

Przetwarzanie Obrazów: Sprawozdanie

Damian Ubowski Maciej Tarach

Warszawa, 2019

Spis treści

1 Wstęp	5
1.1 Format obrazu	5
1.1.1 Struktura formatu	5
1.1.2 Przykładowa struktura IFF	6
1.1.3 Instrukcja obsługi programu	6
2 Operacje ujednoliciania obrazów	7
2.1 Ujednolicenie obrazów szarych geometryczne	7
2.2 Ujednolicenie obrazów szarych rozdzielczościowe	10
2.3 Ujednolicenie obrazów RGB geometryczne	13
2.4 Ujednolicenie obrazów RGB rozdzielczościowe	17
3 Operacje sumowania arytmetycznego obrazów szarych	21
3.1 Sumowanie (określonej) stałej z obrazem	22
3.2 Sumowanie dwóch obrazów	22
3.3 Mnożenie obrazu przez zadaną liczbę	22
3.4 Mnożenie obrazu przez inny obraz	22
3.5 Mieszanie obrazów z określonym współczynnikiem	22
3.6 Potęgowanie obrazu (z zadaną potęgą)	22
3.7 Dzielenie obrazu przez (zadaną) liczbę	22
3.8 Dzielenie obrazu przez przez inny obraz	22
3.9 Pierwiastkowanie obrazu	22
3.10 Logarytmowanie obrazu	22
4 Operacje sumowania arytmetycznego obrazów barwowych	23
4.1 Sumowanie (określonej) stałej z obrazem	24
4.2 Sumowanie dwóch obrazów	24
4.3 Mnożenie obrazu przez zadaną liczbę	24
4.4 Mnożenie obrazu przez inny obraz	24
4.5 Mieszanie obrazów z określonym współczynnikiem	24
4.6 Potęgowanie obrazu (z zadaną potęgą)	24

4.7	Dzielenie obrazu przez (zadaną) liczbę	24
4.8	Dzielenie obrazu przez przez inny obraz	24
4.9	Pierwiastkowanie obrazu	24
4.10	Logarytmowanie obrazu	24
5	Operacje geometryczne na obrazie	25
5.1	Przemieszczenie obrazu o zadany wektor	25
5.2	Jednorodne skalowanie obrazu	27
5.3	Niejednorodne skalowanie obrazu	29
5.4	Obracanie obrazu o dowolny kąt	30
5.5	Symetrie względem osi układu	32
5.6	Symetrie względem zadanej prostej	34
5.7	Wycinanie fragmentów obrazu	36
5.8	Kopiowanie fragmentów obrazów	37
6	Operacje na histogramie obrazu szarego	39
6.1	Obliczanie histogramu	39
6.2	Przemieszczanie histogramu	39
6.3	Rozciąganie histogramu	39
6.4	Progowanie lokalne	39
6.5	Progowanie globalne	39
7	Operacje na histogramie obrazu barwowego	41
7.1	Obliczanie histogramu	41
7.2	Przemieszczanie histogramu	41
7.3	Rozciąganie histogramu	41
7.4	Progowanie 1-progowe lokalne	41
7.5	Progowanie wielo-progowe lokalne	41
7.6	Progowanie 1-progowe globalne	41
7.7	Progowanie wielo-progowe globalne	41
8	Operacje morfologiczne na obrazach binarnych	43
8.1	Okrawanie (erozja)	43
8.2	Nakładanie (dylatacja)	43
8.3	Otwarcie	43
8.4	Zamknięcie	43
9	Operacje morfologiczne na obrazach szarych	45
9.1	Okrawanie (erozja)	45
9.2	Nakładanie (dylatacja)	45
9.3	Otwarcie	45

SPIS TREŚCI 5

9.4 Zamknięcie	45
10 Filtrowanie liniowe i nieliniowe	47
10.1 Filtr dolnoprzepustowy uśredniający	47
10.2 Filtr dolnoprzepustowy Gaussowski	47
10.3 Operator Roberts'a	47
10.4 Operator Prewitt'a	47
10.5 Operator Sobel'a	47
10.6 Filtr kompasowy	47
10.7 Gradient wektora kierunkowego	47
10.8 Filtr medianowy	47
10.9 Filtr maksymalny	47
10.10 Filtr minimalny	47
10.11 Filtr płaskorzeźbowy	47

Rozdział 1

Wstęp

1.1 Format obrazu

Wybranym przez nas formatem obrazów cyfrowych jest DjVu, który jest oparty na zaawansowanej metodzie segmentacji obrazu. Tworzenie pliku DjVu polega na rozdzieleniu dowolnie skomplikowanego obrazu na odrębne warstwy, a następnie poddaniu warst odrębnym optymalizacjom i kompresjom. Format ten stosuje ładowanie progresywne, kodowanie arytmetyczne, oraz kompresję stratną dzięki czemu przy minimalnej ilości przestrzeni dyskowej można delektować się obrazami i dokumentami w wysokiej jakości.

1.1.1 Struktura formatu

Pliki DjVu rozpoczynają się od swojej “Magic number” potwierdzającej rodzaj pliku i mającej wartość `0x41 0x54 0x26 0x54`. Następnie czerpiąc inspirację ze struktury IFF (**Interchange File Format**) plik dzieli się na kawałki (*ang. chunks*) zawierające interesujące nas cenne dane. Takie jak szerokość lub wysokość obrazu, dpi, informacje o kolorach, rozmieszczeniu pikseli, etc. Każdy kawałek składając się z ID typu, długości zawartości i samej zawartości tworzy zwarty format. Identyfikator typu określa rolę w jakiej przyjdzie służyć kawałkowi. Do dyspozycji ma ich całkiem sporo, ale uwzględniając najbardziej przydatne w naszym kontekście to ograniczymy liczbę do:

- * BGjp - warstwa tylna przechowywana przy użyciu kodowania JPEG.
- * BFjp - warstwa przednia w formacie JPEG.
- * INFO - opisuje wysokość, szerokość, rozdzielczość, wersję kodera, oraz flagi wskazujące na obrót obrazu.

