assgn3_D1171708 Brian

In this assignment. Firstly, Input the size of the frame and the RGB value,

```
do{
  printf("Enter the size of frame in pixel (between 4 and 20: )");
  scanf("%d",&size_blank);
}while(size_blank<4||size_blank>20);

printf("Enter the RGB value of the frame color: ");
scanf("%d %d %d",&R_value,&G_value,&B_value);
```

set up the print_header

```
// Print the image file head.
void print_header(Header header) {
  printf("Type:            %c%c\n", header.Type[0], header.Type[1]); // Two fixed characters, "BM".
  printf("Size:            %u\n", header.Size); // File size in bytes.
  printf("Resserved:       %c%c%c%c\n", header.Reserved[0], header.Reserved[1],
                                      header.Reserved[2], header.Reserved[3]); // Reserved field.
  printf("OffsetBits:      %u\n", header.OffsetBits); // Offset.
  printf("InfoSize:        %u\n", header.InfoSize); // Information size in byte.
  printf("Width:           %u\n", header.Width); // Image width in pixel.
  printf("Height:          %u\n", header.Height); // Image height in pixel.
  printf("Planes:          %d\n", header.Planes); // Number of image planes in the image, must be 1.
  printf("BitPerPixel:     %d\n", header.BitPerPixel); // Number of bits used to represent the data for each pixel.
  printf("Compression:     %u\n", header.Compression); // Value indicating what type of compression.
  printf("ImageSize:       %u\n", header.ImageSize); // Size of the actual pixel data, in bytes.
  // Preferred horizontal resolution of the image, in pixels per meter.
  printf("XResolution:     %u\n", header.XResolution);
  // Preferred vertical resolution of the image, in pixels per meter.
  printf("YResolution:     %u\n", header.YResolution);
  // Value is zero except for indexed images using fewer than the maximum number of colors.
  printf("Colors:          %u\n", header.Colors);
  // Number of colors that are considered important when rendering the image.
  printf("ImportantColors: %u\n", header.ImportantColors);
  // End of output commands
}
```

setup the header of the reduced image

```
  for (i=0; i<4; i++) // Reserved field, four characters.
    reduced_header.Reserved[i] = io_header.Reserved[i];
  reduced_header.OffsetBits = io_header.OffsetBits; // Offset.
  reduced_header.InfoSize = io_header.InfoSize; // Information size in byte.
  // Update the width and length of the image pixels, half size of the input image.
  // If the input image has odd width or length, take the ceiling.
  reduced_header.Width = ceil((float) io_header.Width / 2.0); // Image width in pixel.
  reduced_header.Height = ceil((float) io_header.Height / 2.0); // Image height in pixel.
  reduced_header.Planes = io_header.Planes; // Number of image planes in the image, must be 1.
  reduced_header.BitPerPixel = io_header.BitPerPixel; // Number of bits used to represent the data for each pixel.
  reduced_header.Compression = io_header.Compression; // Value indicating what type of compression.
  // Size of the actual pixel data, in bytes.
  // Compute the image size of the reduced image.
  // "ceil((float) reduced_header.Width * 3.0 / 4.0) * 4" makes the number of bytes of a row to be
  // greater than or eqaul to width*3 and a multiple of 4.
  reduced_header.ImageSize = ceil((float) reduced_header.Width * 3.0 / 4.0) * 4 * reduced_header.Height;
  // Compute the file size in bytes.
  reduced_header.Size = io_header.Size - io_header.ImageSize + reduced_header.ImageSize;
  reduced_header.XResolution = io_header.XResolution; // Preferred horizontal resolution of the image, in pixels per meter.
  reduced_header.YResolution = io_header.YResolution; // Preferred vertical resolution of the image, in pixels per meter.
  reduced_header.Colors = io_header.Colors; // Value is zero except for indexed images using fewer than the maximum number of colors.
  reduced_header.ImportantColors = io_header.ImportantColors; // Number of colors that are considered important when rendering the image.
```

fill the color of the pixel to the reduced image

```c
// Allocate memory space for image pixel data of the reduced image.
reduced_imageData = (unsigned char *) malloc(reduced_header.ImageSize);

// Perform image reduction.// Copy even rows and enen columns of the original input image data to the reduced image.
// Compute the row size of the original image.
fillings = (4 - (io_header.Width * 3) % 4) % 4; // The number of filling bytes at the end of a row.
rowSize = io_header.Width * 3 + fillings; // The number of bytes in a row of the original image.

// Compute the row size of the reduced image.
fillings = (4 - (reduced_header.Width * 3) % 4) % 4; // The number of filling bytes at the end of a row.
rowSize_reduced = reduced_header.Width * 3 + fillings; // The number of bytes in a row of the reduced image.
// Copy even rows and enen columns of the original input image data to the reduced image.
for (i = 0; i < reduced_header.Height; i++) // Go through all rows of the reduced image.
  for (j = 0; j < reduced_header.Width; j++) { // Go through all pixels in each of the reduced image.
    k = i * 2 * rowSize + j * 2 * 3; // Pixel index of the original input image.
    k_reduced = i * rowSize_reduced + j * 3; // Pixel index of the reduced image.
    reduced_imageData[k_reduced] = io_imageData[k]; // Copy blue level.
    reduced_imageData[k_reduced+1] = io_imageData[k+1]; // Copy green level.
    reduced_imageData[k_reduced+2] = io_imageData[k+2]; // Copy red level
  }
```

