Inżynieria obrazów

Dithering i rasteryzacja

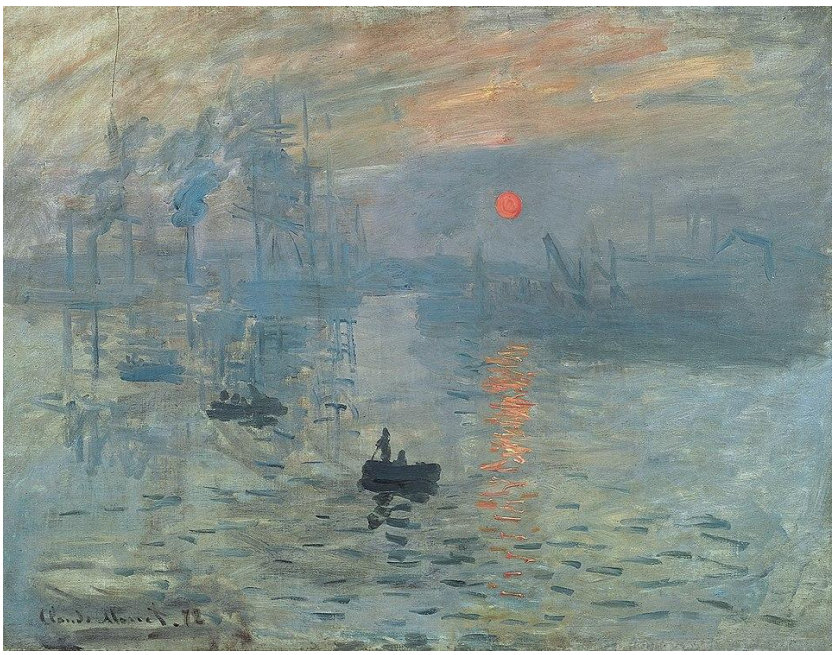
Marcin Lasak 272886

Spis treści

1. Wstęp teoretyczny

1.1 Dithering

Jest to efekt zastosowania szumu na obrazie, w celu zniwelowania błędu kwantyzacji. Przy obrazie kolorowym, polega na wykorzystaniu dostępnej palety kolorów, w celu stworzenia złudzenia istnienia koloru spoza palety. Wykorzystuje podobną technikę do tej, którą kultywowała w sztuce część impresjonistów – w szczególności pointylistów. Abstrahując od tematyki tego nurtu, a skupiając się na wykorzystywanych technikach, możemy zauważyć dwa zjawiska, które w odbiorze obrazu polegają na zawodności ludzkiej percepcji. Po pierwsze, synteza. Impresjoniści często korzystali z uproszczonych kształtów, w przeciwieństwie do wcześniejszego malarstwa akademickiego, które to dopiero przez nas jest interpretowane wzrokowo i nasz umysł wypełnia luki obrazem, który malarz chciał przekazać. Przykładem może być „Impresja. Wschód słońca" (rysunek 1).



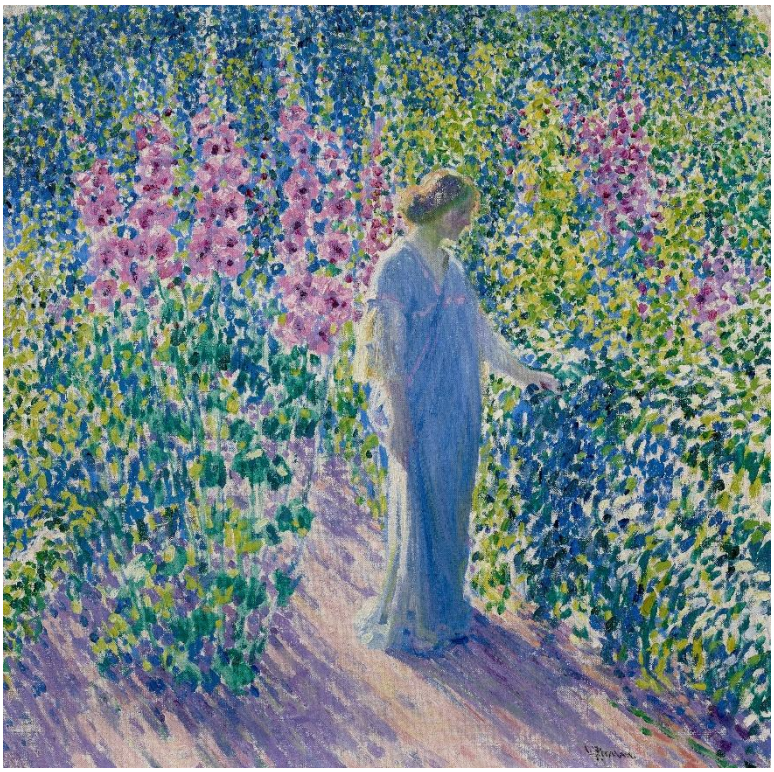
Rysunek 1

Na obrazie widzimy ograniczoną paletę barwną oraz proste kształty bez szczegółów. Mimo wszystko odbiorca widzi na obrazie port we mgle, wczesnym porankiem. Gdyby jednak przyjrzeć się szczegółom obrazu, są to bardzo proste kształty. W rzeczywistości jednak nasz umysł właśnie tak postrzega obrazy, korzystając z wielu uproszczeń odbieranych bodźców. Zauważmy np. jak nienaturalne wydają nam się obrazy, na których każdy element wydaje się wyostrzony.

W pierwszym przykładzie skupiliśmy się na syntezie kształtów. Kolejnym używanym przez impresjonistów narzędziem była synteza kolorów tj. korzystanie z ograniczonej palety barw lub/i korzystanie z niej w sposób, który nie odpowiada realnym kolorom danego obiektu. Przyjrzyjmy się rysunkowi nr 2. Przedstawia on wycinek obrazu Louisa Ritmana, przedstawiającego kobietę przy malwach. W przybliżeniu widzimy wiele plam kolorów, które nijak pasują nam do obrazu roślinności. Widzimy odcienie niebieskiego, do tego ciemny fiolet, a nawet plamy bordo. Gdy jednak spojrzymy na rysunek nr 3 przedstawiający cały obraz, kolory te zlewają się w naszych oczach tworząc wrażenie realnego krajobrazu. Nasz

umysł interpretuje niebieski jako cień, mimo że przy innym połączeniu kolorów, mógłby on równie dobrze zdawać się nam niebem lub karoserią Syreny.

