CMPT 365 (E100): Multimedia Systems

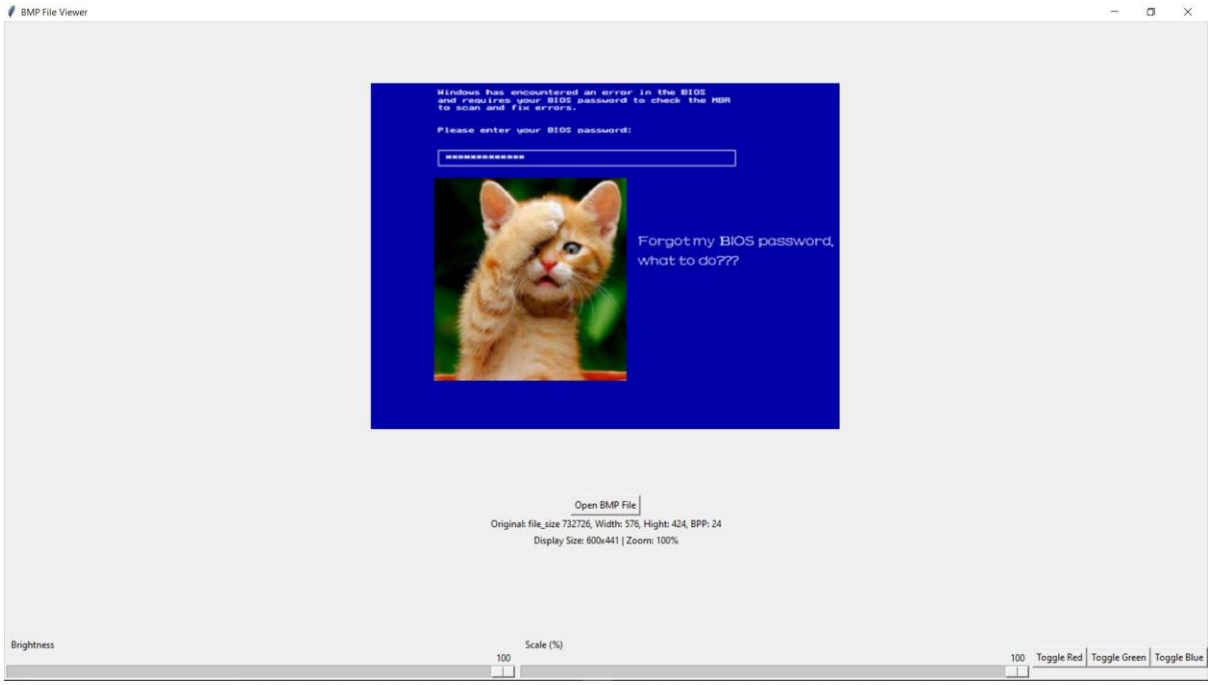
Programming Assignment 1: BMP File Viewer

Utsav Patel

301573109

usp@sfu.ca

Program Screenshots:



The above Screenshot is the image of the GUI displaying one of the given sample BMP images. Just below the displayed image there is a “Open BMP File” button that allows us to navigate throw the file explorer and open only a BMP File.

