Sprawozdanie
Inżynieria Obrazów
Laboratorium 4

Katsiaryna Kolyshko 276708

Dr. Inż. Jan Nikodem

Zadanie 1

Zadanie 1.

Zaimplementować dithering (algorytm Floyda–Steinberga) w skali szarości.
Wskazówki:

- odwzorować w języku Python kod ze slajdu 5,
- wygodnie będzie pracować na wartościach typu uint8,
- funkcja find_closest_palette_color() realizuje redukcję palety barw,
  w najprostszym układzie wystarczy zaokrąglić wartości w macierzy pikseli,
  `return round(value / 255) * 255`,
- dodanie błędu kwantyzacji może spowodować wyjście poza zakres typu,
- należy odpowiednio się przed tym zabezpieczyć i przyciąć wartości,
- w przeciwnym wypadku zaobserwujemy pojedyncze *bad pixele*,
- należy też uważać na zakres indeksów przy odwołaniach do tablicy pikseli.

Algorytm Floyda-Steinberga jest jedną z najbardziej znanych metod ditheringu stosowanych w przetwarzaniu obrazów cyfrowych. Dithering to technika, która symuluje większą liczbę kolorów niż jest faktycznie dostępna w urządzeniu wyjściowym, poprzez odpowiednie rozmieszczenie dostępnych kolorów.

Algorytm Floyda-Steinberga to adaptacyjna technika ditheringu z rozproszoną wartością błędu (error diffusion), opracowana przez Roberta Floyda i Louisa Steinberga w 1976 roku. W przeciwieństwie do prostych metod ditheringu bazujących na progowaniu (thresholding), metoda ta przechowuje informację o błędzie kwantyzacji i rozprasza go na sąsiednie, jeszcze nieprzetworzone piksele.

Algorytm działa poprzez przetwarzanie obrazu piksel po pikselu, najczęściej od lewej do prawej i od góry do dołu. Dla każdego piksela wykonywane są następujące kroki:

1. Aktualny piksel jest kwantyzowany do najbliższej wartości z dostępnej palety kolorów.
2. Obliczany jest błąd kwantyzacji (różnica między oryginalną a nową wartością).
3. Błąd ten jest rozprowadzany na sąsiednie, jeszcze nieprzetworzone piksele według określonego schematu dystrybucji.

W klasycznym algorytmie Floyda-Steinberga błąd jest rozprowadzany według następującego schematu (macierzy współczynników):

* 7/16

3/16 5/16 1/16

gdzie * oznacza aktualnie przetwarzany piksel, a wartości ułamkowe reprezentują część błędu, która jest dodawana do odpowiednich sąsiednich pikseli.

Algorytm Floyda-Steinberga znajduje zastosowanie w wielu obszarach przetwarzania obrazu:

1. Redukcja głębi barw - konwersja obrazów z dużą liczbą kolorów na obrazy o ograniczonej palecie kolorów, np. z 24-bitowych (True Color) na 8-bitowe lub nawet binarne (czarno-białe).
2. Druk - przygotowanie obrazów do druku na urządzeniach o ograniczonej możliwości reprodukcji półtonów.
3. Wyświetlacze o ograniczonej palecie - optymalizacja obrazów dla starszych lub prostszych wyświetlaczy.
4. Kompresja obrazów - jako część algorytmów kompresji z utratą jakości.

Zalety algorytmu Floyda-Steinberga

1. Wysoka jakość wizualna - zapewnia lepsze wyniki wizualne niż proste metody progowania.
2. Zachowanie szczegółów - dithering pozwala zachować więcej szczegółów oryginalnego obrazu.
3. Prostota implementacji - algorytm jest stosunkowo łatwy do zaimplementowania.
4. Wydajność obliczeniowa - wymaga tylko jednego przejścia przez obraz, co czyni go efektywnym obliczeniowo.

Wady algorytmu Floyda-Steinberga

1. Wzorce serpentyniowe - może tworzyć charakterystyczne wzorce, które są widoczne i mogą być niepożądane w niektórych zastosowaniach.
2. Akumulacja błędów - w pewnych przypadkach może dochodzić do nadmiernej akumulacji błędów. Akumulacja błędów w kontekście algorytmu Floyda-Steinberga to zjawisko, w którym błędy kwantyzacji rozprowadzane na sąsiednie piksele mogą się kumulować (nawarstwiać) w pewnych regionach obrazu. Prowadzi to do nadmiernego nagromadzenia błędu w niektórych obszarach, co objawia się jako niepożądane artefakty wizualne, takie jak ciemne lub jasne plamy, czy też nieregularne wzory, które nie występowały w oryginalnym obrazie.
3. Trudności z równoległym przetwarzaniem - sekwencyjny charakter algorytmu utrudnia równoległe przetwarzanie.
4. Problemy na krawędziach - na krawędziach obrazu mogą występować artefakty związane z brakiem możliwości rozprowadzenia błędu poza obszar obrazu.

Zastosowania w codziennym życiu

Algorytm Floyda-Steinberga i podobne techniki ditheringu są obecne w wielu urządzeniach i usługach, z których korzystamy na co dzień:

1. Drukarki - szczególnie drukarki atramentowe wykorzystują techniki ditheringu do symulacji półtonów.
2. E-czytniki - urządzenia z wyświetlaczami e-ink często stosują dithering do wyświetlania obrazów w skali szarości.

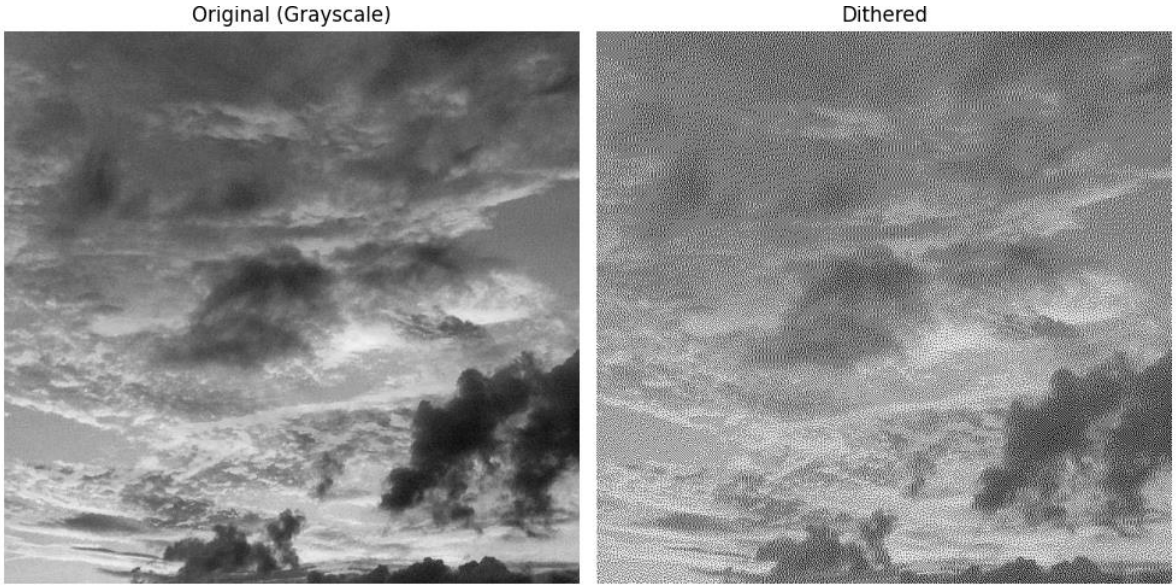
3. Monitory i wyświetlacze - niektóre panele LCD o niższej głębi kolorów używają ditheringu do symulowania większej liczby kolorów.
4. Grafika komputerowa i gry - szczególnie w starszych systemach lub grach retro, gdzie ograniczenia sprzętowe wymuszały stosowanie ditheringu.
5. Przetwarzanie wideo - niektóre kodeki wideo używają technik ditheringu do redukcji artefaktów po kompresji.

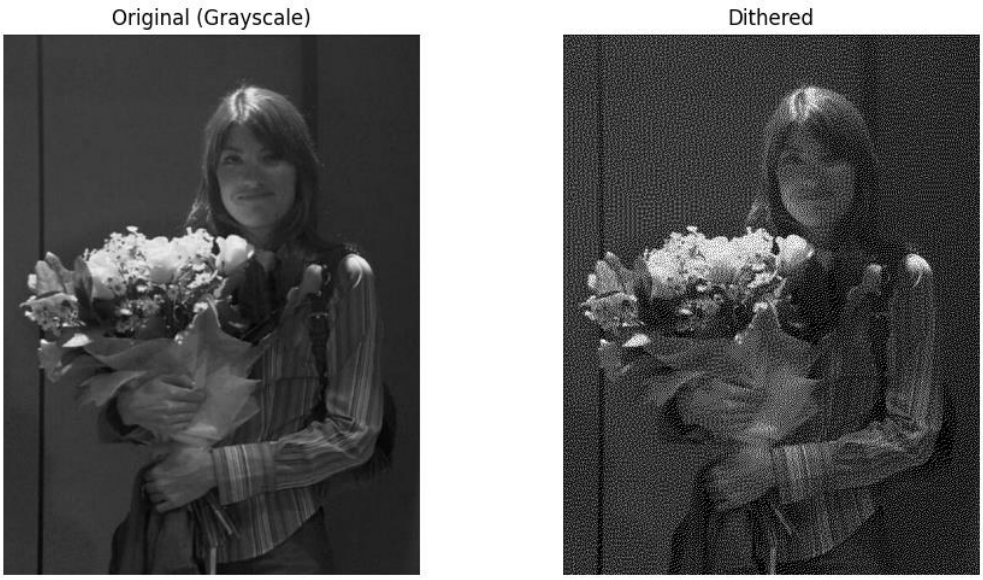
Implementacja algorytmu Floyda-Steinberga w skali szarości jest szczególnie przydatna, gdy chcemy zredukować głębię koloru obrazu do mniejszej liczby odcieni szarości, zachowując jednocześnie jak najwięcej szczegółów oryginalnego obrazu. W przypadku konwersji do obrazu binarnego (czarno-białego), algorytm ten pozwala na uzyskanie wysokiej jakości reprezentacji obrazu przy użyciu tylko dwóch kolorów.

