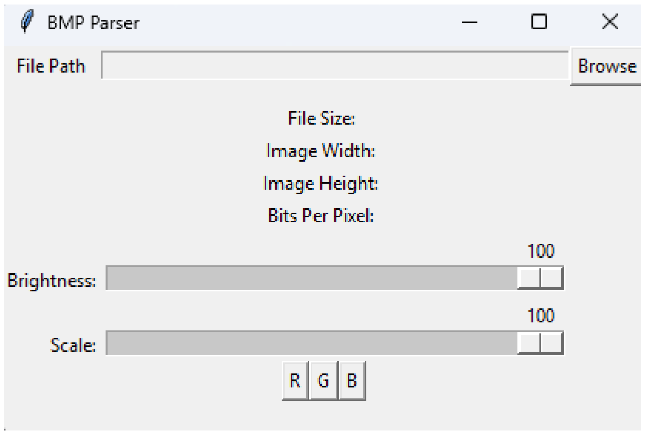
BMP File Parser: Report

Andy Boyuan Liu

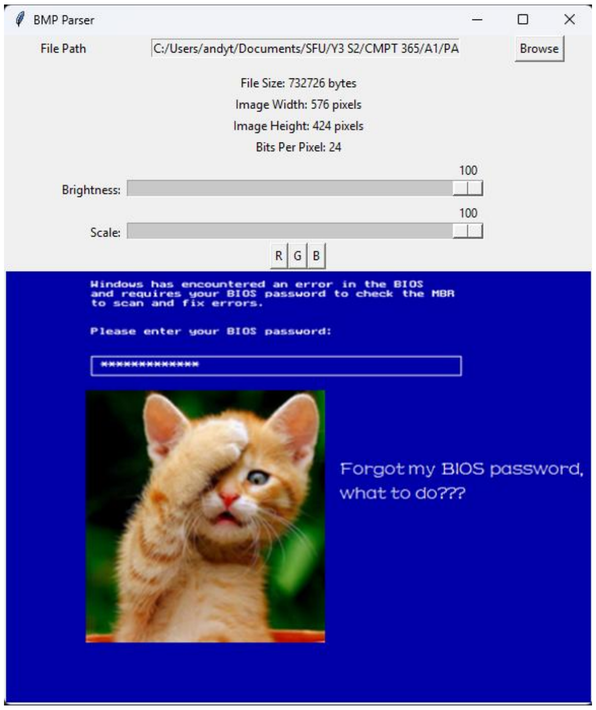
Program Screenshots:

- Program on first open
    - The UI is simple and clean, information is easily accessible and clear to the user.



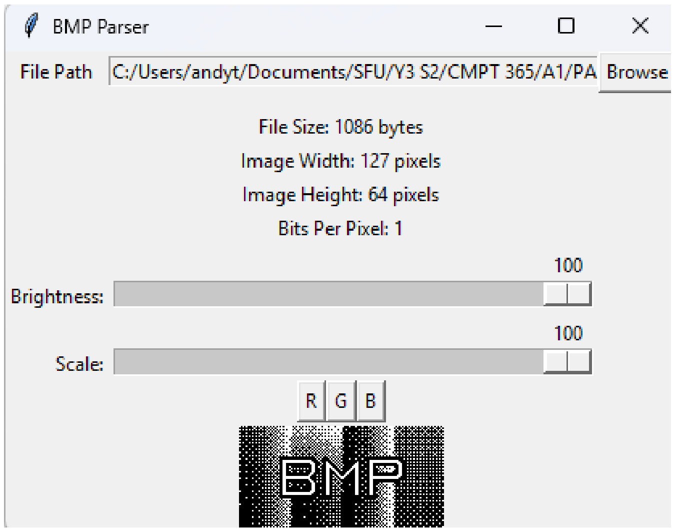
File Size:
Image Width:
Image Height:
Bits Per Pixel:

- Program displaying the sample images


BMP Parser
File Path
Browse
Brightness:
100
Scale:
100
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

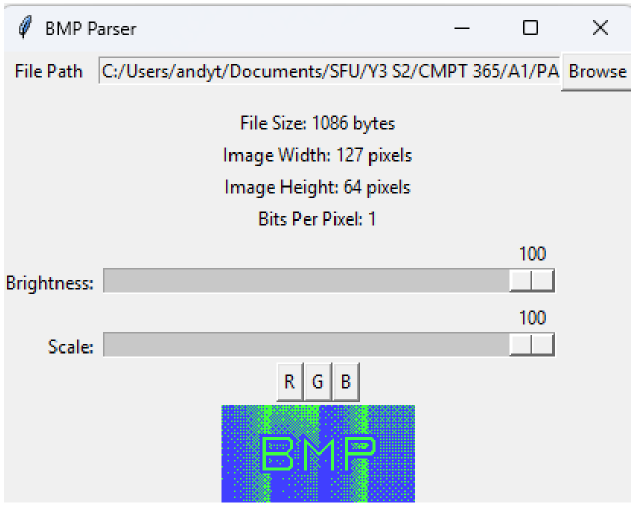
C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24



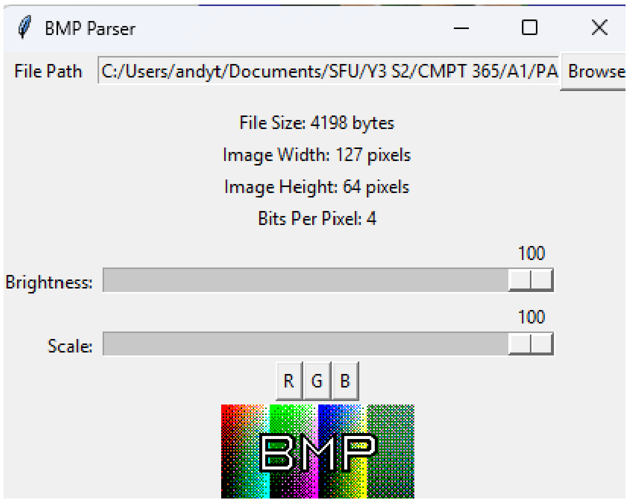
C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA

File Size: 1086 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 1



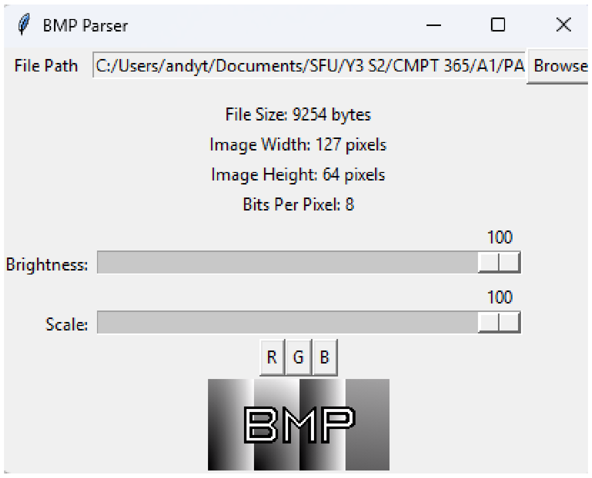
C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA

File Size: 1086 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 1



C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA

File Size: 4198 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 4


BMP Parser

File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 9254 bytes

Image Width: 127 pixels

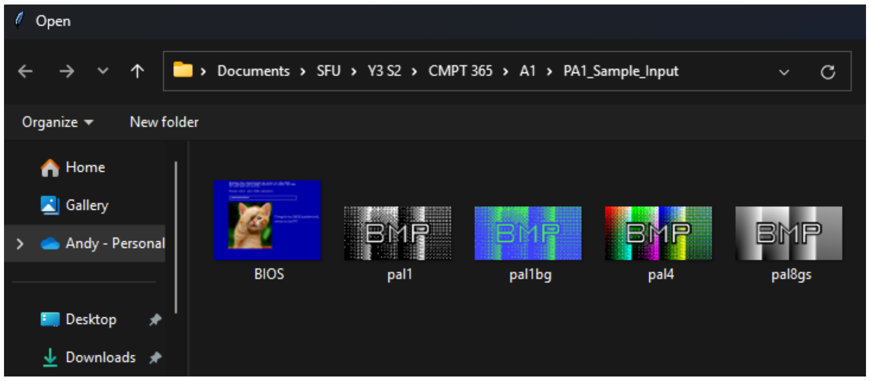
Image Height: 64 pixels

Bits Per Pixel: 8

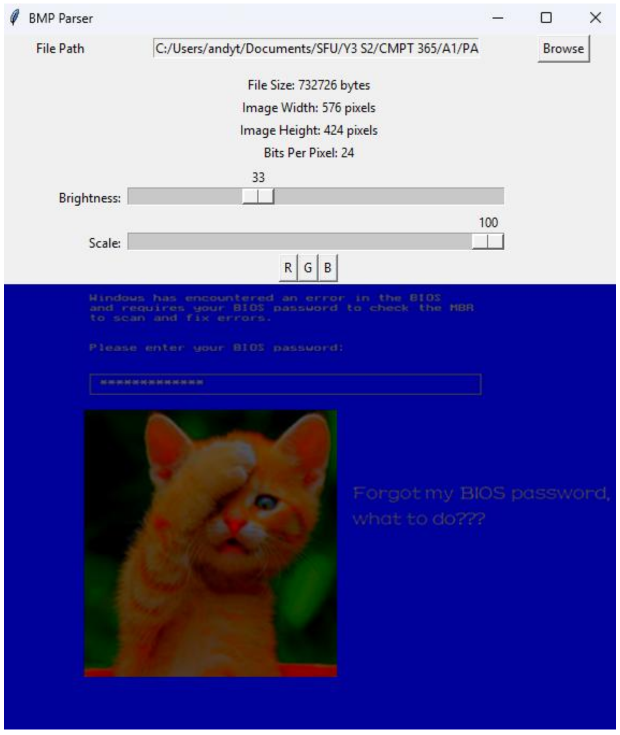
Brightness: 100

Scale: 100

- Requirements met
  - From the above screenshots you can clearly see the metadata labels meet all the requirements and correctly show requested metadata
  - There is a browse button that opens a file explorer prompt for the user to provide a bmp file



    - The prompt only accepts bmp files
  - The bmp images are reasonable sizes and can be easily viewed
  - Brightness sliders work from 0-100%


BMP Parser

File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 732726 bytes

Image Width: 576 pixels

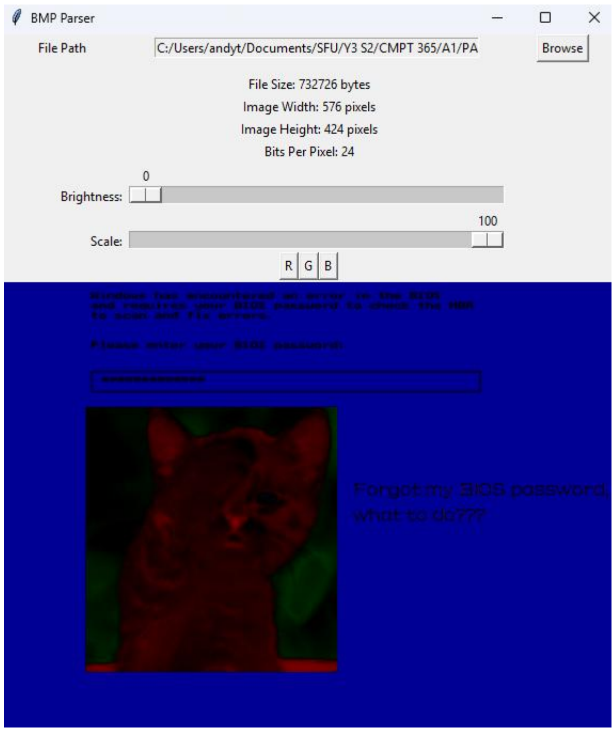
Image Height: 424 pixels

Bits Per Pixel: 24

Brightness: 33

Scale: 100

R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
****
Forgot my BIOS password, what to do???


BMP Parser

File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 732726 bytes

Image Width: 576 pixels

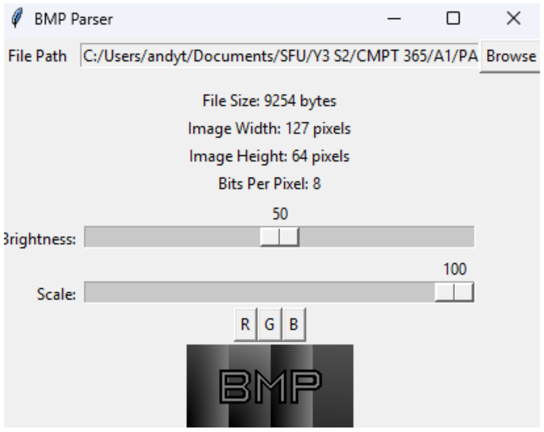
Image Height: 424 pixels

Bits Per Pixel: 24

Brightness: 0

Scale: 100

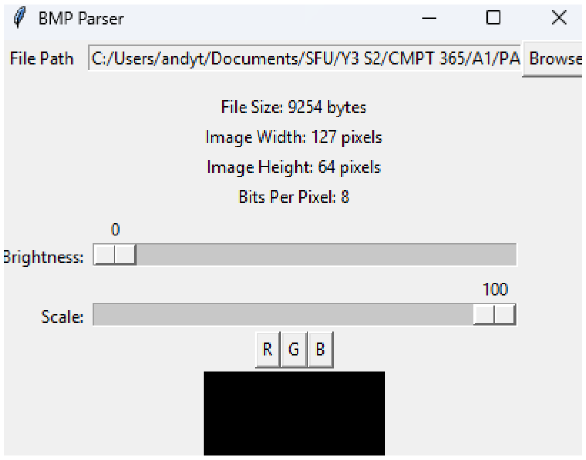
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???



File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 9254 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 8

Brightness: 50
Scale: 100

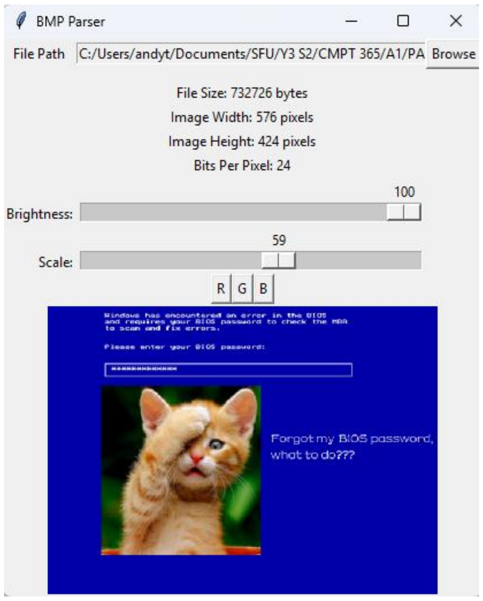


File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 9254 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 8

