ЗАПИСКА

Листів 44, ілюстрацій 15, джерел 7, додатків 2.

У змісті пояснювальної записки до роботи було розглянуто основні проблеми обробки зображень в системах управління технічним зором, методи отримання відеоданих (завантаження відео з файлу чи веб-камери). Опис можливостей бібліотеки OpenCV для колірних перетворень, тобто змінення кольорового простору в RGB (за варіантом). Також буде досліджено можливості бібліотеки для геометричних перетворень відео-зображення на прикладі скосу за заданими параметрами користувачем. Розгляд можливостей OpenCV для виконання операції виділення меж з різними порогами canny і фільтрації відео-зображень білатеральним фільтром з різними параметрами. Всі перетворення буде реалізовано на Python з реалізацією класу для обробки відеоданих та проаналізовано результати роботи. Далі буде описано висновки з курсової роботи, перелік використаних джерел та додатки з кодом та екранами роботи програми.

КЛЮЧОВІ СЛОВА: відеозображення, OpenCV, Python, фільтрація, геометричні перетворення, метод, клас, обробка зображення, RGB кольоровий простір, Canny, білатеральний фільтр, нахил кадру за коефіцієнтом

ЗМІСТ

ПЕРЕЛІК ВИКОРИСТАНИХ ПОЗНАЧЕНЬ

ТЗ – технічний зір;
СТЗ – системи технічного зору;
ПЗ – програмне забезпечення;
ін. – інше;

_____

ВСТУП

Системи технічного зору та обробки зображень (англ. «machine vision systems» і «image processing») відіграють ключову роль в автоматизації та аналізі візуальної інформації. Їх використовують у різних галузях, включно з промисловістю, медициною і робототехнікою, для виконання завдань, які потребують автоматичного сприйняття та інтерпретації візуальних даних. Тобто основна мета таких систем — аналіз візуальної інформації та прийняття рішень на основі отриманих даних.

Сучасні СТЗ включають збір даних з використанням різних камер і сенсорів, обробку зображень із застосуванням методів фільтрації та виділення ключових ознак, а також аналіз та інтерпретацію даних на основі алгоритмів машинного навчання. Ці етапи дають змогу системам розпізнавати об'єкти, класифікувати їх і приймати автоматизовані рішення в різних сферах [1].

Для обробки зображень та відео використовують безліч алгоритмів, включно з фільтрацією зображень для придушення шумів, детектуванням країв за допомогою алгоритмів Canny, Sobel і Laplacian та ін., сегментацією зображень із застосуванням порогових методів і алгоритмів кластеризації (K-means, Watershed), а також розпізнаванням об'єктів із застосуванням згорткових нейронних мереж (наприклад, CNN). Розробка і впровадження таких систем неможливі без спеціалізованого програмного забезпечення та апаратних засобів, зокрема бібліотеки OpenCV, TensorFlow, PyTorch, апаратні платформи GPU, FPGA і спеціалізовані процесори (TPU), а також мови програмування Python, C++ і MATLAB. Майбутнє цієї технології пов'язане з розвитком штучного інтелекту, збільшенням обчислювальних потужностей, розвитком квантових обчислень для оброблення зображень і широким застосуванням у сфері розумних міст [4].

Ця робота буде присвячена обробці відео-зображення за допомогою мови програмування Python з використанням бібліотеки OpenCV. В ході програми, буде завантажено відео з файлу чи веб-камери, далі за вибором користувача застосовано одне або декілька перетворень: скіс зображення за вказаним параметром, виділення меж з різними порогами Canny, кольоровий простір RGB чи білатеральний фільтр з різними параметрами. Для відображення застосунку, буде використано бібліотека Tkinter, що моє в собі багато зручних функцій та налаштувать графічного інтерфейсу.

1 ОГЛЯД ПРОБЛЕМИ ОБРОБКИ ЗОБРАЖЕНЬ В СИСТЕМАХ УПРАВЛІННЯ З ТЕХНІЧНИМ ЗОРОМ

Системи технічного зору виступають ключовим елементом автоматизованих технологій, що забезпечують моніторинг і контроль у різних сферах сучасного життя. Основна проблема обробки зображень у таких системах полягає у необхідності швидкої та точної ідентифікації об'єктів у складних умовах. Шум, низька якість зображень, змінні умови освітлення та необхідність обробки великих обсягів даних у реальному часі створюють значні виклики для розробників таких систем. Додатково, великі потоки даних вимагають ефективних методів стискання без втрати важливої інформації, а також адаптивних алгоритмів, що можуть налаштовуватися під різні умови експлуатації. Окрім цього, виникають проблеми сумісності з різними платформами, оскільки системи повинні інтегруватися в різноманітні апаратні середовища, забезпечуючи стабільність роботи та швидкість обробки даних. Складність систем управління ТЗ також полягає у необхідності їхньої адаптації до змінних умов навколишнього середовища, що вимагає розробки алгоритмів самообучення та оптимізації [2].

Використання алгоритмів машинного навчання та глибокого навчання, таких як нейронні мережі, значно покращило можливості СТЗ. Однак, ці методи мають ряд обмежень, включаючи потребу у великих обчислювальних ресурсах та значній кількості анотованих даних для тренування моделей. Окрім того, нейронні мережі можуть бути вразливими до атак, що створює ризики для безпеки систем. Високий рівень енергоспоживання ускладнює використання таких алгоритмів у мобільних та вбудованих системах, що вимагає розробки нових енергоефективних моделей. Також існує проблема пояснюваності рішень, оскільки багато алгоритмів глибокого навчання є "чорними ящиками", і їх висновки важко інтерпретувати та перевіряти. Крім того, проблема узагальнення моделей викликає труднощі, оскільки треновані нейромережі можуть демонструвати знижену ефективність при застосуванні в нових або незнайомих середовищах. Одним з можливих рішень є використання гібридних підходів, що поєднують класичні методи комп'ютерного зору з сучасними нейромережевими моделями, що дозволяє знизити вимоги до обчислювальних ресурсів та підвищити гнучкість систем.

Ще однією важливою проблемою є забезпечення надійності та безпеки таких систем. Помилки в розпізнаванні можуть призвести до серйозних наслідків, особливо в критичних застосуваннях, таких як автономний транспорт або медична діагностика. Недосконалі алгоритми можуть помилково класифікувати

об'єкти, що може спричинити небезпечні ситуації, наприклад, у випадку автопілотів чи систем відеоспостереження. Тому дослідники працюють над покращенням алгоритмів виявлення та ідентифікації об'єктів, розробляючи методи адаптивного навчання, що можуть коригувати свої моделі в процесі експлуатації, а також поєднання різних сенсорних даних, таких як зображення, радарні та лідарні дані, для підвищення точності розпізнавання. Важливим аспектом є також етичні питання використання ТЗ, адже впровадження таких технологій потребує відповідності законодавчим нормам щодо захисту персональних даних і конфіденційності. Крім того, необхідно враховувати питання відповідальності при ухваленні рішень системами штучного інтелекту, особливо у випадках, коли помилковий аналіз зображення може призвести до серйозних економічних або людських втрат [3].

Можна виявити декілька прикладів вирішення проблем в системах управління ТЗ. Одним із практичних рішень для покращення стабільності та якості розпізнавання є впровадження попередньої обробки зображень шляхом адаптивної нормалізації освітлення. Це дозволяє компенсувати змінні умови освітлення та покращити контрастність об'єктів. Також ефективним є застосування методів згладжування шумів, таких як гаусове фільтрування та медіанне згладжування, що дозволяє зменшити вплив артефактів та покращити якість вхідних даних для нейронних мереж.

Щодо вирішення проблеми високих обчислювальних витрат, багато дослідників пропонують використовувати глибокі згорткові нейронні мережі, оптимізовані для роботи на спеціалізованих апаратних прискорювачах, таких як GPU або TPU. Це дозволяє зменшити енергоспоживання та прискорити обробку зображень у реальному часі. Також впроваджуються методи квантування нейромереж, що значно зменшує їхній розмір та підвищує ефективність роботи вбудованих пристроїв.

Для підвищення надійності та безпеки розробляються комплексні системи обробки зображень, що поєднують дані з різних сенсорів. Наприклад, інтеграція камер із лідаром дозволяє значно покращити точність розпізнавання об'єктів, особливо в умовах недостатнього освітлення. Додатково використовуються методи ансамблевого навчання, які поєднують результати кількох незалежних моделей для підвищення точності класифікації та зменшення ймовірності помилкових рішень [1].

2 МЕТОДИ ТА ЗАСОБИ ОТРИМАННЯ ВІДЕОДАНИХ

2.1 Завантаження відео з файлу

Завантаження відео з файлу дозволяє отримувати відеопотік для подальшої обробки, аналізу чи візуалізації. Python і OpenCV надають простий та ефективний інтерфейс для роботи з відеофайлами різних форматів. При завантаженні відео необхідно враховувати кодек (програмний або апаратний засіб, який стискає або декомпресує відеофайли. Він визначає, як відеодані зберігаються та передаються, впливаючи на якість, розмір і сумісність відео. При завантаженні відео з файлу важливо враховувати кодек, оскільки деякі з них можуть бути несумісними з OpenCV або вимагати додаткових бібліотек для декодування), розширення файлу та параметри читання. Можна зчитувати відео кадр за кадром, обробляти зображення в реальному часі та застосовувати алгоритми комп'ютерного зору [5].

Для завантаження відео з файлу маємо функцію *load_video(source=0)*, де першим кроком, за допомогою ініціалізації об’єкта *cv2.VideoCapture* створюємо екземпляр із зазначенням шляху до відеофайлу. Це дозволяє OpenCV отримати доступ до відеофайлу та підготувати його для обробки. Далі, використовуємо метод *.read()* для зчитування кадрів, який повертає два значення: *ret* (булеве значення, що вказує на успішність зчитування) і *frame* (матриця пікселів поточного кадру). Якщо *ret* є *False*, це означає, що відео закінчилося або виникла помилка. Для відображення використовується *cv2.imshow()*, щоб показати зчитаний кадр у вікні, де цикл триває, поки кадри успішно зчитуються. Останнім етапом, для закриття всіх вікон після завершення обробки, викликаємо *cap.release()*, щоб звільнити ресурси, і *cv2.destroyAllWindows()*, щоб закрити всі створені вікна відображення [7].

Програмний код маємо в Додатку А (стор. 27) та екранні форми виконання в Додатку Б (рис.Б.1).