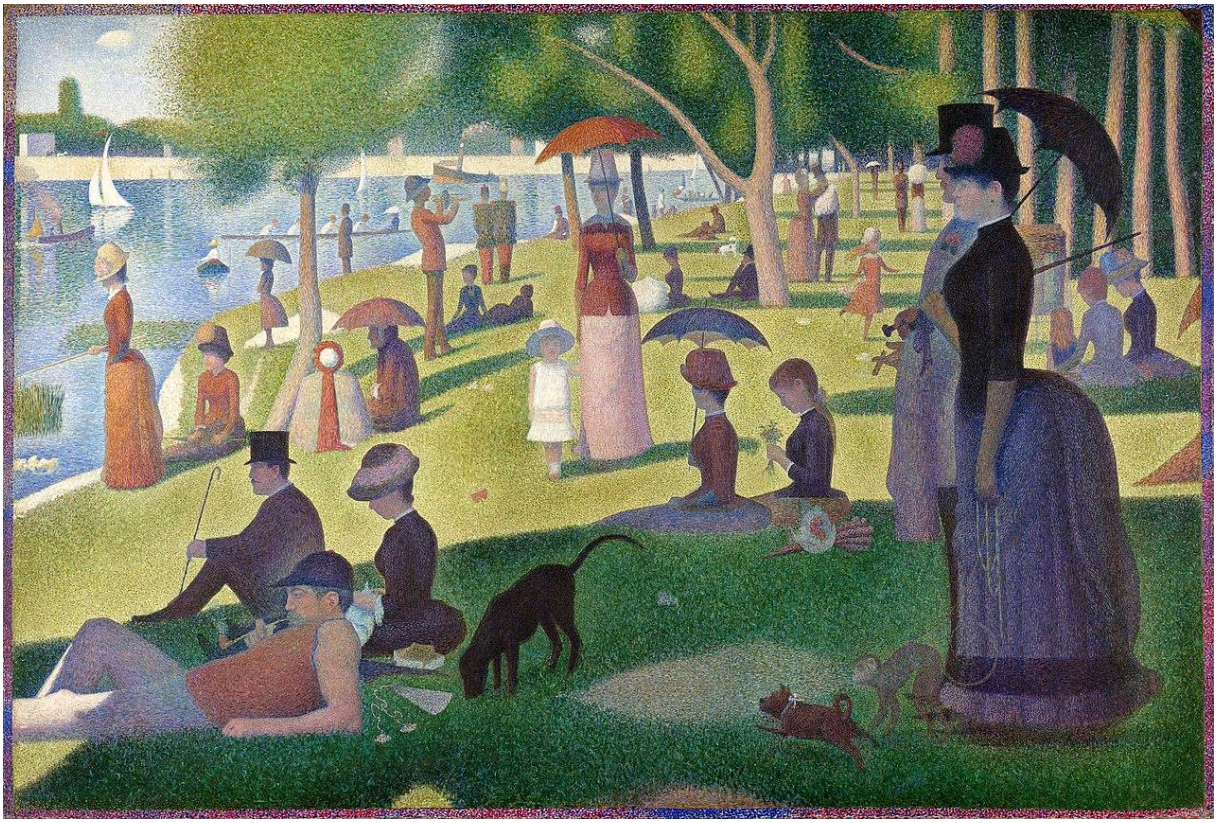
Rysunek 2



Rysunek 3

Finalnie jednak część malarzy maksymalnie posunęło się w tej technice, tworząc tym samym nurt zwany pointylizmem – obraz składa się z tysięcy małych kropek kolorów, które w oku człowieka łączą się w pełnoprawny obraz (rysunek 4). Dithering jest właśnie techniką, która wykorzystuje dokładnie ten sam schemat. Zmniejszamy paletę kolorów i wprowadzamy do

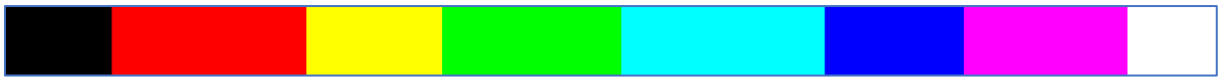
obrazu szum w taki sposób, aby wydawało nam się, że na obrazie są kolory, których realnie tam nie ma.



Rysunek 4

Jak mówi nam nazwa tej techniki (dithering - rozsiewanie), na obrazie rozsiewamy piksele dwóch barw składowych, które w naszym oku mają łączyć się w jeden kolor. Dithering jest stosowany przy zapobieganiu tzw. pasmowaniu przy redukcji palety kolorów. Zjawisko to obrazują rysunki 5 i 6 (5 – oryginał, 6 – zredukowana paleta; pojawia się pasmowanie; niebieskie obramowania dodano w sprawozdaniu).

Rysunek 5



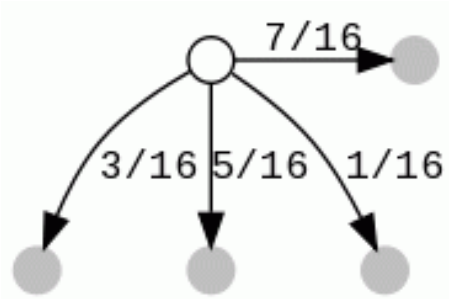
Rysunek 6

Jednym z najpopularniejszych algorytmów ditheringu jest algorytm Floyda-Steinberga. Osiąga on dithering za pomocą dyfuzji błędów ,tzn. przesuwa (dodaje) resztkowy błąd kwantyzacji piksela na sąsiednie piksele, aby zająć się nim później. Rozprowadza go zgodnie z macierzą daną rysunkiem nr 7.

$$\begin{bmatrix} & & * & \frac{7}{16} & \cdots \\ \cdots & \frac{3}{16} & \frac{5}{16} & \frac{1}{16} & \cdots \end{bmatrix}$$

Rysunek 7

Macierz tę możemy przedstawić także za pomocą grafu (rysunek 8). Dla każdego przetwarzanego piksela i decyzji o kwantyzacji (zmniejszenie palety kolorów), obliczany jest błąd, który następnie propagowany jest na sąsiednie komórki wg grafu (macierzy): 7/16 do komórki kolejnej, 5/16 poniżej, 1/16 po ukosie do przodu i 3/16 po ukosie do tyłu.



Rysunek 8

1.2 Rasteryzacja

Jest to działanie polegające na jak najwierniejszym odwzorowaniu figury płaskiej za pomocą grafiki rastrowej. Z oczywistych powodów, nie jesteśmy w stanie realnie odtworzyć na ekranie rastrowym niektórych figur. Wyświetlacze większości dzisiejszych urządzeń są mapami pikseli, a co za tym idzie, nie jesteśmy w stanie wiernie odwzorować na nich większości prostych (poza $y = a$, $y = x + b$, $x = a$) oraz krzywych. Z tego powodu zawsze posługujemy się przybliżeniami. Problem ten prowadzi do całej gałęzi rozwoju algorytmów, mianowicie algorytmów rasteryzacji, które to mają na celu jak najwierniej odwzorować obiekt ciągły, w dziedzinie dyskretnej.

Jednym z podstawowych algorytmów rasteryzacji figur płaskich jest algorytm Bresenhama. W tym algorytmie, kolejny zaznaczony piksel jest sąsiadem poprzedniego, w położeniu góra-dół, prawo-lewo, ukosy. Tzn. kolejny piksel jest oddalony o jeden piksel w metryce Czebyszewa. Wybór ruchu odbywa się z uwzględnieniem tzw. zmiennej decyzyjnej[1].

1.3 Antyaliasing przestrzenny

Jest to technika minimalizująca zniekształcenia podczas przedstawiania obrazu o wysokiej rozdzielczości w niższej rozdzielczości. Krawędzie są wizualnie wygładzone (usunięcie „schodków”), co skutkuje bardziej estetycznym obrazem.

