

Aplikacja okienkowa do przetwarzania obrazów

Jan Opala i Gabriela Majstrak

March 2025

Spis treści

1 Wstęp	3
2 Implementacja	3
2.1 Aspekty techniczne	3
2.2 Interfejs aplikacji	3
3 Podstawowa funkcjonalność	4
3.1 Wczytywanie obrazu	5
3.2 Dokonanie przekształcenia	5
3.3 Wyświetlanie obrazu	7
3.4 Wyświetlanie histogramu	9
3.4.1 Projekcje	9
3.4.2 Histogramy	10
3.5 Cofanie	14
3.6 Zapisywanie przekształconego obrazu	15
4 Metody przetwarzania obrazów	16
4.1 Podstawowe przekształcenia obrazów	16
4.1.1 negative	16
4.1.2 to_grayscale	17
4.1.3 adjust_brightness	18
4.1.4 adjust_contrast	19
4.1.5 vignette	20
4.1.6 add_noise	21
4.2 Filtry	23
4.2.1 Filtr uśredniający	24
4.2.2 Filtr gaussowski	24
4.2.3 Filtr wyostrzający	26
4.3 Wykrywanie krawędzi	27
4.3.1 roberts_cross	27
4.3.2 sobel_operator	29
5 Wnioski	30

1 Wstęp

Aplikacja okienkowa Image Processor służy do nakładania prostych przekształceń na wgrany obraz i do zapisania efektu swojej pracy jako plik .png. Dostępne przekształcenia obejmują proste operacje na zdjęciach (jasność, negatyw, odcienie szarości), filtry (gaussowski, uśredniający, wyostrzający), wykrywanie krawędzi (krzyż Roberts i operator Sobla), a także wprowadzanie szumu i winiety.

2 Implementacja

2.1 Aspekty techniczne

Aplikacja została napisana w języku Python w wersji 3.9.5 i składa się wyłącznie z jednego pliku, konkretnie `App.py`. W aplikacji wykorzystano następujące biblioteki:

- **CustomTkinter** – biblioteka służąca do zbudowania interfejsu
- **Tkinter** – podstawowa biblioteka tkinter do wczytywania plików z dysku
- **PIL** – biblioteka służąca do wyświetlania obrazu (oryginalnego i przetworzonego)
- **Matplotlib** – biblioteka służąca do sporządzania wykresów m.in. projekcji
- **NumPy** – potrzebna do operacji na macierzach, czyli wszystkich przekształceń na zdjęciach zaimplementowanych w programie

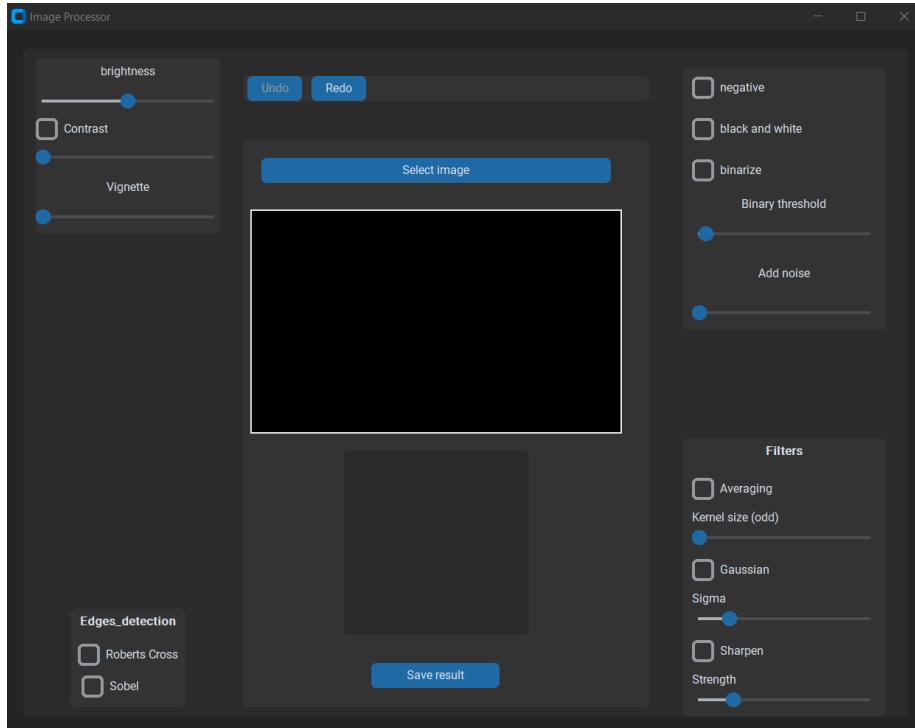
2.2 Interfejs aplikacji

Interfejs graficzny aplikacji zaimplementowany przy pomocy `customtkinter` składa się z następujących paneli:

- **SidePanel** – zawiera suwaki: jasność, kontrast (z checkboxem), winieta
- **Edges_Panel** – zawiera dwa checkboxy obejmujące dwie różne metody wykrywania pamięci (operator Sobela, krzyż Robertsa)
- **RightPanel** – na tym panelu znajdują się checkboxy: negatyw, skala szarości, binaryzacja oraz suwaki: próg binaryzacji i szum
- **FilterPanel** – zawiera checkboxy z filtrami (uśredniający, gaussowski, wyostrzajający) wraz z parametrem ustawianym suwakiem dla każdego z nich
- **MainPanel** – główny panel na którym znajdują się po bokach wszystkie pozostałe panele (układ utworzony przez grid). Na środku pojawiają się przyciski: 'undo' i 'redo' (wraz z zaimplementowanym cofaniem za pomocą struktury stosu), a poniżej obecny jest przycisk 'Select image' służący do

wyboru obrazu z dysku. Na środku ekranu znajduje się okienko do wyświetlania aktualnego przekształconego obrazu. Poniżej znajduje się przestrzeń do wyświetlenia wykresu (histogramu, dla binaryzacji są to projekcje) oraz przycisk 'Save result' służący do zapisu efektu pracy

Powyżej opisany interfejs można zobaczyć na zrzucie ekranu:



Rysunek 1: Wygląd interfejsu aplikacji po jej uruchomieniu

3 Podstawowa funkcjonalność

Opis podstawowej funkcjonalności aplikacji prezentuje się następująco:

1. Użytkownik uruchamia program
2. Użytkownik wczytuje zdjęcie do przetworzenia
3. Użytkownik wybiera przekształcenia jakie chce zastosować. W przypadku możliwości wyboru parametrów ustala konkretną wartość suwakiem. Niechciane zmiany wycofuje przyciskiem 'Undo'
4. Użytkownik zapisuje przekształcony obraz na dysku

Powyższe działanie zostało zaimplementowane w następujący sposób:

3.1 Wczytywanie obrazu

Po kliknięciu w przycisk 'Select image', który wykonuje metodę `load_image`.

```
self.select_btn = ctk.CTkButton(self.image_frame, text="Select image", command=self.load_image)
```

Powyższa metoda implementowana jest następująco:

```
def load_image(self):
    file_path = filedialog.askopenfilename(filetypes=[("Image files", "*.png;*.jpg;*.jpeg;*.bmp")])
    if file_path:
        self.original_image_array= np.array(Image.open(
            file_path))
        self.processed_image_array = self.
            original_image_array.copy()
        self.side_panel.brightness_slider.set(0)
        self.side_panel.contrast_slider.set(0)
        self.side_panel.vignette_slider.set(0)
        self.side_panel.contrast.deselect()
        self.Edges_Panel.sobel_checkbox.deselect()
        self.Edges_Panel.roberts_cross_checkbox.deselect()
        self.right_panel.binarize_checkbox.deselect()
        self.right_panel.black_and_white.deselect()
        self.right_panel.negative_checkbox.deselect()
        self.right_panel.threshold.set(0)
        self.filter_panel.avg_filter_checkbox.deselect()
        self.filter_panel.gaussian_filter_checkbox.
            deselect()
        self.filter_panel.sharpen_filter_checkbox.
            deselect()
        self.display_image()
        self.display_histogram()
    self.undo_stack.clear()
    self.redo_stack.clear()
    self.undo_btn.configure(state="disabled")
    self.redo_btn.configure(state="disabled")
```

Powyższa funkcja wykorzystuje metodę `filedialog` z biblioteki Tkinter i pozwala użytkownikowi wybrać ścieżkę do pliku będącego obrazem. Następnie plik konwertowany jest do struktury NumPy array oraz resetowane są wszystkie checkboxy, suwaki, a także stos cofania.

3.2 Dokonanie przekształcenia

Żeby dokonać przekształcenia użytkownik musi zaznaczyć jeden z kilku checkboxów lub poruszyć któryś ze sliderów. Wtedy wykonana zostaje metoda `self.update_image`. Przykładowo:

```
self.side_panel.brightness_slider.configure(command=self.  
    update
```

która to z kolei metoda jest o następującej treści:

```
def update_image(self, event=None):  
    if self.processed_image_array is not None:  
        self.undo_stack.append(self.  
            processed_image_array.copy())  
    if len(self.undo_stack) > 5:  
        self.undo_stack.pop(0)  
    self.undo_btn.configure(state="normal")  
    self.redo_stack.clear()  
    self.redo_btn.configure(state="disabled")  
  
    image = self.original_image_array.copy()  
    brightness = self.side_panel.brightness_slider.get()  
    image = self.adjust_brightness(image, brightness)  
    contrast = self.side_panel.contrast_slider.get()  
    vignette = self.side_panel.vignette_slider.get()  
    image = self.vignette(image, vignette)  
    noise = self.right_panel.add_noise_slider.get()  
    image = self.add_noise(image, noise)  
  
    if self.Edges_Panel.roberts_cross_checkbox.get():  
        image = self.roberts_cross(image)  
  
    if self.Edges_Panel.sobel_checkbox.get():  
        image = self.sobel_operator(image)  
  
    if self.side_panel.contrast.get():  
        image = self.adjust_contrast(image, contrast)  
  
    if self.right_panel.negative_checkbox.get():  
        image = self.negative(image)  
  
    if self.right_panel.black_and_white.get():  
        image = self.to_greyscale(image)  
  
    if self.right_panel.binarize_checkbox.get():  
        threshold = int(self.right_panel.threshold.get())  
        image = self.binarize(image, threshold)  
  
    if self.filter_panel.avg_filter_checkbox.get():  
        size = int(self.filter_panel.avg_kernel_slider.  
            get())  
        if size % 2 == 0: size += 1  
        image = self.apply_average_filter(image,  
            kernel_size=size)
```

```

if self.filter_panel.gaussian_filter_checkbox.get():
    sigma = self.filter_panel.gaussian_sigma_slider.
        get()
    size = 2 * int(3 * sigma) + 1
    image = self.apply_gaussian_filter(image, sigma=
        sigma, kernel_size=size)

if self.filter_panel.sharpen_filter_checkbox.get():
    strength = self.filter_panel.
        sharpen_strength_slider.get()
    image = self.apply_sharpen_filter(image,
        strength=strength)

self.processed_image_array = image
self.display_image()

if self.right_panel.binarize_checkbox.get():
    self.display_projection(self.
        processed_image_array)
else:
    self.display_histogram()

```

Logika powyższego kodu jest następująca:

1. Obecna wersja obrazu (przed zadanym przekształceniem) zostaje dodana do stosu cofania
2. Program duplikuje obraz by przekształcenia były wykonywane na kopii
3. Dla każdego z przekształceń zostaje sprawdzone czy użytkownik zaznaczył np. checkbox
4. Program wykonuje wszystkie zaznaczone przekształcenia (aktualizujemy zmienną `image` według schematu: `image = self.przekształcenie(image, parametr)`)
5. Image zostaje przypisane do zmiennej przechowującej przetworzony obraz w postaci NumPy array
6. Obraz zostaje wyświetlony wraz z histogramem lub projekcjami