Brightness: 0
Scale: 100

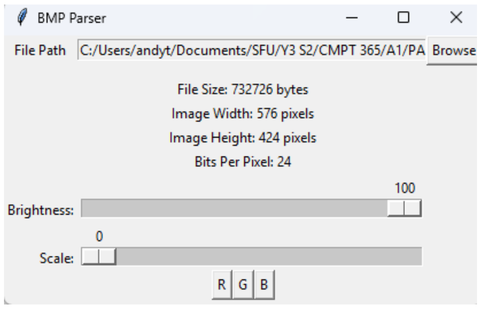
- Scale sliders work from 0-100%


BMP Parser
R G B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

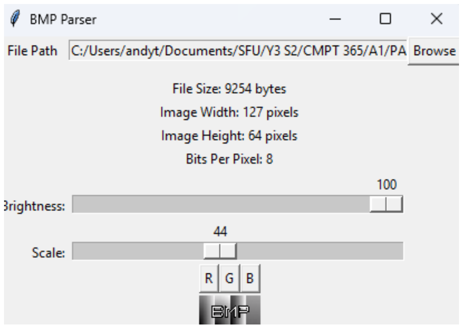
Brightness: 100
Scale: 59



File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100
Scale: 0

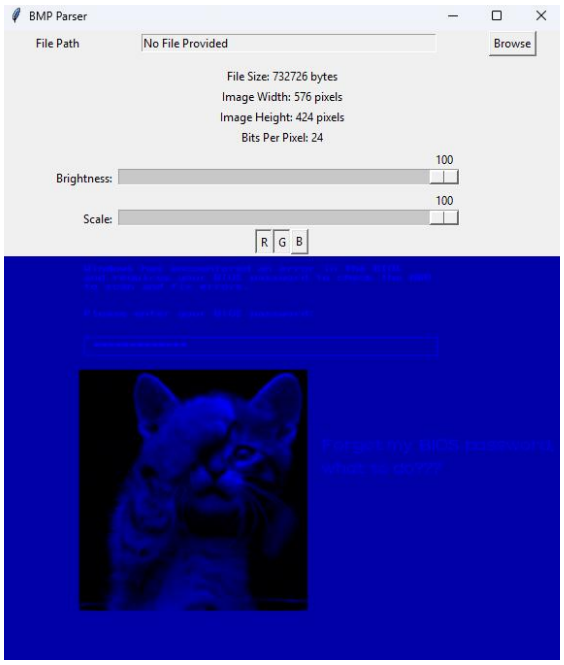


File Path C:/Users/andyt/Documents/SFU/Y3 S2/CMPT 365/A1/PA Browse

File Size: 9254 bytes
Image Width: 127 pixels
Image Height: 64 pixels
Bits Per Pixel: 8

Brightness: 100
Scale: 44

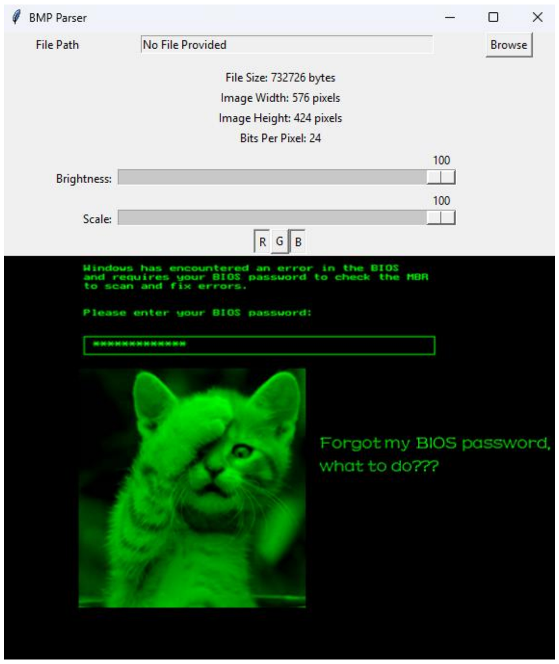
- RGB buttons to toggle the channels



File Path No File Provided Browse

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

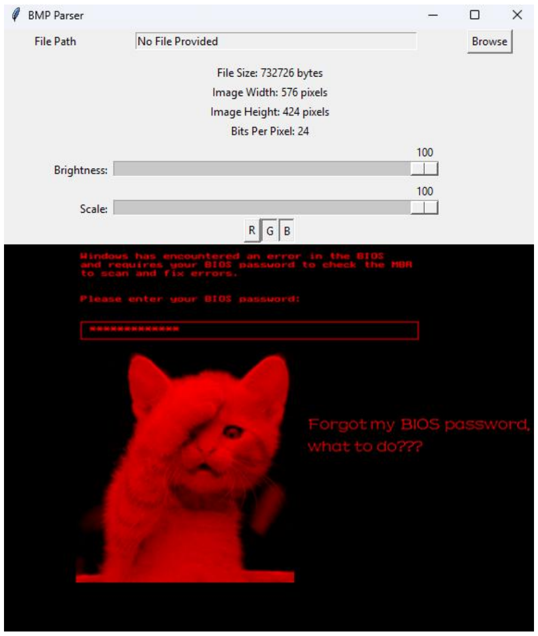
Brightness: 100
Scale: 100


BMP Parser
R G B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path No File Provided Browse

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100
Scale: 100

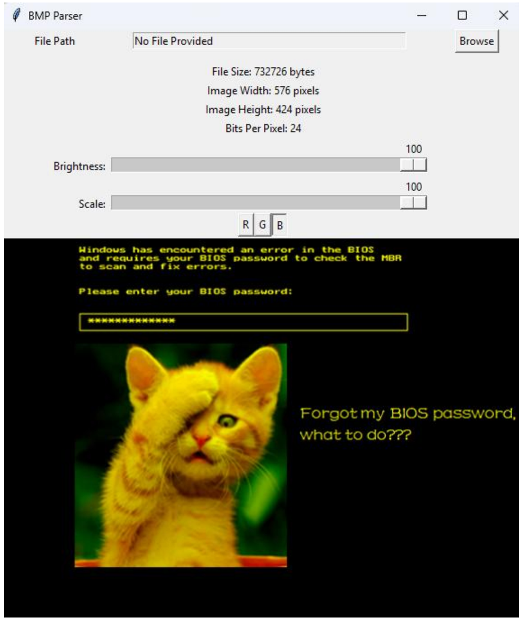

BMP Parser
Browse
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path: No File Provided

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100

Scale: 100

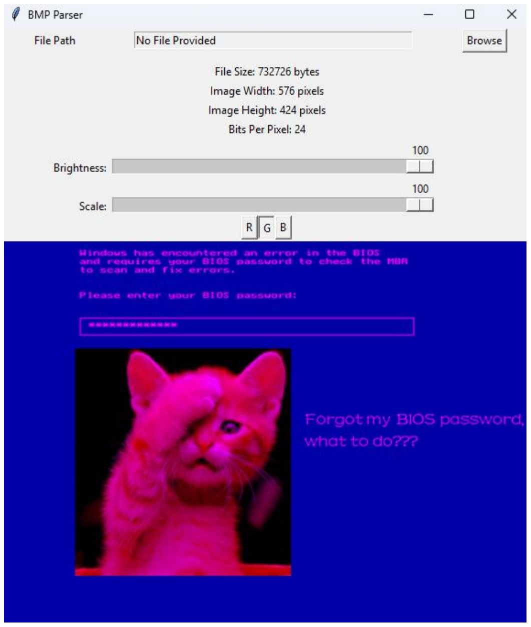

BMP Parser
Browse
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path: No File Provided

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100

Scale: 100

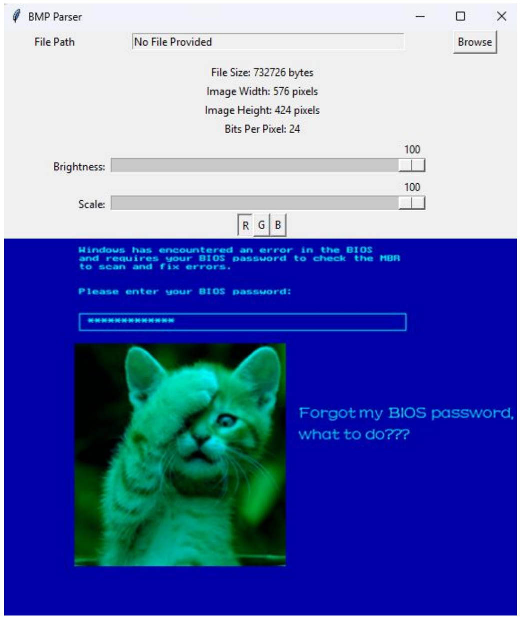

BMP Parser
Browse
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path: No File Provided

File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100

Scale: 100


BMP Parser
Browse
R
G
B
Windows has encountered an error in the BIOS and requires your BIOS password to check the MBR to scan and fix errors.
Please enter your BIOS password:
Forgot my BIOS password, what to do???