Jedną z metod antyaliasingu jest SSAA (**S**uper**S**ampling **A**nti-**A**liasing). Polega on na wyrenderowaniu obrazu w dużo wyższej rozdzielczości, aby później go wyskalować w

[1] https://pl.wikipedia.org/wiki/Algorytm_Bresenhama

rozdzielczości ustalonej przez użytkownika. Wskutek tych operacji metoda ta potrzebuje dużo więcej zasobów, względem innych rodzajów antyaliasingu np. TAA, czy FXAA[2].

Określenie Supersampling („superpróbkowanie”) odnosi się do metody próbkowania wykorzystywanej w tej technice. Próbki kolorów nie są pobierane w tylko jednym punkcie wewnątrz piksela, lecz w kilku różnych, więc na jeden piksel przypada kilka próbek koloru.

[2] https://ithardware.pl/poradniki/antyaliasing_dlss_ray_tracing_poznaj_ustawienia_graficzne-17021-3.html

2. Zadania

2.1 Dithering w skali szarości

W pierwszym zadaniu należało zaimplementować dithering algorytmem Floyda-Steinberga w skali szarości, wykorzystując podany kod (rysunek 9).

```
for each y from top to bottom do
    for each x from left to right do
        oldpixel := pixel[y][x]
        newpixel := find_closest_palette_color(oldpixel)
        pixel[y][x] := newpixel
        quant_error := oldpixel - newpixel
        pixel[y    ][x + 1] := pixel[y    ][x + 1] + quant_error * 7 / 16
        pixel[y + 1][x - 1] := pixel[y + 1][x - 1] + quant_error * 3 / 16
        pixel[y + 1][x    ] := pixel[y + 1][x    ] + quant_error * 5 / 16
        pixel[y + 1][x + 1] := pixel[y + 1][x + 1] + quant_error * 1 / 16
```

Rysunek 9

Zadanie wykonano w języku Python z wykorzystaniem biblioteki numpy, do obliczeń oraz Pillow, do wyświetlania obrazów. Wykorzystano również funkcję pomocniczą show_image() używaną na poprzednich laboratoriach.

W pliku lab4.py zaimplementowano funkcje find_closest_palette_color() (zadaną wprost w instrukcji) oraz floyd_steinberg_dithering() (rysunek 10). Pierwsza funkcja służy do kwantyzacji koloru piksela do czerni lub bieli. W drugiej funkcji wykonujemy właściwy dithering. Na podstawie obrazu wejściowego tworzymy tablicę numpy. Następnie korzystamy z kodu z rysunku nr 9. Dla każdego piksela obrazu ustalamy kolor – czerń lub biel, obliczamy błąd między oryginalnym kolorem, a nowym (o ile „zaokrągliliśmy" kolor), a następnie propagujemy ten błąd na sąsiednie piksele wg proporcji danych algorytmem Floyda-Steinberga. Na koniec upewniamy się, że współrzędne koloru są w zakresie 0-255 i wyświetlamy obraz po zmianie typu na całkowitoliczbowy zapisywany na ośmiu bitach.

W celu lepszego zobrazowania wpływu ditheringu przy redukcji palety, zaimplementowano także funkcję reduce_palette() (rysunek 11), która redukuje paletę kolorów bez ditheringu.

lab4.py

```python
def find_closest_palette_color(value):
    return round(value / 255) * 255

def floyd_steinberg_dithering(image):

    pixels = np.array(image, dtype=float)

    height, width = pixels.shape

    #kod ze slajdu 5
    for y in range(height):
        for x in range(width):
            old_pixel = pixels[y, x]
            new_pixel = find_closest_palette_color(old_pixel)
            pixels[y, x] = new_pixel
            quant_error = old_pixel - new_pixel

            if x + 1 < width:
                pixels[y, x + 1] += quant_error * 7 / 16
            if y + 1 < height and x > 0:
                pixels[y + 1, x - 1] += quant_error * 3 / 16
            if y + 1 < height:
                pixels[y + 1, x] += quant_error * 5 / 16
            if y + 1 < height and x + 1 < width:
                pixels[y + 1, x + 1] += quant_error * 1 / 16

    #

    pixels = np.clip(pixels, 0, 255)
    return Image.fromarray(pixels.astype(np.uint8))
```

Rysunek 10

lab4.py

```python
def reduce_palette(image):

    pixels = np.array(image, dtype=np.uint8)

    reduced_pixels = (np.round(pixels / 255) * 255).astype(np.uint8)

    return Image.fromarray(reduced_pixels)
```

Rysunek 11

W celu przetestowania działania algorytmu utworzono funkcje process_1 oraz process_1a. Pierwsza z nich redukuje paletę kolorów z ditheringiem, natomiast druga – bez.

Wybranie opcji nr 1 w programie main.exe (powstały na podstawie main.py) spowoduje wykonanie redukcji palety kolorów kolejno bez i z ditheringiem na pliku munch.png (rysunek 12), zawierającym obraz „Krzyk” E. Muncha. Wyniki działania programu dane są rysunkami 13 i 14.

Obrazy wyjściowe to munch.pngoutput_palette.png oraz munch.pngoutput_palette.png.

Rysunek 12 Oryginał

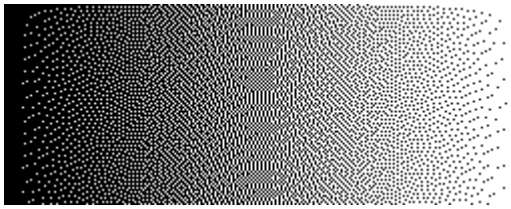
Rysunek 13 Redukcja palety bez ditheringu

Rysunek 14 Redukcja palety z ditheringiem

Widzimy wyraźny wpływ ditheringu na odbiór przez nas przetwarzanego obrazu. Wpływ ten uwypukla się jeszcze bardzie w przypadku spektrum przejścia czerń-biel (rysunki 15, 16, 17; niebieskie obramowanie dodano w sprawozdaniu).

Rysunek 15 Oryginał

Rysunek 16 Redukcja palety bez ditheringu



Rysunek 17 Redukcja palety z ditheringiem

Na rysunku nr 16 widzimy opisywane we wstępie teoretycznym pasmowanie.

Warto zaznaczyć, że w funkcjach process_1() oraz process_1a() obrazy wejściowe konwertowane są automatycznie do skali szarości, w przypadku gdy są kolorowe (jak w przypadku „Krzyku" Muncha). Jest to konieczne dla aktualnej struktury algorytmu, która wykorzystuje jedną tablicę dwuwymiarową (w przeciwieństwie do tablicy trójek współrzędnych w przypadku obrazu kolorowego).

2.2 Dithering w kolorze

Kolejne zadanie polegało na rozszerzeniu implementacji algorytmu Floyda-Steinberga z zadania pierwszego o obsługę koloru. Wykorzystywane są te same biblioteki jak w poprzednim zadaniu.