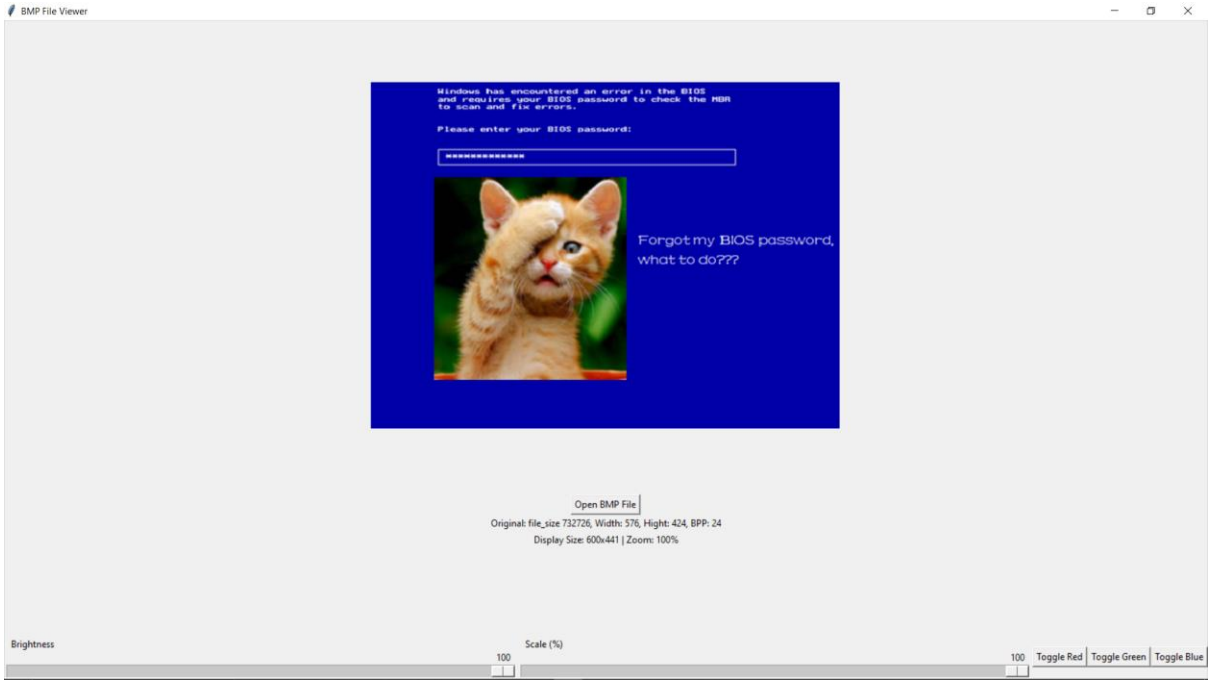
The opened image would be displayed under the canvas, which helps us to see the small-scale images in larger ratio. As we have been provided with the simple images as small as 127x64 which would be displayed at 600x305 cause of canvas.

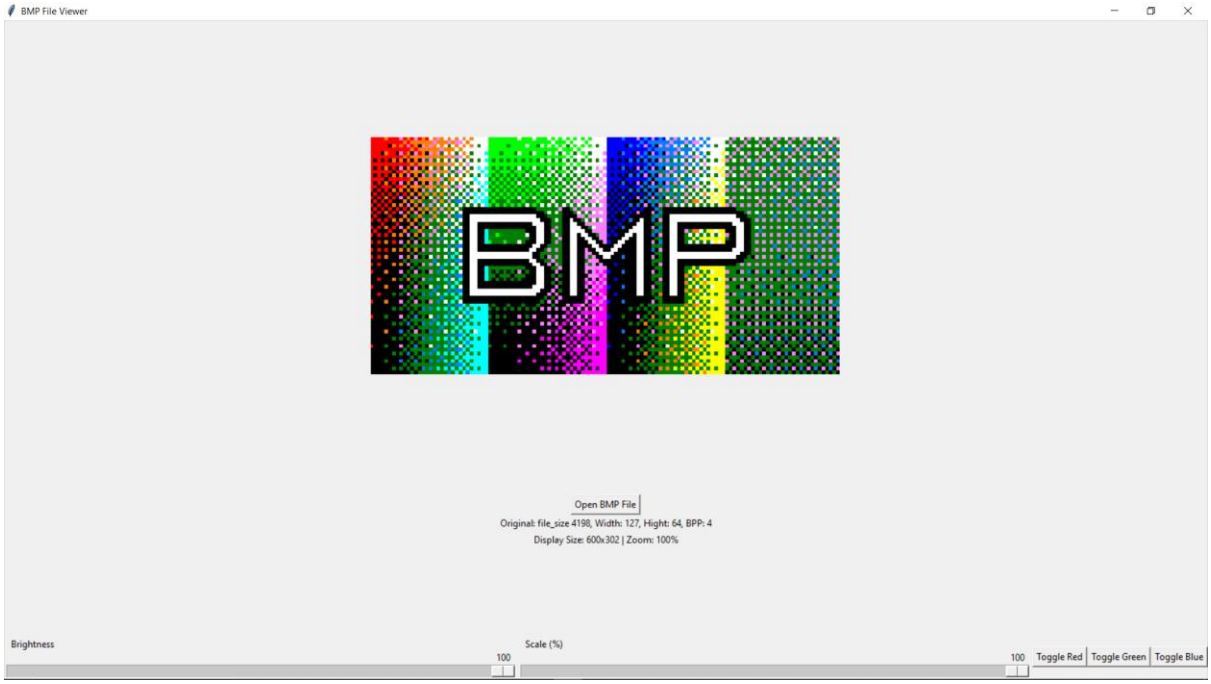
The GUI displays information about the selected image, including the original file size, width, height, and bits per pixel (BPP).