2.2 Захват відео з веб-камери

Захоплення відео з веб-камери є важливим аспектом комп'ютерного зору та обробки відео, яке має застосування в різних сферах, таких як безпека, відеоконференції, розпізнавання облич, а також у іграх та розвагах.

Першим кроком визначимося з поняттям «веб-камера» та з принципом роботи. Отже веб-камера — це невеликий електронний пристрій, який здатний захоплювати відеодані у реальному часі. Веб-камери працюють за принципом захоплення світла, яке потрапляє до сенсора камери, де воно перетворюється на електричні сигнали. Ці сигнали обробляються цифровими схемами і перетворюються на зображення, яке потім передається на комп'ютер або мобільний пристрій. При захопленні і передачі відео використовуються різні формати і кодеки. Одним з найпопулярніших є MJPEG (Motion JPEG) та H.264. Перший формат має наперед записані кадри, які передаються по одному, тоді як інший, більш сучасний і стиснутий формат, що використовується для потокового відео.

Наступним кроком маємо саме способи захоплення відео. Зазвичай це виконується за допомогою API (Application Programming Interface) (набір функцій і протоколів, які дозволяють програмам взаємодіяти між собою) або бібліотек, які спрощують роботу з апаратним забезпеченням. Найчастіше використовуюсть бібліотеки OpenCV чи Ffmpeg [2].

В нашому випадку, будемо використовувати бібліотеку OpenCV, де за допомогою функції *cv2.VideoCapture(0).* Число у дужках представляє індекс або ідентифікатор відеопристрою, яке можна використати для захоплення відео.

Далі маємо таки ж самі функції, як і у випадку завантаження відео з файлу. Тобто метод *.read()* для зчитування кадрів, для відображення — *cv2.imshow()*, для закриття всіх вікон після завершення обробки — *cap.release()*, та *cv2.destroyAllWindows()*, щоб закрити всі створені вікна відображення [7].

Програмний код маємо в Додатку А (стор. 27) та екранні форми виконання в Додатку Б (рис.Б.2). На екрані роботи буде показано помилку, бо відсутнє підключення камери до комп’ютеру через технічні причини.

3 КОЛІРНІ ПЕРЕТВОРЕННЯ ВІДЕО-ЗОБРАЖЕНЬ

3.1 Можливості бібліотеки OpenCV з колірних перетворень

Колірні перетворення є одними з найважливіших операцій при обробці зображень, оскільки вони дозволяють змінювати представлення кольорів, адаптувати зображення для різних цілей та виділяти певні характеристики.

OpenCV підтримує роботу з безліччю популярних колірних моделей, що дає змогу обирати найвідповідніший формат для конкретного завдання. Стандартним для OpenCV є колірний простір BGR (Blue, Green, Red), де порядок каналів — синій, зелений, червоний. BGR є основним колірним форматом, який використовується як вхідний формат у більшості прикладів [4].

Іншим важливим простором є HSV (Hue, Saturation, Value), що часто використовується для виділення об'єктів за кольором, оскільки він розділяє інформацію про колір (Hue) від яскравості (Value) і насиченості (Saturation). Також маємо перетворення у відтінки сірого (Grayscale), що є фундаментальною операцією, що зменшує обсяг даних [3].

OpenCV також підтримує перемикання в інші колірні простори, такі як HLS (Hue, Lightness, Saturation), Lab, YcrCb. Для детекції облич використовують YcrCb [7].

Основною функцією для виконання цих перетворень є *cv2.cvtColor()*, яка приймає зображення і прапор, що вказує тип перетворення. OpenCV також дозволяє розділяти й об'єднувати канали кольорового зображення. Це корисно для обробки окремих колірних компонентів. Крім основних перетворень, OpenCV надає й інші функції, корисні під час роботи з кольором, як-от нормалізація (зміна діапазону значень пікселів), аналіз гістограми (розподіл кольорів) та порогове перетворення (перетворення на бінарне зображення) [4].

Таким чином, бібліотека OpenCV має потужні можливості для колірних перетворень. Вона підтримує широкий спектр колірних просторів, надає зручні функції для їх конвертації, дає змогу працювати з окремими колірними каналами та має додаткові інструменти для їх конвертації.

3.2 Метод *cv2.cvtColor()* для перетворення відеозображення у кольоровий-простір RGB

Відеозображення у цифровому вигляді зазвичай зчитується у форматі з набором кадрів, кожен з яких є двумірною матрицею пікселів. Кожен піксель має набір значень каналів кольору, що визначають його зовнішній вигляд. Для кольорових зображень найпоширенішим є простір RGB (Red, Green, Blue), де кожен канал окремо кодує інтенсивність відповідного кольору [7].

За допомогою OpenCV відеозображення зчитується як послідовність кадрів типу *cv2.VideoCapture()*. Кожен кадр — це об'єкт *NumPy* масиву розміром (height, width, 3) з типом даних *uint8*, тобто це беззнакове ціле число довжиною 8 біт. Максимальне його значення може бути 255, а мінімальне — 0. Канали у цьому масиві за замовчуванням розташовані у порядку BGR, а саме:

$$Піксель = [B, G, R].$$

Математично це можна записати у вигляді перестановки індексів:

$$RGB = frame[:, :, [2, 1, 0]],$$

де frame — це вихідний масив у форматі BGR, а RGB — отриманий масив у форматі RGB.

Якщо розписати детально, то маємо для кожного пікселя (i,j) і кожного каналу k у вихідному зображенні:

$$BGR_{i,j,k} = RGB_{i,j,p(k)},$$

де функція p(k) — це перестановка індексів каналів:

$$p(0) = 2, p(1) = 1, p(2) = 0.$$

Інакше кажучи, маємо три кольорові канали: синій, зелений та червоний, — в яких міняються місцями перший та останній кольори, тоді як зелений залишається незмінним [4] (рис.3.2.1).

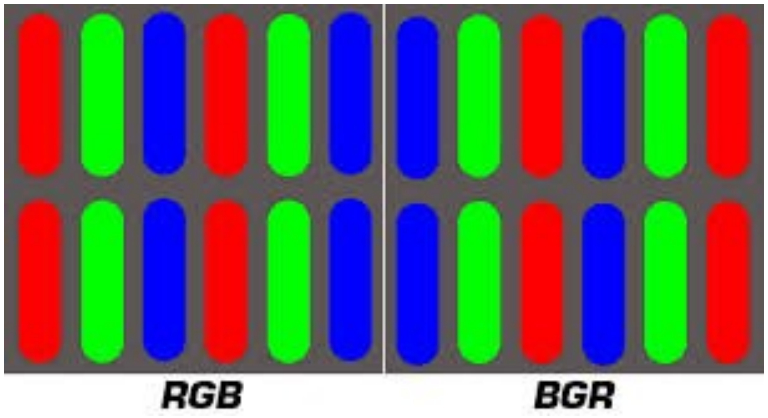

Рисунок 3.2.1 — Порівняння кольорових просторів BGR і RGB

3.3 Реалізація на Python й аналіз результатів

Реалізації кольорового перетворення на Python для зручності була виділена в окрему функцію під назвою *convert_to_rgb(frame).* Вона приймає на вхід один параметр — *frame*, який є кадром відео у форматі BGR, бо це стандартний формат для зображень у бібліотеці OpenCV. Перетворення основного формату в RGB простір було зроблено за допомогою функції *cv2.cvtColor ()*, з заданими у дужках параметрами: *frame* — вихідний кадр та *cv2.COLOR_BGR2RGB* — код кольорового перетворення.

Результатом роботи функції є новий масив (відеозображення), в якому компоненти кольору розташовані у порядку RGB: перша компонента — червоний колір, друга — зелений, третя — синій.

Програмний код маємо в Додатку А (стор. 28) та екранні форми виконання в Додатку Б (рис.Б.3).

4 ГЕОМЕТРИЧНІ ПЕРЕТВОРЕННЯ ВІДЕО-ЗОБРАЖЕНЬ

4.1 Можливості бібліотеки OpenCV з геометричних перетворень

Для виконання геометричних перетворень бібліотека OpenCV має широкий спектр інструментів. Вони необхідні для зміни форми, розміру, орієнтації або перспективи зображень з метою їх аналізу, корекції або отримання нових ефектів. Є багато видів перетворень для обробки відеозображень, але є 5 основних типів:

1. Зміна розміру. Для зменшення чи збільшення зображення використовується функція *cv2.resize()*, в дужках цієї функції задається потрібні параметри висоти та ширини або масштабні коефіцієнти по осях. Це дозволяє зменшити обчислювальні витрати або підігнати під формат.

2. Обертання. Цей вид геометричних перетворень виконується за допомогою функції *cv2.getRotationMatrix2D()*, яка створює матрицю обертання навколо заданої точки на певний кут. Далі використовуються афінні перетворення за функцією *cv2.warpAffine()* для застосування цієї матриці та отримання обернутого зображення. Частіше використовується для корекції орієнтації або створення художніх ефектів.

3. Масштабування і зсув. Також за допомогою афінних перетворень можна масштабувати або зсувати зображення у будь-якому напрямку. Для цього формуються відповідні матриці трансформації і застосовуються через *cv2.warpAffine()*.

4. Проективні перетворення. Ці перетворення змінюють зображення так, щоб воно відповідало новій точці зору або "перспективі". Вони формуються функцією *cv2.getPerspectiveTransform()* (для 3-4 точок) і застосовуються через *cv2.warpPerspective()*. Це дозволяє виправляти або імітувати зміни перспективи, наприклад, для "розгортання" фотографій або виправлення спотворень.

5. Зміщення зсув зображення виконується за допомогою афінної матриці, яка переміщує всі пікселі на вказану кількість по осях X і Y.

Бібліотека OpenCV забезпечує гнучкий інструментарій для виконання різноманітних геометричних перетворень зображень. Це дозволяє ефективно виконувати корекцію, масштабування, обертання та інші трансформації, що є незамінними у комп’ютерному зорі та обробці зображень [4].

4.2 Метод *cv2.warpAffine()* для скосу зображення

Для відтворення нахилу відеозображення було використано афінну матрицю, яка застосовується для перетворення зображення (або координат) з можливістю: зсувати, обертати, масштабувати, нахиляти.

Афінне перетворення у двовимірному просторі описується матрицею розміру 2×3:

$$M=\begin{bmatrix} a\,b\,t_x \\ c\,d\,t_y \end{bmatrix},$$

де a, b, c, d — коефіцієнти трансформації (масштабування, поворот, нахил), а t_x, t_y - зміщення (переміщення) по осях X і Y.

