Національний технічний університет України «КПІ ім. Ігоря Сікорського»
Факультет Інформатики та Обчислювальної Техніки

Кафедра інформаційних систем та технологій

Лабораторна робота №3

з дисципліни «Технології Computer Vision»

на тему

«ДОСЛІДЖЕННЯ АЛГОРИТМІВ ФОРМУВАННЯ ТА ОБРОБКИ РАСТРОВИХ ЦИФРОВИХ ЗОБРАЖЕНЬ»

Виконала:
студентка групи ІС-12
Павлова Софія

Перевірив:
Баран Д. Р.

Київ – 2024

1. Постановка задачі

Мета роботи:

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 II рівня:

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.

4	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 здійснюватися у межах графічного вікна.	Паралелепіпед Метод інтерполяції: кривими Безьє. Метод видалення невидимих ліній та поверхонь: алгоритм Варнока.

Завдання III рівня:

Виконати завдання II рівня складності – програмний скрипт №1. Реалізувати розробку програмного скрипта №2, що реалізує виділення контору обраного об’єкту на цифровому растровому зображенні. За необхідності передбачити корекцію кольору цифрового растрового зображення для покращення якості виділення контору обраного об’єкту. Цифрове зображення обрати самостійно.

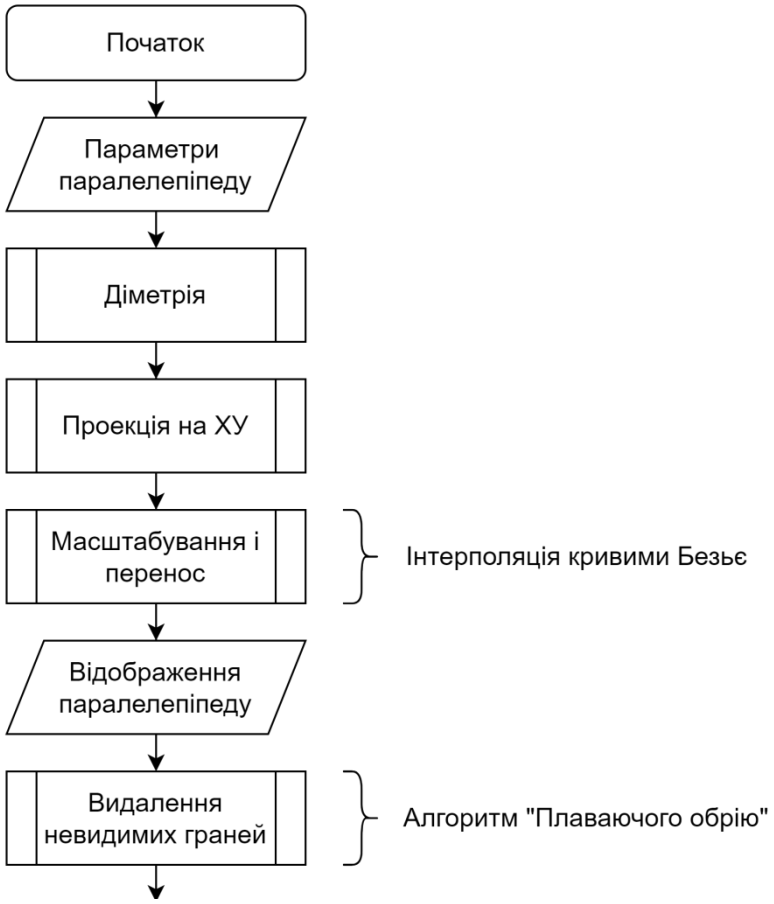
2. Виконання

2.1. Побудова векторного зображення 3D графічного об’єкту з видаленням невидимих граней

Алгоритм програми

За варіантом необхідно застосувати метод інтерполяції ***кривими Безьє*** та ***алгоритм Варнока*** для видалення невидимих граней паралелепіпеда. Але так, як алгоритм Варнока використовується для визначення видимих частин області у прямокутнику з точки зору її розташування відносно прямокутника. Ми можемо його використати, щоб визначити, які грані паралелепіпеда знаходяться в межах вікна анімації. Однак для визначення видимих граней доцільно використати інші алгоритми, наприклад алгоритм «***Плаваючого обрію***».

Розробимо скрипкове рішення, яке буде будувати векторне зображення паралелепіпеда, використовуючи інтерполяцію ***кривими Безьє*** та видалятиме невидимі грані фігури алгоритмом «***Плаваючого обрію***».



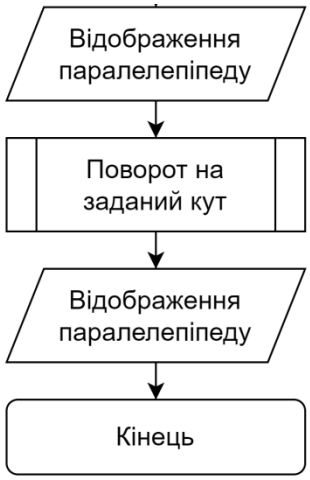


Рисунок 1 – Блок схема програми

Програмна реалізація

Використаємо скрипт для генерації паралелепіпеда, диметричної проекції та проекції на осі XY з лабораторної роботи №1.

