Университет ИТМО
Кафедра ВТ

Лабораторная работа №5
Низкоуровневое программирование

Выполнил: Федоров Сергей
Группа: P33113

Санкт-Петербург
2020 г.

Задание лабораторной работы:

Реализовать на языке С считывание и запись BMP картинки, а так же функцию поворота этого изображения на произвольный угол.

Выполнение:

```
Lab5/
.
├── Makefile
├── bmp.c
├── bmp.h
├── image.c
├── image.h
├── images
│   ├── lab6_result.bmp
│   ├── lab6_test_big.bmp
│   └── lab6_test_smoll.bmp
└── main.c
```

main.c

```c
//
// Created by Sergey Fedorov on 11/6/20.
//

#include "bmp.h"

#include <stdio.h>

#define PATH_TO_IMAGE "images/lab6_test_big.bmp"
#define PATH_TO_IMAGE_RES "images/lab6_result.bmp"
#define ANGLE 45

int main() {
    struct bmp_image* b_image;
    FILE* test;
    test = fopen(PATH_TO_IMAGE, "rb");

    bmp_image_read(&b_image, test);

    struct image* image;

    bmp_to_image(b_image, &image, 0);

    printf("SIZE: %d %d\n", image->height, image->width);

    image_rotate(image, ANGLE * M_PI / 180.0);

    struct bmp_image* b_image_2;

    image_to_bmp(image, &b_image_2, 1);

    printf("TOP HEADER: %s %d %d %d\n",
           b_image->header.bfType, b_image->header.bfSize, b_image->header.bfReserved, b_image-
>header.bfOffBits);

    printf("TOP HEADER: %s %d %d %d\n",
           b_image_2->header.bfType, b_image_2->header.bfSize, b_image_2->header.bfReserved,
b_image_2->header.bfOffBits);
```

```
    printf("BOTTOM HEADER: %d %d %d %d %d %d %d %d %d %d %d\n",
           b_image->header.biSize,
           b_image->header.biWidth,
           b_image->header.biHeight,
           b_image->header.biPlanes,
           b_image->header.biBitCount,
           b_image->header.biCompression,
           b_image->header.biSizeImage,
           b_image->header.biXPelsPerMeter,
           b_image->header.biYPelsPerMeter,
           b_image->header.biClrUsed,
           b_image->header.biClrImportant
    );

    printf("BOTTOM HEADER: %d %d %d %d %d %d %d %d %d %d %d\n",
           b_image_2->header.biSize,
           b_image_2->header.biWidth,
           b_image_2->header.biHeight,
           b_image_2->header.biPlanes,
           b_image_2->header.biBitCount,
           b_image_2->header.biCompression,
           b_image_2->header.biSizeImage,
           b_image_2->header.biXPelsPerMeter,
           b_image_2->header.biYPelsPerMeter,
           b_image_2->header.biClrUsed,
           b_image_2->header.biClrImportant
    );

    FILE* wtest;
    wtest = fopen(PATH_TO_IMAGE_RES, "wb");

    bmp_image_write(b_image_2, wtest);
}
```

image.h

```
//
// Created by Sergey Fedorov on 11/6/20.
//
#include <stdint.h>


#ifndef LOW_LEVEL_PROGRAMMING_ITMO_2020_IMAGE_H
#define LOW_LEVEL_PROGRAMMING_ITMO_2020_IMAGE_H

#define M_PI (3.14159265358979323846)

struct pixel {
    uint8_t b, g, r;
};

struct image {
    int32_t width, height;
    struct pixel* data;
};

void free_image(struct image* image);

void image_rotate(struct image* image, double angle);

// TODO
//void image_blur(struct image* image);
//void image_dilate(struct image* image);
//void image_erode(struct image* image);

#endif //LOW_LEVEL_PROGRAMMING_ITMO_2020_IMAGE_H
```