3.3 Wyświetlanie obrazu

Program wyświetla obraz w ten sposób, że przyjmuje w postaci NumPy array obraz oryginalny lub przetworzony a następnie wyświetla go za pomocą metody:

```

def display_image(self):
    max_width, max_height = 500, 300
    img_height, img_width = self.processed_image_array.
        shape[:2]
    scale = min(max_width / img_width, max_height /
        img_height)
    img_width = int(img_width * scale)
    img_height = int(img_height * scale)

    img_array = self.processed_image_array
    if len(img_array.shape) == 2:
        img_array = np.stack((img_array,) * 3, axis=-1)
        # grayscale -> RGB

    try:
        img_pil = Image.fromarray(img_array.astype(np.
            uint8))
    except Exception as e:
        return

    resized_image = img_pil.resize((img_width,
        img_height))
    img_tk = ImageTk.PhotoImage(resized_image)

    self.canvas.delete("all")
    self.canvas.config(width=max_width, height=
        max_height)
    x_offset = max_width / 2
    y_offset = max_height / 2

    self.canvas.create_image(x_offset, y_offset, anchor=
        tk.CENTER, image=img_tk)
    self.canvas.image = img_tk

```

Wyświetlanie obrazu zachodzi następująco:

1. Jeśli obraz (w postaci NumPy array) jest w skali szarości (ma 2 kanały) to zostaje przekonwertowany na RGB
2. Przy pomocy biblioteki PIL NumPy array zostaje zamienione w obraz
3. Przestrzeń na wyświetlanie obrazu zostaje wyczyszczona (nie ma na niej obrazu, który był poprzednio)
4. Na wyżej wspomnianej przestrzeni pojawi się obraz odpowiednio przeskalowany

3.4 Wyświetlanie histogramu

Poniżej obszaru poświęconego na wyświetlanie obrazu znajduje się przestrzeń ns której generowane są następujące wykresy:

- Projekcje jeśli zaznaczona jest binaryzacja
- Histogram w przeciwnym przypadku

3.4.1 Projekcje

Projekcje to rodzaj wykresu, w którym to wyświetlany jest osobno histogram pionowy informujący o liczbie czarnych pikseli w konkretnej kolumnie oraz poziomy analogicznie opisujący obraz w wierszach. Po zastosowaniu operacji binaryzacji (więcej w późniejszych sekcjach dokumentacji) obraz zostaje przekonwertowany na macierz białych i czarnych pikseli (gdzie konkretna wartość progu decyduje o przyporządkowaniu danej oryginalnej wartości do jednej lub drugiej grupy). Projekcje wykonywane są wyłącznie kiedy aktywny jest checkbox zaznaczający binaryzację. Zaimplementowane zostały one w następujący sposób:

```

def display_projection(self, image):
    for widget in self.histogram_frame.winfo_children():
        widget.destroy()

    if len(image.shape) == 3:
        image = self.to_greyscale(image)

    threshold = int(self.right_panel.threshold.get())
    binary_image = np.where(image >= threshold, 1, 0)

    vertical_proj = np.sum(binary_image, axis=0)
    horizontal_proj = np.sum(binary_image, axis=1)

    fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(5, 3),
                                  gridspec_kw={'height_ratios': [1, 1]})
    ax1.plot(horizontal_proj)
    ax1.set_title("Horizontal projection")
    ax1.set_xlim(0, len(horizontal_proj))

    ax2.plot(vertical_proj)
    ax2.set_title("Vertical projection")
    ax2.set_xlim(0, len(vertical_proj))

    plt.tight_layout()

    canvas = FigureCanvasTkAgg(fig, master=self.
                               histogram_frame)
    canvas.draw()
    canvas.get_tk_widget().pack()

    plt.close(fig)

```

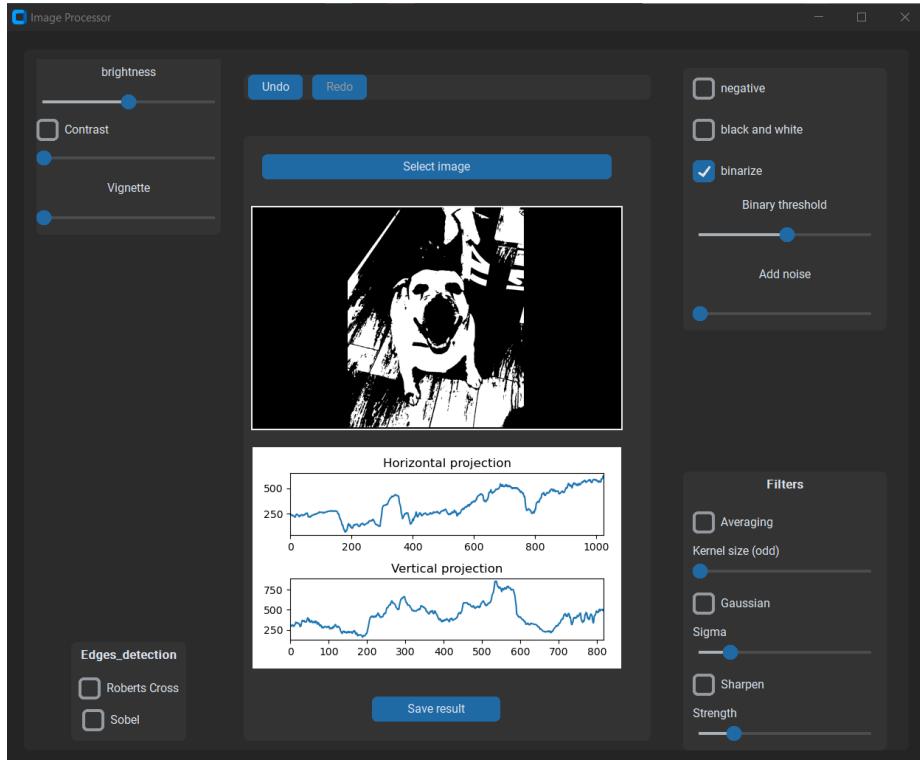
W powyższej metodzie:

1. Usuwane są wcześniej wygenerowane wykresy
2. Zastosowane jest przejście do skali szarości jeśli obraz jest RGB (liczba kanałów równa 3)
3. Próg binaryzacji jest wzięty od użytkownika. W oparciu o niego stosowana jest binaryzacja
4. W poziomie oraz pionie sumowane są liczności czarnych pikseli, a następnie są w oparciu o te dane generowane histogramy