Будь-яка точка (x, y) перетворюється за формулою:

$$x'=ax+by+tx;$$
$$y'=cx+dy+ty.$$

У даному випадку для нахилу (або скосу) відеозображення маємо наступну афінну матрицю:

$$M=\begin{bmatrix} 1\,x\,0 \\ 0\,1\,0 \end{bmatrix},$$

де x — параметр скосу, який може бути додатнім чи від'ємним, але не може бути нулем, бо нахилу зображення не буде, тобто маємо зображення без обробки [2].

4.3 Реалізація на Python й аналіз результатів

Для реалізації в Python була застосована окрема функція *shear_frame*, яка виконує «зсув» зображення — спотворення, за якого зображення «нахиляється» по певній осі. Це досягається за допомогою афінного перетворення. З функцією маємо два параметри: *frame* — вихідне зображення для обробки та *shear_factor* — коефіцієнт shear (нахилу), який визначає, наскільки сильно зображення буде «нахилене».

Маємо отримання висоти та ширини відеозображення за допомогою *frame.shape[:2]*, що повертає розміри для подальшої обробки. Далі маємо створення афінної матриці *shear_matrix* розміром 2×3. Саме ця матриця задає зсув за горизонталлю із нахилом, при цьому залишаючи вертикальну вісь без змін. Наступним кроком, маємо розрахувати ширину нового зображення, щоб вмістити нахилене. У разі зсуву по горизонталі (shear) вихідна ширина повинна збільшитися, щоб уся картинка помістилася на екрані заданого розміру. Тому маємо нову змінну *new_width*, яка в свою чергу має формулу:

$$new\ width = width + |shear\ factor|\, shear\ factor * height.$$

Якщо shear_factor позитивний, зображення нахилене вправо. Якщо від'ємний — вліво. І використовуємо *abs()* для врахування обох варіантів.

Останнім кроком, маємо викликати функцію *cv2.warpAffine()* при заданих параметрах *frame*, *shear_matrix*, *(new_width, height)* — розміри вихідного зображення, щоб воно вмістило нахилене. Та потім повертаємо перетворене зображення за допомогою *return*.

В якості результату маємо виконання нахилу відео за горизонтальною віссю, змінюючи його форму так, щоб було видно ефект зсуву з нахилом. При даному перетворенні користувачеві необхідно вводити коефіцієнт зсуву, що може наочно продемонструвати роботу цієї функції.

Програмний код маємо в Додатку А (стор. 29) та екранні форми виконання в Додатку Б (рис.Б.4).

5 ОПЕРАЦІЇ З ВІДЕО-ЗОБРАЖЕННЯМИ

5.1 Можливості бібліотеки OpenCV для виконання операцій

Обробка зображень за допомогою фільтрів і детекторів країв є важливою частиною комп'ютерного зору. В OpenCV реалізовано численні алгоритми для поліпшення зображень, виділення меж і аналізу структури кадру.

Операціями з відео-зображенями можна назвати всі функції, які використовувати для роботи від зчитування та відтворення відео та обробки до запису відео чи геометричних перетворень. Але в даному випадку представимо способи обробки кадрів [3] (таблиці 5.1.1).

Таблиця 5.1.1 — Способи обробки кадрів

Функція	Призначення
Фільтр Гауса (*cv2.GaussianBlur()*)	Цей метод для згладжування зображень, що ґрунтується на застосуванні гаусового ядра (розподілу Гауса). Він використовується для зменшення шуму, згладжування деталей і усунення дрібних дефектів. Цей фільтр особливо корисний перед виконанням операцій виявлення меж або сегментації, оскільки шуми можуть спотворювати результати.
Медіанний фільтр (*cv2.medianBlur()*)	Це нелінійний фільтр, застосовуваний для видалення дуже яскравих або темних точок, які можуть з'являтися через перешкоди або дефекти сенсора. Він особливо ефективний у фотографіях з великим шумом.
Фільтр Собеля (*cv2.Sobel()*)	Оператор для виділення меж і країв на зображенні, який заснований на обчисленні перших похідних інтенсивності зображення за горизонталлю і вертикаллю, що дає змогу визначити

	напрямки і силу кордонів.
Фільтр Канні (*cv2.Canny()*)	Один із найвідоміших і найбільш широко використовуваних методів детекції меж. Він заснований на кількох кроках: згладжування зображення, обчислення градієнтів, придушення немаксимумів і порогова сегментація

5.2 Метод *cv2.Canny()* для виділення меж з різними порогами Canny

Функція *cv2.Canny()* в OpenCV використовується для виявлення контурів на зображенні. Цей метод був розроблений Джоном Кенні у 1986 році. Він ефективно знаходить межі об’єктів, навіть при наявності шуму.

Алгоритм складається з декількох послідовних етапів: згладжування, обчислення градієнтів, придушення немаксимальних значень,подвійний поріг та трасування країв з гістерезисом.

Для уникнення хибних країв зображення спочатку розмивається за допомогою гаусового фільтра. Далі визначається зміна яскравості пікселів у горизонтальному (X) та вертикальному (Y) напрямках, зазвичай за допомогою оператора Собеля. Потім видаляються всі пікселі, які не є локальними максимумами у напрямку градієнта. Може використовувати два пороги: високий поріг — визначає сильні краї, низький поріг — визначає слабкі краї. Слабкі краї зберігаються тільки тоді, коли вони зв’язані з сильними — це зменшує шум[5].

5.3 Реалізація на Python й аналіз результатів

В Python маємо окрему функцію *canny_frame()*, з параметром *frame* — початковий кадр для обробки. Для виділення меж операцією *cv2.Canny()* потрібно, попередньо перевести кольорове зображення з простору BGR у градації сірого методом *cv2.cvtColor()*.

Наступним кроком досліджуємо виявлення меж за різних порогів для порівняння. Тобто при нижньому порозі маємо значення 50 і 150 — це чутливіше виявлення меж та можуть ще з'являтися більше шумів, при середньому зі значеннями порогів 100 і 200 маємо помірну чутливість, при порогах 150 і 250 —

суворіші та тільки найсильніші межі. Це робиться для порівняння впливу порогів на виявлення меж.

Та для кращого перегляду три отриманих зображення було об’єднано по горизонталі в один загальний кадр за допомогою функції *cv2.hconcat()*. Останнім кроком повертається об'єднане зображення, що містить три варіанти меж.

У якості результату маємо екрани роботи в Додатку Б (рис.Б. 30) та програмний код в Додатку А (стор.5).

6 ФІЛЬТРАЦІЯ ВІДЕО-ЗОБРАЖЕНЬ

6.1 Можливості бібліотеки OpenCV з фільтрації

Фільтрація зображень в бібліотеці OpenCV дозволяє виконувати зм’якшення, розмиття, зняття шуму, підсилення деталей і багато іншого. Більшість прикладів фільтрів було наведено в операціях з відео-зображеннями, але можно привести ще декілька основних фільтрів:

1. Просте розмиття (Blur) — це техніка обробки зображень, яка зменшує деталі та шум шляхом усереднення значень пікселів у певній області. Виконується за допомогою функції *cv2.blur()*. Тобто, як результат, кожен піксель замінюється середнім значенням всіх пікселів у вказаній області. Це призводить до згладжування різких переходів, країв і шумів.

2. Білатеральний фільтр — це тип нелінійного фільтру, який використовується для зменшення шумів у зображеннях, зберігаючи при цьому різкі краї та деталі. Це особливо корисно для зменшення шумів, не розмиваючи важливі контури. Виконується за допомогою функції *cv2.bilateralFilter()*.

3. Шарпінг (загострення) — це процес підвищення контрасту на межах об'єктів, щоб зробити зображення більш чітким і різким. Зазвичай можна зробити за допомогою фільтрів, що підсилюють різкість контурів, наприклад, через операцію різниці між зображенням і його розмитою версією.

4. Застосування довільного фільтра за допомогою *cv2.filter2D()*. Ця функція дозволяє застосовувати будь-який користувацький фільтр до зображення. А саме, передається двомірний масив, який буде застосовано через операцію згортки до зображення.

5. Лінійні фільтри (каскад фільтрів) — це послідовність застосувань кількох фільтрів один за одним. Це дозволяє комбінувати різні ефекти обробки для досягнення бажаного результату[6].

6.2 Метод *cv2.bilateralFilter()* з різними параметрами для фільрації відеозображення

Білатеральний фільтр (або білялінг-фільтр) — це вид нелінійного згладжуючого фільтру, який застосовується для зменшення шумів на зображеннях,

одночасно зберігаючи чіткість країв і деталей. Він особливо ефективний для усунення шуму без розмиття важливих контурів.

Особливість даного методу — це врахування двох факторів при обчисленні нового значення кожного пікселя: просторової близкості та колірної схожості, тобто, враховує, наскільки пікселі близькі просторово і за кольором[3].

6.3 Реалізація на Python й аналіз результатів

На практиці реалізацію білатерального фільтру можна записати одним рядком, за допомогою функції *cv2.bilateralFilter()*. В якості параметрів для методу, маємо вхідний відіо-потік — *frame* та три числа, які вводяться користувачем: *diameter* — діаметр сусідніх пікселів, які враховуються при фільтрації, *sigma_color* — параметр яскравості, тобто наскільки сильно фільтр враховує різницю в кольорі, *sigma_space* — параметр відстані, враховується відстань між пікселями. Для параметрі, які будуть вводитись користувачем, було обрано діапазони за яких буде спостерігатися змінення зображення.

Програмний код маємо в Додатку А (стор. 31) та екранні форми виконання в Додатку Б (рис.Б.6).

7 РЕАЛІЗАЦІЯ КЛАСУ ДЛЯ ОБРОБКИ ВІДЕОДАНИХ

7.1 Поняття класу та його створення на Python

Клас — це основний будівельний блок об'єктно-орієнтованого програмування, який слугує шаблоном для створення об'єктів. У контексті таких галузей, як комп'ютерний зір і обробка зображень, класи дають змогу об'єднати в одному об'єкті властивості та функції, необхідні для виконання конкретних завдань [4].

В структурі класу маємо декілька головних елементів, по-перше, у Python оголошення класу виконується за допомогою ключового слова *class*, за яким слідує ім'я класу. Зазвичай ім'я пишуть із великої літери. Найважливіша частина класу — це його конструктор, який викликається автоматично при створенні нового об'єкта. У ньому задаються початкові значення атрибутів об'єкта [2]. У якості кодового слова *__init__(self)*, де *self* — посилання на сам об'єкт, що дає змогу звертатися до його атрибутів і методів.

