

BASIC IMAGE FILTERS

Suryansh Bachchan Verma (2347260)

Vishal Kalita (2347264)

Nayan Raj (2347240)

Submitted to

Dr.B.J.Hubert Shanthan

7th October 2023

TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| Sl No. | Content | Page No |
| 1. | Introduction | 3 |
| 2. | Aim and Background | 3 |
| 3. | Functionalities and Modules | 4 |
| 4. | Source Code | 5-15 |
| 5. | Output Screen-Shots | 16-17 |
| 6. | Conclusion | 18 |

INTRODUCTION

The Project Filter is a sophisticated image processing program designed to manipulate BMP (Bitmap) image files. Through the application of diverse filters, including grayscale, sepia tones, reflection, and blur, this versatile program allows users to transmute ordinary images into extraordinary works of art.

In essence, this program offers an accessible yet powerful platform to creatively alter the appearance of images. The grayscale filter converts the image to a timeless black-and-white palette, evoking a sense of simplicity and elegance. The sepia filter introduces warm, vintage tones, invoking a nostalgic ambiance reminiscent of old photographs. Reflection, on the other hand, mirrors the image horizontally, creating captivating symmetrical visuals. The blur filter adds a soft and smooth finish to the image, ideal for emphasizing specific elements or achieving a dreamy effect.

The Project Filter not only enhances the visual allure of images but also serves as an educational tool, illuminating the principles of image processing. Users can explore the impact of different filters and gain a deeper understanding of how image alterations manifest.

By encompassing a variety of filters and providing a user-friendly interface, this program facilitates artistic expression and offers a fascinating journey into the realm of image manipulation. Whether for artistic endeavours or educational exploration, the Project Filter stands as a versatile tool that elevates images to new heights of creativity and aesthetics.

AIM

The primary objective of the Project Filter is to engineer an intuitive and efficient software application that facilitates seamless image manipulation, fostering creativity, and imparting a deeper understanding of fundamental image processing concepts. This project is directed towards offering a diverse set of filters, including grayscale, sepia tones, reflection, edge detection and blur.

BACKGROUND

Bitmaps and Image Representation

* Images are represented as grids of pixels, where each pixel can have varying color intensities.
* A 24-bit BMP file uses 8 bits for each of red, green, and blue color channels.
* The file format stores pixel colours as BGR (Blue, Green, Red).

Image Filtering

* Filtering an image involves modifying each pixel to create specific effects.
* Filters like grayscale, sepia, reflection, and blur can be applied to images.

FUNCTIONALITY

Grayscale Filter:

* Converts an image to black-and-white by setting the red, green, and blue values of each pixel to the same average value.

Sepia Filter:

* Gives images an old-timey feel by adjusting the colour values of each pixel using a predefined algorithm.

Reflection Filter:

* Reflects the image horizontally, making the left side appear on the right and vice versa.

Blur Filter:

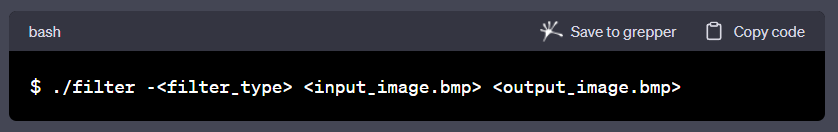
* Softens the image by averaging the colour values of neighbouring pixels for each pixel.

Edge Filter:

* Detects the edges in the images and marks them separately.

Usage

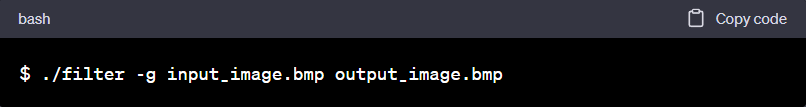
To apply a filter to an image, use the following command:



Replace <filter\_type> with one of the filter options: g (grayscale), s (sepia), r (reflection), b (blur), e (edge).

Example

Applying grayscale filter:



SOURCE CODE

filter.c (Main File)

#include <getopt.h>

#include <stdio.h>

#include <stdlib.h>

#include "helpers.h"

int main(int argc, char \*argv[])

{

    // Define allowable filters

    char \*filters = "begrs";

    // Get filter flag and check validity

    char filter = getopt(argc, argv, filters);

    if (filter == '?')