Ta funkcjonalność na konkretnym przykładzie wygląda następująco:

3.4.2 Histogramy

Bez konieczności zaznaczania żadnego z przekształceń obrazu (dla oryginalnego zdjęcia), jak również po nanieśieniu przez użytkownika dowolnej operacji innej



Rysunek 2: Wygląd interfejsu aplikacji po zaznaczeniu binaryzacji. Na środku widać przetworzony obraz, a na dole widoczne są projekcje

niż binaryzacja, pod wyświetlonym obrazem pojawia się wykres histogramów. Działa w analogiczny sposób do projekcji, tyle że sporządzane są histogramy dla liczności pikseli dla każdego z trzech kanałów RGB. Implementacja metody prezentuje się następująco:

```
def display_histogram(self):
    for widget in self.histogram_frame.winfo_children():
        widget.destroy()

    if len(self.processed_image_array.shape) == 3:

        r_histogram, r_bins = np.histogram(self.
            processed_image_array[..., 0], bins=256,
            range=(0, 255))
        g_histogram, g_bins = np.histogram(self.
            processed_image_array[..., 1], bins=256,
            range=(0, 255))
        b_histogram, b_bins = np.histogram(self.
            processed_image_array[..., 2], bins=256,
```

```

        range=(0, 255))

fig, ax = plt.subplots(figsize=(5, 3))
ax.bar(r_bins[:-1], r_histogram, width=1, color=
       'red', alpha=0.5, label='Red')
ax.bar(g_bins[:-1], g_histogram, width=1, color=
       'green', alpha=0.5, label='Green')
ax.bar(b_bins[:-1], b_histogram, width=1, color=
       'blue', alpha=0.5, label='Blue')
ax.set_xlim(0,255)
ax.legend()
else:
    gray_histogram, bins = np.histogram(self.
                                         processed_image_array, bins=256, range
                                         =(0,255))
    fig, ax = plt.subplots(figsize=(5, 3))
    ax.bar(bins[:-1], gray_histogram, width=1, color=
           'black', label='Grey')
    ax.set_xlim(0,255)

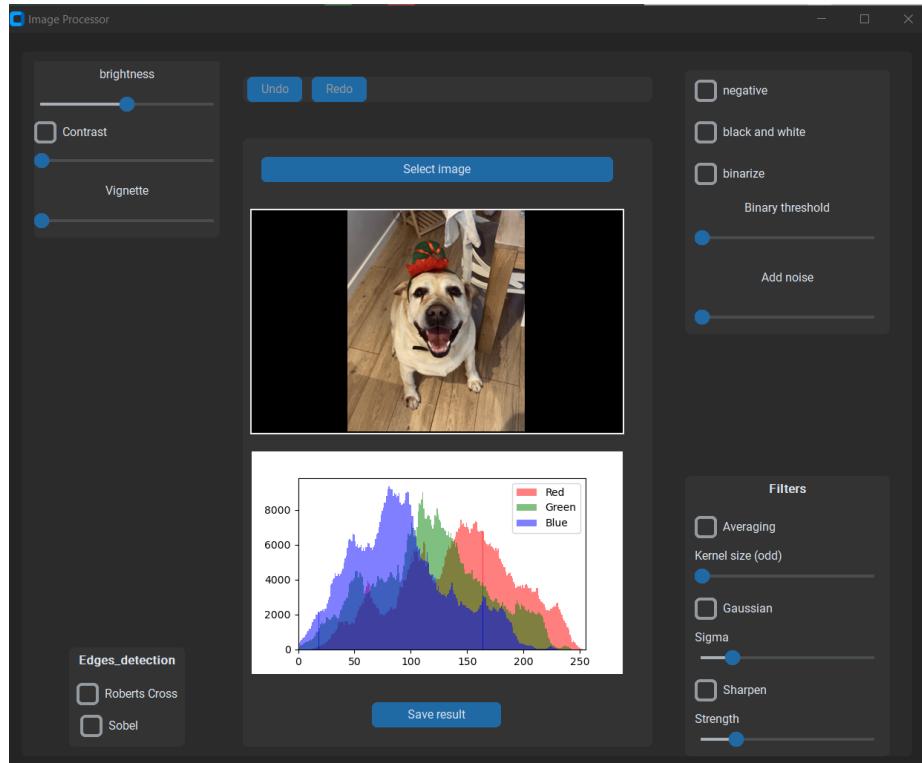
canvas = FigureCanvasTkAgg(fig, master=self.
                           histogram_frame)
canvas.draw()
canvas.get_tk_widget().pack()

plt.close(fig)

```

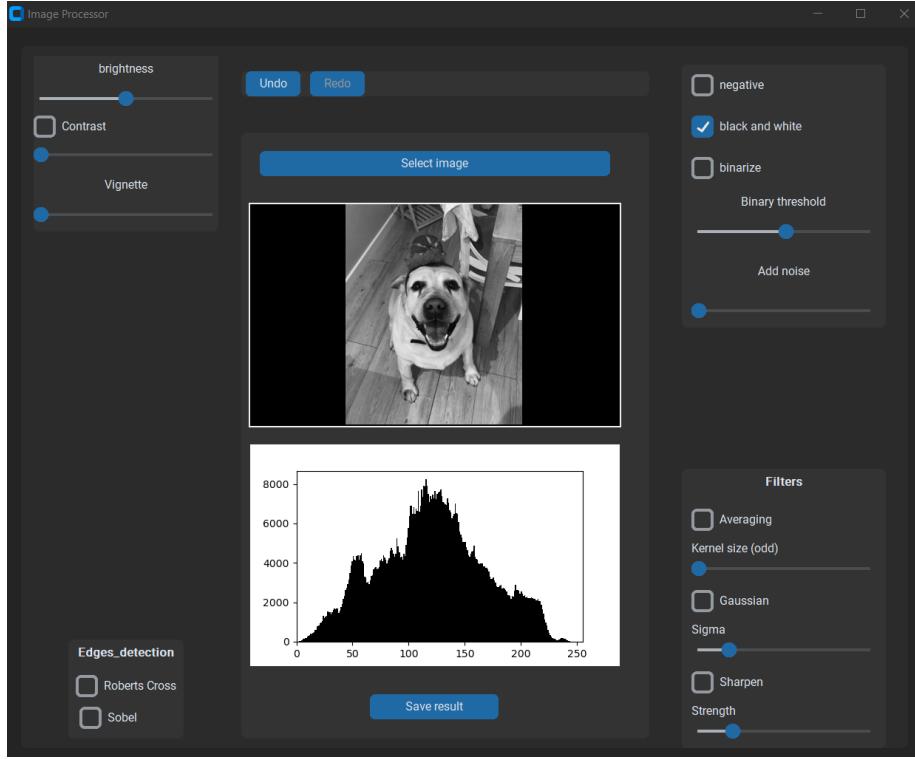
W powyższym kodzie usuwane są wykresy (jeśli istnieją), a następnie generowane są dla obrazów kolorowych trzy histogramy liczności danej wartości (0-255) w danym kanale. Jeśli obraz jest czarnobiały, to analogiczny histogram jest przygotowywany ale dla wartości pikseli w jednym kanale.