1.1.2 Przykładowa struktura IFF

FORM:DJVU [14260]

INFO [10]

Sjbz [13133]

FG44 [181]

BG44 [935]

Powyższa struktura przedstawia dokument składający się z jednej strony, na co wskazuje *FORM:DJVU*, wraz z grafiką. Ten znacznik informuje, że mamy do czynienia z kontenerem o długości 14260 bajtów, który może zawierać inne kawałki dokumentu. Zgodnie z konwencją, po identyfikatorze typu i informacji o długości znajduje się zawartość kawałka. W tym wypadku jak i w każdym innym po *FORM:DJVU* powinno znaleźć się *INFO* z podstawowymi informacjami. Jeśli konwencji i wymagań specyfikacyjnych stało się zadość wtedy czas nastąpił na jakieś wizualne atrakcje takie jak *Sjbz*, czyli masce wyboru pomiędzy kolorami z warstwy przedniej (*FG44*) i tylnej (*BG44*).

1.1.3 Instrukcja obsługi programu

W celu uruchomienia kodu źródłowego będzie niezbędny:

- * [DjVuLibre](#) ($\geq 3.5.21$)
- * [Python](#) (≥ 2.6 lub $3.X$)
- * [Cython](#) (≥ 0.19 , lub ≥ 0.20 dla Python 3)
- * [pkg-config](#) (POSIX)

Rozdział 2

Operacje ujednolicania obrazów

Ujednolicanie obrazów oznacza sprowadzenie ich do wspólnego gruntu pod względem określonego parametru. W tym wypadku będziemy ujednolicać obrazy pod względem geometrycznym (ilości kolumn i wierszy pikseli) i następnie rozdzielczościowym (wypełnienia pikselami). Sekwencyjność tych operacji jak i one same nie są w stanie spowodować spadku jakości obrazu.

2.1 Ujednolicenie obrazów szarych geometryczne

Algorytm

Opis

Algorytm geometrycznego ujednolicenia obrazów ma za zadanie sprowadzić oba obrazy do tej samej liczby pikseli w każdym wierszu i każdej kolumnie.

Kroki

1. Porównaj szerokości i wysokości obu obrazów i wybierz największe.
2. Jeśli pierwszy lub drugi obraz mają szerokość lub wysokość mniejszą od największej dostępnej to:
 - (a) Utwórz czarne tło
 - (b) Przenieś z wyśrodkowaniem piksle na czarne tło
3. Jeśli żaden z warunków jest niespełniony to nie rób nic

Rysunek 2.1: Przed uruchomieniem algorytmu (od lewej): obraz 1 (1067x1067, 300dpi), obraz 2 (2133x2133, 300dpi)



Rysunek 2.2: Po uruchomieniu algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



Kod źródłowy algorytmu

```

def geometricGray(self):
    print('geometric gray unification start')
    width, height = self.firstDecoder.width, self.firstDecoder.
                    height
    if width < self.maxWidth or height < self.maxHeight:
        # Create black background
        firstResult = numpy.zeros((self.maxHeight, self.maxWidth),
                                  numpy.uint8)
        # Copy smaller image to bigger
        startWidthIndex = int(round((self.maxWidth - width) / 2))
        startHeightIndex = int(round((self.maxHeight - height) /
                                      2))
        pixelsBuffer = self.firstDecoder.getPixels()
        for h in range(0, height):
            for w in range(0, width):
                firstResult[h + startHeightIndex, w + startWidthIndex] =
                    pixelsBuffer[h, w]
        img = Image.fromarray(firstResult, mode='L')
        img.save('Resources/ggUnification_1.png')
        print('first image done')

    width, height = self.secondDecoder.width, self.
                    secondDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        # Create black background
        secondResult = numpy.zeros((self.maxHeight, self.maxWidth),
                                   numpy.uint8)
        # Copy smaller image to bigger
        startWidthIndex = int(round((self.maxWidth - width) / 2))
        startHeightIndex = int(round((self.maxHeight - height) /
                                      2))
        pixelsBuffer = self.secondDecoder.getPixels()
        for h in range(0, height):
            for w in range(0, width):
                secondResult[h + startHeightIndex, w + startWidthIndex] =
                    pixelsBuffer[h, w]
        img = Image.fromarray(secondResult, mode='L')
        img.save('Resources/ggUnification_2.png')
        print('second image done')
    print('geometric gray unification done')

```

2.2 Ujednolicenie obrazów szarych rozdzielczościowe

Algorytm

Opis

Po użyciu ujednolicenia geometrycznego można użyć ujednolicenia rozdzielczościowego, które przeskakuje obraz z mniejszej postaci do większej dzięki czemu nie zostanie nam czarna ramka wokół obrazu. Wynikiem będzie większy obraz niż początkowo bez czarnego obwodu wokół. Mniejszy obraz można przeskalać do większych wymiarów przenosząc wszystkie piksele z uwzględnieniem luk pomiędzy nimi i następnie użycia interpolacji do zamazania tych luk. Interpolacja działa na zasadzie pobierania wartości z okolicznych pikseli i wyciągania z nich średniej, która posłuży jako baza koloru dla nowego piksela.

Kroki

1. Ustalenie nowych wymiarów obrazu
2. Obliczenie odległości pomiędzy pikselami ($scaleFactoryH$, $scaleFactoryW$)
3. Naniesienie pikseli z mniejszego obrazu na większy z uwzględnieniem luk
4. Interpolacja

Kod źródłowy algorytmu

```
def rasterGray(self):
    print('raster gray unification start')
    self._scaleUpGray(self.firstDecoder, 'Resources/
                           rgUnification_1.png')
    print('first image done')
    self._scaleUpGray(self.secondDecoder, 'Resources/
                           rgUnification_2.png')
    print('second image done')
    print('raster gray unification done')

def _scaleUpGray(self, decoder, outputPath):
    width, height = decoder.width, decoder.height
```

2.2. UJEDNOLICENIE OBRAZÓW SZARYCH ROZDZIELCZOŚCIOWE13

Rysunek 2.3: Skutki braku interpolacji



Rysunek 2.4: Przed uruchomieniem algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