Next, I have the Display size that will show the real time change in the image size while scaling the image with the giver bar at the very bottom alongside with the scale percentage.

At the end there is also a Brightness adjustment bar that would allow user to adjust the brightness of the displaying image. Last but now the lease

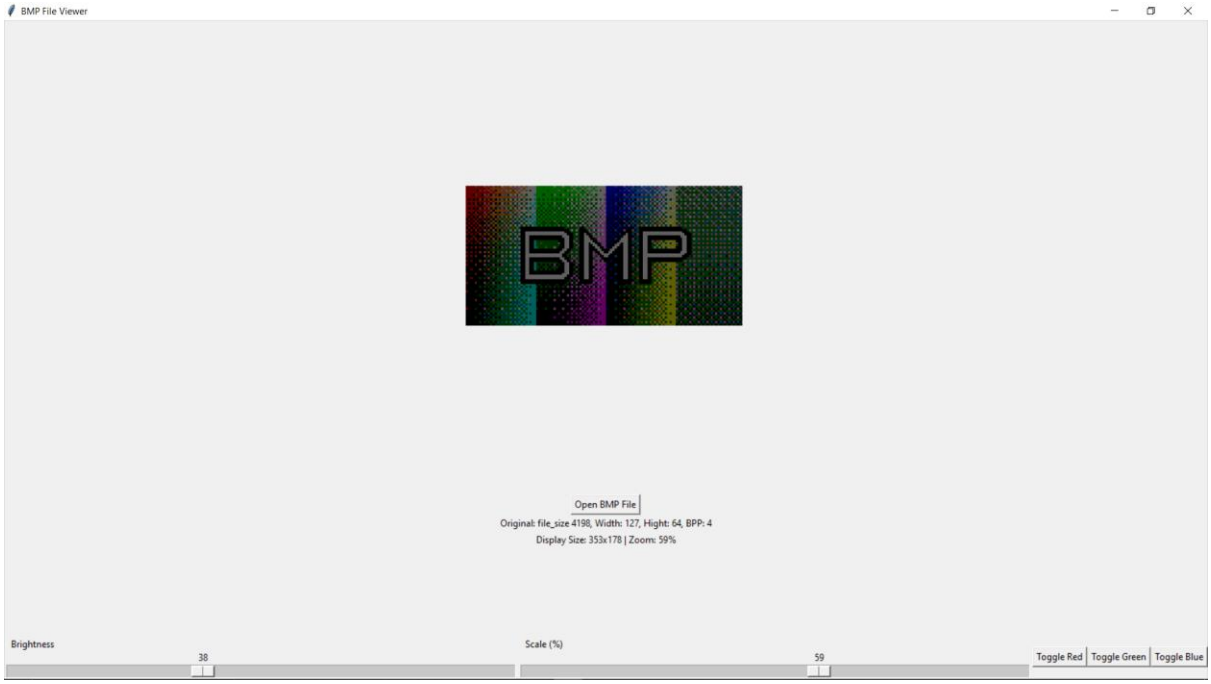
The GUI provides buttons for toggling red, green, and blue channels.