Set up the feature of the merge_image by adding the 3*size input blank both width and height.

```c
out_header.Type[0] = 'B'; // Two fixed characters, "BM".
out_header.Type[1] = 'M';
out_header.Size = io_header.Size; // File size in bytes.
for (i=0; i<4; i++) // Reserved field, four characters.
  out_header.Reserved[i] = io_header.Reserved[i];
out_header.OffsetBits = io_header.OffsetBits; // Offset.
out_header.InfoSize = io_header.InfoSize; // Information size in byte.
// Update the width and length of the image pixels, half size of the input image.
// If the input image has odd width or length, take the ceiling.
out_header.Width = ceil((float) io_header.Width+(3*size_blank)); // Image width in pixel.
out_header.Height = ceil((float) io_header.Height+(3*size_blank)); // Image height in pixel.
out_header.Planes = io_header.Planes; // Number of image planes in the image, must be 1.
out_header.BitPerPixel = io_header.BitPerPixel; // Number of bits used to represent the data for each pixel.
out_header.Compression = io_header.Compression; // Value indicating what type of compression.
// Size of the actual pixel data, in bytes.
// Compute the image size of the reduced image.
// "ceil((float) reduced_header.Width * 3.0 / 4.0) * 4" makes the number of bytes of a row to be
// greater than or eqaul to width*3 and a multiple of 4.
out_header.ImageSize = ceil((float) out_header.Width * 3.0 / 4.0) * 4 * out_header.Height;
// Compute the file size in bytes.
out_header.Size = io_header.Size - io_header.ImageSize + out_header.ImageSize;
out_header.XResolution = io_header.XResolution; // Preferred horizontal resolution of the image, in pixels per meter.
out_header.YResolution = io_header.YResolution; // Preferred vertical resolution of the image, in pixels per meter.
out_header.Colors = io_header.Colors; // Value is zero except for indexed images using fewer than the maximum number of colors.
out_header.ImportantColors = io_header.ImportantColors; // Number of colors that are considered important when rendering the image.
```

malloc new palette and image_data and fill with the color which user input

```c
out_imageData = (unsigned char *) malloc(out_header.ImageSize);
fillings = (4 - (out_header.Width * 3) % 4) % 4; // The number of filling bytes at the end of a row.
rowSize = out_header.Width * 3 + fillings;
out_palette = (unsigned char *) malloc(out_header.OffsetBits - out_header.InfoSize - 14); // Allocate memory space for the palette.
fread(out_palette, out_header.OffsetBits - out_header.InfoSize - 14, 1, fptr);  // Read palette from the image file.

for (i = 0; i < out_header.Height; i++) { // Go through all rows.
  for (j = 0; j < out_header.Width; j++) { // Go through all pixels in row i.
    k = i * rowSize + j * 3; // The starting index in imageData of the pixel to be transformed.
    out_imageData[k] =  B_value; // value of B
    out_imageData[k+1] = G_value; // Copy to value of G.
    out_imageData[k+2] = R_value; // Copy to value of R.
    }
}
```

Then, compute the blank and fill the imagedata with four reduced image.

```
// Reuse rowSize and rowSize_reduced computed earlier.
for (i = 0; i < reduced_header.Height; i++) { // Go through all rows of the reduced image.
  for (j = 0; j < reduced_header.Width; j++) { // Go through all pixels in each row of the reduced image.
    k_reduced = i * rowSize_reduced + j * 3; // Pixel index of the reduced image.
    // Computer pixel index of the merged image.
    // Note that row 0 is the bottom row and column 0 is the left-most column.
    k_1 = (i + (out_header.Height+size_blank) / 2) * rowSize + (out_header.Width - 1 - j-size_blank) * 3; // 1st quadrant.
    k_2 = (i + (out_header.Height+size_blank) / 2 ) * rowSize + (j+size_blank)* 3; // 2nd quadrant.
    k_3 = (reduced_header.Height - 1 - i+size_blank) * rowSize + (j+size_blank) * 3; // 3rd quadrant.
    k_4 = (reduced_header.Height - 1 - i+size_blank) * rowSize + (out_header.Width - 1 - j-size_blank) * 3; // 4th quadrant.
    out_imageData[k_1] = reduced_imageData[k_reduced]; // Copy the pixel in the 1st quadrant.
    out_imageData[k_1+1] = reduced_imageData[k_reduced+1];
    out_imageData[k_1+2] = reduced_imageData[k_reduced+2];
    out_imageData[k_2] = reduced_imageData[k_reduced]; // Copy the pixel in the 2nd quadrant.
    out_imageData[k_2+1] = reduced_imageData[k_reduced+1];
    out_imageData[k_2+2] = reduced_imageData[k_reduced+2];
    out_imageData[k_3] = reduced_imageData[k_reduced]; // Copy the pixel in the 3rd quadrant.
    out_imageData[k_3+1] = reduced_imageData[k_reduced+1];
    out_imageData[k_3+2] = reduced_imageData[k_reduced+2];
    out_imageData[k_4] = reduced_imageData[k_reduced]; // Copy the pixel in the 4th quadrant.
    out_imageData[k_4+1] = reduced_imageData[k_reduced+1];
    out_imageData[k_4+2] = reduced_imageData[k_reduced+2];
  }
}
```

Free all memory space of using

```
write_image_file(file_name, out_header, out_palette, out_imageData);
free(io_palette); // Release memory space of palette of the input image.
free(io_imageData); // Release memory space of image pixel data of the input image.
free(out_palette);
free(io_imageData);
free(reduced_imageData); // Release memory space of image pixel data of the reduced image.
```