Атрибути — це змінні всередині класу, які зберігають інформацію про конкретний об'єкт. Наприклад, для класу, пов'язаного із зображеннями, атрибутами можуть бути шлях до файлу (*image_path*), завантажене зображення (*image*), параметри обробки тощо.

Методи — це функції, визначені всередині класу, які описують поведінку об'єктів. Наприклад, метод *load_image()* може завантажувати зображення з файлу, а *process_image()* — виконувати його обробку. У літературі підкреслюється, що методи використовують атрибути об'єкта і можуть змінювати їхній стан. Розрізняються різні типи методів:

1) Методи екземпляра (використовують *self*) — працюють із конкретним об'єктом.

2) Клас-методи (з декоратором *@classmethod*) — оперують класом цілком.

3) Статичні методи (*@staticmethod*) — не залежать ні від класу, ні від конкретного об'єкта, можуть використовуватися для допоміжних функцій [6].

7.2 Реалізація класу

Першим кроком реалізації класу є ініціалізація графічного інтерфейсу з усіма необхідними елементами управління, такими як кнопки, чекбокси, поля

для введення даних і області для відображення відео. Інтерфейс дозволяє користувачу запускати та зупиняти відео, вибирати файли, а також налаштовувати фільтри для обробки кадрів.

Для завантаження користувачем відеофайлу клас відкриває стандартний діалог для вибору файлу відео, що дозволяє зручно обрати потрібний відеофайл із файлової системи. Після вибору відео воно відкривається і готове до відтворення та обробки. Користувач може запустити відтворення відео, а також поставити його на паузу або повернути до перегляду за допомогою однієї кнопки. Це реалізовано через відповідні кнопки, які керують станом відтворення, а сама логіка обробки кадрів виконується у циклі, що оновлює зображення на екрані.

В інтерфейсі є дві області: одна для показу оригінального відео, інша — для обробленого. Це дозволяє користувачу порівнювати вихідний кадр з його обробкою в режимі реального часу. Для обробки кадру маємо застосування різних фільтрів: RGB конвертація, зміщення, Canny, білатеральний фільтр. Ці фільтри можна активувати або деактивувати за допомогою чекбоксів. Для деяких фільтрів передбачені додаткові налаштування (параметри), що дозволяє користувачу налаштовувати ступінь обробки зображень у реальному часі. При активації чекбоксів з’являються або зникають відповідні поля для введення параметрів фільтрів. Це дає змогу користувачу швидко і зручно регулювати параметри обробки кадрів без необхідності перезапуску програми або ручного редагування коду.

Для опису атрибутів класу сформуємо таблицю 7.2.1.

Таблиця 7.2.1 — Опис атрибутів класу

Назва атрибуту	Тип даних	Призначення
root	*tk.Tk*	Головне вікно додатку
left_label	*tk.Label*	Надпис для області відображення оригінального відео
right_label	*tk.Label*	Надпис для області відображення обробленого відео
left_canvas	*tk.Canvas*	Полотно для оригінального відео

right_canvas	*tk.Canvas*	Полотно для обробленого відео
capp	*cv2.VideoCapture*	Об'єкт для роботи з відеофайлом
is_playing	*bool*	Статус відтворення відео (грає/пауза)
fun1_var	*tk.BooleanVar*	Логічна змінна стану чекбоксу "Filter RGB"
fun2_var	*tk.BooleanVar*	Логічна змінна стану чекбоксу "Video Shifting"
fun3_var	*tk.BooleanVar*	Логічна змінна стану чекбоксу "Filter Canny"
fun4_var	*tk.BooleanVar*	Логічна змінна стану чекбоксу "Filter Bilateral"
entry_2	*tk.Entry*	Поле для введення параметра зміщення при фільтрі "Video Shifting"
entry1, entry2, entry3	*tk.Entry*	Поля для введення параметрів білатерального фільтру (діаметр, sigmaColor, sigmaSpace)
label_2, label1, label2, label3	*tk.Label*	Динамічні мітки для полів введення параметрів фільтрів

Також, для зручності, запишемо опис методів для класу з поясненнями параметрів та результатів у таблицю 7.2.2.

Таблиця 7.2.2 — Опис методів класу

Назва методу	Параметри	Результат роботи
__init__(self, root)	*root (tk.Tk)*	Ініціалізує графічний інтерфейс, створює всі елементи управління та налаштовує їх

update_frames(self)	-	Зчитує, обробля та відображає відеокадри у відповідних канвасах. Викликається кожні 30 мс.
start_videos(self)	-	Запускає відтворення відео, якщо файл відкритий.
toggle_pause(self)	-	Перевіряє та змінює стан паузи/відтворення, оновлює текст кнопки відповідно
choose_video_file(self)	-	Відкриває діалогове вікно для вибору відеофайлу, закриває попередній, якщо був відкритий
toggle_func2_fields(self)	-	Відображає або приховує поля введення для параметра зміщення при активації відповідного чекбоксу
toggle_func4_fields(self)	-	Відображає або приховує поля введення для параметрів білатерального фільтра при активації чекбоксу

Перегляд повного кода у Додатку А (стор. 32) та екранні форми виконання в Додатку Б (рис.Б.7-).

ВИСНОВКИ

В період роботи над курсовим проектом було досліджено теоретичні матеріали щодо принципі роботи бібліотеки OpenCV у програмному середовищі Python. На практиці було розроблено клас VideoApp, який успішно реалізує інтуїтивно зрозумілий та функціональний графічний інтерфейс для обробки відео в реальному часі. За допомогою цього додатку користувач має можливість легко завантажувати відеофайли, запускати їх відтворення та керувати процесом за допомогою зручних елементів управління — кнопок і чекбоксів. Основною перевагою є можливість застосовувати різноманітні фільтри до кадрів відео, включаючи конвертацію кольорів, зміщення, фільтр Canny та білатеральний фільтр, а також налаштовувати їх параметри самостійно. Це забезпечує гнучкість у налаштуванні обробки та створює умови для експериментування та досліджень у галузі комп’ютерного зору. Реалізація можливості одночасного відображення оригінального та обробленого відео дозволяє користувачу безпосередньо порівнювати результати застосованих фільтрів, що є важливим для аналізу їх ефективності.

Загалом, клас VideoApp демонструє ефективний підхід до створення інтерактивних мультимедійних додатків із широкими можливостями налаштування та зручним користувацьким інтерфейсом, що може бути корисним у навчальних, дослідницьких та практичних цілях у сфері обробки відеоданих.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Computer and machine vision: Theory, algorithms, practicalities [Текст] / Davies E. R. 4th ed. Amsterdam : Elsevier, 2012 — 286 с.

2. Электронный ресурс: https://www.imageprocessingplace.com/ – Ресурс, пов'язаний із книжкою «Digital Image Processing» Гонсалеса та Вудса.

3. Computer Vision: Algorithms and Applications [Текст] / Richard Szeliski , 2nd ed., 2022 — 925с.

4. Learning OpenCV: Computer Vision with the OpenCV Library [Текст] / Gary Bradski и Adrian Kaehler, 1st ed., 2008 — 238с.

5. Электронный ресурс: http://docs.opencv.org – Документация по библиотеке OpenCV.

6. Электронный ресурс: https://www.pyimagesearch.com/ – Посібник із захоплення відео в Python OpenCV.

7. Practical OpenCV [Текст] / Samarth Brahmbhatt, 2013 — 119с.

ДОДАТОК А

Лістинг коду

Лістинг коду для зчитування відео з файлу та з камери:

```
import cv2 # import library for working with videos


def load_video(source=0):  # open the video stream from the specified
source (file or webcam)
    cap = cv2.VideoCapture(source)
    if not cap.isOpened():  # checking the opening of a video
        print("Error: Could not open video source")
        return None
    return cap


def risize_video(frame):
    original_height, original_width = frame.shape[:2]
    new_width = 600
    aspect_ratio = new_width / original_width
    new_height = int(original_height * aspect_ratio)
    return cv2.resize(frame, (new_width,new_height) )


def play_video(source=0):
    cap = load_video(source) # open the video stream from the specified
source (file or webcam)
    if cap is None:
        return
    while True:
        ret, source_frame = cap.read()
        frame = risize_video(source_frame)
        if not ret:
            print("Error: Failed to read frame or video ended")
            break


        # display the original video in the window
        cv2.imshow("Original Video", frame)


    cap.release()
    cv2.destroyAllWindows()


play_video("Avia.mp4")
```

Лістинг коду для перетворення відеозображення у кольоровийпростір RGB:

```
    import cv2 # import library for working with videos
    import numpy as np # import library for working with mathematical expressions and matrices


    def load_video(source=0):   # open the video stream from the specified source (file or webcam)
        cap = cv2.VideoCapture(source)
        if not cap.isOpened():  # checking the opening of a video
            print("Error: Could not open video source")
            return None
        return cap


    def convert_to_rgb(frame): # converting a frame to RGB
        return cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
    def play_video(source=0):
         cap = load_video(source) # open the video stream from the specified source (file or webcam)
        if cap is None:
            return

        try:
            shear_factor = float(input("Enter the bevelling factor: "))
        except ValueError:
            print("Error: Invalid input. Please enter a numerical value.")
            return

        while True:
            ret, frame = cap.read()
            if not ret:
                print("Error: Failed to read frame or video ended")
                break

            # display the original video in the window
            cv2.imshow("Original Video", frame)

            # display the converted RGB video
            frame_rgb = convert_to_rgb(frame)
            cv2.imshow("RGB Video", frame_rgb)

            if cv2.waitKey(30) & 0xFF == ord('a'):
                break

        cap.release()
        cv2.destroyAllWindows()
```

```
play_video("Avia.mp4")
```