Rysunek 2.5: Po uruchomieniu algorytmu (od lewej): obraz 1 (2133x2133, 300dpi), obraz 2 (2133x2133, 300dpi)



```

scaleFactoryW = float(self.maxWidth) / width
scaleFactoryH = float(self.maxHeight) / height
if width < self.maxWidth or height < self.maxHeight:
    pixelsBuffer = decoder.getPixels()
    result = numpy.zeros((self.maxHeight, self.maxWidth),
                         numpy.uint8)

    # Fill values
    for h in range(height):
        for w in range(width):
            if w%2 == 0:
                result[int(scaleFactoryH * h), int(round(
                    scaleFactoryW * w)) + 1] =
                    pixelsBuffer[h, w]
            if w%2 == 1:
                result[int(round(scaleFactoryH * h)) + 1, int(
                    scaleFactoryW * w)] =
                    pixelsBuffer[h, w]

    # Interpolate
    self._interpolateGray(result)
    img = Image.fromarray(result, mode='L')
    img.save(outputPath)

def _interpolateGray(self, result):
    for h in range(self.maxHeight):
        for w in range(self.maxWidth):

```

```

value = 0
count = 0
if result[h, w] == 0:
    for hOff in range(-1, 2):
        for wOff in range(-1, 2):
            hSafe = h if ((h + hOff) > (self.maxLength - 2)) |
                ((h + hOff) < 0) else (h + hOff)
            wSafe = w if ((w + wOff) > (self.maxLength - 2)) |
                ((w + wOff) < 0) else (w + wOff)
            if result[hSafe, wSafe] != 0:
                value += result[hSafe, wSafe]
                count += 1
            result[h, w] = value / count

```

2.3 Ujednolicenie obrazów RGB geometryczne

Algorytm

Opis

Algorytm geometrycznego ujednolicenia obrazów ma za zadanie sprowadzić oba obrazy do tej samej liczby pikseli w każdym wierszu i każdej kolumnie. Różnica pomiędzy tym przypadkiem a szarym sprawia, że ważne jest użycie odpowiednich struktur danych w taki sposób aby każdy z kanałów RGB był w stanie się pomieścić. Niewątpliwie ważne jest struktura danych uwzględniała kolejność w jakim kolory są przechowywane, inaczej może dojść do sytuacji w której nie dostaniemy oczekiwanej rezultatu.

Kroki

1. Porównaj szerokości i wysokości obu obrazów i wybierz największe.
2. Jeśli pierwszy lub drugi obraz mają szerokość lub wysokość mniejszą od największej dostępnej to:
 - (a) Utwórz czarne tło
 - (b) Przenieś z wyśrodkowaniem piksele na czarne tło z uwzględnieniem każdego z kanałów RGB
3. Jeśli żaden z warunków jest niespełniony to nie rób nic

Rysunek 2.6: Przed uruchomieniem algorytmu (od lewej): obraz 1 (512x512, 300dpi), obraz 2 (1024x1024, 300dpi)



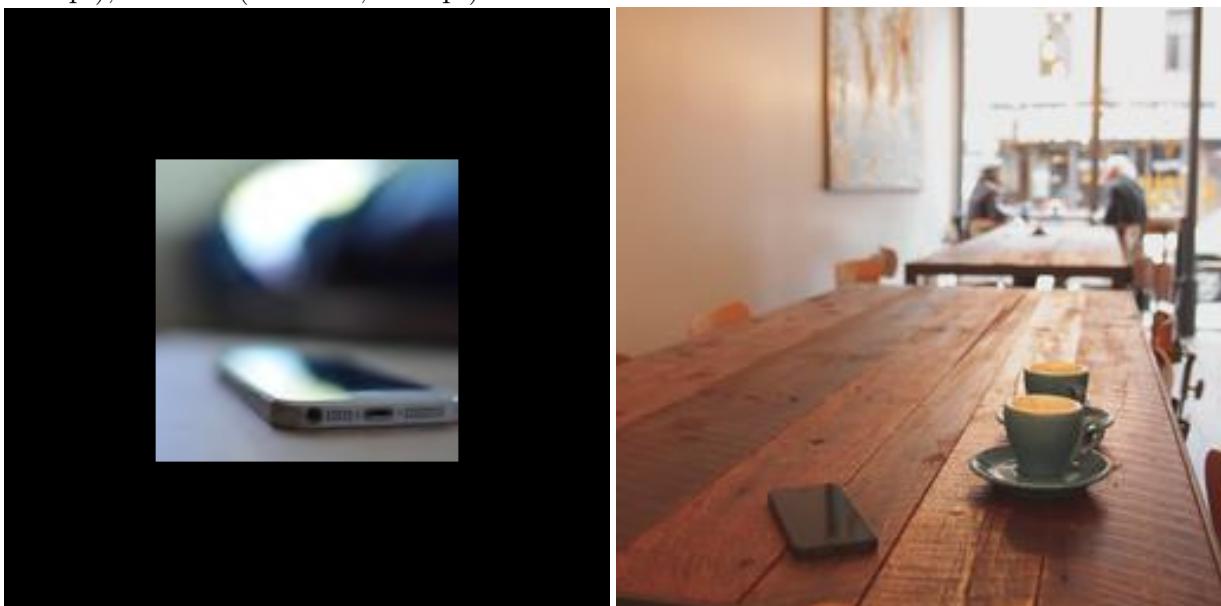
Rysunek 2.7: Po uruchomieniu algorytmu (od lewej): obraz 1 (1024x1024, 300dpi), obraz 2 (1024x1024, 300dpi)



Rysunek 2.8: Przed uruchomieniem algorytmu (od lewej): obraz 3 (126x126, 300dpi), obraz 4 (256x256, 300dpi)



Rysunek 2.9: Po uruchomieniu algorytmu (od lewej): obraz 3 (126x126, 300dpi), obraz 4 (256x256, 300dpi)



Kod źródłowy algorytmu

```

def geometricColor(self):
    print('geometric color unification start')
    self.firstDecoder.setColor()
    width, height = self.firstDecoder.width, self.firstDecoder.
                                height
    if width < self.maxWidth or height < self.maxHeight:
        result = self._paintInMiddleColor(self.firstDecoder)
        img = Image.fromarray(result, 'RGB')
        img.save('Resources/gcUnification_1.png')
        print('first image done')

    self.secondDecoder.setColor()
    width, height = self.secondDecoder.width, self.
                                secondDecoder.height
    if width < self.maxWidth or height < self.maxHeight:
        result = self._paintInMiddleColor(self.secondDecoder)
        img = Image.fromarray(result, 'RGB')
        img.save('Resources/gcUnification_2.png')
        print('second image done')
    print('geometric color unification done')

def _paintInMiddleColor(self, decoder):
    # Create black background
    result = numpy.full((self.maxHeight, self.maxWidth, 3), 0,
                        numpy.uint8)
    # Copy smaller image to bigger
    width, height = decoder.width, decoder.height
    startWidthIndex = int(round((self.maxWidth - width) / 2))
    startHeightIndex = int(round((self.maxHeight - height) / 2))
    pixelsBuffer = decoder.getPixels24Bits()
    for h in range(0, height):
        for w in range(0, width):
            result[h + startHeightIndex, w + startWidthIndex] =
                pixelsBuffer[h, w]
    return result

