Report

1. **BMP format I/O**:

The BMP image format is a raster graphics image file format used to store bitmap digital images, independently of the display device.

Bitmap file header:

File Header:	Bytes	Description
File type	2	Should be set to 0x424D in hexadecimal.
File size	4	This field stores the size of the BMP file in bytes.
Reserved	4	This field are reserved for specific purposes. Typically set to zero.
Data Offset	4	This field specifies where the pixel data begins in the file.

DIB header (Bitmap information header):

There are some other versions of information header with different features, the typical one is of 40 bytes which contains the following features.

Bitmap Information Header	Bytes	Immediately follow the File Header, which provides details about the image.
Header Size	4	The size of this header. (40 bytes for the Windows BMP format)
Image Width	4	Specifies the width of the image in pixels.
Image Height	4	Specifies the height of the image in pixels.
Color Planes	2	Indicates the number of color planes, which is usually set to 1.
BitsPerPixel	2	Specifies the number of bits used to represent each pixel's color. Common values include 1, 4, 8, 16, 24, 32 bits.
Compression	4	Specifies the compression method used, which can be uncompressed (0) or various compression algorithms like RLE.
Image size	4	Indicates the size of the image data in bytes. This is usually set to zero for uncompressed BMPs.
X Resolution	4	Specifies the horizontal resolution of the image in pixels per meter.
Y Resolution	4	Specifies the vertical resolution of the image in pixels per meter.
Colors Used	4	Indicates the number of colors in the color palette. For 24-bit BMPs, this is typically set to zero.
Important Color	4	Specifies the number of important colors. For most BMPs, this field is set to zero.

Reading: First preparing the data structure to store the header information, then creating the space according to the image size and store the pixel data.

✓ We need to deal with the padding input. When the input image width times channels are not the multiple of 4 bytes like the following figure, the data-size is with extra padding space.

- Row size: $225 \times 24 \div 8 = 675$ bytes
- Data row size: ceil $(225 \times 24 \div 8 \div 4) \times 4 = 676$ bytes
- Real data size: $676 \times 225 = 152100$ bytes

```
void BMP::bmp_read(const string fname) {
    ifstream infile(fname, ios_base::binary);

    if(infile) {
        infile.read((char*)&file_header, sizeof(file_header));
        if(file_header.signature[0] != 'B' || file_header.signature[1] != 'M') {
            cerr << "Error: Not a valid BMP file." << endl;
            return;
        }
        infile.read((char*)&bmp_info_header, sizeof(bmp_info_header));

        width = bmp_info_header.width;
        height = bmp_info_header.height;
        bitsPerPixel = bmp_info_header.bitsPerPixel;

        // jump to the pixel data location
        infile.seekg(file_header.pixelDataOffset, infile.beg);

        // resize the pixel data vector
        data.resize(height * width * bitsPerPixel / 8);

        // store pixel data
        infile.read((char*)data.data(), data.size());
    }
    infile.close();
    showBmpHeader(file_header);
    showBmpInfoHeader(bmp_info_header);
}
```

```
============================
Bmp Header
file type: 0x4d42
file size: 152154
reservedByte: 00
Pixel Offset: 54
============================
============================
Bmp Info Header
sizeOfInfoHeader: 40
width: 225
height: 225
numberOfColorPlanes: 1
bitsPerPixel: 24
compressionMethod: 0
rawBitmapDataSize: 152100
xPixelPerMeter: 0
yPixelPerMeter: 0
colorTableEntries: 0
importantColors: 0
============================
```

Writing: Write back the header file with original or custom header information.

```
void BMP::write_headers_and_data(ofstream &outfile, vector<uint8_t> &outdata, BmpHeader BH, BmpInfoHeader BIH, bool custom_header) {
    if(!custom_header) {
        outfile.write((const char*)&file_header, sizeof(file_header));
        outfile.write((const char*)&bmp_info_header, sizeof(bmp_info_header));
        outfile.write((const char*)outdata.data(), outdata.size());
    } else {
        outfile.write((const char*)&BH, sizeof(BH));
        outfile.write((const char*)&BIH, sizeof(BIH));
        outfile.write((const char*)outdata.data(), outdata.size());
    }
}
void BMP::bmp_write(const string fname, vector<uint8_t> &outdata, BmpHeader BH, BmpInfoHeader BIH, bool custom_header) {
    ofstream outfile(fname, ios_base::binary);
    if(outfile) {
        write_headers_and_data(outfile, outdata, BH, BIH, custom_header);
    } else {
        cout << "Error! File not correctly written!" << endl;
    }
    cout << "File written successfully!" << endl;
    outfile.close();
}
```

Horizontal Flip: In my implementation method, I compute two offsets. One is the offset at the front of each, the other is the reverse one. Hence, I will swap the values by using two offsets.

```
int offset_front = y * real_row_width + x * channels;
int offset_back = y * real_row_width + (width - x - 1) * channels;

// do swap
swap(data_flip[offset_front + 0], data_flip[offset_back + 0]);
swap(data_flip[offset_front + 1], data_flip[offset_back + 1]);
swap(data_flip[offset_front + 2], data_flip[offset_back + 2]);
if(channels == 4) {
    swap(data_flip[offset_front + 3], data_flip[offset_back + 3]);
}
```

2. **Resolution:**

In this section, we want to use fewer bits space to represent the original 8 bits color information. However, we need to store back data to the 24 bits-per-pixel or 32 bits-per-pixel format. Namely, the red, green, blue and alpha are still stored in 8 bits representation. Hence, we need to do left shift to restore the value back to the bmp file.

Since quantization resolution function doesn't change the image height and width and others header info, we need only to restore the quantized data with the original header information to the output file.