W tym celu zmodyfikowano funkcje z zadania pierwszego (rysunek 18).

lab4.py

```python
def find_closest_palette_color_k(value,k):
    return round((k - 1) * value / 255) * 255 / (k - 1)

def floyd_steinberg_dithering_color(image, k=2):

    pixels = np.array(image, dtype=float)
    height, width, channels = pixels.shape

    for y in range(height):
        for x in range(width):
            old_pixel = pixels[y, x].copy()
            new_pixel = np.zeros(3)
            for c in range(channels):
                new_pixel[c] = find_closest_palette_color_k(old_pixel[c], k)
            pixels[y, x] = new_pixel
            quant_error = old_pixel - new_pixel

            if x + 1 < width:
                pixels[y, x + 1] += quant_error * (7 / 16)
            if y + 1 < height and x > 0:
                pixels[y + 1, x - 1] += quant_error * (3 / 16)
            if y + 1 < height:
                pixels[y + 1, x] += quant_error * (5 / 16)
            if y + 1 < height and x + 1 < width:
                pixels[y + 1, x + 1] += quant_error * (1 / 16)

    pixels = np.clip(pixels, 0, 255)
    return Image.fromarray(pixels.astype(np.uint8))
```

Rysunek 18

Funkcja find_closest_palette_color_k() pochodzi wprost z instrukcji. Różni się od pierwowzoru tym, że paleta danej składowej może się składać z k kolorów, a nie tylko dwóch.

W funkcji floyd-steinberg_color() wprowadzono kwantyzację i dyfuzję błędu dla każdej składowej. Tak więc operujemy tutaj na tablicy trójek, w przeciwieństwie do zwykłej tablicy dwuwymiarowej z zadania 1.

Dodatkowo zaimplementowano funkcję redukującą paletę barw bez ditheringu, tak jak w poprzednim punkcie, z tą różnicą, że dla obrazu kolorowego. Funkcja ta dana jest rysunkiem nr 19.

lab4.py

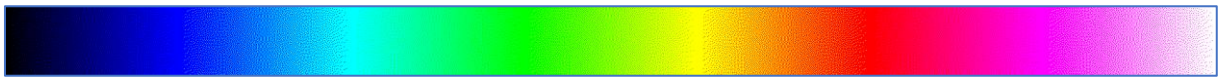
```
def reduce_colors(image, k=2):

    pixels = np.array(image, dtype=float)
    height, width, channels = pixels.shape

    for y in range(height):
        for x in range(width):
            for c in range(channels):
                pixels[y, x, c] = find_closest_palette_color_k(pixels[y, x, c], k)

    pixels = np.clip(pixels, 0, 255)
    return Image.fromarray(pixels.astype(np.uint8))
```

Rysunek 19

Finalnie zaimplementowano funkcję process_2(), która wywołuje kolejno funkcje reduce_colors() oraz floyd-steinberg_color(). Zostaje ona wywołana po wybraniu opcji nr 2 w programie main.exe. Wyniki działania programu dane są rysunkami nr 20 i 21. Również rysunki 5 i 6 ze wstępu teoretycznego powstały z wykorzystaniem tej funkcji. Rysunek 22 prezentuje spektrum z rysunku piątego ze zredukowaną paletą barw, ale z zastosowaniem ditheringu (niebieskie obramowanie dodano w sprawozdaniu).

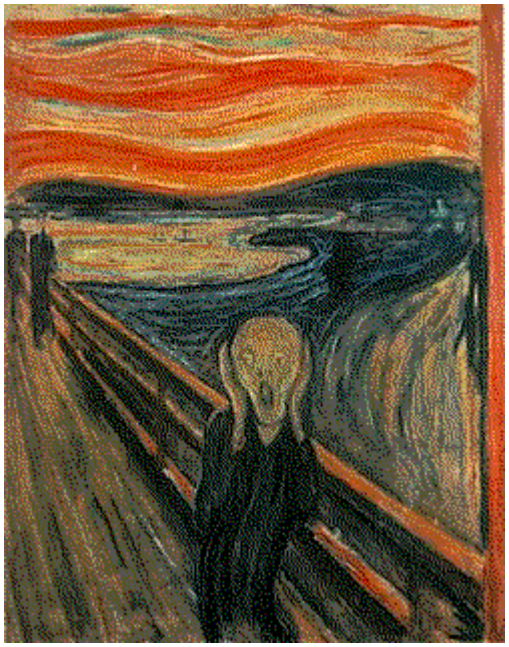
Rysunek 20 Redukcja palety bez ditheringu

Rysunek 21 Redukcja palety z ditheringiem



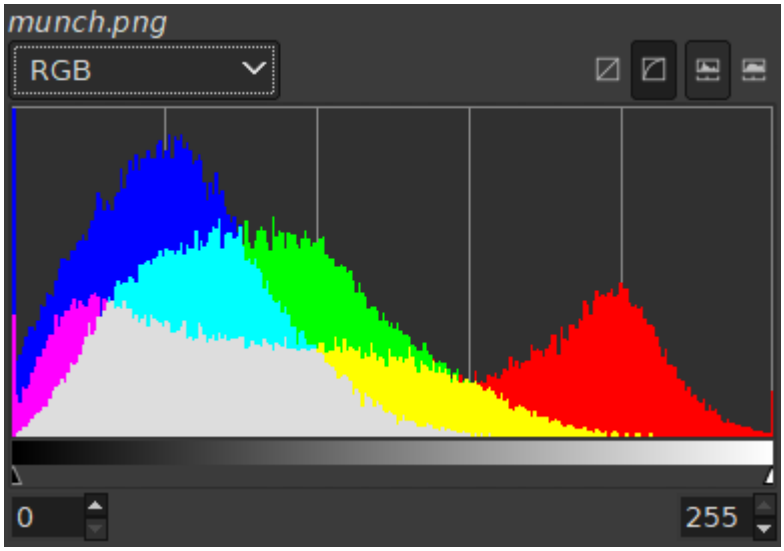
Rysunek 22

W naszym programie współczynnik k ustawiony jest domyślnie na 2. Jednak gdy poszerzymy paletę kolorów tylko o 6 kolorów (k=4), otrzymamy obraz wyglądający prawie identycznie jak oryginał (rysunek 23).

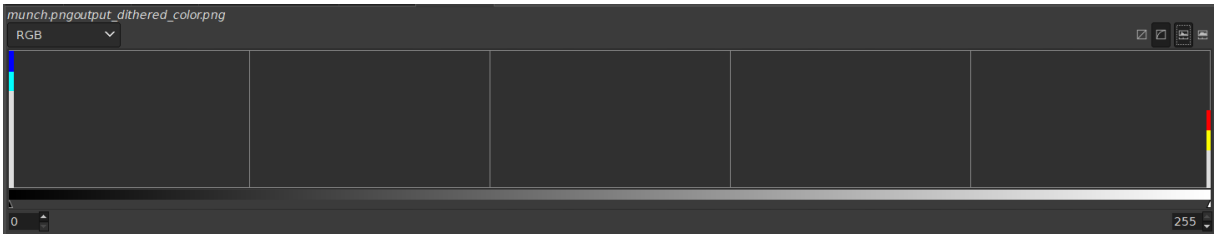


Rysunek 23

Rysunki nr 24 i 25 prezentują histogramy odpowiednio rysunków 5 i 21. Histogramy pochodzą z programu GIMP.



