Programming Assignment 1 - BMP Image Editor

Yecheng Wang

301540271

ywa415@sfu.ca

Spring 2025, CMPT 365 @ Simon Fraser University

Essential Code Explanation

Graphical User Interface (GUI) Initialization

The `init_gui(self)` function is responsible for setting up the graphical user interface (GUI) of the BMP Image Editor using the Tkinter library. It creates elements such as a button to open BMP files, a label to display metadata, a canvas for rendering the image, a sliders for adjusting brightness and scale, and three buttons that allow the user to toggle the visibility of rgb channels.

```
# Initialize the GUI
def init_gui(self):
    browse_btn = tk.Button(self.root, text="Open BMP File", command=self.open_file)
    browse_btn.pack()

    self.metadata_label = tk.Label(self.root, text="\nOpen a BMP file to view its metadata.\n")
    self.metadata_label.pack()

    self.canvas = tk.Canvas(self.root, width=400, height=400, bg="gray")
    self.canvas.pack()

    self.brightness_slider = tk.Scale(self.root, from_=0, to=100, orient="horizontal", label="Brightness",
command=self.adjust_brightness)
    self.brightness_slider.set(50)
    self.brightness_slider.pack()

    self.scale_slider = tk.Scale(self.root, from_=0, to=100, orient="horizontal", label="Scale (%)",
command=self.scale_image)
    self.scale_slider.set(50)
    self.scale_slider.pack()

    self.rgb_frame = tk.Frame(self.root)
    self.rgb_frame.pack()

    self.r_btn = tk.Button(self.rgb_frame, text="Toggle R", command=lambda: self.toggle_channel('R'))
    self.r_btn.grid(row=0, column=0)

    self.g_btn = tk.Button(self.rgb_frame, text="Toggle G", command=lambda: self.toggle_channel('G'))
    self.g_btn.grid(row=0, column=1)

    self.b_btn = tk.Button(self.rgb_frame, text="Toggle B", command=lambda: self.toggle_channel('B'))
    self.b_btn.grid(row=0, column=2)
```

Opening a BMP File

The `open_file(self)` function is triggered when the user clicks the "Open BMP File" button. It begins by opening a file selection dialog that allows users to choose a BMP file from their system. Once a file is selected, the function reads the binary data of the file and verifies whether it is a valid BMP image by checking if the first two bytes match the BMP signature ("BM"). If the file is valid, it proceeds to extract the metadata, display the metadata within the GUI, and call the appropriate function to load the image. If an invalid file is selected, an error message is displayed.

```
# Open a BMP file and check if it is a valid BMP file using the signature bits
def open_file(self):
    self.file_path = filedialog.askopenfilename(filetypes=[("BMP files", "*.bmp")])
    if not self.file_path:
        return

    try:
        with open(self.file_path, "rb") as bmp_file:
            self.image_data = bmp_file.read()

        if len(self.image_data) < 2:
            messagebox.showerror("Error", "File is too small to be a valid BMP file.")
            return

        if self.image_data[:2] != b'BM':
            messagebox.showerror("Error", "Invalid BMP file! The file signature does not match 'BM'.")
            return

        self.parse_metadata()
        self.display_metadata()
        self.load_image()

    except FileNotFoundError:
        messagebox.showerror("Error", "File not found.")
    except PermissionError:
        messagebox.showerror("Error", "Permission denied.")
    except Exception as e:
        messagebox.showerror("Error", f"Failed to open file: {e}")
```

