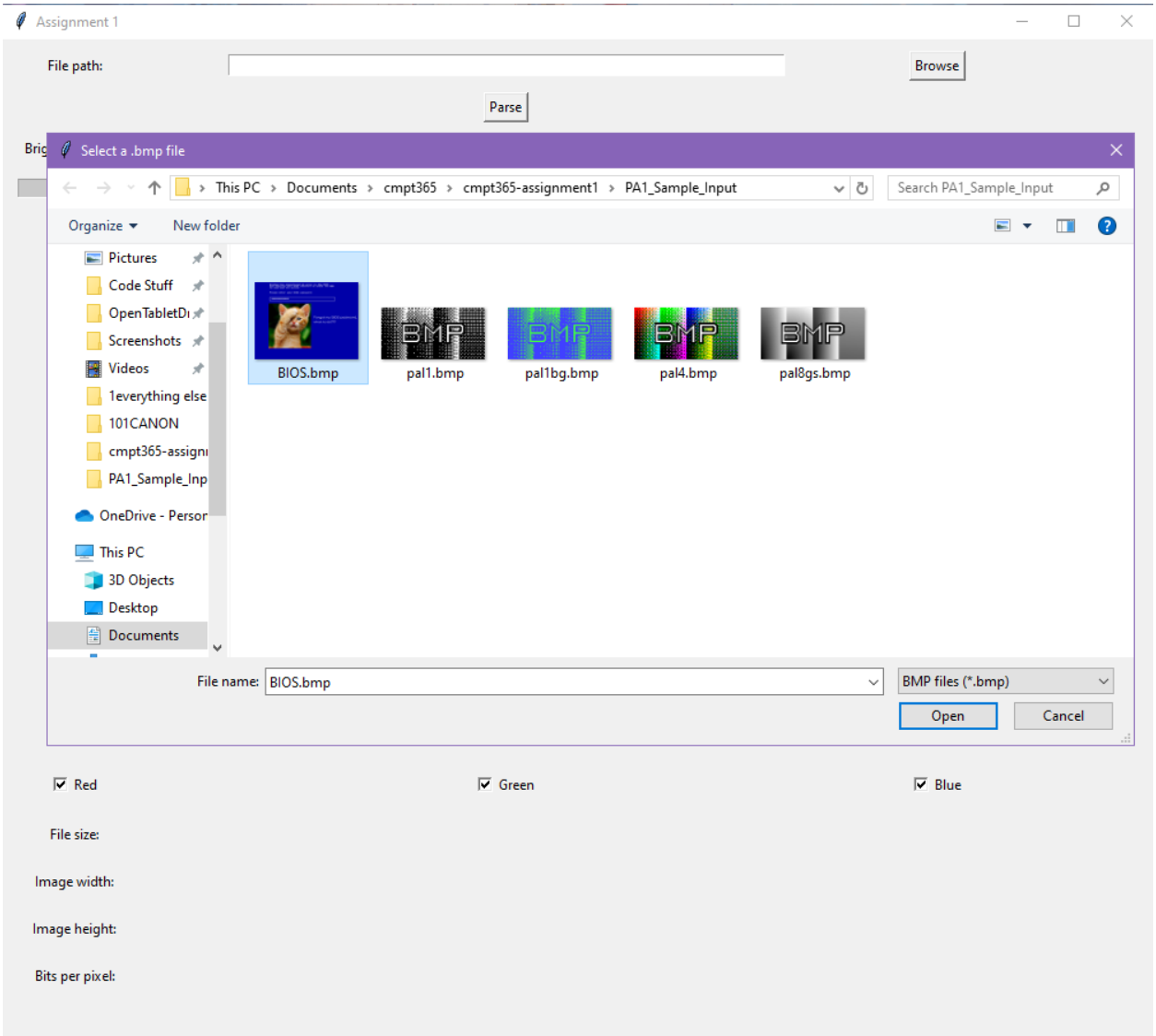
CMPT 365 – Programming Assignment 1