Bits	Value space													Shift
8	0	…	63	64	…	127	128	…	191	192	…	254	255	<< 0
6	0	…	15	16	…	31	32	…	47	48	…	63	63	<< 2
4	0	…	3	4	…	7	8	…	11	12	…	15	15	<< 4
2	0	…	0	1	…	1	2	…	2	3	…	3	3	<< 6
Value shift result														
6	0	…	60	64	…	124	128	…	188	192		252	252	
4	0	…	48	64	….	112	128	…	176	192		240	240	
2	0	…	0	64	..	64	128	…	128	192		192	192	

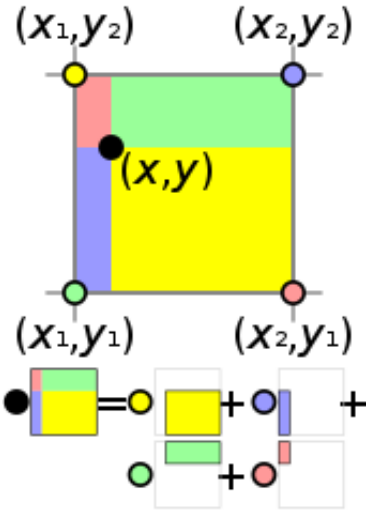
```cpp
// quant to 2, 4, 6 bits representations
void BMP::QuantResolution() {
    filename.erase(filename.find(".bmp"), 4);
    string s = filename.substr(filename.find("input")+5);
    string ofname[3];
    ofname[0] = "output" + s + "_1.bmp";   // 6 bits
    ofname[1] = "output" + s + "_2.bmp";   // 4 bits
    ofname[2] = "output" + s + "_3.bmp";   // 2 bits

    vector<uint8_t> data_quant;
    // data_quant.resize(height * width * bitsPerPixel / 8);
    data_quant.resize(real_image_size);
    int channels = bitsPerPixel / 8;
    int real_row_width = ceil(width * channels / 4.0) * 4;
    int quant_rate;

    for(int r=0; r<3; ++r){
        quant_rate = pow(2, r * 2 + 2);
        for(int y=0; y<height; ++y) {
            for(int x=0; x<width; ++x) {
                int offset = y * real_row_width + x * channels;
                data_quant[offset + 0] = (uint8_t)(((int)data[offset + 0] / quant_rate) * quant_rate);
                data_quant[offset + 1] = (uint8_t)(((int)data[offset + 1] / quant_rate) * quant_rate);
                data_quant[offset + 2] = (uint8_t)(((int)data[offset + 2] / quant_rate) * quant_rate);
                if(channels == 4) {
                    data_quant[offset + 3] = (uint8_t)(((int)data[offset + 3] / quant_rate) * quant_rate);
                }
            }
        }
        bmp_write(ofname[r], data_quant, file_header, bmp_info_header, false);
    }
    return;
}
```

3. **Scaling:** We need to use Bilinear interpolation algorithm to accomplish this operation.

Bilinear Interpolation:

Bilinear interpolation, namely, linear interpolation on both x-axis and y-axis. This is a mathematical method frequently used in computer vision or signal processing, which is to estimate the value of a point within a grid or between two known data points.

The following figure shows the mathematical implementation of the algorithm, where the point in interest is the weighted product of the four grid points nearby it. The weighted relation of the grid point is the area opposite across the target point.



The mathematic representation:

$$f(x,y) \approx w_{11} f(Q_{11}) + w_{12} f(Q_{12}) + w_{21} f(Q_{21}) + w_{22} f(Q_{22})$$

$$w_{11} = (x_2 - x)(y_2 - y)/((x_2 - x_1)(y_2 - y_1))$$
$$w_{12} = (x_2 - x)(y - y_1)/((x_2 - x_1)(y_2 - y_1))$$
$$w_{21} = (x - x_1)(y_2 - y)/((x_2 - x_1)(y_2 - y_1))$$
$$w_{22} = (x - x_1)(y - y_1)/((x_2 - x_1)(y_2 - y_1))$$

Hence, my method is to let the width pad to the multiple of 4 and recompute the ratio of height and width. After computing all the data, write back the data and new info header with the correct information.

```
void BMP::BilinearInterpolate(vector<uint8_t> &indata, vector<uint8_t> &outdata, int OW, int OH, int TW, int TH, int CH) {
    int w_l, h_l, w_h, h_h;
    int a, b, c, d;
    int offset;
    float pixel;
    float dx, dy;
    float w_ratio = float(OW-1) / (TW-1);
    float h_ratio = float(OH-1) / (TH-1);

    for(int y=0; y<TH; y++) {
        for(int x=0; x<TW; x++) {
            for(int k=0; k<CH; k++) {

                w_l = floor(w_ratio * x);
                h_l = floor(h_ratio * y);
                w_h = min(ceil(w_ratio * x), float(OW-1));
                h_h = min(ceil(h_ratio * y), float(OH-1));

                dx = (w_ratio * x) - w_l;
                dy = (h_ratio * y) - h_l;

                a = (int)indata[h_l*OW*CH + w_l*CH + k];   //  a ----- b
                b = (int)indata[h_l*OW*CH + w_h*CH + k];   //  |       |
                c = (int)indata[h_h*OW*CH + w_l*CH + k];   //  |       |
                d = (int)indata[h_h*OW*CH + w_h*CH + k];   //  c ----- d

                pixel = a * (1 - dx) * (1 - dy) + b * (dx) * (1 - dy) + c * (1 - dx) * (dy) + d * (dx) * (dy);
                outdata[y*TW*CH + x*CH + k] = (uint8_t)pixel;
            }
        }
    }
    return;
}
```