Digital Image Processing Homework 1 Report

0550222 葉胤呈

I. Image Input and Output

Bitmap image file(BMP) format is an image storage format. It is mainly consisted of header file, image data, and color table. In BMP file from byte perspective, Header file (54 byte), image data (width*height*channel), color table, and ending padding zeros, which are two-bytes long, are concatenate in order in the file.

For the header file, we can obtain format information about this picture (Detail contents are in Figure 1). The file length of this header file is 54 bytes long. In this lab, since we have the uncompressed file, the most crucial information is width, height, and bits per pixel. Others information is not really matter.

	Shift	Name	Size	Footnote
Bitmap File Header	0000h	Identifier	2	'BM'
	0002h	File Size	4	The size of the image (Unit: byte)
	0006h	Reserved	4	
	000Ah	Bitmap Data Offset	4	The shift of image data from the very start
Bitmap Info Header	000Eh	Bitmap Header Size	4	the length of Bitmap Info Header
	0012h	Width	4	Unit: pixel
	0016h	Height	4	Unit: pixel
	001Ah	Planes	2	
	001Ch	Bits Per Pixel	2	1: 1-bit image 4: 4-bit image 8: 8-bit image 16: 16-bit image 24: 23-bit image 32: 32-bit image
	001Eh	Compression	4	0: uncompressed 1: RLE 8-bit/pixel 2: RLE 4-bit/pixel

				3: bit fields
	0022h	Bitmap Data Size	4	
	0026h	H-Resolution	4	
	002Ah	V-Resolution	4	
	002Eh	Used Colors	4	
	0032h	Important Colors	4	
Palette	0036h	Palette	4	

Figure 1. BMP Header file content

Before we get these necessarily information from header file, we can start to deal with the image data. The image data follows the header file and it is arranged as an array which indicate the pixels from the lower left part of the image to the upper right. The size of the image data is known by:

$$\text{ImageData} = (\text{Width} + \text{PaddingZeros}) * \text{Height} * \left(\frac{BitsPerPixel}{8}\right)(Byte)$$

In the input stage, we should carefully pay attention on the padding zeros of the image. The image browsers on computer will read the BMP file in a minimum group of 4 bytes. That means if we do not pad zeros to the image file which is not divisible by 4 bytes in row, we will not get an in alignment rows in file and will lead the image browsers cannot open the file correctly. For example, if we get a 450*250*3(width, height, channel), we will have a row with 450*3(mod 4) = 2 remains. So, we have to pad 4-2=2 zeros on the tail of each row to make rows divisible.

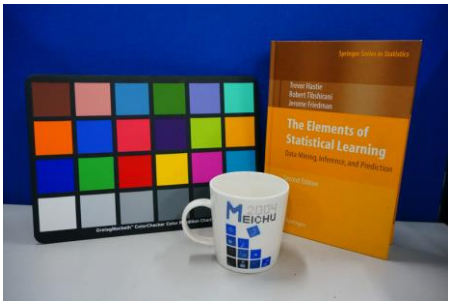
II. Resolution

In this part, we will quantize the pixels aka color quantization and compare those results with the original image. Color quantization reduces the number of colors used in an image. If we only have very limited storage system such as embedded circuits, we will benefit from saving the storage space by quantization. Also, knowing the fact that human eyes is fairly good at seeing small differences in brightness over a relatively large area, but not so good at distinguishing the exact strength of a high frequency brightness variation. Image compression may be used by doing color quantization. Figure 2 shows two images. Input1 is a relatively low frequency (not rapidly change) image and Input 2 is a high frequency image. Both of them are quantized by 64, 16, 2

levels. For 64 levels, both of the images are seems quite well. For 16 levels, Inputs 1 starts to have some distortion, especially on the white desktop with shadow. However, most parts of the Input 2 still look good. The reason is that most parts of this picture are composed by high frequency component such as the colorful notes on the wall. For detailed observation, we can see the black board, an element contains a relatively low-frequency components, starts to have some visible losses. For 2 level quantization, there are just two option, dark and bright. So, both of the images have a very poor quality. By testing every quantization options, I suggest the acceptable quantization levels are input1 for 128 levels and input2 for 16 level. I think this experiments somehow support the fact that human eyes are not so sensitive to high frequency components in image.