Rysunek 24



Rysunek 25

Widzimy, że każdy z kanałów został sprowadzony do dwóch wartości.

2.3 Rysowanie linii i trójkąta

W tym zadaniu należało zaimplementować rysowanie jednokolorowych linii oraz trójkąta z wykorzystaniem algorytmu Bresenhama. Wykorzystano te same biblioteki co w poprzednich zadaniach.

Rozpoczęto od zaimplementowania funkcji draw_point(), edge_function() oraz is_point_in_triangle() (rysunek 26).

Pierwsza z nich pochodzi ze strony datko.pl (zasugerowano jej użycie w isntrukcji). Służy ona do narysowania punktu o zadanym kolorze w zadanych współrzędnych. Funkcja edge_function() wykorzystuje iloczyn wektorowy potrzebny do sprawdzenia, po której stronie krawędzi leży punkt. Iloczyn wektorowy jest zawsze prostopadły do płaszczyzny, wyznaczanej przez wektory na których ów iloczyn wykonujemy. Zależnie od położenia tych dwóch wektorów względem siebie, znak współrzędnej z iloczynu będzie dodatni, bądź ujemny (skierowanie wektora w górę lub w dół). Zatem dla odcinka AB i punktu P, iloczyn $\overrightarrow{AP} \times \overrightarrow{AB} = (x_{AP}, y_{AP}, 0) \times (x_{AB}, y_{AB}, 0) = (0,0,z)$ pozwala nam określić położenie punktu P względem odcinka AB. Odpowiada za to współrzędna z (jej znak). Samą współrzędną z możemy prosto obliczyć ze wzoru $z = x_{AP}y_{AB} - x_{AB}y_{AP}$. Dzięki regule prawej dłoni możemy

określić wtedy, po której stronie leży punkt P. Następnie funkcja is_point_in_triangle() sprawdza, po której stronie każdej z krawędzi trójkąta znajduje się badany punkt. Badana jest wartość składowej z. Dla każdej krawędzi. Jeśli wszystkie są dodatnie, lub wszystkie są ujemne, punkt leży wewnątrz trójkąta lub dokładnie na jego brzegu. Natomiast jeśli znaki wszystkich składowych nie są zgodne, to punkt leży poza trójkątem.

lab4.py

```python
def draw_point(image, x, y, color=(255, 255, 255)):
    image[image.shape[0] - 1 - y, x, :] = color


def edge_function(ax, ay, bx, by, px, py):
    return (px - ax) * (by - ay) - (py - ay) * (bx - ax)


def is_point_in_triangle(p, a, b, c):

    px, py = p
    ax, ay = a
    bx, by = b
    cx, cy = c


    w1 = edge_function(ax, ay, bx, by, px, py)
    w2 = edge_function(bx, by, cx, cy, px, py)
    w3 = edge_function(cx, cy, ax, ay, px, py)

    has_neg = (w1 < 0) or (w2 < 0) or (w3 < 0)
    has_pos = (w1 > 0) or (w2 > 0) or (w3 > 0)


    return not (has_neg and has_pos)
```

Rysunek 26

Funkcje te użyto do zaimplementowania rysowania linii i trójkąta. Wpierw, dla późniejszego porównania, zaimplementowano „naiwne” rysowanie linii, tj. z wykorzystaniem zwykłego zaokrąglania (rysunek 27).

lab4.py

```python
def draw_line_rounded(image, x0, y0, x1, y1, color):
    if x0!=x1:
        a=(y1-y0)/(x1-x0)
        b=y1-a*x1
        dx=abs(x1-x0)

        for x in range(dx):
            y=a*(x+x0)+b
            draw_point(image, int(x+x0), int(round(y)), color)

    else:
        dy=abs(y1-y0)
        if y1>y0:
            for y in range(dy):
                draw_point(image, x0, y+y0, color)
        else:
            for y in range(dy):
                draw_point(image, x0, y+y1, color)
```

Rysunek 27

Wykorzystuje ona równanie prostej, obliczane na podstawie podanych punktów, a kolejne piksele kolorowane są na wysokości będącej zaokrągleniem wartości tej funkcji dla punktu x. W przypadku linii danych równaniem $x = a$ rysowany jest odcinek prostopadły do osi OX.

Następnie zaimplementowano rysowanie linii z wykorzystaniem algorytmu Bresenhama w funkcji draw_line() (rysunek 28). Wykorzystuje ona wspominaną we wstępie teoretycznym zmienną decyzyjną, będącą różnicą między teoretyczną linią, a linią po rasteryzacji. Po ustaleniu przyrostu wartości współrzędnych x i y oraz określeniu kierunku ruchu. Następnie w pętli korzystamy z podwojonej zmiennej decyzyjnej, do której porównujemy aktualne odchylenie współrzędnych od teoretycznej linii.

lab4.py

```python
def draw_line(image, x0, y0, x1, y1, color):
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    D = dx - dy

    while True:
        draw_point(image, x0, y0, color)

        if x0 == x1 and y0 == y1:
            break

        e2 = 2 * D
        if e2 > -dy:
            D -= dy
            x0 += sx
        if e2 < dx:
            D += dx
            y0 += sy
```

Rysunek 28

Funkcja draw_filled_triangle() (rysunek 29) realizuje rysowanie wypełnionego kolorem trójkąta.

lab4.py

```python
def draw_filled_triangle(image, a, b, c, color):
    xmin = min(a[0], b[0], c[0])
    xmax = max(a[0], b[0], c[0])
    ymin = min(a[1], b[1], c[1])
    ymax = max(a[1], b[1], c[1])

    for y in range(ymin, ymax + 1):
        for x in range(xmin, xmax + 1):
            if is_point_in_triangle((x, y), a, b, c):
                draw_point(image, x, y, color)
```

Rysunek 29

Wpierw wyznaczamy współrzędne wierzchołków prostokąta, w którym znajduje się trójkąt o zadanych wierzchołkach. Następnie iterujemy po wszystkich punktach tego prostokąta, sprawdzając, czy znajdują się one wewnątrz zadanego trójkąta przy pomocy funkcji is_point_in_triangle(). Jeśli tak, punkt jest kolorowany.

