BMP IMAGE EDITING SYSTEM IN C

Project Report

Submitted By:
Priyanshu Purohit
Batch 37
SAP ID: 590021863

Abstract

This project report presents the complete development of a BMP Image Editing System implemented in the C programming language. The system enables loading, editing, manipulating, and saving BMP image files using low-level file handling and pixel processing techniques. It integrates several key operations such as cropping, rotating (clockwise and anti-clockwise), flipping (horizontal and vertical), adjusting opacity, and inverting colors. The project demonstrates modular programming, dynamic memory allocation, pointer manipulation, algorithm design, and applying fundamental C concepts to build a practical, real-world application.

The system is menu-driven and interacts with users through a simple and intuitive terminal-based interface. The project showcases how bitmap images can be manually interpreted through byte-level operations, emphasizing a deep understanding of file formats, computer graphics fundamentals, and data structures. The final application is functional, efficient, and extendable for future enhancements such as PNG/JPG support, GUI integration, and advanced image processing filters.

Problem Definition

Digital images are an essential component of modern computing, used in applications ranging from mobile apps to scientific visualization. While numerous tools exist for editing images, academic and low-level implementations often overlook how these operations work internally. Bitmap (BMP) images are one of the simplest and most accessible formats for understanding the structural representation of images. The goal of this project is to design and develop an image editing system capable of reading and manipulating BMP images at the pixel level.

The challenge lies in manually parsing BMP file headers, handling row padding, interpreting pixel arrangements, and processing image transformations without using any pre-built image-processing libraries. Students must rely solely on core C concepts such as structs, pointers, dynamic memory allocation, loops, conditionals, and modular programming.

The BMP Image Editing System must:

1. Load 24-bit and 32-bit BMP images correctly.
2. Parse headers, pixel arrays, and row padding.
3. Perform image editing operations such as cropping, rotating, flipping, opacity adjustment, and color inversion.
4. Save the edited images back into valid BMP files.
5. Use a menu-driven interface with proper user input validation.
6. Maintain clean modular design with separate .c and .h files.
7. Ensure no memory leaks or invalid accesses occur during editing operations.

This project aims to reinforce core C programming concepts while building a fully functional, real-world application based on systems-level image processing.

System Design

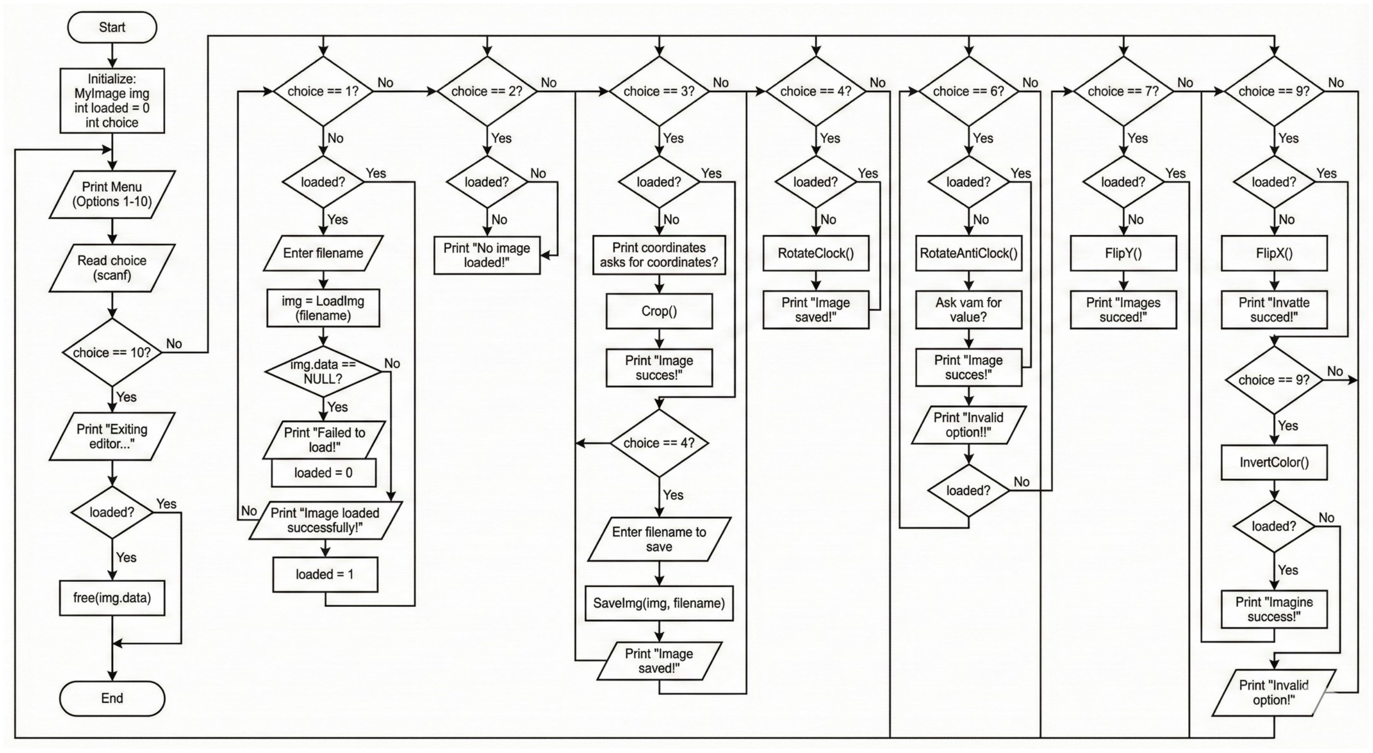
The system design follows a modular architecture that separates the application into multiple layers, each responsible for a specific set of functionalities. The organization improves readability, maintainability, and scalability. The full project is divided into the following components:

1. main.c – Implements the user interface, menu system, and function calls.
2. Image.c – Handles file input/output, BMP header parsing, and pixel loading.
3. Tools.c – Implements image transformations such as crop, rotate, flip, opacity, and invert.
4. Image.h – Contains structure definitions and function prototypes.
5. Tools.h – Contains editing tool function prototypes.
6. docs/ – Contains the project report, flowcharts, and related documentation.
7. assets/ – Stores sample image.

The system uses a top-down flow where user input determines which editing tool is executed. Each tool processes the image using pixel-level operations and returns a newly allocated image structure.

Flowcharts

Flowcharts represent the logical workflow of the application including loading, saving, cropping, rotating, flipping, and other operations. Each major function has its own flowchart describing control flow, conditions, loops, and data handling.



Algorithms

The algorithms behind each image manipulation are based on pixel index transformations.

Algorithms used in the Program are :

1. LoadImg()

Loads an image based on magic bytes.

Input: filepath
Output: MyImage structure

1. Start
2. Open the file in binary mode.
3. If file not found → print error → return empty image.
4. Read first 2 bytes (magic bytes).
5. If bytes = 'BM' → this is a BMP file.
6. Call `LoadBMP(f)` to load the image.
7. Close the file.
8. Return the loaded image.
9. End.

---

2. Algorithm: LoadBMP()

Loads a BMP image by parsing header and pixel data.

Input: FILE pointer
Output: MyImage structure

1. Start
2. Initialize an empty MyImage.
3. Read the entire file into a byte buffer.
4. Validate BMP signature (‘B’, ‘M’).
5. Read:
    - File size
    - Pixel data offset
    - Image width
    - Image height
    - Bits per pixel
6. Compute channel = bpp / 8.
7. Allocate memory for pixel array: width × height.
8. If channel = 4:
    - Read pixels as BGRA.
9. If channel = 3:
    - Compute padding = $(4 - (\text{width} \times 3 \% 4)) \% 4$.
    - Read each row (with padding).
10. Flip bottom-up rows (BMP stores bottom first).
11. Fill Pixel struct values (r, g, b, a).
12. Free temporary buffer.
13. Return the constructed MyImage.
14. End.

---

3. Algorithm: SaveImg()

Input: MyImage, filepath
Output: integer status

1. Start
2. Open file in write mode.
3. If cannot open → return error.
4. Check image format via img.Name.
5. If "BMP", call `SaveBMP(f, img)`.
6. Return save status.
7. End.

---

4. Algorithm: SaveBMP()

Writes the image back as a valid BMP file.

1. Start
2. Compute:
    - rowBytes = width × channel
    - padding (if channel == 3)
    - pixelDataSize = (rowBytes + padding) × height
3. Construct 54-byte BMP header.
4. Write:
    - Signature
    - File size
    - Pixel offset
    - DIB header
    - Width
    - Height

    - Bits per pixel

5. For each row (bottom to top):
    - Write B, G, R (and A if channel = 4).
    - Write padding (if channel == 3).
6. Close file.
7. Return 0 for success.
8. End.