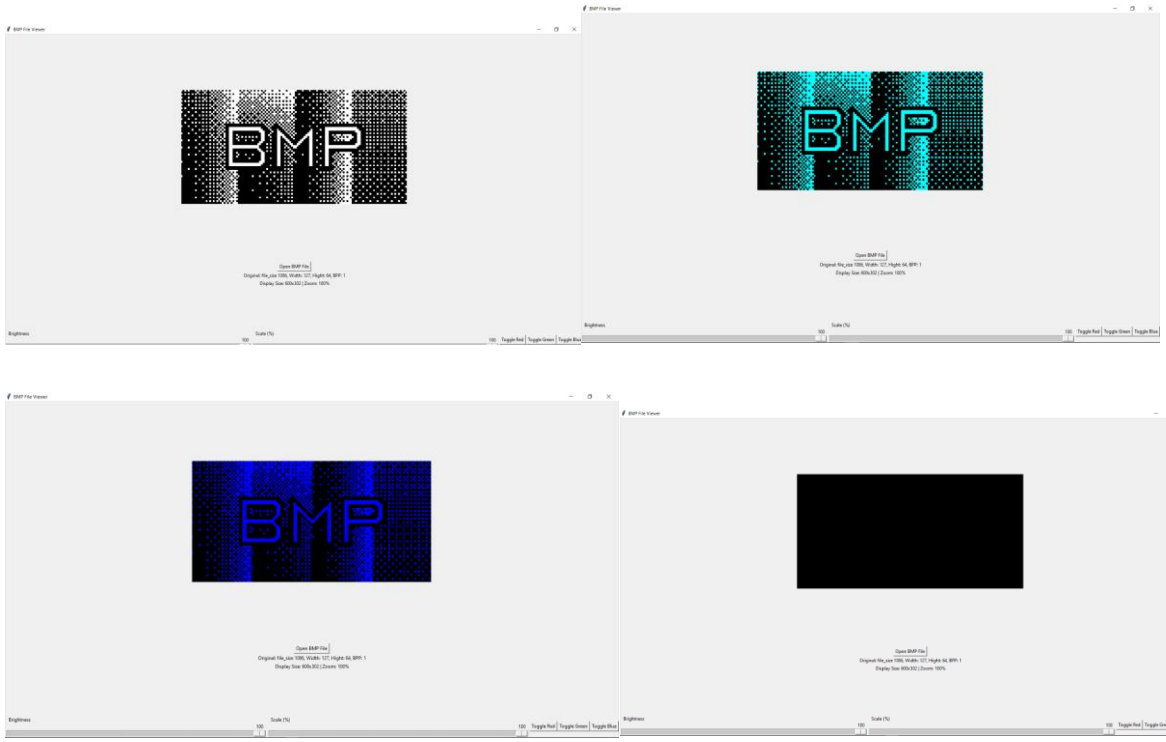


The needed metadata display from the program requirements can be seen in these two screenshots. Crucially, it accurately writes the BPP for every picture. 1, 4, 8 and 24.

As we can see for the second screenshot. the display size of the image is greater than the original size. That create the generous size of image we can observe.



This screenshot demonstrates how the sidebars for brightness and scale are operating correctly. It has been adjusted to 59% zoom and 38 brightness.



The first image is the original; the second is after the red has been toggled; the third is after the green has been toggled; and the final one is after the blue has been toggled, turning the image black.

Essential Code Explanation:

1. open_file

Opens a BMP file using a file dialog and reads its contents.

```
def open_file(self):
    file_path = filedialog.askopenfilename(filetypes=[("BMP files", "*.bmp")])
    if file_path:
        try:
            with open(file_path, "rb") as f:
                bmp_bytes = f.read()
                self.validate_bmp(bmp_bytes)
                self.parse_bmp(bmp_bytes)
        except Exception as e:
            messagebox.showerror("Error", f"Failed to load file:\n{str(e)}")
```

It uses filedialog.askopenfilename to prompt the user for a file. The selected file is read in binary mode, and the validate_bmp and parse_bmp functions are called to check and extract image data.

2. validate_bmp

Validates if the selected file is a valid BMP image.

```
def validate_bmp(self, bmp_bytes):
    if len(bmp_bytes) < 54:
        raise ValueError("Not a valid BMP file (header too small)")
    if bmp_bytes[0] != ord('B') or bmp_bytes[1] != ord('M'):
        raise ValueError("Not a valid BMP file (missing BM header)")
```

Checks if the file has a minimum header size (54 bytes) and if the first two bytes match the ASCII characters 'BM' (which signifies a BMP file).

