

**Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки**

Лабораторна робота №3
з дисципліни
«Технології Computer Vision»
на тему
«Дослідження алгоритмів формування та обробки векторних цифрових зображень»

Виконала:
Студентка групи ІМ-21
Кривохата Марія Юріївна
Номер у списку групи: 12

Перевірів: Баран Д. Р.

Київ 2024

Мета: Виявити дослідити та узагальнити особливості реалізації алгоритмів формування та обробки векторних цифрових зображень на прикладі застосування алгоритмів інтерполяції, апроксимації та згладжування складних 3D растрових об'єктів та застосування технологій видалення невидимих граней та ребер.

Завдання III рівень:

Здійснити синтез математичних моделей та розробити програмний скрипт, що забезпечує **реалізацію векторних алгоритмів над 2D, 3D графічними примітивами та з цифровими зображеннями.**

Технічні умови реалізації завдання наведені у таблиці додатку.

Завдання I рівня – максимально 7 балів.

Здійснити виконання завдання лабораторної роботи із застосуванням алгоритму інтерполяції для побудови векторного зображення 2D, 3D графічного об'єкту.

Завдання II рівня – максимально 8 балів.

Здійснити виконання завдання лабораторної роботи із застосуванням алгоритму інтерполяції для побудови векторного зображення 2D, 3D графічного об'єкту та алгоритму видалення невидимих ліній та поверхонь.

Завдання III рівня – максимально 9 балів.

Здійснити виконання завдання II рівня складності – програмний скрипт №1.

Реалізувати розробку програмного скрипта №2, що реалізує виділення контуру обраного об'єкту на цифровому растровому зображенні. За необхідності передбачити корекцію кольору цифрового растрового зображення для покращення якості виділення контуру обраного об'єкту.

Цифрове зображення обрати самостійно.

20	Відображення 3D фігури реалізується з використанням аксонометричної проекції будь-якого типу. Обрати самостійно: бібліотеку, розмір графічного вікна, розмір фігури, динаміку зміни положення фігури, кольорову гамму графічного об'єкту. Всі операції перетворень мають	Піраміда з трикутною основою. Метод інтерполяції: метод найменших квадратів. Метод видалення невидимих ліній та поверхонь: алгоритм Варнока.
	здійснюватися у межах графічного вікна.	

Результати виконання лабораторної роботи

Результати архітектурного проектування та математичні моделі:

1) Інтерполяція за допомогою методу найменших квадратів

Для реалізації цього завдання було використано можливості бібліотеки numpy. Загальний порядок дій у даній програмі такий:

1. **Застосовуємо перетворення до піраміди за допомогою матриць повороту та переносу** (математична модель та спосіб виконання взяті з першої лабораторної роботи)
Обертання забезпечує сегмент $T_{11}[3 \times 3]$ матриці перетворень T .

Довкола осі x:

$$T_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Довкола осі y:

$$T_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2. **Проектуємо піраміду на площину xy** (математична модель також взята з першої лабораторної роботи)

Узагальнена математична модель аксонометричної проєкції:

$$A' = [x, y, z, 1] * \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Проекція на xy

$$T_{xy}^{ort} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3. **Проходимося по всім ребрам та для кожного окремо застосовуємо алгоритм інтерполяції за допомогою МНК**

Мета методу найменших квадратів - знайти таку пряму $y = ax + b$, яка мінімізує суму квадратів відстаней між фактичними точками і відповідними значеннями на прямій. Тобто наша задача - знайти пряму, яка найкраще проходить між вершинами $p1$ з координатами $(x1, y1)$ та $p2$ з координатами $(x2, y2)$.

X – матриця вхідних значень. Перший стовпець містить значення координат x для кожної точки на прямій, а другий — одиниці.

$$X = \begin{bmatrix} x_1 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}$$

Вектор вихідних значень Y містить значення координат y для кожної точки:

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}$$

Для того, щоб знайти найточніші коефіцієнти a та b , потрібно мінімізувати помилку, яка розраховується за наступною формулою:

$$error = \sum_{i=1}^n (y_i - (ax_i + b))^2$$

А безпосередньо для знаходження a та b буде використовуватись функція бібліотеки `numpy`. Функція `np.linalg.lstsq(X, Y, rcond=None)` повертає розв'язок цієї задачі у вигляді коефіцієнтів a та b , де:

$$coeff = \begin{bmatrix} a \\ b \end{bmatrix}$$

Отримані коефіцієнти використовуються для обчислення значень y для кожного x -значення між x_1 та x_2 :

$$y = ax + b$$

Повний алгоритм виконання програми виглядає так:



2) Видалення невидимих ліній та поверхонь за алгоритмом Варнока

Принцип роботи цього алгоритму одночасно здається і дуже простим у своїй концепції, і складним для реалізації через необхідність врахувати всі можливі випадки.

Головна парадигма алгоритму - для обробки областей із малою кількістю інформації докладається мало зусиль і, навпаки, для областей з великим інформаційним змістом витрачаються значні ресурси.


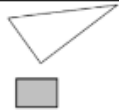
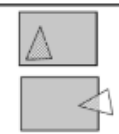
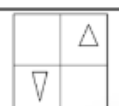
Діюча сутність – активне вікно, геометрична область, яка переміщується вздовж площини проекції та розбиває її на вікна меншого розміру.

Вміст активного (прямокутника) вікна аналізується на взаємне розташування з проекціями елементів 3D-сцени (з багатокутниками проекцій граней).

Процедура прийняття рішення застосовується для кожної малої області. Якщо вікно порожнє або його вміст простий, то відбувається візуалізація вікна. Якщо ця умова не виконується, то вікно розбивається на частини доти, поки вміст вікна не стане достатньо простим для аналізу. Декомпозиція вікна може відбуватись до пікселя.