Генерація фігури

Лістинг коду:

```
# Ініціалізація вікна
def init_window(xw, yw):
    win = GraphWin('Паралелепіпед', xw, yw)
    win.setBackground('white')

    return win

# Ініціалізація паралелепіпеда
def init_figure():
    # Вхідна матриця фігури
    prlpd = np.array([[0, 0, 0, 1],
                      [2, 0, 0, 1],
                      [2, 1, 0, 1],
                      [0, 1, 0, 1],
                      [0, 0, 2, 1],
                      [2, 0, 2, 1],
                      [2, 1, 2, 1],
                      [0, 1, 2, 1]])

    print('\nВхідна матриця')
    print(prlpd)

    return prlpd

# Блок головних викликів
if __name__ == '__main__':

    # Константи
    xw = 600  # ширина вікна
```

```
    yw = 600  # висота вікна

    # Головні виклики

    '''Монохром'''
    win = init_window(xw, yw)
    prlpd = init_figure()
```

Результат:

Бачимо вхідну матрицю паралелепіпеда.

```
Вхідна матриця
[[0 0 0 1]
 [2 0 0 1]
 [2 1 0 1]
 [0 1 0 1]
 [0 0 2 1]
 [2 0 2 1]
 [2 1 2 1]
 [0 1 2 1]]
```

Рисунок 1 – Вхідна матриця

Диметрична проекція

Лістинг коду:

```
# Аксонометрія
def dimetri(figure, teta_x, teta_y):
    teta_rx = (3 / 14 * teta_x) / 180
    teta_ry = (3 / 14 * teta_y) / 180

    f_1 = np.array(
        [[mt.cos(teta_rx),  0,  -mt.sin(teta_rx),   0],
         [0,                1,  0,                  0],
         [mt.sin(teta_rx),  0,  mt.cos(teta_rx),    1],
         [0,                0,  0,                  0]])
    ft_1 = f_1.T
    pr_xy_1 = figure.dot(ft_1)

    f_2 = np.array(
        [[1,    0,                  0,                  0],
         [0,    mt.cos(teta_ry),    mt.sin(teta_ry),    0],
         [0,    -mt.sin(teta_ry),   mt.cos(teta_ry),    0],
         [0,    0,                  0,                  1]])
    ft_2 = f_2.T
    pr_xy_2 = pr_xy_1.dot(ft_2)

    print('\nДиметрія')
    print(pr_xy_2)

    return pr_xy_2

# Блок головних викликів
if __name__ == '__main__':
```

```
    # Константи
    [...]

    teta_x = 160
    teta_y = 80

    # Головні виклики

    '''Монохром'''
    [...]
    # Диметрія
    prlpd_2 = dimetri(prlpd, teta_x, teta_y)
```

Результат:

Бачимо матрицю диметричної проекції.

```
Диметрія
[[ 0.          0.09509419  0.99546828  0.        ]
 [ 1.96382838  0.13110189  1.37240532  0.        ]
 [ 1.96382838  1.12657017  1.27731113  0.        ]
 [ 0.          1.09056247  0.90037409  0.        ]
 [-0.37865299  0.28184285  2.95039714  0.        ]
 [ 1.58517539  0.31785055  3.32733418  0.        ]
 [ 1.58517539  1.31331883  3.23223999  0.        ]
 [-0.37865299  1.27731113  2.85530295  0.        ]]
```

Рисунок 2 – Матриця диметричної проекції

Проекція на XY

Лістинг коду:

```
# Проекція на XY (Z=0)
def project_xy(figure):
    f = np.array(
        [[1, 0, 0, 0],
         [0, 1, 0, 0],
         [0, 0, 0, 0],
         [0, 0, 0, 1]])
    ft = f.T
    pr_xy = figure.dot(ft)

    print('\nПроекція на XY')
    print(pr_xy)

    return pr_xy

# Блок головних викликів
if __name__ == '__main__':

    # Константи
    [...]

    # Головні виклики

    '''Монохром'''
```

```
    [...]
    # Проекція на XY
    pr_xy_3 = project_xy(prlpd_2)
```

Результат:

Бачимо матрицю проекції на осі XY (Z=0).

```
Проекція на XY
[[ 0.          0.09509419  0.          0.        ]
 [ 1.96382838  0.13110189  0.          0.        ]
 [ 1.96382838  1.12657017  0.          0.        ]
 [ 0.          1.09056247  0.          0.        ]
 [-0.37865299  0.28184285  0.          0.        ]
 [ 1.58517539  0.31785055  0.          0.        ]
 [ 1.58517539  1.31331883  0.          0.        ]
 [-0.37865299  1.27731113  0.          0.        ]]
```

Рисунок 3 – Матриця проекції на XY

Масштабування та зміщення фігури

Видозмінимо функцію масштабування та зміщення фігури так, щоб вона використовувала ***криві Безьє*** для інтерполяції.

Математична модель (інтерполяція кривими Безьє)

Кубічна крива Безьє має вигляд:

$$R(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3, \qquad t \in [0;1]$$

у матричній формі:

$$R(t) = P \cdot M \cdot T,$$

$$R(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}, \qquad P = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \end{bmatrix}, \qquad M = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -1 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \qquad T = \begin{bmatrix} 1 \\ t \\ t^2 \\ t^3 \end{bmatrix}$$

Визначимо для цього окремий клас з методами:

- $evaluate(self, t)$ для обчислення нонвої точки на кривій;
- $scale(self, factor)$ для масштабування кривих Безьє;

- $translate(self, translation)$ для зміщення кривих Безьє;
- $bernstein(n, i, t)$ для обчислення значення ***функції Бернштейна*** для заданих параметрів.