Algorytm Floyda-Steinberga, mimo że został opracowany kilkadziesiąt lat temu, pozostaje istotnym narzędziem w przetwarzaniu obrazów cyfrowych i nadal znajduje zastosowanie w nowoczesnych technologiach.

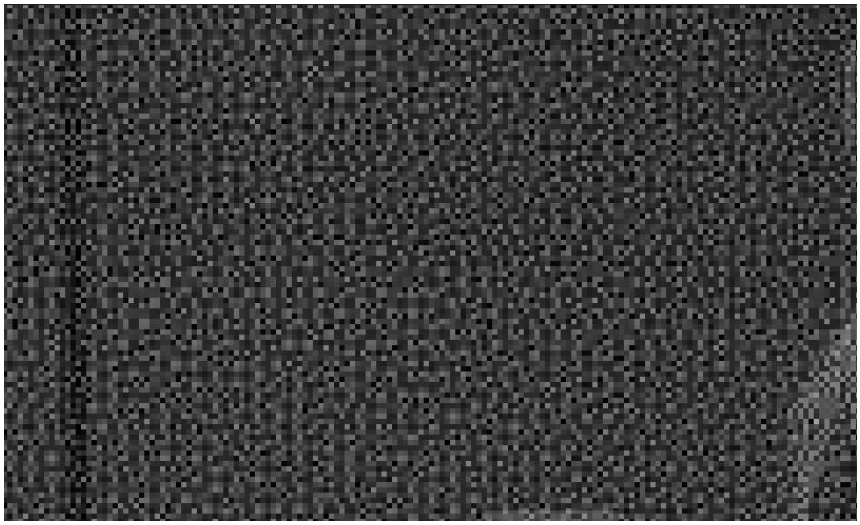
Analiza wyników

Po wypróbowaniu zadania 1 na dwóch obrazkach widzimy wyniki poniżej. Kod działa w następujący sposób: pyta użytkownika o obrazek, następnie konwertuje obrazek na obraz w skali szarości, a następnie wykonuje dithering. Poniżej znajdują się 2 zdjęcia przed i po: zdjęcie nieba i zdjęcie kobiety z kwiatami.





Jeśli powiększymy zdjęcia, które są rozproszone, zobaczymy dziwną papkę pikseli, która nie ma większego sensu. Ale jeśli pomniejszymy i spojrzymy na zdjęcie normalnie, nadal będziemy w stanie poprawnie rozpoznać obiekty na zdjęciu. Jest to związane z przestrzenną integracją sygnałów wzrokowych w naszym mózgu - zdolność do łączenia małych elementów w spójne całości. To właśnie dlatego gazety, stare monitory czy drukarki mogły skutecznie wyświetlać/drukować obrazy przy użyciu tylko czarnego tuszu na białym papierze, a nasze oczy nadal interpretowały je jako zawierające różne odcienie szarości.



Funkcja find_closest_palette_color() w kodzie pokazuje, że obrazy zostały zredukowane do palety zawierającej tylko dwa kolory: czarny (0) i biały (255). Każdy piksel jest zaokrąglany do jednej z tych wartości poprzez dzielenie przez 255 i mnożenie wyniku ponownie przez 255, co daje tylko wartości 0 lub 255. Oznacza to, że piksele o wartości poniżej 128 stają się czarne, a piksele o wartości 128 i powyżej stają się białe. To bardzo drastyczna redukcja z oryginalnych 256 odcieni szarości do tylko 2 poziomów.

Kod pokazuje, że błąd kwantyzacji (różnica między oryginalną wartością piksela a jego nową wartością po zaokrągleniu) jest dystrybuowany do sąsiednich pikseli zgodnie z klasyczną matrycą Floyda-Steinberga (7/16 dla piksela po prawej, 3/16 dla piksela na dole po lewej, 5/16 dla piksela na dole, 1/16 dla piksela na dole po prawej).

Podsumowanie:

1. Algorytm utrzymuje średnią luminancję regionów obrazu przez rozprowadzanie błędu.
2. Piksele są rozmieszczone w sposób nieregularny, co zapobiega tworzeniu widocznych regularnych wzorów.
3. Ludzkie oko ma ograniczoną rozdzielczość i naturalnie integruje małe struktury w większe całości.

Opis działania kodu

find_closest_palette_color()

Funkcja ta przyjmuje wartość piksela w skali szarości i kwantyzuje ją do czarnego (0) lub białego (255). Działa poprzez podzielenie wartości wejściowej przez 255, zaokrąglenie do najbliższej liczby całkowitej (0 lub 1), a następnie pomnożenie ponownie przez 255. To efektywnie zmusza każdy piksel do bycia albo czystym czarnym, albo czystym białym, co jest pierwszym krokiem w procesie ditheringu.

apply_floyd_steinberg_dithering()

Jest to główna funkcja implementująca algorytm ditheringu Floyda-Steinberga. Otwiera plik obrazu, konwertuje go do skali szarości jeśli to konieczne, i przetwarza piksel po pikselu. Dla każdego piksela określa najbliższy kolor z palety (czarny lub biały) i oblicza błąd kwantyzacji (różnicę między oryginalną wartością piksela a nową wartością czarno-białą). Ten błąd jest następnie rozprowadzany do

sąsiednich pikseli zgodnie ze schematem dyfuzji Floyda-Steinberga: 7/16 na prawo, 3/16 na dół w lewo, 5/16 bezpośrednio w dół i 1/16 na dół w prawo. Ta dyfuzja błędu tworzy iluzję większej liczby poziomów szarości niż faktycznie występuje, zachowując ogólną luminancję obrazu przy użyciu tylko dwóch kolorów.

main()

Jest to funkcja wejściowa, która obsługuje interakcję z użytkownikiem i przepływ programu. Prosi użytkownika o ścieżki do plików wejściowych i wyjściowych, weryfikuje istnienie pliku wejściowego i zarządza procesem ditheringu. Jeśli obraz wejściowy nie jest już w trybie skali szarości, konwertuje go, tworząc w razie potrzeby plik tymczasowy. Po przetworzeniu próbuje wyświetlić zarówno oryginalny, jak i poddany ditheringowi obraz obok siebie za pomocą biblioteki matplotlib (jeśli jest dostępna). Zawiera również obsługę błędów i czyszczenie wszelkich plików tymczasowych utworzonych podczas przetwarzania.

Zadanie 2

Zadanie 2.

uwzględnić kolory w algorytmie Floyda–Steinberga.
Wskazówki:

- uwagi i kroki do wykonania są analogiczne jak w zadaniu poprzednim,
- dodatkowo umożliwić wybór ile wartości składowych chcemy zachować,
  domyślnie mają to być dwie wartości dla każdej składowej – 0 i 255,
  `return round((k - 1) * value / 255) * 255 / (k - 1),`
- należy obliczyć błąd kwantyzacji osobno dla każdej składowej koloru,
- dla porównania wyświetlić jak wygląda obrazek po samej redukcji barw,
- skuteczność algorytmu potwierdzić prezentując histogram kolorów.

Rozszerzenie algorytmu Floyda-Steinberga z obrazów w skali szarości na obrazy kolorowe wymaga uwzględnienia wielowymiarowej przestrzeni kolorów. W przeciwieństwie do obrazów w skali szarości, gdzie mamy do czynienia z jednym kanałem jasności, obrazy kolorowe zazwyczaj składają się z trzech kanałów (RGB).

Główne różnice przy przetwarzaniu obrazów kolorowych:

1. Niezależne przetwarzanie kanałów - w implementacji kolorowej każdy kanał koloru (R, G, B) jest przetwarzany niezależnie, z osobnym obliczaniem i dystrybucją błędu kwantyzacji dla każdego kanału.

2. Kwantyzacja wielowymiarowa - redukcja palety kolorów staje się problemem wielowymiarowym. Formuła round((k - 1) * value / 255) * 255 / (k - 1) pozwala na kwantyzację każdego kanału do k dyskretnych poziomów, gdzie k to pożądana liczba wartości dla każdego kanału (domyślnie 2, co daje wartości 0 i 255).

3. Paleta kolorów - przy k=2 dla każdego kanału, otrzymujemy paletę 8 kolorów ($2^3 = 8$): czarny, czerwony, zielony, niebieski, cyjan, magenta, żółty i biały.

Zalety ditheringu kolorowego:

1. Zachowanie kolorów - możliwość symulowania znacznie większej liczby kolorów niż jest faktycznie dostępnych w ograniczonej palecie.
2. Redukcja bandingu - eliminacja widocznych "pasm" w gradientach kolorów, które często pojawiają się przy prostej kwantyzacji.
3. Elastyczność redukcji - możliwość wyboru liczby poziomów dla każdego kanału pozwala na precyzyjne dostosowanie kompromisu między jakością a rozmiarem palety.

Zastosowania ditheringu kolorowego:

1. Redukcja głębi kolorów - konwersja obrazów 24-bitowych (True Color) do formatów o mniejszej liczbie kolorów, np. 8-bitowych (256 kolorów).
2. Grafika internetowa - optymalizacja obrazów dla starszych przeglądarek lub urządzeń o ograniczonej mocy obliczeniowej.
3. Grafika retro - tworzenie stylistycznej grafiki naśladującej ograniczenia starszych systemów komputerowych.
4. Druk kolorowy - przygotowanie obrazów do druku metodą rastrową, gdzie mieszanie kolorów jest ograniczone.

