
// Розділяємо 1 3-канальне зображення на 3 1-канальні
     cv::split(image1,planes);
// Додаємо до синього каналу
     planes[0]+= image2;
// Об'єднуємо 3 1-канальні зображення в 1 3-канальне
    cv::merge(planes,result);
```

Функція `cv::merge` виконує обернену операцію, тобто створює кольорове зображення з трьох 1-канальних зображень.

2.10 Перепризначення зображення

У прикладах цієї лабораторної роботи ми дізналися, як читати і змінювати піксельні значення зображення. Останній приклад навчить змінювати вигляд зображення, переміщуючи його пікселі. Значення пікселів не змінюються цим процесом; скоріше змінюється положення кожного пікселя, який перепризначається на нове місце. Це корисно для створення спеці-

альних ефектів на зображенні або для виправлення спотворень зображення, викликаних, наприклад, об'єктивом.

Для того, щоб скористатися функцією `remap` OpenCV, потрібно просто визначити карту, яка буде використовуватися в процесі перепризначення. По-друге, ми повинні застосувати цю карту до вхідного зображення. Очевидно, що саме спосіб визначення карти визначатиме ефект, який буде створено. У нашому прикладі ми визначимо функцію перетворення, яка створить хвилеподібний ефект на зображенні.

```
// Перепризначення зображення створенням ефекту хвилі
void wave(const cv::Mat &image, cv::Mat &result) {

// Функції карти
cv::Mat srcX(image.rows,image.cols,CV_32F);
cv::Mat srcY(image.rows,image.cols,CV_32F);

// Створення відображення
for (int i=0; i<image.rows; i++) {
     for (int j=0; j<image.cols; j++) {
// нове розташування пікселя в (i, j)
srcX.at<float>(i,j)= j; // залишаемося на тих самих
стовпцях пікселів, спочатку на рядку i
// тепер переміщуемося після синусоїди
srcY.at<float>(i,j)= i+5*sin(j/10.0);
          }
     }
// Застосування відображення
cv::remap(image,     // вихідне зображення
          result,    // зображення призначення
          srcX,      // x map
          srcY,      // y map
          cv::INTER_LINEAR);   // метод інтерполяції
}
```

Метою перепризначення є створення нової версії зображення, в якому пікселі змінили положення. Щоб побудувати це нове зображення, нам потрібно знати, яка вихідна позиція для кожного пікселя в зображенні призначення. Отже, необхідна функція відображення є такою, яка дасть нам початкові позиції пікселів як функцію нових позицій пікселя. Це називається **зворотним відображенням**, оскільки перетворення описує, як пікселі нових зображень відображаються назад до вихідного зображення. У OpenCV зворотне відображення описується з використанням двох карт: по одній для x-координат і для y-координат.

Обидві вони представлені екземплярами з плаваючою точкою `cv::Mat`:

```
// функції карти
cv::Mat srcX(image.rows,image.cols,CV_32F); // x-map
cv::Mat srcY(image.rows,image.cols,CV_32F); // y-map
```

Розмір цих матриць визначатиме розмір зображення призначення. Значення пікселя (`i, j`) кінцевого зображення може бути прочитане у вихідному зображенні з використанням наступного рядка коду:

```
( srcX.at<float>(i,j) , srcY.at<float>(i,j) )
```

Наприклад, простий ефект перегортання (поворот) зображень, подібний тому, який ми продемонстрували в першій лабораторній роботі, може бути створений наступними картами:

```
// Створення відображення
for (int i=0; i<image.rows; i++) {
    for (int j=0; j<image.cols; j++) {

// горизонтальне перегортання
srcX.at<float>(i,j)= image.cols-j-1;
srcY.at<float>(i,j)= i;
    }
}
```

Щоб сформувати отримане зображення, просто викличемо функцію OpenCV `remap`:

```
// Застосування відображення
cv::remap(image,              // вихідне зображення
          result,         // зображення призначення
          srcX,           // x map
          srcY,           // y map
          cv::INTER_LINEAR);   // метод інтерполяції
```

Цікаво відзначити, що дві карти містять значення з плаваючою комою. Отже, піксель у пункті призначення може відображатися назад до неінтегрального значення (тобто розташування між пікселями). Це дуже зручно, тому що дозволяє визначити функцію відображення нашого вибору. Наприклад, у нашому прикладі перепризначення ми використовували синусоїдальну функцію для визначення трансформації. Однак це також означає, що ми повинні інтерполювати значення віртуальних пікселів між реальними пікселями. Існують різні способи виконання піксельної інтерполяції, а останній параметр функції `remap` дозволяє вибрати метод, який буде використовуватися.

Піксельна інтерполяція є важливою концепцією в обробці зображень; цю тему буде розглянуто в лабораторній роботі *«Фільтрація зображень»*.

2.11 Контрольні запитання

1. Як отримати білий або чорний піксель на кольоровому зображенні?

2. Який порядок каналів кольорового зображення використовує за замовчуванням OpenCV?

3. Що означає наступний запис `cv::Vec2d`?

4. Для чого може знадобитися зменшення кількості кольорів у зображенні?

5. Що означає *ефективна ширина* рядка зображення?

6. Яким чином на ефективність обробки може вплинути доповненння рядка зображення кількома додатковими пікселями?

7. Як знайти кількість пікселів у кожному рядку зображення?

8. Як знайти загальну кількість байтів (у тому числі – додані пікселі) у рядку зображення?

9. Як в OpenCV отримати доступ до кожного окремого елемента зображення?

10. Що в OpenCV називається перетворенням на місці? Яка вигода від нього?

11. Яким чином в OpenCV можна створити ідентичну копію зображення?

12. Як в OpenCV виміряти час виконання функції або частини коду?

13. В чому полягають переваги, а в чому – застереження використання методу `at` для доступу до пікселів?

14. Що швидше виконується: більш короткий цикл з декількома операторами, чи більш довгий цикл з однією операцією, якщо загальна кількість оброблюваних елементів однакова?

15. Запишіть формулу згортки двох сигналів.

16. Що таке маска та як переконатися, що в результаті її застосування результати залишаються в межах визначеного діапазону значення пікселя?

2.12 Хід роботи

2.12.1 Підготовчі стадії

Поза навчальною лабораторією треба зробити наступне:

1. Користуючись підрозділами 2.1 – 2.10, а також рекомендованою літературою, ознайомитися із складом та особливостями функцій для маніпуляцій з пікселями в OpenCV.

2. Оформити першу частину звіту з виконання лабораторної роботи, в якій зазначити прізвище студента, мету лабораторних досліджень та дати письмові відповіді на контрольні запитання (п. 2.11).

3. Для виконання поставлених завдань необхідно більш детально розглянути принципи і основні етапи реалізації процедур для маніпуляцій з пікселями в OpenCV на сайті [2].

4. Складіть таблицю, в якій відобразіть оператори та функції OpenCV, які можна застосувати до пікселів зображення.

2.12.2 Завдання до роботи

1. Уважно вивчити матеріали попередніх підрозділів, виконуючи наведені приклади обробки зображень і використовуючи власні зображення.
2. Додати «солі» до тестового зображення.
3. Додати «перцю» до тестового зображення.
4. Перевірити процес сканування тестового зображення за допомогою покажчиків, виконавши зменшення кількості кольорів у кожному каналі у 64 рази. Виміряти час виконання процесу.
5. Перевірити процес сканування тестового зображення з ітераторами, виконавши зменшення кількості кольорів у кожному каналі у 64 рази. Виміряти час виконання процесу.
6. Перевірити процес сканування тестового зображення з використанням методу `at` для доступу до пікселів. Виміряти час виконання процесу.
7. Розроюити тест безперервності, в якому замість звичайного подвійного циклу над рядками та стовпцями створити один цикл для безперервного зображення. Виміряти час виконання процесу.
8. Перевірити процес сканування тестового зображення з сусіднім доступом, виконавши зменшення кількості кольорів у кожному каналі у 64 рази. Виміряти час виконання процесу.
9. Покращити різкість чорно-білої версії тестового зображення щонайменше двома способами та порівняти результат.
10. Поєднайте два зображення за допомогою арифметичного оператора.
11. Додайте «дощ» до синього каналу тестового зображення.
12. Створіть хвилеподібний ефект на зображенні.

2.13 Вимоги до звіту по роботі

Звіт про виконання роботи повинен містити:

1. Прізвище та групу студента, а також назву, мету та завдання лабораторної роботи.
2. Відповіді на контрольні питання.
3. Таблицю з переліком функцій OpenCV для маніпуляцій з пікселями.
4. Короткий перелік основних дій в процесі виконання завдань.
5. Текст розроблених програм з коментарями (проект у zip-архіві в якості додатку до звіту).
6. Скрін-шоти роботи розроблених програм.
7. Висновки, в яких треба зазначити, чи досягнута мета даної лабораторної роботи. Які саме конкретно маніпуляції з пікселями дозволяє (перерахувати по пунктах) робити OpenCV? Які конкретно практичні навички

роботи з зображеннями здобуті в ході дослідження?

8. Електронну версію звіту з лабораторної роботи у форматі 2020_Прізвище-ЛР2.doc або *.docx разом з додатками (програмними проектами в архіві) треба завантажити у відповідному розділі сторінки дисципліни в системі дистанційного навчання “Moodle”.

2.14 Орієнтовні варіанти завдань

Завдання за варіантами видаються викладачем після виконання студентом підготовчої стадії (п. 2.12.1), типових завдань (п. 2.12.2), здавання звіту та його перевірки.

3 Лабораторна робота №3. Обробка кольорів зображення

Мета роботи: практично дослідити порівняння кольорів з використанням шаблону дизайну стратегії, сегментацію зображення за допомогою алгоритму GrabCut, перетворення представлення кольорів і представлення кольорів за допомогою відтінку, насиченості та яскравості.

3.1 Загальні теоретичні відомості про обробку кольорів

Можливість бачити світ кольоровим – це одна з важливих характеристик людського зору. Сітківка ока містить в собі спеціалізовані фоторецептори, які називають *колбочками*, і які відповідають за сприйняття кольорів. Існує три типи колбочок, які відрізняються діапазоном довжин хвиль, що вони поглинають. Використовуючи сигнали від різних клітин, людський мозок здатен створити сприйняття кольорів. Більшість тварин мають лише такі клітини, як палички. Вони представляють собою фоторецептори з кращою чутливістю до світла, і охоплюють повний спектр видимого світла без розрізнення кольорів. В людському оці палички розташовані в основному на периферії сітківки, тоді як колбочки сконцентровані в центральній частині.

В цифровому зображенні кольори зазвичай відтворюються з використанням червоного, зеленого та синього адитивних основних кольорів. Вони були обрані через те, що при поєднанні можуть утворювати широку гаму різних кольорів. Насправді цей набір основних кольорів добре імітує трихроматичне кольорове сприйняття людської системи зору, оскільки різні колбочки мають чутливість в районі червоного, зеленого та синього спектру. В цій лабораторній роботі ми поекспериментуємо з кольором пікселя і побачимо, як може бути сегментоване зображення, спираючись на інформацію про колір. Додатково ми вивчимо, що використання різних представлень кольру під час виконання обробки зображення іноді може бути корисним.

3.2 Порівняння кольорів з використанням шаблону дизайну стратегії

Давайте припустимо, що нам потрібно скласти простий алгоритм, який визначає всі пікселі зображення, що мають заданий колір. Для цього алгоритм має приймати зображення та колір, як вхідні дані, і повертати бінарне зображення, яке показує пікселі вказаного кольору. Точність, з якою ми хочемо приймати колір, буде ще одним параметром, який треба задати перед виконанням алгоритму.

Задля досягення цієї цілі в лабораторній роботі буде використовуватися **шаблон дизайну стратегії**. Цей об’єктно-орієнтований шаблон дизайну являє собою чудовий шлях інкапсулювання алгоритму в класі. Тоді надалі легшою стає заміна даного алгоритму іншим, або поєднання кількох алгоритмів один з одним задля створення більш складних процесів. Додатково цей шаблон полегшує розгортку алгоритму, ховаючи якомога більшу частину його складності за інтуітивним програмним інтерфейсом.

3.2.1 Методика використання

Як тільки алгоритм інкапсульований в клас з використанням шаблону дизайну стратегії, він може бути розгорнутий за допомогою екземпляру цього класу. Зазвичай екземпляр буде створено при ініціалізації програми. Під час побудови екземпляр класу буде ініціалізувати різні параметри алгоритму з їх значенням за замовчуванням, так що він одразу буде готовий до використання. Значення параметрів алгоритму також можуть бути прочитані і встановлені з використанням відповідних методів. У випадку додатку з графічним інтерфейсом ці параметри можуть бути відображені і змінені з використанням різних віджетів (текстових полів, повзунків і так далі), так що користувач може легко експериментувати з ними.

Ми покажемо структуру класу `strategy` в наступному підрозділі. Почнемо з прикладу того, як він може бути розгорнутий і використаний. Давайте напишемо просту функцію `main`, яка буде запускати наш алгоритм визначення запропонованого кольору:

```
int main()
{
      //1. Створити об'єкт-обробник зображення
      ColorDetector cdetect;

      //2. Прочитати вхідне зображення
      cv::Mat image= cv::imread("boldt.jpg");
      if (image.empty()) return 0;

      //3. Задати вхідні параметри
      cdetect.setTargetColor(230,190,130); //синє небо

      //4. Обробити зображення і вивести результат
      cv::namedWindow("result");
      cv::Mat result = cdetect.process(image);
      cv:imshow("result",result);

      cv::waitKey();
      return 0;
}
```

Виконання цієї програми для виявлення синього неба на кольоровій версії зображення палацу створить наступний вивід (рисунок 3.1):

Рисунок 3.1 – Виявлення синього неба на кольоровому зображенні

Тут білі пікселі позначають позитивний результат виявлення задуманого кольору, білі – негативний.

Очевидно, алгоритм, який ми інкапсулювали в цьому класі, відносно простий (як ми побачимо далі, він складається всього з одного циклу сканування одного параметру допустимого відхилення). Шаблон дизайну стратегії стає справді потужним, коли реалізований більш складний алгоритм, який має багато кроків і включає декілька параметрів.

3.2.2 Опис методики

Центральний процес цього алгоритму простий для побудови. Це простий цикл сканування, яки1 проходить через кожен піксель, порівнюючи його колір з цільовим кольором. Використовуючи те, що ми вивчили в підрозділі сканування зображення за допомогою ітераторів попереднього розділу, цей цикл може бути написаний наступним чином:

```
//отримати ітератори
cv::Mat_<cv::Vec3b>::const_iterator it= im-
age.begin<cv::Vec3b>();
cv::Mat_<cv::Vec3b>::const_iterator itend= im-
age.begin<cv::Vec3b>();
cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

//для кожного пікселя
for(;it!= itend;++it,++itout){

      //розрахувати відстань від цільового кольору
      if(getDistanceToTargetColor(*it)<=maxDist){
```

```
      *itout= 255;
      } else {
      *itout= 0;
      }
}
```

Змінна зображення cv::Mat стосується вхідного зображення, тоді як result стосується двійкового вихідного зображення. Отже, перший крок складається з налаштування потрібних ітераторів. Тоді цикл сканування стає простим для реалізації. Зверніть увагу, що ітератори вхідного зображення оголошені, як const, оскільки значення їх елементів не змінні. Відстань між поточним кольором пікселя та цільовим кольором оцінюється для кожного пікселя, щоб перевірити, чи знаходится він в рамках допустимого відхилення параметру, як це визначено у maxDist. Якщо це – саме той випадок, вихідному зображенню присвоюється значення 255 (білий); якщо ні, присвоюється значення 0 (чорний). Для обчислення відстані до цільового кольору, використовується метод getDistanceToTargetColor. Існують різні шляхи для обчислення цієї відстані.

Один може, наприклад, обчислювати евклідову відстань між трьома векторами, які містять RGB значення кольорів. Щоб залишати ці обчислення простими, ми складаємо абсолютну різницю значень RGB (це також відомо, як відстань міського кварталу). Зверніть увагу, що в сучасній архітектурі евклідова відстань з плаваючою точкою може бути обчислена швидше, ніж звичайна відстань міського кварталу (додатково ви також можете використовувати квадрати евклідових відстаней для уникнення затратних квадратних коренів); це також іноді береться до уваги у нашому дизайні. Також, для більшої гнучкості, ми написали метод getDistanceToTargetColor, спираючись на метод getDistanceToColor, наступним чином:

```
//Обчислює відстань від цільового кольору
int getDistanceToTargetColor(const cv::Vec3b& color) const
{
return getColorDistance(color, target);
}
//обчислює відстань міського кварталу для двох кольорів
int getColorDistance(const cv::Vec3b& color1, const cv::Vec3b&
color2)const {
      return abs(color1[0]-color2[0]) +
             abs(color1[1]-color2[1]) +
             abs(color1[2]-color2[2]);
}
```

Зверніть увагу, як ми використали cv::Vec3d для того, щоб зберігати три беззнакові символи, які представляють RGB-значення кольору. Змінна target, очевидно, посилається на заданий цільовий колір і, як ми далі побачимо, визначається, як змінна члену в алгоритмі класу, який ми задаємо.

Тепер давайте довершимо визначення методу обробки. Користувачі будуть надавати вхідне зображення і, як тільки сканування завершиться, результатом буде повернене зображення.

```
Cv::Mat ColorDetector::process(const cv::Mat &image) {

      //Перерозподіл бінарної мапи у разі необхідності
      //той-же розмір, що у вхідного зображення, але один
      //канал
      result.create(image.size(),CV_8U);

      //обробка верхнього циклу проходить тут
      return result;
}
```

Щоразу, коли цей метод викликається, важливо перевіряти, чи вихідне зображення, яке містить результуючу бінарну мапу, потребує перерозподілу, щоб підходити під розміри вхідного зображення. Ось чому ми використовуємо метод create з cv::Mat. Запам'ятайте, що цей метод застосовний лише для перерозподілу, якщо задані розмір та/або глибина не відповідають теперішній структурі зображення.

Тепер, коли ми маємо визначеними основні методи обробки, давайте подивимося, які додаткові методи повинні бути додані для розгортання цього алгоритму. Ми заздалегіть визначили, які вхідні і вихідні дані вимагає наш алгоритм. Надалі ми визначимо атрибути класу, які будуть містити ці дані:

```
class ColorDetector {
      private :

      //мінімальна прийнятна відстань
      int maxDist;
      //цільовий колір
      cv:Vec3b target;

      //зображення, що містить результуючу двійкову мапу
      cv::Mat result
```

Для того, щоб створити екземпляр класу, що інкапсулює наш алгоритм (який ми назвали ColorDetector), нам необхідно визначити конструктор. Пам'ятаймо, що одна з цілей шаблону дизайну стратегії – це робити розгортку алгоритму якомога простішою. Найпростіший конструктор, який можна задати, – порожній. Він створить екземпляр алгоритму класу в належному стані. Тоді ми хочемо, щоб конструктор ініціалізував всі вхідні параметри до їх значень за замовчуванням (або значення, про які відомо, що вони зазвичай дають гарний результат). В нашому випадку ми вирішимо, що відстань 100 загалом прийнятне значення параметру допустимого відхилення. Ми також встановимо цільовий колір за замовчуванням. Ми

обрали чорний без якоїсь вагомої причини. Ідея полягає в тому, щоб переконатися, що ми завжди починаємо з передбачуваних і задовільних вхідних значень:

```
//порожній конструктор
//тут ініціалізуються параметри за замовчуванням
ColorDetector() : maxDist(100), target(0,0,0) {}
```

Іншим варіантом могло б бути не створювати порожній конструктор, а змусити користувача ввести цільовий колір і відстань від кольору в більш складному конструкторі :

```
//інший конструктор з ціллю і відстанню
ColorDetector(uchar blue, uchar green, uchar red, int mxDist)
```

На данному етапі користувач, який створює екземпляр класу нашого алгоритму, може негайно викликати метод процессу з задовільним зображенням і отримати задовільний результат. Це – ще одна ціль шаблону стратегії: переконатися, що алгоритм завжди запускається з задовільними параметрами. Очевидно, що користувач цього класу захоче використати свої власні налаштування. Це здійснюється наданням користувачу належних геттерів та сетерів. Давайте почнемо з параметру допустимого відхилення `color`:

```
//задає поріг відстані кольору
//поріг має бути додатнім,
//інакше поріг відстані задається як 0
void setColorDistanceThreshold(int distance) {

      if (distance<0)
      distance=0;
      maxDist= distance;
      }

      // отримує поріг відстані від кольору
      int getColorDistanceThreshold() const {
      return maxDist;
      }
```

Зверніть увагу, що ми спочатку перевіряємо вірність вхідних даних. Знову ж-таки, це робиться для того, щоб переконатися, що наш алгоритм ніколи не запуститься у незадовільному стані. Цільовий колір може бути заданний схожим способом наступним чином:

```
// Задає кольору бути визначеним
void setTargetColor(uchar blue,
                    uchar green,
                    uchar red) {
```

```
      // порядок BGR
      target = cv::Vec3b(blue, green, red);
}
// задає кольору бути визначеним
void setTargetColor(cv::Vec3b color) {
      target= color;
}
// задає кольору бути визначеним
cv::Vec3b getTargetColor() const {
      return target;
}
```

Цього разу цікаво зауважити, що ми надали користувачу два визначення методу setTargetColor. У першій версії визначення три компоненти кольору вказані як три аргументи, тоді як у другій версії cv:Vec3b використовується для зберігання значень кольорів. Знову ж таки, ціль — полегшити використання алгоритму нашого класу. Користувач може просто вибрати сетер, що найкраще задовольняє його потреби.

3.2.3 Розвиток алгоритму

Алгоритм-приклад, який використовується в цьому підрозділі, складається з визначення пікселів зображення, які мають колір, достатньо близький до заданого цільового кольору. Ці обчислення можуть бути виконані іншим чином. Що цікаво, функція OpenCV виконує те ж завдання для того, щоб виділити пов’язаний компонент заданого кольору. Також реалізація шаблону дизайну стратегії може бути доповнена з використанням об’єктів функції. Нарешті, OpenCV визначає базовий клас cv::Algorithm, який реалізує концепції шаблону дизайну стратегії.

3.2.3.1Розрахунок відстані між двома векторами кольорів

Щоб розрахувати відстань між двома векторами кольорів, ми використовуємо наступну просту формулу:

```
return abs(color[0]-target[0])+
       abs(color[1]-target[1])+
       abs(color[2]-target[2]);
```

Однак OpenCV включає функцію для розрахування евклідової норми вектора. Отже, ми могли обчислити нашу відстань наступним чином:

```
return static_cast<int>(
       cv::norm<int,3>(cv::Vec3i(color[0]-target[0],
                                 color[1]-target[1],
                                 color[2]-target[2])));
```

Тоді, з використанням цього визначення методу getDistance, було б отримано дуже схожий результат. Тут ми використовуємо cv::Vec3i (трьохвекторний масив цілих чисел) тому, що результат віднімання – це цілочисельне значення.

Також цікаво пригадати з другого розділу, Маніпуляція пікселями, що матриця OpenCV та векторна структура даних включає визначення базових арифметичних операторів. Тож можна запропонувати наступне визначення обчислення відстані:

```
return static_cast<int>( cv::norm<uchar,3>(color- target));//
не правильно!
```

Це визначення може виглядати правильним на перший погляд, однак воно хибне. Тому, що всі ці оператори завжди включають виклик saturate_cast (дивіться підрозділ «Сканування зображення з доступом до сусіда» в попередній лабораторній роботі), щоб переконатися, що результат лишається у рамках вхідного типу (тут це `uchar`). Тож у випадках, коли цільове значення більше за відповідне значення кольору, значення 0 буде прикріплене замість від'ємного значення, яке можна було очікувати. Коректним формулюванням буде наступне:

```
cv::Vec3b dist;
cv::absdiff(color,target,dist);
return cv::sum(dist)[0];
```

Однак використання двох викликів функції для розрахунку відстані між двома тривекторними масивами не ефективне.

3.2.3.2Використання функцій OpenCV

У цьому підрозділі ми використали цикл з ітераторами для виконання нашого обчислення. Альтернативно, ми могли б досягти того самого результату, викликаючи послідовність функцій OpenCV. Спосіб виявлення кольору буде записано наступним чином:

```
cv::Mat ColorDetector::process(const cv::Mat
&image) {
    cv::Mat output;
    // compute absolute difference with target
    color
    cv::absdiff(image,cv::Scalar(target),output);

    // split the channels into 3 images
    std::vector<cv::Mat> images;
    cv::split(output,images);
```

```
        // add the 3 channels (saturation might occurs
        here)
        output= images[0]+images[1]+images[2];
        // застосування порогового значення
        cv::threshold(output,      // те ж саме
        input/output image
            output,
            maxDist,                // порогове
                                       значення
                                       (має бути менше
                                       за 256)
            255,                    // максимальне
                                       значення
            cv::THRESH_BINARY_INV); // режим порогового
                                       значення

        return output;
    }
```

Цей метод використовує функцію `absdiff`, яка обчислює абсолютну різницю між пікселями зображення і, в даному випадку, – скалярним значенням. Замість скалярного значення може бути надане інше зображення у якості другого аргументу для цієї функції. У останньому випадку буде застосована попіксельна різниця. Як наслідок, два зображення мають бути одного розміру. Тоді ндивідуальні канали зображення різниці видобуваються з використанням функції `split` (обговорюється у пункті “модифікація методу” підрозділу “виконання простої арифметики зображень” лабораторної роботи 2, “маніпуляція пікселями”) для того, щоб була можливість скласти їх разом. Важливо звернути увагу, що результат цієї суми іноді може бути більшим за 255, та, оскільки завжди застосовується насиченість, результат зупиниться на 255. Наслідком буде те, що в такому варіанті параметр `maxDist` також має бути меншим за 256. Це має бути виправленим, якщо ви вважаєте таку поведінку неприйнятною.

Останній крок – це створення двійкового зображення, використовуючи функцію `cv::threshold`. Ця функція звичайно використовується для порівняння всіх пікселів з пороговим значенням (третій параметр), і в нормальному пороговому режимі (`cv::THRESH_BINARY`) вона прикріпляє задане максимальне значення (четвертий параметр) до всіх пікселів, більших заданого порогу, і 0 – до інших пікселів. Тут ми використовуємо зворотний режим (`cv::THRESH_BINARY_INV`), в якому максимальне задане значення прикріплене до пікселів, які мають значення нижче, або рівне пороговому. Цікавими також є режими `cv::THRESH_TOZERO` та

cv::THRESH_TOZERO_INV, які залишають пікселі більші, або менші порогового значення незмінними.

Використання функцій OpenCV загалом гарна ідея. Тоді ви можете швидко будувати складні застосування і потенціально зменшити кількість багів. Результат часто більш ефективний (завдячуючи зусиллям по оптимізації, вкладеним співучасниками розробки OpenCV). Однак, коли виконується багато проміжних кроків, можна виявити, що наслідковий метод споживає більше пам'яті.

3.2.3.3Функція floodFill

Наш клас ColorDetector визначає пікселі в зображенні, які мають колір, подібний даному цільвому кольору. Рішення прийняти піксель чи ні приймається на попіксельній основі. Функція cv::floodFill відбувається у дуже схожим шляхом з однією важливою різницею: в цьому випадку рішення про прийняття пікселя також залежить від його сусідів. Ідея в тому, щоб визначити пов'язану область певного кольору. Користувач задає місцезнаходження початкового пікселя і допустиме відхилення параметрів, що визначають подібність кольору.

Насіннєвий піксель задає шуканий колір і з цього початкового місцезнаходження розглядаються сусіди для того, щоб визначити пікселі схожого кольору. Потім сусіди прийнятих сусідів також розглядаються і так далі. Таким чином з зображення буде видобуто область сталого кольору. Наприклад, щоб виявити на нашому зображенні-прикладі область синього неба, можна діяти наступним чином:

```
cv::floodFill(image,          // вхідне/вихідне
                                 зображення
cv::Point(100, 50),           // початкова точка
cv::Scalar(255, 255, 255),    // перефарбований колір
(cv::Rect*)0,                 // обмеження області
                                 Перефарбованого набору
cv::Scalar(35, 35, 35),       // верхній/нижній поріг
                                 різниці
cv::Scalar(35, 35, 35),       // однакові у більшості
                                 випадків
cv::FLOODFILL_FIXED_RANGE); // пікселі у порівнянні з
                                 початковим
```

Насіннєвий піксель (100,50) розташований на небі. Всі пов'язані пікселі будуть перевірені і ті, що мають подібний колір, будуть перефарбовані у новий колір, визначений третім параметром. Щоб визначити, подібний колір чи ні, різні порогові значення задаються незалежно для значень, вищих чи нижчих за еталонний колір. Тут ми використовуємо режим фіксованого діапазону, який має на увазі, що перевірені пікселі всі будуть порівняні з кольором насіннєвого пікселя. Режим за замовчуванням – це

такий, в якому кожен перевірений піксель порівнюється з кольором його сусідів. Отриманий наступний результат (рисунок 3.2):

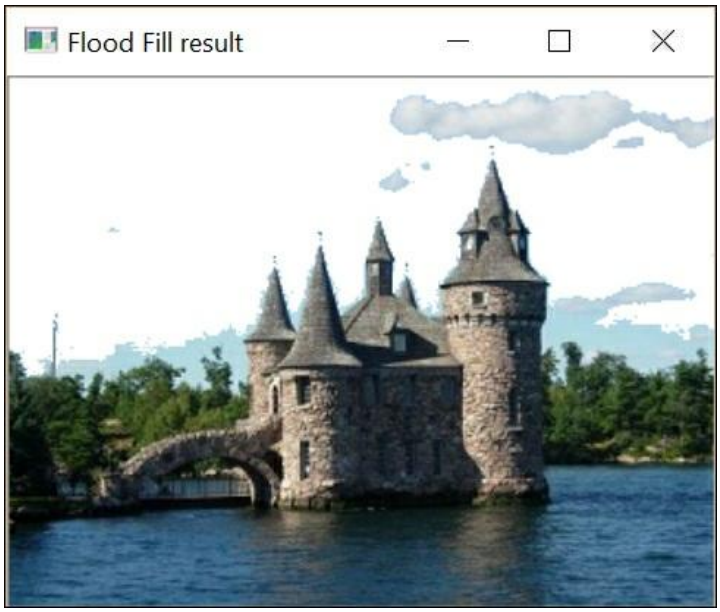


Рисунок 3.2 – результат виконання функції flood fill

Єдина поєднана область перефарбована алгоритмом (тут ми пофарбували небо у білий). Внаслідок, навіть, якщо ще десь є пікселі з подібним кольором (наприклад у воді), вони не будуть розпізнані, якщо вони не приєднані до області неба.

3.2.3.4Функтор, або функціональний об’єкт

Використання перевантаження операторів C++ призведе дотого, що екземпляри будуть поводити себе, як функції. Ідея полягає в тому, щоб перевантажити метод `operator()` так, що виклик методу обробки класу виглядає точно, як звичайний виклик функції. Наслідковий екземпляр класу називається *функціональним об’єктом*, або *функтором*. Часто функтор включає повний конструктор. Таким чином він може бути використаний негайно після створення. Наприклад, ви можете визначити повний конструктор нашого класу `ColorDetector` наступним чином :

```
// повний конструктор
ColorDetector(uchar blue, uchar green, uchar red, int
maxDist=100):
              maxDist(maxDist) {
      // цільовий колір
      setTargetColor(blue, green, red);
}
```

Очевидно, що ви все ще можете використовувати сетери і геттери, що були визначені раніше. Метод-функтор може бути визначений наступним чином :

```
cv::Mat operator()(const cv::Mat &image) {
// тут код, що визначає колір
}
```

Щоб виявити даний колір цим методом-функтором, просто напишіть наступний уривок коду :

```
ColorDetector colordetector(230,190,130,// колір
                            100);       // поріг
cv::Mat result= colordetector(image);   // виклик функтору
```

Як ви можете бачити, виклик методу виявлення кольору тепер виглядає, як виклик функції.

3.2.3.5Базовий клас OpenCV для алгоритмів

OpenCV пропонує багато алгоритмів, що виконують різноманітні завдання комп'ютерного зору. Для полегшення їх використання, більшість цих алгоритмів утворили підклас звичайного базового класу, що має назву `cv:Algorithm`. Він реалізує деякі концепції, присвячені шаблону дизайну стратегії. По перше, всі ці алгоритми створені динамічно з використанням спеціалізованих статичних методів, які переконуються, що алгоритм завжди створюється в задовільному стані (це значить з задовільними значеннями за замовчуванням для незаданих параметрів). Давайте розглянемо, наприклад, один з цих підкласів, cv:ORB; Цей оператор буде точкою інтересу, яка буде обговорена в у лабораторній роботі «Виявлення точок інтересу». Тут ми просто використаємо його, як наглядний приклад алгоритму.

Тому екземпляр алгоритму створено наступним чином :

```
cv::Ptr<cv::ORB> ptrORB = cv::ORB::create();//стан за
                                              замовчуванням
```

Після створення алгоритм може бути використаним. Наприклад, загальні методи `read` та `write` можуть бути використані для завантаження або збереження стану алгоритму. Алгоритм також має спеціалізовані методи (у випадку `ORB`, наприклад, методи `detect` та `compute` можуть бути використані для запуску їх основних обчислювальних підрозділів). Алгоритм також має спеціалізовані методи-сеттери, що дозволяють задавати їх внутрішні параметри. Зверніть увагу, що ми можемо оголосити вказівник як `cv::Ptr<cv::Algorithm>`, але в цьому випадку ми не матимемо можливості використовувати його спеціалізовані методи.

3.3 Сегментація зображення за допомогою алгоритму GrabCut

Попередній підрозділ показав, як інформація про колір може бути корисною для сегментації зображення на зони, що відповідають окремим елементам композиції. Об'єкти часто мають різні кольори і можуть бути виявлені розпізнаванням зон одного кольору. OpenCV пропонує реалізацію популярного алгоритму для сегментації зображення: алгоритм `GrabCut`. `GrabCut` – це складний і ресурсозатратний алгоритм, але він зазвичай дає дуже точні результати. Це найкращий алгоритм для виділення об'єкта на передньому плані статичного зображення (наприклад, щоб вирізати і вставити об'єкт з одного зображення до іншого).

3.3.1 Методика використання

Функція `cv:grabCut` проста у використанні. Вам лише потрібно ввести зображення і відмітити деякі його пікселі як ті, що належать до переднього або заднього плану. Спираючись на ці часткові відмітки, алгоритм розрізнить розподіл на передній і задній план для всього зображення.

Один з методів задати часткові відмітки для переднього і заднього плану – визначивши прямокутник, всередині якого знаходиться об'єкт переднього плану:

```
//визначити обмежувальний прямокутник
//пікселі за межами цього прямокутника
//будуть помічені як задній план
cv::Rect rectangle(5,70,260,120);
```

Це визначає наступну зону на зображенні (рисунок 3.3)

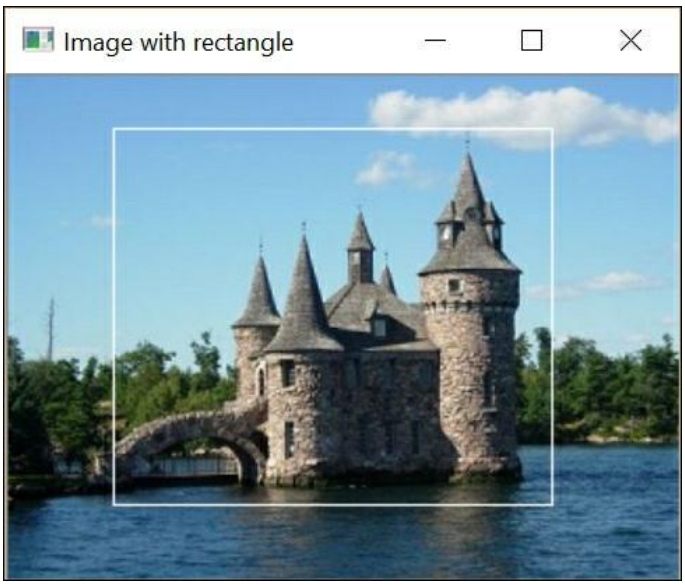


Рисунок 3.3 – Обмеження зони зображення прямокутником

Всі пікселі за межами цього прямокутника будуть відмічені як задній план.

Додатково до вхідного зображення і малюнку його сегментаціїї, виклик функції cv::grabCut вимагає визначення двох матриць, які міститимуть модель побудовану алгоритмом наступним чином :

```
cv::Mat result;
// segmentation (4
possible values)
cv::Mat bgModel,fgModel;
// the models
(internally used)
// GrabCut segmentation
cv::grabCut(image,                  // вхідне зображення
            result,                 // результат сегментації
            Rectangle,              // прямокутник, що
                                       містить передній план
            bgModel,fgModel,        // моделі
            5,                      // кількість ітерацій
            cv::GC_INIT_WITH_RECT);// використовувати
                                       прямокутник
```

Зауважте, як ми задали, що використовуємо режим обмежувального прямокутника прапором cv::GC_INIT_WITH_RECT у якості останнього аргументуфункції (у наступній секції, “як це працює”, буде обговорено інший доступний режим). Вхідне/вихідне зображення сегментації може мати одне з наступних значень :

- cv::GC_BGD: це значення пікселів, які точно належать до заднього плану(наприклад пікселі за межами прямокутника у нашому випадку)
- cv::GC_FGD: це значення пікселів, які точно належать до переднього плану(таких нема у нашому прикладі)
- cv::GC_PR_BGD : це значення пікселів, які вірогідно належать до заднього плану
- cv::GC_PR_FGD : це значення пікселів, які вірогідно належать до переднього плану (початкові значення пікселів у межах прямокутника у нашому прикладі)

Ми отримали двійкове зображення сегментації витягнувши пікселі що мали значення рівне cv::GC_PR_FGD. Це досягається за допомогою наступного коду:// Отримати пікселі що помічені як вірогідний передній планcv::compare(result,cv::GC_PR_FGD,result,cv::CMP_EQ);// Сформувати вихідне зображеннясv::Matforeground(image.size(),CV_8UC3,cv::Scalar(255,255,255));image.copyTo(foreground,// пікселі задньогог плану не копіюютьсяresult);Щоб витягнути всі пікселі переднього плану, тобто зі значеннями рівними

cv::GC_PR_FGD або cv::GC_FGD, можливо наступним чином перевірити значення першого біту:// перевірка першого біту побітовим iresult= result&1; // буде 1, якщо передній планЦе можливо тому що ці константи визначені як значення 1 і 3, тоді як інші дві (cv::GC_BGD та cv::GC_PR_BGD) визначені як 0 і 2. У нашому прикладі отримано той самий результат тому що зображення сегментації не містить у собі пікселів cv::GC_FGD (були введені лише пікселі cv::GC_BGD).Буде отримано наступне зображення(рисунок 3.4):

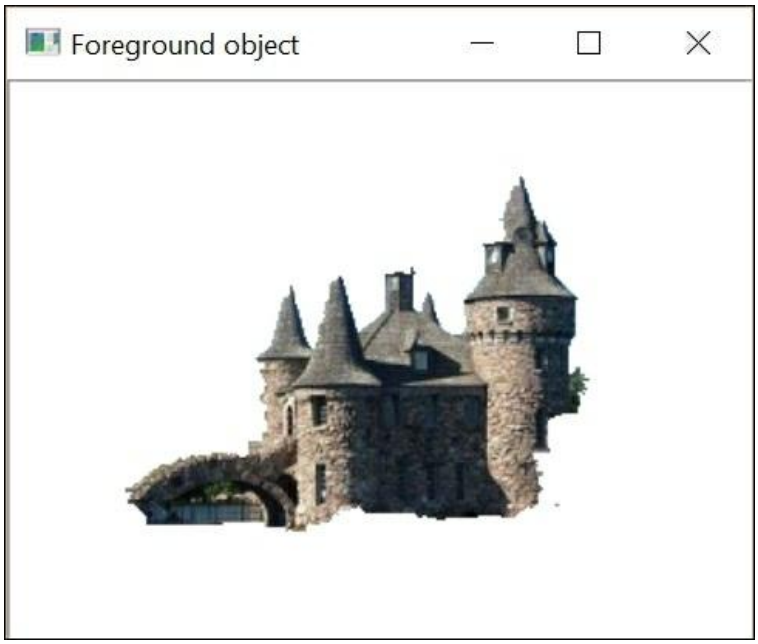


Рисунок 3.4 - виділення об’єкту переднього план

3.3.2 Опис методики

У попередньому прикладі алгоритм GrabCut був здатний виділити об’єкти переднього плану простим заданням прямокутника, всередині якого містився об’єкт (замок). Інакше можна прикріпити значення cv::GC_BGD та cv::GC_FGD до деяких конкретних пікселів вхідного зображення, які надаються використанням зображення-маски як другого аргументу функції cv::grabCut. Потім ви задасте GC_INIT_WITH_MASK як прапор режиму вводу. Ці вхідні мітки можуть бути отримані, наприклад, проханням до користувача інтерактивно відмітити декілька елементів зображення. Також можливо поєднати ці два методи вводу.

Використовуючи цю вхідну інформацію, алгоритм GrabCut створює розподілення на передній і задній план діючи наступним чином. Спочатку мітка переднього плану (cv::GC_PR_FGD) навмання присвоюється всім невідміченим пікселям. Спираючись на поточну класифікацію, алгоритм групує пікселі у кластери однакового кольору (так, що виходить k кластерів для переднього плану і k кластерів для заднього плану). Нвступний крок —

визначити розподілення на передній і задній план введенням межі між пікселями переднього і заднього плану.

Це досягається процессом оптимізації, що намагається з'єднати пікселі з однаковими мітками і що накладає штрафи за розміщення межі в областях відносно рівномірної насиченості. Ця проблема оптимізації може бути ефективно вирішена використанням алгоритму Graph Cuts, методу, що може знаходити оптимальне рішення проблеми представляючи її зв'язним графом на якому задля складання оптимальної конфігурації застосовуються розрізи. Отримана сегментація забезпечує нові мітки для пікселів.

Процес розбиття на кластери потім може бути повтоений і може бути знайдена нова оптимальна сегментація і так далі. Тому алгоритм GrabCut — це ітераційна процедура, яка поступово вдосконалює результат сегментації. Залежно від складності сцени, гарне рішення може бути знайдене за більшу чи меншу кількість ітерацій (в простих випадках буде достатньо однієї ітерації).

Це пояснює аргументи функції, де користувач може вказати кількість застосованих ітерацій. Дві внутрішні моделі, що зберігаються алгоритмом, передаються як аргументи функції (і повертаються). В наслідок цього можливо викликати функцію з моделлю останнього запуску ще раз якщо є бажання вдосконалити результат сегментації виконанням додаткових ітерацій.

3.4 Перенесення представлення кольору

Кольоровий простір RGB заснований на використанні червоного, зеленого та синього адитивних основних кольорів. В першому підрозділі цього розділу ми бачили, що ці основні кольори були обрані через те, що вони можуть давати гарний спектр кольорів, що добре узгоджуються з принципами зору людини. Це часто колірний простір за замовчуванням в цифрових зображеннях, тому щу це спосіб яким отримуються кольорові зображення, тобто з використанням червоного, зеленого і синього фільтрів. Додатково, червоний, зелений і синій канали нормалізовані так, що змішані в рівній кількості, результують в сірий колір різного рівня насиченості, від чорного(0,0,0) до білого (255,255,255).

Нажаль розрахунок відстані між кольорами, використовуючи колірний простір RGB – це не найкращій спосіб вимірювання схожості двох даних кольорів. І справді, RGB — простір кольорів, що не однорідний для сприйняття. Це означає, що два кольори на заданій відстані можуть виглядати досить схоже, тоді як два інші кольори, розділені тією самою відстанню, можуть виглядати дуже по різному.

Для вирішення цієї проблеми були введені інші представлення кольорів, що мають властивість однорідності для сприйняття. Зокрема, CIE L*a*b* - одна з таких кольорових моделей. Після перенесення зображення в це представлення, Евклідова відстань між пікселем і цільовим кольором буде змістовною мірою візуальної схожості між двома кольорами. В цьому

підрозділі ми покажемо вам, як переносити кольори з одного представлення в інше щоб працювати з іншими колірними просторами.

3.4.1 Методика використання

Перетворення зображень між різними колірними просторами легко здійснюється за допомогою функції cv :: cvtColor OpenCV. Давайте повернемося до класу ColorDetector першого підрозділу цього розділу, Порівняння кольорів за допомогою шаблону дизайну Стратегії. Ми переносимо вхідне зображення до колірного простору CIE L * a * b * на початку методу процесу:

```
cv::Mat ColorDetector::process(const cv::Mat &image) {
// переназначимо двійкову мапу у разі необхідності
// той же розмір, що і вхідне зображення, але один канал
result.create(image.rows,image.cols,CV_8U);

// Перенесення в колірний простір Lab
cv::cvtColor(image, converted, CV_BGR2Lab);

// отримуємо ітератори перенесеного зображення
cv::Mat_<cv::Vec3b>::iterator it=
converted.begin<cv::Vec3b>();
cv::Mat_<cv::Vec3b>::iterator itend=
converted.end<cv::Vec3b>();
// отримуємо ітератори вихідного зображення
cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

// для кожного пікселя
for ( ; it!= itend; ++it, ++itout) {
```

Змінна converted містить зображення після перенесення кольорів. У класі ColorDetector вона визначена як атрибут класу:

```
class ColorDetector {
      private:
      // зображення що зберігає зображення з перенесеними
         кольорами
      cv::Mat converted;
```

Також необхідно перетворити вхідний цільовий колір. Ви можете це зробити створивши тимчасове зображення, яке містить лише один піксель. Зверніть увагу, що вам потрібно зберігати той самий підпис, як і в попередніх підрозділах, щоб користувач продовжував надавати цільовий колір у RGB:

```
// Встановлює кольору бути виявленим
void setTargetColor(unsigned char red, unsigned char green,
                    unsigned char blue) {
```

```
// тимчасове однопіксельне зображення
cv::Mat tmp(1,1,CV_8UC3);
tmp.at<cv::Vec3b>(0,0)= cv::Vec3b(blue, green, red);

// Перенесення зображення в колірний простір Lab
cv::cvtColor(tmp, tmp, CV_BGR2Lab);

target= tmp.at<cv::Vec3b>(0,0);
}
```

Якщо застосування попереднього рецепта зкомпілювати з цим зміненим класом, тепер воно виявить пікселі цільового кольору за допомогою колірної моделі CIE L * a * b * .

3.4.2 Опис методики

Коли зображення переноситься з одного колірного простіру в інший, лінійне або нелінійне перетворення застосовується до кожного вхідного пікселя щоб створити вихідні пікселі. Тип пікселів вихідного зображення буде співпадати з типом вхідного зображення. Навіть якщо ви працюєте з восьмибітними пікселями більшість часу, ви можете також використовувати перенесення з зображеннями з плаваючою точкою (у випадку яких вважається, що значення пікселів змінюється у межах від 0 до 1.0) або з цілочисельними зображеннями(з пікселями що зазвичай змінюються у межах від 0 до 65535). Однак, інтервал значень пікселя залежить від конкретного колірного простору і типу зображення призначення. Наприклад у колірному просторі CIE L*a*b* канал L, що представляє яскравість кожного пікселя, змінюється у межах від 0 до 100 і змінює масштаб у межах від 0 до 255 у випадку восьмибітних зображень.

Канали a і b відповідають хроматичним компонентам. Ці канали містять інформацію про колір пікселя, незалежно від його яскравості. Ці значення змінюються у межах від -127 до 127. Для восьмибітних зображень 128 додається до кожного значення, щоб вмістити його в інтервал від 0 до 255. Однак зверніть увагу, що восьмибітне перенесення кольору внесе помилки округлення, що зробить перетворення недосконало зворотнім.

Доступними є найбіль загальновживані колірні простори. Це лише питання надання функції OpenCV правильного коду перенесення колірного простору(для CIE L*a*b* цей код - CV_BGR2Lab). Серед них є YcrCb, що являється колірним простором, що використовується для стиснення JPEG. Щоб перенести з колірного простору BGR у YcrCb, кодом буде CV_BGR2YCrCb. Зверніть увагу, що всі перенесення, що включають три звичайні основні кольори, червоний, зелений, синій, доступні у порядку RGB та BGR.

Колірний простір CIE L*u*v* є ще одним однорідним для сприйняття колірним простором. Ви можете переносити з BGR у CIE L*u*v використо-

вуючи ту саму формулу перенесення для каналу яскравості, але використовувати інше представлення каналів хроматичності. Також зверніть увагу, що ,черезе те що ці два колірні простори деформують інтервал кольору RGB щоб зробити його однорідним для сприйняття, ці перетворення нелінійні(отже вони дорогі для обчислення).

Також існує колірний простір CIE XYZ(з кодом CV_BGR2XYZ). Це стандартний колірний простір, що використовується для представлення будь який відчутний колір незалежним від пристрою шляхом. У обчисленнях колірних просторів L*u*v та L*a*b, колірний простір XYZ використовується як проміжне представлення. Перетворення між RGB та XYZ лінійне. Також цікаво звернути увагу, що канал Y відповідає версії зображення з рівнем сірого.

HSV та HLS — цікаві колірні простори, тому що вони розкладають кольори на такі їх компоненти, як відтінок і насиченість, плюс значення компоненту яскравості, що є більш природнім шляхом описання кольорів для людей. Наступний підрозділ буде представляти цей колірний простір.

Ви також можете переносити кольорові зображення в інтенсивності рівня сірого. Виходом буде одноканальне зображення:

```
cv::cvtColor(color, gray, CV_BGR2Gray);
```

Також можливо виконати перенесення у зворотньому напрямку, але тоді три канали отриманого зображення будуть однаково заповнені відповідними значеннями зображення рівня сірого.

3.5 Представлення кольорів з використанням відтінків, насиченості та яскравості

В цьому розділі ми експериментували з кольорами зображення. Ми використовували різні кольорові простори і намагалися визначити зони зображення рівномірного кольору. Спочатку було обрано простір кольорів RGB і, не зважаючи на те, що це ефективне представлення для захоплення і відображення кольорів в електронному зображенні, це представлення не дуже інтуїтивне. Насправді, це не шлях, яким люди думають про кольори, вони найчастіше описують кольори в поняттях їх глибини, яскравості або барвистості (тобто яскравий чи пастельний колір). Тоді, щоб допомогти користувачам визначати колір, користуючись більш інтуїтивними для них властивостями, був представлений кольоровий простір, заснований на принципах відтінків, насиченості і яскравості. В цьому підрозділі ми дослідимо поняття відтінку, насиченості та яскравості, як засоби описання кольору.

3.5.1 Методика використання

Перенесення BGR зображення в інший кольоровий простір робиться з використанням функції cv::cvtColor, яка була досліджена в попередньому підрозділі. Тут ми використаємо конверсію коду CV_BGR2HSV:

```
// перенести у простір HSV
cv::Mat hsv;
cv::cvtColor(image, hsv, CV_BGR2HSV);
```

Ми можемо повернутися назад до простору BGR, використовуючи код CV_HSV2BGR. Ми можемо візуалізувати кожен компонент HSV розділивши канали перенесеного зображення на три незалежні зображення наступним чином:

```
// розділимо три канали на три зображення
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// channels[0] це відтінок
// channels[1] це насиченість
// channels[2] це значення
```

Зверніть увагу, що третій канал - це значення кольору, тобто приблизний показник яскравості кольору. Оскільки ми працюємо над 8-бітовими зображеннями, OpenCV змінює значення каналу, щоб покрити діапазон від 0 до 255 (за винятком відтінку, який змінюється між 0 і 0180, як це буде пояснено в наступному розділі). Це дуже зручно, оскільки ми можемо відображати ці канали як зображення сірого рівня.

Канал значення палацу буде виглядати наступним чином (рисунок 3.5):

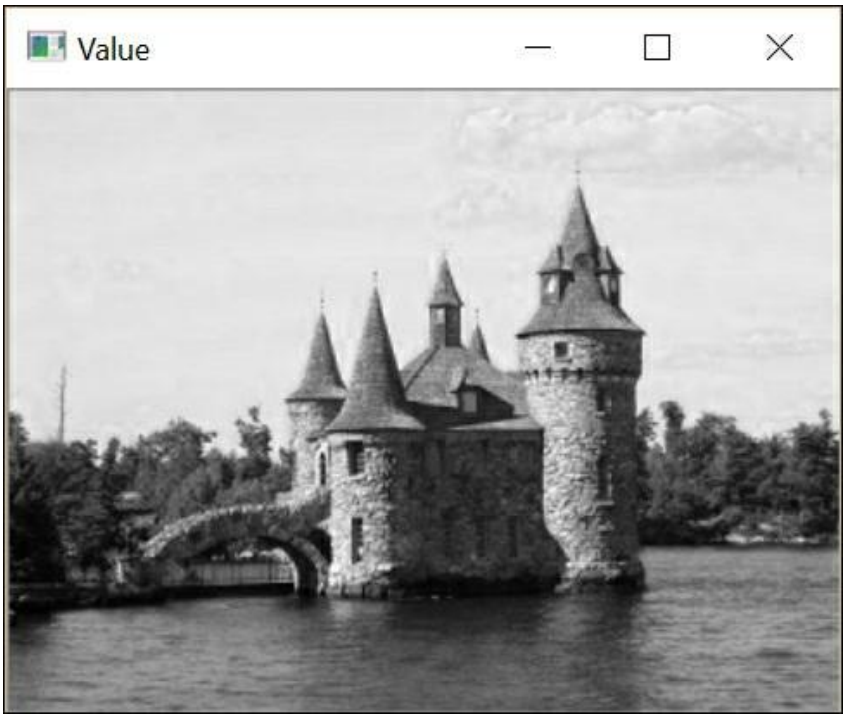


Рисунок 3.5 - Канал значення

Також зображення в каналі насиченості виглядатиме наступним чином (рисунок 3.6):

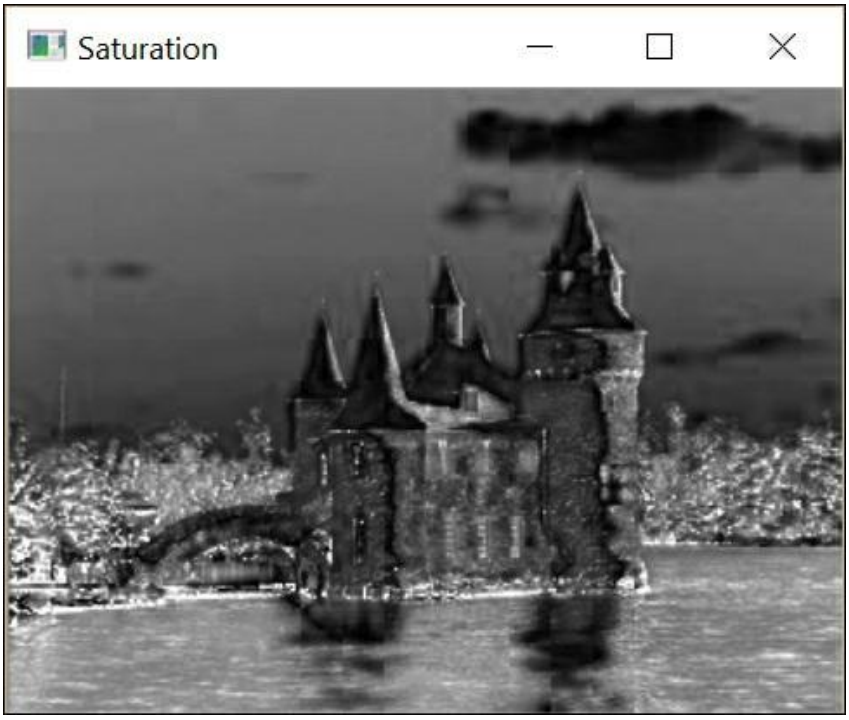


Рисунок 3.6 - Канал насиченості

Нарешті, зображення з каналом відтінку виглядає наступним чином(рисунок 3.7):

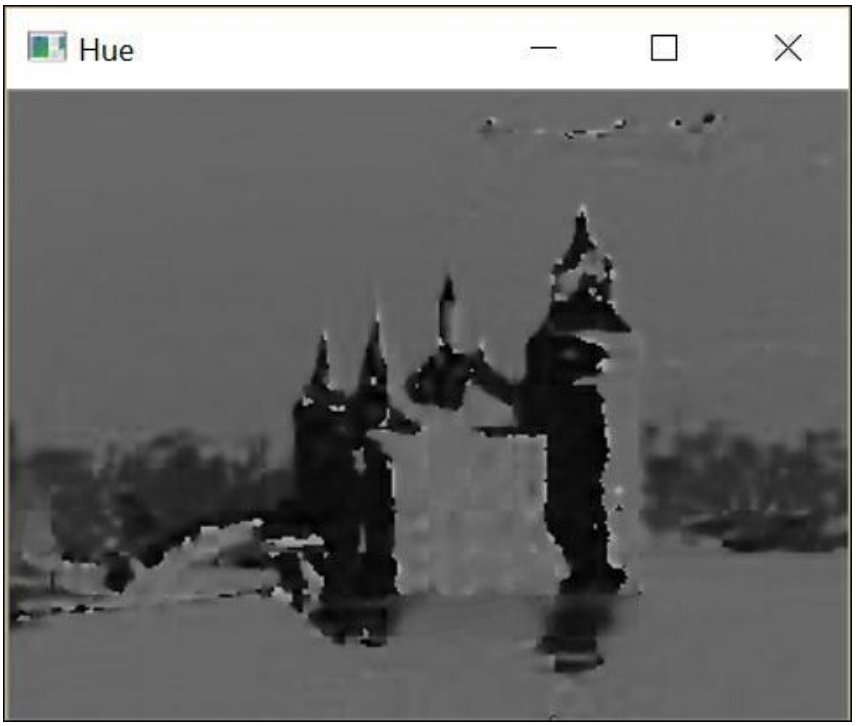


Рисунок 3.7 - Канал відтінку

Ці зображення інтерпретуються в наступному розділі.

3.5.2 Опис методики

Колірний простір відтінок/насиченість/значення був запропонований оскільки це представлення відповідає способу, яким люди природньо схильні впорядковувати кольори. І справді люди надають перевагу лписанню кольорів інтуітивними атрибутами, такими як глибина тону, барвистість і яскравість. Ці три атрибути є основою більшості феноменальних колірних просторів. Відтінок указує на домінуючий колір. Імена, що ми діємо кольорам (такі як зелений, жовтий, синій та червоний) відповідають різним значенням відтінку. Насиченість показує нам, наскільки яскравим є колір. Пастельні кольори мають низьку насиченість, тоді як кольори веселки дуже насичені. Нарешті яскравість — це суб'єктивний атрибут, який стосується світності кольору. Інші феноменальні колірні простори використовують концепцію значення кольору, або світлість кольору як спосіб охарактеризувати відносну інтенсивність кольору.

Ці кольорові компоненти намагаються імітувати інтуїтивне людське сприйняття кольорів. Як наслідок, для них немає стандартного визначення. У літературі ви знайдете кілька різних визначень і формул відтінку, насиченості і яскравості. OpenCV пропонує дві реалізації феноменальних

колірних просторів: колірні простори HSV і HLS. Формули перетворення дещо відрізняються, але вони дають дуже схожі результати.

Компонент значення, ймовірно, найлегше інтерпретувати. У реалізації OpenCV простору HSV вона визначається як максимальне значення трьох компонентів BGR. Це дуже спрощена реалізація концепції яскравості. Для визначення яскравості, яка краще відповідає візуальній системі людини, слід використовувати L-канал однорідних L * a * b * і L * u * v * колірних просторів. Наприклад, канал L враховує той факт, що зелений колір здається людині яскравішим, ніж, наприклад, синій колір тієї ж інтенсивності.

Для обчислення насиченості OpenCV використовує формулу, засновану на мінімальних і максимальних значеннях компонентів BGR:

```
S = (max(R, G, B)-min(R, G, B))/max(R, G, B)
```

Ідея полягає в тому, що колір у градаціях сірого, в якому всі три компоненти R, G і B є рівними, буде відповідати абсолютно ненасиченому кольору; отже, він буде мати значення насичення 0. Насиченість - це значення між 0 і 1.0. Для 8-бітових зображень насиченість збільшується до значення від 0 до 255, а при відображенні у вигляді зображення сірого кольору яскравіші ділянки відповідають кольорам, які мають вищий колір насичення.

Наприклад, з зображення насичення в попередньому розділі можна бачити, що синій колір води більш насичений, ніж світло-синій пастельний колір неба, як і очікувалося. Різні відтінки сірого мають, за визначенням, значення насичення, рівне нулю (тому що в цьому випадку всі три компоненти BGR рівні). Це можна спостерігати на різних дахах замку, які зроблені з темно-сірого каменю. Нарешті, у зображенні насиченості, ви могли помітити деякі білі плями, розташовані в областях, які відповідають дуже темним областям вихідного зображення. Вони є наслідком використовуваного визначення насичення. Дійсно, оскільки насичення вимірює тільки відносну різницю між максимальним і мінімальним значеннями BGR, триплет, такий як (1,0,0), дає ідеальну насиченість 1,0, навіть якщо цей колір буде розглядатися як чорний. Отже, значення насичення, виміряні в темних областях, ненадійні і не повинні розглядатися.

Відтінок кольору, як правило, представлений значенням кута від 0 до 360, причому червоний колір - 0 градусів. У випадку 8-бітового зображення, OpenCV ділить цей кут на два, щоб він містився в діапазоні 1 байт. Отже, кожне значення відтінку відповідає заданому колірному відтінку, незалежно від його яскравості і насиченості. Наприклад, як небо, так і вода мають однакове значення відтінку, приблизно 200 градусів (інтенсивність, 100), що відповідає блакитному відтінку; Зелений колір дерев на задньому плані має відтінок близько 90 градусів. Важливо зазначити, що відтінок менш надійний при оцінці кольорів, які мають дуже низьку насиченість.

Колірний простір HSB часто представлений конусом, де кожна точка всередині відповідає певному кольору. Кутове положення відповідає відтінку кольору, насиченість - відстань від центральної осі, а яскравість - висотою. Вершина конуса відповідає чорному кольору, для якого тон і насиченість не визначені (рисунок 3.8):

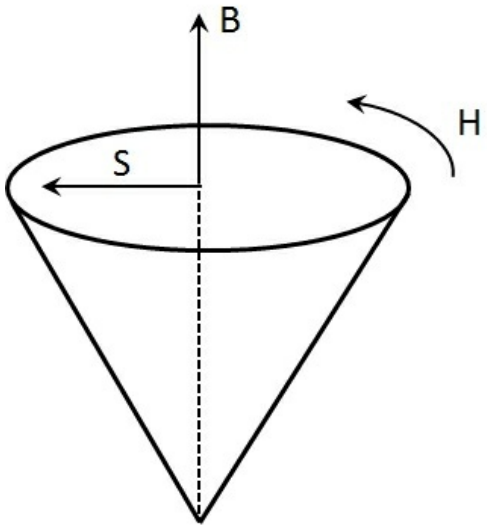


Рисунок 3.8 - Представлення колірного простору hsb у вигляді конуса

Ми також можемо згенерувати штучне зображення, яке продемонструє різні поєднання відтінку і насиченості.

```
cv::Mat hs(128, 360, CV_8UC3);
for (int h = 0; h < 360; h++) {
      for (int s = 0; s < 128; s++) {
            hs.at<cv::Vec3b>(s, h)[0] = h/2; // всі кути
                                                відтінку
            // від високої насиченості до низької
            hs.at<cv::Vec3b>(s, h)[1] = 255-s*2;
            hs.at<cv::Vec3b>(s, h)[2] = 255; // постійне
                                                значення
      }
}
```

Стовпчики наступного скриншоту показують різні можливі відтінки (від 0 до 180), тоді як різні рядки демонструють ефект насиченості. Верхня частина зображення показує повністю насичений колір, тоді як нижня частина відповідає ненасиченому кольору. Значення яскравості 255 було присвоєне всім відображеним кольорам(рисунок 3.9):

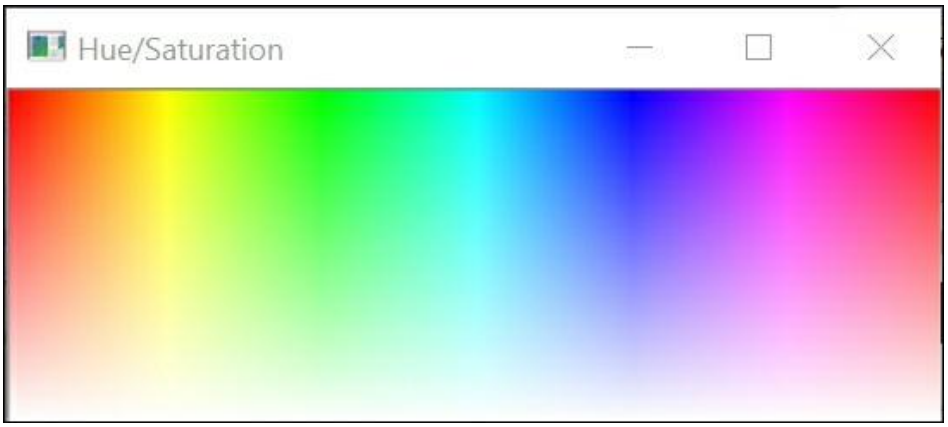


Рисунок 3.9 – демонстрація можливих відтінків і ефекту насиченості

Цікавий ефект може бути створений експериментами зі значеннями HSV. Декілька кольорових ефектів, що можуть бути створенні з використанням програм редагування фотографій, виконані за допомогою цього кольорового простору. Наприклад, ви можете вирішити змінити зображення, присвоївши постійну яскравість всім пікселям зображення без зміни відтінку і насиченості. Це може бути зроблено наступним чином:

```
// переносимо у простір HSV
cv::Mat hsv;
cv::cvtColor(image, hsv, CV_BGR2HSV);
// розділяємо три канали на три зображення
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// Канал значення буде 255 для всіх пікселів
channels[2]= 255;
// Поєднуємо канали назад
cv::merge(channels,hsv);
// Переносимо назад у BGR
cv::Mat newImage;
cv::cvtColor(hsv,newImage,CV_HSV2BGR);
```

Це дає наступне зображення, що виглядає як малюнок олівцями(рисунок 3.10).

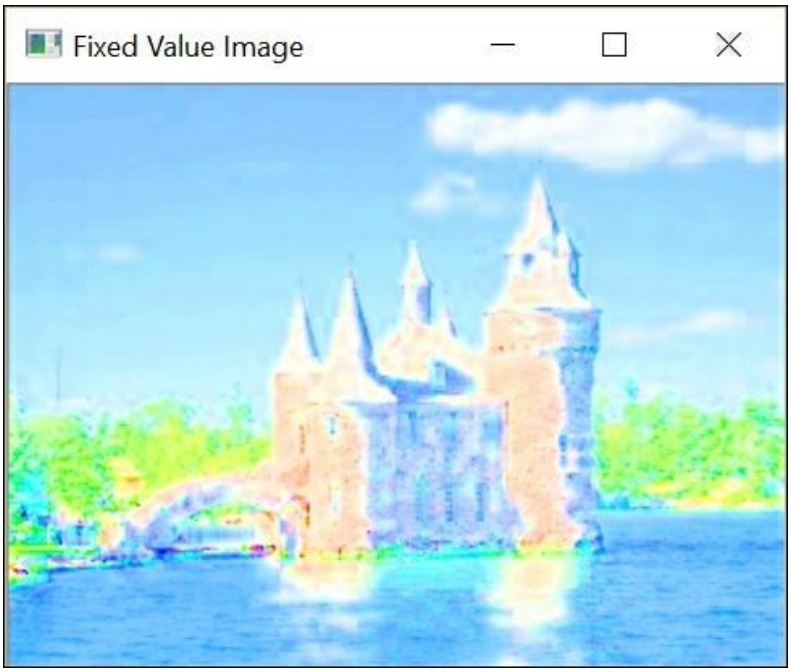


Рисунок 3.10 - присвоєння постійної яскравостівсім пікселям без зміни відтінку і насиченості

3.5.3 Розвиток методу

Колірний простір HSV також можна дуже впевнено використовувати, коли ви бажаєте шукати об’єкти заданого кольору.

3.5.3.1Використання кольорів для виявлення. Виявлення тону шкіри

Колірна інформація може бути дуже корисною для первинного виявлення конкретних об’єктів. Наприклад, виявлення дорожних знаків у додатку-помічнику водія може покладатися на кольори стандартних знаків щоб швидко визначити потенційних кандидатів у дорожні знаки. Виявлення кольору шкіри — це інший приклад у якому виявлені ділянки шкіри можуть бути використані як показник присутності людини на зображенні. Цей підхід дуже часто використовується у роспізнаванні жестів, де виявлення тону шкіри використовується щоб виявити позицію рук.

У загальному випадку для виявлення об’єкту з використанням кольору нам спочатку необхідно накопичити велику базу даних зразків зображень, що містять об’єкти, захоплені в різних умовах огляду. Вони будуть використані щоб визначити параметри вашого класифікатору. Вам також необхідно обрати представлення кольору, що ви будете використовувати для класифікації. Для виявлення тону шкіри різні дослідження показують, що кольори шкіри різних етнічних груп добре кластирезуються у просторі відтінок/насиченість. З цієї причини ми будемо просто використовувати значення відтінку і насиченості щоб визначати тони шкіри на наступному зображенні(рисунок 3.11):

Рисунок 3.11 - Вхідне зображення

Ми визначили функцію, що класифікує пікселі зображення, як шкіру, або не шкіру, просто спираючись на інтервали значень(мінімальний і максимальний відтінок та мінімальна і максимальна насиченість):

```
void detectHScolor(const cv::Mat& image, // вхідне
                                            зображення
           double minHue, double maxHue, // Проміжок
                                            відтінку
           double minSat, double maxSat, // Проміжок
                                            насиченості
           cv::Mat& mask) {              // Вихідна маска

      // перенесення в простір HSV
      cv::Mat hsv;
      cv::cvtColor(image, hsv, CV_BGR2HSV);

      // розбивання трьох каналів на три зображення
      std::vector<cv::Mat> channels;
      cv::split(hsv, channels);
      // channels[0] — це відтінок
      // channels[1] — це насиченість
      // channels[2] — це значення

      // Маскування відтінку
      cv::Mat mask1; // менше максимального відтінку
      cv::threshold(channels[0], mask1, maxHue, 255,
      cv::THRESH_BINARY_INV);
      cv::Mat mask2; // більше мінімального відтінку
      cv::threshold(channels[0], mask2, minHue, 255,
                  cv::THRESH_BINARY);
      cv::Mat hueMask; // маска відтінку
```

```
      if (minHue < maxHue)
            hueMask = mask1 & mask2;
      else // якщо інтервал перетинає вісь нуля градусів
            hueMask = mask1 | mask2;
      // маскування насиченості
      // між мінімальною і максимальною насиченістю
      cv::Mat satMask; // маска насиченості
      cv::inRange(channels[1], minSat, maxSat, satMask);

      // комбінована маска
      mask = hueMask & satMask;
}
```

Маючи великий набір зразків шкіри (і не шкіри) у нашому розпорядженні, ми могли б використати імовірнісний підхід, у якому буде підрахована ймовірність спостерігання даного кольору у класі шкіри проти спостерігання того самого кольору у класі не шкіри. Тут ми емпірично визначаємо допустимий інтервал відтінку/насиченості для нашого тестового зображення (пам’ятайте, що 8-ми бітна версія відтінку лежить у проміжку від 0 до 180, а насиченість лежить у проміжку від 0 до 255) :