```

2.4 Ujednolicenie obrazów RGB rozdzielczościowe

Algorytm

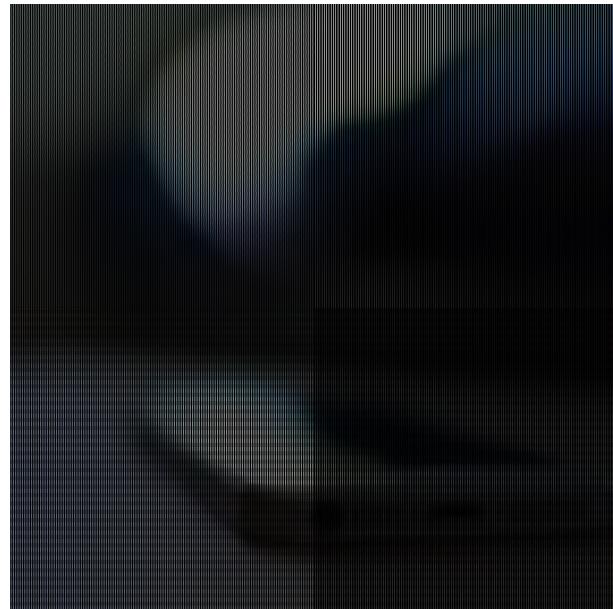
Opis

Po użyciu ujednolicenia geometrycznego można użyć ujednolicenia rozdzielczościowego, które przeskala obraz z mniejszej postaci do większej dzięki czemu nie zostanie nam czarna ramka wokół obrazu. Wynikiem będzie większy obraz niż początkowo bez czarnego obwodu wokół. Mniejszy obraz można przeskalać do większych wymiarów przenosząc wszystkie piksele z uwzględnieniem luk pomiędzy nimi i następnie użycia interpolacji do zamazania tych luk. Interpolacja działa na zasadzie pobierania wartości z okolicznych pikseli i wyciągania z nich średniej, która posłuży jako baza koloru dla nowego piksela.

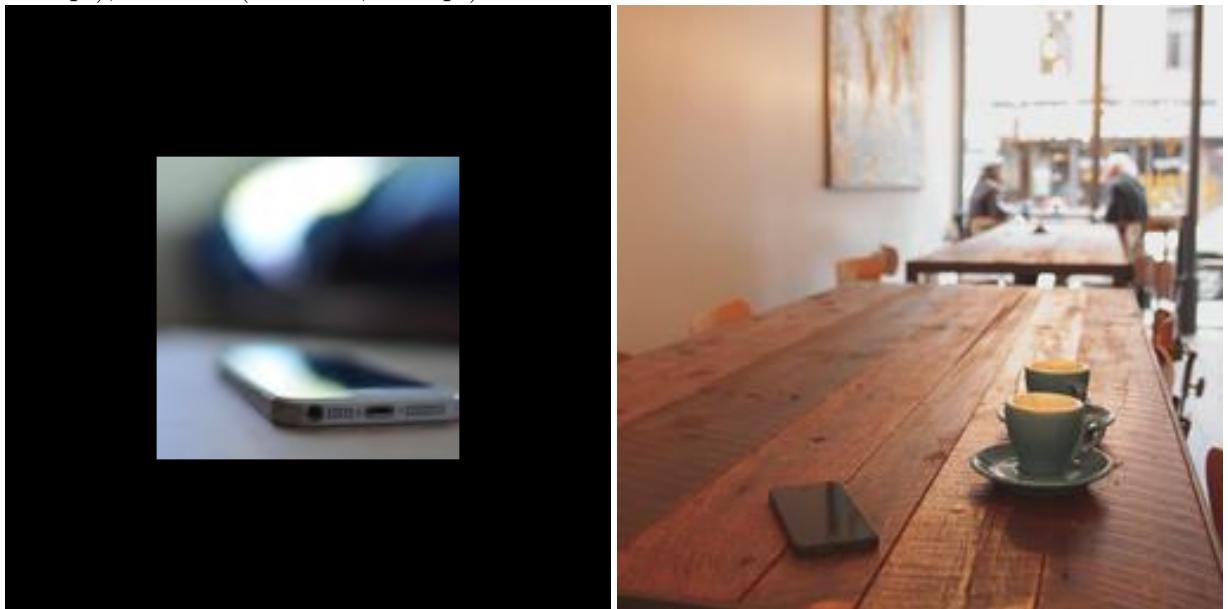
Kroki

1. Ustalenie nowych wymiarów obrazu
2. Obliczenie odległości pomiędzy pikselami ($scaleFactoryH$, $scaleFactoryW$)
3. Nanieśenie pikseli z mniejszego obrazu na większy z uwzględnieniem luk
4. Interpolacja

Rysunek 2.10: Skutki braku interpolacji



Rysunek 2.11: Przed uruchomieniem algorytmu (od lewej): obraz 1 (256x256, 300dpi), obraz 2 (256x256, 300dpi)



Rysunek 2.12: Po uruchomieniu algorytmu (od lewej): obraz 1 (256x256, 300dpi), obraz 2 (256x256, 300dpi)



Kod źródłowy algorytmu

```

def rasterColor(self):
    print('rastar color unification start')
    self.firstDecoder.setColor()
    self._scaleUpColor(self.firstDecoder, 'Resources/
                                         rcUnification_1.png')
    print('first image done')
    self.secondDecoder.setColor()
    self._scaleUpColor(self.secondDecoder, 'Resources/
                                         rcUnification_2.png')
    print('second image done')
    print('rastar color unification done')

def _scaleUpColor(self, decoder, outputPath):
    width, height = decoder.width, decoder.height
    scaleFactoryW = float(self.maxWidth) / width
    scaleFactoryH = float(self.maxHeight) / height
    if width < self.maxWidth or height < self.maxHeight:
        pixelsBuffer = decoder.getPixels24Bits()
        result = numpy.full((self.maxHeight, self.maxWidth, 3), 1
                           , numpy.uint8)
        # Fill values
        for h in range(height):
            for w in range(width):