Skuteczność ditheringu kolorowego można ocenić poprzez:

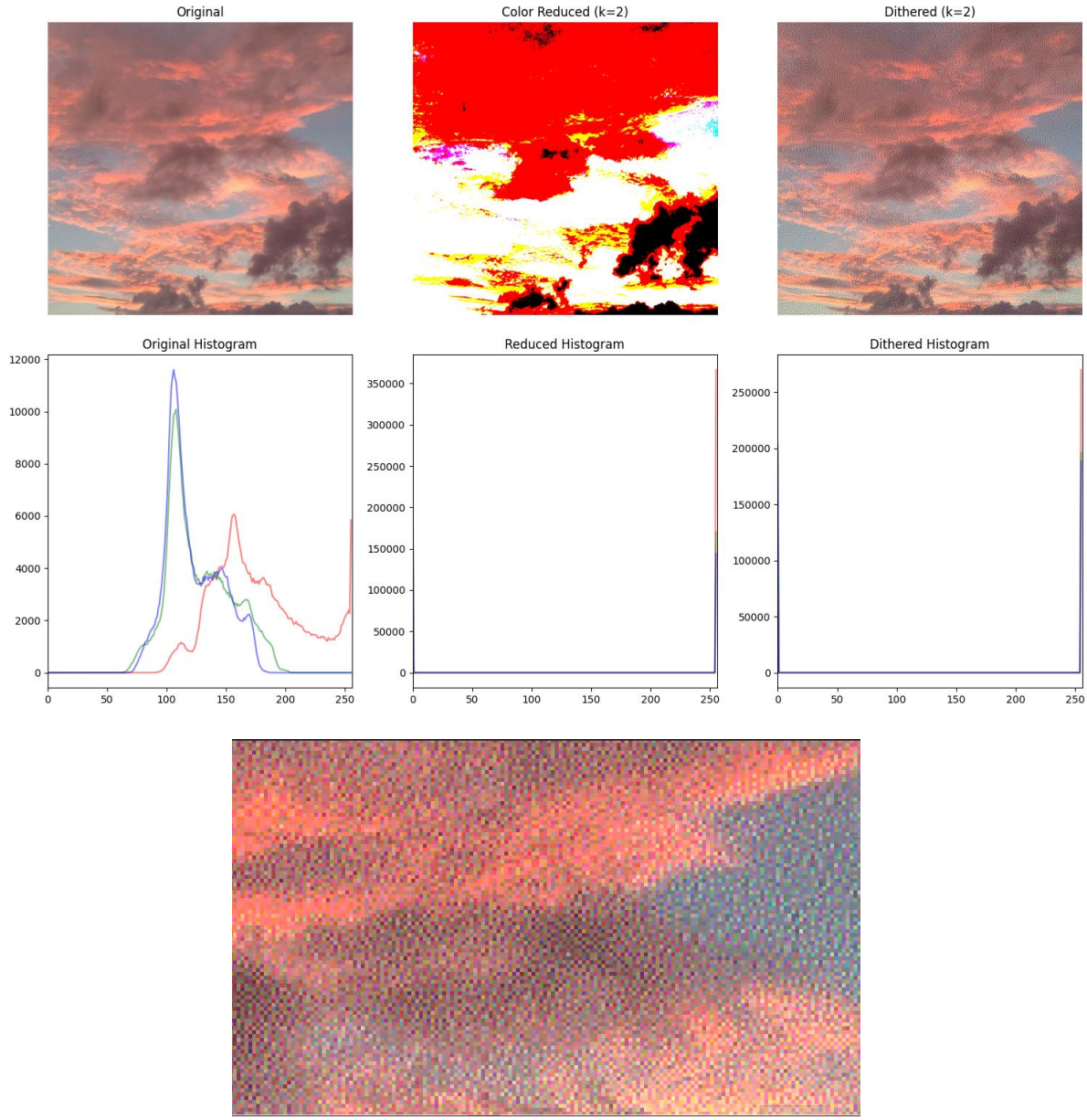
1. Porównanie wizualne - zestawienie obrazu po samej redukcji kolorów (bez ditheringu) z obrazem po zastosowaniu algorytmu Floyda-Steinberga.
2. Analiza histogramu - histogram kolorów przed i po ditheringu pomaga zrozumieć, jak algorytm wpływa na dystrybucję kolorów w obrazie.
3. Badanie zachowania detali - ocena, jak dobrze zachowane są szczegóły, krawędzie i subtelne przejścia kolorów.

Warto zauważyć, że dithering kolorowy z bardzo ograniczoną paletą (np. 8 kolorów) jest znacznie bardziej wymagający niż dithering w skali szarości, ponieważ musi symulować zarówno zmiany jasności, jak i barwy. Mimo to, algorytm Floyda-Steinberga radzi sobie zaskakująco dobrze, tworząc wizualnie akceptowalne rezultaty nawet przy drastycznej redukcji liczby kolorów.

Analiza wyników

Poniżej znajdują się wyniki działania kodu. Program pyta o liczbę poziomów na kanał. Poniżej znajdują się wyniki dla liczb: 2, 4, 100, 256.

Liczba k = 2 (8 kolorów):

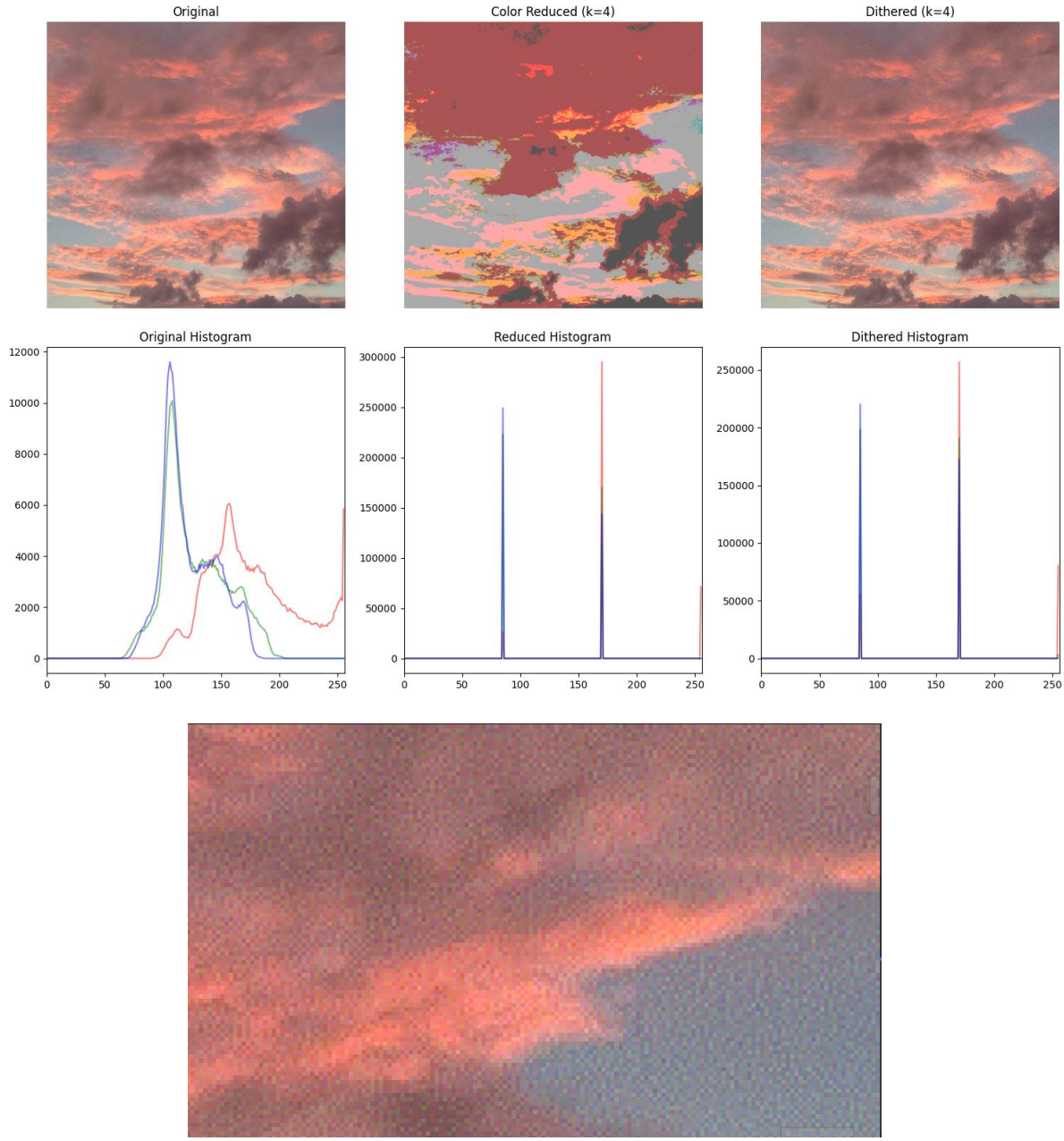


Obraz: Przy k=2 obraz po samej redukcji kolorów (środkowy) jest drastycznie uproszczony i prawie nierozpoznawalny. Dominują jednolite, jaskrawe obszary czerwieni, bieli i innych podstawowych kolorów, a szczegóły zostały całkowicie utracone.

Histogram: Histogram obrazu zredukowanego pokazuje tylko kilka ostrych pików w miejscach, gdzie k=2 zezwala na wartości (0 i 255 dla każdego kanału). Histogram jest ekstremalnie uproszczony w porównaniu z oryginalnym, płynnym rozkładem.

Dithering: Zastosowanie ditheringu Floyda-Steinberga znacząco poprawiło jakość wizualną. Mimo używania tylko 8 kolorów (2^3), obraz z ditheringiem jest zaskakująco podobny do oryginału. Poprzez rozprowadzanie błędu kwantyzacji, algorytm stworzył iluzję znacznie większej liczby kolorów.