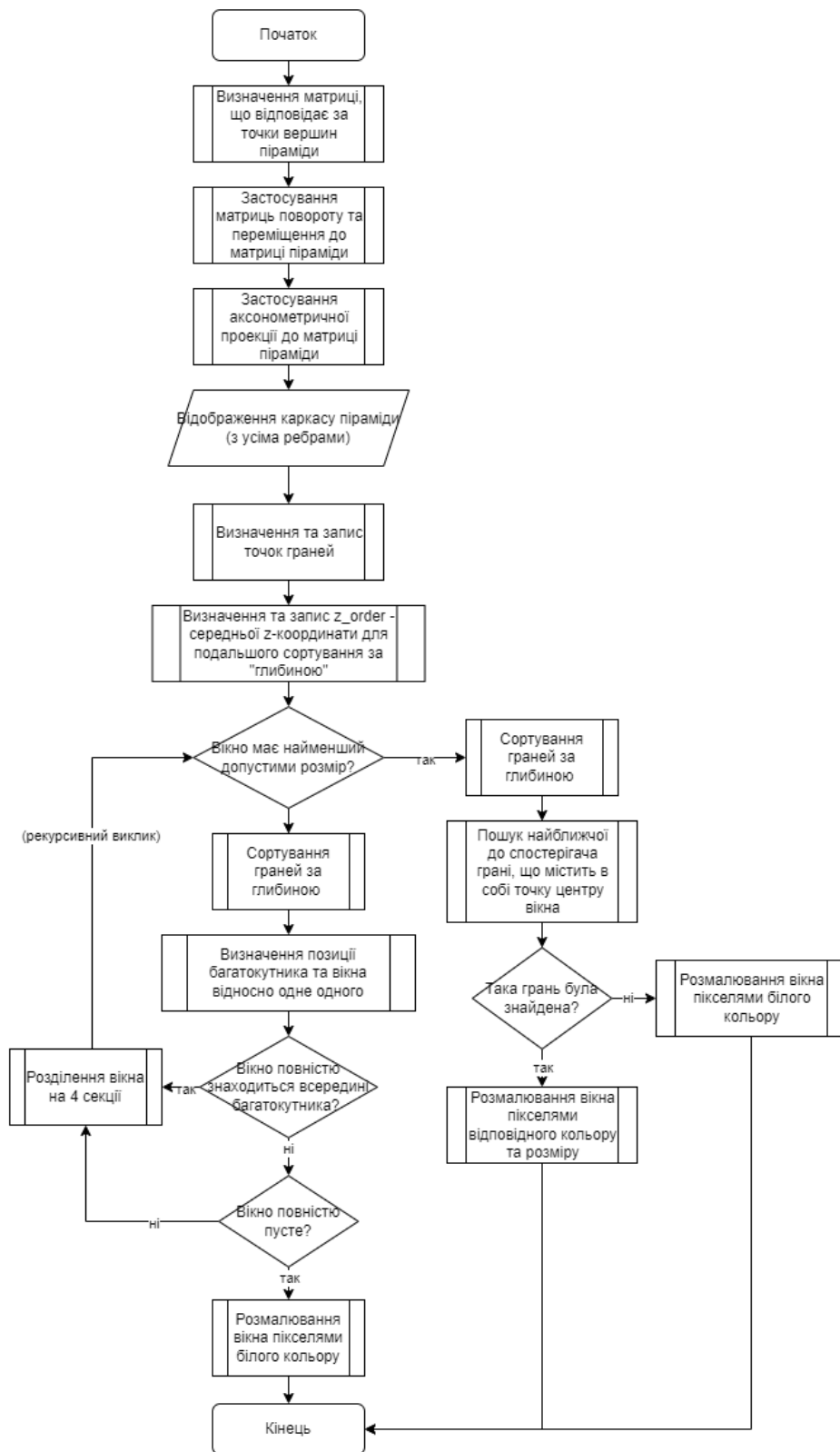
Чітко описати логіку цього алгоритму можна наступним чином:

Логіка алгоритму Варнока:

якщо вікно повністю покрите проекцією найближчого до спостерігача багатокутника (багатокутник охоплює вікно), то вікно зафарбовується кольором проекції цього багатокутника;	
якщо до вікна не потрапила жодна з проекцій елементів сцени (всі багатокутники сцени зовнішні відносно вікна), то його зафарбовують кольором фону;	
якщо існує єдиний внутрішній (багатокутник повністю знаходиться у вікні) або єдиний перетинаючий багатокутник , то все вікно зафарбовується кольором фону, а потім частина вікна, яка відповідає багатокутнику, зафарбовується основним кольором;	
якщо не виконалась жодна з умов , то вікно розбивається на чотири частини та для кожної з них повторюються попередні перевірки.	

(Скріншот взятий із лекції. На сторонніх ресурсах цей алгоритм описаний занадто заплутаними та складними для розуміння концепціями)

Через таке врахування всіх можливих випадків, алгоритм виконання є досить непростим:



Алгоритм Варнока

3) Виділення контуру об'єкта на цифровому растровому зображенні

Для того, щоб виділити контур на растровому зображенні, скористаємось бібліотекою `pylab` та таким порядком дій:

1. **Застосуємо фільтр `топо` до картинки** (для кожного пікселя визначаємо, чи він ближчий до чорного, чи до білого кольору)

Для цього буде застосована наступна математична модель

Піксель фарбується у білий колір, якщо виконується така умова:

$S > \text{threshold}$, де

$$S = R + G + B$$

$$\text{threshold} = \text{round}((255 + \text{factor}) / 2) * 3$$

R, G, B – червона, зелена та синя компоненти картинки

factor – коефіцієнт ефекта `топо`, введений користувачем

round – функція округлення

Піксель фарбується у чорний колір якщо виконується така умова:

$S \leq \text{threshold}$

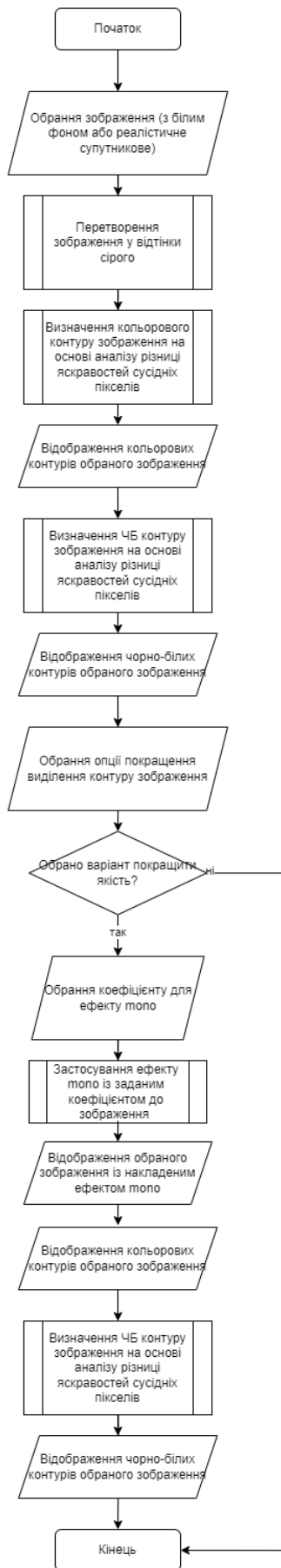
2. **Проаналізуємо зміни яскравості між сусідніми пікселями**