Extracting and Displaying Metadata

The `parse_metadata(self)` function is responsible for extracting key metadata from the BMP file, including file size, image dimensions (width and height), and bits per pixel (BPP). These values are retrieved directly from the BMP file header using appropriate byte offsets and converted into human-readable format. The `display_metadata` function then takes this extracted information and formats it into a string, displaying it within the GUI.

```python
# Function to parse the metadata of the BMP file and take the correct bits for each metadata
def parse_metadata(self):
    self.metadata['file_size'] = int.from_bytes(self.image_data[2:6], 'little')
    self.metadata['width'] = int.from_bytes(self.image_data[18:22], 'little')
    self.metadata['height'] = int.from_bytes(self.image_data[22:26], 'little')
    self.metadata['bpp'] = int.from_bytes(self.image_data[28:30], 'little')

# Display the metadata of the BMP file
def display_metadata(self):
    file_size = self.metadata['file_size']
    for unit in ['B', 'KB', 'MB', 'GB']:
        if file_size < 1024.0:
            break
        file_size /= 1024.0
    file_size_str = f"{file_size:.2f} {unit}"

    metadata_text = (
        f"File Size: {file_size_str}\n"
        f"Width: {self.metadata['width']} pixels\n"
        f"Height: {self.metadata['height']} pixels\n"
        f"BPP: {self.metadata['bpp']} bits per pixel"
    )
    self.metadata_label.config(text=f"\nMetadata:\n\n{metadata_text}")
```

Loading the BMP Image

The `load_image(self)` function determines the appropriate method for loading the BMP file based on its bits per pixel (BPP). If the image is a 24-bit true color BMP, the function calls `load_24bpp_image`, whereas for indexed color images (1, 4, or 8 BPP), it calls `load_indexed_image`. After the image is successfully loaded, the function updates the displayed image to reflect any transformations or adjustments made by the user.

```python
# Load the BMP image and check the bits per pixel to determine the image type
# 24 bits per pixel is a true color image, while 1, 4, and 8 bits per pixel are indexed color images
def load_image(self):
    try:
        bpp = self.metadata['bpp']
        if bpp not in [1, 4, 8, 24]:
            raise ValueError("Unsupported bits per pixel. Only 1, 4, 8, and 24 are supported.")

        offset = int.from_bytes(self.image_data[10:14], 'little')
        width = self.metadata['width']
        height = self.metadata['height']

        if bpp == 24:
            self.load_24bpp_image(offset, width, height)
        else:
            self.load_indexed_image(offset, width, height, bpp)

        self.update_displayed_image()
    except Exception as e:
        messagebox.showerror("Error", f"Failed to load image: {e}")
```

Loading a 24-Bit BMP Image

The `load_24bpp_image(self, offset, width, height)` function is designed to handle BMP images that use 24-bit true color representation. This format stores each pixel as three consecutive bytes (blue, green, and red), meaning that each pixel consists of a BGR triplet. Since BMP files store images from bottom to top, this function also reverses the order of the rows while constructing the final image. The function reads the raw pixel data from the specified offset and populates a new PIL image object with the correct RGB values.

```python
# Load a 24-bit per pixel BMP image
def load_24bpp_image(self, offset, width, height):
    row_size = ((24 * width + 31) // 32) * 4
    pixel_array = self.image_data[offset:]
    pixels = []

    for row in range(height):
        start = row * row_size
        end = start + width * 3
        pixels.append(pixel_array[start:end])

    pixels.reverse()

    self.image = Image.new("RGB", (width, height))
    for y, row in enumerate(pixels):
        for x in range(width):
            b, g, r = row[x * 3:x * 3 + 3]
            self.image.putpixel((x, y), (r, g, b))
```

Loading Indexed BMP Images (1, 4, and 8 BPP)