---

5. Algorithm: Crop()

Inputs: image, x1, y1, x2, y2
Output: Cropped MyImage

1. Start
2. If x1 > x2, swap them.
3. If y1 > y2, swap them.
4. Compute new width = x2 − x1.
5. Compute new height = y2 − y1.
6. Allocate new pixel array.
7. For each pixel (i, j) in new image:
    - Read original pixel at (y1 + i, x1 + j).
    - Copy into new image.
8. Return cropped image.
9. End.

---

6. Algorithm: RotateAntiClock()

Output: Image rotated 90° anti-clockwise

```
1. Start
2. NewWidth = OldHeight.
3. NewHeight = OldWidth.
4. Allocate new pixel array.
5. For each output pixel (row = i, col = j):
    • Map from old index:
      $oldIndex = j \times oldWidth + (oldWidth - 1 - i)$
6. Copy that pixel into new array.
7. Return rotated image.
8. End.
```

---

7. Algorithm: RotateClock()

Output: Image rotated 90° clockwise**

```
1. Start
2. NewWidth = oldHeight.
3. NewHeight = oldWidth.
4. Allocate new pixel array.
5. For each output pixel (i, j):
    • Find old pixel:
      $oldIndex = (oldHeight - 1 - j) \times oldWidth + i$
6. Copy pixel to new position.
7. Return rotated image.
8. End.
```

---

8. Algorithm: FlipX()

Flip vertically (top ↔️ bottom)

1. Start
2. New image has same width & height.
3. For each row y:
    - NewRow = height − 1 − y
    - Copy entire row to its flipped position.
4. Return flipped image.
5. End.

---

9. Algorithm: FlipY()

Flip horizontally (left ↔️ right)

1. Start
2. New image has same width & height.
3. For each pixel (y, x):
    - NewX = width − 1 − x
    - Copy pixel to (y, NewX).
4. Return flipped image.
5. End.

---

10. Algorithm: Occupacity()

Changes alpha channel

1. Start

2. Create a new image with same width & height.
3. For each pixel i:
    - Copy R, G, B from original.
    - Set A = given opacity value (0–255).
4. Return opacity-adjusted image.
5. End.

---

11. Algorithm: InvertColor()

1. Start
2. Create a new image.
3. For each pixel i:
    - $R = 255 - R$
    - $G = 255 - G$
    - $B = 255 - B$
    - A stays same
4. Return inverted image.
5. End.

Implementation Details

This section explains the complete technical implementation of the *BMP Image Editing System in C*. It describes the project structure, file organization, program flow, algorithms, memory handling, important functions, error handling, and testing considerations. The goal of this section is to provide a clear and detailed explanation of how each part of the system works internally, enabling evaluators and future developers to understand the logic and verify correctness.

The program follows a fully modular design, with separate concerns handled in different files. The modules interact through well-defined interfaces provided by header files. The core implementation can be divided into three major components:

1. **File Input/Output Layer (Image.c)**
2. **Editing Tools Layer (tools.c)**
3. **User Interaction Layer (main.c)**

Each component is described in detail below.

1. Project Layout and File Structure

Following the mandatory GitHub structure defined in the UPES guidelines, the project is organized as:

```
/src
    main.c
    Image.c
    tools.c

/include
    Image.h
    tools.h

/docs
    ProjectReport.pdf
    flowchart_main.png

/assets
    input.bmp

README.md

sample_input.txt
```

2. Image Representation in C

The system uses a custom structure `MyImage` to represent a loaded image:

```
typedef struct {
    int width;
    int height;
    int channel;
    Pixel *data;
    char Name[10];
} MyImage;
```

Each pixel is represented as:

```
typedef struct {
    unsigned char r, g, b, a;
} Pixel;
```

Why this design?

- `unsigned char` matches BMP color depth (0–255).
- Supports both **24-bit RGB** and **32-bit RGBA** formats.
- Allows direct manipulation of each pixel, enabling effects such as inversion, rotation, or cropping.