Przykład użycia dla oryginalnego, kolorowego obrazu:



Rysunek 3: Wygląd interfejsu aplikacji po załadowaniu zdjęcia. Na środku widać oryginalny obraz, a na dole widoczne są histogramy

Przykład użycia dla obrazu po zastosowaniu przejścia do skali szarości:



Rysunek 4: Wygląd interfejsu aplikacji po zastosowaniu przejścia do skali szarości. Na środku widać przetworzony obraz, a na dole widoczny jest histogram dla jednego kanału

3.5 Cofanie

Mechanizm cofania został zaimplementowany przez utworzenie w MainPanelu pustych list `undo_stack` i `redo_stack` oraz przycisków 'Undo' i 'Redo' wykonujących metody o tej samej nazwie. Konkretniej:

```
def undo(self):
    if self.undo_stack:
        self.redo_stack.append(self.
            processed_image_array.copy())
        self.processed_image_array = self.undo_stack.pop()
        self.display_image()
        self.display_histogram()

    if not self.undo_stack:
```

```

        self.undo_btn.configure(state="disabled")
        self.redo_btn.configure(state="normal")

    def redo(self):
        if self.redo_stack:
            self.undo_stack.append(self.
                processed_image_array.copy())
            self.processed_image_array = self.redo_stack.pop()
            self.display_image()
            self.display_histogram()

        if not self.redo_stack:
            self.redo_btn.configure(state="disabled")
            self.undo_btn.configure(state="normal")

```

W powyższym kodzie po wciśnięciu odpowiedniego przycisku przywracana jest cofnięta lub przywrócona wersja zdjęcia, a następnie aktualizowany jest wyświetlany obraz i histogram. Oprócz wciskania przycisków powyższą funkcjonalność można również osiągnąć przy pomocy klasycznych skrótów klawiszowych ctrl+Z oraz ctrl+Y dzięki linijkom kodu znajdującym się w klasie wywołania aplikacji, czyli `ImageProcessorApp`:

```

self.bind("<Control-z>", lambda event: self.main_panel.undo())
self.bind("<Control-y>", lambda event: self.
    main_panel.redo())

```

3.6 Zapisywanie przekształconego obrazu

Obrazy na które użytkownik nanieśnie przekształcenie mogą zostać zapisane przy kliknięciu przycisku 'Save result'. Wykonywana jest wtedy metoda `save_image` zaimplementowana w następujący sposób:

```

def save_image(self):
    if self.processed_image_array is not None:
        file_path = filedialog.asksaveasfilename(
            defaultextension=".png", filetypes=[("PNG
            files", "*.png"), ("JPEG files", "*.jpg")])
    if file_path:
        saved_image = Image.fromarray(self.
            processed_image_array)
        saved_image.save(file_path)

```

W powyższym kodzie, o ile przetworzony obraz istnieje (jeśli nie dokonano żadnego przekształcenia, to jest nim oryginalny obraz), program prosi użytkownika o nazwanie pliku .png do pobrania. Następnie w wybranej ścieżce zapisywany jest obraz, który powstał przez przekształcenie struktury NumPy array w obraz (biblioteka PIL).

4 Metody przetwarzania obrazów

Wyniki wszystkich opisanych poniżej metod będziemy prezentować na następującym zdjeciu:



Rysunek 5: Oryginalne zdjęcie

4.1 Podstawowe przekształcenia obrazów

Kluczową funkcjonalnością aplikacji Image Processor jest zastosowanie podstawowych przekształceń. Są one oparte o proste operacje na macierzach (między innymi dodawanie lub odejmowanie wartości od elementów macierzy) przy zastosowaniu biblioteki NumPy i w implementacji nie wykorzystują masek.

4.1.1 negative

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B)

Działanie: Każdy piksel jest przekształcany zgodnie ze wzorem

$$P' = 255 - P$$

operacja ta powoduje, że obszary jasne stają się ciemne, a kolory zostają odwrócone(np. czerwony zamienia się na cyan)

Wyjście: Obraz reprezentowany jako numpy array, o tych samych wymiarach co na wejściu z odwroconymi wartościami pikseli

```
def negative(self, image):
    return 255 - image
```

przykład wywołania funkcji:



Rysunek 6: przykład wywołania negatywu

4.1.2 to_grayscale

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B)

Działanie: Funkcja pobiera 3 tablice kanałów kolorów, a następnie tworzy średnią ważoną wartości RGB, według standarowej metody konwersji do odcieni szarości zgodnej z modelem jasności ludzkiego oka, według wzoru:

$$OUT = 0.299 * R + 0.587 * G + 0.114 * B$$

Wyjście: Obraz reprezentowany jako dwuwymiarowa numpy array o wymiarach (wysokość, szerokość) o wartości pikseli w skali szarości

```
def to_grayscale(self, image):
    return np.dot(image[... , :3], [0.299, 0.587, 0.114])
        .astype(np.uint8)
```

Przykład wywołania funkcji:



Rysunek 7: przykład wywołania zamiany do skali szarości

4.1.3 adjust_brightness

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3) lub (wysokość, szerokość), gdzie wartości pikseli mieszczą się w zakresie od 0 do 255.