The `load_indexed_image(self, offset, width, height, bpp)` function is responsible for loading BMP images that use indexed color representation. These images use a color palette, meaning that each pixel is represented by an index that maps to a predefined set of RGB values. The function first retrieves the color palette and then extracts pixel data based on the image's BPP. For 1-bit images, each bit represents a pixel, while 4-bit and 8-bit images use multiple bits per pixel. The extracted colors are then used to create an RGB image for display.

```python
# Load an indexed color BMP image for 1, 4, and 8 bits per pixel
def load_indexed_image(self, offset, width, height, bpp):
    palette_size = 2 ** bpp
    palette = self.image_data[54:54 + palette_size * 4]
    row_size = ((bpp * width + 31) // 32) * 4
    pixel_array = self.image_data[offset:]
    pixels = []

    for row in range(height):
        start = row * row_size
        end = start + row_size
        pixels.append(pixel_array[start:end])

    pixels.reverse()

    self.image = Image.new("RGB", (width, height))
    for y, row in enumerate(pixels):
        for x in range(width):
            if bpp == 1:
                byte_index = x // 8
                bit_index = 7 - (x % 8)
                index = (row[byte_index] >> bit_index) & 1
            elif bpp == 4:
                byte_index = x // 2
                index = (row[byte_index] >> 4) & 0xF if x % 2 == 0 else row[byte_index] & 0xF
            elif bpp == 8:
                index = row[x]
            else:
                raise ValueError("Unsupported bits per pixel")

            b, g, r, _ = palette[index * 4:index * 4 + 4]
            self.image.putpixel((x, y), (r, g, b))
```

Applying Image Transformations

The `update_displayed_image(self)` function is responsible for applying transformations to the currently loaded image. These transformations include brightness adjustments, scaling, and RGB channel toggling. First, the function converts the image from RGB to the YUV color space, where brightness modifications are applied by adjusting the luminance (Y) component. The modified image is then converted back to RGB. The function also calls `manual_scale_image` to handle scaling, ensuring that the image is resized appropriately. Finally, it applies the RGB channel toggling by filtering out selected color channels.

```python
# Update the displayed image by adjusting brightness, scaling, and applying RGB filters
# The image is converted to YUV color space to adjust brightness, then converted back to RGB
# The RGB filters are applied by setting the R, G, B channels to 0 if the channel is disabled
def update_displayed_image(self):
    if self.image:
        # Convert image to YUV color space
        yuv_array = self.rgb_to_yuv(self.image)

        # Adjust brightness by modifying the Y channel
        brightness_factor = self.brightness_slider.get() / 50  # Normalize 0.5 - 2.0
        yuv_array[:, :, 0] *= brightness_factor  # Scale Y (luminance)

        # If brightness is 0%, set YUV to black
        if self.brightness_slider.get() == 0:
            yuv_array[:, :, :] = 0  # Set Y, U, V to 0

        # Convert back to RGB
        modified_image = self.yuv_to_rgb(yuv_array)

        # # Apply scaling
        scale_factor = max(0.01, self.scale_slider.get() / 100.0)
        modified_image = self.manual_scale_image(modified_image, scale_factor)

        # Apply RGB filters
        r_on, g_on, b_on = self.rgb_enabled['R'], self.rgb_enabled['G'], self.rgb_enabled['B']
        pixels = modified_image.load()
        for x in range(modified_image.width):
            for y in range(modified_image.height):
                r, g, b = pixels[x, y]
                r = r if r_on else 0
                g = g if g_on else 0
                b = b if b_on else 0
                pixels[x, y] = (r, g, b)

        # Update current image for display
        self.current_image = modified_image
        self.display_image()
```

Manual Image Scaling

The `aggregate_scale_image(self, image, scale_factor)` function implements manual image scaling using nearest neighbor interpolation. This function computes new pixel values by mapping them to their nearest neighbors in the original image. This method is efficient for preserving the overall structure of the image while resizing it. The function ensures that scaling is performed correctly even for very small or very large scale factors.

```python
# Uses a manual method to aggregate the pixels to scale the image
def aggregate_scale_image(self, image, scale_factor):
    if scale_factor <= 0:
        return image  # Return the original image if scale is 0

    original_width, original_height = image.size
    new_width = max(1, int(original_width * scale_factor))
    new_height = max(1, int(original_height * scale_factor))

    # Get pixel data
    original_pixels = np.array(image)

    # Create an empty new image array
    new_pixels = np.zeros((new_height, new_width, 3), dtype=np.uint8)

    for y in range(new_height):
        for x in range(new_width):
            # Compute nearest neighbor source pixel
            orig_x = min(original_width - 1, int(x / scale_factor))
            orig_y = min(original_height - 1, int(y / scale_factor))
            new_pixels[y, x] = original_pixels[orig_y, orig_x]

    # Convert back to PIL image
    return Image.fromarray(new_pixels)
```