Цей етап виконується за допомогою можливостей бібліотеки `matplotlib` та її функції `contour`

3. **Виділимо контур на основі проробленого аналізу**

Окрім цього, надамо користувачу можливість покращити якість виділення контуру за допомогою налаштування різних коефіцієнтів `топо` фільтра. Спочатку будемо виводити непокращену картинку, а далі вже давати вибір із її покращенням. Також надамо можливість користувачу обрати серед двох зображень – перше містить фігуру на білому фоні, тому контури там можна розпізнати легше а друге містить реальне супутникове зображення.

Повний алгоритм виконання програми виглядатиме так:



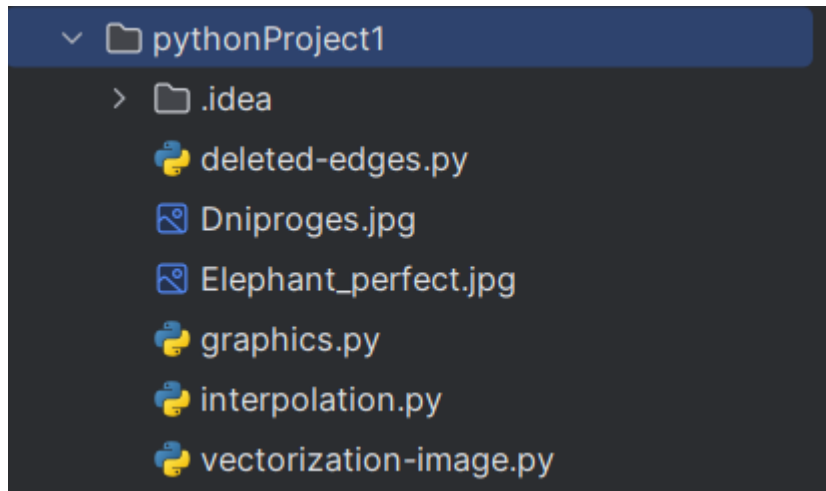
Опис структури проекту програми:

interpolation.py – реалізація задачі першого рівня, інтерполяція за допомогою алгоритму МНК

deleted-edges.py – реалізація задачі другого рівня, прибирання невидимих граней піраміди

vectorization-image.py – реалізація задачі третього рівня, векторизація реального зображення

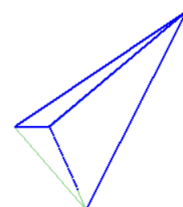
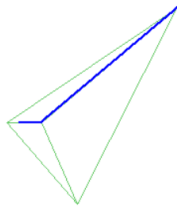
Dniproges.jpg, Elephant_perfect.jpg – картинки, що використовуються для показу роботи програми третього рівня

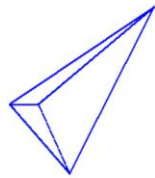


Результати роботи програми відповідно до завдання:

1) Інтерполяція за допомогою методу найменших квадратів

Програма виводить у вікні анімацію, де поступово промальовуються інтерпольовані контури



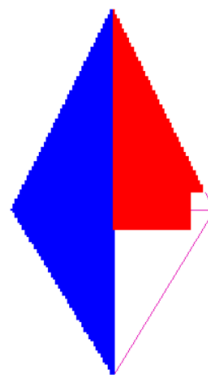
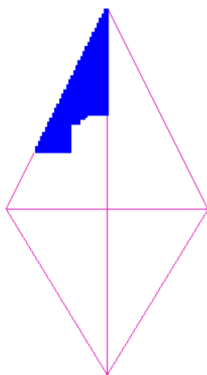


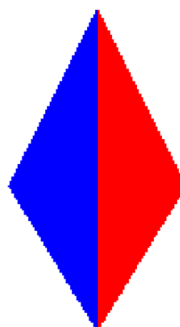
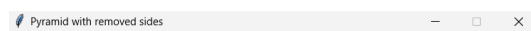
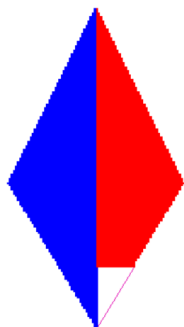
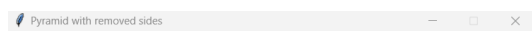
2) Видалення невидимих ліній та поверхонь за алгоритмом Варнока

Через особливість алгоритму та складної імплементації, замальовані сторони з'являються не моментально, тож, спостерігаючи за виконанням програми, можна чітко побачити прогрес розмальовки:

(рожевий контур необхідний для розуміння того, як піраміда виглядає до перетворень)

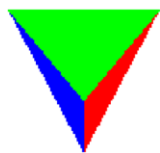
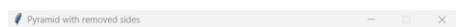
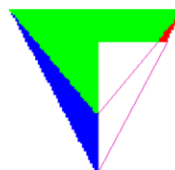
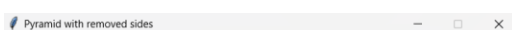
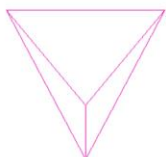
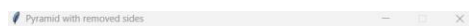
Випадок, коли видно 2 грані



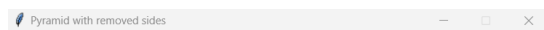
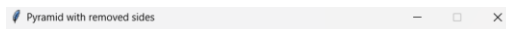
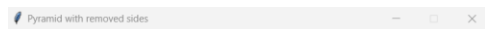


Подивимось також з інших ракурсів, щоб впевнитись, що програма працює правильно:

Випадок, коли видно 3 грані



Випадок, коли видно 1 грань

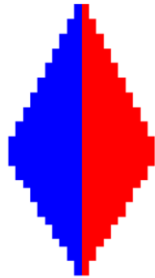


Цікавинка у програмі – коефіцієнт поділу вікна:

Додатково було додано параметр, який регулює величину прямокутників, з яких будується зображення піраміди (якщо казати алгоритмічною мовою, то цей параметр регулює ступінь декомпозиції вікна за алгоритмом Варнока). Збільшивши цей параметр, можна ще чіткіше побачити складові піраміди, з яких вона була намальована, бо вони стають більшими.