File Path: No File Provided

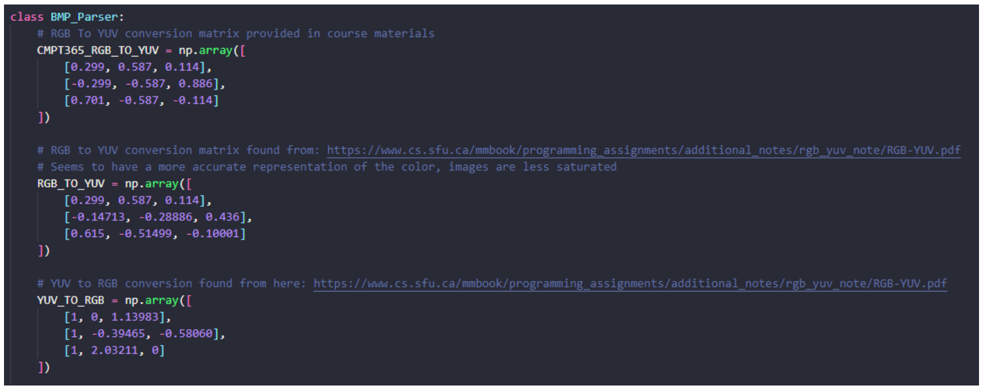
File Size: 732726 bytes
Image Width: 576 pixels
Image Height: 424 pixels
Bits Per Pixel: 24

Brightness: 100

Scale: 100

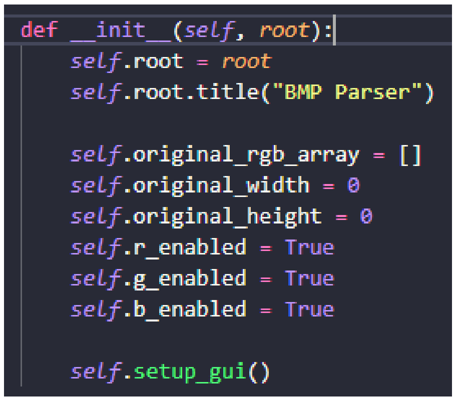
Essential Code Explanation

- Program was written with OOP
  - Did this for organizational purposes and to make the code more modular. This might help in the future when we add new additions to the program
- The class is BMP_Parser, below are the class attributes



```python
class BMP_Parser:
    # RGB To YUV conversion matrix provided in course materials
    CMPT365_RGB_TO_YUV = np.array([
        [0.299, 0.587, 0.114],
        [-0.299, -0.587, 0.886],
        [0.701, -0.587, -0.114]
    ])

    # RGB to YUV conversion matrix found from: https://www.cs.sfu.ca/mmbook/programming_assignments/additional_notes/rgb_yuv_note/RGB-YUV.pdf
    # Seems to have a more accurate representation of the color, images are less saturated
    RGB_TO_YUV = np.array([
        [0.299, 0.587, 0.114],
        [-0.14713, -0.28886, 0.436],
        [0.615, -0.51499, -0.10001]
    ])

    # YUV to RGB conversion found from here: https://www.cs.sfu.ca/mmbook/programming_assignments/additional_notes/rgb_yuv_note/RGB-YUV.pdf
    YUV_TO_RGB = np.array([
        [1, 0, 1.13983],
        [1, -0.39465, -0.58060],
        [1, 2.03211, 0]
    ])
```

-

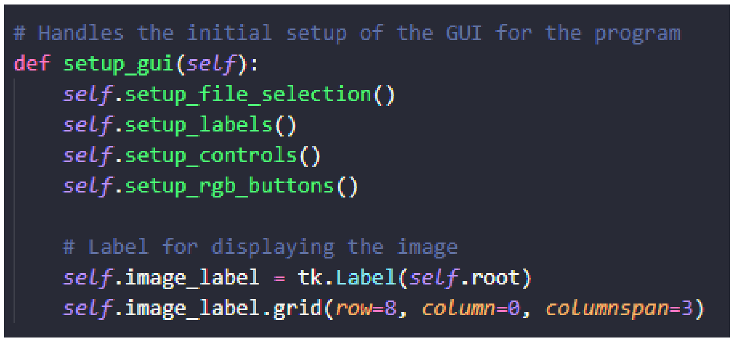
- I chose different RGB to YUV (and vice versa) conversion matrices because they seemed to give a more realistic representation of the colors in the images.
  - Here is a link to the resource (it is from SFU): https://www.cs.sfu.ca/mmbook/programming_assignments/additional_notes/rgb_yuv_note/RGB-YUV.pdf
- An instance of the class takes a tkinter root object and sets up the GUI
  - 

```
def __init__(self, root):
    self.root = root
    self.root.title("BMP Parser")

    self.original_rgb_array = []
    self.original_width = 0
    self.original_height = 0
    self.r_enabled = True
    self.g_enabled = True
    self.b_enabled = True

    self.setup_gui()
```

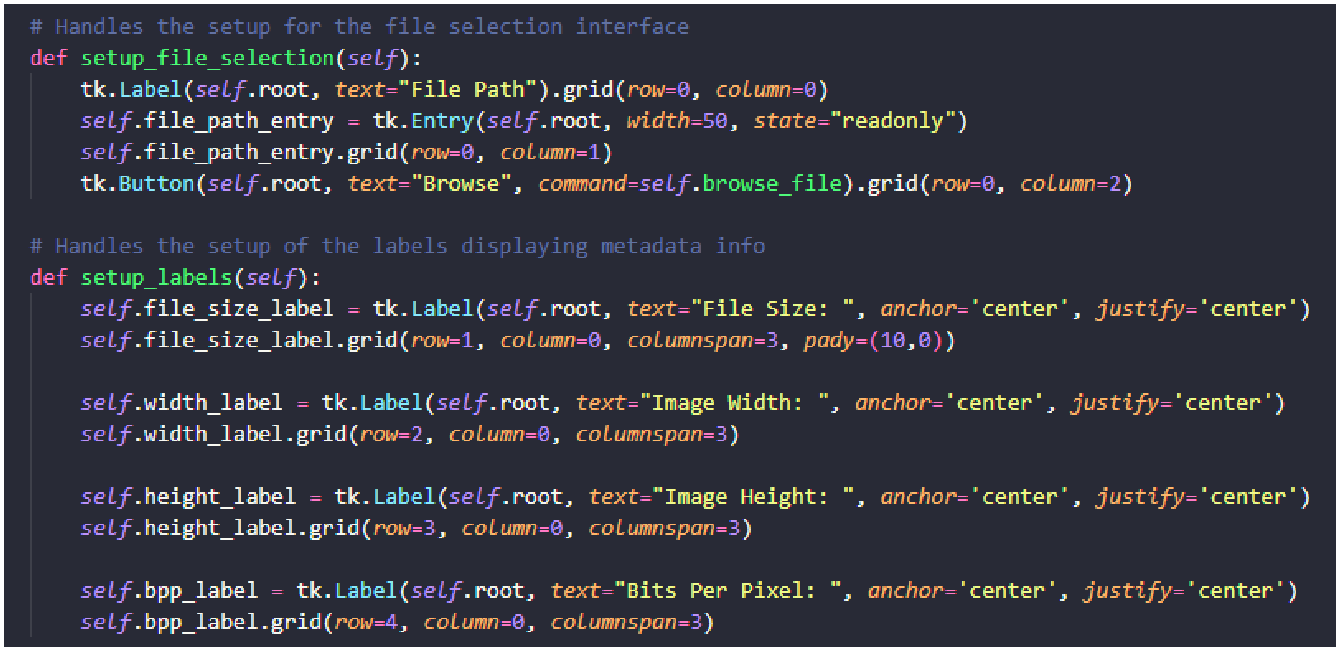
Function Explanations:

setup_gui(self):