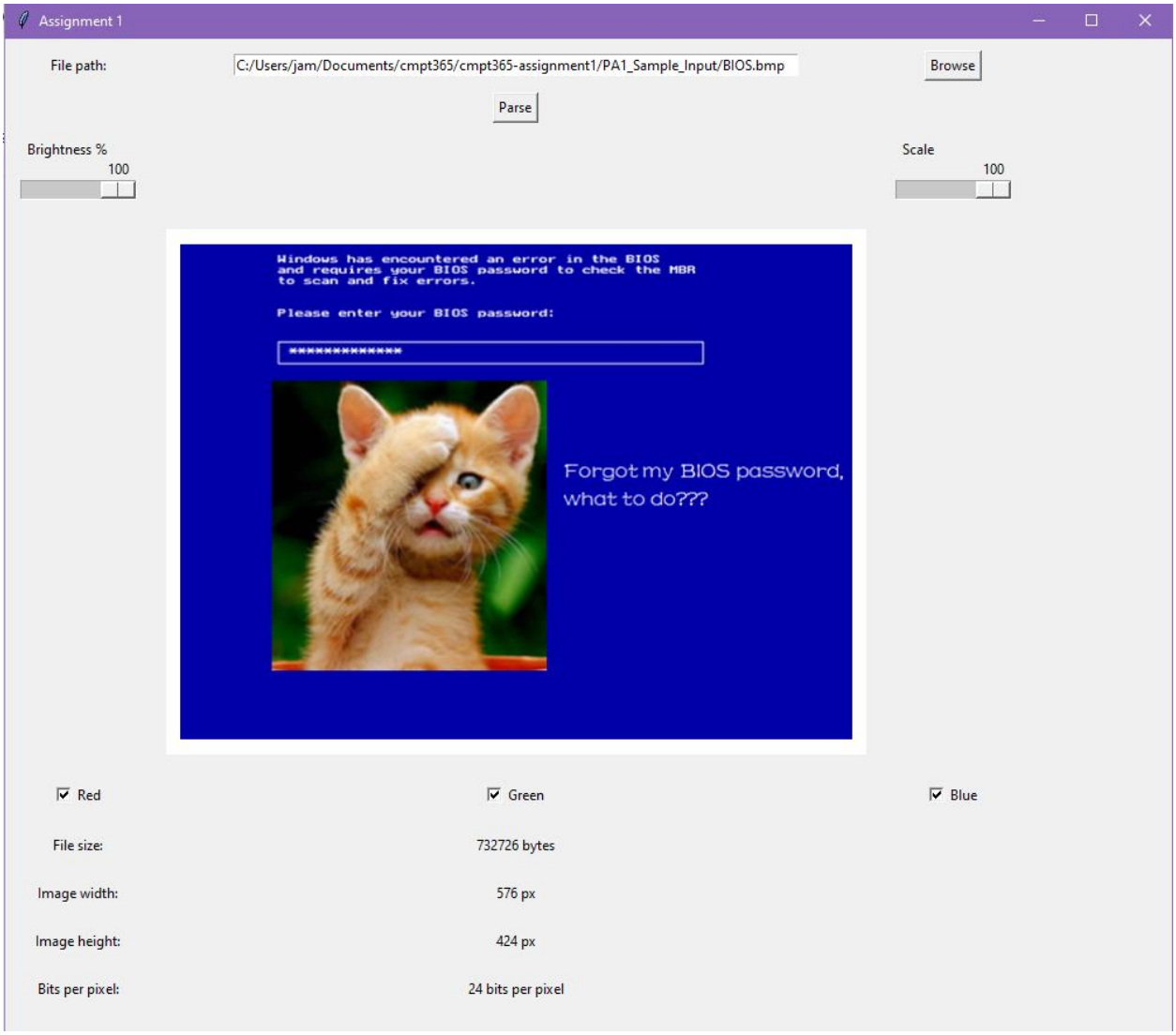
James Cam, 301562474, jca548@sfu.ca

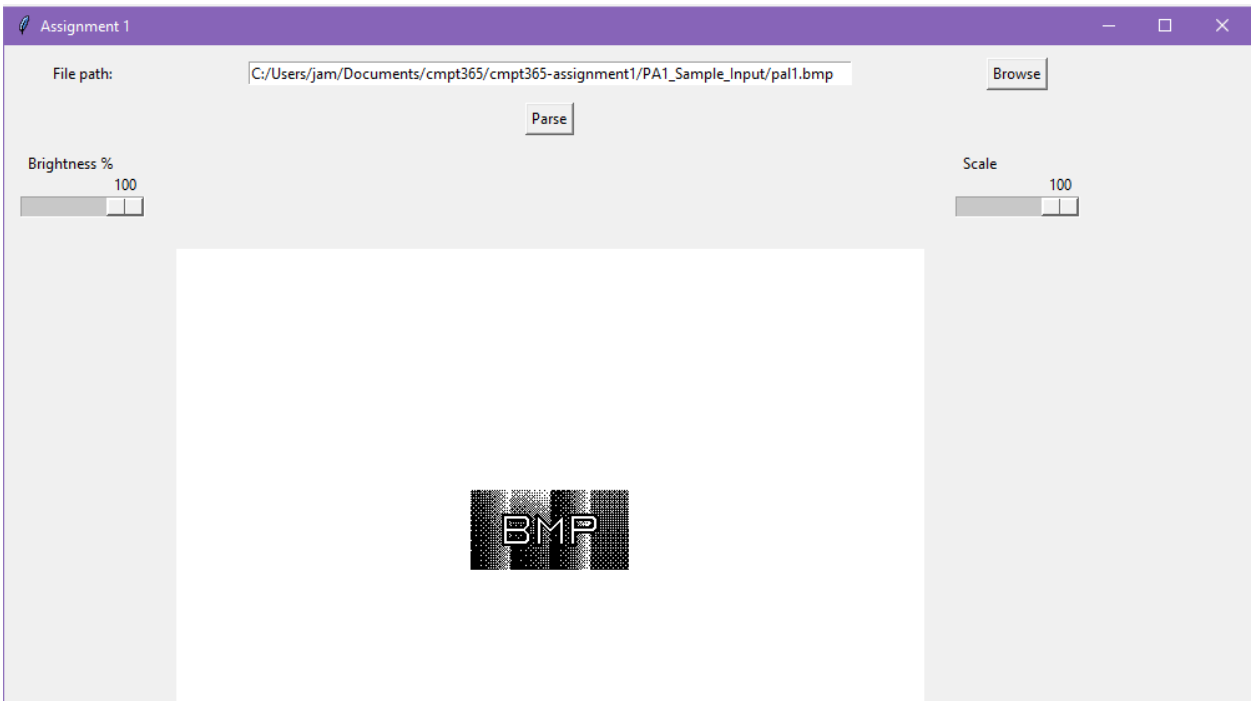