Rysowanie odbywa się przy pomocy dwóch funkcji: process_3() oraz process_3a() (rysunek 30).

lab4.py

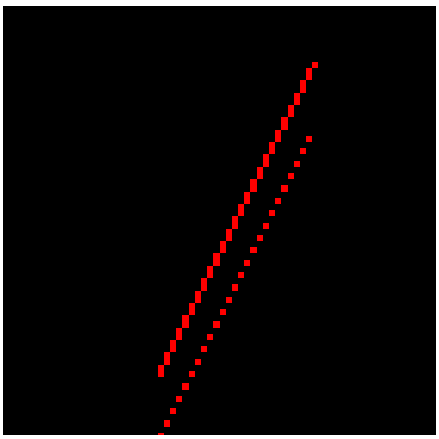
```
def process_3():

    width, height = 200, 200

    image = np.zeros((height, width, 3), dtype=np.uint8)

    draw_line(image, 5, 5, 50, 50, (255, 0, 0))
    draw_line(image, 50, 50, 50, 100, (0, 255, 0))

    a = (60, 30)
    b = (160, 80)
    c = (100, 160)
    draw_filled_triangle(image, a, b, c, (0, 0, 255))

    Image.fromarray(image).save("lines_and_triangle.png")
    Image.fromarray(image).show()

def process_3a():
    width, height = 70, 70
    image = np.zeros((height, width, 3), dtype=np.uint8)

    draw_line_rounded(image, 25, 0, 50, 50, (255, 0, 0))
    draw_line(image, 25, 10, 50, 60, (255, 0, 0))

    Image.fromarray(image).save("lines.png")
    Image.fromarray(image).show()
```

Rysunek 30

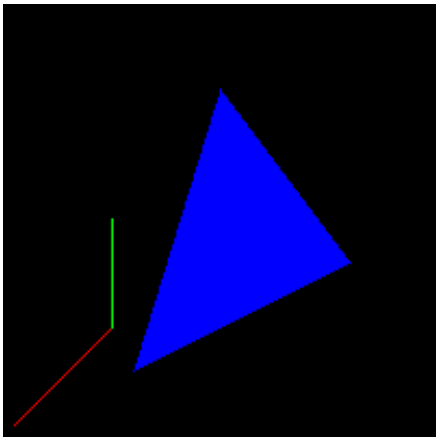
Funkcja process_3() tworzy tablicę numpy wypełnioną trójkami zer (czerń), która będzie tłem na którym rysowana jest linia oraz trójkąt. Następnie rysuje dwie linie (czerwoną i zieloną) oraz wypełniony niebieskim kolorem trójkąt.

Funkcja process_3a() działa podobnie, lecz rysuje dwie linie o tym samym nachyleniu wykorzystując rysowanie „naiwne” oraz z wykorzystanie algorytmu Bresenhama. Zmniejszono także rozmiar tablicy, aby lepiej zauważyć, jak dokładnie rysowane są piksele.

Wybranie opcji nr 3 w programie main.exe skutkuje wywołaniem kolejno funkcji process_3a oraz process_3(). Wyniki działania programu dane są rysunkami 31 i 32.



Rysunek 31



Rysunek 32

Na rysunku nr 31 widzimy ewidentnie różnicę między rysowaniem „naiwnym”, a z wykorzystaniem algorytmu Bresenhama. Dolna linia (rysowanie „naiwne”) nie jest ciągła i przy małej rozdzielczości widzimy że jest zbiorem dyskretnym. Górna linia natomiast (algorytm Bresenhama), jest ciągła.

2.4 Interpolacja koloru

Kolejnym zadaniem było wprowadzenie interpolacji liniowej koloru dla rysowania linii i trójkąta z poprzedniego zadania. Korzystano z bibliotek jak w poprzednich zadaniach. Wpierw zaimplementowano funkcję interpolacji koloru, realizującą wzór podany w instrukcji (rysunek 33). Oblicza ona interpolowaną wartość koloru dla każdej składowej i zwraca trójkę współrzędnych RGB.

lab4.py

```
def interpolate_color(color_a, color_b, t):
    return tuple([
        int(color_a[i] + t * (color_b[i] - color_a[i]))
        for i in range(3)
    ])
```

Rysunek 33

Funkcja draw_line_with_color() (rysunek 34) jest modyfikacją funkcji draw_line() z poprzedniego zadania. Dodano do niej interpolację kolorów za pomocą funkcji interpolate_color().

lab4.py

```python
def draw_line_with_color(image, x0, y0, color0, x1, y1, color1):
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    err = dx - dy

    steps = max(dx, dy)
    step = 0

    while True:
        if steps != 0:
            t = step / steps
        else:
            t = 0
        color = interpolate_color(color0, color1, t)
        draw_point(image, x0, y0, color)

        if x0 == x1 and y0 == y1:
            break

        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy
        step += 1
```

Rysunek 34

Rysowanie wypełnionego trójkąta z interpolacją kolorów było zadaniem bardziej skomplikowanym. Wpierw zaimplementowano ważenie pól trójkątów. Funkcja barycentric_weights() (rysunek 35) wyznacza współczynniki barycentryczne[3] zadanego punktu tj. współczynniki mówiące nam jaki wpływ będzie miał dany wierzchołek na ten punkt.

lab4.py

```python
def barycentric_weights(p, a, b, c):
    det = (b[1] - c[1])*(a[0] - c[0]) + (c[0] - b[0])*(a[1] - c[1])
    if det == 0:
        return 0, 0, 0
    w1 = ((b[1] - c[1])*(p[0] - c[0]) + (c[0] - b[0])*(p[1] - c[1])) / det
    w2 = ((c[1] - a[1])*(p[0] - c[0]) + (a[0] - c[0])*(p[1] - c[1])) / det
    w3 = 1 - w1 - w2
    return w1, w2, w3
```

Rysunek 35

Określmy trójkąt jako ABC, a zadany punkt jako P. Wagi obliczane są jako ilorazy pól APB, BPC i CPA przez pole całego trójkąta ABC. W kodzie używamy wzoru na pole trójkąta $P = \frac{1}{2}a \times b$, przy czym pomijamy ułamek $\frac{1}{2}$, gdyż i tak ulegnie on skróceniu.

[3] https://anutom.blogspot.com/2011/07/wsporzedne-barycentryczne.html

Następnie zaimplementowano właściwą funkcję rysującą trójkąt wypełniony kolorem z wykorzystaniem interpolacji, draw_filled_triangle_with_color() (rysunek 36).

lab4.py

```
def draw_filled_triangle_with_color(image, a, color_a, b, color_b, c, color_c):
    xmin = min(a[0], b[0], c[0])
    xmax = max(a[0], b[0], c[0])
    ymin = min(a[1], b[1], c[1])
    ymax = max(a[1], b[1], c[1])

    for y in range(ymin, ymax + 1):
        for x in range(xmin, xmax + 1):
            w1, w2, w3 = barycentric_weights((x, y), a, b, c)
            if (w1 >= 0) and (w2 >= 0) and (w3 >= 0):
                r = int(w1 * color_a[0] + w2 * color_b[0] + w3 * color_c[0])
                g = int(w1 * color_a[1] + w2 * color_b[1] + w3 * color_c[1])
                b_col = int(w1 * color_a[2] + w2 * color_b[2] + w3 * color_c[2])
                draw_point(image, x, y, (r, g, b_col))
```