```
// виявлення тону шкіри
cv::Mat mask;
detectHScolor(image, 160, 10, // відтінок від 320 градусів
                              до 20 градусів
              25, 166,        // насиченість від ~0.1
                              до 0.65
              mask);
// показати замасковане зображення
cv::Mat detected(image.size(), CV_8UC3, cv::Scalar(0, 0, 0));
image.copyTo(detected, mask);
```

Як результат отримано наступне зображення виявлення(рисунок 3.12):

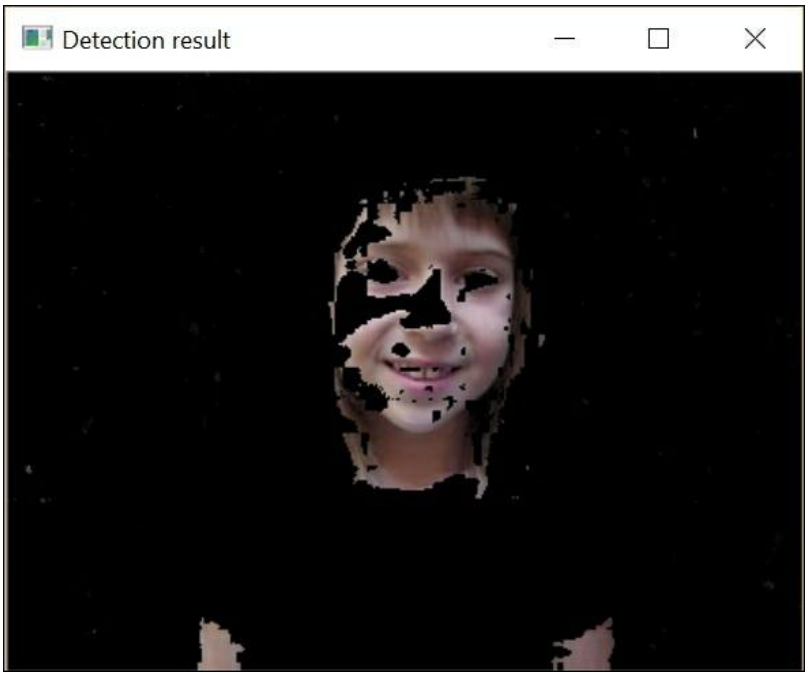


Рисунок 3.12 - Визначення тону шкіри

Зверніть увагу, що, для простоти, при виявленні ми не брали до уваги яскравість кольору. На практиці виключення яскравіших кольорів знизить можливість хибного виявлення яскравих червонуватих кольорів як шкіри. Очевидно, що надійне і точне виявлення кольору шкіри вимагатиме набагато ретельнішого аналізу. Дуже важко гарантувати гарне виявлення серед різних зображень, через те що багато факторів впливає на подання кольору у фотографії, такі як баланс білого і умови освітлення. Не зважаючи на це, як було показано, використання інформації про відтінок/насиченості як первинного детектору дає нам прийнятні результати.

3.6 Контрольні запитання

1. Що таке колбочки та за що вони відповідають?
2. Які основні кольори використовуються в цифровому зображенні та в чому їхня особливість?
3. Що таке шаблон дизайну стратегії і для чого він використовується?
4. Яким чином можна визначити поняття «відстань міського кварталу»?
5. Що таке евклідова відстань і яким способом визначається евклідова норма методами *OpenCV*?
6. Для чого використовується та які особливості мають маски зображень в *OpenCV*?
7. Перерахуйте відомі Вам колірні простори.
8. Дайте стислий опис колірного простору *HSV*.

3.7 Хід роботи

3.7.1 Підготовчі стадії

Поза навчальною лабораторією треба зробити наступне:

1. Користуючись підрозділами 3.1 – 3.5, а також рекомендованою літературою, ознайомитися із можливостями *OpenCV* щодо обробки кольорів зображення, складом та особливостями відповідних функцій.

2. Оформити першу частину звіту з виконання лабораторної роботи, в якій зазначити прізвище студента, мету лабораторних досліджень та дати письмові відповіді на контрольні питання (п. 3.6).

3.7.2 Завдання до роботи

Для того, щоб практично засвоїти можливості *OpenCV* щодо обробки кольорів, необхідно розробити код та з власими зображеннями:

1. Виконати програмну реалізацію алгоритму для розпізнавання заданого кольору на зображенні. В якості міри приналежності кольору заданому використовувати різницю значень кольорів пікселів.

2. Виконати розпізнавання кольору на зображенні, використовуючи функцію `floodFill`.

3. Виконати сегментацію зображення за допомогою алгоритму `GrabCut`.

4. Виконати розпізнавання кольору для зображення у колірних системах *LAB* та *HSV*.

5. Встановити постійну яскравість усім пікселям зображення в системі *HSV*.

6. Виконати розпізнавання тону шкіри.

3.7.3 Пояснення до виконання завдань

Оскільки маємо справу з кольоровими зображеннями, то кожен піксель міститиме три значення – для кожного з каналів моделі *BGR*. Для порівняння схожості таких пікселів та заданого кольору (який також задається масивом із трьох значень) доцільно використати суму абсолютних різниць між значеннями відповідних кольорових каналів.

Слід зазначити, що функція `floodFill` вважає розпізнаними лише ті пікселі, які мають такий же колір, як заданий початковий піксель, а ткож пов’язані через інші пікселі-сусіди з таким же кольором. Тобто, створюється «ланцюг» із пікселів з однаковим кольором, початок якого – задана користувачем точка.

Для розпізнавання кольору на зображенні у колірних системах *LAB* та *HSV*, перш за все, необхідно виконати конвертацію зображення із *BGR* в *HSV* та *LAB*. Для цього можна скористатися функцією `cvtColor`.

Для розпізнавання тону шкіри можна використати функцію `inRange`, яка залишає на зображенні лише ті кольори, які знаходяться у

заданому проміжку. Шукані границі, в яких приблизно знаходиться колір шкіри, як правило, встановлюють емпіричним шляхом.

3.8 Вимоги до звіту по роботі

Звіт про виконання роботи повинен містити:

1. Прізвище та групу студента, а також назву, мету та завдання лабораторної роботи.
2. Відповіді на контрольні питання.
3. Таблицю з переліком функцій за категоріями, які підтримуються .
4. Текст розробленої програми з коментарями.
5. Скрін-шоти роботи розробленої програми.
6. Висновки, в яких треба зазначити, чи досягнута мета даної лабораторної роботи. Яку саме конкретно обробку кольорів на зображеннях (перерахувати по пунктах) дозволяє робити бібліотека *OpenCV*?

3.9 Орієнтовні варіанти завдань

Варіанти завдань видаються викладачем після виконання студентом підготовчої стадії (п.3.7.1).

4 Лабораторна робота №4. Використання гістограм для обробки зображень

Мета роботи: Експериментуючи з зображеннями, навчитися обчислювати та вирівнювати гістограми, застосовувати таблиці пошуку для зміни зображення та виявляти на ньому певний вміст, використовувати алгоритм середнього зсуву для пошуку об'єкта, отримувати подібні зображення шляхом порівняння гістограм, відстежувати об'єкти, використовуючи гістограми.

4.1 Обчислення гістограми зображення

Зображення складається з пікселів, які мають різні значення. Наприклад, у 1-канальному зображенні з градаціями рівнів сірого кожен піксель має ціле значення між 0 (чорним) і 255 (білим). Залежно від вмісту зображення всередині його можна знайти різні рівні сірого відтінку.

Гістограма – це проста таблиця, яка дає кількість пікселів, які мають задане значення у зображенні (або іноді у наборі зображень). Таким чином, гістограма зображення з градаціями сірого матиме 256 записів (entries) (або бункерів, контейнерів – bin). Bin 0 дає кількість пікселів на зображенні, які мають значення 0, bin 1 – кількість пікселів, які мають значення 1, і так далі.

Очевидно, що, якщо підрахувати всі записи гістограми, отримаємо загальну кількість пікселів. Гістограми також можуть бути нормалізовані так, щоб сума бункерів дорівнювала 1. У цьому випадку кожен бункер дає відсоток пікселів, які мають це конкретне значення на зображенні.

Обчислення гістограми з OpenCV можна легко зробити за допомогою функції `cv::calcHist`. Це загальна функція, яка може обчислити гістограму декількох каналів зображення будь-якого типу і діапазону значень пікселів. Зробимо її більш простою у використанні, спеціалізуючи клас для випадку 1-канальних зображень градацій сірого. Для інших типів зображень завжди можна безпосередньо скористатися функцією `cv::calcHist`, яка пропонує необхідну гнучкість. У наступному розділі буде розглянуто кожен з її параметрів.

Поки ініціалізація нашого спеціалізованого класу виглядає наступним чином:

```
//Створення гістограми зображень градацій сірого
   class Histogram1D {
     private:
      int histSize[1]; // кількість бункерів у гістограмі
       float hranges[2]; // діапазон значень const
       float* ranges[1]; // вказівник на діапазон значень
       int channels[1]; // номер каналу для дослідження

      public:
      Histogram1D() {
```

```
// Підготуймо аргументи за замовчуванням для одновимір-
ної гістограми
        histSize[0]= 256;         // 256 бункерів
        hranges[0]= 0.0;          // від 0 (включно)
        hranges[1]= 256.0;        // до 256 (виключно)
        ranges[0]= hranges;
        channels[0]= 0;           // дивимося на канал 0
      }
```

За допомогою визначених змінних членів обчислення гістограми сірого рівня можна виконати за допомогою наступного методу:

```
    // Обчислює 1D-гістограму
    cv::Mat getHistogram(const cv::Mat &image) {
      cv::Mat hist;
      // Обчислити 1D гістограму за допомогою calcHist
      cv::calcHist(&image, 1, // Гістограма 1 зображення
                              // тільки
                   channels,  // використовуваного
                              // каналу без маски
                   cv::Mat(), // використовується
                   hist,      // результуюча гістограма
                   1,         // це 1D гістограма
                   histSize,  // кількість діапазонів
                              // значень пікселів
      );

      return hist;
    }
```

Тепер програмі потрібно просто відкрити зображення, створити екземпляр `Histogram1D` і викликати метод `getHistogram`:

```
    // Зчитування вхідного зображення
cv::Mat image= cv::imread("group.jpg", 0);//відкрито Ч/Б

    // Об'єкт гістограми
    Histogram1D h;

    // Обчислити гістограму
    cv::Mat histo= h.getHistogram(image);
```

Об'єкт `histo` є простим одновимірним масивом з `256` записами. Таким чином, можна читати кожен бін, просто петляючи над масивом:

```
    // Петля над кожним біном
    for (int i=0; i<256; i++)
      cout << "Value " << i << " = "
     <<histo.at<float>(i) << endl;
```

В якості прикладу нижче наведені деякі значення для певного сюжетного зображення:

```
Value 7 = 159
Value 8 = 208
Value 9 = 271
Value 10 = 288
Value 11 = 340
Value 12 = 418
Value 13 = 432
Value 14 = 472
Value 15 = 525
```

Очевидно, з цієї послідовності чисел важко отримати зрозуміле значення. З цієї причини часто зручно відображати гістограму як функцію, наприклад, за допомогою тієї ж гістограми. Наступні методи створюють такий графік:

```
// Обчислює 1D-гістограму і повертає зображення
cv::Mat getHistogramImage(const cv::Mat &image, int
zoom=1)
 {
       // Спочатку обчислити гістограму
       cv::Mat hist=  getHistogram(image);
       // Створити зображення
       return getImageOfHistogram(hist, zoom);
     }
// Створити зображення гістограми (статичний метод)
static cv::Mat getImageOfHistogram
(const cv::Mat  &hist, int zoom) {
       // Отримати мінімальні та максимальні значення бін
       double maxVal = 0;
       double minVal = 0;
       cv::minMaxLoc(hist, &minVal, &maxVal, 0, 0);

       // Отримати розмір гістограми
       int histSize = hist.rows;
// Квадратне зображення, на якому відображатиметься гі-
стограма
       cv::Mat histImg(histSize*zoom, histSize*zoom,
                       CV_8U, cv::Scalar(255));
// Встановити найвищу точку на 90% nbins (тобто висоту
зображення)
       int hpt = static_cast<int>(0.9*histSize);

       // Намалюймо вертикальну лінію для кожного bin
```

```
       for (int h = 0; h < histSize; h++)
      {float binVal = hist.at<float>(h);
         if (binVal>0)     {
int intensity = static_cast<int>(binVal*hpt /maxVal);
cv::line(histImg, cv::Point(h*zoom, histSize*zoom),
cv::Point(h*zoom, (histSize -intensity)*zoom),
cv::Scalar(0), zoom)
                              }
      }
      return histImg;
}
```

Використовуючи метод `getImageOfHistogram`, можна отримати зображення функції гістограми у вигляді графіка, який малюється за допомогою ліній:

```
// Відображати гістограму як зображення
cv::namedWindow("Histogram");
cv::imshow("Histogram", h.getHistogramImage(image));
```

Спробуйте це зробити для власного тестового зображення, наприклад, групового фото на пленері або в інтер'єрі.

На побудованій гістограмі, наприклад, може статися, що зображення буде мати великий пік значень середнього рівня сірого і немалу кількість інших (скоріш за все темних) пікселів. Переконайтеся в цій «двогорбості», спробувавши інші свої зображення. До речі, ці дві групи переважно відповідають фону і передньому плану зображення. Це можна перевірити, встановивши поріг зображення при переході між цими двома групами. Для цього можна використовувати зручну функцію OpenCV, а саме – функцію `cv::threshold`. Для створення бінарного (двійкового) зображення, ми перекриваємо зображення при мінімальному значенні безпосередньо перед його збільшенням у бік високого піку гістограми (значення сірого, наприклад, становить `70`):

```
cv::Mat thresholded; // Вихідний бінарний образ
cv::threshold(image,thresholded,70, // Порогове
                                    // значення
              255,// Присвоєне значення пікселів
threshold value cv::THRESH_BINARY); // Тип порогу
```

За умови правильного вибору порогу отримане двійкове зображення ясно покаже нам фонову / передню частину сегментації.

Функція `cv::calcHist` має багато параметрів, що дозволяють використовувати їх у багатьох контекстах, які є такими:

```
void calcHist(const Mat*images, // Вихідні зображення
        int nimages,    // Кількість вихідних
                        // зображень (зазвичай 1)
    const int*channels, // Список каналів,
                  // які будуть використовуватися
        InputArray mask,     // Маска введення
                             //(розгляд пікселів)
        OutputArray hist,    // Вихідна гістограма
        int dims, // Розмірність гістограми
        const int*histSize, // Кількість бункерів
                             // у кожному вимірі
    const float**ranges,// Діапазон кожного виміру
    bool uniform=true,  // Істина, якщо однаково
                        //рознесені бункери
    bool accumulate=false)    // Для накопичення
                              // кількох викликів
```

У більшості випадків гістограма буде чи з 1-канального, чи 3-канального зображення. Однак функція дозволяє задати багатоканальне зображення, розподілене по декількох зображеннях (тобто декілька cv::Mat). Ось чому масив вхідних зображень є першим параметром цієї функції. Шостий параметр, `dims`, вказує розмірність гістограми, наприклад, `1` для 1D гістограми. Навіть, якщо ви аналізуєте багатоканальне зображення, при обчисленні гістограми не потрібно використовувати всі його канали. Канали, які слід розглядати, перераховані в масиві каналів, який має задану розмірність. У нашій реалізації класу цей єдиний канал за замовчуванням є каналом 0. Сама гістограма описується кількістю бункерів у кожному вимірі (це масив `histSize` цілих чисел) і мінімальними (включно) і максимальними (ексклюзивними) значеннями в кожному вимірі (задається масивом діапазонів 2-елементних масивів).

Також можна визначити неоднорідну гістограму (другий останній параметр у такому випадку буде встановлено у `false`). У цьому випадку потрібно вказати межі кожного буфера.

Як і в багатьох функціях OpenCV, можна задати маску, яка вказує, які пікселі потрібно включити до підрахунку (усі пікселі, для яких значення маски дорівнює `0`, ігноруються). Можна вказати два додаткові параметри, обидва з яких є булевими значеннями. Перший вказує, чи є гістограма однорідною або ні (істина є типовою). Другий дозволяє накопичити результат кількох розрахунків гістограми.

Якщо цей останній параметр є істинним, то кількість пікселів зображення буде додано до поточних значень, знайдених у вхідній гістограмі. Це корисно, коли потрібно обчислити гістограму групи зображень.

Отримана гістограма зберігається в екземплярі `cv::Mat`. Дійсно, клас `cv::Mat` можна використовувати для маніпулювання загальними *N*-

вимірними матрицями. Нагадаємо, що цей клас визначає метод для матриць розмірності 1, 2 і 3. Саме тому ми змогли записати наступний код при доступі до кожного біну 1D-гістограми в методі `getHistogramImage`:

```
float binVal = hist.at<float>(h);
```

Зверніть увагу, що значення в гістограмі зберігаються як значення `float`.

4.2 Обчислення гістограм кольорових зображень

Використовуючи ту ж функцію `cv::calcHist`, можна обчислити гістограми багатоканальних зображень. Наприклад, клас, який обчислює гістограми кольорових зображень BGR, можна визначити наступним чином:

```
class ColorHistogram {
     private:
          int histSize[3]; // Розмір кожного виміру
          float hranges[2];   // Діапазон значень
          const float* ranges[3];  // Діапазони для
                                   //кожного виміру
     int channels[3]; // Канал, який слід розглядати

     public:
ColorHistogram() {

// Підготуймо аргументи за замовчуванням для кольорової  гістограми, кожен з яких має однаковий розмір і діапазон
histSize[0]= histSize[1]= histSize[2]= 256;
hranges[0]= 0.0;    // Діапазон BRG від 0 до 256
hranges[1]= 256.0;
        ranges[0]= hranges;  // У цьому класі
        ranges[1]= hranges;  // усі канали мають
        ranges[2]= hranges; //однаковий діапазон

        channels[0]= 0;   // трьох каналів: B
        channels[1]= 1;   // G
        channels[2]= 2;   // R
}
```

У цьому випадку гістограма буде тривимірною. Тому нам потрібно вказати діапазон для кожного з трьох вимірів. У випадку нашого зображення BGR три канали мають один і той же `[0,255]` діапазон. Підготувавши таким чином аргументи, обчислимо гістограму кольорового зображення наступним методом:

```
// Обчислює гістограму.
cv::Mat getHistogram(const cv::Mat &image) {
     cv::Mat hist;

// Обчислити гістограму
cv::calcHist   (&image, 1,     // Гістограма 1
                       // зображення тільки
                channels, // використовуваного
                       // каналу без маски
                cv::Mat(),// використовується
                hist,     // результуюча гістограма
                3,   // Це тригонографічна кількість
                histSize, // діапазонів значень
                ranges    // пікселів
                );

     return hist;
}
```

Повертається тривимірний екземпляр `cv::Mat`. При виборі гістограми з `256` бінів ця матриця має `(256)^3` елементів, що представляє більше 16 мільйонів записів. У багатьох додатках було б краще зменшити кількість бункерів при обчисленні гістограми. Крім того, можна використовувати структуру даних `cv::SparseMat`, яка розроблена для представлення великих розріджених матриць (тобто матриць з дуже невеликими ненульовими елементами), не споживаючи занадто багато пам'яті. Функція `cv::calcHist` має версію, яка повертає одну таку матрицю. Тому, для того, щоб використовувати `cv::SparseMatrix`, можна просто змінити попередній метод:

```
// Обчислює гістограму.
cv::Mat getHistogram(const cv::Mat &image) {
                    cv::Mat hist;

// Обчислити гістограму
cv::calcHist   (&image, 1,     // Гістограма 1
                            // зображення тільки
                channels,       // використовуваного
                            // каналу без маски
                cv::Mat(),// Використовується
                     hist,// результуюча гістограма
           3,   // Це тригонографічна кількість
                histSize, // Діапазонів значень
                ranges    // Пікселів
```

```
                );

     return hist;
}
```

Гістограма в цьому випадку є тривимірною, що ускладнює її представлення. Можливим варіантом, що ілюструє розподіл кольорів у зображенні, може бути відображення окремих R, G і B гістограм.

4.3 Застосування таблиць пошуку для зміни зовнішнього вигляду зображення

Гістограми зображень фіксують спосіб відображення сцени, використовуючи доступні значення інтенсивності пікселів. Аналізуючи розподіл значень пікселів на зображенні, можна використати цю інформацію для зміни та, можливо, поліпшення зображення. Розглянемо метод, за допомогою якого можна використати просту *функцію відображення*, представлену **таблицею**, для зміни значень пікселів зображення. Таблиці пошуку часто створюються з розподілів гістограм.

Таблиця пошуку є простою функцією один-до-одного (або багато-до-одного), яка визначає, як значення пікселів перетворюються в нові значення. Це 1D-масив з `256` записів (у випадку звичайних зображень з градаціями сірого). Запис до таблиці дає нове значення інтенсивності відповідного рівня сірого, яке виражається наступним чином:

```
newIntensity = lookup[oldIntensity];
```

Функція `cv::LUT` у OpenCV застосовує таблицю пошуку до зображення для створення нового зображення. Оскільки таблиці пошуку часто побудовані з гістограм, додамо цю функцію до нашого класу `Histogram1D`:

```
static cv::Mat applyLookUp      (
          const cv::Mat& image, // Вхідне зображення
          const cv::Mat& lookup)    // 1x256 8U
       {  cv::Mat result;      // Вихідне зображення
          // Застосувати таблицю пошуку
          cv::LUT(image,lookup,result);
          return result;
       }
```

Коли таблиця пошуку застосовується до зображення, це призводить до створення нового зображення, в якому значення інтенсивності пікселів змінено, як це передбачено таблицею пошуку. Просте перетворення можна визначити наступним чином:

```
// Створити таблицю інверсії зображення
```

```
cv::Mat lut(1,256,CV_8U); // 256x1 matrix

for (int i=0; i<256; i++)  {
     //0 стає 255, 1 стає 254 і т.д.
     lut.at<uchar>(i)= 255-i;
                                  }
```

Це перетворення просто інвертує інтенсивність пікселів, тобто інтенсивність 0 стає 255, 1 стає 254, і так далі до 255, що стає 0. Застосування такої таблиці пошуку до зображення призведе до негативу вихідного зображення.

4.4 Розтягування гістограми для поліпшення контрасту зображення

Можна поліпшити контраст зображення, визначивши таблицю пошуку, яка змінює гістограму вихідного зображення. Наприклад, якщо на гістограмі зображення майже немає пікселів зі значенням, що перевищує 200, можна розтягнути гістограму, щоб підвищити контраст. Для цього в процедурі використаємо поріг, який визначає відсоток пікселів, яким може бути присвоєно мінімальне значення інтенсивності (0) і максимальне значення інтенсивності (255) на розтягнутій гістограмі.

Тобто ми повинні знайти найнижчі (imin) та найвищі (imax) значення інтенсивності так, щоб мати необхідну кількість пікселів нижче або вище зазначеного процентиля. Це виконується за допомогою наступних циклів (тут hist – обчислена 1D-гістограма):

```
// Кількість пікселів у процентилі
float number =  image.total()*percentile;

// Знаходимо лівий край гістограми
int imin = 0;
for (float count=0.0; imin < 256; imin++) {
// Кількість пікселів у imin і нижче має бути > number
     if ((count+=hist.at<float>(imin)) >= number)
     break;
                                          }

// Знаходимо правий край гістограми
int imax = 255;
for (float count=0.0; imax >= 0; imax--) {
// кількість пікселів у imax і нижче має бути > number
     if ((count += hist.at<float>(imax)) >= number)
     break;
                                          }
```

Потім значення інтенсивності можуть бути перепризначені таким чином, щоб значення `imin` було змінено на інтенсивність `0`, а значенню `imax` присвоєно значення `255`. Інтенсивності `i` в інтервалі [`imin`; `imax`] просто лінійно переписуються наступним чином:

```
255.0*(i-imin)/(imax-imin);
```

4.5 Застосування таблиці пошуку для кольорових зображень

У лабораторній роботі, присвяченій *Маніпуляції з пікселями*, ми визначили функцію зменшення кількості можливих кольорів, яка змінює значення BGR зображення. Ми це зробили, прокрутивши всі пікселі зображення і застосувавши функцію зменшення кольору до кожного з них. Насправді, було б набагато ефективніше попередньо вирахувати всі скорочення кольорів, а потім змінити кожен піксель, використовуючи таблицю пошуку. Це дійсно легко зробити з того, про що ми дізналися в попередньому розділі. Нова функція зменшення кольору буде записана наступним чином:

```
void colorReduce(cv::Mat &image, int div=64)   {

      // Створення 1D-таблиці пошуку
      cv::Mat lookup(1,256,CV_8U);

      // Визначення пошуку зменшення кольору
      for (int i=0; i<256; i++)
           lookup.at<uchar>(i)= i/div*div + div/2;
      // Таблиця пошуку, застосована на всіх каналах
      cv::LUT(image,lookup,image);
                                              }
```

Схема зменшення кольору тут застосовується правильно, оскільки, коли одновимірна таблиця пошуку застосовується до багатоканального зображення, це одночасно відбувається індивідуально до всіх каналів. Коли таблиця пошуку має більше одного виміру, вона має бути застосована до зображення з однаковою кількістю каналів.

4.6 Вирівнювання гістограми зображення

У попередньому підрозділі ми показали, як можна покращити контраст зображення, розтягуючи гістограму так, щоб вона займала весь діапазон доступних значень інтенсивності. Ця стратегія дійсно являє собою легке виправлення, яке може ефективно покращити якість зображення. Однак у багатьох випадках візуальний недолік зображення не в тому, що воно використовує занадто вузький діапазон інтенсивностей.

Швидше за все, деякі значення інтенсивності використовуються набагато частіше, ніж інші. Гістограма типового сюжетного зображення, як правило, має багато пікселів з інтенсивністю середньо-сірого кольору, тоді як темніші та яскравіші пікселі досить рідкісні. Одним з можливих способів поліпшення якості зображення може бути рівне використання всіх наявних інтенсивностей пікселів. Ця ідея лежить в основі концепції **вирівнювання гістограми**, тобто створення якомога більш рівної гістограми зображення.

OpenCV пропонує просту у використанні функцію, яка виконує вирівнювання гістограми. Її називають наступним чином:

```
cv::equalizeHist(image,result);
```

Звичайно, гістограма не може бути абсолютно рівною, тому що таблиця пошуку є глобальним перетворенням багато-до-одного. Однак можна бачити, що загальний розподіл гістограми тепер є більш рівномірним, ніж вихідний.

В абсолютно однорідній гістограмі всі бункери матимуть рівну кількість пікселів. Це означає, що `50` відсотків пікселів повинні мати інтенсивність, меншу, ніж `128` (середнє значення інтенсивності), `25` відсотків повинні мати інтенсивність нижче `64` і так далі. Це спостереження можна виразити, використовуючи правило, що в однорідній гістограмі, `p%` пікселів повинні мати значення інтенсивності, менше або рівне `255*p%`. Правило, що використовується для вирівнювання гістограми, полягає в тому, що інтенсивність `i` треба відобразити в інтенсивність, яка відповідає відсотку пікселів, які мають значення інтенсивності нижче `i`. Тому необхідну таблицю пошуку можна побудувати з наступного рівняння:

```
lookup.at<uchar>(i)= static_cast<uchar> (
                          255.0*p[i]/image.total());
```

Тут `p[i]` – кількість пікселів, які мають інтенсивність, меншу або рівну `i`. Функцію `p[i]` часто називають *кумулятивною гістограмою*, тобто це гістограма, яка містить пікселі, кількість яких нижче або дорівнює заданій інтенсивності, замість того, щоб містити кількість пікселів, які мають певне значення інтенсивності. Нагадаємо, що `image.total()` повертає кількість пікселів у зображенні, тому `p[i]/image.total()` – це частка (відсоток) пікселів.

Як правило, вирівнювання гістограми значно покращує зовнішній вигляд зображення. Однак, залежно від візуального вмісту, якість результату може змінюватися від зображення до зображення.

4.7 Проектування гістограми для виявлення певного вмісту зображення

Гістограма є важливою характеристикою вмісту зображення. Якщо подивитися на область зображення, що показує певну текстуру або конкретний об'єкт, то гістограму цієї області можна розглядати як функцію, яка дає ймовірність того, що даний піксель належить цій конкретній текстурі або об'єкту. У цьому підрозділі ми дізнаємося, як концепцію зворотного **проектування гістограми** можна вигідно використовувати для виявлення певного вмісту зображення.

Припустимо, у нас є зображення, і ми хочемо виявити певний вміст всередині нього (наприклад, на власному зображенні *хмари на небі*). Перше, що потрібно зробити, це вибрати область інтересу, що містить зразок того, що ви шукаєте. Ця область знаходиться в певному прямокутнику на тестовому зображенні.

У нашій програмі область інтересу отримаємо наступним чином:

```
cv::Mat imageROI;
imageROI = image(cv::Rect(216,33,24,30));
// Область з хмарами(приклад)
```

Далі витягнемо гістограму цієї області інтересу. Це легко виконати за допомогою класу `Histogram1D`, визначеного раніше в цій лабораторній роботі, наступним чином:

```
Histogram1D h;
cv::Mat hist= h.getHistogram(imageROI);
```

Нормалізуючи цю гістограму, отримуємо функцію, яка дає нам вірогідність того, що піксель заданого значення інтенсивності належить визначеній області, як зазначено нижче:

```
cv::normalize(histogram,histogram,1.0);
```

Повернення проекції гістограми складається з заміни кожного значення пікселя у вхідному зображенні відповідним значенням ймовірності, зчитаним у нормалізованій гістограмі. Функція OpenCV виконує це завдання наступним чином:

```
cv::calcBackProject(&image,
                    1,   // на одному зображенні
                         // використовуваних каналів,
                channels, // заснованих на розмірності
// гістограми на гістограмі, що повертається назад,
                histogram,// утворюється зворотне
```

```
                result,    // проекційне зображення
                ranges,    // діапазонів значень,
                             // коефіцієнт масштабу-
                вання
                255.0      // вибирається таким чином,
// щоб значення ймовірності 1 відображалося в 255
                      );
```

Якщо ми застосуємо поріг на тестовому зображенні, то отримаємо найбільш ймовірні пікселі *хмарності* (приклад, а можна – чогось іншого):

```
cv::threshold (result, result, threshold,
                 255, cv::THRESH_BINARY);
```

Попередній результат може розчарувати, оскільки, крім хмар (у прикладі), інші області також, можливо, були помилково виявлені. Важливо розуміти, що функція ймовірності була виділена з простої гістограми сірого рівня. Багато інших пікселів у зображенні мають ті ж інтенсивності, що і пікселі хмар, і пікселі тієї ж інтенсивності замінюються тим же значенням ймовірності, коли повертають проекцію гістограми. Одним з рішень для поліпшення результату виявлення буде використання інформації про колір. Однак для цього потрібно змінити формат виклику функції `cv::calBackProject`.

Функція `cv::calBackProject` подібна до функції `cv::calcHist`. Значення, асоційовані з пікселем, відносяться до одного контейнера (потенційно багатовимірної) гістограми. Але замість збільшення лічильника `bin`, функція `cv::calBackProject` призначає відповідному пікселю у вихідному зображенні зворотної проекції значення, прочитане в цьому біні. Перший параметр цієї функції визначає вхідні зображення (найчастіше тільки одне). Потім необхідно вказати номери каналів, які ми бажаємо використовувати. Гістограма, яка передається функції, на цей раз є вхідним параметром; її розмірність має відповідати одному з масивів списку каналів. Як і у випадку `cv::calcHist`, параметр діапазонів визначає кордони вхідних гістограм у вигляді масиву плаваючих масивів, кожен з яких визначає діапазон (мінімальне та максимальне значення) кожного каналу.

Отриманий результат є зображенням, що містить обчислену карту ймовірності. Оскільки кожен піксель замінюється на значення, знайдене в гістограмі у відповідному контейнері, результуюче зображення має значення між `0.0` і `1.0` (якщо нормалізована гістограма була надана в якості вхідних даних). Останній параметр дозволяє додатково масштабувати ці значення шляхом множення їх на заданий коефіцієнт.

4.8 Проектування кольорових гістограм

Багатовимірні гістограми також можуть бути повернені на зображення. Давайте визначимо клас, який інкапсулює процес зворотної проекції. Спочатку визначимо необхідні атрибути і ініціалізуємо дані наступним чином:

```
class ContentFinder {
     private:  // Параметри гістограми
          float  hranges[2];
          const  float* ranges[3];
          int    channels[3];
          float   threshold;   // Поріг рішення
          cv::Mat histogram;   // Вхідна гістограма

     public:
          ContentFinder() : threshold(0.1f) {

// У цьому класі всі канали мають однаковий діапазон
       ranges[0]= hranges;
       ranges[1]= hranges;
       ranges[2]= hranges;
}
```

Введено атрибут порогу (`threshold`), який використовується для створення двійкової карти, що показує результат виявлення. Якщо для цього параметра встановлено від'ємне значення, повертається невизначена карта ймовірностей. Вхідна гістограма нормалізується (це, однак, не потрібно) наступним чином:

```
// Встановити вихідну гістограму
void setHistogram(const cv::Mat& h)  {
                histogram = h;
                cv::normalize(histogram,histogram,1.0);
                                    }
```

Для гістограми зворотного проектування потрібно просто вказати зображення, діапазон (ми припустили тут, що всі канали мають однаковий діапазон), а також список використовуваних каналів. Метод `find` здійснює зворотну проекцію.

Доступні дві версії цього методу. Перша, що використовує три канали зображення, є більш загальною:

```
// Спрощена версія, в якій використовуються всі кана-
ли
// з діапазоном [0,256] за замовчуванням
cv::Mat find(const cv::Mat& image)  {
```

```
     cv::Mat result;
     hranges[0]= 0.0;     // діапазон за замовчуванням
                          //[0,256   [хребти   [1]   =
                    256,0;
     channels[0]= 0;// трьох каналів
     channels[1]= 1;
     channels[2]= 2;
 return     find(image,     hranges[0],     hranges[1],
channels);
                                        }
// Знаходить пікселі, що належать гістограмі
cv::Mat find(const cv::Mat& image,
             float minValue,
             float maxValue,
             int *channels) {
                           cv::Mat result;
                           hranges[0]= minValue;
                           hranges[1]= maxValue;
//Розмірність гістограми dim відповідає списку кана-
лів
     for (int i=0; i<histogram.dims; i++)
     this->channels[i]= channels[i];
cv::calcBackProject(&image, 1, // ми використовуємо
                    channels, // тільки один канал
                              // зображення
                    histogram,// використовуваний
                              // гістограмою
                    result,   // ми використовуємо
    // заднє проекційне зображення.
                    ranges,   // Діапазон значень
                              // Для кожного виміру
                    55.0 // коефіцієнт масштабування
                         // вибирається таким
                         // чином, щоб значення
               // гістограми 1 відображалося на 255
                         );
}
// Поріг зворотної проекції для отримання бінарного
зображення
if (threshold>0.0)
     cv::threshold(result, result, 255.0*threshold,
                           255.0, cv::THRESH_BINARY);
     return result;
}
```

Давайте тепер використаємо гістограму BGR на кольоровій версії зображення, яке використовували раніше. Цього разу спробуємо виявити область блакитного неба. Спочатку завантажимо кольорове зображення, визначимо область інтересу і обчислимо тривимірну гістограму на зменшеному колірному просторі наступним чином:

```
// Завантаження кольорового зображення
ColorHistogram hc;
cv::Mat color= cv::imread("Test3_3.jpg");

// екстракт представляє регіон інтересу
imageROI= color(cv::Rect(0,0,100,45));
// Область Синє небо

// Отримати тривимірну гістограму кольору (8 бінів на
канал)
hc.setSize(8); // 8x8x8
cv::Mat shist= hc.getHistogram(imageROI);
```

Далі обчислюють гістограму і використовують метод `find`, щоб визначити частину неба у зображенні наступним чином:

```
// Створити пошук вмісту
ContentFinder finder;
// Встановити гістограму зворотної проекції
finder.setHistogram(shist);
finder.setThreshold(0.05f);

// Отримати зворотну проекцію гістограми кольору
Cv::Mat result= finder.find(color);
```

Колірний простір BGR зазвичай не найкращий для ідентифікації кольорових об'єктів у зображенні. Тут, щоб зробити його більш надійним, ми зменшили кількість кольорів до обчислення гістограми (згадаймо, що початковий простір BGR нараховує більше 16 мільйонів кольорів). Витягнута гістограма являє собою типовий розподіл кольорів для області неба. Спробуйте переконатися у здатності виявляти частину неба на іншому зображенні.

Зверніть увагу, що використання гістограми, побудованої з декількох зображень неба, повинно підвищити точність цього виявлення.

У цьому випадку обчислення розрідженої гістограми було б кращим з точки зору використання пам'яті. Ви повинні бути в змозі повторити цю вправу за допомогою `cv::SparseMat` цього разу. Крім того, якщо потрібно знайти об'єкт яскравого кольору, більш ефективним буде скористатися

каналом відтінку колірного простору HSV. В інших випадках використання компонентів кольоровості перцептивно рівномірного простору (наприклад, `L*a*b*`) може становити кращий вибір.

4.9 Використання алгоритму середнього зсуву для пошуку об'єкта

Результатом зворотної проекції гістограми є карта ймовірності, яка виражає ймовірність того, що даний фрагмент зображення буде знайдено у певному місці зображення. Припустимо, що тепер ми знаємо приблизне розташування об'єкта на зображенні. Карту ймовірності можна використовувати для пошуку точного розташування об'єкта. Найбільш вірогідним буде розташування, яке максимізує цю ймовірність всередині даного вікна.

Тому, якщо ми починаємо з початкового розташування і ітеративно переміщаємося в спробі збільшити локальну ймовірну міру, то маємо можливість знайти точне розташування об'єкта. Це виконується **алгоритмом середнього зсуву**.

Припустимо, на тестовому зображенні ми хочемо позначити обличчя людини у якості об'єкта, що нас цікавить. Цього разу опишемо цей об'єкт, використовуючи канал відтінку колірного простору HSV. Це означає, що потрібно перетворити зображення в HSV, а потім витягти канал відтінку і обчислити гістограму 1D відтінку визначеного ROI. Проглянемо наступний код:

```
// Зчитування опорного зображення
cv::Mat image= cv::imread("Lab3_Test3.jpg");
// ROI обличчя людини
cv::Rect rect(110, 45, 35, 45);
cv::Mat imageROI= image(rect);
// Отримаємо гістограму відтінку обличчя людини
int minSat=65;
ColorHistogram hc;
cv::Mat colorhist= hc.getHueHistogram(imageROI,minSat);
```

Як видно, гістограма відтінку отримана за допомогою зручного методу, який ми додали до нашого класу `ColorHistogram` наступним чином:

```
// Обчислити гістограму 1D Hue
// Вихідне зображення BGR перетворюється в HSV
// Пікселі з низькою насиченістю ігноруються
cv::Mat getHueHistogram(const cv::Mat &image,
                        int minSaturation=0) {

cv::Mat hist;
// Перетворити на колірний простір HSV
```

```
cv::Mat hsv;
cv::cvtColor(image, hsv, CV_BGR2HSV);

// Маска для використання (чи ні)
cv::Mat mask;
// створення маски, якщо потрібно
if (minSaturation>0) {

// Розбиття 3 каналів на 3 зображення
std::vector<cv::Mat>  v;
cv::split(hsv,v);
// Маскуємо пікселі з низькою насиченістю
cv::threshold(v[1],mask,minSaturation,
255, cv::THRESH_BINARY);
}
// Підготуймо аргументи для гістограми 1D-відтінків
hranges[0]= 0.0;    // діапазон від 0 до 180
hranges[1]= 180.0;
channels[0]= 0;// канал відтінку

// Обчислити гістограму
cv::calcHist   (&hsv, 1, // Гістограма 1 зображення
                    channels, // тільки каналу,
                    mask,     // використовуваного
                              // двійковою маскою,
                    hist,     // отримана
                    1,        // гістограма - це 1D
                              // діаграма
                    histSize, // числа діапазонів
                    ranges    // значень пікселів
                );

return hist;
}
```

Потім отримана гістограма передається нашому екземпляру класу ContentFinder наступним чином:

```
ContentFinder finder;
finder.setHistogram(colorhist);
```

Давайте тепер відкриємо друге зображення, де ми хочемо знайти місце обличчя іншої людини. Це зображення потрібно спочатку перетворити на простір HSV, а потім повернути назад гістограму першого зображення. Звернімося до наступного коду:

```
image= cv::imread("Lab3_Test4.jpg");
// Перетворити на HSV
cv::cvtColor(image, hsv, CV_BGR2HSV);
// Отримати зворотну проекцію гістограми відтінку
int ch[1]={0};
finder.setThreshold(-1.0f); // немає порогу
cv::Mat result= finder.find(hsv,0.0f,180.0f,ch);
```

Тепер, з початкової прямокутної області (тобто положення особи людини на початковому зображенні), алгоритм `cv::meanShift` OpenCV оновлюватиме прямий об'єкт на місці обличчя нової людини наступним чином:

```
// Початкове положення вікна
cv::Rect rect(110,260,35,40);

// Об'єкт пошуку з середнім зсувом
cv::TermCriteria criteria(
cv::TermCriteria::MAX_ITER | cv::TermCriteria::EPS,
10,       // Повторюємо ітерацію до 10 разів
     ; // або поки зміна в центроїді не перевищує 1px
cv::meanShift(result,rect,criteria);
```

Заради порівняння відображаються початкові (червоні) та нові (зелені) розташування облич.

У цьому прикладі ми використовували компонент кольору колірного простору HSV, щоб охарактеризувати об'єкт, який шукали. Ми зробили цей вибір, тому що обличчя людини має дуже характерний колір; отже, відтінок пікселів повинен зробити обличчя легко ідентифікованим.

Отже, першим кроком є перетворення зображення в колірний простір HSV.

Компонент відтінку є першим каналом отриманого зображення, коли використовується прапор `CV_BGR2HSV`. Це 8-бітний компонент, який змінюється від `0` до `180` (з `cv::cvtColor` перетворене зображення має той же тип, що і вихідне зображення). Для вилучення зображення відтінку 3-канальне зображення HSV розбивається на три 1-канальні зображення, використовуючи функцію `cv::split`. Три зображення вставляються в екземпляр `std::vector`, а зображення відтінку – перший запис вектора (тобто в індексі `0`).

При використанні компонента відтінку кольору завжди важливо враховувати його насиченість (це другий запис вектора).

Дійсно, коли насиченість кольору низька, інформація про відтінок стає нестабільною і ненадійною. Це пов'язано з тим, що для низько-

насичених кольорів компоненти B, G і R майже рівні. Це ускладнює визначення точного кольору, який представляється.

Отже, ми вирішили проігнорувати компонент кольору з низькою насиченістю. Тобто, вони не враховуються в гістограмі (використовуючи параметр `minSat`, який маскує пікселі з насиченістю нижче цього порогу в методі `getHueHistogram`).

Алгоритм середнього зсуву є ітераційною процедурою, яка знаходить локальні максимуми функції ймовірності. Це робиться шляхом пошуку центроїда, або зваженого середнього, точки даних усередині попередньо визначеного вікна. Потім алгоритм переміщує центр вікна до розташування центроїда і повторює процедуру, поки центр вікна не сходиться до стабільної точки. Реалізація OpenCV визначає два критерії зупинки: максимальну кількість ітерацій (MAX_ITER) і значення зміщення віконного центру, нижче якого позиція вважається сходиною до стабільної точки (EPS). Ці два критерії зберігаються в екземплярі `cv::TermCriteria`. Функція `cv::meanShift` повертає кількість виконаних ітерацій. Очевидно, що якість результату залежить від якості карти ймовірності, що надається на заданій початковій позиції.

Зауважимо, що тут для представлення зображення використано гістограму кольорів. Також можна використовувати гістограми інших функцій для представлення об'єкта (наприклад, гістограму орієнтації краю).

4.10 Отримання подібних зображень за допомогою порівняння гістограми

Отримання зображень на основі вмісту є важливою проблемою в комп'ютерному зорі. Вона складається з пошуку набору зображень, які представляють вміст, схожий на дане зображення запиту. Оскільки гістограми є ефективним способом характеризувати вміст зображення, спробуємо їх використати для вирішення проблеми пошуку зображень на основі вмісту.

Ключовим моментом тут є можливість вимірювати подібність між двома зображеннями, просто порівнюючи їх гістограми. Для функції вимірювання, яка оцінить, наскільки різні, або як подібні зображення, треба визначити дві гістограми. Раніше були запропоновані різні подібні заходи, і OpenCV пропонує деякі з них у своїй реалізації функції `cv::compareHist`.

Щоб порівняти еталонне зображення з набором зображень і знайти ті, які найбільш схожі з цим зображенням запиту, ми створили клас `ImageComparator`. Цей клас містить посилання на зображення запиту та вхідне зображення разом з їх гістограмами. Крім того, оскільки ми будемо виконувати порівняння за допомогою гістограм кольорів, клас `ColorHistogram` використовується в нашому класі `ImageComparator`:

```
class ImageComparator { private:
  cv::Mat refH;          // довідкова гістограма
  cv::Mat inputH;        // гістограма вхідного
                         // зображення

  ColorHistogram hist;   // генерація гістограми
  int nBins;            // кількість бінів, що
                        // використовуються в
                        // кожному колірному каналі
  public:
  ImageComparator() :nBins(8) {

  }
```

Щоб отримати надійну міру подібності, гістограма повинна бути обчислена за зменшеною кількістю бункерів. Тому клас дозволяє вказати кількість бінів, які будуть використовуватися в кожному каналі BGR. Зображення запиту задається за допомогою відповідного сеттера, який також обчислює еталонну гістограму наступним чином:

```
// встановити і обчислити гістограму еталонного зобра-
ження

  void setReferenceImage(const cv::Mat& image)  {

    hist.setSize(nBins);
    refH= hist.getHistogram(image);
  }
```

Нарешті, метод порівняння порівнює еталонне зображення з заданим вхідним зображенням. Наступний метод повертає число, яке вказує, наскільки подібні два зображення:

```
// Порівняймо зображення, використовуючи гістограми
BGR

double compare(const cv::Mat& image) {

   inputH= hist.getHistogram(image);

// Порівняння гістограми за допомогою перетину
return cv::compareHist(refH,inputH,
 cv::HISTCMP_INTERSECT);
```

```
}
```

Попередній клас можна використовувати для отримання зображень, які схожі з даним зображенням запиту. Опорне зображення надається екземпляру класу наступним чином:

```
ImageComparator c;
c.setReferenceImage(image);
```

Зображення запиту, яке можна використати, – це кольорова версія тесту, що відображалося у підрозділі *Проектування гістограми для виявлення певного вмісту зображення* раніше в цій лабораторній роботі. Це зображення слід порівняти з серією подібних зображень та показати їх в порядку від найбільш подібних до найменш схожих.

Більшість заходів зіставлення гістограм засновані на порівнянні bin-by-bin (контейнера з контейнером). Ось чому важливо працювати зі зменшеною гістограмою, яка поєднує сусідній колір в один і той же бункер при вимірюванні подібності двох кольорових гістограм. Виклик `cv::compareHist` простий. Просто вводимо дві гістограми, і функція повертає виміряну відстань. Конкретний метод вимірювання, який ви хочете використовувати, вказується за допомогою прапора. У класі `ImageComparator` використовується метод перетину (з прапором `cv::HISTCMP_INTERSECT`). Цей метод просто порівнює для кожного контейнера два значення в кожній гістограмі і зберігає мінімальне значення. Таким чином, мірою подібності є сума цих мінімальних значень.

Отже, два зображення, які мають гістограми без загального кольору, отримають значення перетину `0`, тоді як дві ідентичні гістограми отримають значення, яке дорівнює загальній кількості пікселів.

Іншими доступними методами є вимірювання Chi-Square (`cv::HISTCMP_CHISQR` прапор), який підсумовує нормалізовану квадратну різницю між бункерами; метод кореляції (прапор `cv::HISTCMP_CORREL`), який заснований на нормалізованому операторі взаємної кореляції, що використовується в обробці сигналів для вимірювання подібності між двома сигналами; міра Бхаттачарії (прапор cv::HISTCMP_BHATTACHARYYA) і розбіжності Kullback-Leibler (прапор `cv::HISTCMP_KL_DIV`), які використовуються в статистиці для оцінки подібності між двома ймовірнісними розподілами.

4.11 Підрахунок пікселів з інтегральними зображеннями

У попередніх підрозділах ми дізналися, що гістограма обчислюється, проходячи через всі пікселі зображення і накопичуючи підрахунок того, як часто кожне значення інтенсивності відбувається на цьому зображенні. Іноді нас цікавить лише обчислення гістограми в певній області зображення. Справді, накопичення суми пікселів усередині субрегіону зображення є

звичайним завданням у багатьох алгоритмах комп'ютерного зору. Тепер припустимо, що ми повинні обчислити декілька таких гістограм по декількох регіонах, що цікавлять нас у зображенні. Всі ці розрахунки можуть швидко стати дуже дорогими.

У такій ситуації є інструмент, який може різко підвищити ефективність підрахунку пікселів над субрегіонами зображення: інтегральне зображення.

Інтегральні зображення були введені як ефективний спосіб підсумовування пікселів у цікавлячих областях зображення. Вони широко використовуються в додатках, які включають, наприклад, обчислення над нестійкими вікнами на декількох масштабах.

У цьому підрозділі пояснимо принцип, що стоїть за інтегральними зображеннями. Наша мета полягає в тому, щоб показати, як пікселі можуть бути підсумовані по прямокутній області, використовуючи тільки три арифметичні операції.

Для прикладу попрацюємо з зображенням, на якому ідентифікується область інтересу, що показує людину на велосипеді.

Інтегральні зображення корисні, коли потрібно зібрати пікселі в кількох областях зображення. Зазвичай, якщо ви хочете отримати суму всіх пікселів над інтересною областю, можна написати наступний код:

```
// Відкрити зображення
cv::Mat image= cv::imread("Lab3_test5.bmp",0);
// Визначити roi зображення (тут людина на велосипеді)
int xo=97, yo=112;
int width=25, height=30;
cv::Mat roi(image,cv::Rect(xo,yo,width,height));
// Обчислювальна сума
// повертає скаляр з багатоканальними зображеннями
cv::Scalar sum= cv::sum(roi);
```

Функція `cv::sum` просто перекриває всі пікселі області і накопичує суму. Використовуючи інтегральне зображення, це можна зробити за допомогою трьох адитивних операцій. Однак спочатку потрібно обчислити інтегральне зображення наступним чином:

```
// Обчислити цілісне зображення
cv::Mat integralImage;
cv::integral(image,integralImage,CV_32S);
```

Як буде пояснено далі, той же результат можна отримати, використовуючи цей простий арифметичний вираз на обчисленому інтегральному зображенні, наступним чином:

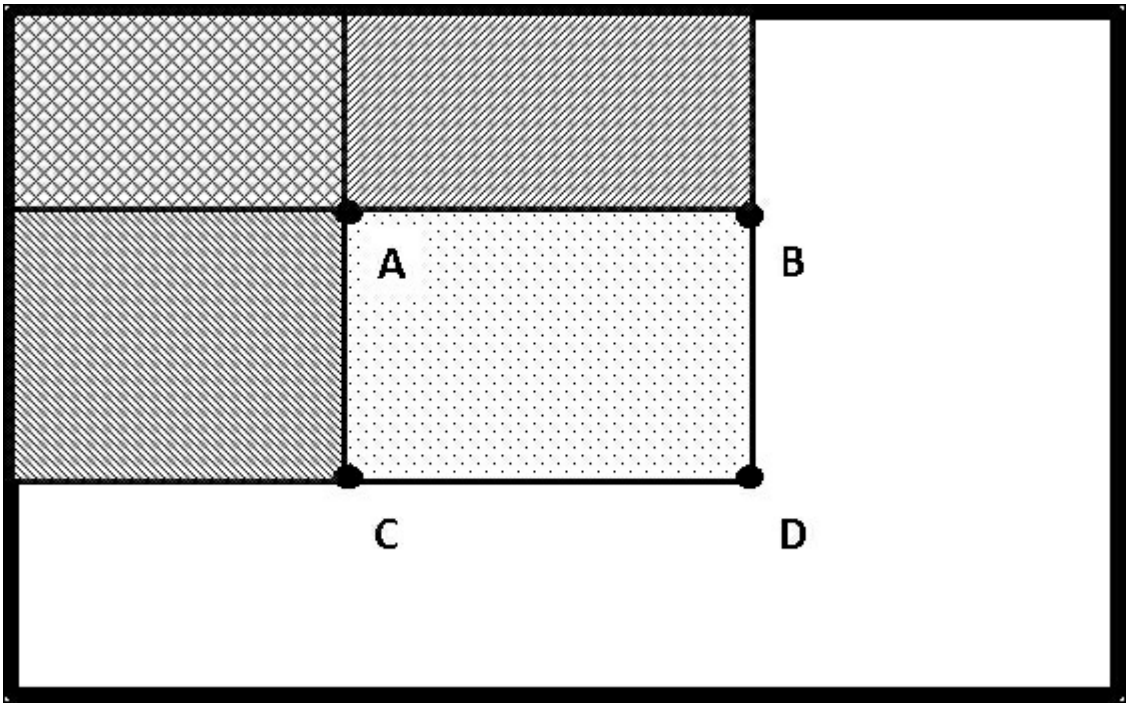
```
// Отримати суму по площі, використовуючи три дода-
вання                                    / віднімання
int                                              sumInt=
integralImage.at<int>(yo+height,xo+width)-
integralImage.at<int>(yo+height,xo)-
integralImage.at<int>(yo,xo+width)+
integralImage.at<int>(yo,xo);
```

Обидва підходи дають той самий результат. Проте обчислення інтегрального зображення є дорогим, оскільки доведеться прокрутити всі пікселі зображення.

Ключ полягає в тому, що як тільки це початкове обчислення буде зроблено, потрібно буде лише додати чотири значення, щоб отримати суму по регіону, що цікавить, незалежно від розміру цього регіону. Інтегральні зображення стають вигідними для використання тоді, коли декілька таких сум пікселів повинні бути обчислені для декількох областей різних розмірів.

Ми познайомилися з концепцією інтегральних зображень через коротку демонстрацію магії з ними, тобто, як вони можуть бути використані для дешевого обчислення суми пікселів у прямокутних областях. Щоб зрозуміти, як вони працюють, давайте тепер визначимо, що таке інтегральне зображення. Інтегральне зображення отримують шляхом заміни кожного пікселя на величину суми всіх пікселів, розташованих у верхньому лівому квадранті, розділених цим пікселем. Інтегральне зображення може бути обчислене шляхом одноразового сканування зображення. Дійсно, інтегральне значення поточного пікселя задається інтегральним значенням пікселя вище цього поточного пікселя плюс значення сукупної суми поточного рядка. Тому інтегральне зображення є новим зображенням, що містить суми пікселів. Щоб уникнути переповнення, це зображення зазвичай є зображенням значень `int (CV_32S)` або плаваючими значеннями `(CV_32F)`.

Наприклад, на наступному малюнку піксель A в цьому інтегральному зображенні міститиме суму пікселів, що містяться у верхньому лівому куті області, яка ідентифікована за допомогою схеми з подвійним штрихуванням:



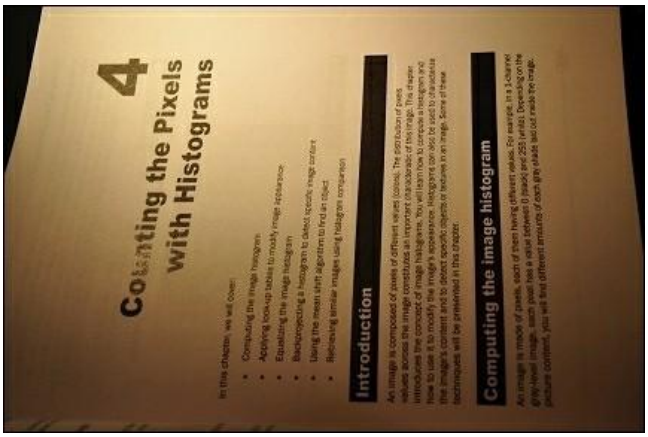
Як тільки інтегральне зображення було обчислено, будь-яке підсумовування над прямокутною областю може бути легко отримане через чотири піксельних доступи, і ось чому. Посилаючись на попередній рисунок, ми можемо побачити, що сума пікселів всередині області, розділеної пікселями **A**, **B**, **C** і **D**, може бути отримана зчитуванням інтегрального значення в пікселі **D**, з якого віднімаються значення пікселів над **B** і лівою частиною **C**. Однак, при цьому ми відняли двічі суму пікселів, розташованих у верхньому лівому куті **A**. Тому потрібно повторно додати інтегральну суму до **A**. Формально, тоді сума пікселів усередині **A**, **B**, **C** і **D** задається `A-B-C + D`. Якщо ми використовуємо метод `cv::Mat` для доступу до значень пікселів, ця формула перетворюється на наступне:

```
// вікно at (xo, yo) розмірної ширини по висоті
return(integralImage.at<cv::Vec<T,N>>(yo+height,xo
        +width)-integralImage.at<cv::Vec<T,N>>
        (yo+height,xo)-
        integralImage.at<cv::Vec<T,N>>
        (yo,xo+width)+
        integralImage.at<cv::Vec<T,N>>(yo,xo));
```

Тому складність цього обчислення є постійною, незалежно від розміру області, що цікавить. Зауважимо, що для простоти ми використовували метод `at` класу `cv::Mat`, що не є найефективнішим способом доступу до значень пікселів (див. лабораторну роботу *Маніпуляція пікселями*).

4.12 Адаптивний поріг

Застосування порога на зображенні для створення бінарного зображення може бути хорошим способом витягти змістовні елементи зображення. Припустимо, що у нас є наступне зображення книги:



Оскільки ми зацікавлені в аналізі тексту, застосуймо поріг до цього зображення наступним чином:

```
// З використанням фіксованого порогу
cv::Mat binaryFixed;
cv::threshold (image,binaryFixed,70,255,
                cv::THRESH_BINARY);
```

Ми отримаємо певний результат.

Насправді, незалежно від того, яке значення ми вибиремо для порога, в деяких частинах зображення отримаємо відсутній текст, тоді як в інших частинах текст зникає під тінью. Щоб подолати цю проблему, одне з можливих рішень полягає у використанні локальної порогової величини, яка обчислюється з околів кожного пікселя. Ця стратегія називається **адаптивним порогом**, і вона складається з порівняння кожного пікселя зі середнім значенням сусідніх пікселів. Пікселі, які явно відрізняються від їх місцевих значень, будуть розглядатися як відхилення і будуть обрізані процесом встановлення порогу.

Адаптивне порогове значення вимагає обчислення локальних значень навколо кожного пікселя. Це потребує підсумовування кількох зображень вікон, які можуть бути обчислені ефективно через інтегральне зображення. Отже, першим кроком є обчислення наступного інтегрального зображення:

```
// Обчислити цілісне зображення
cv::Mat iimage;
cv::integral(image,iimage,CV_32S);
```

Тепер можна пройти через всі пікселі і обчислити середнє значення за квадратом. Для цього можна використовувати наш клас `IntegralImage`, але цей метод неефективний для доступу до пікселів. На цей раз будемо ефективні, перебираючи зображення за допомогою по-

кажчиків, як робили в роботі *Маніпуляція пікселями*. Цей цикл виглядає наступним чином:

```
int blockSize= 21;  // розмір пікселя околиці
int threshold=10;   // буде порівняно
//(середній поріг)
                    // для кожного рядка
int halfSize= blockSize/2;
for (int j=halfSize; j<nl-halfSize-1; j++) {

// отримати адресу рядка j
uchar* data= binary.ptr<uchar>(j);
int* idata1= iimage.ptr<int>(j-halfSize);
int* idata2= iimage.ptr<int>(j+halfSize+1);

// Для кожного пікселя рядка
for (int i=halfSize; i<nc-halfSize-1; i++) {

// Обчислювальна сума
int   sum=   (idata2[i+halfSize+1]-data2[i-halfSize]-
idata1[i+halfSize+1] +idata1[i-halfSize])
/(blockSize*blockSize);

// Застосовувати адаптивний поріг
if (data[i]<(sum-threshold))
data[i]= 0; else
data[i]=255;
}
}
```

У цьому прикладі використовується околиця розміром `21x21`. Щоб обчислити кожне середнє, ми повинні отримати доступ до чотирьох інтегральних пікселів, які розділяють квадратне сусідство: два розташовані на лінії, вказаної `idata1`, і дві на лінії, вказаної `idata2`. Поточний піксель порівнюється з обчисленим середнім, з якого вираховується порогове значення (тут, встановлене на `10`). Це означає, що відхилені пікселі чітко відрізняються від їх місцевих значень.

В результаті отримаємо двійкове зображення, яке набагато краще, ніж те, яке ми отримали, використовуючи фіксований поріг. Адаптивне порогове значення є звичайною технікою обробки зображень. Як таке, воно також реалізується в OpenCV наступним чином:

```
cv::adaptiveThreshold(image,    // вхідне зображення
   binaryAdaptive,            // вихідного двійкового
   255,                         // зображення max
```

```
    cv::ADAPTIVE_THRESH_MEAN_C, // значення для методу
        cv::THRESH_BINARY,      // виведення порогового
        blockSize,              // типу розміру блоку
        threshold);             // порогового значення
```

Цей виклик функції дає точно той самий результат, який отримано за допомогою нашого інтегрального зображення. Крім того, замість того, щоб використовувати локальне значення для порогу, ця функція дозволяє використовувати гауссову зважену суму (прапором методу буде `cv::ADAPTIVE_THRESH_GAUSSIAN_C`) у цьому випадку. Цікаво відзначити, що наша реалізація трохи швидше, ніж виклик `cv::adaptiveThreshold`.

Нарешті, варто згадати, що можна також написати процедуру адаптивного порогу за допомогою операторів зображення OpenCV. Це буде зроблено наступним чином:

```
cv::Mat filtered;
cv::Mat binaryFiltered;
// Вікно фільтра обчислювального середнього
// пікселів над прямокутною областю
cv::boxFilter(image,filtered,CV_8U,cv::Size(blockSize
,blockSize
));
// Перевірити, якщо піксель більше
//(середнє + поріг)
binaryFiltered= image>= (filtered-threshold);
```

4.13 Візуальне відстеження з використанням гістограм

Як ми дізналися з попередніх підрозділів, гістограма являє собою надійне глобальне уявлення про вигляд об'єкта. У цьому підрозділі ми продемонструємо корисність інтегральних зображень, показуючи, як можна знайти об'єкт у зображенні шляхом пошуку області зображення, яка представляє гістограму, подібну до цільового об'єкта. Ми досягли цього, використовуючи *алгоритм середнього зсуву*, базуючись на понятті зворотної проекції гістограми та локальному пошуку через середній зсув. Цього разу ми знайдемо об'єкт, виконавши явний пошук областей подібних гістограм над повним зображенням.

У спеціальному випадку, коли інтегральне зображення використовується на бінарному зображенні з значень 0 і 1, інтегральна сума дає число пікселів, які мають значення 1 в заданій області. Використаємо цей факт, щоб обчислити гістограму зображення з градаціями сірого.

Функція `cv::integral` також працює з багатоканальними зображеннями. Можна скористатися цим фактом для обчислення гістограм суб-

регіонів зображень за допомогою інтегральних зображень. Нам просто потрібно перетворити зображення на багатоканальне з бінарних площин. Кожна з цих площин пов'язана з бункером нашої гістограми і показує, які пікселі мають значення, яке потрапляє до цього бункера. Наступна функція створює такі багатошарові зображення з градацій сірого:

```
// Перетворення в багатоканальне зображення з бінарних
// площин. nPlanes має бути ступенем 2
void convertToBinaryPlanes(const cv::Mat& input,
                                cv::Mat& output,
                                int nPlanes) {
      // кількість бітів для маскування
      int n= 8-static_cast<int>(
                    log(static_cast<double>
 (nPlanes))/log(2.0));
      // Маски, що використовуються для усунення
      // найменших значень бітів
      uchar mask= 0xFF<<n;

      // Створити вектор бінарних зображень
      std::vector<cv::Mat> planes;
    // Зменшити до nBins, усуваючи найменш значущі біти
      cv::Mat reduced= input&mask;

      // Обчислити кожну бінарну площину зображення
      for (int i=0; i<nPlanes; i++) {
        // Дорівнює 1 для кожного пікселя
  i << shift planes.push_back((reduced==(i<<n))&0x1);
      }

      // Створити багатоканальне зображення
      cv::merge(planes,output);
    }
```

Інтегральні обчислення зображень також можуть бути інкапсульовані в один зручний шаблонний клас наступним чином:

```
template <typename T, int N>
          class IntegralImage {

cv::Mat integralImage;
public:
IntegralImage(cv::Mat image) {

// (дороге) обчислення інтегрального зображення
cv::integral(image,integralImage,
```

```
                cv::DataType<T>::type);
}

// Обчислити суму за субрегіонами будь-якого
// розміру від доступу до 4 пікселів
cv::Vec<T,N> operator()(int xo, int yo,
                        int width, int height) {

// Вікно at (xo, yo) розмірної ширини по висоті
return (integralImage.at<cv::Vec<T,N>> (yo+height,
                                        xo+width)-
integralImage.at<cv::Vec<T,N>>(yo+height,xo)-
integralImage.at<cv::Vec<T,N>>(yo,xo+width)+
integralImage.at<cv::Vec<T,N>>(yo,xo));
}

};
```

Тепер ми хочемо знайти, де людина на велосипеді, яку ми позначили на попередньому зображенні, знаходиться в наступному зображенні. Давайте спочатку обчислимо гістограму людини у вихідному зображенні. Ми можемо виконати це за допомогою класу `Histogram1D`, який ми вбудували в підрозділі *Обчислюючи гістограму зображення*. Тут ми виробляємо 16-бітну гістограму наступним чином:

```
// Гістограма 16 бункерів
Histogram1D h;
h.setNBins(16);
// Обчислити гістограму зображення
roi cv::Mat refHistogram=h.getHistogram(roi);
```

Попередня гістограма буде використовуватися як опорне представлення для знаходження цільового об'єкта (людини на велосипеді) у наступному зображенні.

Припустимо, що єдина інформація, яку ми маємо, полягає в тому, що людина рухається більш-менш горизонтально по зображенню. Оскільки ми будемо мати багато гістограм для обчислення в різних місцях, обчислюємо інтегральне зображення, як на попередньому кроці:

```
// Спочатку створити 16-плоский бінарний образ
cv::Mat planes;
convertToBinaryPlanes(secondImage,planes,16);
// Потім обчислити цілісне зображення
IntegralImage<float,16> intHistogram(planes);
```

Щоб виконати пошук, ми прокручуємо діапазон можливих розташувань і порівнюємо поточну гістограму з опорною. Наша мета – знайти розташування з найбільш схожою гістограмою:

```
double maxSimilarity=0.0;
int xbest, ybest;
// Контур над горизонтальною смугою навколо
// розташування людини на початковому зображенні
for (int y=110; y<120; y++) {
for (int x=0; x<secondImage.cols-width; x++) {

// Обчислити гістограму з 16 бінів за допомогою
// цілісного зображення
histogram= intHistogram(x,y,width,height);
// Обчислити відстань від вихідної гістограми
double distance=cv::compareHist    (refHistogram,
         histogram, CV_COMP_INTERSECT);

// Знаходимо положення найбільш подібної гістограми
if (distance>maxSimilarity) {