Лістинг коду для скосу зображення:

```
import cv2 # import library for working with videos
import numpy as np # import library for working with mathematical expressions and matrices


def load_video(source=0):   # open the video stream from the specified source (file or webcam)
    cap = cv2.VideoCapture(source)
    if not cap.isOpened():  # checking the opening of a video
        print("Error: Could not open video source")
        return None
    return cap
def shear_frame(frame, shear_factor): # application of bevelling
    height, width = frame.shape[:2]

    # calculate the new width after the shift
    shear_matrix = np.float32([[1, shear_factor, 0], [0, 1, 0]])

     # create a blank image to insert the result
    new_width = int(width + abs(shear_factor) * height)

    # apply the affine transformation
        sheared_frame = cv2.warpAffine(frame, shear_matrix, (new_width, height))

    return sheared_frame


def play_video(source=0):
     cap = load_video(source) # open the video stream from the specified source (file or webcam)
    if cap is None:
        return

    try:
        shear_factor = float(input("Enter the bevelling factor: "))
    except ValueError:
        print("Error: Invalid input. Please enter a numerical value.")
        return

    while True:
        ret, frame = cap.read()
        if not ret:
            print("Error: Failed to read frame or video ended")
            break

        # display the original video in the window
        cv2.imshow("Original Video", frame)
```

```
            # bevel parameter entry
            sheared_frame = shear_frame(frame, shear_factor)
            cv2.imshow("Sheared Video", sheared_frame)

            if cv2.waitKey(30) & 0xFF == ord('a'):
                break

        cap.release()
        cv2.destroyAllWindows()



play_video("Avia.mp4")
```

Лістинг коду для виділення меж з різними порогами Canny:

```
import cv2 # import library for working with videos
import numpy as np # import library for working with mathematical expressions and matrices


def load_video(source=0):   # open the video stream from the specified source (file or webcam)
        cap = cv2.VideoCapture(source)
        if not cap.isOpened():  # checking the opening of a video
            print("Error: Could not open video source")
            return None
        return cap
    def canny_frame(frame):
        # converts images in gray
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)


        # different Canny thresholds
        edges_low = cv2.Canny(gray, 50, 150)   # low threshold
        edges_mid = cv2.Canny(gray, 100, 200)  # middle threshold
        edges_high = cv2.Canny(gray, 150, 250) # high threshold


        # combining frames for easy viewing
        combined = cv2.hconcat([edges_low, edges_mid, edges_high])
        return combined
    def play_video(source=0):
         cap = load_video(source) # open the video stream from the specified source (file or webcam)
        if cap is None:
            return

        while True:
            ret, frame = cap.read()
            if not ret:
```

```
                print("Error: Failed to read frame or video ended")
                break

            # display the original video in the window
            cv2.imshow("Original Video", frame)
            # display the canny video
            combined = canny_frame(frame)
            cv2.imshow("Canny Video", combined)


            if cv2.waitKey(30) & 0xFF == ord('a'):
                break

        cap.release()
        cv2.destroyAllWindows()



play_video("Avia.mp4")
```

Лістинг коду для перетворення білатерального фільтра з різними параметрами:

```
    import cv2 # import library for working with videos
    import numpy as np # import library for working with mathematical expressions and matrices


    def load_video(source=0):   # open the video stream from the specified source (file or webcam)
        cap = cv2.VideoCapture(source)
        if not cap.isOpened():  # checking the opening of a video
            print("Error: Could not open video source")
            return None
        return cap
    def risize_video(frame):
        original_height, original_width = frame.shape[:2]
        new_width = 600
        aspect_ratio = new_width / original_width
        new_height = int(original_height * aspect_ratio)
        return cv2.resize(frame, (new_width,new_height) )
    def bilateral_filter(frame, diameter, sigma_color, sigma_space):
        # using Bilateral filter to a frame
        return cv2.bilateralFilter(frame, diameter, sigma_color, sigma_space)


    def play_video(source=0):
         cap = load_video(source) # open the video stream from the specified source (file or webcam)
        if cap is None:
            return
        try:
```

```
            diameter = int(input("Enter filter diameter (from 5 to 25): "))
              sigma_color = int(input("Enter the sigmaColor value (from 50 to
200): "))
             sigma_space = int(input("Enter a sigmaSpace value (from 50 to 200):
"))
        except ValueError:
            print("Input error! Default values are used.")
            diameter = 15
            sigma_color = 75
            sigma_space = 75


        while True:
            ret, source_frame = cap.read()
            frame = risize_video(source_frame)
            if not ret:
                print("Error: Failed to read frame or video ended")
                break

            # display the original video in the window
            cv2.imshow("Original Video", frame)


            # display the bilateral filter video
                  bilateral = bilateral_filter(frame, diameter, sigma_color,
sigma_space)
            cv2.imshow("Bilateral Filter Video", bilateral)


            if cv2.waitKey(30) & 0xFF == ord('a'):
                break

        cap.release()
        cv2.destroyAllWindows()



    play_video("Avia.mp4")
```

Лістинг коду для реалізації класу:

```
import tkinter as tk # graphical interface library
from tkinter import ttk, filedialog
import cv2 # OpenCV library
from PIL import Image, ImageTk # PIL image manipulation libraries
import threading
import course as c # importing the course module


class VideoApp:

    def update_frames(self):
        if not self.is_playing:
```

```
                return
            # read frame from video
            ret, frame = self.capp.read()
            if ret:
                # changing the frame size
                frame = cv2.resize(frame, (400, 300))
                # conversion of frames from RGB to BGR
                frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
                # save the original frame
                self.frame = frame
                # display the original frame on the left
                img_left = Image.fromarray(frame)
                self.left_photo = ImageTk.PhotoImage(image=img_left)
                     self.left_canvas.create_image(0, 0, image=self.left_photo,
anchor=tk.NW)
                # original processing
                processed_frame = frame


                # filters depending on the selected checkboxes
                if self.func1_var.get():
                    # conversion frames from BGR to RGB
                    processed_frame = c.convert_to_rgb(processed_frame)


                if self.func2_var.get():
                    # shifting for frames
                    try:
                                shear_value = float(self.entry_2.get()) # shift
parameter
                    except (AttributeError, ValueError):
                        shear_value = 0 # default value for shift
                            processed_frame = c.video_shifting(processed_frame,
shear_value)


                if self.func3_var.get():
                    # canny filter for frames
                    processed_frame = c.filter_canny(processed_frame)


                if self.func4_var.get():
                    # bilateral filter for frames
                    try:
                        d = int(self.entry1.get())
                        sigma_color = int(self.entry2.get())
                        sigma_space = int(self.entry3.get())
                    except (AttributeError, ValueError):
                            d, sigma_color, sigma_space = 15, 75, 75 # default
values for the Bilateral filter
                        processed_frame = c.filter_bilateral(processed_frame, d,
sigma_color, sigma_space)
```

```
                # processed frame on the right
                img_right = Image.fromarray(processed_frame)
                self.right_photo = ImageTk.PhotoImage(image=img_right)
                   self.right_canvas.create_image(0, 0, image=self.right_photo,
anchor=tk.NW)

            # frame update every 30 ms
            self.root.after(30, self.update_frames)


        # start the video display
        def start_videos(self):
            if not self.capp or not self.capp.isOpened():
                return
            self.is_playing = True
            self.update_frames()


        # switching pause/resume
        def toggle_pause(self):
            self.is_playing = not self.is_playing
            if self.is_playing:
                self.update_frames()
              self.btn_pause_video.config(text="Pause" if self.is_playing else
"Resume")


        # select a video file in the dialogue box
        def choose_video_file(self):
              filetypes = ("Video files", "*.mp4 *.avi *.mov"), ("All files",
"*.*")
             filename = filedialog.askopenfilename(title="Select a video file",
filetypes=filetypes)
            if filename:
                if hasattr(self, 'capp') and self.capp:
                    self.capp.release()
                self.capp = cv2.VideoCapture(filename)


        # initialising the user interface
        def __init__(self, root):
            # programme operation screen
            self.root = root
            self.root.title("Video comparison")
            self.root.geometry("810x550")
            self.root.configure(bg="pink")


            # label for original video
             self.left_label = tk.Label(root, text="Original video", bg="pink",
fg="black", font=("Arial", 12, "bold"), pady=5)
            self.left_label.grid(row=0, column=0, sticky="n")
            # label for filter video
```

```
            self.right_label = tk.Label(root, text="Filter video", bg="pink",
fg="black", font=("Arial", 12, "bold"), pady=5)
            self.right_label.grid(row=0, column=1, sticky="n")

            # canvas for video display
                  self.left_canvas = tk.Canvas(root, width=400, height=300,
bg='white')
            self.left_canvas.grid(row=1, column=0)

                  self.right_canvas = tk.Canvas(root, width=400, height=300,
bg='white')
            self.right_canvas.grid(row=1, column=1)

            # opening a video file
            self.capp = None
            self.is_playing = False

            # variables for the state of the checkboxes
            self.func1_var = tk.BooleanVar()
            self.func2_var = tk.BooleanVar()
            self.func3_var = tk.BooleanVar()
            self.func4_var = tk.BooleanVar()

            # frame for checkboxes and buttons
            self.checkbox_frame = tk.Frame(root, bg="pink")
            self.checkbox_frame.grid(row=6, column=0, columnspan=2, pady=10)

            # video start button
             self.btn_load_left = tk.Button(self.checkbox_frame, text="Load the
video", font=("Arial", 10, "bold"), bg="lightblue", fg="black",
activebackground="#add8e6", bd=2, relief="raised", command=self.start_videos)
            self.btn_load_left.grid(row=0, column=1, sticky="w")

            # video selection button
             self.btn_choose_video = tk.Button(self.checkbox_frame, text="Select
the video", font=("Arial", 10, "bold"), bg="lightgreen", fg="black",
activebackground="#90ee90", bd=2, relief="raised",
command=self.choose_video_file)
            self.btn_choose_video.grid(row=0, column=0, padx=3)

            # pause and resume video button
             self.btn_pause_video = tk.Button(self.checkbox_frame, text="Pause",
font=("Arial", 10, "bold"), bg="orange", fg="black", activebackground="#ffcc80",
bd=2, relief="raised", command=self.toggle_pause)
            self.btn_pause_video.grid(row=0, column=2, padx=2)

            # checkboxes for filters
```

```
                    self.chk_function1  =  tk.Checkbutton(self.checkbox_frame,
text="Filter RGB", variable=self.func1_var, bg="pink", font=("Arial", 10))
            self.chk_function1.grid(row=1, column=0, sticky="w")

                    self.chk_function2  =  tk.Checkbutton(self.checkbox_frame,
text="Video  Shifting",  variable=self.func2_var,  bg="pink",  font=("Arial",  10),
command=self.toggle_func2_fields)
            self.chk_function2.grid(row=2, column=0, sticky="w")

                    self.chk_function3  =  tk.Checkbutton(self.checkbox_frame,
text="Filter Canny", variable=self.func3_var, bg="pink", font=("Arial", 10))
            self.chk_function3.grid(row=3, column=0, sticky="w")

                    self.chk_function4  =  tk.Checkbutton(self.checkbox_frame,
text="Filter Bilateral", variable=self.func4_var, bg="pink", font=("Arial", 10),
command=self.toggle_func4_fields)
            self.chk_function4.grid(row=4, column=0, sticky="w")

        def toggle_func2_fields(self):
            # display or hide the input field for the shift parameter
            if self.func2_var.get():
                  self.label_2 = tk.Label(self.checkbox_frame, text="The shift
parameter:", bg="pink")
                self.label_2.grid(row=2, column=1, padx=10)
                self.entry_2 = tk.Entry(self.checkbox_frame, width=10)
                self.entry_2.grid(row=2, column=2, padx=5)
            else:
                if hasattr(self, 'label_2'):
                    self.label_2.grid_remove()
                if hasattr(self, 'entry_2'):
                    self.entry_2.grid_remove()


        def toggle_func4_fields(self):
            # display or hide input fields for Bilateral filter parameters
            if self.func4_var.get():
                  self.label1 = tk.Label(self.checkbox_frame, text="The filter
diameter (from 5 to 25):", bg="pink")
                self.label1.grid(row=4, column=1, padx=5)
                self.entry1 = tk.Entry(self.checkbox_frame, width=10)
                self.entry1.grid(row=4, column=2, padx=5)

                      self.label2  =  tk.Label(self.checkbox_frame,  text="The
sigmaColor value (from 50 to 200):", bg="pink")
                self.label2.grid(row=5, column=1, padx=5)
                self.entry2 = tk.Entry(self.checkbox_frame, width=10)
                self.entry2.grid(row=5, column=2, padx=5)
```

```
                        self.label3 = tk.Label(self.checkbox_frame, text="The
sigmaSpace value (from 50 to 200):", bg="pink")
                self.label3.grid(row=6, column=1, padx=5)
                self.entry3 = tk.Entry(self.checkbox_frame, width=10)
                self.entry3.grid(row=6, column=2, padx=5)
            else:
                if hasattr(self, 'label1'):
                    self.label1.grid_remove()
                if hasattr(self, 'entry1'):
                    self.entry1.grid_remove()
                if hasattr(self, 'label2'):
                    self.label2.grid_remove()
                if hasattr(self, 'entry2'):
                    self.entry2.grid_remove()
                if hasattr(self, 'label3'):
                    self.label3.grid_remove()
                if hasattr(self, 'entry3'):
                    self.entry3.grid_remove()


    # start the app
    if __name__ == "__main__":
        root = tk.Tk()
        app = VideoApp(root)
        root.mainloop()


    import cv2 # import library for working with videos
    import numpy as np # import library for working with mathematical
expressions and matrices


    def load_video(source=0):  # open the video stream from the specified
source (file or webcam)
        cap = cv2.VideoCapture(source)
        if not cap.isOpened():  # checking the opening of a video
            print("Error: Could not open video source")
            return None
        return cap
    # converting a frame to RGB
    def convert_to_rgb(frame):
        frame_rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        return frame_rgb
    # shifting for frames
    def video_shifting(frame, shear_factor=0.2):
        height, width = frame.shape[:2]
        shear_matrix = np.float32([[1, shear_factor, 0], [0, 1, 0]])
        new_width = int(width + abs(shear_factor) * height)
            sheared_frame = cv2.warpAffine(frame, shear_matrix, (new_width,
height))
        return sheared_frame
    # canny filter for frames
```

```
    def filter_canny(frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        edges_low = cv2.Canny(gray, 50, 150)
        edges_mid = cv2.Canny(gray, 100, 200)
        edges_high = cv2.Canny(gray, 150, 250)
        combined = cv2.hconcat([edges_low, edges_mid, edges_high])
        combined_rgb = cv2.cvtColor(combined, cv2.COLOR_GRAY2BGR)
        return combined_rgb
    # bilateral filter for frames
    def filter_bilateral(frame, diameter=15, sigma_color=75, sigma_space=75):
        return cv2.bilateralFilter(frame, diameter, sigma_color, sigma_space)


    def play_video(source=0):
         cap = load_video(source) # open the video stream from the specified
source (file or webcam)
        if cap is None:
            return
```

