Digital Image Processing: Homework 1 Report

Task1: Image input/flip/output

a. BMP format:

BMP, short for Bitmap, is an image format primarily used on Windows-based systems. Most of BMP files consist of three primary parts: *file headers* & *info headers* & *raw data*.

file headers (14 bytes):

```
struct BMPHeader {
    uint16_t type;
    uint32_t size;
    uint16_t reserved1;
    uint16_t reserved2;
    uint32_t offset;
};
```

info headers (40 bytes):

```
struct BMPInfoHeader {
    uint32_t size;
    int32_t width;
    int32_t height;
    uint16_t planes;
    uint16_t bitsPerPixel;
    uint32_t compression;
    uint32_t imageSize;
    int32_t xPixelsPerMeter;
    int32_t yPixelsPerMeter;
    uint32_t colorsUsed;
    uint32_t colorsImportant;
};
```

raw data:

The size of raw data =

the size of whole file - (the size of file headers + the size of info headers)

The raw data section follows the header information. It contains the actual image data, pixel by pixel. Each pixel may contain the N channel.

For example, In a 32-bits BMP, each pixel is represented by 4 bytes (RGBA). As for 24-bits BMP, each pixel is represented by 3 bytes (RGB), but the actual order may not be RGB.

Bit depth:

The color information in BMP files is determined by the “bit depth”. which means the number of bits used to represent the color of each pixel. For example, 8-bits means 256 colors or grayscale. And there are 3 colors, so for each pixel we used 24 bits to represent it.

b. Read BMP:

I simply read BMP files by standard c++ file io. And store it into a vector with type unsigned char (1 byte).

```
/* Read BMP */
string filename = "input" + input_num + ".bmp";
std::ifstream file(filename, std::ios::in | std::ios::binary);

if (!file.is_open()) {
    std::cerr << "Error opening the file" << std::endl;
    return 1;
}

BMPHeader header;
BMPInfoHeader infoHeader;

// Read the headers
file.read(reinterpret_cast<char*>(&header), sizeof(BMPHeader));
file.read(reinterpret_cast<char*>(&infoHeader), sizeof(BMPInfoHeader));

// Check if it's a BMP file
if (header.type != 0x4D42) {
    std::cerr << "Not a BMP file" << std::endl;
    return 1;
}

int bitsPerPixel = infoHeader.bitsPerPixel;
int width = infoHeader.width;
int height = infoHeader.height;
int num_channel = bitsPerPixel / 8;
int imageSize = width * height * num_channel; // Each pixel has RGB or RGBA

// Allocate memory to store pixel data
std::vector<unsigned char> data(imageSize);
// Read pixel data
file.read(reinterpret_cast<char*>(data.data()), imageSize);
// Close the file
file.close();
```

Task2: Resolution

a. Discussion

Because we have reduced the number of bits, the color range that a single channel can represent will decrease, resulting in effects such as reduced resolution.

b. How I do it

For example, if we want make each pixel with resolution 3*8 bits turn to 3*6 bits. All we need to do is discard the (8-6) least significant bits, and pad them with 0. eg. 01101111 turn to 01101100. I use shift right and shift left to finish the process.

```
int k = 8 - reso; // k is the number of discarded bits
for(int i = 0; i < data_copy.size(); i+=num_channel){
    // discard k least significant bits, and shift back to padding them with 0
    for(int c = 0; c < num_channel; c++){
        data_copy[i+c] = (data_copy[i+c] >> k) << k;
    }
}
```

Save the vector to BMP file.

```
string filename = "output" + input_num + "_" + to_string(k/2) + ".bmp";
ofstream output(filename, ios::out | ios::binary);
if (!output.is_open()) {
    std::cerr << "Error creating the output file" << std::endl;
    return;
}

// Write the headers
output.write(reinterpret_cast<const char*>(&header), sizeof(BMPHeader));
output.write(reinterpret_cast<const char*>(&infoHeader), sizeof(BMPInfoHeader));

// Write the quantized pixel data
output.write(reinterpret_cast<const char*>(data_copy.data()), data_copy.size());

// Close the output file
output.close();
```

Task3: Scaling

a. Bilinear Interpolation:

Bilinear interpolation is a method used to estimate the values of pixels at non-integer coordinates. We take the four corners to do color interpolation to get the value of the pixel. Take (3.7 , 2.3) for example, we take the color at (3, 2), (4, 2), (3, 3), (4, 3) to get the interpolation value.

```
int new_height = int(height / rate);
int new_width = int(round((width / rate) / 4.0) * 4.0); //approximate to the nearest multiple of 4
int new_ImageSize = new_height * new_width * num_channel;
// cout << "w, h, image_size: " << new_width << " " << new_height << " " << new_ImageSize << endl;

vector<unsigned char> scaledData(new_ImageSize, 255);
float sourceX, sourceY, x_weight, y_weight;
int sourceX_floor, sourceY_floor;
for(int y = 0; y < new_height; y++){
    for(int x = 0; x < new_width; x++){
        sourceX = x * (width - 1) / (new_width - 1);
        sourceY = y * (height - 1) / (new_height - 1);
        sourceX_floor = int(sourceX);
        sourceY_floor = int(sourceY);
        x_weight = sourceX - sourceX_floor;
        y_weight = sourceY - sourceY_floor;

        for(int c = 0; c < num_channel; c++){
            int b1 = data[num_channel * (sourceX_floor + sourceY_floor * width) + c];
            int b2 = data[num_channel * (sourceX_floor + (sourceY_floor+1) * width) + c];
            int b3 = data[num_channel * ((sourceX_floor+1) + sourceY_floor * width) + c];
            int b4 = data[num_channel * ((sourceX_floor+1) + (sourceY_floor+1) * width) + c];
            int tmp = static_cast<int>((1 - x_weight) * (1 - y_weight) * b1 + (1 - x_weight) * y_weight * b2 +
                        x_weight * (1 - y_weight) * b3 + x_weight * y_weight * b4);

            scaledData[num_channel * (x + y * new_width) + c] = static_cast<unsigned char>(tmp);
        }
    }
}
```

Note that the new width after down/up scaling must be the multiple of 4. Otherwise, the result will be weird.