Date: Oct. 3rd, 2025

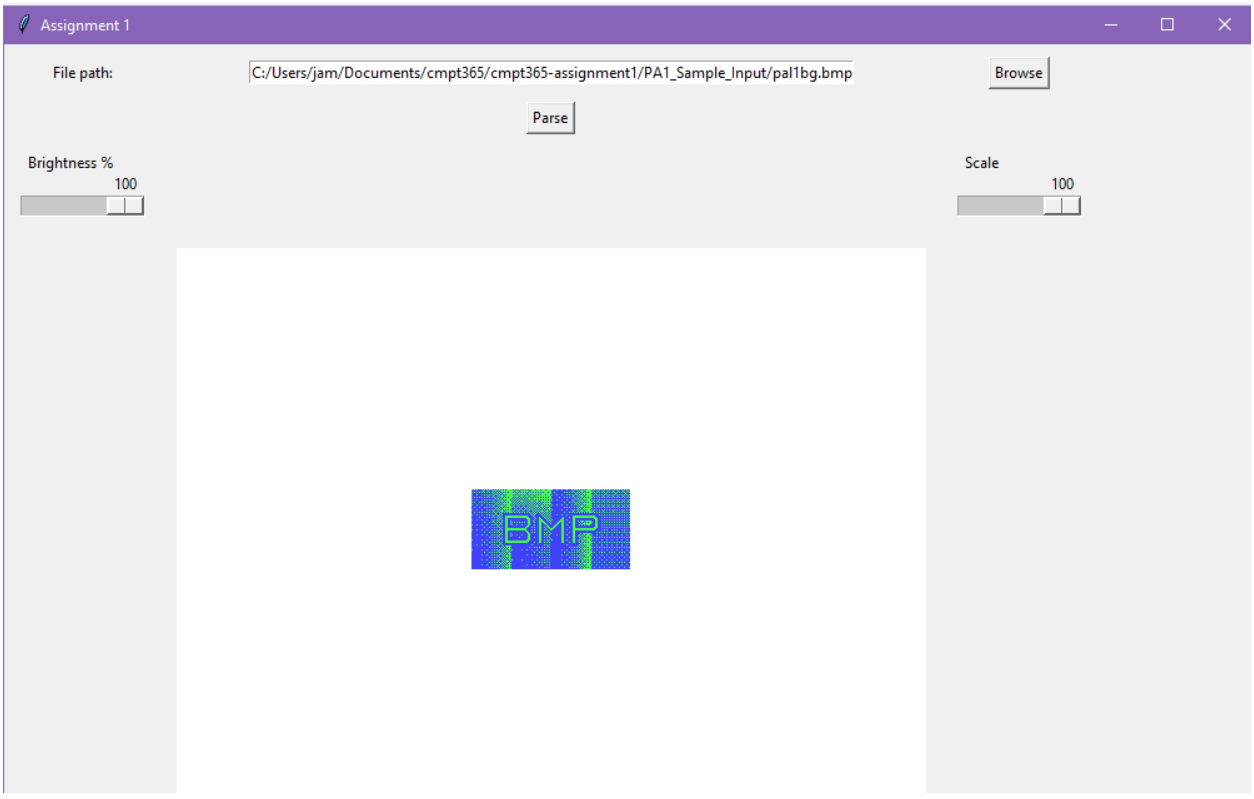
Program Screenshots:

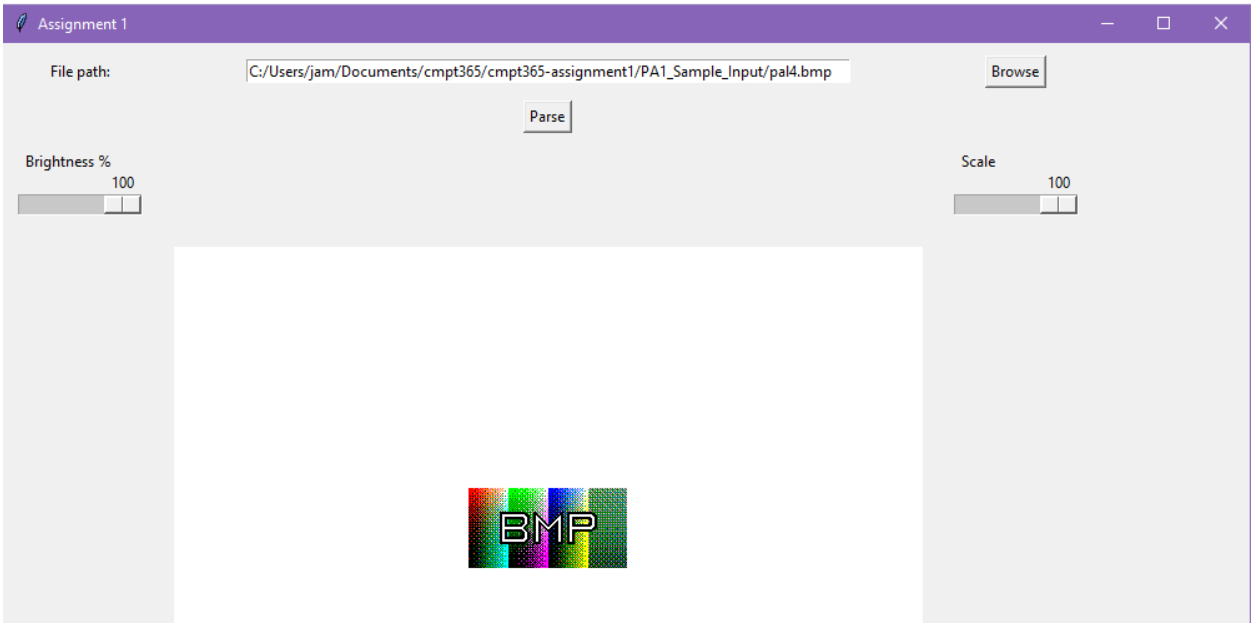
The following screenshots will show what each ".bmp" file looks like within the program before showcasing the functionality of the other features on the sample file "BIOS.bmp".