Rysunek 36

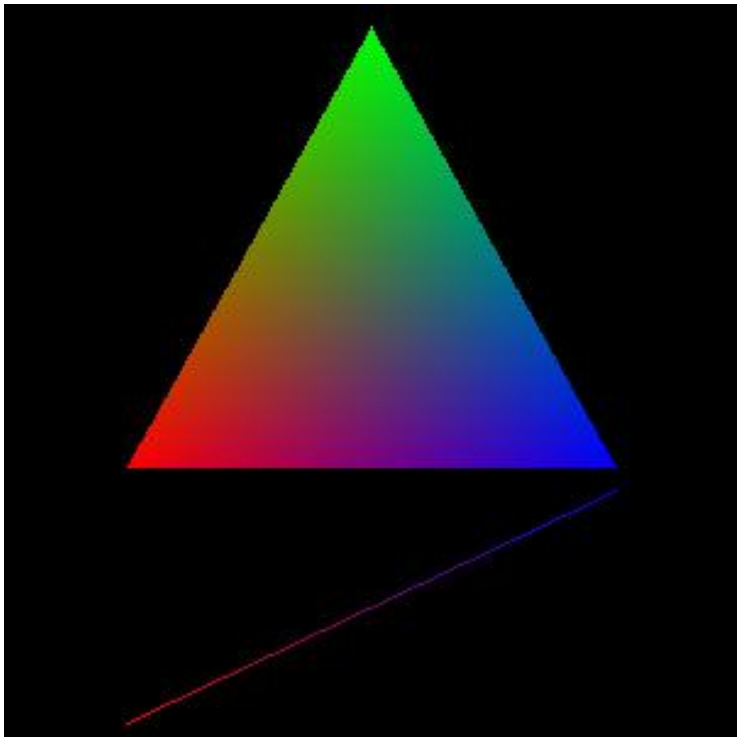
Funkcja ta jest modyfikacją funkcji rysująca trójkąt z poprzedniego zadania. Różnica tkwi wewnątrz drugiej pętli. Dla każdego punktu trójkąta wyznaczane są współrzędne barycentryczne względem trójkąta o wierzchołkach a, b, c. Użycie funkcji is_point_in_triangle() zastąpiono przyrównaniem wag do 0, ponieważ jeśli któraś z nich jest mniejsza od zera, to punkt leży poza trójkątem. Następnie obliczane są współrzędne RGB wg wzoru z instrukcji (rysunek 37).

$$\vec{C}_P = \frac{\lambda_0}{\lambda} \cdot \vec{V}0 + \frac{\lambda_1}{\lambda} \cdot \vec{V}1 + \frac{\lambda_2}{\lambda} \cdot \vec{V}2$$

Rysunek 37

Finalnie rysowany jest punkt o zadanym kolorze. Funkcje draw_line_with_color() oraz draw_filled_triangle_with_color() wywoływane są w funkcji process_4(), stworzonej analogicznie jak process_3(), ze zmianą funkcji rysujących jednokolorowe linie i trójkąty, na te z interpolacją koloru.

Wybranie opcji nr 4 w programie main.exe wywołuje funkcję process_4(). Wynik działania programu dany jest rysunkiem nr 38.



Rysunek 38

2.5 SSAA

Ostatnim zadaniem było zaimplementowanie wygładzania krawędzi z wykorzystaniem SSAA. Skorzystano z bibliotek języka Python jak w poprzednich zadaniach.

Wpierw zaimplementowano funkcję downscale_image() (rysunek 39), która zmniejsza wyrenderowany w większej rozdzielczości obraz do rozmiaru docelowego.

lab4.py

```python
def downscale_image(image, scale):
    small_height = image.shape[0] // scale
    small_width = image.shape[1] // scale
    small_image = np.zeros((small_height, small_width, 3), dtype=np.uint8)

    for y in range(small_height):
        for x in range(small_width):
            block = image[y * scale:(y + 1) * scale, x * scale:(x + 1) * scale, :]
            avg_color = block.mean(axis=(0, 1))
            small_image[y, x, :] = avg_color

    return small_image
```

Rysunek 39

Obliczane są wymiary przeskalowanego obrazu, a na ich podstawie tworzona jest tablica numpy wypełniona trójkami zer (czerń). Następnie przechodzimy po każdym bloku obrazu. Rozmiar bloku to zadana skala. Dla każdego bloku obliczana jest średnia, która definiuje kolor nowego piksela na pomniejszonym obrazie (średnia dla każdego z kanałów). Obliczone współrzędne są przypisywane do piksela odpowiadającego przetwarzanemu blokowi.

Całość SSAA wykonywana jest w funkcji process_5() (rysunek 40).

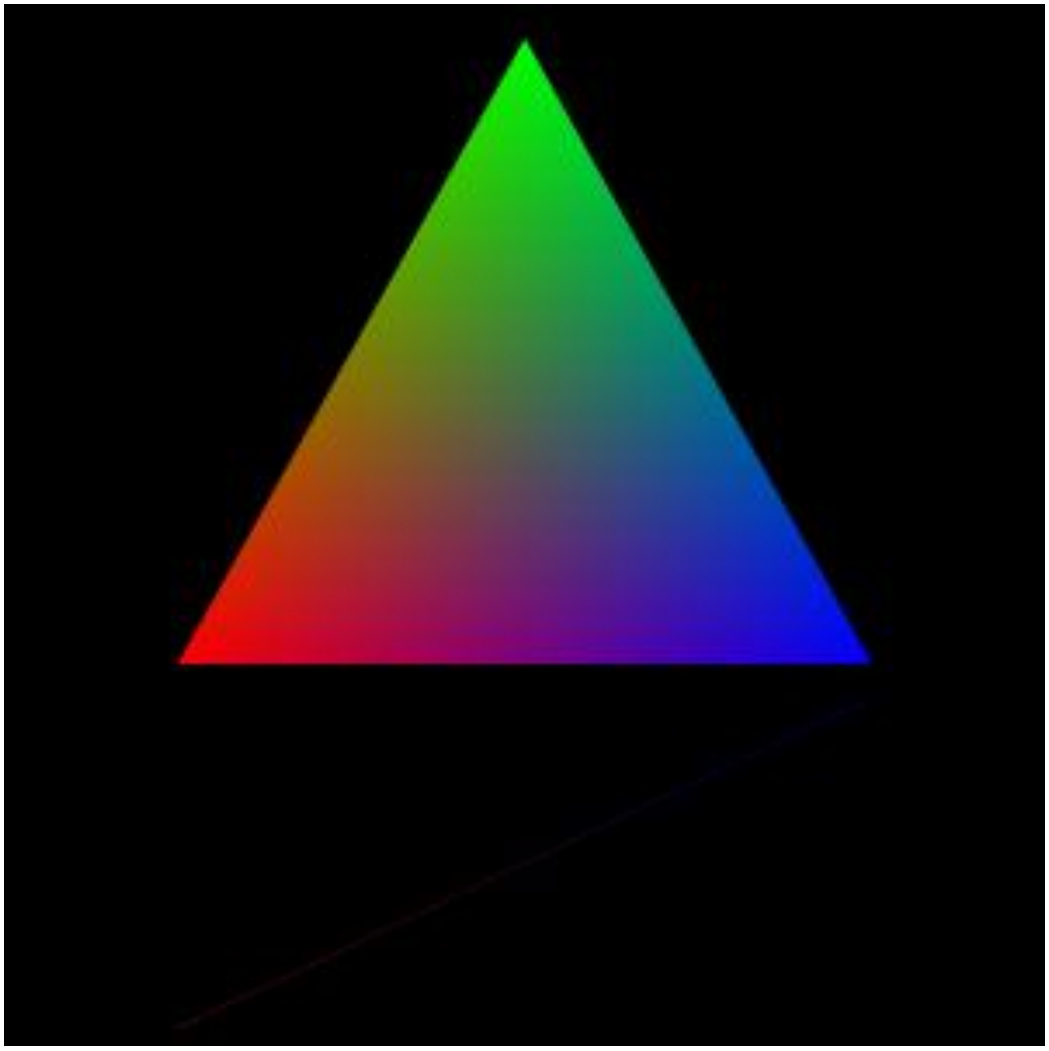
lab4.py

```python
def process_5():
    scale = 4
    width, height = 300, 300
    big_image = np.zeros((height * scale, width * scale, 3), dtype=np.uint8)

    draw_line_with_color(big_image, 50* scale, 5* scale, (255, 0, 0),
                         250* scale, 100* scale, (0, 0, 255))

    a = (50* scale, 110* scale)
    color_a = (255, 0, 0)
    b = (150* scale, 290* scale)
    color_b = (0, 255, 0)
    c = (250* scale, 110* scale)
    color_c = (0, 0, 255)
    draw_filled_triangle_with_color(big_image, a, color_a, b, color_b, c, color_c)

    final_image = downscale_image(big_image, scale)

    Image.fromarray(final_image).save("SSAA_interpolated_lines_and_triangle.png")
    Image.fromarray(final_image).show()
```