Liczba k = 4 (64 kolory):

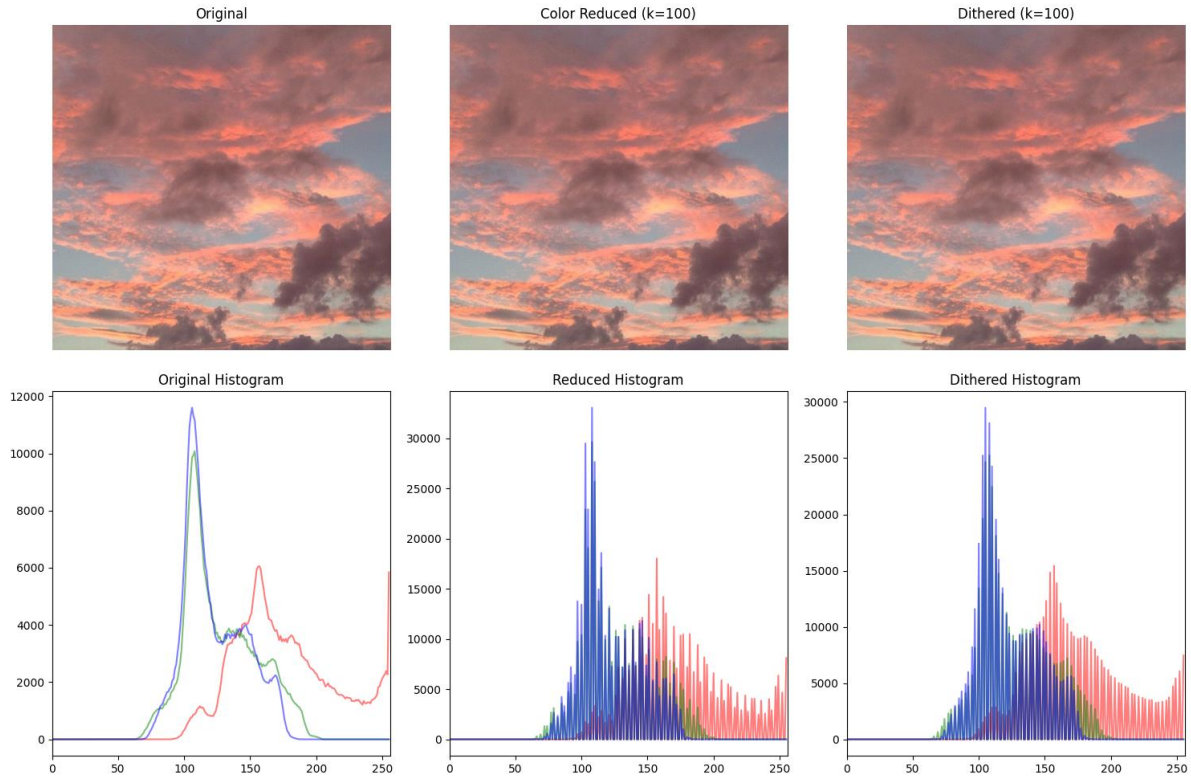


Obraz: Przy k=4 obraz po samej redukcji jest już znacznie lepszy niż przy k=2, ale nadal widoczne są obszary jednolitych kolorów i "bandy" (pasmowanie) w miejscach, gdzie powinny być płynne przejścia.

Histogram: Histogram pokazuje więcej pików (do 4 poziomów na kanał), co pozwala na lepsze przybliżenie oryginalnego rozkładu kolorów, choć nadal występują wyraźne luki między wartościami.

Dithering: Obraz po ditheringu jest już prawie nie do odróżnienia od oryginału. Przy 64 kolorach plus dithering, ludzkie oko praktycznie nie dostrzega różnicy w porównaniu z oryginalnymi milionami kolorów.

Liczba k = 100 (1 milion kolorów):

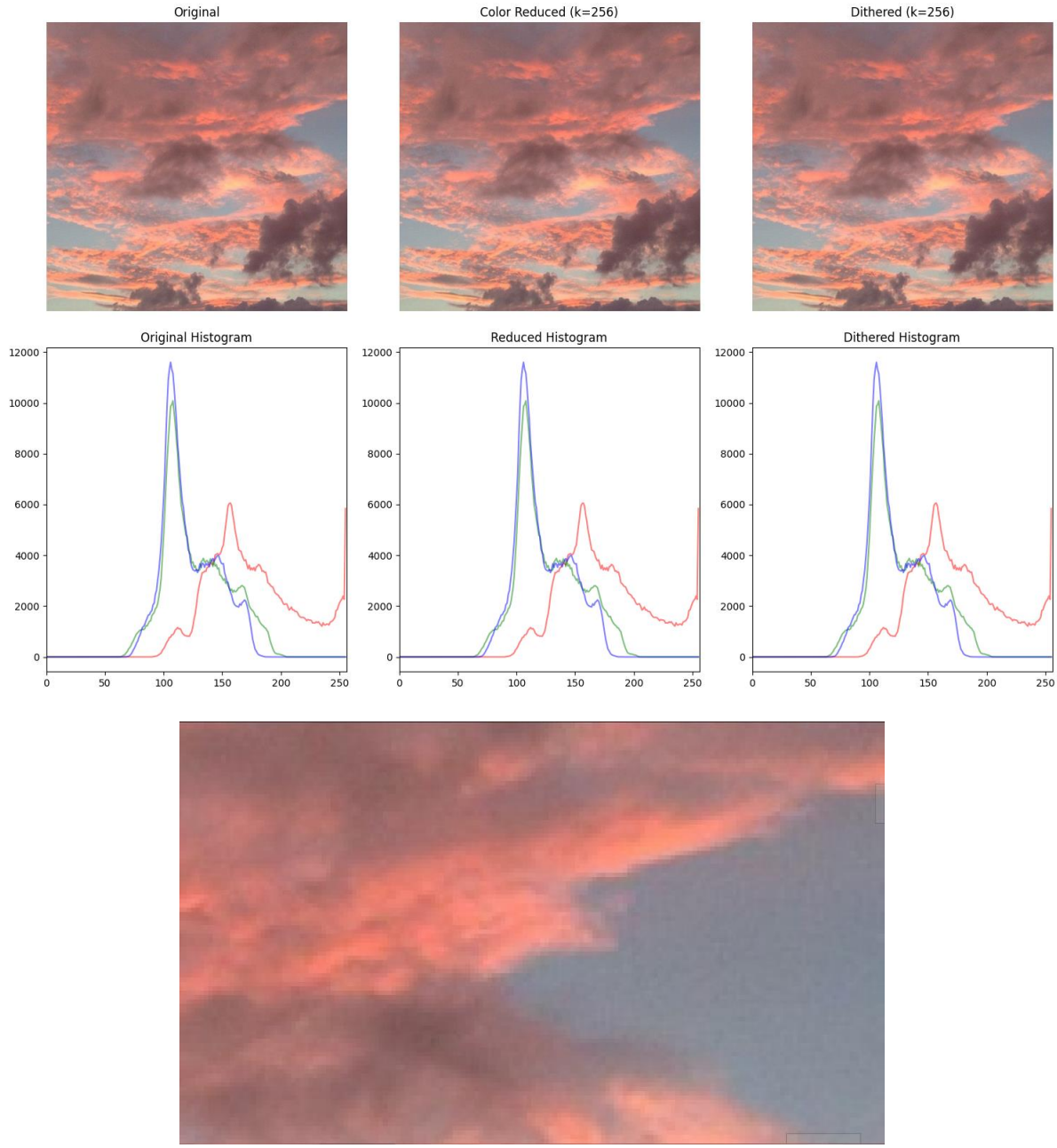


Obraz: Przy k=100 obraz po redukcji kolorów jest już bardzo zbliżony do oryginału. Trudno dostrzec różnice bez bezpośredniego porównania.

Histogram: Histogram zredukowanego obrazu wykazuje wiele dyskretnychpików, ale są one na tyle gęsto rozmieszczone, że ogólny kształt rozkładu jest bardzo podobny do oryginału. Widać wyraźne "grzebienie" reprezentujące dozwolone wartości.

Dithering: Przy tak dużej liczbie kolorów bazowych, wpływ ditheringu jest minimalny. Histogramy i obrazy z redukcją kolorów i z ditheringiem są prawie identyczne, ponieważ błąd kwantyzacji jest już bardzo mały.

Liczba k = 256 (16,7 miliona kolorów):



Obraz: Przy k=256 wszystkie trzy obrazy (oryginalny, zredukowany i z ditheringiem) są wizualnie identyczne. To logiczne, ponieważ kanały RGB w standardowych obrazach mają właśnie 256 poziomów jasności na kanał.

Histogram: Wszystkie trzy histogramy są praktycznie identyczne, co potwierdza, że przy k=256 nie następuje żadna faktyczna redukcja kolorów w stosunku do oryginału.

Dithering: Dithering nie wprowadza żadnych widocznych zmian, ponieważ nie ma błędu kwantyzacji do rozprowadzenia.

Podsumowanie i porównanie

Porównując wszystkie cztery wartości k, można zauważyć następujące prawidłowości:

1. Wpływ parametru k: Im niższa wartość k, tym bardziej drastyczna redukcja kolorów i tym ważniejszy staje się dithering. Przy k=2 różnica między obrazem zredukowanym a z ditheringiem jest ogromna, podczas gdy przy k=256 różnica jest niezauważalna.
2. Histogramy: Histogramy doskonale obrazują proces kwantyzacji - przy niskich wartościach k widać pojedyncze, ostre piki, które wraz ze wzrostem k stają się coraz gęstsze, aż przy k=256 praktycznie odtwarzają oryginalny rozkład.
3. Skuteczność ditheringu: Dithering jest najbardziej skuteczny przy niskich wartościach k. Przy k=2 i k=4 algorytm Floyda-Steinberga znakomicie symuluje kolory, których faktycznie nie ma w palecie. Przy wyższych wartościach k jego wpływ maleje.
4. Kompromis między jakością a rozmiarem palety: Dla większości zastosowań k=4 (64 kolory) z ditheringiem zapewnia doskonałą jakość wizualną przy drastycznej redukcji liczby kolorów, co może być ważne w zastosowaniach o ograniczonych zasobach.