Działanie: Funkcja dodaje wartość ‘brightness’ (pobraną z suwaka o zakresie [-255, 255]) do każdego piksela obrazu. Jeśli ‘brightness’ jest liczbą dodatnią, obraz zostaje rozjaśniony, a jeśli jest ujemna, obraz zostaje przyciemniony. Wartości poza zakresem [0, 255] są przycinane do tego zakresu za pomocą funkcji np.clip, aby zapobiec przekroczeniu dozwolonego zakresu wartości pikseli.

Wyjście: Obraz reprezentowany jako numpy array o tych samych wymiarach co wejściowy obraz, ale z dostosowaną jasnością, gdzie wartości pikseli są w zakresie od 0 do 255.

```
def adjust_brightness(self, image, brightness):
    return np.clip(image + brightness, 0, 255).astype(np
        .uint8)
```

Przykład wywołania funkcji:



Rysunek 8: przykład wywołania rozjaśniania

4.1.4 adjust_contrast

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3) lub (wysokość, szerokość), gdzie wartości pikseli mieszą się w zakresie od 0 do 255.

Działanie: Funkcja jest wywoływaną tylko wtedy, gdy zaznaczony jest checkbox umożliwiający zmianę kontrastu. Parametr `contrast` jest pobierany z suwaka, który ma zakres wartości od 0 do 20. Funkcja normalizuje wartości pikseli obrazu względem maksymalnej wartości piksela, a następnie podnosi je do potęgi określonej przez parametr `contrast`. Jeśli `contrast < 1`, obraz staje się bardziej kontrastowy, natomiast jeśli `contrast > 1`, kontrast obrazu zostaje zmniejszony. Wynikowa wartość jest przeskalowana do zakresu [0, 255].

Wyjście: Obraz reprezentowany jako numpy array o tych samych wymiarach co wejściowy obraz, ale z dostosowanym kontrastem, gdzie wartości pikseli są w zakresie od 0 do 255.

```
def adjust_contrast(self, image, contrast):
    return 255*(image/np.max(image))**contrast
```

Przykład wywołania:



Rysunek 9: przykład wywołania kontrastu

4.1.5 vignette

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B).

Działanie: Funkcja tworzy maskę winiety, której wartość zależy od odległości od środka obrazu. Aby obliczyć maskę, najpierw generujemy siatkę współrzędnych dla każdego piksela w obrazie. Używamy funkcji `np.linspace`, aby uzyskać wektory współrzędnych w osi x i y w zakresie od -1 do 1:

$$x = \text{np.linspace}(-1, 1, \text{width})$$

$$y = \text{np.linspace}(-1, 1, \text{height})$$

Następnie tworzymy dwuwymiarową siatkę współrzędnych za pomocą funkcji `np.meshgrid`, która generuje pełną macierz współrzędnych X i Y dla każdego punktu obrazu:

$$X, Y = \text{np.meshgrid}(x, y)$$

Siatka (X, Y) zawiera wartości dla każdej współrzędnej w przestrzeni obrazu. Wartość maski winiety w punkcie (x, y) jest obliczana na podstawie odległości od środka obrazu, korzystając ze wzoru:

$$V(x, y) = \exp \left(-\text{factor} \cdot \left(\sqrt{x^2 + y^2} \right)^2 \right)$$

Gdzie $\sqrt{x^2 + y^2}$ to odległość od środka obrazu dla współrzędnych x i y . Następnie, dla każdego piksela obrazu, obliczamy nową wartość na podstawie maski

winiety $V(x, y)$ poprzez mnożenie wartości piksela przez odpowiednią wartość maski:

$$\text{Nowa wartość piksela} = \text{Oryginalna wartość piksela} \cdot V(x, y)$$

Wyjście: Obraz reprezentowany jako numpy array o tych samych wymiarach co wejściowy obraz, ale z nałożoną winietą.

```
def vignette(self, image, factor):
    height, width, _ = image.shape
    x = np.linspace(-1,1, width)
    y = np.linspace(-1,1, height)
    X, Y = np.meshgrid(x, y)
    distance = np.sqrt(X ** 2 + Y ** 2)
    vignette_mask = np.exp(-factor*(distance**2))
    image = (image * vignette_mask[:, :, np.newaxis]).astype
        (np.uint8)
    return image
```

Przykład wywołania:



Rysunek 10: przykład wywołania winiety

4.1.6 add_noise

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B).

Działanie: Funkcja dodaje losowy szum do obrazu, aby uzyskać efekt zniekształcenia. Szum jest generowany przez funkcję `np.random.randint`, która

tworzy macierz o tym samym kształcie co obraz, wypełnioną losowymi liczbami całkowitymi z przedziału [0, 255] (przy założeniu 8-bitowej głębokości kolorów). Szum jest skalowany przez parametr `noise_factor`, który kontroluje intensywność szumu i jest pobierany z suwaka w interfejsie użytkownika. Wartość `noise_factor` może przyjmować wartości w przedziale [0, 1], gdzie wartość 0 oznacza brak szumu, a wartość 1 oznacza pełne dodanie losowego szumu. Wzór generowania szumu to:

```
Szum = np.random.randint(0, 256, image.shape, dtype=np.uint8)
```

Następnie, do oryginalnego obrazu dodawany jest ten szum, a wynikowy obraz jest ograniczany do zakresu wartości pikseli [0, 255] za pomocą funkcji `np.clip`, aby zapobiec przekroczeniu dozwolonych wartości:

```
noisy_image = np.clip(image + noise * noise_factor, 0, 255)
```

Wartość `noise_factor` kontroluje intensywność dodawanego szumu. Im większa wartość `noise_factor`, tym bardziej obraz jest zniekształcony przez szum.

