Scrolling Image Processor App Documentation

Name	id
Hagar Galal	42010084
Hadeer Elkady	42010436
Hala Khaled	42020007
Verina Gamal	42010044
Zeyad Mohamed	42010448

Dr /Marian Wagdy

Eng /Nora Elamin

1. Introduction

- Overview

The Scrolling Image Processor App is a versatile image processing tool developed using Tkinter, OpenCV, PIL (Pillow), and NumPy. It provides a user-friendly interface for performing various image manipulations, from basic operations like resizing and color conversion to advanced tasks such as morphology operations and contrast adjustments.

- Purpose

The primary purpose of this application is to offer a convenient platform for users, including both beginners and experienced individuals, to explore and apply a wide range of image processing techniques. Whether it's simple adjustments like resizing or more complex tasks like morphological operations, the app caters to diverse user needs.

- Features

Scrolling Canvas: The app features a scrolling canvas for viewing and processing images of various sizes.

Image Processing Operations: A comprehensive set of image processing operations is available, including resizing, color conversion, brightness adjustment, sharpening, and segmentation.

Color Operations: Users can selectively convert image colors and adjust brightness according to predefined options.

Morphology Operations: The app supports standard morphology operations like dilation, erosion, opening, and closing.

Smoothing Operations: Various smoothing operations, such as average, median, max, and min filtering, are provided.

Contrast Operations: Users can enhance image contrast through equalization and stretching.

User Interface: The interface is designed for ease of use, featuring clear widget layouts and interactive elements.

2. Application Structure

- Class Overview

  The main class, ScrollingImageProcessorApp, orchestrates the application. It handles the initialization of the Tkinter window, the creation of widgets, and the implementation of image processing functions. Key components include the scrolling canvas, buttons for uploading and processing images, and comboboxes for selecting processing options.

```
class ScrollingImageProcessorApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Scrolling Image Processor")
```

- Widget Components

  Canvas: The scrolling canvas allows users to view images of varying sizes comfortably.

  Buttons: Various buttons, such as "Upload Image," "Process Image," and specific operation buttons, trigger corresponding actions.

  Comboboxes: Dropdown lists facilitate the selection of image processing options.

```
        self.upload_button = tk.Button(self.inner_frame, text="Upload Image",
command=self.upload_image, font=("Helvetica", 12), bg="#285059", fg="white")
        self.upload_button.pack(pady=5)
```

```
 self.processing_options = ttk.Combobox(self.inner_frame,
values=self.get_processing_options(), font=("Helvetica", 12))
        self.processing_options.pack(pady=1)
```

```
ef get_processing_options(self):
        return [
            "Resizing",
            "Color Conversion (BGR to Grayscale)",
            "Color Channel Swapping from BGR to RGB",
            "Color Channel Swapping from BGR to GBR",
            "Image Complementing",
            "increase Brightness",
            "Decrease Brightness",
            "Sharpening",
            "Thresholding Segmentation (Global)",
            "Thresholding Segmentation (Adaptive)"
        ]
```

- Styling Choices

  The app's styling choices, including color schemes, font selections, and border designs, contribute to an aesthetically pleasing and cohesive user interface.

```
self.root.geometry("1200x600")
        self.root.configure(bg="#ECECEC")

        # Create main frame
        self.main_frame = tk.Frame(root, bg="#ECECEC")
        self.main_frame.pack(expand=True, fill="both")

        # Create canvas for scrolling
        self.canvas = tk.Canvas(self.main_frame, bg="#fff", width=1200, height=400,
scrollregion=(0, 0, 2400, 800))
        self.canvas.pack(side=tk.LEFT, fill="both", expand=True)

        # Scrollbars
        self.y_scrollbar = ttk.Scrollbar(self.main_frame, orient="vertical",
command=self.canvas.yview)
        self.y_scrollbar.pack(side=tk.LEFT, fill="y")

        # Configure canvas scrolling
        self.canvas.config(yscrollcommand=self.y_scrollbar.set)

        # Create widgets inside canvas
        self.inner_frame = tk.Frame(self.canvas, bg="#fff")
        self.canvas.create_window((0, 0), window=self.inner_frame, anchor="nw")
```

```
        self.header_label = tk.Label(
        self.inner_frame,
        text="Scrolling Image Processor",
        font=("Arial", 22, "italic bold"),  # Changed font family, size, and style
        bg="#285059",  # Changed background color
        fg="white",  # Changed text color to black
        pady=12,  # Adjusted padding on the y-axis
        padx=8,  # Adjusted padding on the x-axis
        borderwidth=3,  # Increased border width
        # Changed border relief style to 'raised'
        )
self.header_label.pack(fill="x")

        self.canvas_image_original = tk.Canvas(self.inner_frame, bg="#fff", width=600,
height=400,highlightthickness=2,highlightbackground="black")
        self.canvas_image_original.pack(side=tk.LEFT)

        self.canvas_image_processed = tk.Canvas(self.inner_frame, bg="#fff", width=600,
height=400,highlightthickness=2,highlightbackground="black")
        self.canvas_image_processed.pack(side=tk.RIGHT)

        self.upload_button = tk.Button(self.inner_frame, text="Upload Image",
command=self.upload_image, font=("Helvetica", 12), bg="#285059", fg="white")
        self.upload_button.pack(pady=5)
        self.processing_options = ttk.Combobox(self.inner_frame,
values=self.get_processing_options(), font=("Helvetica", 12))
        self.processing_options.pack(pady=1)
```