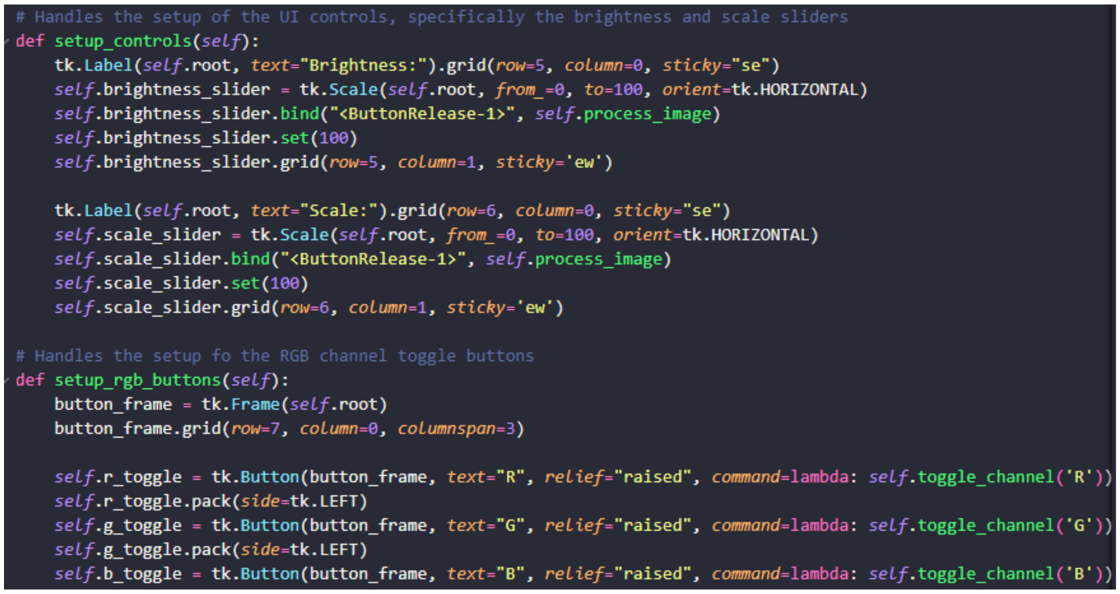
The function calls helper functions each responsible with setting up a section of the UI. The UI is constructed with tkinter widgets. The code for each of these functions is pretty simple as they only handle the UI setup.



```
# Handles the initial setup of the GUI for the program
def setup_gui(self):
    self.setup_file_selection()
    self.setup_labels()
    self.setup_controls()
    self.setup_rgb_buttons()

    # Label for displaying the image
    self.image_label = tk.Label(self.root)
    self.image_label.grid(row=8, column=0, columnspan=3)
```



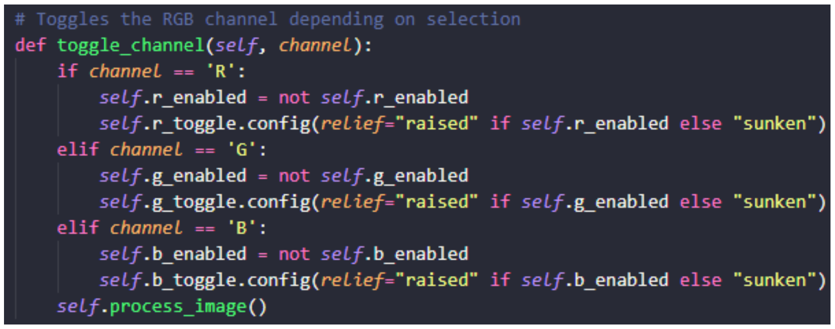
```
# Handles the setup for the file selection interface
def setup_file_selection(self):
    tk.Label(self.root, text="File Path").grid(row=0, column=0)
    self.file_path_entry = tk.Entry(self.root, width=50, state="readonly")
    self.file_path_entry.grid(row=0, column=1)
    tk.Button(self.root, text="Browse", command=self.browse_file).grid(row=0, column=2)

# Handles the setup of the labels displaying metadata info
def setup_labels(self):
    self.file_size_label = tk.Label(self.root, text="File Size: ", anchor='center', justify='center')
    self.file_size_label.grid(row=1, column=0, columnspan=3, pady=(10,0))

    self.width_label = tk.Label(self.root, text="Image Width: ", anchor='center', justify='center')
    self.width_label.grid(row=2, column=0, columnspan=3)

    self.height_label = tk.Label(self.root, text="Image Height: ", anchor='center', justify='center')
    self.height_label.grid(row=3, column=0, columnspan=3)

    self.bpp_label = tk.Label(self.root, text="Bits Per Pixel: ", anchor='center', justify='center')
    self.bpp_label.grid(row=4, column=0, columnspan=3)
```



```
# Handles the setup of the UI controls, specifically the brightness and scale sliders
def setup_controls(self):
    tk.Label(self.root, text="Brightness:").grid(row=5, column=0, sticky="se")
    self.brightness_slider = tk.Scale(self.root, from_=0, to=100, orient=tk.HORIZONTAL)
    self.brightness_slider.bind("<ButtonRelease-1>", self.process_image)
    self.brightness_slider.set(100)
    self.brightness_slider.grid(row=5, column=1, sticky='ew')

    tk.Label(self.root, text="Scale:").grid(row=6, column=0, sticky="se")
    self.scale_slider = tk.Scale(self.root, from_=0, to=100, orient=tk.HORIZONTAL)
    self.scale_slider.bind("<ButtonRelease-1>", self.process_image)
    self.scale_slider.set(100)
    self.scale_slider.grid(row=6, column=1, sticky='ew')

# Handles the setup fo the RGB channel toggle buttons
def setup_rgb_buttons(self):
    button_frame = tk.Frame(self.root)
    button_frame.grid(row=7, column=0, columnspan=3)

    self.r_toggle = tk.Button(button_frame, text="R", relief="raised", command=lambda: self.toggle_channel('R'))
    self.r_toggle.pack(side=tk.LEFT)
    self.g_toggle = tk.Button(button_frame, text="G", relief="raised", command=lambda: self.toggle_channel('G'))
    self.g_toggle.pack(side=tk.LEFT)
    self.b_toggle = tk.Button(button_frame, text="B", relief="raised", command=lambda: self.toggle_channel('B'))
```

toggle_channel(self, channel):

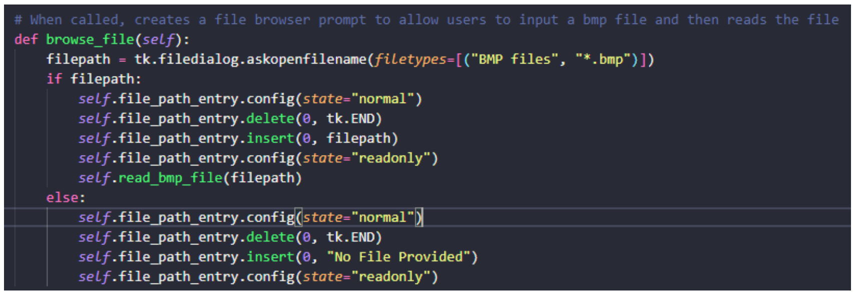
This function is called when the RGB channel toggle buttons are clicked. It updates the image with the new RGB settings and give a visual indication of whether a button has been toggled.



```
# Toggles the RGB channel depending on selection
def toggle_channel(self, channel):
    if channel == 'R':
        self.r_enabled = not self.r_enabled
        self.r_toggle.config(relief="raised" if self.r_enabled else "sunken")
    elif channel == 'G':
        self.g_enabled = not self.g_enabled
        self.g_toggle.config(relief="raised" if self.g_enabled else "sunken")
    elif channel == 'B':
        self.b_enabled = not self.b_enabled
        self.b_toggle.config(relief="raised" if self.b_enabled else "sunken")
    self.process_image()
```

browse_file(self):

This is called when the browse file button is pressed and simply opens a file explorer prompt to allow users to open a bmp file. If no file is given, it will set the file_path_entry (which normally would display the file path) to a message saying “No File Provided”