Podsumowując, algorytm Floyda-Steinberga jest szczególnie wartościowy przy dużej redukcji palety kolorów, gdzie znacząco poprawia percepcję obrazu. Ta właściwość czyni go niezastąpionym w wielu praktycznych zastosowaniach, od grafiki retro po druk i optymalizację obrazów.

Opis działania kodu

find_closest_palette_color()

Funkcja ta znajduje najbliższy kolor z ograniczonej palety dla podanej wartości piksela. Parametr k określa liczbę poziomów dla każdego kanału koloru. Wykorzystuje formułę round((k - 1) * value / 255) * 255 / (k - 1), która kwantyzuje wartość do jednego z k równomiernie rozłożonych poziomów w zakresie 0-255. Jeśli k wynosi 1, funkcja zawsze zwraca 0 (ponieważ mamy tylko jeden możliwy poziom).

apply_color_reduction()

Funkcja ta stosuje prostą redukcję kolorów bez ditheringu. Otwiera obraz RGB, a następnie dla każdego piksela i każdego kanału koloru znajduje najbliższy dozwolony poziom przy użyciu funkcji find_closest_palette_color. Efektem jest obraz z ograniczoną paletą kolorów, ale bez rozpraszania błędu kwantyzacji, co powoduje wyraźne prążkowanie (banding) w obszarach płynnych przejść kolorów.

apply_floyd_steinberg_dithering()

To główna funkcja implementująca algorytm ditheringu Floyda-Steinberga dla obrazów kolorowych. Działa podobnie jak wersja dla skali szarości, ale przetwarza wszystkie trzy kanały RGB niezależnie. Dla każdego piksela określa najbliższy dozwolony kolor w ograniczonej palecie, oblicza błąd kwantyzacji dla każdego kanału, a następnie rozprowadza ten błąd do sąsiednich pikseli według wzoru Floyda-Steinberga. To rozprowadzanie błędu tworzy iluzję większej liczby kolorów, zachowując jednocześnie ogólną kolorystykę obrazu.

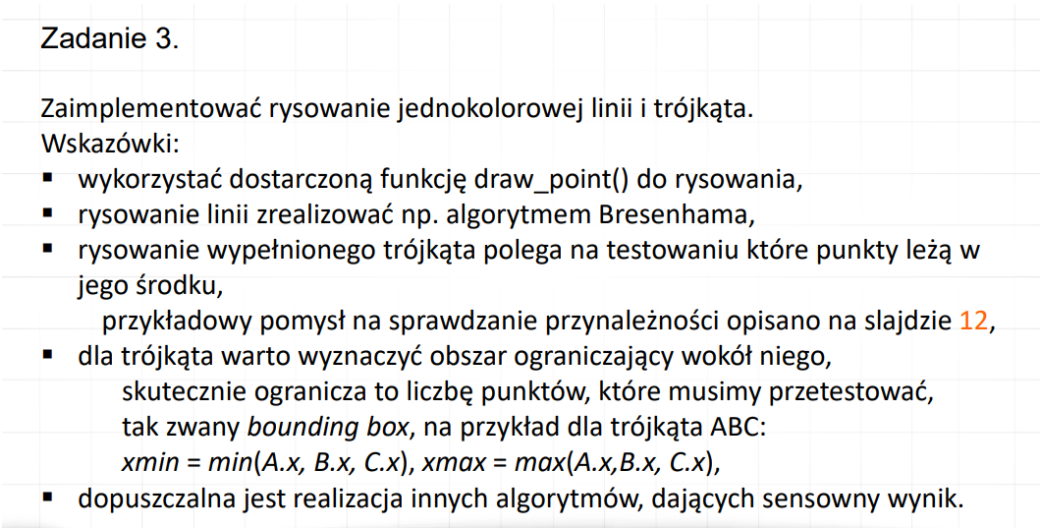
plot_color_histogram()

Funkcja pomocnicza, która tworzy histogram kolorów dla obrazu. Rysuje oddzielne histogramy dla kanałów czerwonego, zielonego i niebieskiego, co pozwala na wizualną analizę rozkładu kolorów w obrazie przed i po przetworzeniu.

main()

Funkcja wejściowa programu zarządzająca interakcją z użytkownikiem i przepływem danych. Pozwala użytkownikowi na wybór pliku wejściowego, katalogu wyjściowego oraz parametru k, określającego liczbę poziomów na kanał kolorów. Następnie przetwarza obraz na dwa sposoby: stosując prostą redukcję kolorów oraz dithering Floyda-Steinberga. Na koniec tworzy porównawczą wizualizację zawierającą oryginał, obraz po redukcji kolorów oraz obraz po ditheringu, wraz z odpowiednimi histogramami kolorów, co pozwala na lepsze zrozumienie efektów tych technik.

Zadanie 3



Zadanie 3.

Zaimplementować rysowanie jednokolorowej linii i trójkąta.
Wskazówki:

- wykorzystać dostarczoną funkcję draw_point() do rysowania,
- rysowanie linii zrealizować np. algorytmem Bresenhama,
- rysowanie wypełnionego trójkąta polega na testowaniu które punkty leżą w jego środku,
  przykładowy pomysł na sprawdzanie przynależności opisano na slajdzie 12,
- dla trójkąta warto wyznaczyć obszar ograniczający wokół niego,
  skutecznie ogranicza to liczbę punktów, które musimy przetestować,
  tak zwany *bounding box*, na przykład dla trójkąta ABC:
  xmin = min(A.x, B.x, C.x), xmax = max(A.x,B.x, C.x),
- dopuszczalna jest realizacja innych algorytmów, dających sensowny wynik.

Rasteryzacja to proces konwersji obiektów wektorowych (takich jak linie czy wielokąty) na obraz rastrowy składający się z pikseli. Jest to kluczowy element w grafice komputerowej, ponieważ większość urządzeń wyświetlających (monitory, wyświetlacze smartfonów) działa w oparciu o matryce pikseli.

Algorytm Bresenhama

Algorytm Bresenhama to efektywna metoda rasteryzacji linii, opracowana przez Jacka Bresenhama w 1962 roku. Jest ceniona za swoją wydajność, ponieważ wykorzystuje wyłącznie operacje na liczbach całkowitych (dodawanie, odejmowanie i przesunięcia bitowe), unikając kosztownych operacji zmiennoprzecinkowych.

Podstawowa zasada działania algorytmu Bresenhama polega na śledzeniu błędu akumulacji podczas przemieszczania się wzdłuż linii i podejmowaniu decyzji o przesunięciu w pionie lub po przekątnej na podstawie tego błędu.

Rasteryzacja trójkątów jest fundamentalną operacją w grafice komputerowej, ponieważ trójkąty są podstawowymi prymitywami używanymi do budowania bardziej złożonych modeli 3D.

Podstawowa metoda rasteryzacji trójkąta polega na przetestowaniu wszystkich pikseli w obrębie prostokątnego obszaru ograniczającego (bounding box) i sprawdzeniu, które z nich leżą wewnątrz trójkąta. Istnieje kilka metod określania, czy punkt leży wewnątrz trójkąta:

1. Metoda współrzędnych barycentrycznych - punkt P leży wewnątrz trójkąta ABC, jeśli można go wyrazić jako kombinację liniową wierzchołków trójkąta z współczynnikami dodatnimi sumującymi się do 1.
2. Metoda półpłaszczyzn - punkt leży wewnątrz trójkąta, jeśli znajduje się po tej samej stronie każdej z trzech prostych wyznaczonych przez boki trójkąta.
3. Metoda pola trójkątów - punkt leży wewnątrz trójkąta, jeśli suma pól trójkątów utworzonych przez punkt i dwa kolejne wierzchołki trójkąta jest równa polu oryginalnego trójkąta.

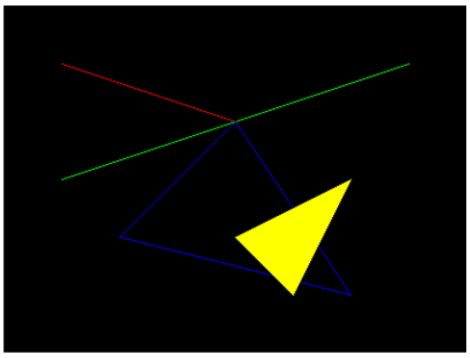
Aby zoptymalizować proces rasteryzacji trójkąta, stosuje się technikę bounding box (prostokątnego obszaru ograniczającego). Zamiast testować wszystkie piksele obrazu, testujemy tylko piksele znajdujące się w prostokącie, którego granice wyznaczają wartości minimalne i maksymalne współrzędnych wierzchołków trójkąta:

$$xmin = min(A.x, B.x, C.x), xmax = max(A.x, B.x, C.x)$$

$$ymin = min(A.y, B.y, C.y), ymax = max(A.y, B.y, C.y)$$

Rasteryzacja w grafice 2D i 3D opiera się na tych samych podstawowych algorytmach, z tą różnicą, że w grafice 3D najpierw następuje projekcja obiektów z przestrzeni trójwymiarowej na płaszczyznę dwuwymiarową, a dopiero potem rasteryzacja otrzymanych kształtów 2D. Mimo rozwoju technik opartych na śledzeniu promieni (ray tracing), rasteryzacja pozostaje dominującą techniką renderowania w grach komputerowych ze względu na jej wydajność i możliwość renderowania w czasie rzeczywistym. Zaawansowane techniki rasteryzacji, takie jak deferred shading czy tile-based rendering, pozwalają na tworzenie fotorealistycznej grafiki przy zachowaniu wydajności niezbędnej do płynnego działania gier.