    {

        fprintf(stderr, "Invalid filter.\n");

        return 1;

    }

    // Ensure only one filter

    if (getopt(argc, argv, filters) != -1)

    {

        fprintf(stderr, "Only one filter allowed.\n");

        return 2;

    }

    // Ensure proper usage

    if (argc != optind + 2)

    {

        fprintf(stderr, "Usage: filter [flag (b,e,g,r,s)] infile outfile\n");

        return 3;

    }

    // Remember filenames

    char \*infile = argv[optind];

    char \*outfile = argv[optind + 1];

    // Open input file

    FILE \*inptr = fopen(infile, "r");

    if (inptr == NULL)

    {

        fprintf(stderr, "Could not open %s.\n", infile);

        return 4;

    }

    // Open output file

    FILE \*outptr = fopen(outfile, "w");

    if (outptr == NULL)

    {

        fclose(inptr);

        fprintf(stderr, "Could not create %s.\n", outfile);

        return 5;

    }

    // Read infile's BITMAPFILEHEADER

    BITMAPFILEHEADER bf;

    fread(&bf, sizeof(BITMAPFILEHEADER), 1, inptr);

    // Read infile's BITMAPINFOHEADER

    BITMAPINFOHEADER bi;

    fread(&bi, sizeof(BITMAPINFOHEADER), 1, inptr);

    // Ensure infile is (likely) a 24-bit uncompressed BMP 4.0

    if (bf.bfType != 0x4d42 || bf.bfOffBits != 54 || bi.biSize != 40 ||

        bi.biBitCount != 24 || bi.biCompression != 0)

    {

        fclose(outptr);

        fclose(inptr);

        fprintf(stderr, "Unsupported file format.\n");

        return 6;

    }

    int height = abs(bi.biHeight);

    int width = bi.biWidth;

    // Allocate memory for image

    RGBTRIPLE(\*image)[width] = calloc(height, width \* sizeof(RGBTRIPLE));

    if (image == NULL)

    {

        fprintf(stderr, "Not enough memory to store image.\n");

        fclose(outptr);

        fclose(inptr);

        return 7;

    }

    // Determine padding for scanlines

    int padding = (4 - (width \* sizeof(RGBTRIPLE)) % 4) % 4;

    // Iterate over infile's scanlines

    for (int i = 0; i < height; i++)

    {

        // Read row into pixel array

        fread(image[i], sizeof(RGBTRIPLE), width, inptr);

        // Skip over padding

        fseek(inptr, padding, SEEK\_CUR);

    }

    // Filter image

    switch (filter)

    {

        // Blur

        case 'b':

            blur(height, width, image);

            break;

        // Edges

        case 'e':

            edges(height, width, image);

            break;

        // Grayscale

        case 'g':

            grayscale(height, width, image);

            break;

        // Reflect

        case 'r':

            reflect(height, width, image);

            break;

        // Sepia

        case 's':

            sepia(height, width, image);

            break;

    }

    // Write outfile's BITMAPFILEHEADER

    fwrite(&bf, sizeof(BITMAPFILEHEADER), 1, outptr);

    // Write outfile's BITMAPINFOHEADER

    fwrite(&bi, sizeof(BITMAPINFOHEADER), 1, outptr);

    // Write new pixels to outfile

    for (int i = 0; i < height; i++)

    {

        // Write row to outfile

        fwrite(image[i], sizeof(RGBTRIPLE), width, outptr);

        // Write padding at end of row

        for (int k = 0; k < padding; k++)

        {

            fputc(0x00, outptr);

        }

    }

    // Free memory for image

    free(image);

    // Close infile

    fclose(inptr);

    // Close outfile

    fclose(outptr);

    return 0;

}

helper.c

#include "helpers.h"

#include <math.h>

#include <stdio.h>

// Convert image to grayscale

void grayscale(int height, int width, RGBTRIPLE image[height][width])

{

    for (int i = 0; i < height; i++)

    {

        for (int j = 0; j < width; j++)