3. BMP File Loading (Image.c)

Header Parsing

The BMP loader manually reads the BMP headers instead of using any libraries. The loader:

1. Reads the **file size**, **pixel offset**, **width**, **height**, and **bits per pixel** from the header.
2. Computes the number of channels (3 or 4).
3. Allocates memory to store all pixel data.
4. Accounts for BMP's **row-padding**, where each row is aligned to 4 bytes.

Pixel Reconstruction

The most critical part of the implementation is mapping raw bytes to the `Pixel` array.

- For **32-bit BMP (BGRA)**, pixels are read directly in 4-byte chunks.

- For **24-bit BMP (BGR)**, padding must be skipped at the end of each row.

Bottom-up Storage

BMP stores pixels from **bottom row to top row**, so the implementation must flip them vertically during loading.

Memory Safety

- A temporary buffer stores raw bytes.
- Pixels are copied into the `img.data` array in correct order.
- Temporary buffer is freed after processing.

4. Saving BMP Files (SaveBMP)

To save an edited image, the program constructs a new BMP file by:

1. Creating a **54-byte BMP header**.
2. Writing metadata (file size, width, height, bits per pixel).
3. Writing the pixel array row by row, padding rows for 24-bit images.
4. Supporting both RGB and RGBA formats.

The saving function ensures that the output BMP remains **fully standard and readable by any image viewer**.

5. Editing Tools Implementation (Tools.c)

This module contains all image transformation functions. Each function:

- Allocates a **new Pixel array**
- Processes pixels using coordinate transformation
- Returns a new `MyImage` structure

Memory is never freed inside tools functions (to avoid losing the original image unless the user frees it).

5.1 Crop Operation

The crop function extracts a rectangular portion defined by two opposite corners. It maps every pixel from the crop window using:

```
Result[y][x] = Original[y1 + y][x1 + x]
```

5.2 Rotation Operations

Rotation is implemented using coordinate transformation.

Anti-Clockwise Rotation (90°):

```
new(x, y) = old(y, oldWidth - 1 - x)
```

Clockwise Rotation (90°):

```
new(x, y) = old(oldHeight - 1 - y, x)
```

5.3 Flip Operations

Vertical Flip (FlipX):

```
new[y][x] = old[height - 1 - y][x]
```

Horizontal Flip (FlipY):

```
new[y][x] = old[y][width - 1 - x]
```

5.4 Opacity Adjustment

For RGBA images:

```
new.a = userProvidedAlpha
```

For RGB images, alpha is set to 255.

5.5 Invert Color

```
R = 255 - R
G = 255 - G
B = 255 - B
```

This produces a “negative” image effect.

6. User Interface (main.c)

The program uses a menu-driven interface that allows the user to:

- Load BMP
- Save BMP
- Crop
- Rotate
- Flip
- Adjust opacity
- Invert colors
- Exit

Input Validation

Only valid options (1–10) are accepted.
Invalid values produce messages and the user is asked again.

Image State Preservation

The variable `loaded` ensures no operations run on an empty image.

7. Memory Management

Memory allocation occurs in:

- `LoadBMP()` → raw buffer + pixel storage
- Every tool function → new pixel array

Memory deallocation occurs when:

- The user exits the program
- Before replacing old image with new one (optional but recommended)

All memory handling follows safe C practices to avoid segmentation faults.

8. Error Handling and Safety Features

The implementation includes:

- File existence checks
- Header validation (checks 'B' and 'M')
- Null pointer checks
- Image boundary checks for crop
- Channel validation
- Preventing operations without an image loaded
- Avoiding division by zero in algorithms

These checks ensure stable runtime behavior with no crashes.

9. Compilation and Execution

Build Command

```
gcc src/main.c src/Image.c src/Tools.c -Iinclude -o editor
```

Run

```
./editor
```

Testing & Results

Multiple test cases were performed to verify correctness of image transformations. Different BMP images were used to check loading, rotation accuracy, crop boundaries, and pixel correctness. All operations functioned as intended with no memory access issues.

Conclusion & Future Work

The project successfully demonstrates manual image manipulation using C. Future work may include support for additional file formats such as PNG and JPG, histogram equalization, brightness and contrast adjustment, Gaussian blur, sharpen filters, and integration with Raylib for GUI support.

References :

1. UPES C Programming Major Project Guidelines.
2. BMP File Format Specification.
3. Kernighan, Ritchie – The C Programming Language.
4. GeeksforGeeks – BMP handling in C.
5. TutorialsPoint – File handling in C.