Color Space Conversions

The program includes two key functions for handling color space conversions: `rgb_to_yuv(self, image)` and `yuv_to_rgb(self, yuv_array)`. These functions allow the program to convert images between RGB and YUV formats. The YUV color space is useful for adjusting brightness, as the luminance (Y) component can be manipulated separately from color information.

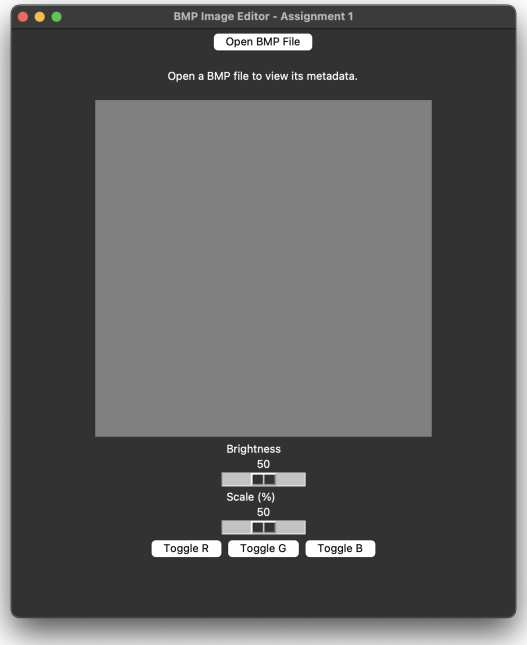
```
# Helper functions to convert between RGB and YUV color spaces
def rgb_to_yuv(self, image):
    img_array = np.array(image, dtype=np.float32) / 255.0
    R, G, B = img_array[:, :, 0], img_array[:, :, 1], img_array[:, :, 2]
    Y = 0.299 * R + 0.587 * G + 0.114 * B
    U = -0.14713 * R - 0.28886 * G + 0.436 * B
    V = 0.615 * R - 0.51499 * G - 0.10001 * B
    return np.stack((Y, U, V), axis=-1)

# Convert YUV array back to RGB image
def yuv_to_rgb(self, yuv_array):
    Y, U, V = yuv_array[:, :, 0], yuv_array[:, :, 1], yuv_array[:, :, 2]
    R = Y + 1.13983 * V
    G = Y - 0.39465 * U - 0.58060 * V
    B = Y + 2.03211 * U
    rgb_array = np.clip(np.stack((R, G, B), axis=-1) * 255.0, 0, 255).astype(np.uint8)
    return Image.fromarray(rgb_array)
```

Program Execution and Event Handling

The final block of code contains the main execution functions that allow the program to run. It includes the Tkinter event loop that listens for user input, ensuring that the image updates dynamically whenever sliders or buttons are used.

```
def adjust_brightness(self, value):
    self.update_displayed_image()

def scale_image(self, value):
    self.update_displayed_image()

def toggle_channel(self, channel):
    self.rgb_enabled[channel] = not self.rgb_enabled[channel]
    self.update_displayed_image()

def display_image(self):
    self.tk_image = ImageTk.PhotoImage(self.current_image)
    self.canvas.create_image(200, 200, image=self.tk_image, anchor=tk.CENTER)

if __name__ == "__main__":
root = tk.Tk()
root.geometry("600x700")
app = BMPViewer(root)
root.mainloop()
```

Project Screenshots



Screenshot of the program as it first loads

Screenshot of the program when a 24 bpp image is loaded (BIOS.png)