Лістинг коду:

```
# Клас кривих Безьє
class BezierCurve:
    def __init__(self, control_points):
        self.control_points = control_points

    def evaluate(self, t):
        n = len(self.control_points) - 1
        result = np.zeros_like(self.control_points[0])
        for i in range(n + 1):
            result += self.control_points[i] * self.bernstein(n, i, t)
        return result

    def scale(self, factor):
        scaled_control_points = [self.control_points * factor]
        return BezierCurve(scaled_control_points)

    def translate(self, translation):
        translated_control_points = [point + translation for point in
self.control_points]
        return BezierCurve(translated_control_points)

    @staticmethod
    def bernstein(n, i, t):
        return scipy.special.comb(n, i) * (1 - t) ** (n - i) * t ** i

# Масштабування та зміщення за допомогою "кривих Безьє"
def scale_and_shift(prlpd, st, dx, dy, dz):
    # Створення пустої матриці для зберігання результату
    pr_xy = np.zeros_like(prlpd)

    for i in range(len(prlpd)):
        # Масштабування кожної точки
        bezier_x = BezierCurve(prlpd[i][0]).scale(st).translate(dx)
        bezier_y = BezierCurve(prlpd[i][1]).scale(st).translate(dy)
        bezier_z = BezierCurve(prlpd[i][2]).scale(st).translate(dz)

        # Отримання нових координат
        pr_xy[i][0] = bezier_x.evaluate(0.5)  # Вибір середньої точки
        pr_xy[i][1] = bezier_y.evaluate(0.5)  # Вибір середньої точки
        pr_xy[i][2] = bezier_z.evaluate(0.5)  # Вибір середньої точки
        pr_xy[i][3] = prlpd[i][3]  # Зберігаємо останній елемент без змін

    print('\nМастабування та зміщення')
    print(pr_xy)

    return pr_xy

# Блок головних викликів
if __name__ == '__main__':

    # Константи
```

```
    [...]
    st = 100  # розмір сторони фігури

    # Обчислення зсуву для центру вікна
    dx = (xw - st) / 2
    dy = (yw - st) / 2
    dz = dy

    # Головні виклики

    '''Монохром'''
    [...]
    # Масштабування та зміщення фігури
    pr_xy_3_scaled_center = scale_and_shift(pr_xy_3, st, dx, dy, dz)
```

Результат:

Бачимо матрицю від масштабованого у 100 разів та зміщеного до центру паралелепіпеда.

```
Мастабування та зміщення
[[250.         259.50941876 250.           0.        ]
 [446.3828382  263.1101886  250.           0.        ]
 [446.3828382  362.65701655 250.           0.        ]
 [250.         359.05624671 250.           0.        ]
 [212.13470108 278.18428521 250.           0.        ]
 [408.51753927 281.78505505 250.           0.        ]
 [408.51753927 381.331883   250.           0.        ]
 [212.13470108 377.73111316 250.           0.        ]]
```

Рисунок 4 – Матриця масштабування та зміщення

Візуалізація каркасу фігури

Також дещо видозмінимо функцію для візуалізації каркасу фігури.

Лістинг коду:

```
# Побудова паралелепіпеда
def prlpd_visualisation(win, pr_xy, color, color_mode):
    Ax = pr_xy[0, 0]
    Ay = pr_xy[0, 1]
    Bx = pr_xy[1, 0]
    By = pr_xy[1, 1]
    Ix = pr_xy[2, 0]
    Iy = pr_xy[2, 1]
    Mx = pr_xy[3, 0]
    My = pr_xy[3, 1]

    Dx = pr_xy[4, 0]
    Dy = pr_xy[4, 1]
    Cx = pr_xy[5, 0]
    Cy = pr_xy[5, 1]
```

```
    Fx = pr_xy[6, 0]
    Fy = pr_xy[6, 1]
    Ex = pr_xy[7, 0]
    Ey = pr_xy[7, 1]

    if not color_mode:
        obj_6 = Polygon(Point(Bx, By), Point(Cx, Cy), Point(Fx, Fy), Point(Ix, Iy))
        # obj_6.setFill(color)
        obj_6.draw(win)

    obj_5 = Polygon(Point(Mx, My), Point(Ax, Ay), Point(Dx, Dy), Point(Ex, Ey))
    if color_mode:
        obj_5.setFill(color[0])
    obj_5.draw(win)

    if not color_mode:
        obj_4 = Polygon(Point(Mx, My), Point(Ix, Iy), Point(Fx, Fy), Point(Ex, Ey))
        # obj_4.setFill(color)
        obj_4.draw(win)

    obj_3 = Polygon(Point(Ax, Ay), Point(Bx, By), Point(Cx, Cy), Point(Dx, Dy))
    if color_mode:
        obj_3.setFill(color[1])
    obj_3.draw(win)

    if not color_mode:
        obj_2 = Polygon(Point(Dx, Dy), Point(Cx, Cy), Point(Fx, Fy), Point(Ex, Ey))
        # obj_2.setFill(color)
        obj_2.draw(win)

    obj_1 = Polygon(Point(Ax, Ay), Point(Bx, By), Point(Ix, Iy), Point(Mx, My))
    if color_mode:
        obj_1.setFill(color[2])
    obj_1.draw(win)

    return [obj_1, obj_3, obj_5]

# Блок головних викликів
if __name__ == '__main__':

    # Константи
    [...]

    # Головні виклики

    '''Монохром'''
    [...]
    # Візуалізація МОНОХРОМУ фігури
    objs = prlpd_visualisation(win, pr_xy_3_scaled_center, color, color_mode=False)
    win.getMouse()
    win.close()
```

Результат:

Бачимо нерозфарбований каркас паралелепіпеда у диметричній проекції.

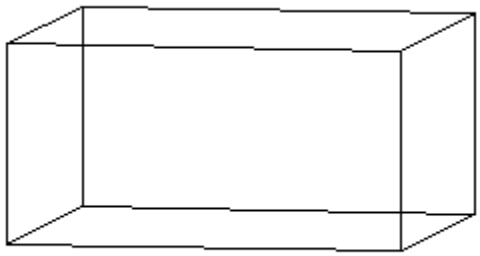


Рисунок 5 – Каркас фігури

Видалення невидимих граней

Як було описано на етапі проектування блок-схеми програми, для даної задачі **некоректно використовувати *алгоритм Варнока***, тому **використаємо алгоритм «*Плаваючого обрію*»**.

Визначення «**видимих граней**» відбувається за допомогою порівняння координат кожного кута грані з максимальними координатами з обраної зони видимості (x_max, y_max, z_max).