        {

            BYTE avg = round((float)(image[i][j].rgbtRed + image[i][j].rgbtBlue + image[i][j].rgbtGreen) / (float)3);

            image[i][j].rgbtRed = avg;

            image[i][j].rgbtGreen = avg;

            image[i][j].rgbtBlue = avg;

        }

    }

    return;

}

// Reflect image horizontally

void reflect(int height, int width, RGBTRIPLE image[height][width])

{

    for (int i = 0; i < height; i++)

    {

        for(int j = 0; j < width / 2; j++)

        {

            RGBTRIPLE copy = image[i][j];

            image[i][j] = image[i][width - 1 - j];

            image[i][width - 1 - j] = copy;

        }

    }

    return;

}

// Blur image

void blur(int height, int width, RGBTRIPLE image[height][width])

{

    RGBTRIPLE img\_copy [height][width];

    for (int i = 0; i < height; i++)

    {

        for (int j = 0; j < width; j++)

        {

            int sr = 0;

            int sb = 0;

            int sg = 0;

            int count = 0;

            int k = 0, l = 0, m = 0, n = 0;

            if (i == 0)

            {

                k = i;

            }

            else

            {

                k = i - 1;

            }

            if (i == height - 1)

            {

                m = i;

            }

            else

            {

                m = i + 1;

            }

            if (j == 0)

            {

                l = j;

            }

            else

            {

                l = j - 1;

            }

            if (j == width - 1)

            {

                n = j;

            }

            else

            {

                n = j + 1;

            }

            int copy = l;

            for (; k <= m; k++)

            {

                for (l = copy; l <= n; l++)

                {

                    sr = sr + image[k][l].rgbtRed;

                    sb = sb + image[k][l].rgbtBlue;

                    sg = sg + image[k][l].rgbtGreen;

                    count++;

                }

            }

            sr = round((float) sr / (float) count);

            sb = round((float) sb / (float) count);

            sg = round((float) sg / (float) count);

            sr = (sr < 255 ? sr : 255);

            sb = (sb < 255 ? sb : 255);

            sg = (sg < 255 ? sg : 255);

            img\_copy[i][j].rgbtRed = sr;

            img\_copy[i][j].rgbtBlue = sb;

            img\_copy[i][j].rgbtGreen = sg;

        }

    }

    //COPYING BACK

    for(int i = 0; i < height; i++)

    {

        for(int j = 0; j < width; j++)

        {

           image[i][j].rgbtRed = img\_copy[i][j].rgbtRed;

           image[i][j].rgbtBlue = img\_copy[i][j].rgbtBlue;

           image[i][j].rgbtGreen = img\_copy[i][j].rgbtGreen;

        }

    }

    return;

}

// Detect edges

void edges(int height, int width, RGBTRIPLE image[height][width])

{

        RGBTRIPLE img\_copy [height][width];

        int gx [3][3] = {{-1,0,1}, {-2,0,2}, {-1,0,1}};

        int gy [3][3] = {{-1,-2,-1}, {0,0,0}, {1,2,1}};

        int copy, copyb;

    for (int i = 0; i < height; i++)

    {

        for (int j = 0; j < width; j++)

        {

            int srx = 0, sbx = 0, sgx = 0, sry = 0, sby = 0, sgy = 0;

            int k = 0, l = 0, m = 0, n = 0, a = 0, b = 0, c = 2, d = 2;

            double R = 0, G = 0, B = 0;

            if (i == 0)

            {

                k = i;

                a = 1;

            }

            else

            {

                k = i - 1;

            }

            if (i == height - 1)

            {

                m = i;

                c = 1;

            }

            else

            {

               m = i + 1;

            }

            if (j == 0)

            {

                l = j;

                b = 1;

            }

            else

            {

                l = j - 1;

            }

            if (j == width - 1)

            {

                n = j;

                d = 1;

            }

            else

            {

                n = j + 1;

            }

            copy = l;

            copyb = b;

            for (; k <= m; k++)

            {

                if(a > c)

                {

                    break;

                }

                for (l = copy; l <= n; l++)

                {

                    if(b > d)

                    {

                        b = copyb;