Wyniki działania



Na czarnym tle (canvas) zostały wyrenderowane następujące elementy:

1. Czerwona linia - poprowadzona ukośnie od współrzędnych (50, 50) do (200, 100), narysowana algorytmem Bresenhama.

2. Zielona linia - biegnąca od (50, 150) do (350, 50), również przy użyciu algorytmu Bresenhama.
3. Niebieski obrys trójkąta - łączący punkty (100, 200), (200, 100) i (300, 250), złożony z trzech linii narysowanych algorytmem Bresenhama.
4. Żółty wypełniony trójkąt - z wierzchołkami w punktach (200, 200), (300, 150) i (250, 250), wyznaczony metodą testowania punktów wewnątrz trójkąta.

Wszystkie te elementy pojawiają się na czarnym tle, dokładnie tak, jak określono w kodzie. Algorytm Bresenhama jest prawidłowo zaimplementowany, ponieważ linie wyglądają gładko, bez żadnych przerw, a algorytm wypełniania trójkątów poprawnie identyfikuje i wypełnia wszystkie punkty wewnątrz żółtego trójkąta.

Celem tej demonstracji jest pokazanie:

Różnokolorowych elementów — czerwone i zielone linie, niebieski kontur trójkąta i żółty wypełniony trójkąt zapewniają wizualną przejrzystość między różnymi technikami renderowania.

Podstawowa weryfikacja implementacji — posiadanie tych różnych elementów potwierdza, że zarówno algorytm linii Bresenhama, jak i algorytm wypełniania trójkątów działają poprawnie.

Widoczność na czarnym tle — użycie jasnych kolorów na czarnym tle sprawia, że elementy są łatwe do zobaczenia.

Różne kąty linii — linie są pod różnymi kątami, aby pokazać, że algorytm Bresenhama działa w różnych kierunkach.

Zarówno kontur, jak i wypełnione trójkąty — pokazanie zarówno konturu trójkąta, jak i wypełnionego trójkąta pokazuje dwa główne sposoby renderowania trójkątów.

Algorytm Bresenhama zaimplementowany w metodzie draw_line_bresenham działa następująco:

1. Najpierw określa różnice między punktami końcowymi w obu osiach (dx i dy) oraz kierunki rysowania (sx i sy).
2. Następnie ustala, która oś jest dominująca (ta, wzdłuż której jest większa zmiana).
3. Inicjalizuje parametr błędu err.
4. W głównej pętli:
    - Rysuje piksel w aktualnej pozycji
    - Zawsze przesuwa się wzdłuż osi dominującej
    - Podejmuje decyzję o przesunięciu wzdłuż drugiej osi na podstawie wartości błędu
    - Aktualizuje błąd

Dzięki temu algorytm zapewnia płynne linie bez przerw, mimo dyskretnej natury pikseli. Widoczne na obrazie czerwona i zielona linia oraz krawędzie niebieskiego trójkąta są narysowane właśnie tym algorytmem.

Żółty trójkąt jest wypełniony przy użyciu metody draw_filled_triangle, która:

1. Wyznacza bounding box (prostokątny obszar ograniczający) wokół trójkąta, określony przez minimalne i maksymalne współrzędne wierzchołków.
2. Dla każdego piksela w tym prostokącie sprawdza, czy leży on wewnątrz trójkąta przy użyciu metody współrzędnych barycentrycznych, zaimplementowanej w funkcji is_point_in_triangle.
3. Jeśli piksel leży wewnątrz trójkąta, koloruje go na żółto.

Metoda sprawdzania, czy punkt leży wewnątrz trójkąta, opiera się na sumowaniu pól trójkątów utworzonych przez testowany punkt i wierzchołki oryginalnego trójkąta. Jeśli suma pól tych trzech trójkątów jest równa polu oryginalnego trójkąta (z pewną tolerancją dla błędów zaokrągleń), punkt leży wewnątrz.

Obraz doskonale ilustruje działanie podstawowych algorytmów rasteryzacji. Algorytm Bresenhama tworzy płynne linie bez przerw, a metoda wypełniania trójkąta poprawnie identyfikuje i koloruje wszystkie piksele leżące wewnątrz trójkąta. Zastosowanie różnych kolorów dla poszczególnych elementów pozwala na łatwą wizualną weryfikację poprawności implementacji.

Opis działania kodu

__init__()

Funkcja inicjalizująca klasę Rasterizer, która tworzy czarny obszar (płótno) o określonej szerokości i wysokości. Obszar ten jest zaimplementowany jako trójwymiarowa tablica NumPy, gdzie dwa pierwsze wymiary odpowiadają pikselom obrazu, a trzeci wymiar reprezentuje trzy kanały koloru (RGB).

clear()

Funkcja ta czyści całe płótno, ustawiając wszystkie piksele na określony kolor (domyślnie czarny). Jest to przydatne, gdy chcemy zacząć rysowanie od nowa bez tworzenia nowego obiektu rasteryzatora.

draw_point()

Prosta funkcja do rysowania pojedynczego piksela na płótnie. Sprawdza, czy współrzędne mieszczą się w granicach płótna, a następnie ustawia kolor danego piksela. Ta funkcja jest podstawowym elementem wszystkich bardziej złożonych algorytmów rysowania.

draw_line_bresenham()

Implementacja algorytmu Bresenhama do rysowania linii. Jest to wydajny algorytm, który używa tylko operacji całkowitoliczbowych do określenia, które piksele należy narysować, aby utworzyć prostą linię między dwoma punktami. Algorytm analizuje, która oś (X czy Y) ma większą zmianę, i odpowiednio dostosowuje procedurę rysowania, aby linia była ciągła i równomiernie narysowana.

calculate_area()

Funkcja pomocnicza obliczająca pole trójkąta na podstawie współrzędnych jego wierzchołków. Wykorzystuje wzór na iloczyn wektorowy w przestrzeni 2D, który daje dwukrotność pola trójkąta (dlatego wynik jest dzielony przez 2).

is_point_in_triangle()

Funkcja sprawdzająca, czy dany punkt leży wewnątrz trójkąta. Wykorzystuje metodę opartą na współrzędnych barycentrycznych i porównywaniu pól trójkątów. Jeśli suma pól trzech trójkątów utworzonych przez punkt testowy i dwa wierzchołki trójkąta oryginalnego jest równa polu całego trójkąta, oznacza to, że punkt leży wewnątrz trójkąta.

draw_filled_triangle()

Funkcja rysująca wypełniony trójkąt. Wykorzystuje prostokątny obszar ograniczający (bounding box) wokół trójkąta, aby zmniejszyć liczbę testowanych punktów. Dla każdego punktu w tym obszarze sprawdza, czy leży on wewnątrz trójkąta, i jeśli tak, rysuje go.

draw_triangle_outline()

Funkcja rysująca kontur trójkąta. Wykorzystuje funkcję draw_line_bresenham do narysowania trzech linii łączących wierzchołki trójkąta.

save_image()

Zapisuje aktualny stan płótna jako plik graficzny, wykorzystując bibliotekę PIL (Python Imaging Library). Pozwala to na zachowanie wyników rasteryzacji do późniejszego użytku.

display()

Wyświetla aktualny stan płótna za pomocą biblioteki matplotlib. Jest to przydatne do natychmiastowej wizualizacji wyników bez konieczności zapisywania i otwierania pliku graficznego.

main()

Przykładowa funkcja demonstrująca użycie klasy Rasterizer. Tworzy obiekt płótna, rysuje na nim dwie linie w różnych kolorach, kontur trójkąta oraz wypełniony trójkąt. Następnie wyświetla rezultat i opcjonalnie zapisuje go do pliku.

Zadanie 4

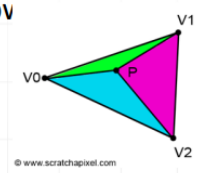
Zadanie 4.

Wprowadzić interpolację koloru w rysowanej linii i trójkącie.
Wskazówki:

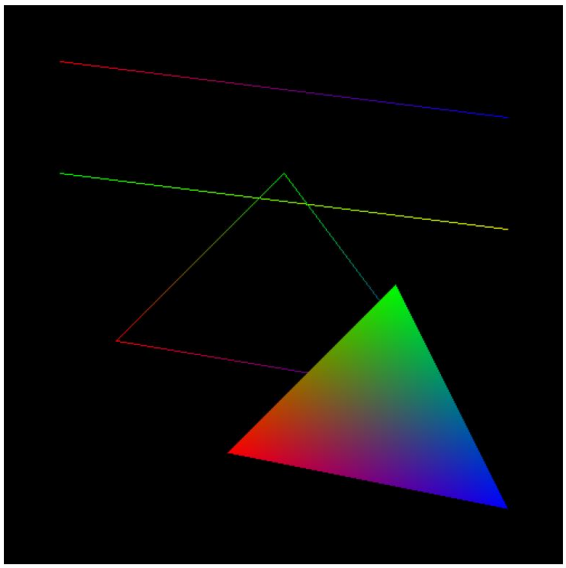
- zakładamy, że kolor jest podany z każdym wierzchołkiem (jak położenie),
- w przypadku linii można wykorzystać interpolację liniową koloru,
  $\vec{C} = \vec{A} + t \cdot (\vec{B} - \vec{A})$, dla $t \in [0; 1]$,
  t określa wtedy postęp w rysowaniu linii (0 = początek, 1 = koniec),

- w przypadku trójkąta wykorzystujemy ważenie względem pól trójkątów

$$\vec{C}_P = \frac{\lambda_0}{\lambda} \cdot \vec{V0} + \frac{\lambda_1}{\lambda} \cdot \vec{V1} + \frac{\lambda_2}{\lambda} \cdot \vec{V2},$$
$$\lambda = Area(V0, V1, V2),$$
$$\lambda_0 = Area(V0, V1, P),\ \lambda_1 = Area(V1, V2, P),\ \lambda_2 = Area(V2, V0, P).$$



Wyniki działania i analiza



Główna różnica między zadaniem 3 a zadaniem 4 polega na sposobie przypisywania kolorów do pikseli. W zadaniu 3 wszystkie piksele danego obiektu (linii lub trójkąta) miały ten sam kolor, natomiast w zadaniu 4 kolor każdego piksela jest płynnie interpolowany między kolorami wierzchołków.