ДОДАТОК Б
Екранні форми виконання

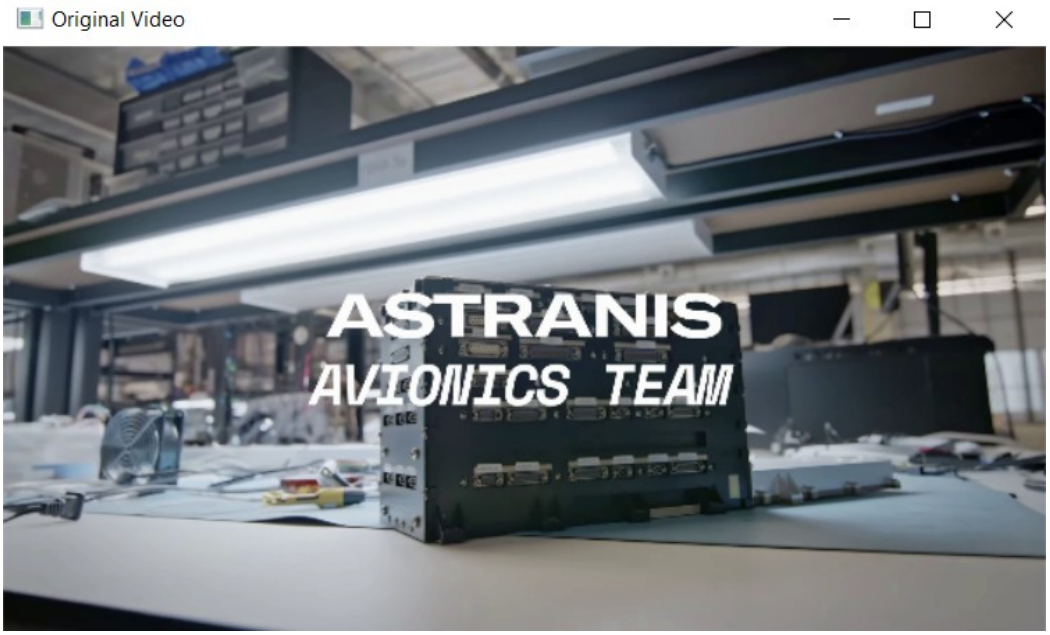

Рисунок Б.1 – Зчитування відео з файлу

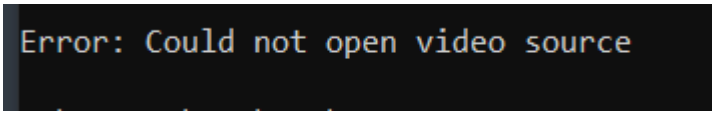

Рисунок Б.2 – Зчитування відео з веб-камери

Рисунок Б.3 – Перетворення у RGB простір

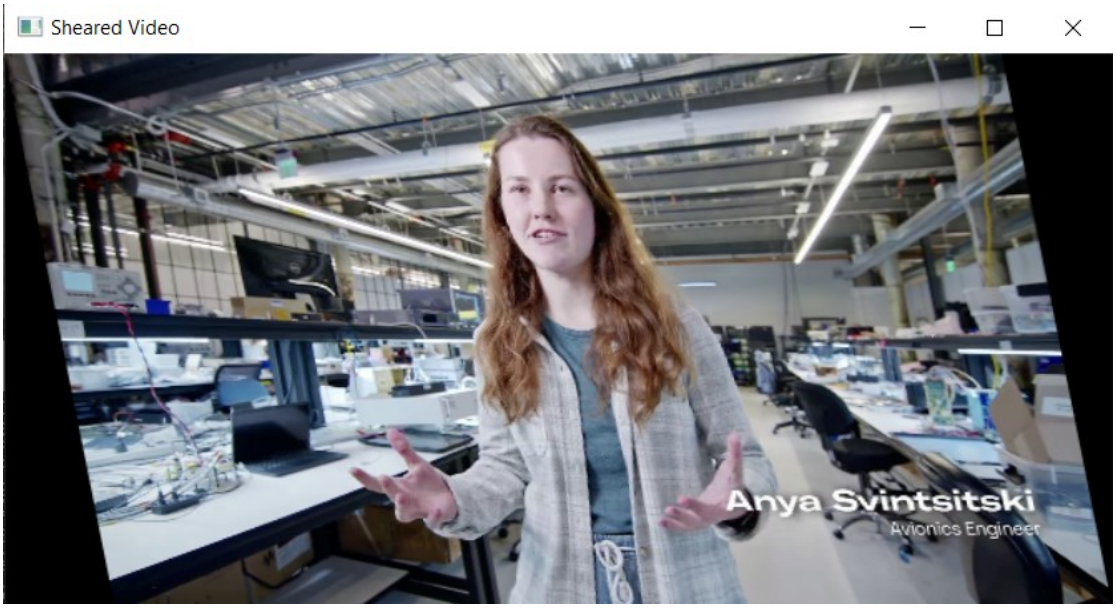

Рисунок Б.4 – Скіс відеозображення (при коефіцієнті 0.2)

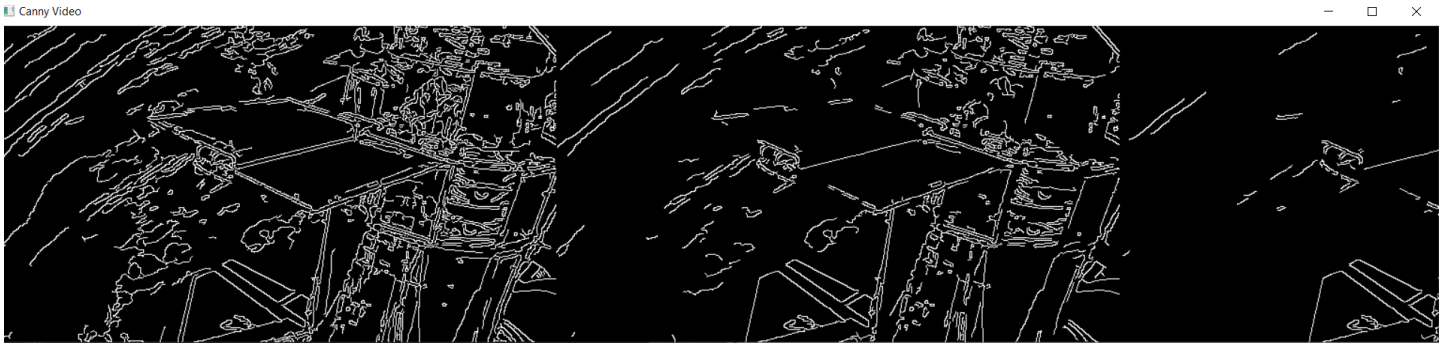

Рисунок Б.5 – Фільтрація Canny

Рисунок Б.6 – Перетворення з білатеральним фільтром (при коефіцієнтах 15, 140, 150)