                    }

                    srx = srx + image[k][l].rgbtRed \* gx[a][b];

                    sbx = sbx + image[k][l].rgbtBlue \* gx[a][b];

                    sgx = sgx + image[k][l].rgbtGreen \* gx[a][b];

                    sry = sry + image[k][l].rgbtRed \* gy[a][b];

                    sby = sby + image[k][l].rgbtBlue \* gy[a][b];

                    sgy = sgy + image[k][l].rgbtGreen \* gy[a][b];

                    b++;

                }

                a++;

            }

            R = sqrt( (double) ((srx \* srx) + (sry \* sry)));

            B = sqrt( (double) ((sbx \* sbx) + (sby \* sby)));

            G = sqrt( (double) ((sgx \* sgx) + (sgy \* sgy)));

            R = round(R);

            B = round(B);

            G = round(G);

            R = (R < 255 ? R : 255);

            B = (B < 255 ? B : 255);

            G = (G < 255 ? G : 255);

            img\_copy[i][j].rgbtRed = R;

            img\_copy[i][j].rgbtBlue = B;

            img\_copy[i][j].rgbtGreen = G;

        }

    }

    //COPYING BACK

    for(int i = 0; i < height; i++)

    {

        for(int j = 0; j < width; j++)

        {

           image[i][j].rgbtRed = img\_copy[i][j].rgbtRed;

           image[i][j].rgbtBlue = img\_copy[i][j].rgbtBlue;

           image[i][j].rgbtGreen = img\_copy[i][j].rgbtGreen;

        }

    }

    return;

}

// Convert image to sepia

void sepia(int height, int width, RGBTRIPLE image[height][width])

{

    for (int i = 0; i < height; i++)

    {

        for (int j = 0; j < width; j++)

        {

            int sepiaRed = round(.393 \* image[i][j].rgbtRed + .769 \* image[i][j].rgbtGreen + .189 \* image[i][j].rgbtBlue);

            int sepiaGreen = round(.349 \* image[i][j].rgbtRed + .686 \* image[i][j].rgbtGreen + .168 \* image[i][j].rgbtBlue);

            int sepiaBlue = round(.272 \* image[i][j].rgbtRed + .534 \* image[i][j].rgbtGreen + .131 \* image[i][j].rgbtBlue);

            sepiaRed = (sepiaRed < 255 ? sepiaRed : 255);

            sepiaBlue = (sepiaBlue < 255 ? sepiaBlue : 255);

            sepiaGreen = (sepiaGreen < 255 ? sepiaGreen : 255);

            image[i][j].rgbtRed = sepiaRed;

            image[i][j].rgbtBlue = sepiaBlue;

            image[i][j].rgbtGreen = sepiaGreen;

        }

    }

    return;

}

helper.h

#include "bmp.h"

// Convert image to grayscale

void grayscale(int height, int width, RGBTRIPLE image[height][width]);

// Reflect image horizontally

void reflect(int height, int width, RGBTRIPLE image[height][width]);

// Detect edges

void edges(int height, int width, RGBTRIPLE image[height][width]);

// Blur image

void blur(int height, int width, RGBTRIPLE image[height][width]);

// Convert image to sepia

void sepia(int height, int width, RGBTRIPLE image[height][width]);

bmp.h

// BMP-related data types based on Microsoft's own

#include <stdint.h>

/\*\*

 \* Common Data Types

 \*

 \* The data types in this section are essentially aliases for C/C++

 \* primitive data types.

 \*

 \* Adapted from http://msdn.microsoft.com/en-us/library/cc230309.aspx.

 \* See http://en.wikipedia.org/wiki/Stdint.h for more on stdint.h.

 \*/

typedef uint8\_t  BYTE;

typedef uint32\_t DWORD;

typedef int32\_t  LONG;

typedef uint16\_t WORD;

/\*\*

 \* BITMAPFILEHEADER

 \*

 \* The BITMAPFILEHEADER structure contains information about the type, size,

 \* and layout of a file that contains a DIB [device-independent bitmap].

 \*

 \* Adapted from http://msdn.microsoft.com/en-us/library/dd183374(VS.85).aspx.