Zmiany w algorytmach:

Dla linii:

- Zadanie 3: Każdy piksel linii ma jednolity kolor określony przy tworzeniu linii.
- Zadanie 4: Kolor każdego piksela jest interpolowany liniowo między kolorem początkowym (A) a końcowym (B) według wzoru:

$$C = A + t \cdot (B - A), \text{ dla } t \in [0; 1]$$

gdzie t określa postęp w rysowaniu linii (0 = początek, 1 = koniec).

Dla trójkątów:

- Zadanie 3: Wszystkie piksele wewnątrz trójkąta miały jednolity kolor.
- Zadanie 4: Kolor każdego piksela P wewnątrz trójkąta jest interpolowany przy użyciu współrzędnych barycentrycznych, które są obliczane na podstawie stosunku pól trójkątów:

$$Cp = (\lambda 0/\lambda) \cdot V0 + (\lambda 1/\lambda) \cdot V1 + (\lambda 2/\lambda) \cdot V2$$

gdzie:

- λ = Area(V0, V1, V2) (całkowite pole trójkąta)
- $\lambda 0$ = Area(V1, V2, P)
- $\lambda 1$ = Area(V2, V0, P)

- $\lambda 2$ = Area(V0, V1, P)

Z obrazu wynikają następne wyniki:

Linie z interpolacją kolorów:

- Górna linia płynnie przechodzi z koloru czerwonego w niebieski.
- Środkowa linia przechodzi z koloru zielonego w żółty.
- Linie tworzące obrys trójkąta również mają interpolowane kolory między wierzchołkami (czerwony, zielony i niebieski).

Wypełniony trójkąt z interpolacją kolorów:

- Trójkąt na dole obrazu ma płynnie interpolowane kolory między wierzchołkami w kolorach czerwonym, zielonym i niebieskim.
- Widoczny jest efekt mieszania kolorów - w środku trójkąta kolory płynnie się przenikają, tworząc całą gamę przejściowych odcieni.

Znaczenie interpolacji kolorów

Interpolacja kolorów jest kluczowym elementem nowoczesnej grafiki komputerowej, który pozwala na:

1. Płynne przejścia tonalne - dzięki interpolacji można uzyskać naturalne przejścia kolorów bez widocznych granic między różnymi odcieniami.
2. Cieniowanie (shading) - w grafice 3D interpolacja kolorów stanowi podstawę dla bardziej zaawansowanych modeli cieniowania, takich jak cieniowanie Gourauda (interpolacja kolorów wierzchołków) czy cieniowanie Phonga (interpolacja normalnych wierzchołków).
3. Realizm wizualny - naturalne obiekty rzadko mają jednolity kolor, a interpolacja pozwala na symulowanie subtelnych zmian w kolorze, które występują w rzeczywistym świecie.

Porównanie efektów wizualnych

Porównując obrazy z zadania 3 i zadania 4:

1. **Zadanie 3** (jednolite kolory):
    - Jednolity kolor linii i trójkątów daje płaski, "kreskówkowy" wygląd.
    - Granice między różnymi obiektami są ostre i wyraźne.
    - Obraz wygląda bardziej schematyczny i uproszczony.
2. **Zadanie 4** (interpolacja kolorów):
    - Płynne przejścia kolorów tworzą efekt bardziej realistyczny i estetyczny.
    - Obraz ma większą głębię i wymiar wizualny.

- Widoczne są subtelne przejścia kolorystyczne, które nadają obrazowi większą dynamikę.

Interpolacja kolorów jest niezbędną techniką w nowoczesnej grafice komputerowej, która znajduje zastosowanie nie tylko w grach wideo i animacjach, ale również w wizualizacjach naukowych, projektowaniu graficznym i innych dziedzinach, gdzie ważna jest jakość wizualna prezentowanych danych.

Opis działania kodu

__init__()

Funkcja inicjalizująca klasę Rasterizer, która tworzy czarny obszar (płótno) o określonej szerokości i wysokości. Płótno jest zaimplementowane jako trójwymiarowa tablica NumPy, gdzie dwa pierwsze wymiary odpowiadają pikselom obrazu, a trzeci wymiar reprezentuje trzy kanały koloru (RGB).

clear()

Funkcja czyszcząca całe płótno, ustawiając wszystkie piksele na określony kolor (domyślnie czarny). Jest to przydatne, gdy chcemy zacząć rysowanie od nowa bez tworzenia nowego obiektu.

draw_point()

Funkcja do rysowania pojedynczego piksela na płótnie. Sprawdza, czy współrzędne mieszczą się w granicach płótna, konwertuje współrzędne na liczby całkowite oraz zapewnia, że kolor jest tablicą uint8 przed ustawieniem go w odpowiednim miejscu na płótnie.

interpolate_color()

Nowa funkcja, która realizuje liniową interpolację między dwoma kolorami. Wykorzystuje formułę $C = A + t \cdot (B - A)$, gdzie t jest parametrem z zakresu [0; 1] określającym pozycję na linii łączącej kolory. Gdy $t = 0$, funkcja zwraca kolor początkowy; gdy $t = 1$, zwraca kolor końcowy.

draw_line_with_color_interpolation()

Rozszerzona funkcja rysowania linii, która oprócz algorytmu Bresenhama do określania pozycji pikseli, wykorzystuje interpolację kolorów. Dla każdego narysowanego piksela oblicza parametr t na podstawie postępu wzdłuż linii, a następnie używa funkcji interpolate_color do określenia odpowiedniego koloru pośredniego. Pozwala to na płynne przejścia kolorów wzdłuż linii.

calculate_area()

Funkcja pomocnicza obliczająca pole trójkąta na podstawie współrzędnych jego wierzchołków, wykorzystując wzór na iloczyn wektorowy w przestrzeni 2D.

is_point_in_triangle()

Funkcja sprawdzająca, czy dany punkt leży wewnątrz trójkąta. Wykorzystuje metodę współrzędnych barycentrycznych, porównując sumę pól trzech podtrójkątów z polem całego trójkąta. Dodaje również obsługę przypadków zdegenerowanych (trójkąty o zerowym polu).

draw_filled_triangle_with_color_interpolation()

Rozszerzona funkcja rysująca wypełniony trójkąt z interpolacją kolorów. Wykorzystuje współrzędne barycentryczne ($\lambda 0$, $\lambda 1$, $\lambda 2$) nie tylko do określenia, czy punkt leży wewnątrz trójkąta, ale również do obliczenia jego koloru. Kolor każdego piksela wewnątrz trójkąta jest ważoną sumą kolorów

wierzchołków, zgodnie z formułą: Cp = (λ0/λ)·V0 + (λ1/λ)·V1 + (λ2/λ)·V2, gdzie wagi są znormalizowanymi współrzędnymi barycentrycznymi.

draw_triangle_outline_with_color_interpolation()

Funkcja rysująca kontur trójkąta z interpolacją kolorów. Wykorzystuje draw_line_with_color_interpolation do narysowania trzech linii z płynnymi przejściami kolorów między wierzchołkami.

save_image() i display()

Funkcje do zapisywania i wyświetlania aktualnego stanu płótna, identyczne jak w poprzedniej wersji.

main()

Przykładowa funkcja demonstrująca użycie klasy Rasterizer z interpolacją kolorów. Tworzy obraz zawierający linię z przejściem od czerwonego do niebieskiego, linię z przejściem od zielonego do żółtego, kontur trójkąta z kolorami RGB w wierzchołkach oraz wypełniony trójkąt z płynnym przejściem między kolorami RGB.

Zadanie 5

Zadanie 5.

Dodać wygładzanie krawędzi w generowanym obrazie.
Wskazówki:

- doskonale sprawdzi się **SSAA** – ang. *Super Sampling Anti-Aliasing*,
- należy wygenerować obraz roboczy w wyższej rozdzielczości,
  wystarczy wysokość, szerokość i wszystkie współrzędne przemnożyć razy 2,
  na tym etapie nie powinno być potrzeby modyfikacji funkcji rysujących,
  zmiana powinna wiązać się jedynie z wywołaniami funkcji rysujących,
  warto w tym miejscu wprowadzić dodatkową zmienną, np. scale = 2,
- obraz wynikowy uzyskać przez przeskalowanie w dół obrazu roboczego,
  1 piksel obrazu wynikowego to 4 uśrednione piksele obrazu roboczego.

SSAA (Super Sampling Anti-Aliasing) to jedna z najstarszych i koncepcyjnie najprostszych technik wygładzania krawędzi (antyaliasingu) w grafice komputerowej. Technika ta została opracowana, aby rozwiązać problem aliasingu, czyli efektu "schodkowania" (jagged edges) widocznego na krawędziach obiektów w obrazach rastrowych.

Zasada działania SSAA

Podstawowa idea SSAA polega na:

1. Renderowaniu obrazu w wyższej rozdzielczości niż docelowa (najczęściej 2×, 4× lub nawet 8× większej).
2. Próbkowaniu (samplingu) większej liczby punktów dla każdego piksela obrazu wynikowego.

3. Zmniejszeniu rozdzielczości poprzez uśrednienie wartości kolorów z sąsiednich pikseli obrazu roboczego.

Na przykład, przy SSAA 2× każdy wymiar obrazu jest podwajany, co oznacza, że obraz roboczy ma 4 razy więcej pikseli niż obraz wynikowy. Następnie każdy piksel obrazu wynikowego jest obliczany jako średnia z 4 odpowiadających mu pikseli obrazu roboczego.

Zalety SSAA

1. Wysoka jakość wygładzania - SSAA oferuje jedne z najlepszych rezultatów wśród technik antyaliasingu, szczególnie dla statycznych obrazów.
2. Koncepcyjna prostota - algorytm jest łatwy do zrozumienia i zaimplementowania.
3. Wygładzanie nie tylko krawędzi - SSAA poprawia jakość całego obrazu, w tym tekstur i cieni.

