Department of Electronics and Electrical Communication Engineering

Indian Institute of Technology Kharagpur

Digital Image and Video Processing Lab (EC69211)

Experiment No: 2

Title: Implementation of Modular Python Functions for BMP Image Processing and Colour Channel Manipulation

Date of submission: 14.08.2024

Name: Unnati Singh

Roll No.: 21EC39027

Instructors:

Prof. Saumik Bhattacharya

Prof. Prabir Kumar Biswas

Introduction:

The BMP (Bitmap) file format is one of the most widely used image formats due to its simplicity and ease of use. It stores image data in a structured manner, making it an ideal choice for learning the fundamentals of image processing. This experiment focuses on the development of Python functions to read, write, and manipulate BMP images.

Files having extension .BMP represent Bitmap Image files that are used to store bitmap digital images. These images are independent of graphics adapter and are also called device independent bitmap (DIB) file format. This independency serves the purpose of opening the file on multiple platforms such as Microsoft Windows and Mac. The BMP file format can store data as two-dimensional digital images in both monochrome as well as colour format with various colour depths.

A Bitmap File Header is similar to other file headers used to identify the file. Since there are different variants to BMP file format, the first 2 bytes of the BMP file format are character “B” and then character “M” in ASCII encoding. All integer values are stored in little-endian format.

After the Bitmap File Header, we have the Bitmap Information Header. The detailed information about the image is represented by this header. Then there is colour palette. A BMP colour palette is an array of structures that specify the RGB intensity values of each colour in a display device’s colour palette. Each pixel in the bitmap data stores a single value used as an index into the colour palette. The colour information stored in the element at that index specifies the colour of that pixel. A BMP file may or may not have the colour palette which can be determined by its bit-width present in the information header. After the colour palette, the pixel data is stored.

Basic BMP File Format				
Name			Size	Description
Header			14 bytes	Windows Structure: BITMAPFILEHEADER
	Signature		2 bytes	'BM'
	FileSize		4 bytes	File size in bytes
	reserved		4 bytes	unused (=0)
	DataOffset		4 bytes	File offset to Raster Data
InfoHeader			40 bytes	Windows Structure: BITMAPINFOHEADER
	Size		4 bytes	Size of InfoHeader =40
	Width		4 bytes	Bitmap Width
	Height		4 bytes	Bitmap Height
	Planes		2 bytes	Number of Planes (=1)
	BitCount		2 bytes	Bits per Pixel 1 = monochrome palette. NumColors = 1 4 = 4bit palletized. NumColors = 16 8 = 8bit palletized. NumColors = 256 16 = 16bit RGB. NumColors = 65536 (?) 24 = 24bit RGB. NumColors = 16M
	Compression		4 bytes	Type of Compression 0 = BI_RGB no compression 1 = BI_RLE8 8bit RLE encoding 2 = BI_RLE4 4bit RLE encoding
	ImageSize		4 bytes	(compressed) Size of Image It is valid to set this =0 if Compression = 0
	XpixelsPerM		4 bytes	horizontal resolution: Pixels/meter
	YpixelsPerM		4 bytes	vertical resolution: Pixels/meter
	ColorsUsed		4 bytes	Number of actually used colors
	ColorsImportant		4 bytes	Number of important colors 0 = all
ColorTable			4 * NumColors bytes	present only if Info.BitsPerPixel <= 8 colors should be ordered by importance
		Red	1 byte	Red intensity
		Green	1 byte	Green intensity
		Blue	1 byte	Blue intensity
		reserved	1 byte	unused (=0)
	repeated NumColors times			
Raster Data			Info.ImageSize bytes	The pixel data

Structure of BMP file

Key Functions in code:

1. `readBMP(path):`
   - **Purpose:** This function reads a BMP image file, extracts important header information, and loads the pixel data into memory for further processing. It supports 24-bit RGB, 8-bit grayscale, and 8-bit colour-indexed BMP images.
   - **How It Works:**
     - File Check: The function first verifies that the file is in BMP format by checking the file extension. If the file is not a BMP, an exception is raised.