Figure 1: The program as the brightness is set to 100%

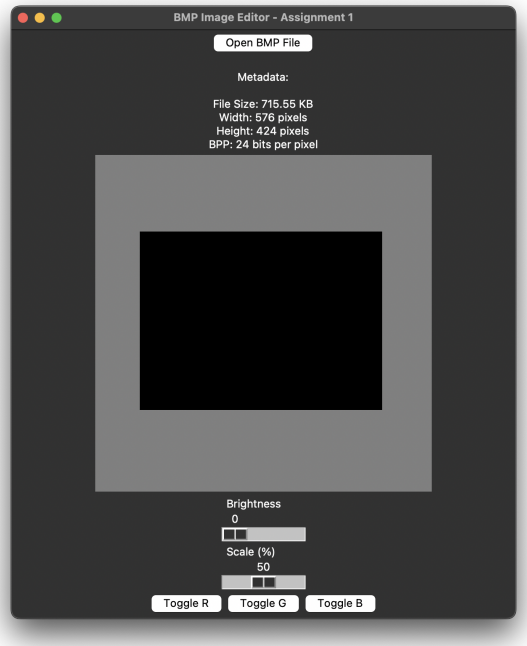


Figure 2: The program as brightness is set to 0%

Figure 3: The program as scaling is set to 100%

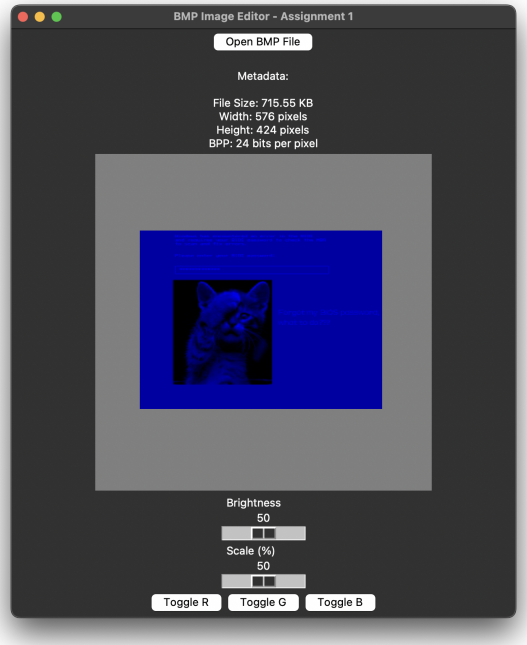


Figure 4: Screenshot of toggling the rgb values, only leaving blue

Figure 5: Screenshot of toggling the rgb values, only leaving red

Figure 6: Screenshot of toggling the rgb values, only leaving green