```

```

if w%2 == 0:
    result[int(scaleFactoryH * h), int(round(
        scaleFactoryW * w)) + 1] =
        pixelsBuffer[h, w]
if w%2 == 1:
    result[int(round(scaleFactoryH * h)) + 1, int(
        scaleFactoryW * w)] =
        pixelsBuffer[h, w]
# Interpolate
self._interpolateColor(result)
img = Image.fromarray(result, mode='RGB')
img.save(outputPath)

def _interpolateColor(self, result):
    for h in range(self.maxHeight):
        for w in range(self.maxWidth):
            r, g, b = 0, 0, 0
            n = 0
            if (result[h, w][0] == 1) & (result[h, w][1] == 1) & (
                result[h, w][2] == 1):
                for hOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        hSafe = h if ((h + hOff) > (self.maxHeight - 2)) |
                            ((h + hOff) < 0) else (h +
                            hOff)
                        wSafe = w if ((w + wOff) > (self.maxWidth - 2)) |
                            ((w + wOff) < 0) else (w +
                            wOff)
                        if (result[hSafe, wSafe][0] > 1) | (result[hSafe,
                            wSafe][1] > 1) | (result[
                                hSafe, wSafe][2] > 1):
                            r += result[hSafe, wSafe][0]
                            g += result[hSafe, wSafe][1]
                            b += result[hSafe, wSafe][2]
                            n += 1
            result[h, w] = (r/n, g/n, b/n)

```


Rozdział 3

Operacje sumowania arytmetycznego obrazów szarych

- 3.1 Sumowanie (określonej) stałej z obrazem**
- 3.2 Sumowanie dwóch obrazów**
- 3.3 Mnożenie obrazu przez zadaną liczbę**
- 3.4 Mnożenie obrazu przez inny obraz**
- 3.5 Mieszanie obrazów z określonym współczynnikiem**
- 3.6 Potęgowanie obrazu (zadaną potęgą)**
- 3.7 Dzielenie obrazu przez (zadaną) liczbę**
- 3.8 Dzielenie obrazu przez inny obraz**
- 3.9 Pierwiastkowanie obrazu**
- 3.10 Logarytmowanie obrazu**

Rozdział 4

Operacje sumowania arytmetycznego obrazów barwowych

- 4.1 Sumowanie (określonej) stałej z obrazem
- 4.2 Sumowanie dwóch obrazów
- 4.3 Mnożenie obrazu przez zadaną liczbę
- 4.4 Mnożenie obrazu przez inny obraz
- 4.5 Mieszanie obrazów z określonym współczynnikiem
- 4.6 Potęgowanie obrazu (zadaną potęgą)
- 4.7 Dzielenie obrazu przez (zadaną) liczbę
- 4.8 Dzielenie obrazu przez inny obraz
- 4.9 Pierwiastkowanie obrazu
- 4.10 Logarytmowanie obrazu

Rozdział 5

Operacje geometryczne na obrazie

Operacje geometryczne przekształcają położenie pikseli (x_1, y_1) w obrazie wejściowym do nowej lokacji (x_2, y_2) w obrazie wynikowym. Dzięki temu możemy dopasować obraz do odpowiedniego układu współrzędnych lub użyć tych operacji do eliminacji geometrycznych zakłóceń obrazu (dystorsji).

5.1 Przemieszczenie obrazu o zadany wektor

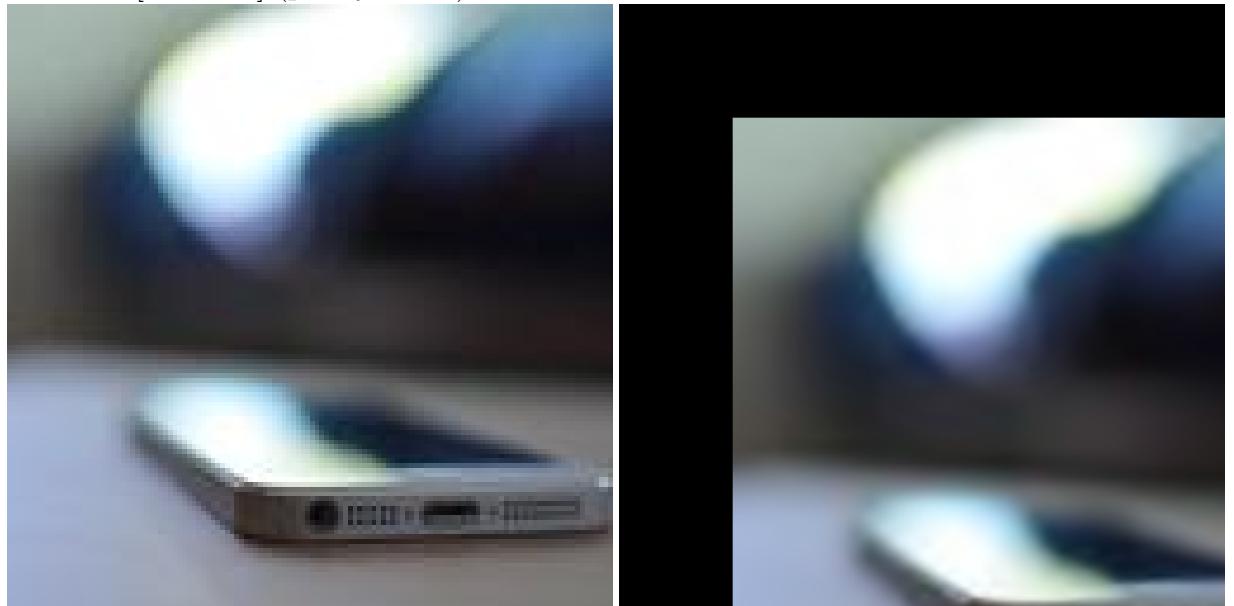
Opis

Operacja translacji wykonuje transformację geometryczną polegającą na przesieniu każdego z punktów obrazu wejściowego w nowe miejsce na obrazie wynikowym. Pod wpływem translacji element obrazu zlokalizowany na (x_1, y_1) zostanie przesunięty na nową pozycję (x_2, y_2) . Różnicą pomiędzy (x_1, y_1) i (x_2, y_2) jest wektor (bx, by) , który jest określony przez użytkownika. Operacja przemieszczenia przybiera postać:

$$x_2 = x_1 + b_x \quad (5.1)$$

$$y_2 = y_1 + b_y \quad (5.2)$$

Rysunek 5.1: Przed uruchomieniem algorytmu (lewy obraz), po przesunięciu o wektor $[100, 100]$ (prawy obraz)