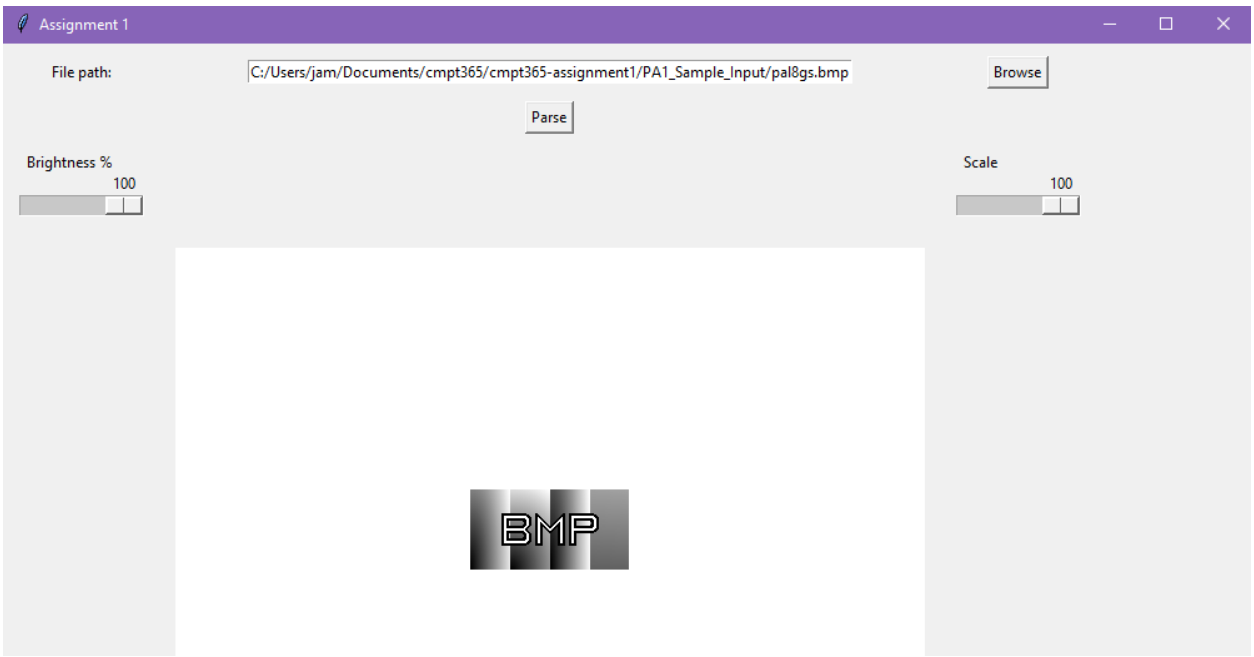


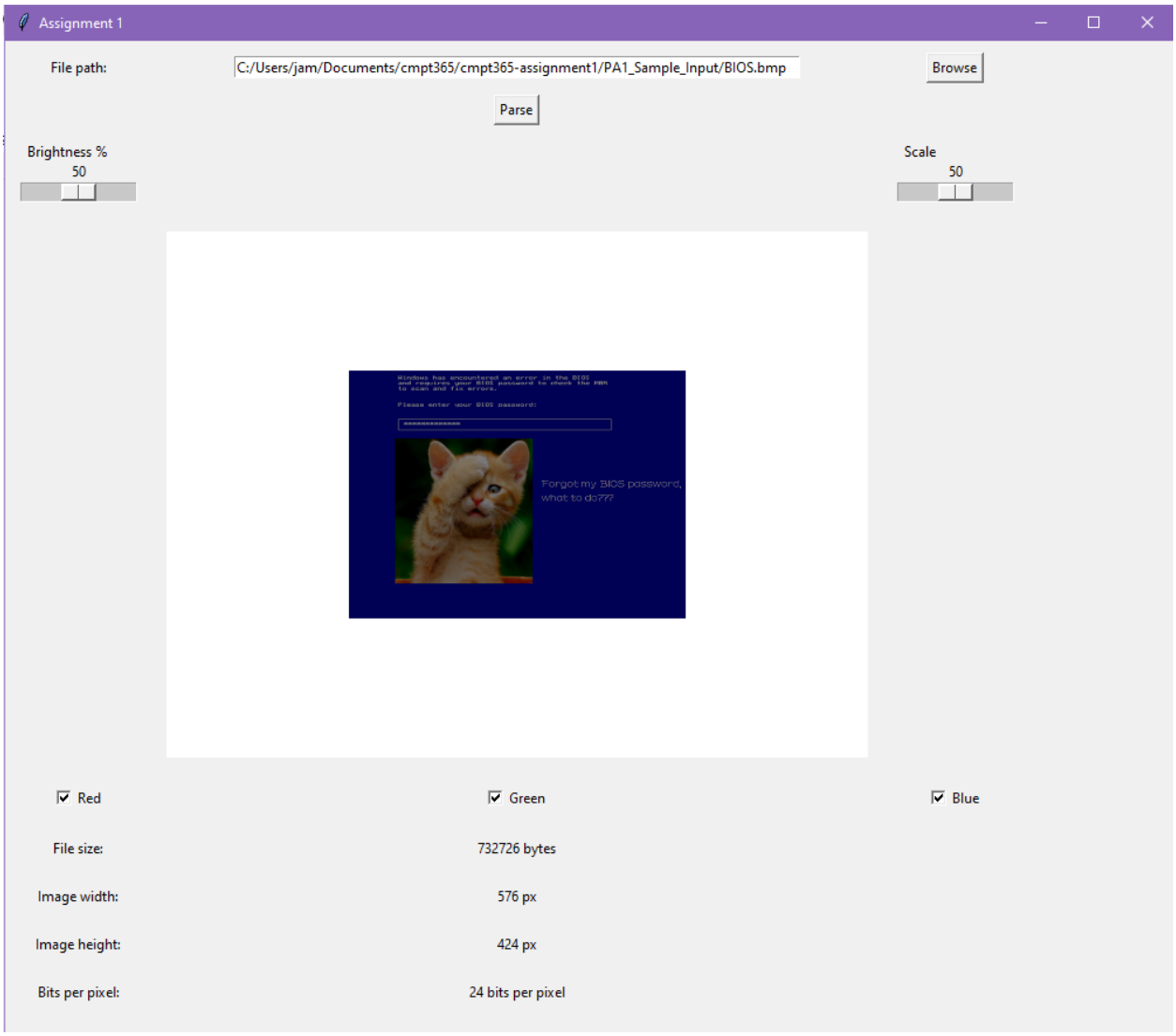

Assignment 1
File path:
C:/Users/jam/Documents/cmpt365/cmpt365-assignment1/PA1_Sample_Input/BIOS.bmp
Browse
Parse
Brightness %
100
Scale
100
Windows has encountered an error in the BIOS
and requires your BIOS password to check the MBR
to scan and fix errors.
Please enter your BIOS password:
****
Forgot my BIOS password,
what to do???
Red
Green
Blue
File size:
732726 bytes
Image width:
576 px
Image height:
424 px
Bits per pixel:
24 bits per pixel


Assignment 1
File path:
C:/Users/jam/Documents/cmpt365/cmpt365-assignment1/PA1_Sample_Input/pal1.bmp