Wyjście: Obraz z dodanym szumem, reprezentowany jako numpy array o tych samych wymiarach co wejściowy obraz.

```
def add_noise(self, image, noise_factor):
    noise = np.random.randint(0, 256, image.shape, dtype=np.uint8)
    noisy_image = np.clip(image + noise * noise_factor,
                          0, 255).astype(np.uint8)
    return noisy_image
```

Przykład wywołania:



Rysunek 11: przykład dodania szumu

4.2 Filtry

Bardziej zaawansowanym rodzajem przekształceń nałożonym na zdjęcia są filtry. W tej i kolejnych sekcjach zostaną opisane metody wykorzystujące operację splotu.

Założmy, że mamy macierz obrazu $I \in R^{M \times N}$ oraz maski (jądra) $K \in R^{m \times n}$. Wtedy obraz w wyniku splotu (w podstawowej wersji bez paddingu) otrzymamy przez przemnożenie każdego wycinka $m \times n$ macierzy obrazu przez maskę w następujący sposób:

$$O(i, j) = \sum_{u=0}^{m-1} \sum_{v=0}^{n-1} K(u, v) \cdot I(i + u, j + v)$$

gdzie $O(i, j)$ to wartość w pozycji (i, j) wynikowej macierzy splotu. Ze względu na to, że splot w takiej postaci nie jest możliwy na krawędziach, to dodajemy padding, czyli pomocnicze piksele umożliwiające naniesienie maski na obraz, ale niepojawiające się w obrazie wynikowym. Implementacja tego algorytmu prezentuje się w sposób następujący:

```
def convolve(self, image, kernel):
    if len(image.shape) == 3:
        return np.stack([self.convolve_channel(image
                                                [..., c], kernel) for c in range(3)], axis
                        =-1)
    else:
```

```

        return self.convolve_channel(image, kernel)

def convolve_channel(self, channel, kernel):
    kh, kw = kernel.shape
    ph, pw = kh // 2, kw // 2
    padded = np.pad(channel, ((ph, ph), (pw, pw)), mode=
                     'reflect')
    output = np.zeros_like(channel)

    for i in range(channel.shape[0]):
        for j in range(channel.shape[1]):
            region = padded[i:i + kh, j:j + kw]
            output[i, j] = np.sum(region * kernel)

    return np.clip(output, 0, 255).astype(np.uint8)

```

W powyższym kodzie zaimplementowana została metoda splotu z paddingiem (metoda `convolve_channel`), która następnie zostaje zastosowana do każdego z kanałów zdjęcia.

4.2.1 Filtr uśredniający

Filtr uśredniający polega na operacji splotu, w której użyta zostaje maska taka, że każdy element jest równy, a ich suma jest równa 1 (macierz znormalizowana). Innymi słowy macierz jądra jest postaci:

$$K_{\text{avg}} = \frac{1}{k^2} \begin{bmatrix} 1 & 1 & \cdots & 1 \\ 1 & 1 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}_{k \times k}$$

Nałożenie filtru uśredniającego zostało zaimplementowane w sposób następujący:

```

def apply_average_filter(self, image, kernel_size=3):
    kernel = np.ones((kernel_size, kernel_size)) / (
        kernel_size * kernel_size)
    return self.convolve(image, kernel)

```

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B)

Wyjście: Analogiczny obraz, w którym widać efekt równomiernego rozmycia (uśrednienia wartości pikseli)

Przykład wywołania:

4.2.2 Filtr gaussowski

Filtr gaussowski polega na wykonaniu podobnej operacji co filtr uśredniający, tyle że maska nie "uśrednia" w sposób jednostajny, a w sposób "normalny"



Rysunek 12: przykład wywołania filtru uśredniającego

na podstawie dwuwymiarowej krzywej Gaussa. W rozumieniu matematycznym maskę można opisać w następujący sposób:

$$K_{\text{gauss}}(i, j) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{i^2 + j^2}{2\sigma^2}\right)$$

przy czym zachodzi następująca własność, że macierz jest $k \times k$ a indeksy są wycentrowane (środkowa element odczytujemy jako $(0, 0)$). Następnie wartości w masce zostają znormalizowane:

$$K_{\text{normalized}} = \frac{K_{\text{gauss}}}{\sum_{i,j} K_{\text{gauss}}(i, j)}$$

Implementacja powyższego filtru w programie:

```
def apply_gaussian_filter(self, image, sigma=1.0,
    kernel_size=5):
    ax = np.linspace(-(kernel_size - 1) / 2., (
        kernel_size - 1) / 2., kernel_size)
    gauss = np.exp(-0.5 * (ax / sigma) ** 2)
    kernel = np.outer(gauss, gauss)
    kernel /= np.sum(kernel)
    return self.convolve(image, kernel)
```

W powyższym kodzie wygenerowano macierz rozmiaru maski taką, że ma 0 w środku i sąsiadujące wartości różnią się o 1 (przy użyciu `np.linspace`). Następnie obliczono dla każdego elementu macierzy wartość dwuwymiarowej krzywej

Gaussa i całość została znormalizowana. Następnie zastosowano splot przy użyciu wcześniej opisanej maski i zadanego obrazu przez użytkownika.

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B)

Wyjście: Analogiczny obraz, w którym widać efekt rozmycia nierównomiernego (zgodnego z rozkładem normalnym)

Przykład wywołania:



Rysunek 13: przykład wywołania filtru gaussowskiego

4.2.3 Filtr wyostrzający

W tym filtrze jako maskę wykorzystano następującą macierz:

$$K_{\text{sharpen}} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 + \alpha & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

w której to parametr α określany jako siła wyostrzenia jest wybierany przez użytkownika za pomocą suwaka w panelu filtrów. Metoda została zaimplementowana w następujący sposób:

```
def apply_sharpen_filter(self, image, strength=1.0):
    kernel = np.array([[0, -1, 0],
                      [-1, 4 + strength, -1],
                      [0, -1, 0]])
    return self.convolve(image, kernel)
```

W powyższej implementacji filtr nakładany jest poprzez operację splotu maski zdefiniowanej wcześniej (`strength= α`) i zdjęcie do przetworzenia. **Wejście:** Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B)
Wyjście: Analogiczny obraz, w którym widać efekt wyostrzenia
Przykład wywołania:



Rysunek 14: przykład wywołania filtra wyostrzającego

4.3 Wykrywanie krawędzi

W programie zaimplementowane zostały również zaawansowane metody służące wykrywaniu krawędzi (granic obszarów o różnych odcieniach jasności) za pomocą bardziej złożonych algorytmów: krzyża Robertsa i operatorów Sobela.