Rysunek 40

Zadajemy skalę jako 4. Tak więc rysujemy linie i trójkąt dokładnie tak samo jak w poprzednim zadaniu, z odpowiednim przeskalowaniem (obraz jest 4 razy większy). Następnie obraz ten przeskalowujemy w dół za pomocą funkcji downscale_image().

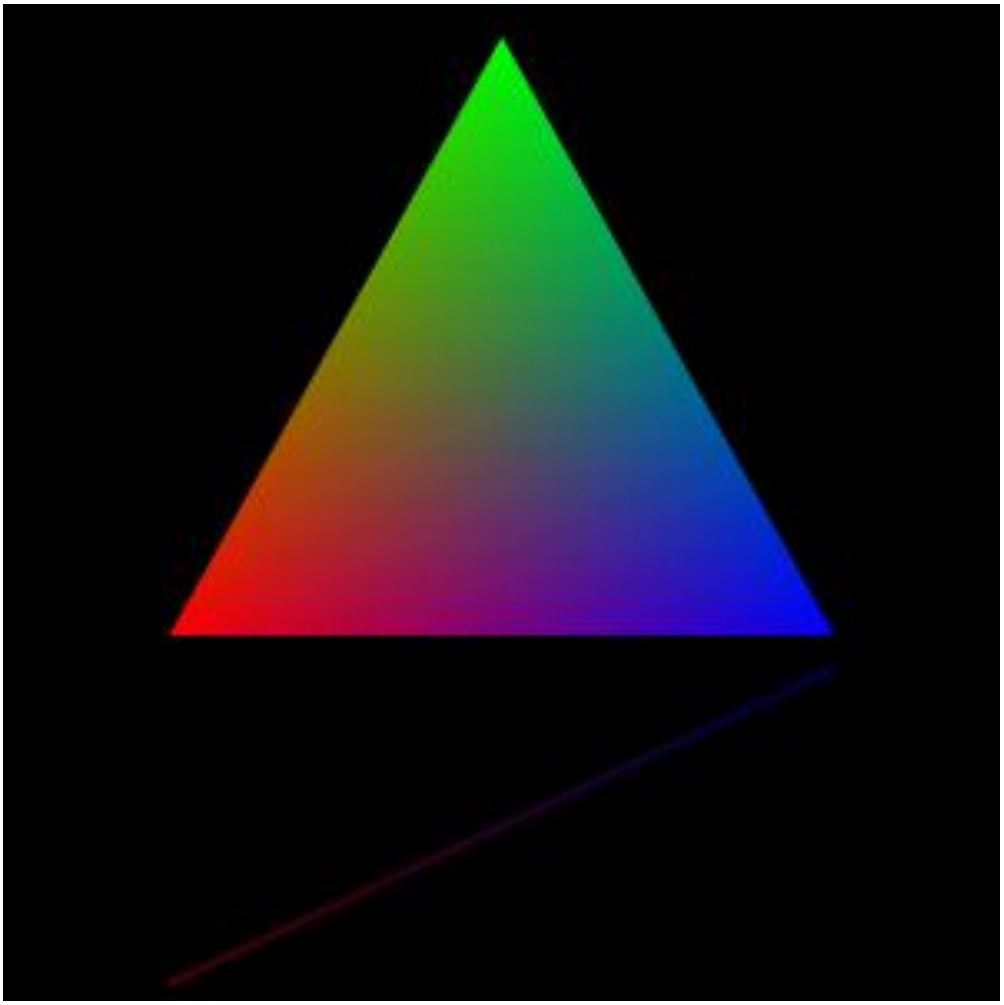
Wybranie opcji nr 5 w programie main.exe wywołuje funkcję process_5(). Wynik jej działania dany jest rysunkiem nr 41.



Rysunek 41

Widzimy ewidentnie gładsze krawędzie niż w poprzednim zadaniu, lecz linia stała się mniej widoczna. Potencjalnym rozwiązaniem problemu jest narysowanie grubszej linii (równoległoboku). Tak więc zaimplementowano funkcję process_5a() (rysunek 42), która jest prawie identyczna do process_5, z tą różnicą, że zamiast jednej linii, rysowane są trzy linie jedna nad drugą.

Wywołanie tej funkcji dodano do operacji wykonywanej przy wyborze opcji nr 5 w programie main.exe. Wynik jej działania dany jest rysunkiem nr 42. Widzimy na nim ewidentnie lepszą widoczność linii. Jest ona również gładsza wizualnie, w porównaniu z poprzednim zadaniem.



Rysunek 42

3. Struktura projektu

Plik wykonywalny main.exe znajduje się w katalogu dist. Zawiera on wszystkie obrazy potrzebne do prawidłowego wykonywania zadań. Katalog nadrzędny zawiera kod programu w języku Python oraz pliki wynikowe. Plik wykonywalny main.exe powstał za pomocą narzędzia PyInstaller.

Źródła

https://pl.wikipedia.org/wiki/Edvard_Munch#/media/Plik:The_Scream.jpg

https://scipython.com/blog/floyd-steinberg-dithering/

https://pl.wikipedia.org/wiki/Puentylizm

https://www.algorytm.org/przetwarzanie-obrazow/algorytm-floyda-steinberga.html

https://pl.wikipedia.org/wiki/Impresja,_wsch%C3%B3d_s%C5%82o%C5%84ca#/media/Plik:Claude_Monet,_Impression,_soleil_levant.jpg

https://onlineonly.christies.com/s/american-atmosphere-important-private-collection/louis-ritman-1889-1963-5/232310

https://pl.wikipedia.org/wiki/Niedzielne_popo%C5%82udnie_na_wyspie_Grande_Jatte#/media/Plik:A_Sunday_on_La_Grande_Jatte,_Georges_Seurat,_1884.jpg

https://pl.wikipedia.org/wiki/Algorytm_Bresenhama

https://ithardware.pl/poradniki/antyaliasing_dlss_ray_tracing_poznaj_ustawienia_graficzne-17021-3.html

https://anutom.blogspot.com/2011/07/wsporzedne-barycentryczne.html