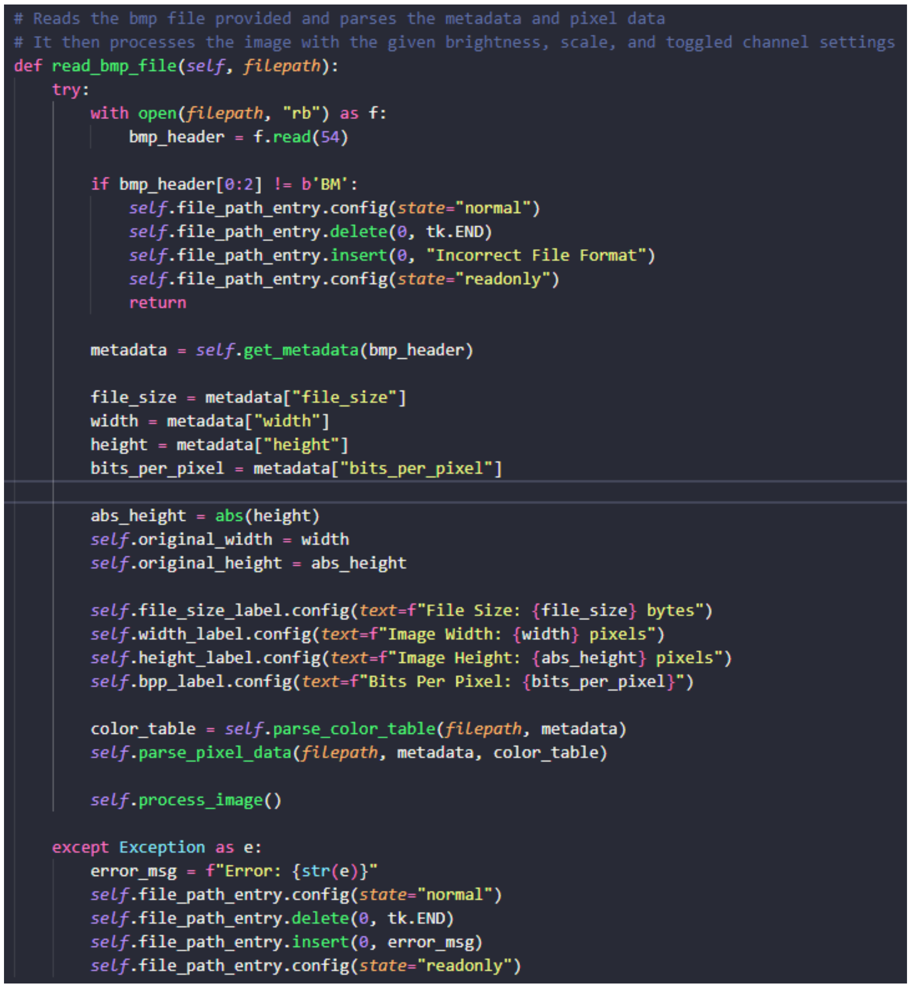


```
# When called, creates a file browser prompt to allow users to input a bmp file and then reads the file
def browse_file(self):
    filepath = tk.filedialog.askopenfilename(filetypes=[("BMP files", "*.bmp")])
    if filepath:
        self.file_path_entry.config(state="normal")
        self.file_path_entry.delete(0, tk.END)
        self.file_path_entry.insert(0, filepath)
        self.file_path_entry.config(state="readonly")
        self.read_bmp_file(filepath)
    else:
        self.file_path_entry.config(state="normal")
        self.file_path_entry.delete(0, tk.END)
        self.file_path_entry.insert(0, "No File Provided")
        self.file_path_entry.config(state="readonly")
```

read_bmp_file(self, filepath):

This function is provided a file path, determined by the user’s file selection, and reads the file. It first checks if the file type is valid by reading the signature in the header. If it is correct, it will continue reading the file. Following the first check, helper functions are called to get the metadata, parse the color table, and parse the pixel data.

This function also updates the metadata labels once it has retrieved the metadata.

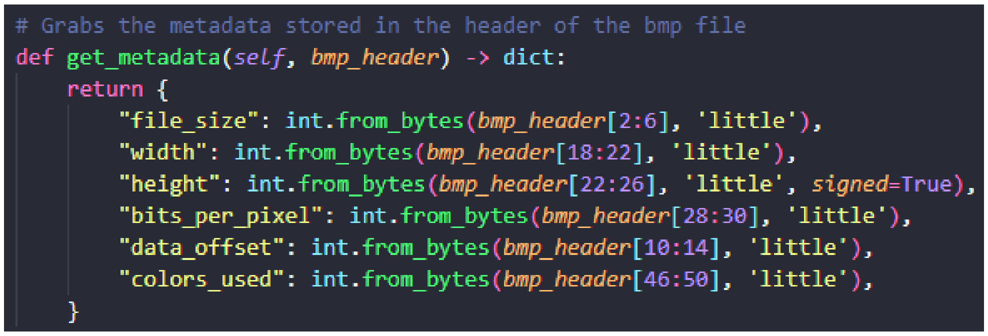


```python
# Reads the bmp file provided and parses the metadata and pixel data
# It then processes the image with the given brightness, scale, and toggled channel settings
def read_bmp_file(self, filepath):
    try:
        with open(filepath, "rb") as f:
            bmp_header = f.read(54)

        if bmp_header[0:2] != b'BM':
            self.file_path_entry.config(state="normal")
            self.file_path_entry.delete(0, tk.END)
            self.file_path_entry.insert(0, "Incorrect File Format")
            self.file_path_entry.config(state="readonly")
            return

        metadata = self.get_metadata(bmp_header)

        file_size = metadata["file_size"]
        width = metadata["width"]
        height = metadata["height"]
        bits_per_pixel = metadata["bits_per_pixel"]

        abs_height = abs(height)
        self.original_width = width
        self.original_height = abs_height

        self.file_size_label.config(text=f"File Size: {file_size} bytes")
        self.width_label.config(text=f"Image Width: {width} pixels")
        self.height_label.config(text=f"Image Height: {abs_height} pixels")
        self.bpp_label.config(text=f"Bits Per Pixel: {bits_per_pixel}")

        color_table = self.parse_color_table(filepath, metadata)
        self.parse_pixel_data(filepath, metadata, color_table)

        self.process_image()

    except Exception as e:
        error_msg = f"Error: {str(e)}"
        self.file_path_entry.config(state="normal")
        self.file_path_entry.delete(0, tk.END)
        self.file_path_entry.insert(0, error_msg)
        self.file_path_entry.config(state="readonly")
```

get_metadata(self, bmp_header) -> dict:

This function takes the bytes from the header of a bmp file and returns a dictionary with the values of each of the important metadata items. I thought this would be a clean and convenient way to handle getting the metadata since with this function, I can just save the metadata items in an array like so:

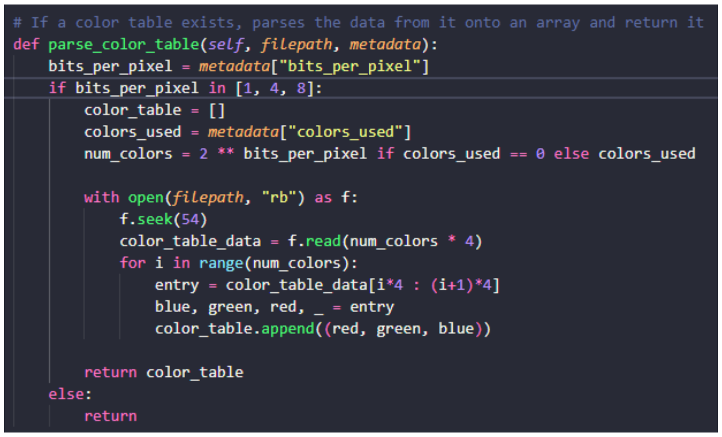
metadata = *self*.get_metadata(bmp_header) and then grab it using metadata[“width”] as an example



```python
# Grabs the metadata stored in the header of the bmp file
def get_metadata(self, bmp_header) -> dict:
    return {
        "file_size": int.from_bytes(bmp_header[2:6], 'little'),
        "width": int.from_bytes(bmp_header[18:22], 'little'),
        "height": int.from_bytes(bmp_header[22:26], 'little', signed=True),
        "bits_per_pixel": int.from_bytes(bmp_header[28:30], 'little'),
        "data_offset": int.from_bytes(bmp_header[10:14], 'little'),
        "colors_used": int.from_bytes(bmp_header[46:50], 'little'),
    }
```

parse_color_table(self, filepath, metadata):