Rysunek 5.2: Przed uruchomieniem algorytmu (lewy obraz), po przesunięciu o wektor $[100, -100]$ (prawy obraz)



Kod źródłowy algorytmu

```

def translate(self, deltaX = 0, deltaY = 0):
    print('translation start')
    height, width = self.decoder.height, self.decoder.width
    image = self.decoder.getPixels24Bits()
    result = numpy.zeros((height, width, 3), numpy.uint8)

    for y in range(height):
        for x in range(width):
            if 0 < y + deltaY < height and 0 < x + deltaX < width:
                result[y + deltaY][x + deltaX] = image[y][x]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/tGeometric.png')
    print('translation done')

```

5.2 Jednorodne skalowanie obrazu

Opis

Skalowanie jednorodne obrazu składa się na pomnożenie współrzędnych każdego piksela przez określoną wartość.

$$x_2 = S_x * x_1 \quad (5.3)$$

$$y_2 = S_y * y_1 \quad (5.4)$$

Przy czym skalowanie jednorodne oznacza, że po zmianie wartości współrzędnych nasz obraz zachowa dawne proporcje. Czyli:

$$S_x = S_y \quad (5.5)$$

Rysunek 5.3: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik 2 (prawy obraz)



Kod źródłowy algorytmu

```

def homogeneousScaling(self, scale = 1.0):
    print('homogeneous scaling start')
    image = self.decoder.getPixels24Bits()

    print('scaling')
    result = self._scaleXY(image, scale)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/hsGeometric.png')
    print('homogeneous scaling done')

def _scaleXY(self, matrix, scale):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.full((height, width, 3), 1, numpy.uint8)
    for y in range(height):
        for x in range(width):
            if scale * y < height and scale * x < width:
                result[int(scale * y)][int(scale * x)] = matrix[y][x]
    return result

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            r, g, b = 0, 0, 0
            n = 0
            if (result[h, w][0] == 1) & (result[h, w][1] == 1) & (
                    result[h, w][2] == 1):
                for hOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        hSafe = h if ((h + hOff) > (height - 2)) | ((h +
                            hOff) < 0) else (h + hOff)
                        wSafe = w if ((w + wOff) > (width - 2)) | ((w +
                            wOff) < 0) else (w + wOff)
                        if (result[hSafe, wSafe][0] > 1) | (result[hSafe,
                            wSafe][1] > 1) | (result[hSafe,
                            wSafe][2] > 1):
                            r += result[hSafe, wSafe][0]
                            g += result[hSafe, wSafe][1]
                            b += result[hSafe, wSafe][2]
                            n += 1
            result[h, w] = (r/n, g/n, b/n)

```

5.3 Niejednorodne skalowanie obrazu

Opis

Skalowanie niejednorodne obrazu składa się na pomnożenie współrzędnych każdego piksela przez określoną wartość.

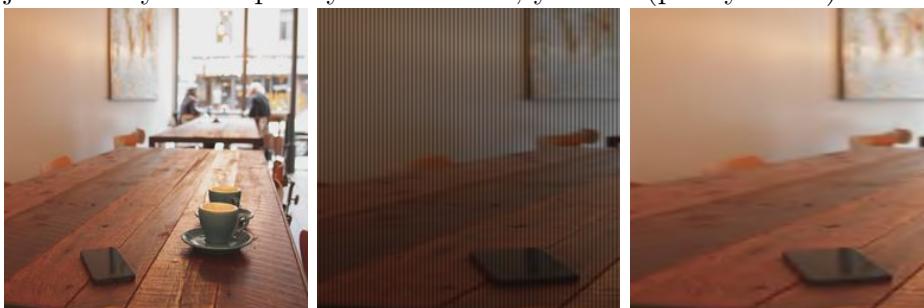
$$x_2 = S_x * x_1 \quad (5.6)$$

$$y_2 = S_y * y_1 \quad (5.7)$$

Przy czym skalowanie niejednorodne oznacza, że po zmianie wartości współrzędnych nasz obraz będzie miał zachowane proporcje. Czyli:

$$S_x \neq S_y \quad (5.8)$$

Rysunek 5.4: Przed uruchomieniem algorytmu (lewy obraz), po skalowaniu jednorodnym o współczynnik $x = 2.0$, $y = 1.0$ (prawy obraz)



Kod źródłowy algorytmu

```
def nonUniformScaling(self, scaleX = 1.0, scaleY = 1.0):
    print('non-uniform scaling start')
    image = self.decoder.getPixels24Bits()

    print('scaling')
    result = self._scale(image, scaleX, scaleY)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/nusGeometric.png')
    print('non-uniform scaling done')

def _scale(self, matrix, scaleX, scaleY):
```

```

height, width = self.decoder.height, self.decoder.width
result = numpy.full((height, width, 3), 1, numpy.uint8)
for y in range(height):
    for x in range(width):
        if scaleY * y < height and scaleX * x < width:
            result[int(scaleY * y)][int(scaleX * x)] = matrix[y][x]
return result

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            r, g, b = 0, 0, 0
            n = 0
            if (result[h, w][0] == 1) & (result[h, w][1] == 1) &
               (result[h, w][2] == 1):
                for hOff in range(-1, 2):
                    for wOff in range(-1, 2):
                        hSafe = h if ((h + hOff) > (height - 2)) | ((h
                                         + hOff) < 0) else (h + hOff)
                        wSafe = w if ((w + wOff) > (width - 2)) | ((w +
                                         wOff) < 0) else (w + wOff)
                        if (result[hSafe, wSafe][0] > 1) | (result[
                            hSafe, wSafe][1] > 1) | (result[
                            hSafe, wSafe][2] > 1):
                            r += result[hSafe, wSafe][0]
                            g += result[hSafe, wSafe][1]
                            b += result[hSafe, wSafe][2]
                            n += 1
            result[h, w] = (r/n, g/n, b/n)