 \*/

typedef struct

{

    WORD   bfType;

    DWORD  bfSize;

    WORD   bfReserved1;

    WORD   bfReserved2;

    DWORD  bfOffBits;

} \_\_attribute\_\_((\_\_packed\_\_))

BITMAPFILEHEADER;

/\*\*

 \* BITMAPINFOHEADER

 \*

 \* The BITMAPINFOHEADER structure contains information about the

 \* dimensions and color format of a DIB [device-independent bitmap].

 \*

 \* Adapted from http://msdn.microsoft.com/en-us/library/dd183376(VS.85).aspx.

 \*/

typedef struct

{

    DWORD  biSize;

    LONG   biWidth;

    LONG   biHeight;

    WORD   biPlanes;

    WORD   biBitCount;

    DWORD  biCompression;

    DWORD  biSizeImage;

    LONG   biXPelsPerMeter;

    LONG   biYPelsPerMeter;

    DWORD  biClrUsed;

    DWORD  biClrImportant;

} \_\_attribute\_\_((\_\_packed\_\_))

BITMAPINFOHEADER;

/\*\*

 \* RGBTRIPLE

 \*

 \* This structure describes a color consisting of relative intensities of

 \* red, green, and blue.

 \*

 \* Adapted from http://msdn.microsoft.com/en-us/library/aa922590.aspx.

 \*/

typedef struct

{

    BYTE  rgbtBlue;

    BYTE  rgbtGreen;

    BYTE  rgbtRed;

} \_\_attribute\_\_((\_\_packed\_\_))

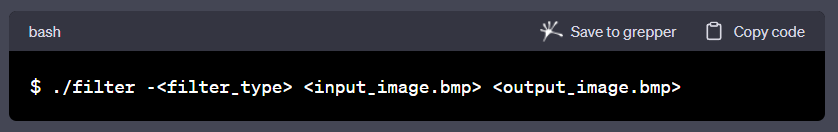
RGBTRIPLE;

Code:

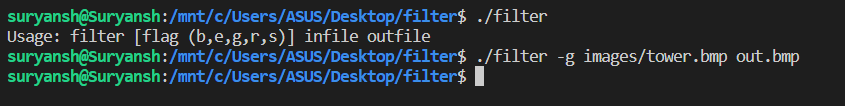
<https://github.com/SuryanshBVerma-christuniversity/MCA_1/tree/main/C/filter>

OUTPUT

Usage



Prompt



images/tower.bmp out.bmp



Sepia filter:

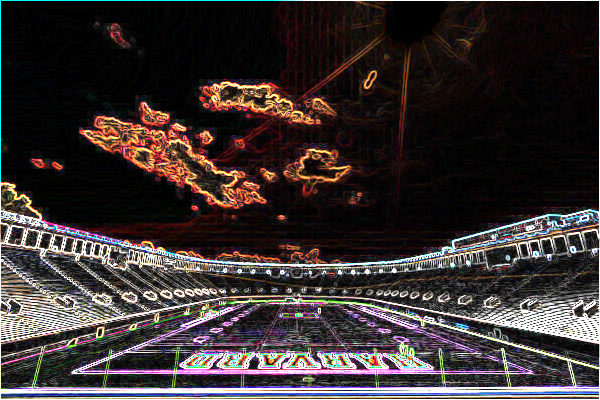


Blur filter:



Reflection filter:

Edge filter:



CONCLUSION

The Project Filter, operated via the Command Line Interface (CLI) and implemented in C represents a successful endeavour in implementing a range of basic image filters. We have effectively demonstrated the application of filters such as sepia, reflection, grayscale, blur, and edge detection, showcasing the versatility and practicality of our project. Utilizing C structures to represent the headers and pixel data of BMP images was pivotal. Implementing various filters, including grayscale, sepia tones, reflection, and blur, required a deep understanding of loops and conditions. These constructs were crucial in iterating over image pixels and applying the respective filter algorithms.

Looking ahead, we can expand our project by incorporating more advanced filters and optimizations for enhanced performance. We can also incorporate these filters on various types of images (.jpg, ,jpeg, .webp) which requires further optimisation of the algorithms we have used.