За замовчуванням цей параметр дорівнює 4 – число було знайдено методом експериментів і обране, бо саме воно забезпечувало чіткий вигляд контуру (зображення були наведені вище), але при цьому відображало “квадратність” складових. Якщо ми збільшимо цей параметр, наприклад, до 10, то отримаємо таке зображення:

Pyramid with removed sides



Дуже добре видно, як контур почав ставати все менш чітким і уривчастим через внесені зміни.

Причина додавання параметру

Під час виконання завдання, я якраз стикнулась із тою проблемою, що клітинки були занадто великі через особливості поділу вікна. Саме тому і було введено цей коефіцієнт.

Для контексту наводжу приклад того, яке зображення виводила початкова версія алгоритму, яка не мала цього коефіцієнта:

Pyramid with Warnock Algorithm



Ось такий був вид на піраміду зверху. Дуже футуристично, але математично максимально не точно. Якщо в одному вікні містились частинки двох або більше граней, то алгоритм вів себе непередбачувано і страждала не тільки “роздільна здатність” (розмір квадратів), але й вся логіка алгоритму.

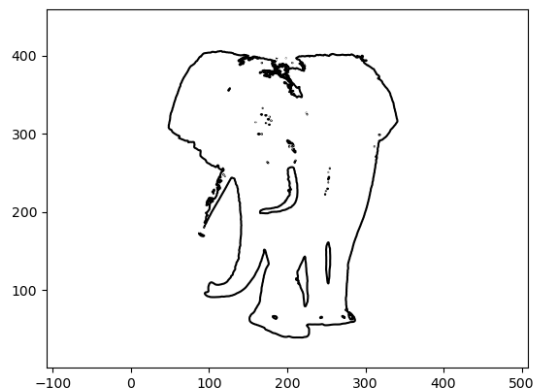
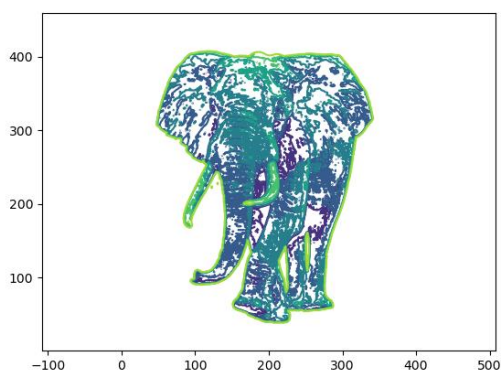
3) Виділення контуру об’єкту на цифровому растровому зображенні

1. Ідеальне зображення

Для тестування обробки ідеального зображення було обрано зображення слона на білому фоні. На цьому етапі перевіряємо як алгоритм поводить себе із зображеннями, де контур об’єкта дуже просто відділити від фону.



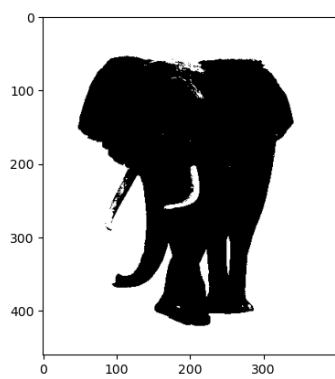
Першочерговий результат обробки:



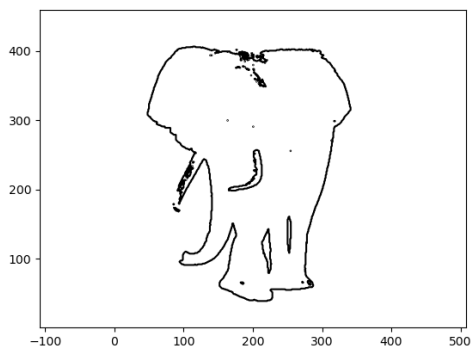
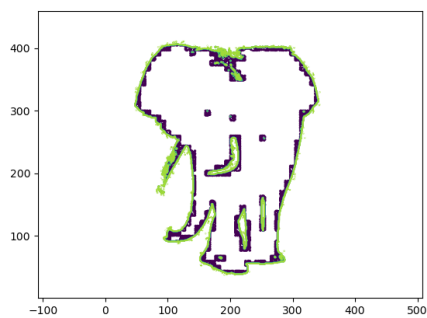
Спробуємо покращити якість виділення контуру за допомогою того, що спочатку перетворимо зображення у монохромне. При цьому використаємо достатньо велике значення монохромного фактору (100).

```
Enhance image quality?  
1 - Yes  
2 - No  
mode:1  
Enter mono factor  
factor:100  
Processing...
```

Монохром:



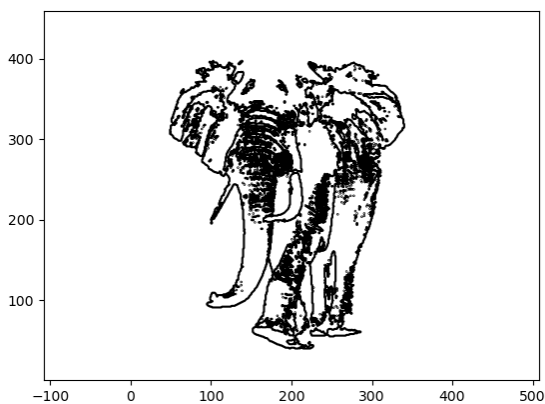
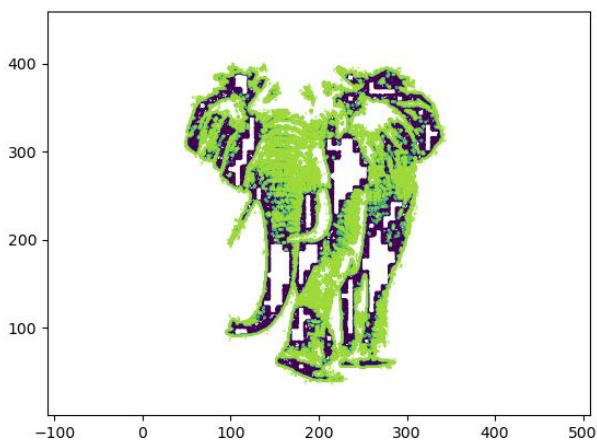
Фінальний варіант:



Бачимо, що порівняно із непокращеною картинкою, цей варіант має чіткіші лінії та не має зайвого шуму.