xbest= x; ybest= y;
maxSimilarity= distance;
           }
     }
}
// намалюємо прямокутник у найкращому місці
cv::rectangle(secondImage,
           cv::Rect(xbest,ybest,width,height),0));
```

Програма визначить розташування з найбільш схожою гістограмою.

Білий прямокутник представляє область пошуку. Розраховано гістограми всіх вікон, які вміщуються в цій області. Ми зберігали постійну величину вікна, але це могло бути гарною стратегією, щоб також шукати трохи менші або більші вікна, щоб врахувати можливі зміни в масштабі. Зауважимо, що для того, щоб обмежити складність цього обчислення, кількість бункерів у гістограмах, які потрібно обчислити, повинна бути низькою. У нашому прикладі ми зменшили його до `16` бункерів.

Отже, площина 0 цього багатошарового зображення містить бінарне зображення, яке показує всі пікселі, які мають значення між `0` і `15`, а площина 1 показує нам пікселі зі значеннями від `16` до `31` і так далі.

Пошук об'єкта полягав у обчисленні гістограм всіх вікон заданого розміру в заданому діапазоні пікселів. Це являє собою обчислення `3200` різних гістограм, які були ефективно обчислені з нашого цілісного зображення. Всі гістограми, повернуті нашим класом `IntegralImage`, містяться в

об'єкті `cv::Vec` (через використання методу `at`). Потім ми використовуємо функцію `cv::compareHist` для ідентифікації найбільш схожої гістограми (пам'ятайте, що ця функція, як і більшість функцій OpenCV, може приймати об'єкт `cv::Mat` або `cv::Vec` через зручний тип `cv::InputArray`).

4.14 Контрольні запитання

1. Що таке гістограма зображення?
2. Що являє собою кумулятивна гістограма?
3. Як отримати білий або чорний піксель на кольоровому зображенні?
4. Що таке інтегральні зображення?

4.15 Хід роботи

4.15.1 Підготовчі стадії

Поза навчальною лабораторією треба зробити наступне:

1. Користуючись підрозділами 4.1 – 4.10, а також рекомендованою літературою, ознайомитися із складом та особливостями функцій для маніпуляцій з гістограмами в OpenCV.
2. Оформити першу частину звіту з виконання лабораторної роботи, в якій зазначити прізвище студента, мету лабораторних досліджень та дати письмові відповіді на контрольні запитання (п. 4.11).
3. Для виконання поставлених завдань необхідно більш детально розглянути принципи і основні етапи реалізації процедур для маніпуляцій з гістограмами в OpenCV на сайті [2].

4.15.2 Завдання до роботи

1. Уважно вивчити матеріали попередніх підрозділів, виконуючи наведені приклади обробки зображень і використовуючи власні зображення.
2. Побудуйте зображення гістограми у вигляді графіка для власного тестового зображення.
3. Підібравши поріг, отримайте двійкове зображення з сегментацією фону та передньої частини.
4. Використовуючи таблицю пошуку, зробіть негатив власного тестового зображення та побудуйте його гістограму.
5. Підвищить контраст власного тестового зображення за допомогою розтягування гістограми з відсотковим відсіченням `1…5%`. Побудуйте гістограму отриманого зображення.
6. Вирівняйте гістограму власного тестового зображення за допомогою власної функції OpenCV. Відобразіть результат та його гістограму.
7. За допомогою власної функції OpenCV для зворотного проектування гістограми виявіть певний вміст зображення. Побудуйте карту ймовірності та відобразіть її у негативному вигляді: від яскравого (низька ймові-

рність приналежності до опорної площі) до темного (висока ймовірність).

8. Застосуйте поріг до попереднього зображення та отримайте зображення з найбільш ймовірними пікселями опорної площі.

9. За допомогою власної функції OpenCV для зворотного проектування гістограми виявіть певний вміст кольорової версії зображення (наприклад, область блакитного неба).

10. Повторіть попередню вправу за допомогою обчислення розрідженої гістограми.

11. Проведіть експерименти, вирішуючи попередні задачі із застосуванням колірного простору HSV, `L*a*b*` або ін.

12. Застосовуючи алгоритмом середнього зсуву, з початкової прямокутної області (тобто положення особи людини на початковому зображенні), відтворіть прямий об'єкт на місці обличчя нової людини на іншовому зображенні.

13. Застосувати поріг до зображення книги.

14. Отримати двійкове зображення книги, використовуючи адаптивний поріг.

15. Визначити розташування певного обєкту на зображенні, відстежуючи його за допомогою гістограм.

4.16 Вимоги до звіту по роботі

Звіт про виконання роботи повинен містити:

1. Прізвище та групу студента, а також назву, мету та завдання лабораторної роботи.

2. Відповіді на контрольні питання.

3. Таблицю з переліком функцій *OpenCV* для маніпуляцій з гістограмами.

4. Короткий перелік основних дій в процесі виконання завдань.

5. Текст розроблених програм з коментарями.

6. Скрін-шоти роботи розроблених програм.

7. Висновки, в яких треба зазначити, чи досягнута мета даної лабораторної роботи. Які саме конкретно маніпуляції з гістограмами дозволяє (перерахувати по пунктах) робити *OpenCV*? Які конкретно практичні навички роботи з зображеннями здобуті в ході дослідження?

8. Електронну версію звіту з лабораторної роботи у форматі 2020_Прізвище-ЛР4.doc або *.docx разом з додатками (програмними проектами в архіві) треба завантажити у відповідному розділі сторінки дисципліни в системі дистанційного навчання “Moodle”.

4.17 Орієнтовні варіанти завдань

Завдання за варіантами видаються викладачем після виконання студентом підготовчої стадії (п. 4.15.1), типових завдань (п. 4.15.2), здавання звіту та його перевірки.

5 Лабораторна робота №5. Оцінка проективних відносин в зображеннях

Мета роботи: дослідження обчислення фундаментальної матриці пари зображень, співпадіння зображень з використанням консенсусу випадкових вибірок, обчислення гомографії між двома зображеннями, виявлення планарної цілі на зображеннях

5.1 Формування зображення

Зображення, як правило, створюються за допомогою цифрової камери, яка захоплює сцени, проектуючи світло, що проходить через її об'єктив, на датчик зображення.

Той факт, що зображення формується проекцією 3D-сцени на 2D площину, зумовлює наявність важливих зв'язків як між сценою та її зображенням, так і між різними зображеннями однієї сцени.

Проекційна геометрія є інструментом, який використовується для опису та характеристики в математичних термінах процесу формування образу. У цій лабораторній роботі ми ознайомимося з деякими фундаментальними проективними відносинами, які існують у зображеннях з кількома видами, і дослідимо, яким чином їх можна використовувати для програмування комп'ютерного зору. Спочатку розглянемо основні поняття, пов'язані з проекцією сцени і формуванням зображення.

Процес, який використовується для створення зображень, принципово не змінився з часів початку фотографії. Світло, що надходить від об’єкту спостереження, захоплюється камерою через фронтальну діафрагму, і світлові промені потрапляють на площину зображення (або датчик зображення), розташовану в камері. Крім того, для концентрування променів від різних елементів сцени використовується об’єктив. Цей процес ілюструється рисунком 5.1.

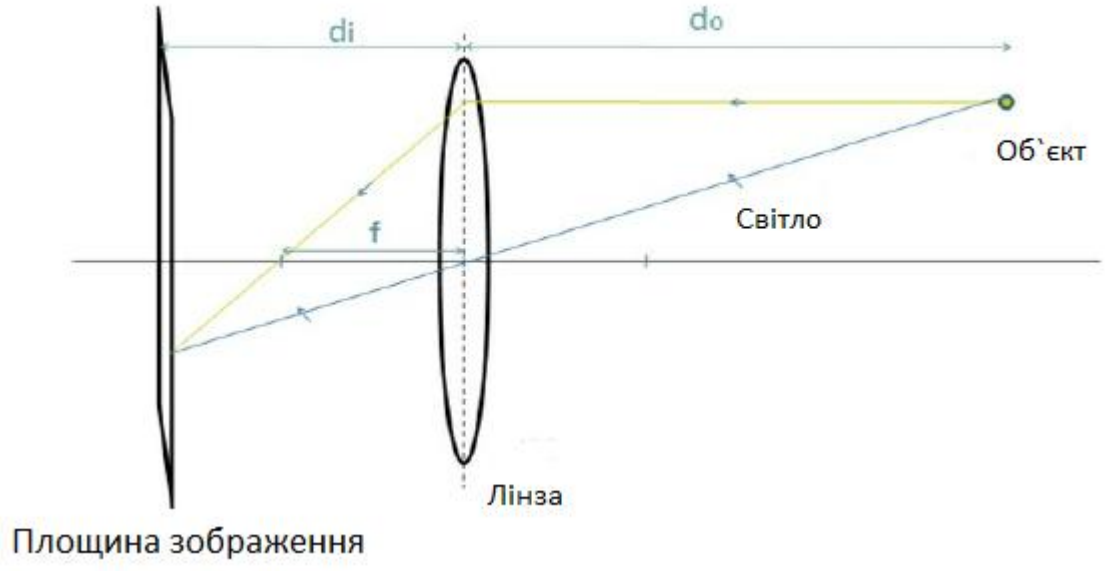


Рисунок 5.1 – Процес створення зображень

ут, *do* – відстань від об'єктива до спостережуваного об'єкта, *di* – відстань від лінзи до площини зображення, а *f* – фокусна відстань лінзи. Ці величини пов'язані з так званим **рівнянням тонких лінз:**

$$\frac{1}{f} = \frac{1}{do} + \frac{1}{di}. \quad (5.1)$$

У комп'ютерному зорі ця модель камери може бути спрощена кількома способами. По-перше, можна нехтувати ефектами об’єктива, вважаючи, що наша камера має нескінченно малу апертуру, яка, теоретично, не змінює зовнішній вигляд зображення. Однак, створивши зображення з нескінченною глибиною, ми ігноруємо ефект фокусування. В даному випадку розглядається лише центральний промінь. По-друге, оскільки більшу частину часу у нас *do >> di*, можна припустити, що площина зображення розташована на фокусній відстані. Нарешті, відповдно до геометрії системи, зображення на площині інвертується. Можна отримати ідентичний, але вертикальний образ, просто позиціонуючи площину зображення перед об'єктивом. Очевидно, це фізично неможливо, але з математичної точки зору повністю рівносильно. Цю спрощену модель часто називають **моделлю камери з** точковим **отвором**, яка представлена наступним чином:

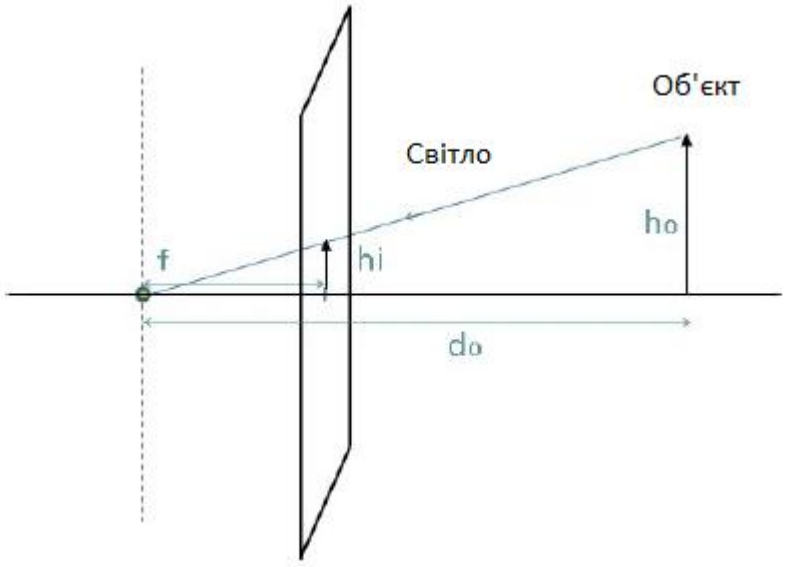


Рисунок 5.2 – Модель камери з точковим отвором

З цієї моделі, використовуючи закон подібних трикутників, можна легко вивести базове проективне рівняння, яке пов'язує сфотографований об'єкт з його зображенням:

$$hi = f\frac{ho}{do}$$

Розмір *(hi)* зображення об'єкта (фізичної висоти *ho*) обернено пропорційний відстані *(do*) від камери, що є природно вірним. Загалом, це спів-

відношення описує, де 3D-точка сцени буде проектуватися на площину зображення з урахуванням геометрії камери. Більш конкретно, якщо припустити, що опорний кадр розташований у точці фокусу, то точка 3D-сцени, розташована в положенні `(X,Y,Z)` проектуються на площину зображення при `(x,y)=(fX/Z,fY/Z)`. Тут координата `Z` відповідає глибині точки (або відстані до камери, позначеній *do* в попередньому рівнянні). Це відношення можна переписати в простій матричній формі через введення однорідних координат, в яких 2D-точки представлені 3-вимірними, а 3D-точки – 4-вимірними векторами. (Додаткова координата є просто довільним масштабним коефіцієнтом `s`, який необхідно буде видалити, коли 2D-координати потрібно витягти з однорідного 3-вимірного вектора):

$$s\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Цю матрицю `3x4` називають *проекційною матрицею*. У тих випадках, коли опорний кадр не вирівняний з фокальною точкою, треба ввести матриці обертання `r` і трансляції `t`. Їх роль – просто виразити проектовану 3D–точку в камеро-центрічній системі координат, що виглядає наступним чином:

$$s\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r1 & r2 & r3 & t1 \\ r4 & r5 & r6 & t2 \\ r7 & r8 & r9 & t3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Перша матриця цього рівняння містить, як кажуть, власні (intrinsic) параметри камери (тут – тільки фокусна відстань, але у наступному розділі введемо ще кілька внутрішніх параметрів). Друга матриця містить зовнішні (extrinsic) параметри, які пов'язують камеру з зовнішнім світом. Слід зазначити, що на практиці координати зображення виражаються в пікселях, а 3D-координати виражаються у світових вимірах (наприклад, метрах). Цей аспект розглянуто в лабораторній роботі *Реконструкція 3D-сцен.*

5.2 Обчислення фундаментальних матриць пари зображень

У першому підрозділі представлене рівняння, що описує, як точка сцени проектується на площину зображення однієї камери. У цьому підрозділі дослідимо проективні відносини, що існують між двома зображеннями, які відображають ту ж саму сцену.

Ці два зображення могли бути отримані шляхом переміщення камери на два різних місця, щоб фотографувати або з двох точок зору, або за допомогою двох камер, кожна з яких робить іншу картинку сцени. Коли ці

дві камери розділені жорсткою базовою лінією, використовується термін **стереобачення.**

Розглянемо дві камери, що спостерігають за певною точкою сцени, як показано на наступному рисунку 5.3.

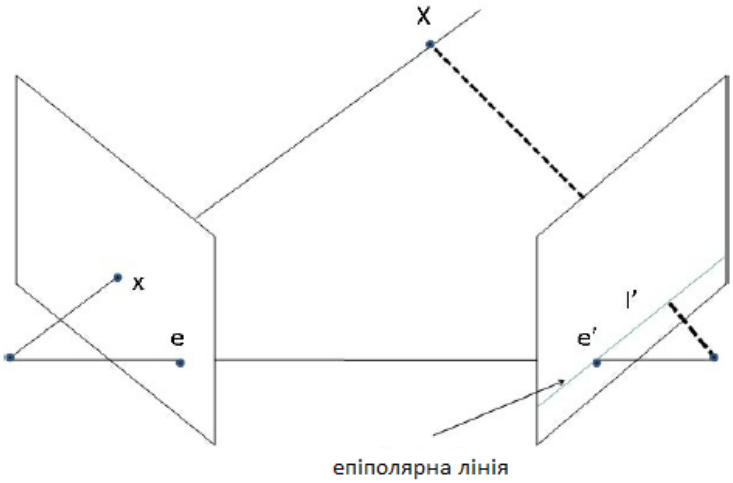


Рисунок 5.3 – Спостерігання за двома об’єктами двома камерами

Знайти зображення *x* 3D-точки *X* можна, проектуючи лінію, що поєднує цю 3D-точку з центром камери. І навпаки, точка сцени, яка має своє зображення в положенні **x** на площині зображення може бути розташована будь-де на цій лінії в 3D-просторі. Це означає, що, якщо треба знайти відповідну до даної точку зображення на іншому зображенні, маємо шукати вздовж проекції цієї лінії на площину другого зображення. Ця уявна лінія називається **епіполярною лінією** точки *x*. Це визначає фундаментальне обмеження, якому мають задовольняти дві відповідні точки, а саме: відповідність заданої точки повинна лежати на епіполярній лінії іншого погляду. А точна орієнтація цієї епіполярної лінії залежить від відповідного положення двох камер. Фактично конфігурація безлічі можливих епіполярних ліній характеризує геометрію двопоглядової системи.

Інше спостереження, яке можна зробити з геометрії цієї двопоглядової системи полягає в тому, що всі епіполярні лінії проходять через одну і ту ж точку. Ця точка відповідає проекції центру однієї камери на іншу камеру (точки *e* і *e'* на рисунку). Ця особлива точка називається **епіполем.**

Математично взаємозв'язок між точкою зображення та відповідною епіполярною лінією може бути виражений з використанням матриці 3x3 наступним чином:

$$\begin{bmatrix} l_1' \\ l_2' \\ l_3' \end{bmatrix} = F \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

У проективній геометрії 2D-лінія також представлена 3-вимірним ве-

ктором. Це відповідає множині 2D-точок (x', y'), які задовольняють рівнянню l_1'x' + l_2'y' + l_3'= 0 (апостроф означає, що ця лінія належить до другого зображення). F називається фундаментальною матрицею. Вона відображає 2D-точку зображення в одному вигляді на епіполярну лінію в іншому вигляді.

Фундаментальну матрицю пари зображень можна знайти, вирішивши набір рівнянь, які включають певну кількість відомих співпадаючих точок двох зображень. Мінімальна кількість таких збігів – сім. Щоб проілюструвати процес оцінки фундаментальної матриці, можна вибрати сім хороших збігів результатів співпадіння особливостей методом виявлення точок інтересу SIFT.

Ці збіги будуть використовуватися для обчислення фундаментальної матриці за допомогою функції cv::findFundamentalMat OpenCV. Ці збіги зберігаються у векторі cv::DMatch, що вказує на індекси екземплярів cv::keypoint. Спочатку ці ключові точки потрібно конвертувати в cv::Point2f для використання з cv::findFundamentalMat. Для цього можна використовувати функцію OpenCV:

```
// Перетворення ключових точок у Point2f
std::vector<cv::Point2f> selPoints1, selPoints2;
std::vector<int> pointIndexes1, pointIndexes2;
cv::KeyPoint::convert(keypoints1,selPoints1,pointInde
xes1);
cv::KeyPoint::convert(keypoints2,selPoints2,pointInde
xes2);
```

Два результуючих вектора selPoints1 і selPoints2 містять відповідні координати точок у двох зображеннях. Вектори pointIndexes1 і pointIndexes2 містять індекси ключових точок, які треба перетворити. Далі виклик функції cv::findFundamentalMat виконується наступним чином:

```
// Обчислити F матрицю з 7 збігів
cv::Mat fundamental= cv::findFundamentalMat(
          selPoints1, // 7 точок у першому зображенні
          selPoints2, // 7 точок у другому зображенні
     cv::FM_7POINT); // метод семи точок
```

Один із способів візуально перевірити достовірність фундаментальної матриці – намалюйте епіполярні лінії деяких вибраних точок. Інша OpenCV функція дозволяє епіполярним лініям заданого набору точок бути обчисленими. Як тільки вони були обчислені, вони можуть бути намальовані за допомогою функції cv::line. Наступні рядки коду виконують ці два кроки (тобто обчислення і малювання епіполярних ліній на зображенні на праворуч від точок на малюнку ліворуч):

```
// Намалювати ліві точки відповідно епіполю
// лінії в правому зображенні
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(
     selPoints1, // image points
     1, // in image 1 (can also be 2)
     fundamental, // F matrix
     lines1); // vector of epipolar lines
// для всіх єпіполярніх точочк
for (vector<cv::Vec3f>::const_iterator it=
lines1.begin();
     it!=lines1.end(); ++it) {
// малювання лінії меж двумя стовпцями
cv::line(image2, cv::Point(0,-(*it)[2]/(*it)[1]),
     cv::Point(image2.cols,
          -((*it)[2]+(*it)[0]*image2.cols)/(*it)
[1]),
          cv::Scalar(255,255,255));
          }
```

Аналогічним чином отримують епіполярні лінії лівого зображення. Наступне зображення показує це.

Рисунок 5.4 – Процес отримання епіполярних ліній зображення

Пам'ятайте, що епіполь одного зображення знаходиться на перетині всіх його епіполярних ліній. Це – проекція центру іншої камери. Зауважимо, що епіполярні лінії можуть перетинатися (і часто) за межами межі зображення. У випадку нашого прикладу, епіполь другого зображення знаходиться в місці, де перша камера буде видима, якщо два зображення були

зроблені в одну мить. Зауважимо також, що результати можуть бути досить нестабільними, коли фундаментальна матриця обчислюється тільки з семи-збігів. Дійсно, підставивши один збіг на інший, це може призвести до суттєво різниому набору епіполярних ліній.

Раніше ми пояснювали, що для точки в одному зображенні фундаментальна матриця дає рівняння лінії, на якій має бути знайдена відповідна їй точка в іншому вигляді. Якщо відповідна до (x, y) точка є (x', y'), припустимо, що F має фундаментальну матрицю між двома видами. Оскільки (x', y') лежить на епіполярній лінії, заданій множенням F на (x, y), і яку виражено в однорідних координатах, маємо наступне рівняння:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}^T F \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = 0$$

Це рівняння виражає зв'язок між двома відповідними точками і відоме як **епіполярне обмеження**. Використання цього рівняння дає можливість оцінювати записи матриці за допомогою відомих збігів. Оскільки записи матриці F подаються з масштабним коефіцієнтом, оцінюються лише вісім записів (дев'ятий може бути довільно встановлено на 1). Кожний збіг дає одне рівняння. Тому, маючи вісім відомих збігів, матрицю можна повністю оцінити, вирішивши отриманий набір лінійних рівнянь. Це робиться, коли використовується прапор `cv::FM_8POINT` з функцією `cv::findFundamentalMat`. Зауважимо, що в цьому випадку можна (і краще) ввести більше восьми збігів. Отримана завищена система лінійних рівнянь може бути вирішена в середньоквадратичному сенсі.

Для оцінки фундаментальної матриці також може бути використано додаткове обмеження. Математично матриця F відображає 2D-точку у 1D-олівець ліній (тобто лінії, які перетинаються в спільній точці). Той факт, що всі ці епіполярні лінії проходять через цю унікальну точку (тобто епіполь) збігів, необхідні для оцінки фундаментальної матриці до семи. У математичних термінах ми говоримо, що фундаментальна матриця має 7 ступенів свободи і тому має ранг 2. На жаль, в даному випадку набір рівнянь стає нелінійним, з трьома можливими рішеннями (в даному випадку `cv::findFundamentalMat` поверне фундаментальну матрицю розміром `9x3`, тобто складаються три матриці `3x3`). Семизбігове рішення оцінки матриц F і може бути викликане в OpenCV за допомогою прапора `cv::FM_7POINT`. Це те, що ми зробили в прикладі в попередньому підрозділі.

Нарешті, слід зазначити, що вибір відповідного набору збігів на зображенні важливий для отримання точної оцінки фундаментальної матриці.

Взагалі, збіги повинні бути добре розподілені по зображеннях і включати точки на різних глибинах сцени. Інакше рішення стане нестабільним. Зокрема, вибрані точки сцени не повинні бути компланарними, оскільки фундаментальна матриця (в даному випадку) стає виродженою.

5.3 Співпадіння зображень за допомогою консенсусу випадкових зразків

Коли дві камери спостерігають одну і ту ж сцену, вони бачать однакові елементи але під різними точками зору. Ми вже вивчили цю функцію проблеми зіставлення точок у попередньому розділі. У цьому розділі ми приходимо до цієї проблеми, і ми навчимося експлуатувати епіполяр обмеження, введений в попередній роботі, щоб відповідати характеристикам зображення більш надійно.

Принцип, якого ми будемо дотримуватися, є простим: коли ми співпадаємо з функцією точок між двома зображеннями, ми приймаємо тільки ті збіги, на які падають відповідні епіполярні лінії. Однак, щоб мати можливість перевірити ці умови, фундаментальна матриця повинна бути відома, але нам потрібні чіткі збіги для оцінки цієї матриці. У цьому підрозділі ми пропонуємо рішення, в якому фундаментальна матриця і набір хороших збігів будуть обчислюватися спільно.

Мета полягає в тому, щоб мати можливість обчислити фундаментальну матрицю і набір коректних збігів між двома переглядами. Для цього всі знайдені точки відповідності будуть перевірені з використанням епіполярного обмеження, введеного в попередньому підрозділі. Для цього ми створили клас, що інкапсулює різні етапи запропонованого надійного узгодження процесу:

```
class RobustMatcher {
 private:
// детектор точок
cv::Ptr<cv::FeatureDetector> detector;
cv::Ptr<cv::DescriptorExtractor> descriptor;
int normType;
float ratio; // максимальне співвідношення між 1 і 2
NN
bool refineF; // якщо вірно буде уточнено матрицю F
bool refineM; // якщо вірно буде уточнено збіги
double distance; // мінімальна дистанція до епіполяру
double confidence; // рівень довіри (ймовірність)
public:
RobustMatcher(const cv::Ptr<cv::FeatureDetector>
&detector,
const cv::Ptr<cv::DescriptorExtractor>
&descriptor=
cv::Ptr<cv::DescriptorExtractor>
()):
```

```
detector(detector), descriptor(descriptor),
normType(cv::NORM_L2), ratio(0.8f),
refineF(true), refineM(true),
confidence(0.98), distance(1.0) {
// в цьому випадку використовуйте відповідний дескри-
птор
if (!this->descriptor) {
this->descriptor = this->detector;
     }
}
```

Користувачі цього класу просто надають детектор і дескриптор ознак їх вибору. Це можна також вказати за допомогою відомих `setFeatureDetector` і `setDescriptorExtractor setter` методів.

Основним методом є метод відповідності, який повертає знайдені збіги ключових точок і оціночної фундаментальної матриці. Метод продовжується у чотирьох різних кроках (чітко визначених у коментарях до наступного коду), який ми зараз розглянемо:

```
// Поєднуйте точки особливості за допомогою RANSAC
// повертає фундаментальну матрицю і набір вихідних
збігів
cv::Mat match(cv::Mat& image1, cv::Mat& image2, //
введення зображення
std::vector<cv::DMatch>& matches, //
виведення зображення
std::vector<cv::KeyPoint>&
keypoints1,//output keypoints
std::vector<cv::KeyPoint>& keypoints2) {
// 1. Виявлення точок особливості
detector->detect(image1,keypoints1);
detector->detect(image2,keypoints2);
// 2. Вилучення дескрипторів ознак
cv::Mat descriptors1, descriptors2;
descriptor->compute(image1,keypoints1,descriptors1);
descriptor->compute(image2,keypoints2,descriptors2);
// 3. Підібрати два дескриптора зображення
// (необов'язково застосувати певний метод перевірки)
// Будівництво збірки з перевіркою
cv::BFMatcher matcher(normType, // відстань
true); // прапорець
// дескриптори відповідності
std::vector<cv::DMatch> outputMatches;
matcher.match(descriptors1,descriptors2,outputMatches
);
// 4. Перевірка відповідності з використанням RANSAC
```

```
cv::Mat fundamental= ransacTest(outputMatches,
keypoints1, keypoints2,
                matches);
     // повернення знайденої фундаментальної матриці
     return fundamental;
     }
```

Перші два кроки просто виявляють точки обчислень і обчислюють їх дескриптори. Далі ми переходимо до відповідності функції, використовуючи `cv::BFMatcher`, як і в попередньому підрозділі. Ми використовуємо прапорець, щоб отримати збірки кращої якості.

Четвертий крок – нова концепція, введена в цьомц підрозділі. Він складається з додаткового тесту фільтрації, який цього разу буде використовувати фундаментальну матрицю для того, щоб відхилити збіги, які не підкоряються епіполярному обмеженню. Цей тест заснований на методі `RANSAC`, який може обчислити фундаментальну матрицю, навіть якщо у збігі встановлено винятки (цей метод буде пояснено в наступному підрозділі).

Використовуючи наш клас `RobustMatcher`, надійне узгодження пари зображень потім легко виконується наступними викликами:

```
// Підготувати збіг (з параметрами за замовчуванням)
// SIFT-детектор і дескриптор
RobustMatcher
rmatcher(cv::xfeatures2d::SIFT::create(250));
// Підібрати два зображення
std::vector<cv::DMatch> matches;

std::vector<cv::KeyPoint> keypoints1, keypoints2;
cv::Mat fundamental = rmatcher.match(image1, image2,
          matches,
          keypoints1,
     keypoints2);
```

Це призводить до *54* збігів, які показані на наступному зображенні:

Рисунок 5.5 – Процес виявлення збігів на зображеннях

У більшості випадків результати збігів будуть хорошими. Однак, може залишитися кілька помилкових збігів; це ті, які випадково впали відповідні епіполярні лінії обчисленої фундаментальної матриці.

У попередньому прикладі ми дізналися, що фундаментальну матрицю, пов'язану з парою зображень, можна оцінити, виходячи з числа збігів особливих точок. Очевидно, щоб бути точним, цей набір збігів повинен складатися тільки з хороших збігів. Проте, в реальному контексті, неможливо гарантувати, що набір збігів, отриманий шляхом порівняння дескрипторів виявлених точок ознаки, буде повністю точним. Саме тому був введений метод оцінки фундаментальної матриці, заснований на стратегії **RANSAC** (**RANdom SAmpling Consensus** – консенсус випадкових вибірок).

Алгоритм RANSAC має на меті оцінити заданий математичний об'єкт з набору даних, який може містити ряд викидів. Ідея полягає в тому, щоб випадковим чином виділити деякі точки даних з набору і виконати оцінку тільки з ними. Кількість вибраних точок повинна бути мінімально необхідною для оцінки математичної сутності. У випадку фундаментальної матриці мінімальним числом є вісім пар збігів (насправді реальний мінімум – сім відповідностей, але лінійний алгоритм з 8 точками обчислюється швидше). Як тільки фундаментальну матрицю оцінено з цих восьми випадкових збігів, всі інші збіги в наборі збігів перевіряються проти епіполярного обмеження, яке випливає з цієї матриці. Всі збіги, які виконують це обмеження (тобто збіги, для яких відповідна ознака знаходиться на невеликій відстані від її епіполярної лінії), ідентифікуються. Ці збіги утворюють набір підтримки обчисленої фундаментальної матриці.

Центральна ідея алгоритму RANSAC полягає в тому, що чим більше набір підтримки, тим вище ймовірність того, що обчислена матриця є правильною. І навпаки, якщо один (або більше) випадково вибраних збвгів є неправильним, то обчислена фундаментальна матриця також буде неправильною, і очікується, що її набір підтримки буде малим. Цей процес повторюється кілька разів, і в кінцевому підсумку матриця з найбільшою підтримкою буде збережена як найбільш вірогідна.

Таким чином, наша мета полягає в тому, щоб підібрати вісім випадкових збігів кілька разів, так що в кінцевому підсумку ми виберемо вісім хороших, які повинні дати нам великий набір підтримки. Залежно від частки неправильних збігів у всьому наборі даних, ймовірність вибору набору з восьми правильних збігів буде відрізнятися. Однак, ми знаємо, що чим більше виборів ми робимо, тим вища наша впевненість у тому, що ми маємо принаймні один хороший збіг серед цих виборів. Точніше, якщо припустити, що набір збігів зроблений з `w%` вдалих (хороші збіги), то ймовірність того, що ми виберемо вісім гарних збігів, є `w8`. Отже, ймовірність того, що вибір містить принаймні один неправильниц збіг `(1-w8)`. Якщо ми робимо `k` виборів, то ймовірність наявності одного випадкового набору, що містить тільки хороші збіги, є `1-(1-w8)k`. Це ймовірність довіри `c`, і ми хочемо, щоб ця ймовірність була якомога вищою, оскільки нам потрібний принаймні один хороший набір збігів для того, щоб отримати правильну фундаментальну матрицю. Отже, при виконанні алгоритму RANSAC необхідно визначити кількість `k` виборів, які необхідно зробити, щоб отримати заданий рівень достовірності.

Використання методу RANSAC для оцінки фундаментальної матриці здійснюється в методі `ransacTest` нашого класу `RobustMatcher`:

```
// Identify good matches using RANSAC
// Return fundamental matrix and output matches
cv::Mat ransacTest(const std::vector<cv::DMatch>&
matches,
std::vector<cv::KeyPoint>& keypoints1,
std::vector<cv::KeyPoint>& keypoints2,
std::vector<cv::DMatch>& outMatches) {
// Convert keypoints into Point2f
std::vector<cv::Point2f> points1, points2;
for (std::vector<cv::DMatch>::const_iterator it=
matches.begin();
it!= matches.end(); ++it) {
// Get the position of left keypoints
points1.push_back(keypoints1[it->queryIdx].pt);
// Get the position of right keypoints
points2.push_back(keypoints2[it->trainIdx].pt);
}
// Compute F matrix using RANSAC
```

```
std::vector<uchar> inliers(points1.size(),0);
cv::Mat fundamental=
cv::findFundamentalMat( points1,
points2, // matching points
inliers, // match status (inlier
or outlier)
cv::FM_RANSAC, // RANSAC method
distance, // distance to epipolar
line
confidence); // confidence
probability
// extract the surviving (inliers) matches
std::vector<uchar>::const_iterator itIn= inli-
ers.begin();
std::vector<cv::DMatch>::const_iterator itM=
matches.begin();
// for all matches
for ( ;itIn!= inliers.end(); ++itIn, ++itM) {
if (*itIn) { // it is a valid match
outMatches.push_back(*itM);
}
}
return fundamental;
}
```

Цей код трохи довгий, тому що ключові точки необхідно перетворити в `cv::Point2f` перед обчисленням матриці F. При використанні функції `cv::findFundamentalMat` з методом `cv::FM_RANSAC` передбачено два додаткові параметри. Одним з таких додаткових параметрів є рівень довіри, який визначає кількість ітерацій, які необхідно виконати (за замовчуванням – `0,99`). Інший параметр – це максимальна відстань до епіполярної лінії для точки, яка вважається inlier. Всі пари збігів, в яких точка знаходиться на більшій відстані від своєї епіполярної лінії, ніж вказана відстань, будуть розглядатися як викид. Функція також повертає `std::vector` символьного значення, що вказує, що відповідний збіг у вхідному наборі був ідентифікований або як вихідний (`0`), або як вхідний (`1`). Це пояснює останній цикл нашого методу, який витягує хороші збіги з початкового набору збігів.

Чим більше хороших збігів у початковому наборі, тим вище ймовірність того, що RANSAC дасть правильну фундаментальну матрицю. Ось чому ми застосовували фільтр перехресної перевірки при узгодженні точок об'єкта. Ви також могли б використати тест відношення, представлений у попередньому прикладі, щоб ще більше поліпшити якість остаточного набору збігів. Це лише питання балансування обчислювальної склад-

ності, кінцевої кількості збігів і необхідного рівня впевненості, що отриманий набір збігів буде містити тільки точні збіги.

Результатом надійного процесу співпадіння, представленого в цьому прикладі, є: 1) оцінка фундаментальної матриці, обчисленої з використанням восьми вибраних збігів, які мають найбільшу підтримку і 2) набір збігів, включених до цього набору підтримки. Використовуючи цю інформацію, можна уточнити ці результати двома способами.

5.3.1 Уточнення фундаментальної матриці

Оскільки тепер у нас є набір хороших показників якості, як останній крок, може бути гарною ідеєю використовувати їх усі для повторної оцінки фундаментальної матриці. Ми вже згадували, що існує лінійний 8-точковий алгоритм для оцінки цієї матриці. Тому ми можемо отримати надмірну систему рівнянь, яка вирішить фундаментальну матрицю в сенсі найменших квадратів. Цей крок можна додати до кінця нашої функції `ransacTest`:

```
// Convert the keypoints in support set into Point2f
points1.clear();
points2.clear();
for (std::vector<cv::DMatch>::const_iterator it=
outMatches.begin();
it!= outMatches.end(); ++it) {
// Get the position of left keypoints
points1.push_back(keypoints1[it->queryIdx].pt);
// Get the position of right keypoints
points2.push_back(keypoints2[it->trainIdx].pt);
}
// Compute 8-point F from all accepted matches
fundamental= cv::findFundamentalMat(
points1,points2, // matching points
cv::FM_8POINT); // 8-point method solved
using SVD
```

Функція `cv::findFundamentalMat` дійсно приймає більше 8 збігів, вирішуючи лінійну систему рівнянь, використовуючи сингулярне розкладання.

5.3.2 Уточнення співпадінь

Ми дізналися, що в системі двох поглядів кожна точка повинна лежати на епіполярній лінії відповідної точки. Це епіполярне обмеження, виражене фундаментальною матрицею. Отже, якщо є гарна оцінка фундаментальної матриці, можна використовувати це епіполярне обмеження для виправлення отриманих збігів, змушуючи їх лежати на своїх епіполярних лініях. Це можна легко зробити за допомогою функції `cv::correctMatches` OpenCV:

```
std::vector<cv::Point2f> newPoints1, newPoints2;
```

```
// refine the matches
correctMatches(fundamental, // F matrix
points1, points2, // original
position
newPoints1, newPoints2); // new position
```

Ця функція продовжується шляхом зміни положення кожної відповідної точки таким чином, щоб вона задовольняла епіполярному обмеженню при мінімізації кумулятивного (квадратного) зміщення.

5.4 Обчислення гомографії між двома зображеннями

Перша частина цієї глави показала нам, як обчислити фундаментальну матрицю пари зображень з набору збігів. У проективній геометрії, інша дуже корисна математична сутність також існує. Це можна обчислити з зображень з декількома переглядами і, як ми побачимо, це матриця зі спеціальними властивостями.

Знову ж таки, розглянемо проективне відношення між 3D-точкою та її зображенням на камеру, яку ми представили у вступі розділу цієї глави. В основному, ми дізналися, що це рівняння стосується 3D точки до його образу використовуючи властивості камери і положення цієї камери (задається обертанням і перекладацькою складовою).

Якщо ми тепер уважно вивчимо це рівняння, то зрозуміємо, що існує два особливих ситуації. Перша ситуація, коли два погляди сцени розділені чистою ротацією. Потім ми можемо спостерігати що четверта колонка зовнішньої матриці буде складатися з *0*s (що перекладається як нульовий):

$$s\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r1 & r2 & r3 & 0 \\ r4 & r5 & r6 & 0 \\ r7 & r8 & r9 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Внаслідок цього проективне співвідношення в цьому особливому випадку стає *3x3* матрицею. Аналогічно цікава ситуація також виникає, коли об'єкт ми спостерігати – площина. У цьому конкретному випадку можна припустити без втрати спільність, що точки на цій площині будуть розташовані при $Z = 0$. Як результат, отримуємо наступне рівняння:

$$s\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r1 & r2 & r3 & t1 \\ r4 & r5 & r6 & t2 \\ r7 & r8 & r9 & t3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ 0 \\ 1 \end{bmatrix}$$

Ця нульова координата точок сцени скасує третій стовпцець проективної матриці, яка потім знову стане *3x3* матрицею. Ця спеціальна матриця називається **гомографією**, і це означає, що при особливих обставинах (тут чисте обертання або планарний об'єкт), світова точка може бути пов'язана з її зображенням за допомогою лінійного відношення. В додаток, оскільки ця матриця є обертовою, ви також можете зв'язати точку зображення один погляд безпосередньо на його відповідну точку на інший погляд, враховуючи ці два погляди відокремлюються чистим обертанням або відображають площину об'єкту. Тоді гомографічне співвідношення має такий вигляд:

$$s\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Тут *H* є матрицею *3x3*. Це відношення до масштабного коефіцієнта представлені тут скалярним значенням *S*. Як тільки ця матриця оцінюється, всі точки в одному перегляді можуть бути перенесені на другий вигляд за допомогою цього відношення. Це властивість, яка буде використана в цьому розділі і наступному. Зауважимо, що, як побічний ефект співвідношення гомографій, фундаментальна матриця в цих випадках стає невизначеною.

Припустимо, що у нас є два зображення, розділені чистим обертанням. Це трапляється, наприклад, при фотографуванні будівлі або краєвиду, обертаючи себе; коли ви досить далеко до об’єкту, а трансляційний компонент є, в даному випадку, незначним. Ці два зображення можна підібрати за допомогою функцій. Нашим вибором є функція `cv::BFMatcher`.

Результат такий:

Рисунок 5.6 – Процес підбору збігів двох зображень

Тоді, як і в попередньому прикладі, ми застосуємо крок RANSAC, що тепер буде включати оцінку гомографії, заснованої на наборі збігів (який, очевидно, містить велику кількість винятків). Це зроблено за допомогою функції `cv::findHomography`, яка дуже схожа на функцію `cv::findFundamentalMat`:

```
     // Знайдіть гомографію між зображенням 1 і зо-
браженням 2
     std::vector<char> inliers;
     cv::Mat homography= cv::findHomography(
                    points1,
                    points2, // відповідних точок
                    inliers, // виводиться відповідна
               версія збігів
                    cv::RANSAC, // RANSAC метод
                    1.); // максимальна відстань до
                    точки перепроекції
```

Нагадаємо, що існує гомографія (замість фундаментальної матриці) тому що наші два зображення розділені чистою ротацією. Ми показуємо тут вхідні ключові точки, визначені аргументом *inliers* функції:

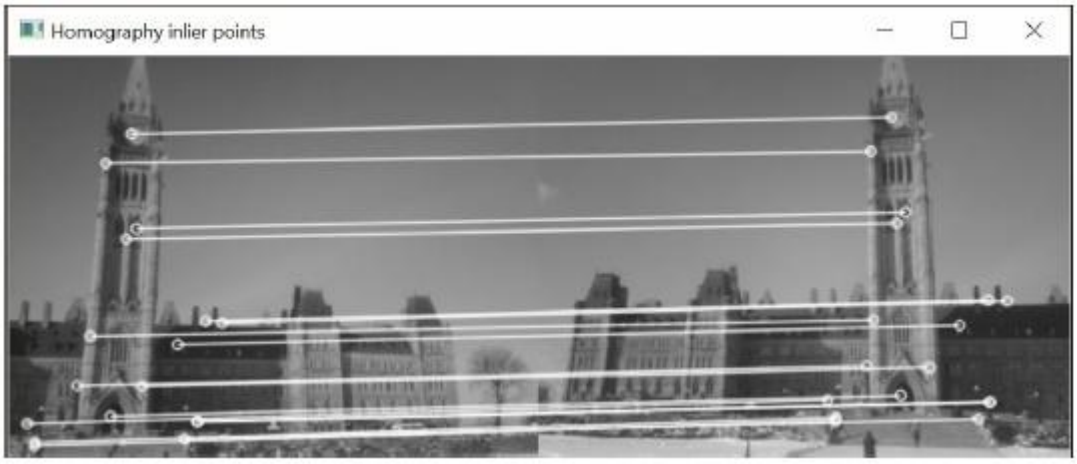


Рисунок 5.7 – Пошук гомографії між зображеннями

Гомографія являє собою матрицю *3x3*, що обертається. Тому можна передавати точки з одного раніше обчисленого зображення на інше. Фактично, це можна зробити для кожного пікселя зображення. Отже, можна передати повне зображення до точки зору другого зображення. Цей процес називається зображенням **мозаїки** або **зшиванням зображень** і часто використовується для створення великої панорами з декількох зображень. Функція OpenCV це робить саме так:

```
// Зшити два зображення
cv::Mat result;
cv::warpPerspective(image1, // вхідне зображення
                result, // вихідне зображення
                homography, // гомографія
                cv::Size(2*image1.cols,image1.rows));
                // розмір вихідного зображення
```

Як тільки це нове зображення буде отримано, його можна додати до іншого зображення для того, щоб розгорнути перегляд (оскільки два зображення тепер з однакової точки зору):

```
    // Копіювати зображення 1 на першій половині повного зображення
    cv::Mat half (result, cv::Rect (0,0, image2.cols, image2.rows));
    image2.copyTo (half); // копіювати image2 до image1 roi
```

Результатом є наступне зображення:

Рисунок 5.8 – Процес зшивання зображень

5.5 Визначення планарної цілі в зображенні

У попередньому підрозділі ми пояснили, що гомографією можна скористатися для зшивання зображень, розділених чистою ротацією, та створення панорами.

У цій частині ми також дізналися про різні образи влощини генерування гомографій між переглядами. Тепер ми побачимо, як ми можемо використовувати цей спосіб, щоб розпізнати планарний об'єкт у зображенні.

Припустимо, ми хочемо виявити виникнення плоского об'єкта в зображенні. Цей об'єкт може бути плакатом, живописом, логотипом, вивісками тощо.

Виходячи з того, що ми дізналися в цій главі, стратегія буде складаються з виявлення точок особливості на цьому планарному об'єкті та спроби порівняти їх з точками зображення на зображенні. Ці точки будуть потім перевіряється за допомогою надійної схеми узгодження, подібної до тієї, яку ми використовували раніше, але цього разу на основі гомографії. Якщо число допустимих збігів є високими, то це має означати, що наш планарний об'єкт є видимим на поточному зображенні.

Давайте визначимо клас `TargetMatcher`, який дуже схожий на наш клас RobustMatcher:

```
class TargetMatcher {
      private:
      // об'єкт детектора точок особливості
      cv::Ptr<cv::FeatureDetector> detector;
      // вказівник на об'єкт екстрактора дескрип-
 тора ознак
      cv::Ptr<cv::DescriptorExtractor>
 descriptor;
```

```
        cv::Mat target; // цільове зображення
        int normType; // для порівняння дескриптора
        векторів
        double distance; // min помилка перепроек-
    ції
        int numberOfLevels; // розмір піраміди
        double scaleFactor; // масштаб між рівнями
        // піраміда цільових зображень та її ключо-
    вих точок
        std::vector<cv::Mat> pyramid;
        std::vector<std::vector<cv::KeyPoint>>
    pyrKeypoints;
    std::vector<cv::Mat> pyrDescriptors;
```

Опорне зображення плоского об'єкта, що підлягає узгодженню, зберігає *цільовий* атрибут. Як буде пояснено в наступному розділі, функції точки будуть виявлені в піраміді зображень цілі послідовно вибираються. Методи зіставлення схожі з методами збігу Клас `RobustMatcher`, за винятком того, що вони включають `cv::findHomography` замість `cv::findFundamentalMat` в методі `ransacTest`.

Для використання класу `TargetMatcher`, специфічного детектора точок функцій і дескриптор повинен бути інстанційним і переданий конструктору:

```
// Підготовка
TargetMatcher
tmatcher(cv::FastFeatureDetector::create(10),
cv::BRISK::create());
tmatcher.setNormType(cv::NORM_HAMMING);
```

Тут ми вибрали детектор FAST разом з BRISK дескриптором, оскільки вони швидкі для обчислення. Потім необхідно вказати ціль, яку потрібно виявити:

```
// встановити цільове зображення
tmatcher.setTarget(target);
```

Ви можете виявити цю ціль у зображенні, викликавши `detectTarget` метод:

```
// відповідність зображення з ціллю
tmatcher.detectTarget(image, corners);
```

Цей метод повертає положення чотирьох кутів цілі в зображення (якщо знайдено). Потім можна намалювати лінії для візуального підтвердження виявлення:

```
// намалюйте кути на зображенні
     if (corners.size() == 4) { // є збіг
          cv::line(image, cv::Point(corners[0]),
               cv::Point(corners[1]),
               cv::Scalar(255, 255, 255), 3);
          cv::line(image, cv::Point(corners[1]),
               cv::Point(corners[2]),
               cv::Scalar(255, 255, 255), 3);
          cv::line(image, cv::Point(corners[2]),
               cv::Point(corners[3]),
               cv::Scalar(255, 255, 255), 3);
          cv::line(image, cv::Point(corners[3]),
               cv::Point(corners[0]),
               cv::Scalar(255, 255, 255), 3);
     }
```

5.6 Контрольні запитання

1. Запишіть рівнянням тонких лінз та поясніть значення його складових.
2. На яких припущеннях базується модель камери з точковим отвором?
3. Що пов'язує та як виглядає базове проективне рівняння?
4. Що називають проекційною матрицею?
5. Що означає термін стереобачення?
6. Поясніть поняття «епіполярна лінія» та «епіполе».
7. Що називається фундаментальною матрицею?
8. В чому полягають основні переваги методу визначення планарної цілі в зображенні?
9. Які основні задачі дозволяє вирішувати метод визначення планарної цілі в зображенні?
10. Розкрийте суть епіполярного обмеження.
11. В чому суть методу обчислення фундаментальних матриць пари зображень?
12. Які концептуальні шляхи вирішення прикладних задач обробки зображень передбачає метод відповідності зображення використання випадковим чином?
13. В яких сферах наразі використовується обробка зображень?

5.7 Хід роботи

5.7.1 Підготовчі стадії

Поза навчальною лабораторією треба зробити наступне:

1. Користуючись підрозділом 5.1, а також рекомендованою літерату-

рою, ознайомитися із принципами формування зображень та особливостями обчислення проекційної та фундаментальної матриць.

2. Оформити першу частину звіту з виконання лабораторної роботи, в якій зазначити прізвище студента, мету лабораторних досліджень та дати письмові відповіді на контрольні запитання (п. 5.6).

3. Для виконання поставлених завдань необхідно більш детально розглянути принципи і основні етапи реалізації процедур для маніпуляцій з проективними відносинами в зображеннях в OpenCV на сайті [2].

5.7.2 Завдання до роботи

1. Обчислити і намалювати епіполярні лінії власних зображень.

2. Виконати приклади обчислення фундаментальної матриці пари власних зображень.

3. Виконати приклади пошуку співпадіння власних зображень за допомогою консенсусу випадкових зразків.

4. Провести обчислення гомографії між двома власними зображеннями.

5. Визначити планарну ціль на власному зображенні.

5.8 Вимоги до звіту по роботі

Звіт про виконання роботи повинен містити:

1. Прізвище та групу студента, а також назву, мету та завдання лабораторної роботи.

2. Відповіді на контрольні питання.

3. Таблицю з переліком функцій OpenCV для маніпуляцій з .

4. Короткий перелік основних дій в процесі виконання завдань.

5. Текст розроблених програм з коментарями.

6. Скрін-шоти роботи розроблених програм.

7. Висновки, в яких треба зазначити, чи досягнута мета даної лабораторної роботи. Які саме конкретно маніпуляції з дозволяє (перерахувати по пунктах) робити OpenCV? Які конкретно практичні навички роботи з зображеннями здобуті в ході дослідження?

8. Електронну версію звіту з лабораторної роботи у форматі 2020_Прізвище-ЛР5.doc або *.docx разом з додатками (програмними проектами в архіві) треба завантажити у відповідному розділі сторінки дисципліни в системі дистанційного навчання “Moodle”.

5.9 Орієнтовні варіанти завдань

Завдання за варіантами видаються викладачем після виконання студентом підготовчої стадії (п. 5.7.1), типових завдань (п. 5.7.2), здавання звіту та його перевірки.

Рекомендована література

1. Laganiere R. OpenCV 3 Computer Vision Application Programming Cookbook**.** – Packt, 2017. – 573 с.
2. OpenCV (Open Source Computer Vision) [Електронний ресурс]. – Режим доступу: http://opencv.org/
3. Computer Vision System Toolbox [Електронний ресурс]. – Режим доступу : https://ww2.mathworks.cn/help/vision/index.html
4. Visual Studio 2019 [Електронний ресурс]. – Режим доступу : https://visualstudio.microsoft.com/ru/vs/
5. OpenCV (сторінка завантаження) [Електронний ресурс]. – Режим доступу: https://sourceforge.net/projects/opencvlibrary/files/4.5.1/opencv-4.5.1-vc14_vc15.exe/download
6. Гонсалес Р., Вудс Р. Цифровая обработка изображений. М.: Техносфера, 2012. – 1072 с.
7. Гонсалес Р., Вудс Р., Эддинс С. Цифровая обработка изображений в среде MATLAB. М.: Техносфера, 2006. – 621 с.
8. Gonzalez R. C., Woods R. E. Digital Image Processing. Prentice Hall, 2002. – 813 p.
9. Уэбб. Физика визуализации медицинских изображений. Т.1, т.2. – Мир, 1991.