Лістинг коду:

```
# Видалення невидимих граней за алгоритмом "Плаваючого обрію"
def removing_faces(pr_dim, pr_xy, x_max, y_max, z_max, win):
    # Матриця без проекції
    aa_x = pr_dim[0, 0]
    aa_y = pr_dim[0, 1]
    aa_z = pr_dim[0, 2]

    bb_x = pr_dim[1, 0]
    bb_y = pr_dim[1, 1]
    bb_z = pr_dim[1, 2]

    ii_x = pr_dim[2, 0]
    ii_y = pr_dim[2, 1]
    ii_z = pr_dim[2, 2]

    mm_x = pr_dim[3, 0]
    mm_y = pr_dim[3, 1]
    mm_z = pr_dim[3, 2]

    #
    dd_x = pr_dim[4, 0]
    dd_y = pr_dim[4, 1]
    dd_z = pr_dim[4, 2]

    cc_x = pr_dim[5, 0]
```

```
    cc_y = pr_dim[5, 1]
    cc_z = pr_dim[5, 2]

    ff_x = pr_dim[6, 0]
    ff_y = pr_dim[6, 1]
    ff_z = pr_dim[6, 2]

    ee_x = pr_dim[7, 0]
    ee_y = pr_dim[7, 1]
    ee_z = pr_dim[7, 2]

    # Перевірка чи знаходяться грані в зоні видимості

    # Фронтальна та Верхня грані
    if (abs(aa_z-z_max) > abs(dd_z-z_max)) and (abs(bb_z-z_max) > abs(cc_z-z_max))  \
            and (abs(ii_z-z_max) > abs(ff_z-z_max)) and (abs(mm_z-z_max) > abs(ee_z-
z_max)):
        flag_f = 1
    else:
        flag_f = 2
    print('flag_f =', flag_f)
    # Ліва та Права грані
    if (abs(dd_x - x_max) > abs(cc_x - x_max)) and (abs(aa_x - x_max) > abs(bb_x -
x_max)) \
            and (abs(mm_x - x_max) > abs(ii_x - x_max)) and (abs(ee_x - x_max) > abs(ff_x
- x_max)):
        flag_r = 1
    else:
        flag_r = 2
    print('flag_r =', flag_r)
    # Задня та Нижня грані
    if (abs(aa_y - y_max) > abs(mm_y - y_max)) and (abs(bb_y - y_max) > abs(ii_y -
y_max)) \
            and (abs(cc_y - y_max) > abs(ff_y - y_max)) and (abs(dd_y - y_max) > abs(ee_y
- y_max)):
        flag_p = 1
    else:
        flag_p = 2
    print('flag_p =', flag_p)

    # Проекція
    a_x = pr_xy[0, 0]
    a_y = pr_xy[0, 1]

    b_x = pr_xy[1, 0]
    b_y = pr_xy[1, 1]

    i_x = pr_xy[2, 0]
    i_y = pr_xy[2, 1]

    m_x = pr_xy[3, 0]
    m_y = pr_xy[3, 1]

    #
    d_x = pr_xy[4, 0]
    d_y = pr_xy[4, 1]

    c_x = pr_xy[5, 0]
    c_y = pr_xy[5, 1]

    f_x = pr_xy[6, 0]
    f_y = pr_xy[6, 1]
```

```
    e_x = pr_xy[7, 0]
    e_y = pr_xy[7, 1]

    # Ліва грань
    obj = Polygon(Point(a_x, a_y), Point(m_x, m_y), Point(e_x, e_y), Point(d_x, d_y))
    if flag_r == 2:
        obj.setFill('blue')
        obj.draw(win)
    # Права грань
    obj = Polygon(Point(b_x, b_y), Point(i_x, i_y), Point(f_x, f_y), Point(c_x, c_y))
    if flag_r == 1:
        obj.setFill('magenta')
        obj.draw(win)
    # Верхня грань
    obj = Polygon(Point(a_x, a_y), Point(b_x, b_y), Point(c_x, c_y), Point(d_x, d_y))
    if flag_p == 1:
        obj.setFill('indigo')
        obj.draw(win)
    # Нижня грань
    obj = Polygon(Point(m_x, m_y), Point(i_x, i_y), Point(f_x, f_y), Point(e_x, e_y))
    if flag_p == 2:
        obj.setFill('gray')
        obj.draw(win)
    # Тильна грань
    obj = Polygon(Point(a_x, a_y), Point(b_x, b_y), Point(i_x, i_y), Point(m_x, m_y))
    if flag_f == 2:
        obj.setFill('cyan')
        obj.draw(win)
    # Фронтальна грань
    obj = Polygon(Point(d_x, d_y), Point(c_x, c_y), Point(f_x, f_y), Point(e_x, e_y))
    if flag_f == 1:
        obj.setFill('purple')
        obj.draw(win)

    return

# Блок головних викликів
if __name__ == '__main__':

    # Константи
    [...]

    # Головні виклики

    '''Монохром'''
    [...]

    '''Видалення граней'''
    [...]
    # Візуалізація КОЛОР фігури З ВИДАЛЕНИМИ ГРАНЯМИ
    removing_faces(prlpd_2, pr_xy_3_scaled_center, (xw * 2), (yw * 2), (yw * 2), win)
    win.getMouse()
    win.close()
```

Результат:

Бачимо лише «видимі» грані паралелепіпеда, розфарбовані у відповідні кольори.

Рисунок 6 – Видалення граней та розфарбування фігури

Поворот фігури

Для наочності операції видалення граней, додатково повернемо фігуру відносно осі OX на заданий кут. Скрипт повороту візьмемо з лабораторної роботи №1.