- Reading BMP Header: It opens the BMP file in binary mode and reads the file content. The header information is extracted, including image width, height, file size, offset, and bit-width.
- Pixel Array Handling: Depending on the bit depth (24-bit or 8-bit), the function processes the pixel array:
- 24-bit RGB: The pixel data is interpreted directly as red, green, and blue (RGB) values.
- 8-bit Grayscale: Each pixel value is repeated across the RGB channels to create a grayscale image.
- 8-bit Colour-Indexed: The function checks if a colour table is present. If so, it uses the table to convert indexed colours into RGB values.
- Output: The function returns a tuple containing the image size (height, width), the pixel array, the offset, and the colour table (if applicable).

2. `writeBMP(outputfilename, pixelarray, size)`:
   - **Purpose:** This function writes pixel data to a new BMP file, effectively creating or saving an image from the processed pixel array.
   - **How It Works:**
     - File Header Construction: The function constructs the BMP file header, which includes file type (BM), file size, reserved bytes, and the offset where the pixel data starts.
     - Info Header Construction: It also constructs the BMP info header, which includes details like header size, image dimensions (width, height), number of colour planes, bits per pixel (24 in this case), compression type, and other metadata.
     - Pixel Data Writing: The pixel array is converted into a binary format, with padding added to each row to ensure that row size is a multiple of 4 bytes (as required by the BMP format).
     - File Writing: The function writes the combined file header, info header, and pixel data into a new BMP file specified by `outputfilename`.

3. `colourchannelmanipulation(filename, channel)`:
   - **Purpose:** This function performs colour channel manipulation on a BMP image, setting one specific colour channel (Red, Green, or Blue) to zero while retaining the values of the other two channels. The modified image is saved as a new BMP file.
   - **How It Works:**
     - Read Image: The function first reads the BMP image using `readBMP(filename)` to obtain the image size and pixel array.
     - Channel Selection: Based on the specified channel ('R', 'G', or 'B'), the function determines which colour channel to manipulate. It assigns c to 0, 1, or 2 corresponding to the red, green, or blue channels, respectively.

- Manipulation: The function iterates over each pixel in the pixel array and sets the specified channel's value to zero.
- Save Modified Image: The manipulated pixel array is then written back to a new BMP file using `writeBMP(newfilename, pixels, size)`, where `newfilename` indicates which channel was manipulated.

Output Images:

The images obtained after manipulating the colour channels can be seen as:

1. corn.bmp:


Original image


Image with blue channel set to 0


Image with green channel set to 0

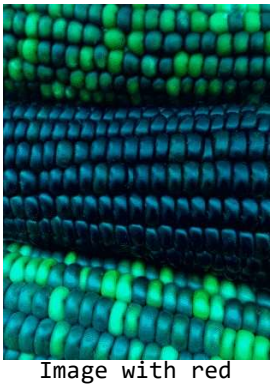
Image with red channel set to 0

2. pepper.bmp:


Original image


Image with blue channel set to 0


Image with green channel set to 0


Image with red channel set to 0

Discussion:

This experiment successfully demonstrated the fundamental operations involved in processing BMP images using Python. By implementing modular functions for reading, writing, and manipulating BMP files, a deeper understanding of image file structures and data handling in digital image processing was achieved.

Understanding BMP Structure:

The BMP format, being straightforward and uncompressed, provides an excellent case study for learning how image data is stored and accessed. The experiment began with parsing the BMP header to extract key details such as image dimensions, bit depth, and the pixel data offset. This information is crucial for correctly interpreting the pixel data and understanding how different bit depths affect image storage.

Colour Channel Manipulation:

The colour channel manipulation task illustrated the impact of individual colour channels (Red, Green, Blue) on the overall appearance of an image. By zeroing out one channel at a time, the experiment highlighted how each channel contributes to the final colour composition of the image. This exercise is particularly valuable for applications in image processing where colour correction or enhancement is required.