3. parse_bmp

Extracts metadata from the BMP file header and updates the UI to display details.

```
def parse_bmp(self, bmp_bytes):
    self.original_file_size = int.from_bytes(bmp_bytes[2:6], 'little')
    self.original_width = int.from_bytes(bmp_bytes[18:22], 'little')
    self.original_height = int.from_bytes(bmp_bytes[22:26], 'little')
    bpp = int.from_bytes(bmp_bytes[28:30], 'little')

    self.metadata_label.config(
        text=f"Original: file_size {self.original_file_size}, Width: {self.original_width}, Hight: {self.original_height}, BPP: {bpp}"
    )
    self.display_image(bmp_bytes)
```

Reads specific byte ranges from the BMP header to determine file size, image width, height, and bits per pixel (BPP). These values are displayed in a label widget.

4. display_image

Processes BMP pixel data and prepares it for rendering.

```
def display_image(self, bmp_bytes):
    bpp = int.from_bytes(bmp_bytes[28:30], 'little')
    width, height = self.original_width, self.original_height

    if bpp == 24:
        self.image_array = self.parse_24bit(bmp_bytes, width, height)
    elif bpp == 1:
        self.image_array = self.parse_1bit(bmp_bytes, width, height)
    elif bpp == 4:
        self.image_array = self.parse_4bit(bmp_bytes, width, height)
    elif bpp == 8:
        self.image_array = self.parse_8bit(bmp_bytes, width, height)
    else:
        raise ValueError(f"Unsupported BPP: {bpp}")

    # Calculating initial scale to fit in square
    self.base_scale = min(self.canvas_size/width, self.canvas_size/height) # The minimum scale factor to f
    self.update_image()
```

Depending on the BPP value, it calls specific parsing functions (parse_24bit, parse_1bit, etc.) to extract pixel information and store it in an image_array. The scaling factor for fitting the image in the canvas is also calculated.

5. parse_24bit

Parses a 24-bit BMP file and extracts RGB pixel values.

```
def parse_24bit(self, bmp_bytes, width, height):
    pixel_data = bmp_bytes[54:]
    row_size = (width * 3 + 3) & ~3
    image_array = np.zeros((height, width, 3), dtype=np.uint8)

    for y in range(height):
        for x in range(width):
            offset = y * row_size + x * 3
            if offset + 2 < len(pixel_data):
                b, g, r = pixel_data[offset:offset + 3]
                image_array[y, x] = [r, g, b]
    return image_array
```

Iterates over the pixel data in groups of three bytes (representing B, G, R). The data is stored in a NumPy array in an (H x W x 3) format.

6. parse_1bit

Parses a 1-bit BMP file and extracts black-and-white pixel values.

```python
def parse_1bit(self, bmp_bytes, width, height):
    # Offset to the pixel data
    pixel_data_offset = int.from_bytes(bmp_bytes[10:14], 'little')
    pixel_data = bmp_bytes[pixel_data_offset:]

    color_table = bmp_bytes[54:62]
    colors = [
        (color_table[i + 2], color_table[i + 1], color_table[i])
        for i in range(0, len(color_table), 4)
    ]

    row_size = (width + 31) // 32 * 4
    image_array = np.zeros((height, width, 3), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            byte_index = row_start + (x // 8)
            bit_index = 7 - (x % 8)
            if byte_index < len(pixel_data):
                pixel_value = (pixel_data[byte_index] >> bit_index) & 1
                image_array[y, x] = colors[pixel_value]

    return image_array
```

Reads each byte and extracts bits using bitwise operations. A color table is used to map each pixel to its corresponding grayscale value.