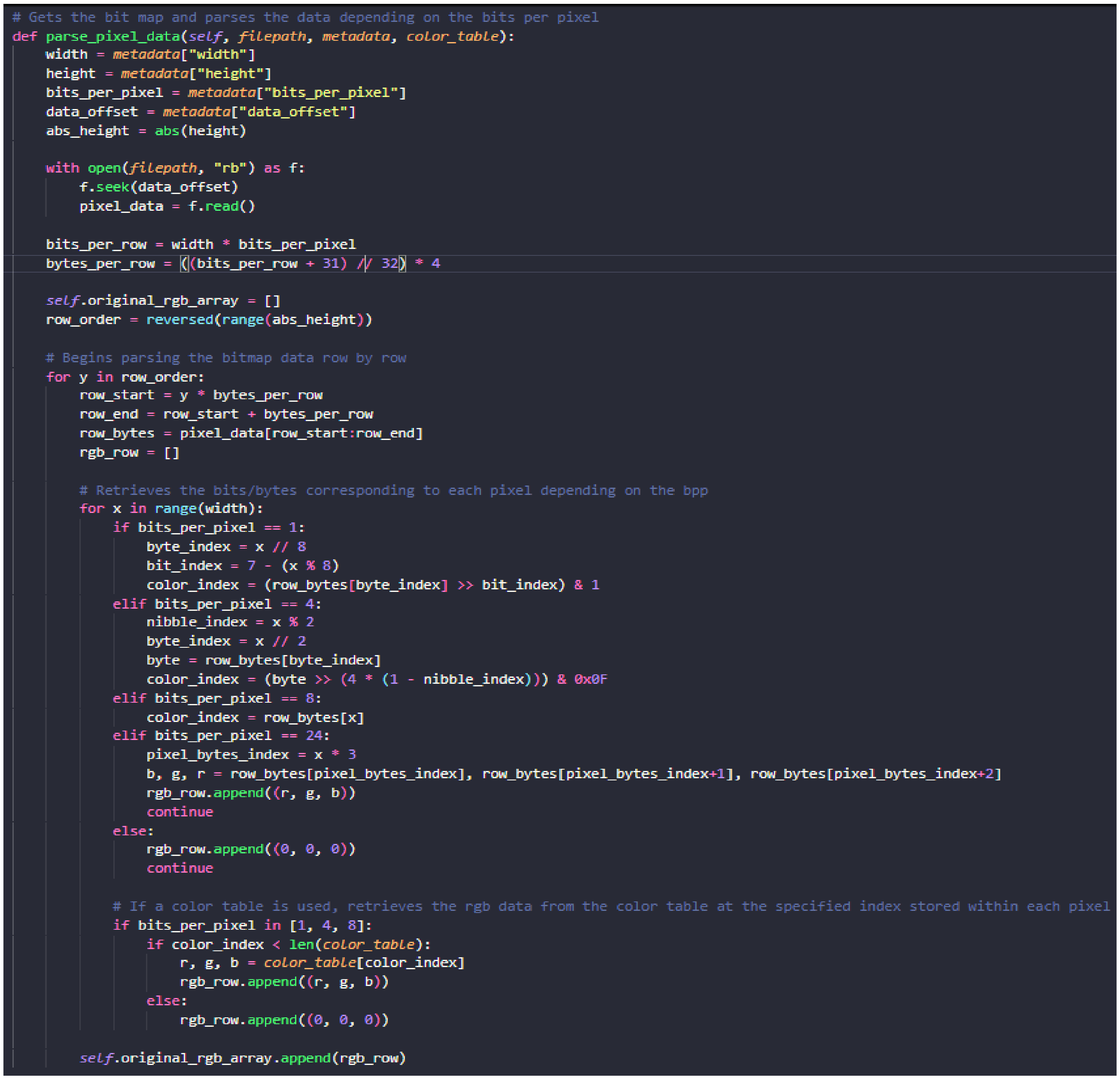
This first checks if a color table is even being used by checking if the bpp is in [1,4,8]. If so that means a color table exists. It then parses the BGR data from the color table section of the bmp and save it to an array that gets returned



```python
# If a color table exists, parses the data from it onto an array and return it
def parse_color_table(self, filepath, metadata):
    bits_per_pixel = metadata["bits_per_pixel"]
    if bits_per_pixel in [1, 4, 8]:
        color_table = []
        colors_used = metadata["colors_used"]
        num_colors = 2 ** bits_per_pixel if colors_used == 0 else colors_used

        with open(filepath, "rb") as f:
            f.seek(54)
            color_table_data = f.read(num_colors * 4)
            for i in range(num_colors):
                entry = color_table_data[i*4 : (i+1)*4]
                blue, green, red, _ = entry
                color_table.append((red, green, blue))

        return color_table
    else:
        return
```

parse_pixel_data(self, filepath, metadata, color_table):

The pixel data in a bmp file is parsed one row at a time in reversed order since the images are stored bottom to top. The rows are then parsed one pixel at a time. The data for each pixel needs to be parsed differently depending on the bpp value.



```python
# Gets the bit map and parses the data depending on the bits per pixel
def parse_pixel_data(self, filepath, metadata, color_table):
    width = metadata["width"]
    height = metadata["height"]
    bits_per_pixel = metadata["bits_per_pixel"]
    data_offset = metadata["data_offset"]
    abs_height = abs(height)

    with open(filepath, "rb") as f:
        f.seek(data_offset)
        pixel_data = f.read()

    bits_per_row = width * bits_per_pixel
    bytes_per_row = ((bits_per_row + 31) // 32) * 4

    self.original_rgb_array = []
    row_order = reversed(range(abs_height))

    # Begins parsing the bitmap data row by row
    for y in row_order:
        row_start = y * bytes_per_row
        row_end = row_start + bytes_per_row
        row_bytes = pixel_data[row_start:row_end]
        rgb_row = []

        # Retrieves the bits/bytes corresponding to each pixel depending on the bpp
        for x in range(width):
            if bits_per_pixel == 1:
                byte_index = x // 8
                bit_index = 7 - (x % 8)
                color_index = (row_bytes[byte_index] >> bit_index) & 1
            elif bits_per_pixel == 4:
                nibble_index = x % 2
                byte_index = x // 2
                byte = row_bytes[byte_index]
                color_index = (byte >> (4 * (1 - nibble_index))) & 0x0F
            elif bits_per_pixel == 8:
                color_index = row_bytes[x]
            elif bits_per_pixel == 24:
                pixel_bytes_index = x * 3
                b, g, r = row_bytes[pixel_bytes_index], row_bytes[pixel_bytes_index+1], row_bytes[pixel_bytes_index+2]
                rgb_row.append((r, g, b))
                continue
            else:
                rgb_row.append((0, 0, 0))
                continue

            # If a color table is used, retrieves the rgb data from the color table at the specified index stored within each pixel
            if bits_per_pixel in [1, 4, 8]:
                if color_index < len(color_table):
                    r, g, b = color_table[color_index]
                    rgb_row.append((r, g, b))
                else:
                    rgb_row.append((0, 0, 0))

        self.original_rgb_array.append(rgb_row)
```

A brief explanation of what I've done for each bpp value:

- bpp = 1
    - since the rows are comprised of bytes, that means that there is data for 8 pixels in each index of the row_bytes array
    - The index of the bytes only increment after 8 pixels have been scanned which explains the x // 8
    - The bit for each pixel is then grabbed by bit shifting to the correct one, then masking all other bits except for the one for the current pixel. The data is then saved as the index for the color table
- bpp = 4
    - Does essentially the same as bpp = 1 except the pixels have 4 bits, meaning we grab 4 bits from every byte in each row. These bits will be saved as the index in the color table
- bpp = 8
    - Same thing as before again, except this one is the simplest because each byte is a pixel. We can easily iterate through the row of bytes to get the corresponding color table index for each pixel
- bpp = 24
    - No more color table. The color data is now in the pixels meaning every 3 bytes contains the complete color data of a pixel.