```

5.4 Obracanie obrazu o dowolny kąt

Opis

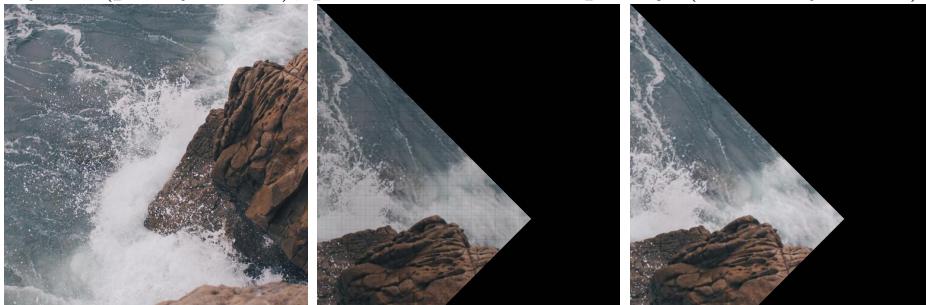
Operacja obrotu wykonywana jest wokół początku układu współrzędnych o kąt φ w taki sposób aby odległość od początku układu do punktu pozostała bez zmian, oraz aby pomiędzy odcinkami danych punków był kąt φ . Właściwości te można pozyskać dzięki wzorom:

$$x' = x \cos(\varphi) - y \sin(\varphi) \quad (5.9)$$

$$y' = x \sin(\varphi) + y \cos(\varphi) \quad (5.10)$$

gdzie (x', y') to nowe współrzędne wyznaczone po obrocie punktu (x, y) o kąt φ .

Rysunek 5.5: Przed uruchomieniem algorytmu (lewy obraz), po obróceniu o kąt 45° (prawy obraz), po obrocie bez interpolacji (środkowy obraz)



Kod źródłowy algorytmu

```

def rotation(self, phi):
    print('rotation start')
    image = self.decoder.getPixels24Bits()

    print('rotating')
    result = self._rotate(image, phi)
    print('interpolation')
    self._interpolateColor(result)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/rGeometric.png')
    print('rotation done')

def _rotate(self, image, phi):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.full((height, width, 3), 1, numpy.uint8)
    radian = math.radians(phi)
    for y in range(height):
        for x in range(width):
            newX = x * math.cos(radian) - y * math.sin(radian)
            newY = x * math.sin(radian) + y * math.cos(radian)
            if newY < height and newY >= 0 and newX >= 0 and newX <
                width:
                result[int(newY)][int(newX)] = image[y][x]
    return result

def _interpolateColor(self, result):
    height, width = self.decoder.height, self.decoder.width
    for h in range(height):
        for w in range(width):
            r, g, b = 0, 0, 0
            n = 0

```

```

if (result[h, w][0] == 1) & (result[h, w][1] == 1) & (
    result[h, w][2] == 1):
    for hOff in range(-1, 2):
        for wOff in range(-1, 2):
            hSafe = h if ((h + hOff) > (height - 2)) | ((h +
                hOff) < 0) else (h + hOff)
            wSafe = w if ((w + wOff) > (width - 2)) | ((w +
                wOff) < 0) else (w + wOff)
            if (result[hSafe, wSafe][0] > 0) | (result[hSafe,
                wSafe][1] > 0) | (result[
                    hSafe, wSafe][2] > 0):
                r += result[hSafe, wSafe][0]
                g += result[hSafe, wSafe][1]
                b += result[hSafe, wSafe][2]
                n += 1
            result[h, w] = (r/n, g/n, b/n)

```

5.5 Symetrie względem osi układu

Opis

Symetria osiowa względem osi OX lub OY sprawia, że punkt (x, y) zmienia się w $(x, -y)$ lub $(-x, y)$ w zależności czy symetria dotyczyła osi OX lub OY. W naszej pracy przyjmujemy, że lewa dolna krawędź obrazu znajduje się w punkcie $(0, 0)$.

Rysunek 5.6: Od lewej: przed uruchomieniem algorytmu, po symetrii wzgędem OX, po symetrii wzgędem OY, po symetrii wzgędem OX oraz OY



Kod źródłowy algorytmu

```
def axisSymmetry(self, ox, oy):
    print('axis symmetry start')
    image = self.decoder.getPixels24Bits()

    print('symmetry operation')
    result = self._symmetryOXorOY(image, ox, oy)

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-AxisSymmetry.png')
```

```

print('axis symmetry done')

def _symmetryOXorOY(self, image, ox, oy):
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width, 3), numpy.uint8)
    for y in range(height):
        for x in range(width):
            if ox and not oy:
                result[y][x] = image[y][(width-1)-x]
            elif not ox and oy:
                result[y][x] = image[(height-1)-y][x]
            elif ox and oy:
                result[y][x] = image[(height-1)-y][(width-1)-x]
    return result

```

5.6 Symetrie względem zadanej prostej

Opis

Przypadek podobny do poprzedniego, lecz tym razem użytkownik podaje wiersz lub kolumnę względem której będzie przebiegała oś symetrii.

Rysunek 5.7: Przed uruchomieniem algorytmu (lewy), po symetrii względem prostej $x=356$ (środkowy), po symetrii względem prostej $y=356$ (prawy)



Kod źródłowy algorytmu

```

def customSymmetryX(self, ox):
    print('custom axis symmetry X start')
    if not self._validateSymmetryAxisX(ox):
        return

    print('symmetry operation X')
    image = self.decoder.getPixels24Bits()

```

```
height, width = self.decoder.height, self.decoder.width
resultWidth = ox*2
result = numpy.zeros((height, resultWidth, 3), numpy.uint8)
for y in range(height):
    for x in range(ox):
        result[y][x] = image[y][x]
        result[y][resultWidth-1-x] = image[y][x]

img = Image.fromarray(result, mode='RGB')
img.save('Resources/Geometric-CustomSymmetryX.png')
print('custom axis symmetry X done')

def _validateSymmetryAxisX(self, ox):
    width = self.decoder.width
    if ox <= 0 or ox > width:
        return False
    return True

def customSymmetryY(self, oy):
    print('custom axis symmetry Y start')
    if not self._validateSymmetryAxisY(oy):
        return

    print('symmetry operation Y')
    image = self.decoder.getPixels24Bits()
    height, width = self.decoder.height, self.decoder.width
    resultHeight = oy*2
    result = numpy.zeros((resultHeight, width, 3), numpy.uint8)
    for y in range(oy):
        for x in range(width):
            result[y][x] = image[y][x]
            result[resultHeight-1-y][x] = image[y][x]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-CustomSymmetryY.png')
    print('custom axis symmetry Y done')

def _validateSymmetryAxisY(self, oy):
    height = self.decoder.height
    if oy <= 0 or oy > height:
        return False
    return True
```

5.7 Wycinanie fragmentów obrazu

Opis

Wycięcie części obrazu jest zaimplementowane za pomocą kopiowania pikseli do obrazu pomocniczego. Skopowane zostają tylko te piksele, które znajdują się wewnątrz podanego zakresu.