Рисунок Б.7 – Екран програми з інтерфейсом

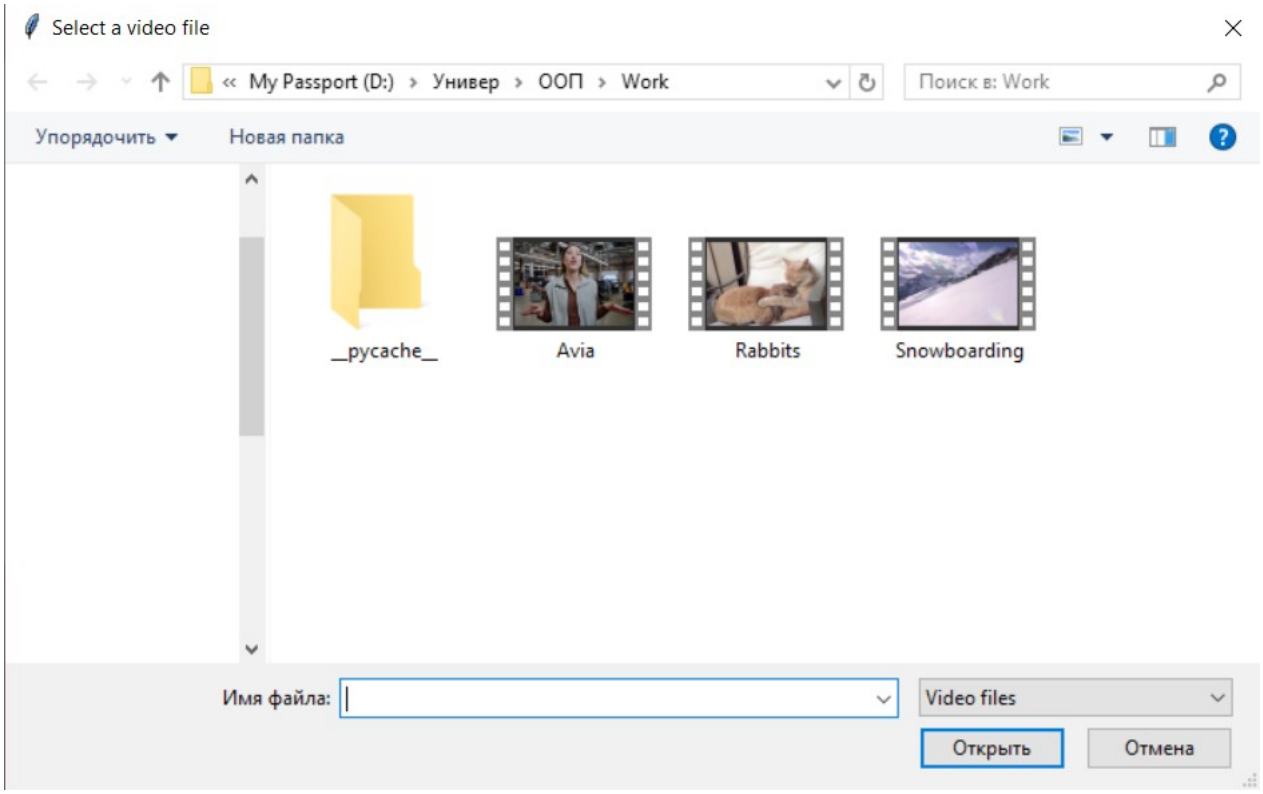

Рисунок Б.8 – Діалогове віко для відкриття відео

Рисунок Б.9 – Екран роботи при запуску відео

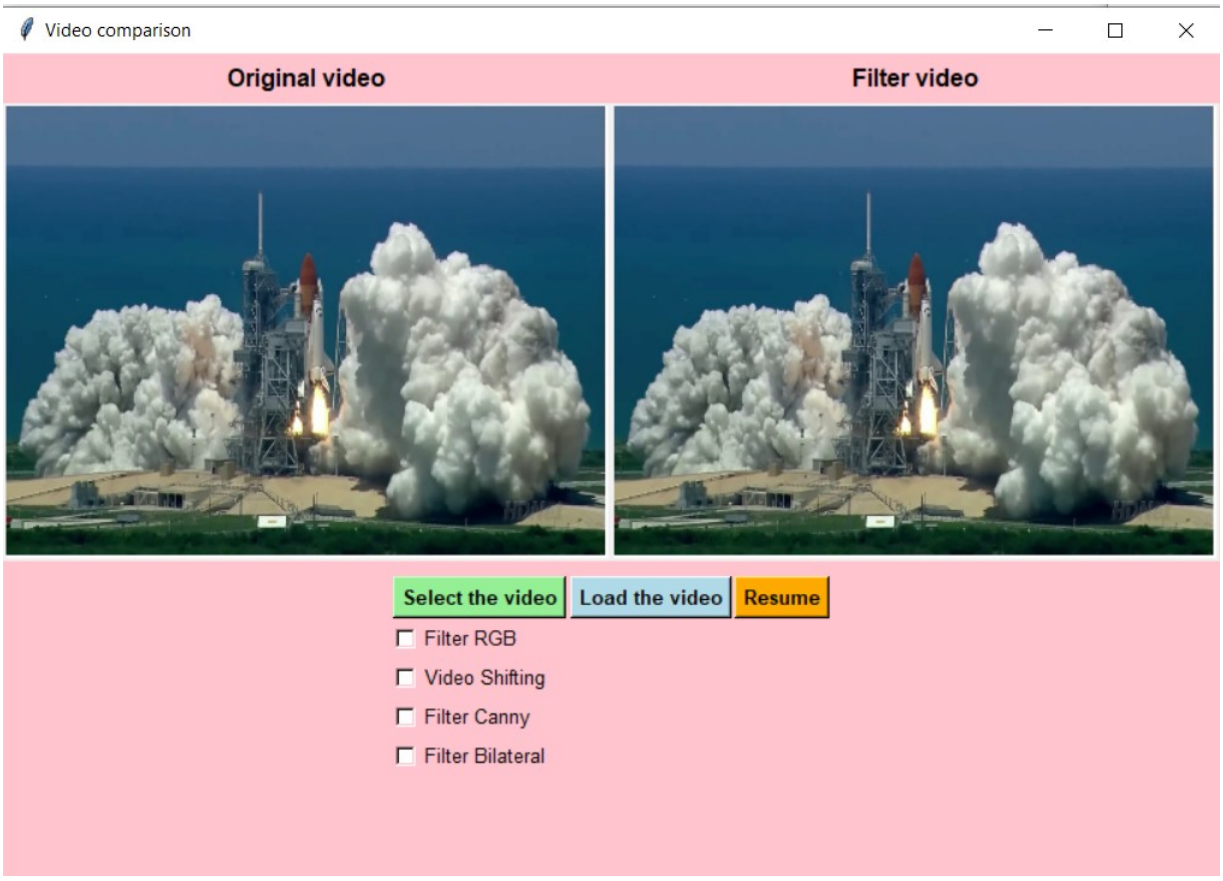

Рисунок Б.10 – Екран роботи при натисканні кнопки паузи

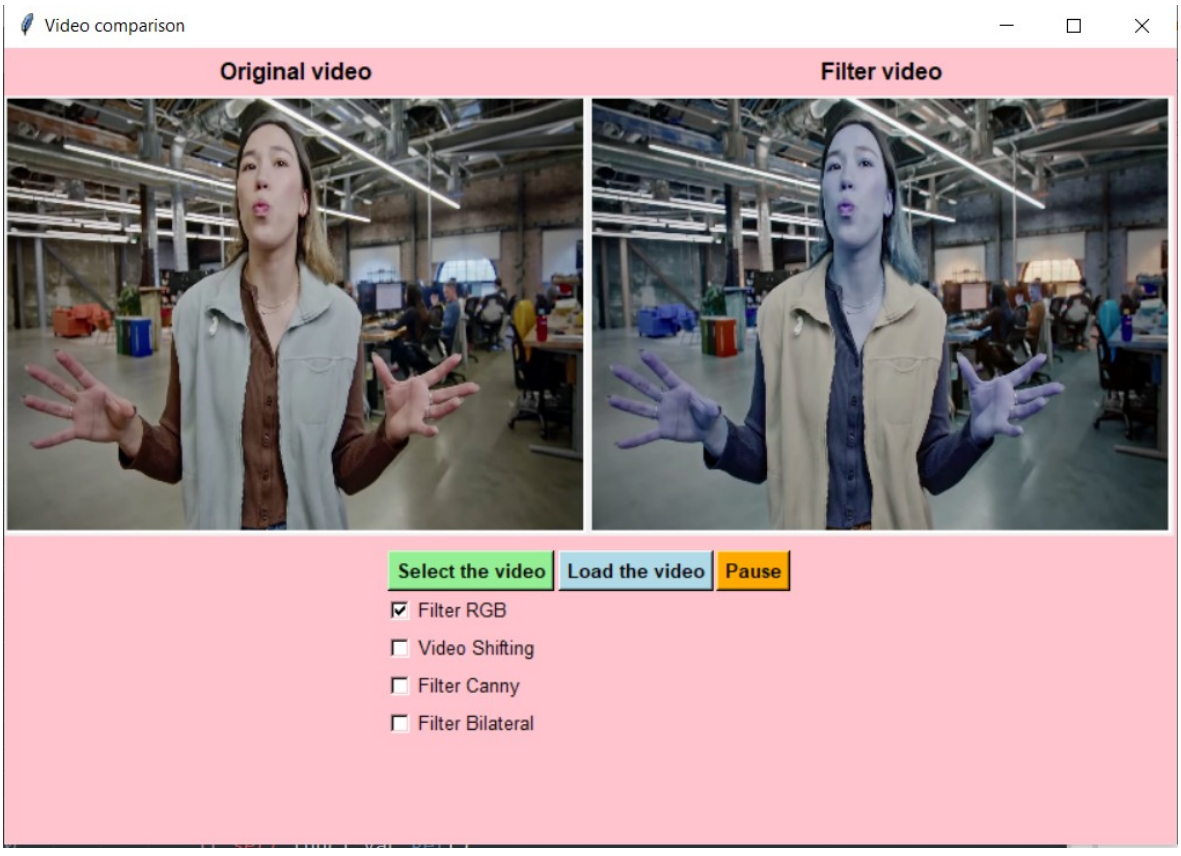