If a color table was used, the last part is to use the index we acquired above and grab the data from the table at that index. All RGB data for each pixel is appended to an array that later gets processed with the brightness, scale, and channel toggle modifications

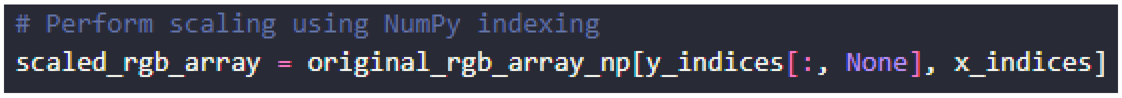
process_image(self, *args):

This function applies all the filtering and modifications that the user specifies to the image. This includes the brightness, scale, and channel toggle changes. The scale is changed first by taking the original array of RGB data and creating a NumPy array from it. I chose NumPy for the array operations because its fast and efficient since it doesn't require loops. I used a nearest-neighbor interpolation technique by basically using NumPy to evenly space out the indices of rows and columns of pixels in the original image to a smaller scaled size. This would remove data to make the image smaller. An example to show how it works:

Original Array: [0,1,2,3,4,5,6,7,8,9,10] -> now we scale this using NumPy.linspace(0, 10, 6)

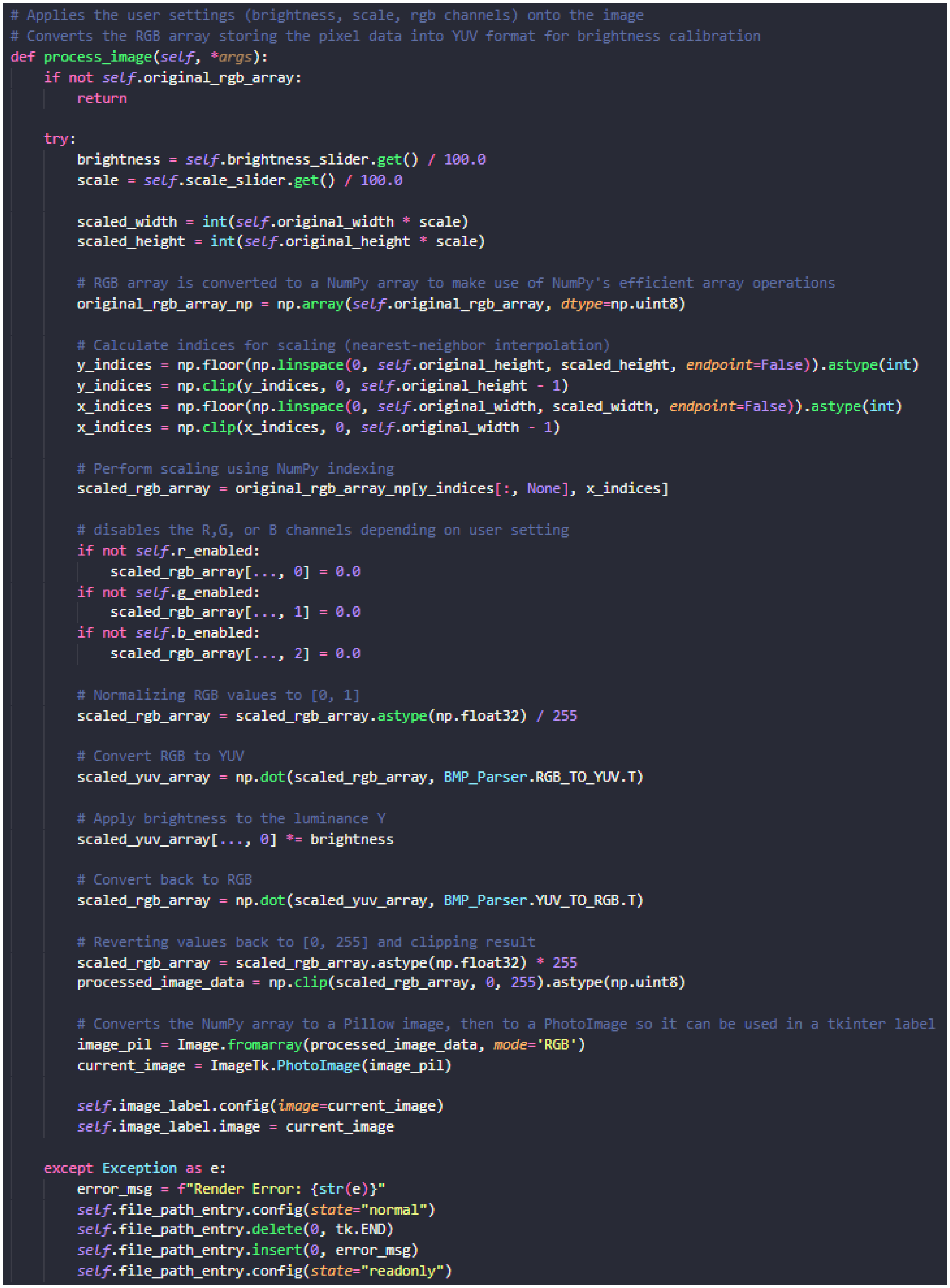
Scaled Array: [0, 2, 4, 6, 8, 10]

This cuts out some of the data to scale down the array. The final scaling step uses the indices to create a scaled down version of the array of pixels by removing rows and columns.



```
# Perform scaling using NumPy indexing
scaled_rgb_array = original_rgb_array_np[y_indices[:, None], x_indices]
```

The following operations are much simpler and just involve some matrix multiplications using NumPy. First the RGB values are normalized then converted to YUV values. After the multiplication the brightness modifier is applied to the Y component of each pixel and then they are converted back to RGB. Like I mentioned, I used different matrices because they seemed to have better color accuracy. The RGB channel toggles just switch the specified color component in the array of RGB data to 0. Once all the processing is done, the RGB array is made into a pillow image object and then a PhotoImage object. This was done just for compatibility's sake to allow the image to be displayed in a tkinter label. All parsing is still done manually. Below is the code for the function



```python
# Applies the user settings (brightness, scale, rgb channels) onto the image
# Converts the RGB array storing the pixel data into YUV format for brightness calibration
def process_image(self, *args):
    if not self.original_rgb_array:
        return

    try:
        brightness = self.brightness_slider.get() / 100.0
        scale = self.scale_slider.get() / 100.0

        scaled_width = int(self.original_width * scale)
        scaled_height = int(self.original_height * scale)

        # RGB array is converted to a NumPy array to make use of NumPy's efficient array operations
        original_rgb_array_np = np.array(self.original_rgb_array, dtype=np.uint8)

        # Calculate indices for scaling (nearest-neighbor interpolation)
        y_indices = np.floor(np.linspace(0, self.original_height, scaled_height, endpoint=False)).astype(int)
        y_indices = np.clip(y_indices, 0, self.original_height - 1)
        x_indices = np.floor(np.linspace(0, self.original_width, scaled_width, endpoint=False)).astype(int)
        x_indices = np.clip(x_indices, 0, self.original_width - 1)

        # Perform scaling using NumPy indexing
        scaled_rgb_array = original_rgb_array_np[y_indices[:, None], x_indices]

        # disables the R,G, or B channels depending on user setting
        if not self.r_enabled:
            scaled_rgb_array[..., 0] = 0.0
        if not self.g_enabled:
            scaled_rgb_array[..., 1] = 0.0
        if not self.b_enabled:
            scaled_rgb_array[..., 2] = 0.0

        # Normalizing RGB values to [0, 1]
        scaled_rgb_array = scaled_rgb_array.astype(np.float32) / 255

        # Convert RGB to YUV
        scaled_yuv_array = np.dot(scaled_rgb_array, BMP_Parser.RGB_TO_YUV.T)

        # Apply brightness to the luminance Y
        scaled_yuv_array[..., 0] *= brightness

        # Convert back to RGB
        scaled_rgb_array = np.dot(scaled_yuv_array, BMP_Parser.YUV_TO_RGB.T)

        # Reverting values back to [0, 255] and clipping result
        scaled_rgb_array = scaled_rgb_array.astype(np.float32) * 255
        processed_image_data = np.clip(scaled_rgb_array, 0, 255).astype(np.uint8)

        # Converts the NumPy array to a Pillow image, then to a PhotoImage so it can be used in a tkinter label
        image_pil = Image.fromarray(processed_image_data, mode='RGB')
        current_image = ImageTk.PhotoImage(image_pil)

        self.image_label.config(image=current_image)
        self.image_label.image = current_image

    except Exception as e:
        error_msg = f"Render Error: {str(e)}"
        self.file_path_entry.config(state="normal")
        self.file_path_entry.delete(0, tk.END)
        self.file_path_entry.insert(0, error_msg)
        self.file_path_entry.config(state="readonly")
```