Wady SSAA

1. Wysoki koszt obliczeniowy - renderowanie obrazu w wyższej rozdzielczości wymaga znacznie większej mocy obliczeniowej.
2. Duże zapotrzebowanie na pamięć - SSAA 2× wymaga 4 razy więcej pamięci, a SSAA 4× aż 16 razy więcej.
3. Niska wydajność - ze względu na koszty obliczeniowe, SSAA nie jest optymalny do zastosowań czasu rzeczywistego.

Zastosowania SSAA

SSAA znajduje zastosowanie głównie w:

1. Renderingu offline - tworzenie wysokiej jakości grafiki, gdzie czas renderingu nie jest krytyczny.
2. Grach komputerowych dla wysokiej klasy sprzętu - niektóre gry oferują opcję SSAA dla graczy z mocnymi komputerami.
3. Wizualizacjach architektonicznych - gdzie jakość obrazu jest ważniejsza niż szybkość renderowania.
4. Grafice do druku - gdzie wymagana jest najwyższa jakość obrazu.

Alternatywne techniki antyaliasingu

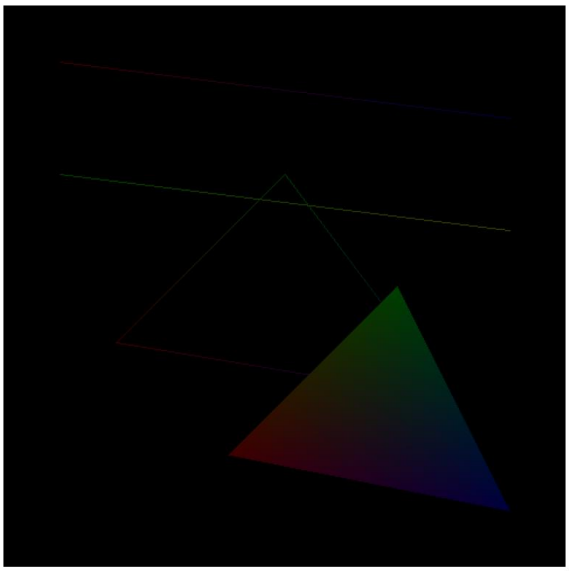
Z powodu wysokich wymagań SSAA, opracowano wiele alternatywnych technik antyaliasingu:

1. MSAA (Multisample Anti-Aliasing) - używa wielu próbek tylko dla krawędzi obiektów, a nie całego obrazu.
2. FXAA (Fast Approximate Anti-Aliasing) - szybki antyaliasing oparty na przetwarzaniu końcowego obrazu.
3. TXAA (Temporal Anti-Aliasing) - wykorzystuje informacje z poprzednich klatek do wygładzania.

4. DLSS (Deep Learning Super Sampling) - wykorzystuje sztuczną inteligencję do upskalowania obrazu.

Wyniki działania

Program ma 2 opcje: wykonanie SSAA na liniach i trójkącie z poprzedniego zadania lub na obrazku dostarczonym przez użytkownika. Poniżej znajdują się wyniki dla linii i trójkątów oraz porównanie obrazków z zadań 4 i 5.



Różnice w jakości krawędzi

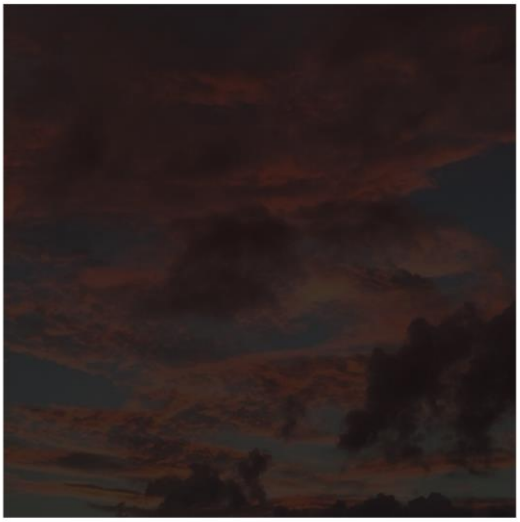
1. Zadanie 4 (bez SSAA):
    - Krawędzie są ostrzejsze i wyraźniej "schodkowane" (efekt aliasingu)
    - Widoczne są pojedyncze piksele, szczególnie na ukośnych liniach
    - Przejścia kolorów są bardziej "skokowe", zwłaszcza w miejscach, gdzie gradient zmienia się szybko
2. Zadanie 5 (z SSAA):
    - Krawędzie są znacznie bardziej wygładzone
    - Efekt "schodkowania" jest prawie niewidoczny
    - Przejścia kolorów są płynniejsze i bardziej subtelne
    - Obraz wygląda na "miększy", bardziej naturalny

Wpływ SSAA na jakość obrazu

W zadaniu 5 zastosowano SSAA (Super Sampling Anti-Aliasing) z mnożnikiem 2, co oznacza, że:

- Obraz był początkowo renderowany w rozdzielczości 4 razy większej (2x w każdym wymiarze)
- Każdy piksel w obrazie końcowym jest wynikiem uśrednienia 4 pikseli z obrazu o wyższej rozdzielczości
- Dzięki temu krawędzie, które w zadaniu 4 były "ostre" i schodkowe, w zadaniu 5 są znacznie bardziej wygładzone

Następnie użyłam SSAA na zdjęciu nieba z zadań 1 i 2:



Po zastosowaniu SSAA obraz wygląda znacznie ciemniejszy niż oryginał. Przyczyny tego zjawiska mogą być następujące:

1. Proces uśredniania kolorów:
    - W procesie SSAA kolory są uśredniane, co może prowadzić do zmniejszenia ich intensywności
    - Jeśli w oryginalnym obrazie występowały jasne piksele otoczone ciemniejszymi, po uśrednieniu będą one ciemniejsze
2. Brak korekcji jasności po uśrednianiu:
    - Algorytm SSAA w implementacji nie zawiera korekcji jasności po procesie uśredniania
    - Funkcja create_output_image() po prostu uśrednia wartości kolorów pikseli bez ich normalizacji
3. Efekt gamma:
    - Przy prostym uśrednianiu kolorów w przestrzeni RGB nie uwzględnia się nieliniowości percepcji jasności (efekt gamma)

- Poprawna implementacja powinna uwzględniać korekcję gamma przed uśrednianiem i po nim

Jest to znany efekt uboczny prostych implementacji SSAA, gdzie obraz po anti-aliasingu może wydawać się ciemniejszy niż oryginał. W profesjonalnych aplikacjach graficznych stosuje się dodatkowe techniki, takie jak korekcja gamma i normalizacja jasności, aby zachować wizualną zgodność obrazu po antyaliasingu z obrazem oryginalnym.

Podsumowanie:

Zadanie 5 demonstruje skuteczność techniki SSAA w poprawie jakości wizualnej obrazów rastrowych poprzez wygładzanie krawędzi i zapewnienie płynniejszych przejść kolorów. Jednakże, prosty algorytm SSAA użyty w implementacji ma pewne ograniczenia, takie jak przyciemnianie obrazu, co jest widoczne w przypadku zdjęcia nieba.

Warto zauważyć, że nowoczesne aplikacje graficzne rzadko używają czystego SSAA ze względu na jego wysokie wymagania obliczeniowe, a zamiast tego stosują bardziej zaawansowane i wydajne techniki antyaliasingu, takie jak MSAA, FXAA czy TAA, które oferują dobry kompromis między jakością a wydajnością.

Opis działania kodu

init()

Funkcja inicjalizująca, która tworzy płótno o zwiększonej rozdzielczości w zależności od parametru scale. Przechowuje zarówno docelowe wymiary jak i wymiary roboczego płótna o wyższej rozdzielczości.

clear()

Funkcja czyszcząca płótno, ustawiając wszystkie piksele na określony kolor (domyślnie czarny).

draw_point()

Funkcja do rysowania pojedynczego piksela na płótnie, z automatycznym skalowaniem współrzędnych do rozdzielczości wewnętrznego bufora.

interpolate_color()

Funkcja realizująca liniową interpolację między dwoma kolorami, wykorzystując formułę $C = A + t \cdot (B - A)$, gdzie t jest parametrem z zakresu [0; 1].

draw_line_with_color_interpolation()

Funkcja rysująca linię z interpolacją koloru, wykorzystująca algorytm Bresenhama do wyznaczania pozycji pikseli oraz interpolację liniową do obliczania ich kolorów.

calculate_area()

Funkcja pomocnicza obliczająca pole trójkąta na podstawie współrzędnych jego wierzchołków.

draw_filled_triangle_with_color_interpolation()

Funkcja rysująca wypełniony trójkąt z interpolacją kolorów, wykorzystująca współrzędne barycentryczne do określenia czy punkt leży wewnątrz trójkąta oraz do obliczenia jego koloru.

draw_triangle_outline_with_color_interpolation()

Funkcja rysująca kontur trójkąta z interpolacją kolorów, wykorzystująca funkcję rysowania linii z gradientem koloru.

create_output_image()

Kluczowa funkcja dla antyaliasingu, która tworzy obraz wynikowy o niższej rozdzielczości poprzez uśrednianie grup pikseli z obrazu roboczego. Dla każdego piksela obrazu wyjściowego oblicza średnią kolorów z odpowiadającego mu kwadratu pikseli z obrazu roboczego.

save_image()

Funkcja zapisująca antyaliasingowany obraz wynikowy do pliku.

display()

Funkcja wyświetlająca antyaliasingowany obraz wynikowy za pomocą matplotlib.

main()

Funkcja demonstracyjna oferująca dwie opcje: rysowanie linii i trójkątów z płynną interpolacją kolorów i antyaliasingiem lub wczytanie i przetworzenie obrazu z pliku z zastosowaniem antyaliasingu.