Browse
Parse
Brightness %
100
Scale
100
BMP

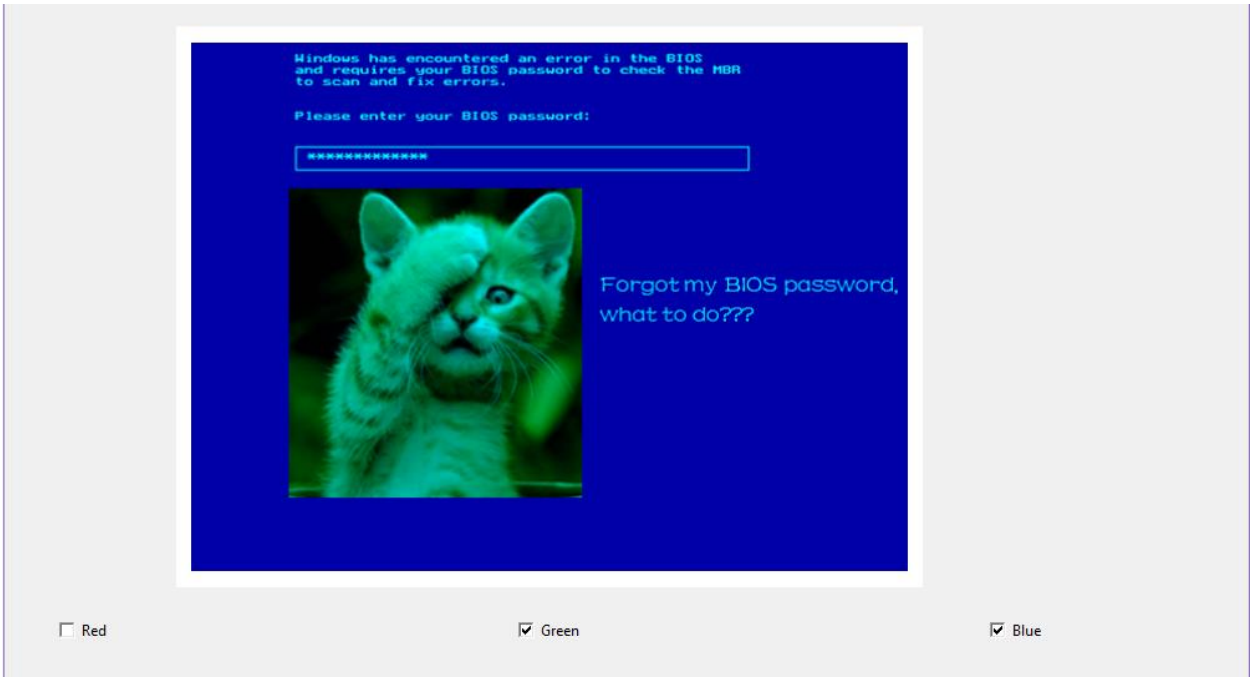

Assignment 1
File path:
C:/Users/jam/Documents/cmpt365/cmpt365-assignment1/PA1_Sample_Input/pal1bg.bmp
Browse
Parse
Brightness %
100
Scale
100
BMP


Assignment 1
File path:
C:/Users/jam/Documents/cmpt365/cmpt365-assignment1/PA1_Sample_Input/pal4.bmp
Browse
Parse
Brightness %
100
Scale
100
BMP


Assignment 1
File path:
C:/Users/jam/Documents/cmpt365/cmpt365-assignment1/PA1_Sample_Input/pal8gs.bmp
Browse
Parse
Brightness %
100
Scale
100
BMP



“BIOS.bmp” at 50% brightness and 50% scale.