Лістинг коду

```
# Обертання навколо X
def rotate_x (figure, teta_x):
    teta_rx = (3 / 14 * teta_x) / 180

    f = np.array(
        [[1,    0,                  0,                  0],
         [0,    mt.cos(teta_rx),    mt.sin(teta_rx),    0],
         [0,    -mt.sin(teta_rx),   mt.cos(teta_rx),    0],
         [0,    0,                  0,                  1]])
    ft = f.T
    pr_xy = figure.dot(ft)

    print('\nОбертання навколо X')
    print(pr_xy)

    return pr_xy

# Блок головних викликів
if __name__ == '__main__':

    # Константи
    [...]

    teta_x = 160
    teta_y = 80

    # Головні виклики

    '''Монохром'''
    [...]

    '''Видалення граней'''
    [...]
```

```
    '''Видалення граней + поворот'''
    # Кути повороту
    teta_x = 180
    teta_y = -90

    [...]
    # Візуалізація повернутої КОЛОР фігури З ВИДАЛЕНИМИ ГРАНЯМИ
    removing_faces(prlpd_2, pr_xy_3_scaled_center, (xw * 2), (yw * 2), (yw * 2), win)
    win.getMouse()
    win.close()
```

Результат:

Бачимо, що невидимі грані паралелепіпеда дійсно видалені.

Рисунок 7 – Поворот фігури з виданеними гранями

2.2. Виділення контору на цифровому растровому зображенні

Алгоритм програми

Для виділення контуру на растровому зображенні використаємо **алгоритм векторизації**, реалізований бібліотекою *Pillow*. Алгоритм виконує наступні кроки:

1. Перетворює зображення у відтінки сірого.
2. Аналізує зміни яскравості між сусідніми пікселями.
3. Відляє контур.

Розробимо скрипкове рішення, яке буде виділяти контур зображення на основі **різниці яскравостей між сусідніми пікселями**. Додамо до програми можливість покращити виявлення контурів за допомогою **переведення зображення у ЧБ** з певним параметром розрізнення (так різниця яскравості стає більшою).

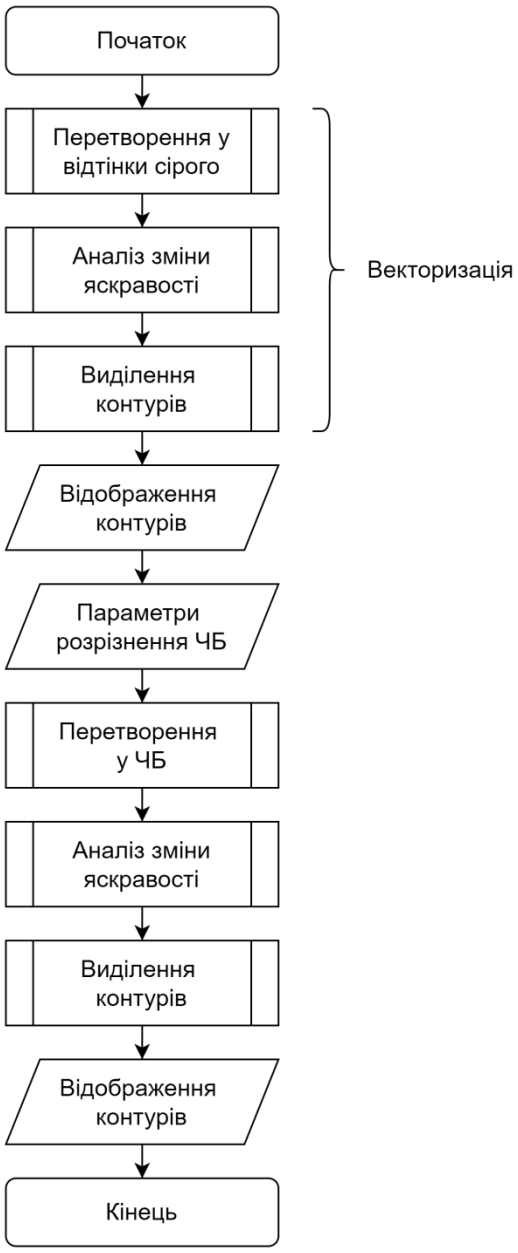


Рисунок 8 – Блок схема програми

Програмна реалізація

Векторизація

Для реалізації векторизації напишемо функцію виділення контурів на зображенні у відтінках сірого. Перетворення зображення у відтінки сірого реалізовано бібліотекою *Pillow*.

Лістинг коду:

```
# Виділення векторного контуру
def vector_circuit(image, title_1, title_2):
    figure()
    contour(image, origin='image')
    axis('equal')
    title(title_1)
    show()
    contour(image, levels=[170], colors='black', origin='image')
    axis('equal')
    title(title_2)
    show()

    return

if __name__ == '__main__':
    file_name = 'image.jpg'

    img = array(Image.open('image.jpg').convert('L'))
    image = Image.open("image.jpg")
    vector_circuit(img, 'Виділення контурів у відтінках сірого (до)', 'Контури (до)')
```

Результат:

Бачимо, що при переведенні у відтінки сірого зображення має багато шумів, через що контури виділяються погано.