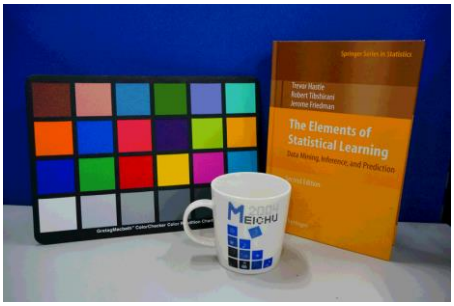
For the implementation, I simply shift the pixel byte right by logarithm of quantization levels then shift it back. So the sampling will be round down to each levels.

$$Pixel_{quantization} = (Pixel_{Original} \gg QuantiFactor) \ll QunatiFactor$$



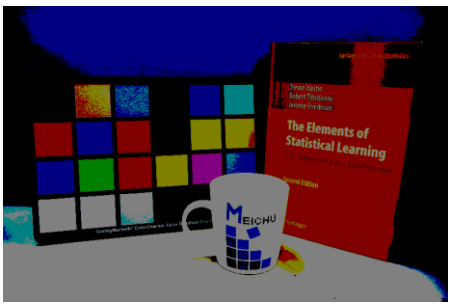
▲Input 1 with 64 level quantization

▲Input 2 with 64 level quantization



▲Input 1 with 16 level quantization

▲Input 2 with 16 level quantization



▲Input 1 with 2 level quantization ▲Input 2 with 2 level quantization

▲ Figure 2. Quantization Results

III. Scaling

In this part, we will scale the image twice larger and half smaller. For scaling down, it is quite simple. I just take an output from every two pixels of the original image. Comparing input1 and input2, we can find that input1 is visibly blur because of the scaling-down process, which is equal to down sampling, but the input2 still seems really robust. The reason is same as the quantization: humans are less sensitive to the change of high frequency components. For up-scaling, I have tried two methods to implement this. First, I simply copy a pixel from original image to the extend four pixels next to it. The results are surprisingly good for both pictures. However, here we are going to takes about bilinear interpolation as the specification suggest. Those results are in Figure 3. The algorithm of bilinear interpolation is as follow. The main concept of bilinear interpolation is that value which will be interpolated is determined by how close to the surrounding points and take the linear combination of those points. We denote output domain pixel coordinate as (X, Y) and input domain pixel coordinate as (x, y) and we have:

$$\begin{aligned}\mathrm{Binlinear(X,Y)} = \mathrm{f}(\lfloor x \rfloor, \lfloor y \rfloor)(1-(\mathrm{x}-\lfloor x \rfloor))(1-(\mathrm{y}-\lfloor y \rfloor)+\\ \mathrm{f}(\lfloor x \rfloor, \lceil y \rceil)(1-(\mathrm{x}-\lfloor x \rfloor))(\mathrm{y}-\lfloor y \rfloor)+\\ \mathrm{f}(\lceil x \rceil, \lfloor y \rfloor)(\mathrm{x}-\lfloor x \rfloor)(1-(\mathrm{y}-\lfloor y \rfloor))+\\ \mathrm{f}(\lceil x \rceil, \lceil y \rceil)(\mathrm{x}-\lfloor x \rfloor)(\mathrm{y}-\lfloor y \rfloor)\end{aligned}$$

and the mapping (X, Y) to input domain (x, y) is

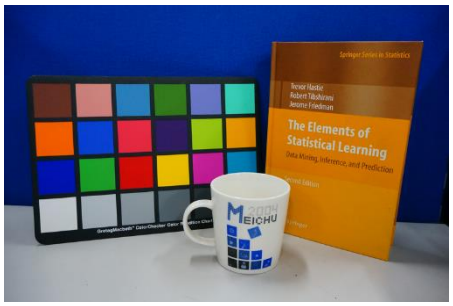
$$\mathrm{y} = \frac{Height_{input}-1}{Height_{output}-1} * Y$$

$$\mathrm{x} = \frac{Width_{input}-1}{Width_{input}-1} * X$$

When implementing the bilinear interpolation, I have noticed an interesting point. Input file format is 8-bit per pixel. That means we have a Gamma-corrected image and we are on the nonlinear RGB domain. However, a linear operation, bilinear interpolation, is performance on nonlinear domain and the results are surprisingly correct. I think the reason why it seems correct is that we just scaling twice so there are not so many points will be insert between known points and the proximate points have the really similar value. So even if we do the linear operation on a nonlinear domain, we can still get a very good approximation.

▲ Input1 down-scaling

▲ Input2 down-scaling



▲ Input1 up-scaling

▲ Input2 up-scaling

▲Figure 3, Scaling Results