Тепер спробуємо використати від'ємний фактор монохром:

```
Enhance image quality?  
1 - Yes  
2 - No  
mode:1  
Enter mono factor  
factor:-50  
Processing...
```



Бачимо, що в такому випадку обводяться деталі всередині об'єкта. Контур залишається достатньо чітким, але з більшою кількістю шуму вже складніше обвести суцільну лінію довкола об'єкта.

Можемо впевнитись, що для отримання чіткого контуру без зайвих деталей всередині об'єкта варто використовувати великі значення монохромного фактору, а для відображення більшої кількості деталей – малі значення.

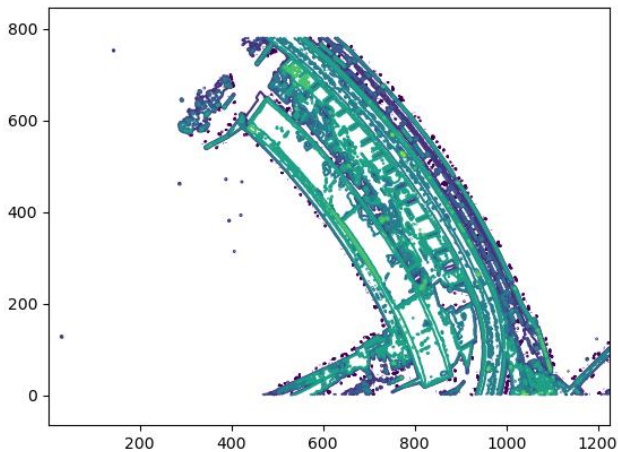
2. Реальне зображення

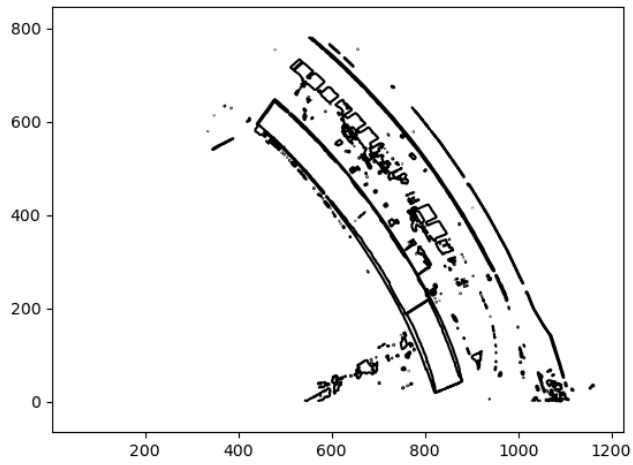
Для тестування виділення контуру об'єкта на реальному зображенні було обрано супутникове фото частини мого міста – Запоріжжя, а конкретніше – фото Дніпрогесу.



Методом перебору було знайдено коефіцієнт монохромного, який в результаті дає достатньо чіткі контури.

Без покращення:

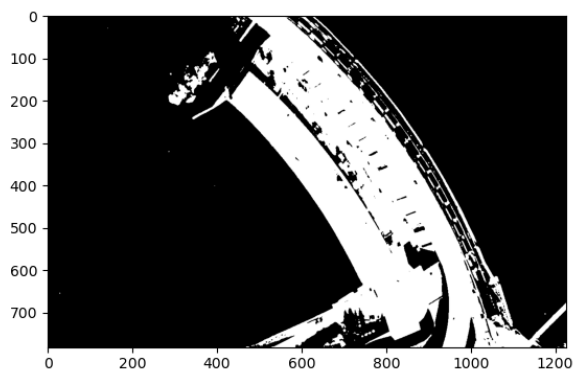




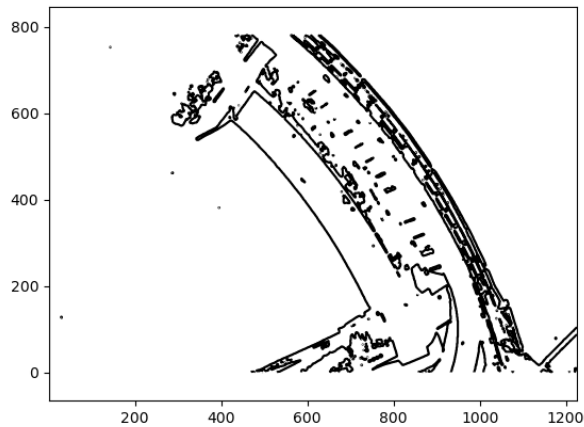
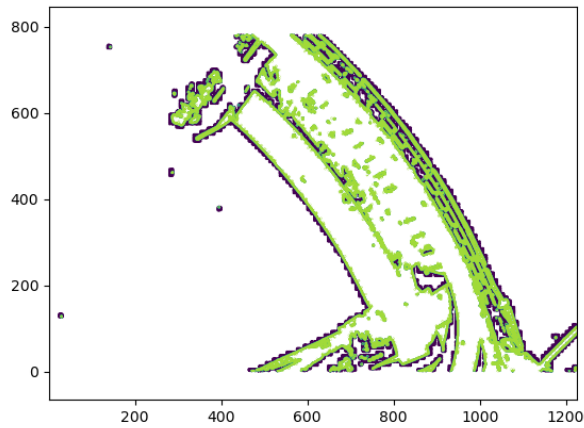
Із покращенням:

```
Enhance image quality?  
1 - Yes  
2 - No  
mode:1  
Enter mono factor  
factor:-135  
Processing...
```

Монохром



Фінальний варіант:



Програмний код, що забезпечує отримання результату:

1) Інтерполяція за допомогою методу найменших квадратів

```
from graphics import *
import numpy as np
import time

def init_window(text, x_wind, y_wind):
    wind = GraphWin(text, x_wind, y_wind)
    wind.setBackground('white')
    return wind

def init_pyramid_points(base_size, height):
    half_base = base_size / 2
    return np.array([
        [0, 0, height, 1], # apex
        [half_base, 0, -half_base, 1], # base vertices
        [-half_base, 0, -half_base, 1],
        [0, base_size, -half_base, 1]
    ])

def project_xy(pyramid):
    projection matrix = np.array([
```

```

        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(projection_matrix.T)

def shift_xyz(pyramid, dx, dy, dz):
    shift_matrix = np.array([
        [1, 0, 0, dx],
        [0, 1, 0, dy],
        [0, 0, 1, dz],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(shift_matrix.T)

def rotate_x(pyramid, angle_degree):
    angle_rad = np.radians(angle_degree)
    rotation_matrix = np.array([
        [1, 0, 0, 0],
        [0, np.cos(angle_rad), np.sin(angle_rad), 0],
        [0, -np.sin(angle_rad), np.cos(angle_rad), 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(rotation_matrix.T)

def rotate_y(pyramid, angle_degree):
    angle_rad = np.radians(angle_degree)
    rotation_matrix = np.array([
        [np.cos(angle_rad), 0, -np.sin(angle_rad), 0],
        [0, 1, 0, 0],
        [np.sin(angle_rad), 0, np.cos(angle_rad), 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(rotation_matrix.T)

def draw_initial_pyramid(projection_xy, color):
    x1, y1 = projection_xy[0, 0], projection_xy[0, 1]
    x2, y2 = projection_xy[1, 0], projection_xy[1, 1]
    x3, y3 = projection_xy[2, 0], projection_xy[2, 1]
    x4, y4 = projection_xy[3, 0], projection_xy[3, 1]

    sides = [
        Polygon(Point(x1, y1), Point(x2, y2), Point(x3, y3)),
        Polygon(Point(x1, y1), Point(x2, y2), Point(x4, y4)),
        Polygon(Point(x1, y1), Point(x3, y3), Point(x4, y4)),
        Polygon(Point(x2, y2), Point(x3, y3), Point(x4, y4))
    ]

    for side in sides:
        side.draw(wind)
        side.setOutline(color)

def interpolate_edges(p1, p2):

```

```

x1, y1 = int(p1[0]), int(p1[1])
x2, y2 = int(p2[0]), int(p2[1])

# constructing least-squares line fitting
n_points = abs(x2 - x1) + 1
X = np.vstack([np.linspace(x1, x2, n_points), np.ones(n_points)]).T
Y = np.linspace(y1, y2, n_points).reshape(-1, 1)

# applying the least-squares method
coeff, _, _, _ = np.linalg.lstsq(X, Y, rcond=None)
line_x = np.linspace(x1, x2, n_points)
line_y = coeff[0] * line_x + coeff[1]

# drawing interpolated points
for (x, y) in zip(line_x.astype(int), line_y.astype(int)):
    point = Point(x, y)
    point.setFill("blue")
    point.draw(wind)
    time.sleep(0.005) # animation delay

def render_pyramid(pyramid_2d):
    edges = [
        (pyramid_2d[0], pyramid_2d[1]), # AB
        (pyramid_2d[1], pyramid_2d[2]), # BC
        (pyramid_2d[2], pyramid_2d[0]), # CA
        (pyramid_2d[0], pyramid_2d[3]), # AD
        (pyramid_2d[1], pyramid_2d[3]), # BD
        (pyramid_2d[2], pyramid_2d[3]) # CD
    ]
    for edge in edges:
        interpolate_edges(*edge)

if __name__ == '__main__':
    win_width = 600
    win_height = 600

    center_x = win_width / 2
    center_y = win_height / 2

    base_size = 100
    height = 150

    wind = init_window('Interpolation usage on the pyramid', win_width,
win_height)
    pyramid = init_pyramid_points(base_size, height)

    rotated_pyramid_x = rotate_x(pyramid, -35)
    rotated_pyramid_xy = rotate_y(rotated_pyramid_x, -70)

    # shifting pyramid to center
    pyramid_centered = shift_xyz(rotated_pyramid_xy, center_x, center_y, 0)

    # projecting onto xy
    projected_pyramid = project_xy(pyramid_centered)

    # drawing initial pyramid
    draw_initial_pyramid(projected_pyramid, '#63c963')

```

```
# drawing interpolated pyramid
render_pyramid(projected_pyramid)

wind.getMouse()
wind.close()
```

2) Видалення невидимих ліній та поверхонь за алгоритмом Варнока

Так як реалізація програми досить складна, то в програмі міститься багато коментарів для чіткого розуміння процесів, що відбуваються.

```
3) from graphics import *
import numpy as np
from enum import Enum

def init_window(text, x_wind, y_wind):
    wind = GraphWin(text, x_wind, y_wind)
    wind.setBackground('white')
    return wind

def init_pyramid_points(base_size, height):
    half_base = base_size / 2
    return np.array([
        [0, 0, height, 1], # apex
        [half_base, 0, -half_base, 1], # base vertices
        [-half_base, 0, -half_base, 1],
        [0, base_size, -half_base, 1]
    ])

def project_xy(pyramid):
    projection_matrix = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(projection_matrix.T)

def shift_xyz(pyramid, dx, dy, dz):
    shift_matrix = np.array([
        [1, 0, 0, dx],
        [0, 1, 0, dy],
        [0, 0, 1, dz],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(shift_matrix.T)

def rotate_x(pyramid, angle_degree):
    angle_rad = np.radians(angle_degree)
    rotation_matrix = np.array([
        [1, 0, 0, 0],
        [0, np.cos(angle_rad), np.sin(angle_rad), 0],
        [0, -np.sin(angle_rad), np.cos(angle_rad), 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(rotation_matrix.T)
```

```

def rotate_y(pyramid, angle_degree):
    angle_rad = np.radians(angle_degree)
    rotation_matrix = np.array([
        [np.cos(angle_rad), 0, -np.sin(angle_rad), 0],
        [0, 1, 0, 0],
        [np.sin(angle_rad), 0, np.cos(angle_rad), 0],
        [0, 0, 0, 1]
    ])
    return pyramid.dot(rotation_matrix.T)

def draw_initial_pyramid(projection_xy, color, wind):
    x1, y1 = projection_xy[0, 0], projection_xy[0, 1]
    x2, y2 = projection_xy[1, 0], projection_xy[1, 1]
    x3, y3 = projection_xy[2, 0], projection_xy[2, 1]
    x4, y4 = projection_xy[3, 0], projection_xy[3, 1]

    sides = [
        Polygon(Point(x1, y1), Point(x2, y2), Point(x3, y3)),
        Polygon(Point(x1, y1), Point(x2, y2), Point(x4, y4)),
        Polygon(Point(x1, y1), Point(x3, y3), Point(x4, y4)),
        Polygon(Point(x2, y2), Point(x3, y3), Point(x4, y4))
    ]

    for side in sides:
        side.draw(wind)
        side.setOutline(color)

class PolygonPosition(Enum):
    SURROUNDING = 1 # Polygon completely contains the window
    INSIDE = 2 # Polygon is completely inside the window
    INTERSECTING = 3 # Polygon partially intersects the window
    OUTSIDE = 4 # Polygon is completely outside the window

class Window:
    def __init__(self, x_min, y_min, x_max, y_max):
        self.x_min = x_min
        self.y_min = y_min
        self.x_max = x_max
        self.y_max = y_max

    def split(self):
        x_mid = (self.x_min + self.x_max) / 2
        y_mid = (self.y_min + self.y_max) / 2

        # Creating four quadrants
        return [
            Window(self.x_min, self.y_min, x_mid, y_mid), # Bottom-left
            Window(x_mid, self.y_min, self.x_max, y_mid), # Bottom-right
            Window(self.x_min, y_mid, x_mid, self.y_max), # Top-left
            Window(x_mid, y_mid, self.x_max, self.y_max) # Top-right
        ]

class Face:
    def __init__(self, points, color, z_order):
        self.points = points