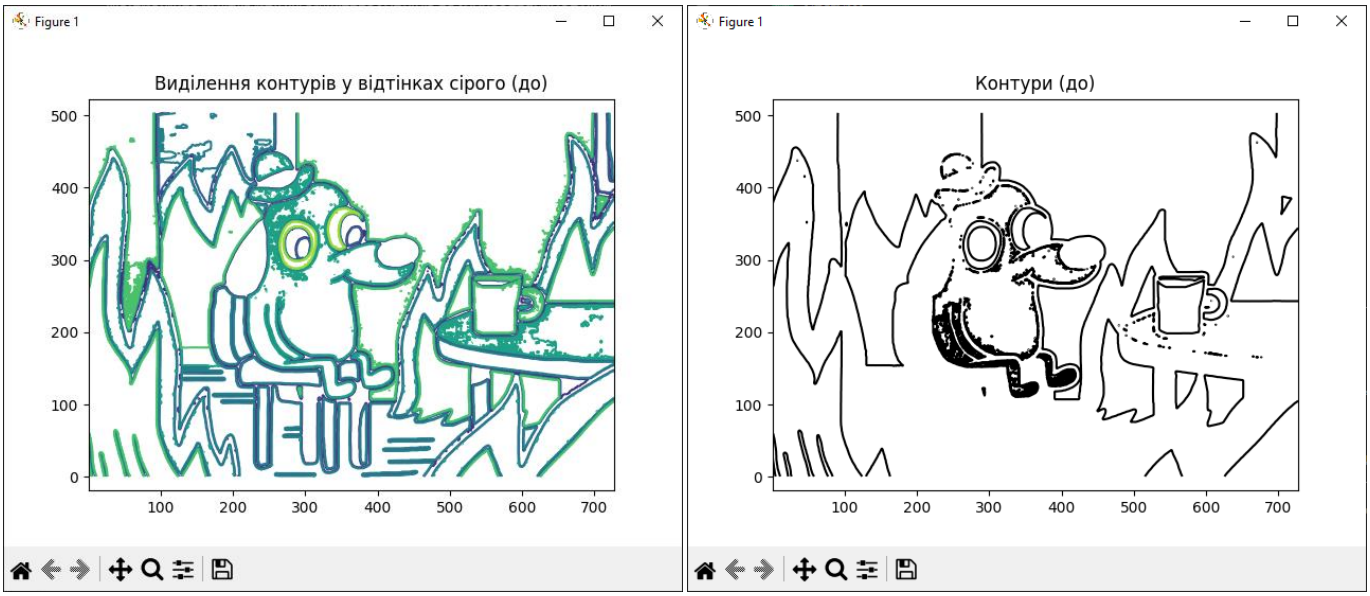


Рисунок 9 – Виділення контурів у відтінках сірого і векторизація

Перетворення в ЧБ

Для покращення отриманих результатів додамо функцію перетворення зображення в ЧБ.

Лістинг коду:

```
# Перетворення зображення з відтінків сірого в ЧБ
def mono(img, title):
    draw = ImageDraw.Draw(img)  # Інструмент для малювання
    width = img.size[0]         # Ширина картинки
    height = img.size[1]        # Висота картинки
    pix = img.load()            # Значення пікселів картинки

    print('Уведіть коефіцієнт розрізнення для ЧБ')
    factor = int(input('factor:'))
    for i in range(width):
        for j in range(height):
            a = pix[i, j][0]
            b = pix[i, j][1]
            c = pix[i, j][2]
            s = a + b + c
            # Рішення, до якого з 2 кольорів поточне значення кольору ближче
            if s > (((255 + factor) // 2) * 3):
                a, b, c = 255, 255, 255
            else:
                a, b, c = 0, 0, 0
            draw.point((i, j), (a, b, c))

    plt.imshow(img)
    plt.title(title)
    plt.show()
    img = img.convert('RGB')
    img.save("image_result.jpg", "JPEG")
    del draw

if __name__ == '__main__':
    [...]

    mono(image, 'Перетворення в ЧБ')
```

Результат:

Експериментально виявлено, що *найкращий коефіцієнт розрізнення ЧБ* $=-85$.

При такому значенні бачимо, що зображення стає значно контрастнішим і має краще виділяти контури.