3. Image Processing Operations

- Resizing
  Users can input desired width and height values for resizing Ensure valid positive integers are entered for width and height.

```
if operation == "Resizing":
            width = int(self.width_entry.get())
            height = int(self.height_entry.get())
            if width > 0 and height > 0:
            processed_image = cv2.resize(processed_image, (width, height))
```

- Color Conversion (BGR to Grayscale)
  Converts the image to grayscale, removing color information.
  No additional parameters required.

```
operation == "Color Conversion (BGR to Grayscale)":
            processed_image = cv2.cvtColor(processed_image, cv2.COLOR_BGR2GRAY)
```

- Color Channel Swapping
  Swap color channels to create various color effects.
  Options include swapping from BGR to RGB or GBR.

```
operation == "Color Channel Swapping from BGR to RGB":
            processed_image = processed_image[:, :, [2, 1, 0]]
        elif operation == "Color Channel Swapping from BGR to GBR":
            processed_image = processed_image[:, :, [1, 0, 2]]
```

- Image Complementing
  Inverts pixel values, creating a negative effect.

```
operation == "Image Complementing":
            processed_image = cv2.bitwise_not(processed_image)
```

- Brightness Adjustment
  Users can increase or decrease brightness using predefined values.

```
operation == "increase Brightness":
            brightness_increase = 50
            processed_image = cv2.convertScaleAbs(processed_image, alpha=1,
beta=brightness_increase)
        elif operation == "Decrease Brightness":
            brightness_decrease = 30
            processed_image = cv2.convertScaleAbs(processed_image, alpha=1,
beta=-brightness_decrease)
```

- Sharpening
  Applies a simple sharpening filter

```
operation == "Sharpening":
            processed_image = cv2.subtract(cv2.dilate( processed_image, None),
cv2.erode( processed_image, None))
```

- Thresholding Segmentation
  Utilizes global or adaptive thresholding for segmentation.

```
operation == "Thresholding Segmentation (Global)":
```

```
            ret,processed_image =
cv2.threshold(processed_image,175,255,cv2.THRESH_BINARY)
        elif operation == "Thresholding Segmentation (Adaptive)":
            gray_image = cv2.cvtColor(processed_image, cv2.COLOR_BGR2GRAY)
            processed_image = cv2.adaptiveThreshold(gray_image, 255,
cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 15, 2)
```

4. Color Operations

- Color Conversion
  Converts the entire image to a selected color.

```
def apply_color(self, color):
        if self.image is None:
            return

        processed_image = self.image.copy()

        if color == "convert to red":
            processed_image[:, :, 1] = 0
            processed_image[:, :, 0] = 0
        elif color == "convert to green":
            processed_image[:, :, 0] = 0
            processed_image[:, :, 2] = 0
        elif color == "convert to blue":
            processed_image[:, :, 1] = 0
            processed_image[:, :, 2] = 0
        elif color == "convert to yellow":
            processed_image[:, :, 0] = 0
        elif color == "convert to purple":
            processed_image[:, :, 1] = 0
        elif color == "convert to cyan":
            processed_image[:, :, 2] = 0

        return processed_image
```

- Brightness Adjustment
  Adjusts brightness for specific color channels.

```
def Change_brightness(self, operation):
        if self.image is None:
                return
        processed_image = self.image.copy()
        if operation == "Change brightness to Red":
                brightness_increase = 50
                processed_image[:, :, 2] = cv2.add(processed_image[:, :, 2],
brightness_increase)
        elif operation == "Change brightness to Green":
                brightness_increase = 50
                processed_image[:, :, 1] = cv2.add(processed_image[:, :, 1],
brightness_increase)
        elif operation == "Change brightness to blue":
                brightness_increase = 50
                processed_image[:, :, 0] = cv2.add(processed_image[:, :, 0],
brightness_increase)
        elif operation == "Change brightness to yellow":
                brightness_increase = 50
                processed_image[:, :, 2] = cv2.add(processed_image[:, :, 2],
brightness_increase)
                processed_image[:, :, 1] = cv2.add(processed_image[:, :, 1],
brightness_increase)
        elif operation == "Change brightness to purple":
                brightness_increase = 50
                processed_image[:, :, 2] = cv2.add(processed_image[:, :, 2],
brightness_increase)
                processed_image[:, :, 0] = cv2.add(processed_image[:, :, 0],
brightness_increase)
        elif operation == "Change brightness to cyan":
                brightness_increase = 50
                processed_image[:, :, 1] = cv2.add(processed_image[:, :, 1],
brightness_increase)
                processed_image[:, :, 0] = cv2.add(processed_image[:, :, 0],
brightness_increase)

        return processed_image
```