Рисунок Б.11 – Екран роботи при RGB фільтрації

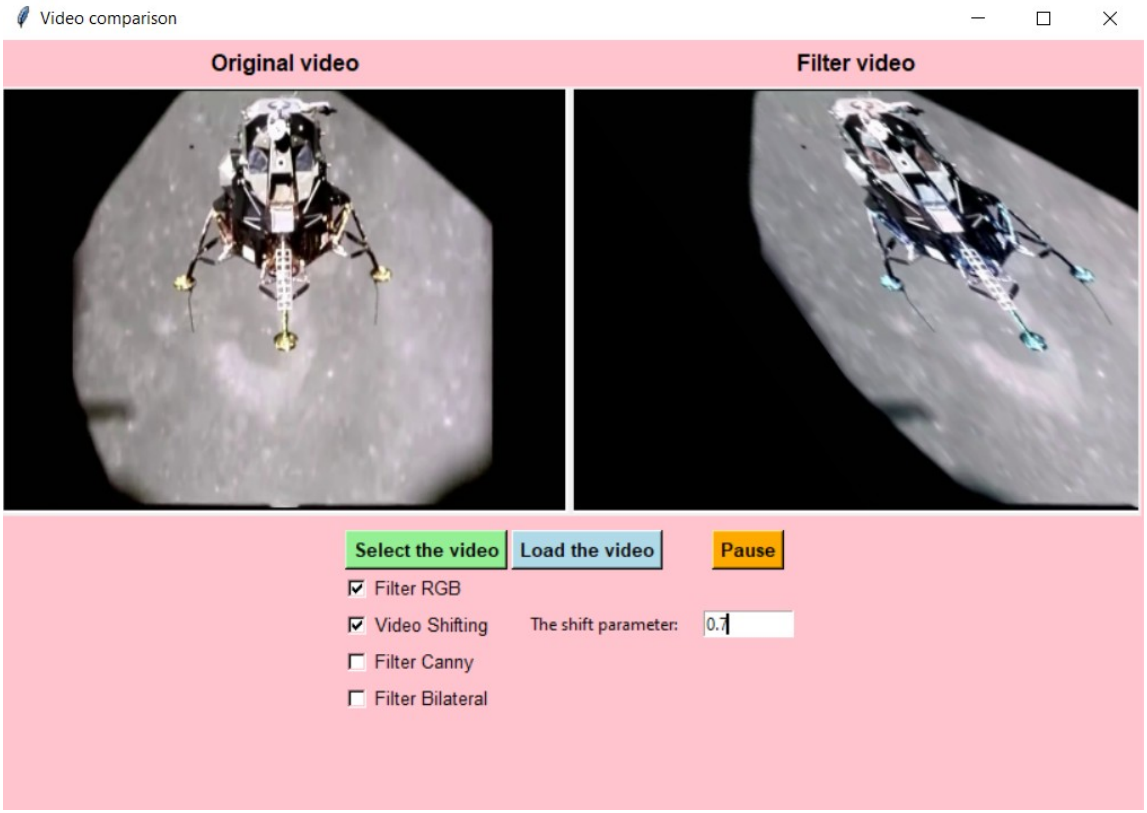

Рисунок Б.12 – Екран роботи при RGB фільтрації та нахилі

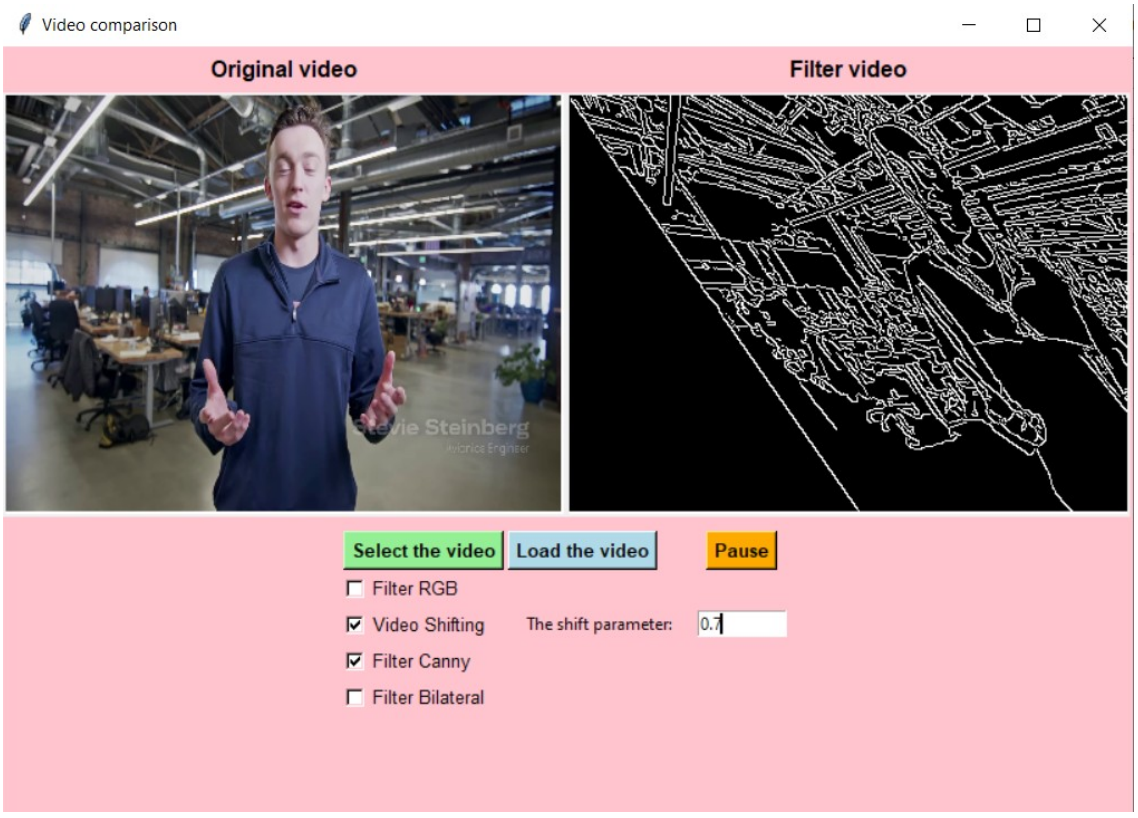

Рисунок Б.13 – Екран роботи при Canny фільтрації та нахилі

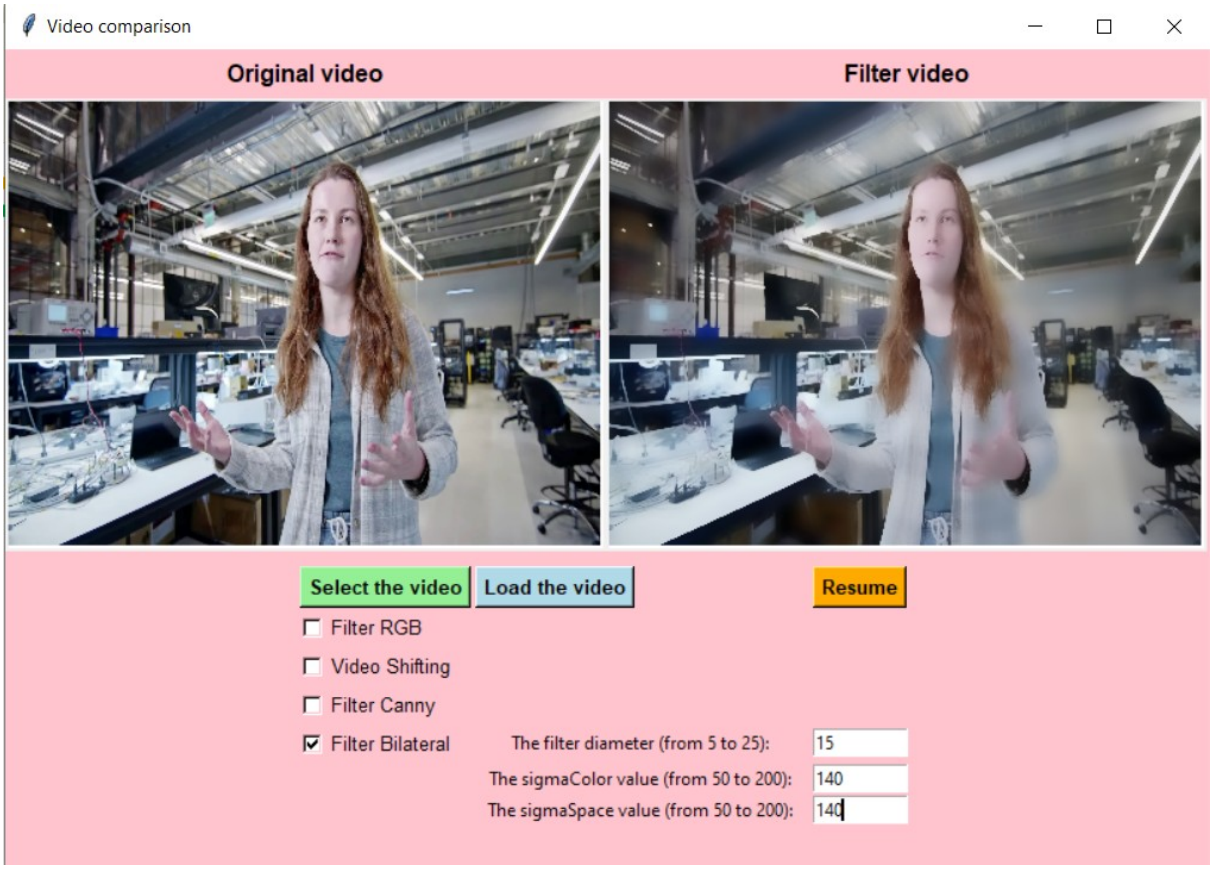

Рисунок Б.14 – Екран роботи при Білатеральній фільтрації