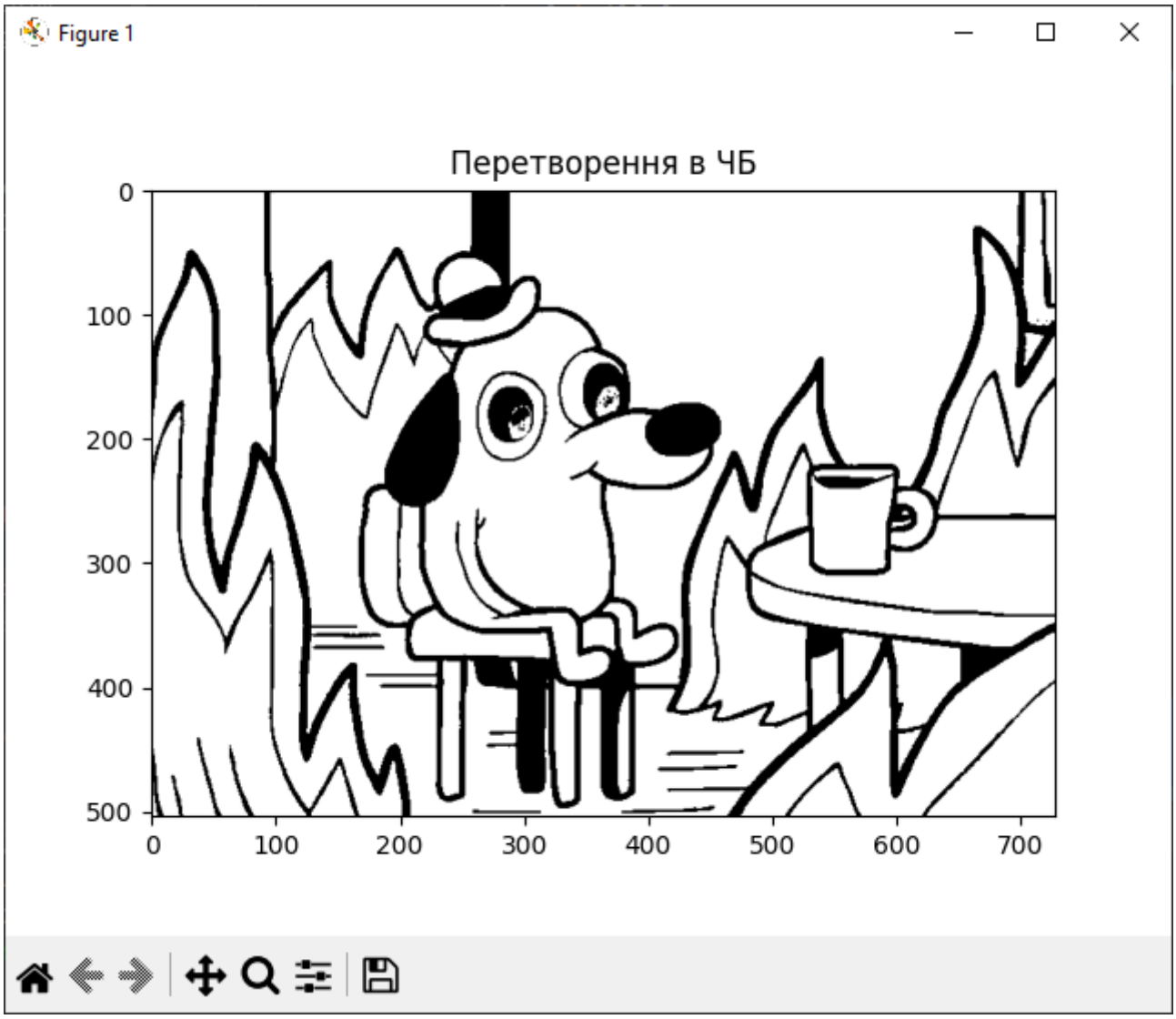


Рисунок 10 – Перетворення растрового зображення в ЧБ

Тепер використаємо алгоритм виділення контурів за допомогою векторизації.

Лістинг коду:

```
if __name__ == '__main__':
    [...]
    img = array(Image.open('image_result.jpg').convert('L'))
    vector_circuit(img, 'Виділення контурів у відтінках сірого (після)', 'Контури
(після)')
```

Результат:

У результаті бачимо, що так, як лінії на зображенні різної товщини, то програма виділила контур з обох сторін товстих ліній. Однак при цьому ми **маємо чудово виділені контури**.

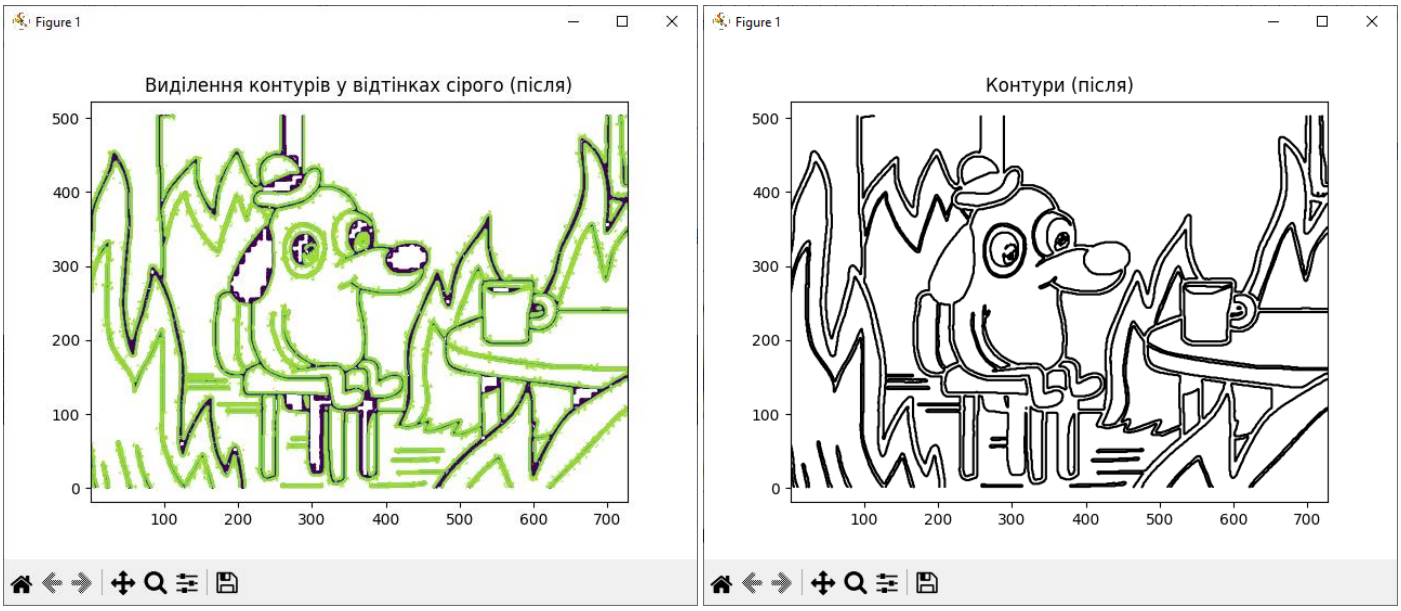


Рисунок 11 - Виділення контурів у відтінках сірого і векторизація

2.3. Аналіз отриманих результатів

Робота з 3D фігурами

Побудовано векторне зображення 3D паралелепіпеда – підтверджено рисунком.

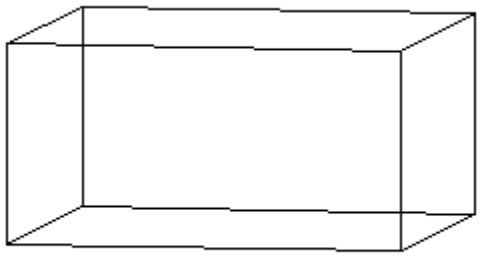


Рисунок 12 – Підтвердження побудови векторного зображення паралелепіпеда

Використано інтерполяцію кривими Безьє – підтверджено скриптом.

Використано алгоритм видалення невидимих граней «**Плаваючого обрію**» - підтверджено рисунками і скриптом.