```

```

        self.color = color
        # Average z-coordinate for depth sorting
        self.z_order = z_order

def get_polygon_position(polygon_points, window):
    # Checking if all polygon points are inside the window
    all_inside = all(
        window.x_min <= x <= window.x_max and
        window.y_min <= y <= window.y_max
        for x, y in polygon_points
    )

    if all_inside:
        return PolygonPosition.INSIDE

    window_corners = [
        (window.x_min, window.y_min), # Bottom-left
        (window.x_max, window.y_min), # Bottom-right
        (window.x_min, window.y_max), # Top-left
        (window.x_max, window.y_max)  # Top-right
    ]

    # Checking whether all window corners inside polygon (polygon
    surrounds the window)
    if all(point_in_polygon(corner, polygon_points) for corner in
    window_corners):
        return PolygonPosition.SURROUNDING

    # Checking whether polygon intersects with window
    if polygons_intersect(polygon_points, window_corners):
        return PolygonPosition.INTERSECTING

    return PolygonPosition.OUTSIDE

def point_in_polygon(point, polygon):
    """
    1. Casting a ray from the point to the right
    2. Counting number of times it intersects polygon edges
    3. If count is odd, point is inside; if even, point is outside
    """
    x, y = point
    n = len(polygon)
    is_inside = False

    j = n - 1
    for i in range(n):
        # Checking if ray cast from point intersects with polygon edge
        if ((polygon[i][1] > y) != (polygon[j][1] > y)) and
            (x < (polygon[j][0] - polygon[i][0]) * (y - polygon[i][1])
            /
            (polygon[j][1] - polygon[i][1]) + polygon[i][0])):
            is_inside = not is_inside
        j = i

    return is_inside

```



```

def polygons_intersect(poly1, poly2):
    for i in range(len(poly1)):
        for j in range(len(poly2)):
            if lines_intersect(
                poly1[i], poly1[(i + 1) % len(poly1)],
                poly2[j], poly2[(j + 1) % len(poly2)]
            ):
                return True # edges from poly1 and poly2 intersect
    return False # edges from poly1 and poly2 do not intersect

def lines_intersect(p1, p2, p3, p4):
    def ccw(A, B, C):
        return (C[1] - A[1]) * (B[0] - A[0]) > (B[1] - A[1]) * (C[0] - A[0])

    # Two lines intersect if points of one line are on opposite sides of
    the other line
    return ccw(p1, p3, p4) != ccw(p2, p3, p4) and ccw(p1, p2, p3) != ccw(p1, p2, p4)

def draw_pixels(window, color, graphics_window, pixel_size=2):
    # Drawing filled pixels of specified size within the given window
    for x in range(int(window.x_min), int(window.x_max), pixel_size):
        for y in range(int(window.y_min), int(window.y_max), pixel_size):
            pixel = Rectangle(Point(x, y), Point(x + pixel_size, y + pixel_size))
            pixel.setFill(color)
            pixel.setOutline(color)
            pixel.draw(graphics_window)

def warnock_algorithm(window, faces, graphics_window, min_size=4):
    # Base case: window is at minimum size
    if (window.x_max - window.x_min <= min_size and
        window.y_max - window.y_min <= min_size):
        # Sorting faces by depth (front to back)
        sorted_faces = sorted(faces, key=lambda f: f.z_order,
reverse=True)
        center_x = (window.x_min + window.x_max) / 2
        center_y = (window.y_min + window.y_max) / 2

        # Finding the closest to front face containing window center
        for face in sorted_faces:
            if point_in_polygon((center_x, center_y), face.points):
                draw_pixels(window, face.color, graphics_window, min_size)
                return

        # Background color if no face found
        draw_pixels(window, "white", graphics_window, min_size)
        return

    sorted_faces = sorted(faces, key=lambda f: f.z_order, reverse=True)

    surrounding_faces = []
    inside_faces = []
    intersecting_faces = []