7. parse_4bit

Parses a 4-bit BMP file and extracts color-indexed pixel values.

```python
def parse_4bit(self, bmp_bytes, width, height):
    # Offset to the pixel data
    pixel_data_offset = int.from_bytes(bmp_bytes[10:14], 'little')
    pixel_data = bmp_bytes[pixel_data_offset:]

    color_table_size = 16 * 4
    color_table = bmp_bytes[54:54 + color_table_size]
    colors = [
        (color_table[i + 2], color_table[i + 1], color_table[i])
        for i in range(0, len(color_table), 4)
    ]

    row_size = (width + 7) // 8 * 4
    image_array = np.zeros((height, width, 3), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            byte_index = row_start + (x // 2)
            if byte_index < len(pixel_data):
                if x % 2 == 0:
                    pixel_value = (pixel_data[byte_index] >> 4) & 0x0F
                else:
                    pixel_value = pixel_data[byte_index] & 0x0F
                image_array[y, x] = colors[pixel_value]

    return image_array
```

Uses a color table to convert 4-bit indices into 24-bit RGB values. Every byte contains two pixels, so bitwise operations are used to extract pixel values.

8. parse_8bit

Parses an 8-bit BMP file and extracts grayscale pixel values.

```
def parse_8bit(self, bmp_bytes, width, height):
    # Offset to the pixel data
    pixel_data_offset = int.from_bytes(bmp_bytes[10:14], 'little')
    pixel_data = bmp_bytes[pixel_data_offset:]

    color_table_size = 256 * 4
    color_table = bmp_bytes[54:54 + color_table_size]
    colors = [
        (color_table[i + 2], color_table[i + 1], color_table[i])
        for i in range(0, len(color_table), 4)
    ]

    row_size = (width + 3) // 4 * 4
    image_array = np.zeros((height, width, 3), dtype=np.uint8)

    for y in range(height):
        row_start = y * row_size
        for x in range(width):
            byte_index = row_start + x
            if byte_index < len(pixel_data):
                pixel_value = pixel_data[byte_index]
                image_array[y, x] = colors[pixel_value]

    return image_array
```

Uses a 256-color lookup table where each pixel corresponds to an entry in the palette, translating indexed colors to RGB values.

9. update_image

Applies scaling and brightness adjustments to the image and updates the display.

```
def update_image(self, event=None):
    if not hasattr(self, 'image_array'):
        return

    user_scale = self.scale_slider.get() / 100
    combined_scale = self.base_scale * user_scale

    new_width = int(self.original_width * combined_scale)
    new_height = int(self.original_height * combined_scale)

    # Applying brightness
    adjusted = np.clip(self.image_array * (self.brightness_slider.get()/100), 0, 255)

    scaled = np.zeros((new_height, new_width, 3), dtype=np.uint8) # Creating a new array to store the scaled image
    for y in range(new_height):
        for x in range(new_width):
            orig_y = min(int(y/combined_scale), self.original_height-1)
            orig_x = min(int(x/combined_scale), self.original_width-1)
            scaled[y, x] = adjusted[orig_y, orig_x]

    self.photo_image = tk.PhotoImage(width=new_width, height=new_height)
    for y in range(new_height):
        for x in range(new_width):
            r, g, b = scaled[y, x]
            r = r if self.show_red else 0
            g = g if self.show_green else 0
            b = b if self.show_blue else 0
            self.photo_image.put(f"#{r:02x}{g:02x}{b:02x}", (x, new_height - 1 - y))

    # Clearing canvas and center image
    self.canvas.delete("all")
    x_pos = (self.canvas_size - new_width) // 2
    y_pos = (self.canvas_size - new_height) // 2
    self.canvas.create_image(x_pos, y_pos, anchor=tk.NW, image=self.photo_image)

    self.updated_metadata_label.config(
        text=f"Display Size: {new_width}x{new_height} | Zoom: {self.scale_slider.get()}%"
    )
```

Computes a scaling factor based on the user's input and adjusts brightness by multiplying pixel values by the brightness factor. The transformed image is then drawn on the Tkinter canvas.

10. toggle_red, toggle_green, toggle_blue

Toggles the visibility of the respective color channels in the displayed image.

```
def toggle_red(self):
    self.show_red = not self.show_red
    self.update_image()

def toggle_green(self):
    self.show_green = not self.show_green
    self.update_image()

def toggle_blue(self):
    self.show_blue = not self.show_blue
    self.update_image()
```

Updates boolean flags (show_red, show_green, show_blue) and refreshes the displayed image, setting color values to zero where necessary.