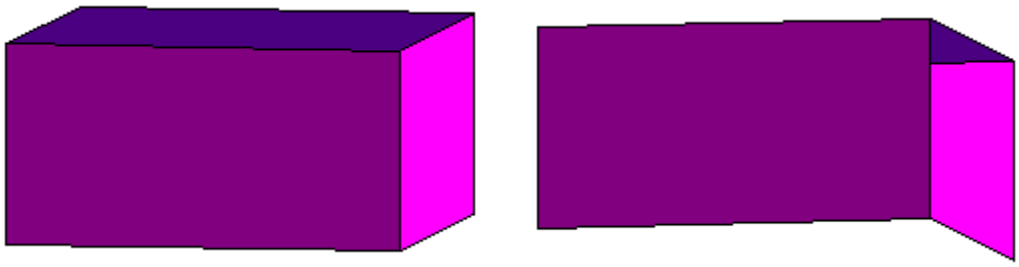


Рисунок 12 – Підтвердження використання алгоритму «Плаваючого обрію»

Виділення контурів на зображенні

Реалізовано векторизацію цифрового растрового зображення – підтверджено рисунками і скриптом.

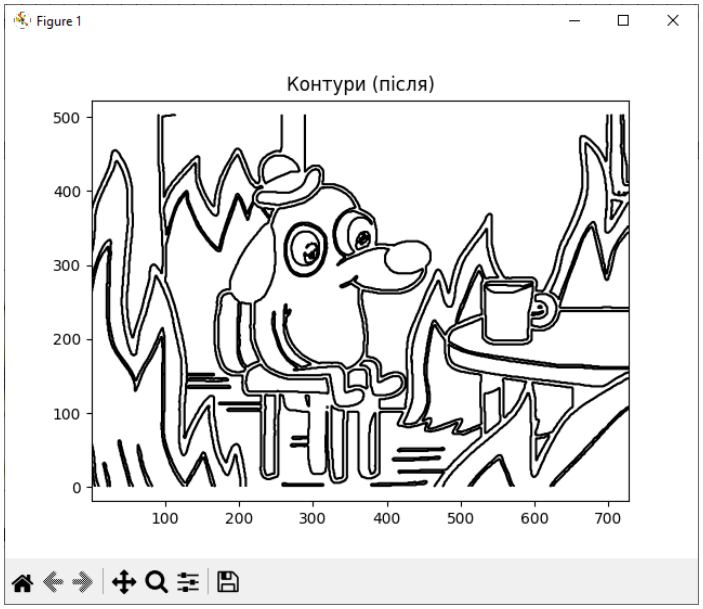


Рисунок 13 – Підтвердження векторизації растрового зображення

Реалізовано виділення контурів на растровому зображенні – підтверджено рисунками і скриптом.

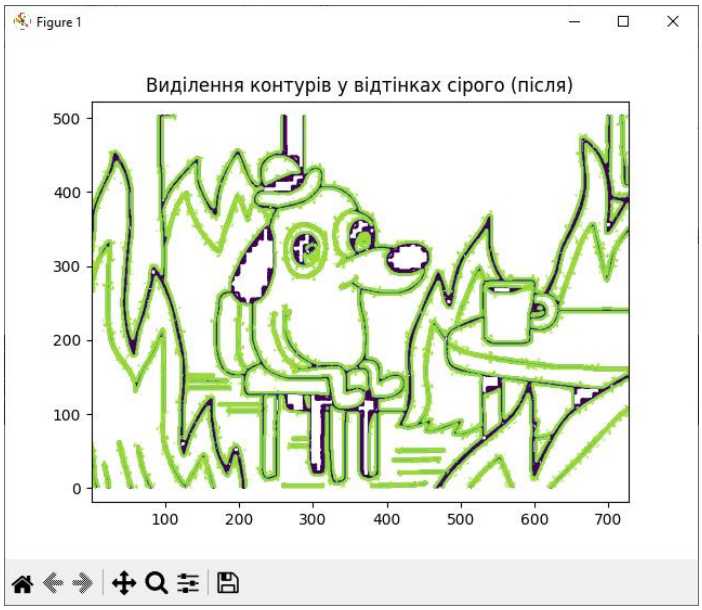


Рисунок 14 – Підтвердження виділення контурів растрового зображення

Застосовано фільтр ЧБ для покращення виділення контурів – підтверджено рисунками і скриптом.

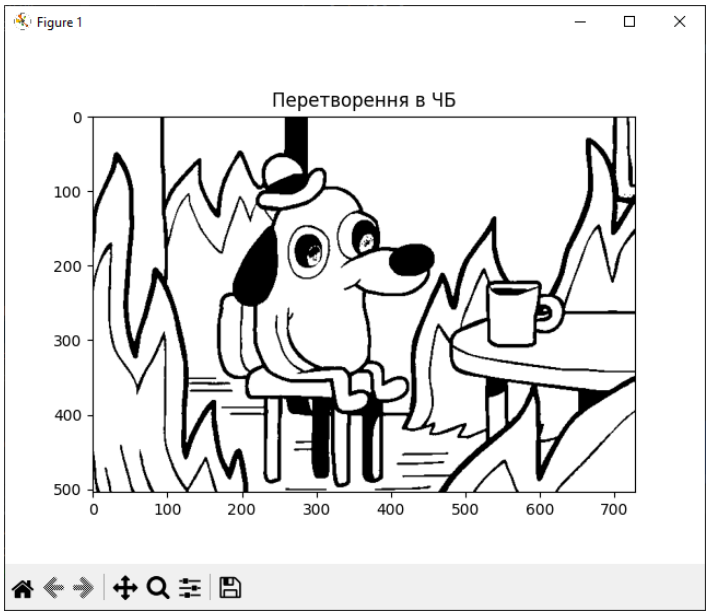


Рисунок 15 – Підтвердження застосування фільтру ЧБ

Висновок:

У результаті виконання лабораторної роботи отримано практичні та теоретичні навички роботи з геометричними 3D об’єктами та растровими зображеннями.

Реалізовано програмний скрипт №1, який виконує побудову векторного зображення 3D паралелепіпеда і видаляє невидимі грані.

Реалізовано програмний скрипт №2, який виконує векторизацію растрового зображення, виділяє і відображає контури і використовує ЧБ фільтр для покращення виявлення контурів.