```

```

    for face in sorted_faces:
        position = get_polygon_position(face.points, window)

        if position == PolygonPosition.SURROUNDING:
            surrounding_faces.append(face)
        elif position == PolygonPosition.INSIDE:
            inside_faces.append(face)
        elif position == PolygonPosition.INTERSECTING:
            intersecting_faces.append(face)

    if surrounding_faces:
        # Window is completely covered, but subdividing for better
        resolution
        subwindows = window.split()
        for subwindow in subwindows:
            warnock_algorithm(subwindow, sorted_faces, graphics_window,
min_size)
    elif not (surrounding_faces or inside_faces or intersecting_faces):
        # Window is empty -> drawing background
        draw_pixels(window, "white", graphics_window, min_size)
    else:
        # Complex case -> subdividing window and recursively calling the
        function
        subwindows = window.split()
        for subwindow in subwindows:
            warnock_algorithm(subwindow, sorted_faces, graphics_window,
min_size)

if __name__ == '__main__':

    win_width = 600
    win_height = 600

    center_x = win_width / 2
    center_y = win_height / 2

    base_size = 200
    height = 250

    wind = init_window('Pyramid with removed sides', win_width,
win_height)

    pyramid = init_pyramid_points(base_size, height)
    rotated_pyramid_x = rotate_x(pyramid, -35)
    rotated_pyramid_xy = rotate_y(rotated_pyramid_x, 0)
    pyramid_centered = shift_xyz(rotated_pyramid_xy, center_x, center_y,
0)
    projected_pyramid = project_xy(pyramid_centered)

    draw_initial_pyramid(projected_pyramid, "#e61bb8", wind)

    faces = []
    colors = ['#FF0000', '#00FF00', '#0000FF', '#FF00FF']

    for i in range(4):
        if i < 3:
            points = [
                (projected_pyramid[0][0], projected_pyramid[0][1]),

```

```

        (projected_pyramid[i + 1][0], projected_pyramid[i +
1][1]),
        (projected_pyramid[(i + 2) % 3 + 1][0],
projected_pyramid[(i + 2) % 3 + 1][1])
    ]
    else:
        points = [
            (projected_pyramid[1][0], projected_pyramid[1][1]),
            (projected_pyramid[2][0], projected_pyramid[2][1]),
            (projected_pyramid[3][0], projected_pyramid[3][1])
        ]

    if i < 3:
        z_order = (pyramid_centered[0][2] +
                    pyramid_centered[i + 1][2] +
                    pyramid_centered[(i + 2) % 3 + 1][2]) / 3
    else:
        z_order = (pyramid_centered[1][2] +
                    pyramid_centered[2][2] +
                    pyramid_centered[3][2]) / 3

    faces.append(Face(points, colors[i], z_order))

main_window = Window(0, 0, win_width, win_height)
warnock_algorithm(main_window, faces, wind, min_size=10) # Bigger
min_size -> bigger pyramid "pixels"

wind.getMouse()
wind.close()

```

4) Виділення контуру об'єкта на цифровому растровому зображенні

```

5) from PIL import Image, ImageDraw
from pylab import *
import sys

def vector_circuit(img):
    figure()
    contour(img, origin='image')
    axis('equal')
    show()
    contour(img, levels=[170], colors='black', origin='image')
    axis('equal')
    show()

    return

def mono(image):
    draw = ImageDraw.Draw(image)
    width = image.size[0]
    height = image.size[1]
    pix = image.load() # pixels' values

    print('Enter mono factor')
    factor = int(input('factor:'))

```

```

print('Processing...')
for i in range(width):
    for j in range(height):
        r = pix[i, j][0]
        g = pix[i, j][1]
        b = pix[i, j][2]
        S = r + g + b
        if (S > (((255 + factor) // 2) * 3)): # deciding what color
the pixel resembles more: white or black
            r, g, b = 255, 255, 255
        else:
            r, g, b = 0, 0, 0
        draw.point((i, j), (r, g, b))

plt.imshow(image)
plt.show()
image.save("mono-img.jpg", "JPEG")
del draw

return

if __name__ == '__main__':

    print("Select source image:")
    print("1 - Ideal image")
    print("2 - Real image")
    mode_1 = int(input("mode: "))

    if mode_1 == 1:
        im = array(Image.open('Elephant_perfect.jpg').convert('L'))
        image = Image.open("Elephant_perfect.jpg")
        vector_circuit(im)

    if mode_1 == 2:
        im = array(Image.open('Dniproges.jpg').convert('L'))
        image = Image.open("Dniproges.jpg")
        vector_circuit(im)

    print("Enhance image quality?")
    print("1 - Yes")
    print("2 - No")
    mode = int(input('mode:'))

    if mode == 1:
        mono(image)
        im = array(Image.open('mono-img.jpg').convert('L'))
        vector_circuit(im)

    if (mode == 2):
        sys.exit()

```

Висновки: виконавши лабораторну роботу №3, я виявила, дослідила та узагальнила особливості реалізації алгоритмів формування та обробки векторних цифрових зображень на прикладі застосування алгоритмів інтерполяції, технологій видалення невидимих граней та ребер та виділення контуру на растрових зображеннях.

Лабораторна робота була досить об'ємною та складною, тому я винесла для себе багато нових концепцій, повторила вже раніше мені відомі (МНК) та розібралась із усіма аспектами, що були незрозумілими.

Під час виконання першого завдання, я застосувала можливості бібліотеки `numpy` та швидко розібралась із задачею. Всі ребра піраміди інтерполюються коректно та відмальовуються як анімація поверх звичайного каркасу піраміди для кращого розуміння процесів, що відбуваються.

Під час виконання третього завдання, цікавою задачею був підбір коефіцієнтів ефекта `topo` та спостереження за тим, як саме він впливає на відображення контуру об'єкта. Також цікавим було порівняння зображення із реальним фоном та з білим. На зображенні з білим фоном не було необхідності настільки довго перебирати різні коефіцієнти, щоб добитись чіткого контуру, бо контраст між фігурою та фоном був великий.

Під час виконання другого завдання, я стикнулась із головною проблемою – складністю алгоритму Варнока. Якись версії програм працювали коректно, але не у всіх ракурсах піраміди, якись – взагалі не видаляли грані, якись – відображали пустий екран. *Але кінцева програма виконує завдання у повному обсязі, а також додатково має критерій ступеня поділу вікна.* Таке рішення потребувало немало часу та зусиль. На мою думку, алгоритм Варнока, не є найкращим вибором для реалізації такої задачі у співвідношенні сил до ефективності (алгоритм неефективно працює на об'ємних задачах, бо містить рекурсію і виконується довго). Але виконання цього завдання стало також цікавим досвідом і я впевнена, що знання цих концепцій знадобиться в подальшому для складніших сфер використання технології `computer vision`.

Отже, всі три програми були виконані у повному обсязі відповідно до завдань і весь матеріал був засвоєний.