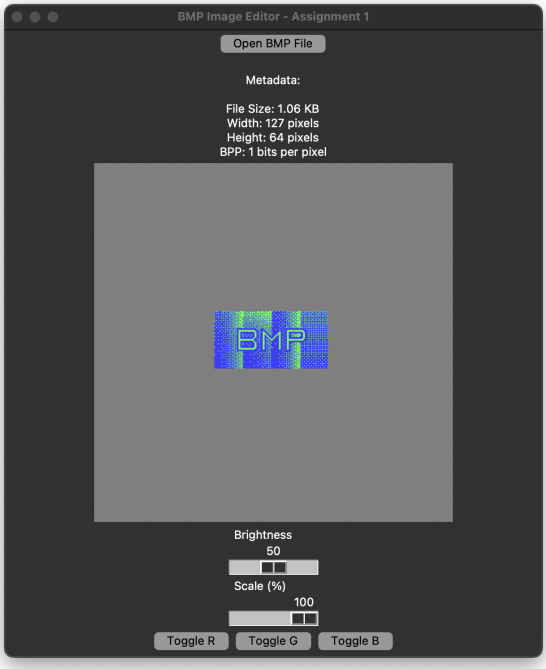


Figure 7: Loading the 1bpp image at 100% scaling

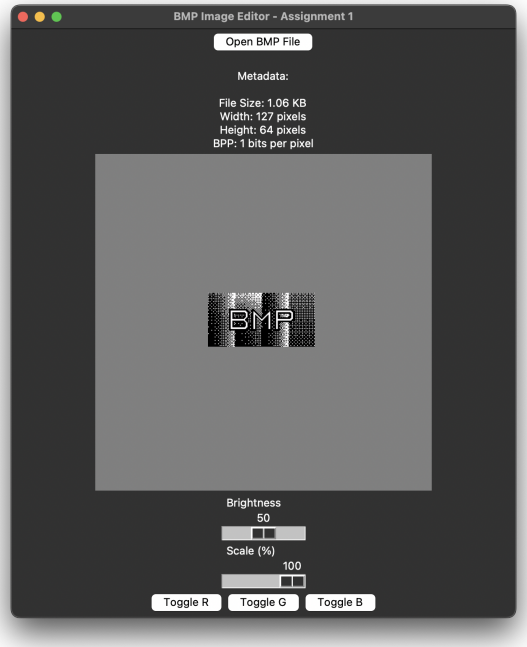


Figure 8: Loading the grayscaled 1bpp image at 100% scaling

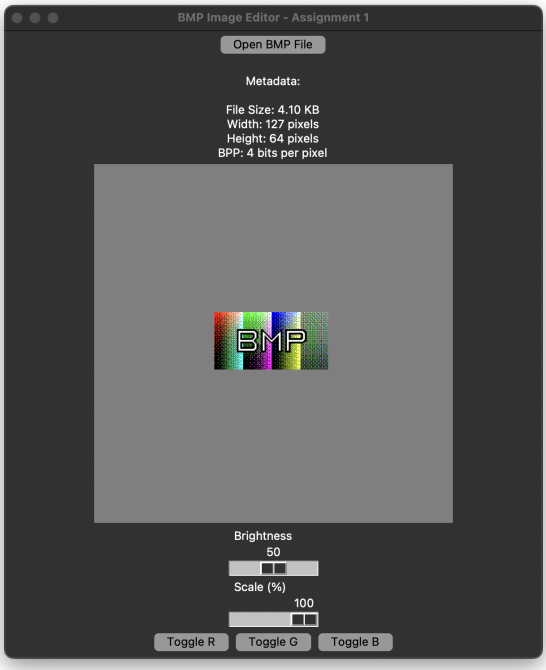


Figure 9: Loading the 4bpp image at 100% scaling

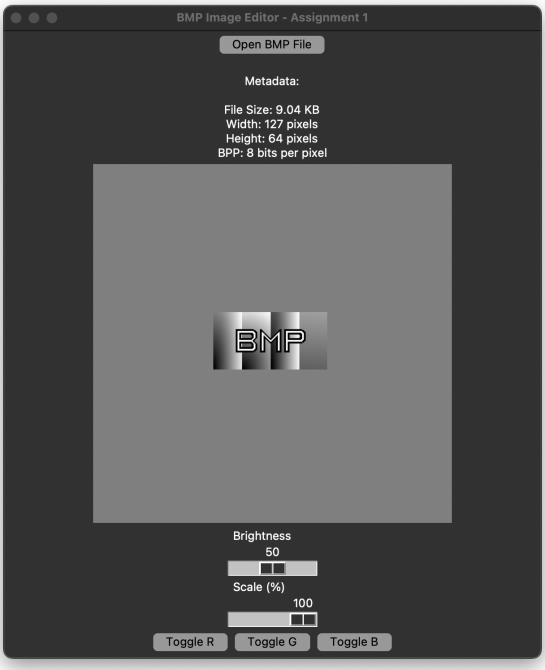


Figure 10: Loading the 8bpp image at 100% scaling