“BIOS.bmp” with different RGB channels enabled.

Essential Code Explanation:

This section will only explain parts of the code that are either not completely obvious, or complex enough to warrant an explanation. Simple functions, for example such as functions that get the value of something will not be explained.

get_color_table(*bmp_bytes, bpp*):

This function retrieves the color table of a ".bmp" file by parsing through the color table only if the bits per pixel (bpp) of the file is 1, 4, or 8 bpp. 24 bpp images do not store color on a color table hence it will return nothing if that's the case.

```
def get_color_table(bmp_bytes, bpp):
  # color table is 4 * numcolors
  # stored as B G R Reserved, and repeated for numcolor times
  if bpp == 1:
    num_colors = 2 # technically 1, but we want 0,1 later
  elif bpp == 4:
    num_colors = 16
  elif bpp == 8:
    num_colors = 256
  else:
    return None

  color_table = []
  for i in range(num_colors):
    # header and infoheader is 54 bytes
    offset = 54 + (i * 4)
    blue = bmp_bytes[offset]
    green = bmp_bytes[offset + 1]
    red = bmp_bytes[offset + 2]
    color_table.append((red, green, blue))

  return color_table
```

parse_bmp_file():

Parse_bmp_file first takes the given file path entered in the textbox and converts it into bytes. After getting the byte information which we will call "**bmp_bytes**", it does a check on the first two bits to confirm whether it is a ".bmp" file and it will notify the user if it is not. If it is a ".bmp" file, it will parse the rest of the bytes to extract relevant information and call display_image to display the image using the extracted bytes.

```
def parse_bmp_file():
    if (file_path_entry.get() == ""):
        return # if no file is there currently, just dont do anything

    with open(f"{file_path_entry.get()}", "rb") as f:
        bmp_bytes = f.read()

    # if the first two bytes dont equal to 19778 (BM in little endian) let user know
    if get_file_type(bmp_bytes) != 19778:
        file_path_entry.delete(0, tk.END)
        file_path_entry.insert(0, "Invalid file, please insert a .bmp file.")
        image.delete("all")
        file_size.config(text="")
        image_width.config(text="")
        image_height.config(text="")
        bits_per_pixel.config(text="")
        return

    display_image(bmp_bytes)
    file_size.config(text=f"{get_file_size(bmp_bytes)} bytes")
    image_width.config(text=f"{get_file_width(bmp_bytes)} px")
    image_height.config(text=f"{get_file_height(bmp_bytes)} px")
    bits_per_pixel.config(text=f"{get_bits_per_pixel(bmp_bytes)} bits per pixel")
```

display_image(*bmp_bytes*):

The display_image function is very long and will be explained through multiple parts and screenshots.

```
def display_image(bmp_bytes):
  # This function will for sure be redone in future iterations in the case that we are extending
  # the current functionality of the current program. This is nightmare code written at 12am..

  global img # without it being global, image will not show
  bpp = get_bits_per_pixel(bmp_bytes)
  width = get_file_width(bmp_bytes)
  height = get_file_height(bmp_bytes)
  pixel_data = bmp_bytes[get_data_offset(bmp_bytes):]
  color_table = get_color_table(bmp_bytes, bpp)
  pixels = [] # contains the pixels for numpy array to modify later
```