Rysunek 5.8: Przed uruchomieniem algorytmu (lewy), po wycięciu kwadratu od piksela $(50, 50)$ do $(300, 300)$ (prawy)



Kod źródłowy algorytmu

```
def crop(self, (x1, y1), (x2, y2)):
    print('cropping image start')
    image = self.decoder.getPixels24Bits()

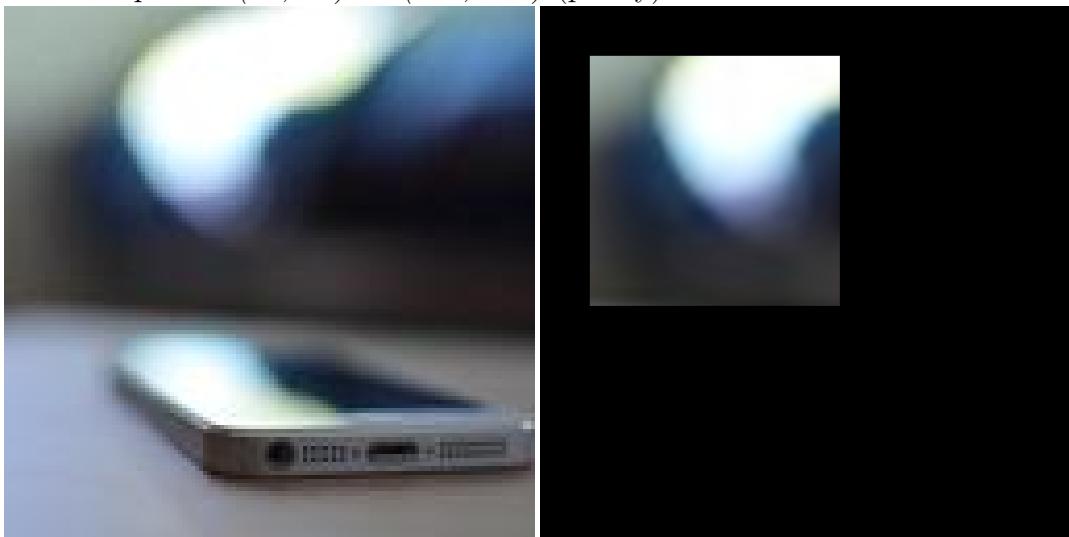
    print('cropping')
    for x in range(x1, x2+1):
        for y in range(y1, y2+1):
            image[y, x] = (0, 0, 0)

    img = Image.fromarray(image, mode='RGB')
    img.save('Resources/Geometric-Crop.png')
    print('cropping image done')
```

5.8 Kopiowanie fragmentów obrazów

Opis

Rysunek 5.9: Przed uruchomieniem algorytmu (lewy), po skopiowaniu kwadratu od piksela $(50, 50)$ do $(300, 300)$ (prawy)



Kod źródłowy algorytmu

```
def copy(self, (x1, y1), (x2, y2)):
    print('copying image start')
    image = self.decoder.getPixels24Bits()
    height, width = self.decoder.height, self.decoder.width
    result = numpy.zeros((height, width, 3), numpy.uint8)

    print('copying')
    for x in range(x1, x2+1):
        for y in range(y1, y2+1):
            result[y, x] = image[y, x]

    img = Image.fromarray(result, mode='RGB')
    img.save('Resources/Geometric-Copy.png')
    print('copying image done')
```


Rozdział 6

Operacje na histogramie obrazu szarego

- 6.1 Obliczanie histogramu
- 6.2 Przemieszczanie histogramu
- 6.3 Rozciąganie histogramu
- 6.4 Progowanie lokalne
- 6.5 Progowanie globalne

42 ROZDZIAŁ 6. OPERACJE NA HISTOGRAMIE OBRAZU SZAREGO

Rozdział 7

Operacje na histogramie obrazu barwowego

- 7.1 Obliczanie histogramu
- 7.2 Przemieszczanie histogramu
- 7.3 Rozciąganie histogramu
- 7.4 Progowanie 1-progowe lokalne
- 7.5 Progowanie wielo-progowe lokalne
- 7.6 Progowanie 1-progowe globalne
- 7.7 Progowanie wielo-progowe globalne

44 ROZDZIAŁ 7. OPERACJE NA HISTOGRAMIE OBRAZU BARWOWEGO

Rozdział 8

Operacje morfologiczne na obrazach binarnych

8.1 Okrawanie (erozja)

8.2 Nakładanie (dylatacja)

8.3 Otwarcie

8.4 Zamknięcie

46 ROZDZIAŁ 8. OPERACJE MORFOLOGICZNE NA OBRAZACH BINARNYCH

Rozdział 9

Operacje morfologiczne na obrazach szarych

9.1 Okrawanie (erozja)

9.2 Nakładanie (dylatacja)

9.3 Otwarcie

9.4 Zamknięcie

48 ROZDZIAŁ 9. OPERACJE MORFOLOGICZNE NA OBRAZACH SZARYCH

Rozdział 10

Filtrowanie liniowe i nieliniowe

- 10.1 Filtr dolnoprzepustowy uśredniający
- 10.2 Filtr dolnoprzepustowy Gaussowski
- 10.3 Operator Roberts'a
- 10.4 Operator Prewitt'a
- 10.5 Operator Sobel'a
- 10.6 Filtr kompasowy
- 10.7 Gradient wektora kierunkowego
- 10.8 Filtr medianowy
- 10.9 Filtr maksymalny
- 10.10 Filtr minimalny
- 10.11 Filtr płaskorzeźbowy