4.3.1 roberts_cross

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B).

Działanie: Funkcja implementuje operator krawędziowy Roberta, który jest używany do detekcji krawędzi na obrazie. Metoda ta działa poprzez zastosowanie dwóch macierzy konwolucji jednego dla kierunku poziomego (x) i drugiego dla kierunku pionowego (y).

Operator Roberta stosuje następujące macierze konwolucji:

$$K_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad K_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Pierwsza macierz (K_x) wykrywa krawędzie w kierunku poziomym, a druga (K_y) w kierunku pionowym.

Po zastosowaniu obu macierzy konwolucji do obrazu, obliczana jest amplituda gradientu dla każdego piksela, która jest równa pierwiastkowi sumy kwadratów gradientów w kierunkach x i y :

$$\text{gradient_magnitude} = \sqrt{\text{grad}_x^2 + \text{grad}_y^2}$$

Wynikowa amplituda gradientu jest następnie ograniczona do zakresu [0, 255] przy pomocy funkcji `np.clip`, aby zapobiec przekroczeniu dozwolonych wartości pikseli. Wartości pikseli są konwertowane na typ `np.uint8`, co daje obraz z wykrytymi krawędziami.

Funkcja zapamiętuje wejściowy obraz oraz wynikowy obraz w strukturze `self.cached_roberts` w celu zoptymalizowania obliczeń. Jeśli wywołanie funkcji odbywa się dla tego samego obrazu, funkcja zwraca wcześniej obliczony wynik bez konieczności ponownej konwolucji.

Wyjście: Obraz przedstawiający krawędzie wykryte przez operatora Roberta, reprezentowany jako numpy array o wymiarach (wysokość, szerokość), o wartościach w zakresie [0, 255].

```
def roberts_cross(self, image):
    if self.cached_roberts is not None and np.
        array_equal(image, self.cached_roberts['input']):
        return self.cached_roberts['output']
    kernel_x = np.array([[1, 0],
                         [0, -1]])
    kernel_y = np.array([[0, 1],
                         [-1, 0]])

    grad_x = self.convolve(image, kernel_x)
    grad_y = self.convolve(image, kernel_y)

    gradient_maginitude = np.sqrt(grad_x**2+grad_y**2)
    result = np.clip(gradient_maginitude, 0, 255).astype
        (np.uint8)
    self.cached_roberts = {
        'input': image.copy(),
        'output': result.copy()
    }
    return result
```

Przykład wywołania:



Rysunek 15: przykład wykrywania krawędzi krzyżem Robertsa

4.3.2 sobel_operator

Wejście: Obraz reprezentowany jako numpy array o kształcie (wysokość, szerokość, 3), gdzie ostatni wymiar odpowiada trzem kanałom koloru (R, G, B).

Działanie:

1. Obraz (o ile jest kolorowy w sensie RGB) zostaje przekształcony do odcienni szarości za pomocą omawianej wcześniej metody `to_greyscale`.

Definiowane są dwie maski Sobela. W naszej aplikacji zaimplementowane zostały:

Maska do detekcji zmian poziomych (o kącie 0°):

$$K_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

oraz analogicznie: **Maska do detekcji zmian pionowych** (o kącie 90°):

$$K_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Wyjście: Obraz przedstawiający krawędzie wykryte przez operator Roberta, reprezentowany jako numpy array o wymiarach (wysokość, szerokość), o wartościach w zakresie [0, 255].

```

def sobel_operator(self, image):
    if len(image.shape) == 3:
        image = self.to_greyscale(image)

    # dla 0 stopni
    kernel_x = np.array([[1, 0, -1],
                         [2, 0, -2],
                         [1, 0, -1]])

    # dla 90 stopni
    kernel_y = np.array([[1, 2, 1],
                         [0, 0, 0],
                         [-1, -2, -1]])

    grad_x = self.convolve(image, kernel_x)
    grad_y = self.convolve(image, kernel_y)

    sobel = np.sqrt(grad_x.astype(np.float32) ** 2 +
                     grad_y.astype(np.float32) ** 2)
    sobel = np.clip(sobel, 0, 255).astype(np.uint8)

    self.cached_sobel = {
        'input': image.copy(),
        'output': sobel.copy()
    }

    return np.stack((sobel,) * 3, axis=-1)

```

Przykład wywołania:

5 Wnioski

Aplikacja mimo, że daje zadowalające efekty opisanych w poprzednim rozdziale metod, działa wolno. Wolne działanie jest szczególnie zauważalne przy wywoływaniu metod używających funkcji `convolve`, czyli każdy z filtrów oraz metody wykrywania krawędzi. Wynika to z dużej złożoności obliczeniowej funkcji `convolve`, która wynosi $O(H * W * k^2)$, gdzie H to wysokość obrazu, W to szerokość, a k to rozmiar jądra operacji splotu. W celu usprawnienia działania aplikacji w przyszłych wersjach można rozważyć użycie do operacji splotu funkcji `convolve` z biblioteki `scipy`.

Aplikacja może również zostać rozszerzona o możliwość sekwencyjnego nakładania filtrów, co pozwoli na modyfikowanie obrazu bez resetowania wcześniejszych przekształceń. Mimo, że byłaby to strata dla złożoności pamięciowej, to dostaliśmy rekompensację w postaci lepszej złożoności czasowej.

Filtr wyostrzania generuje artefakty, takie jak nadmiernie podkreślone krawędzie czy szумy, dla dużych wartości parametrów maski. Zwiększenie współczynników w filtrze prowadzi do intensyfikacji efektu wyostrzania, co może skutkować nie-



Rysunek 16: przykład wykrywania krawędzi operatorem Sobela

naturalnym wyglądem obrazu.

Poniżej prezentujemy ciekawe przykłady przekształceń



Rysunek 17: przykład ciekawego połączenia różnych przekształceń (szum i binaryzacja)



Rysunek 18: przykład ciekawego przekształcenia z artefaktami dla bardzo wysokiego parametru wyostrzania