Besides initializing some variables, display_image first check if the bpp is either 1, 4, 8, or 24 . If the bpp does fall in one of these values, it will look to properly extract information from the pixel data section of the bmp_bytes and store it into an array to use later when displaying the final image.

```
if bpp == 1:
  # (width + 7) // 8 gives the number of bytes width will use (without padding)
  # (((width + 7) // 8 + 3) // 4 ) * 4 gives the number of bytes for width (with padding)
  # simplified: ((width + 31) // 32) * 4
  width_padded = ((width + 31) // 32) * 4
  for y in range(height):
    row_y_start = y * width_padded # gives us the byte index for the start of the yth row
    for x in range(width):
      byte_i = row_y_start + (x // 8)
      bit_i = 7 - (x % 8)
      pixel_val = (pixel_data[byte_i] >> bit_i) & 1 # either 0 or 1
      pixels.extend(color_table[pixel_val])
elif bpp == 4:
  # (4w + 7) // 8 = bytes used (no padding)
  # (((4w + 7) // 8 + 3) // 4) * 4
  width_padded = (((4 * width + 7) // 8 + 3) // 4) * 4
  for y in range(height):
    row_y_start = y * width_padded
    for x in range(width):
      byte_i = row_y_start + (x // 2) # every pixel now occupies 4 bits vs 1 byte in 1ppx
      # traverse by 4 bits per byte now
      if x % 2 == 0:
        pixel_val = (pixel_data[byte_i] >> 4) & 0b1111
      else:
        # no need to shift for the last
        pixel_val = pixel_data[byte_i] & 0b1111
      pixels.extend(color_table[pixel_val])
```

```
elif bpp == 8:
  # no need to account for bits when padding, only bytes itself.
  width_padded = (width + 3) // 4 * 4
  for y in range(height):
    row_y_start = y * width_padded
    for x in range(width):
      byte_i = row_y_start + x
      pixel_val = pixel_data[byte_i]
      pixels.extend(color_table[pixel_val])
elif bpp == 24:
  # same formula like 8bpp, but each pixel (width) is 3 bytes!
  width_padded = ((3 * width) + 3) // 4 * 4
  for y in range(height):
    row_y_start = y * width_padded
    for x in range(width):
      byte_i = row_y_start + (3 * x) # 3 bytes per pixel, each byte stores color
      blue = pixel_data[byte_i]
      green = pixel_data[byte_i + 1]
      red = pixel_data[byte_i + 2]
      pixels.extend([red, green, blue])
else:
  file_path_entry.delete(0, tk.END)
  file_path_entry.insert(0, "This program only accepts files of 1, 4, 8, and 24 bits per pixel!")
  image.config(image="")
  return
```

After extracting all the pixel values, display_image will look to modify the array of pixels based on both the brightness and scale slider, and the enabled/disabled color channels.

```
# change size based on slider:
scale = scale_slider.get()/100
if scale < 1.0:
  new_pixels = []
  new_width = int(width * scale)
  new_height = int(height * scale)

  if new_height == 0 or new_width == 0:
    image.delete("all")
    return

  # calculate how many pixels to skip in the original
  x_skip = width / new_width
  y_skip = height / new_height

  for y in range(new_height):
    for x in range(new_width):
      orig_x = int(x * x_skip)
      orig_y = int(y * y_skip)
      pixel_i = (orig_y * width + orig_x) * 3
      new_pixels.extend([pixels[pixel_i], pixels[pixel_i+1], pixels[pixel_i+2]])
  pixels = new_pixels
  width = new_width
  height = new_height
```

```
# change brightness here based on slider:
brightness = brightness_slider.get()/100
if brightness != 1.0:
  for i in range(0, len(pixels), 3):
    pixels[i] = max(0, min(255, int(pixels[i] * brightness)))
    pixels[i+1] = max(0, min(255, int(pixels[i+1] * brightness)))
    pixels[i+2] = max(0, min(255, int(pixels[i+2] * brightness)))

# enable/disable RGB
show_red, show_green, show_blue = red_channel.get(), green_channel.get(), blue_channel.get()
if not show_red or not show_green or not show_blue:
  for i in range(0,len(pixels), 3):
    if not show_red:
      pixels[i] = 0
    if not show_green:
      pixels[i+1] = 0
    if not show_blue:
      pixels[i+2] = 0
```

Once we modify the given array of pixels through scaling/resizing, brightness changes, or enabling or disabling different RGB channels, we can finally look to display the image. Using numpy, we can reshape the pixel array and flip the array to correctly orient the image before the image is finally created in the canvas.

```
# shape and flip array
pixel_array = np.array(pixels, dtype=np.uint8) # match datatype to RGB (8 bits)
pixel_array = pixel_array.reshape((height, width, 3))
pixel_array = np.flipud(pixel_array)

# Displaying the image using canvas
array_img = Image.fromarray(pixel_array, 'RGB')
img = ImageTk.PhotoImage(array_img)
image.delete("all")
c_width = image.winfo_width()
c_height = image.winfo_height()
x = (c_width - width) // 2
y = (c_height - height) // 2
image.create_image(x, y, anchor=tk.NW, image=img)
```