5. Morphology Operations

- Dilation, Erosion, Opening, Closing
  Users can apply dilation, erosion, opening, or closing morphological operations.

```
def apply_morphology(self, operation):
        if self.image is None:
            return

        processed_image = self.image.copy()

        kernel_size = (5, 5)  # You can adjust the kernel size as needed

        if operation == "Dilation":
            processed_image = cv2.dilate(processed_image, None, iterations=1)
        elif operation == "Erosion":
            processed_image = cv2.erode(processed_image, None, iterations=1)
        elif operation == "Opening":
            processed_image = cv2.morphologyEx(processed_image, cv2.MORPH_OPEN,
cv2.getStructuringElement(cv2.MORPH_RECT, kernel_size))
        elif operation == "Closing":
            processed_image = cv2.morphologyEx(processed_image, cv2.MORPH_CLOSE,
cv2.getStructuringElement(cv2.MORPH_RECT, kernel_size))

        return processed_image
```

6. Smoothing Operations

- Average, Median, Max, Min
  Select from average, median, max, or min filtering for smoothing.

```
 def apply_Smoothing(self, operation):
        if self.image is None:
            return
        processed_image = self.image.copy()
        kernel_size = (5, 5)  # You can adjust the kernel size as needed
        if operation == "Average":
            processed_image = cv2.blur(processed_image, (3, 3))
        elif operation == " Median":
            processed_image = cv2.medianBlur(processed_image, 3)
        elif operation == "Max":
            processed_image = cv2.dilate(processed_image, np.ones((3, 3), np.uint8))
        elif operation == "Min":
          processed_image = cv2.erode(processed_image, np.ones((3, 3), np.uint8))
```

7. Contrast Operations

- Equalization, Stretching
  Enhances contrast through equalization or stretching.

```
def apply_Contrast(self, operation):
        if self.image is None:
            return

        processed_image = self.image.copy()

        kernel_size = (5, 5)  # You can adjust the kernel size as needed

        if operation == "Equalization":
            gray_image = cv2.cvtColor(processed_image, cv2.COLOR_BGR2GRAY)
            processed_image = cv2.equalizeHist(gray_image)
        elif operation == "Stretching":
            processed_image = cv2.dilate(processed_image, np.ones((3, 3),
np.uint8))
        return processed_image
```

8. User Interface

- Widget Details
  1.Canvas
  Two canvas widgets display the original and processed images.
  2.Buttons
  Buttons trigger actions such as image upload, processing, and reverting to the original image.
  3.Comboboxes
  Dropdown lists provide options for various image processing operations.
- Styling and Layout
  1.Aesthetics
  The app utilizes a color scheme of white, black, and shades of blue for a clean and modern look.
  2.Layout
  The layout is organized for intuitive navigation, with clear separation between image display and control widgets.

- Interaction Flow
  Users upload an image, select processing options, adjust parameters if necessary, and click "Process Image" to see the result.
  Additional features, such as saving the processed image and reverting to the original, enhance user control.

9. Code Modularity

- Functions and Methods
  Each image processing operation is encapsulated in a separate function for modularity and maintainability.
  Functions include resizing, color conversion, brightness adjustment, morphology operations, smoothing, and contrast operations.

- Reusability
  Code components are designed for reusability, allowing developers to easily integrate specific operations into other projects.

10. Dependencies

- Library Information
  The app relies on external libraries, including OpenCV, PIL (Pillow), Tkinter, and NumPy.

11. Usage Guide

- Instructions
  1.Launch the app by executing the provided Python script.
  2.Click the "Upload Image" button to select an image for processing.
  3.Choose from the available image processing options in the comboboxes.
  4.Adjust parameters if required (e.g., width, height, brightness values).
  5.Click "Process Image" to apply the selected operation.

6.Save the processed image if desired.

- Example Scenarios
  - Scenario 1: Resizing

1.Upload an image and select "Resizing" as the operation.
2.Enter desired width and height values.
3.Click "Process Image" to see the resized result.

  - Scenario 2: Color Conversion

1.Upload an image and choose "Color Conversion" from the options.
2.The image will be converted to the selected color.

12. Advanced Features

- Image Overlay

Users can overlay one image onto another for creative effects.

- Histogram Analysis

Advanced users can analyze image histograms to understand pixel distribution.

- Custom Filters

Developers can implement custom filters for unique image processing tasks.

13. Error Handling

- Input Validation

The app validates user inputs, ensuring numerical values are entered where required.

- Exception Handling

  Exception handling is implemented to capture and display errors, providing a smooth user experience.

14. Acknowledgments

- Credits and Attributions

1.Acknowledge the contributions of libraries used, such as OpenCV, PIL, and Tkinter.

2.Give credit to any external resources or tutorials that influenced the app's development.


Thank You!