image.c

```c
//
// Created by Sergey Fedorov on 11/6/20.
//

#include "image.h"

#include <float.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>

void free_image(struct image* image) {
    if (image != NULL) {
        free(image->data);
        free(image);
    }

}

int abs_to_int(double a){
    return abs((int) a);
}

double min(double a, double b) {
    return a < b ? a : b;
}

double max(double a, double b) {
    return a > b ? a : b;
}

double sind(double angle) {
    double angleradians = angle * M_PI / 180.0;
    return sin(angleradians) * M_PI / 180.0;
}

double cosd(double angle) {
    double angleradians = angle * M_PI / 180.0;
    return cos(angleradians) * M_PI / 180.0;
}

struct positioned_pixel{
    double x, y;
    struct pixel pixel;
};

void image_rotate(struct image* image, double angle) {
    double center_x, center_y, alpha,
    min_x = DBL_MAX,min_y = DBL_MAX,
    max_x = -DBL_MAX, max_y = -DBL_MAX;

    uint32_t x, y, i, j, k, base_x, base_y, count;

    count = image->width * image->height;


    struct positioned_pixel* pixels;
    pixels = malloc(sizeof(struct positioned_pixel) * count);

    center_x = ((double) image->width) / 2;
    center_y = ((double) image->height) / 2;

    for (y = 0, i = 0; y < image->height; y++) {
        for (x = 0; x < image->width; x++, i++) {

            // Rotation matrix
            pixels[i].x = center_x + (x - center_x) * cos(angle) - (y - center_y) * sin(angle);
            pixels[i].y = center_y + (x - center_x) * sin(angle) + (y - center_y) * cos(angle);
            pixels[i].pixel = image->data[i];
            min_x = min(min_x, pixels[i].x);
            min_y = min(min_y, pixels[i].y);

            max_x = max(max_x, pixels[i].x);
            max_y = max(max_y, pixels[i].y);
        }
    }

    image->width = ceil(max_x - min_x + 1);
    image->height = ceil(max_y - min_y + 1);
    free(image->data);

    // calloc -> make background black
    image->data = calloc(image->width * image->height, sizeof(struct pixel));

    for (i = 0; i < count; ++i) {
        pixels[i].x -= min_x;
        pixels[i].y -= min_y;

        base_x = ceil(pixels[i].x);
        base_y = ceil(pixels[i].y);

        // Mix colours by deviation
        // Positions:
        //             top (1)
        //             ^^^^^
        // left(4) <- center (0) -> right(2)
        //             vvvvv
        //            bottom (3)

        for (k = 0; k < 5; ++k) {
            switch (k) {
                case 0:
                    x = base_x;
                    y = base_y;
                    alpha = (1 - abs_to_int(pixels[i].x - x)) * (1 - abs_to_int(pixels[i].y -
y));

                    break;

                case 1:
                    x = base_x;
                    y = base_y - 1;
                    alpha = (1 - abs_to_int(x - pixels[i].x)) * (1 - min(pixels[i].y - y, 1));
                    break;

                case 2:
                    x = base_x + 1;
                    y = base_y;
                    alpha = (1 - min(x - pixels[i].x, 1)) * (1 - abs_to_int(y - pixels[i].y));
                    break;

                case 3:
                    x = base_x;
                    y = base_y + 1;
                    alpha = (1 - abs_to_int( x - pixels[i].x)) * (1 - min(y - pixels[i].y, 1));
                    break;

                case 4:
                    x = base_x - 1;
                    y = base_y;
                    alpha = (1 - min(pixels[i].x - x, 1)) * (1 - abs_to_int(y - pixels[i].y));
                    break;
            }

            if (x >= 0 && x < image->width && y >= 0 && y < image->height) {
                j = y * image->width + x;

                image->data[j].r += (pixels[i].pixel.r - image->data[j].r) * alpha;
                image->data[j].g += (pixels[i].pixel.g - image->data[j].g) * alpha;
                image->data[j].b += (pixels[i].pixel.b - image->data[j].b) * alpha;
            }
        }
    }

    free(pixels);
}
```

bmp.h

```
//
// Created by Sergey Fedorov on 11/6/20.
//

#include "image.h"

#include <stdio.h>
#include <stdint.h>

#ifndef LOW_LEVEL_PROGRAMMING_ITMO_2020_BMP_H
#define LOW_LEVEL_PROGRAMMING_ITMO_2020_BMP_H

struct __attribute__((packed)) bmp_header {
    char     bfType[2];
    uint32_t bfSize;
    uint32_t bfReserved;
    uint32_t bfOffBits;

    uint32_t biSize;
    int32_t  biWidth;
    int32_t  biHeight;
    uint16_t biPlanes;
    uint16_t biBitCount;
    uint32_t biCompression;
    uint32_t biSizeImage;
    int32_t  biXPelsPerMeter;
    int32_t  biYPelsPerMeter;
    uint32_t biClrUsed;
    uint32_t biClrImportant;
};

struct bmp_pixel;

struct bmp_image {
    struct bmp_header header;
    struct bmp_pixel* pixels;
};

enum bmp_read_result {
    READ_OK = 0,
    READ_INVALID_SIGNATURE,
    READ_INVALID_BITS,
    READ_INVALID_HEADER,
    READ_INVALID_BMP_FORMAT,
    READ_INVALID_BAD_PIXEL
};

enum bmp_write_result {
    WRITE_OK = 0,
    WRITE_BAD_HEADER,
    WRITE_BAD_BODY,
    WRITE_BAD_OFFSET
};

enum bmp_read_result bmp_image_read(struct bmp_image** bmp_image, FILE* file);
void free_bmp_image(struct bmp_image* image);

void bmp_to_image(struct bmp_image* bmp_image, struct image** new_image_p, int free_bmp);
void image_to_bmp(struct image* image, struct bmp_image** new_bmp_image_p, int free_image);

enum bmp_write_result bmp_image_write(const struct bmp_image* bmp_image, FILE * file);

#endif //LOW_LEVEL_PROGRAMMING_ITMO_2020_BMP_H
```

bmp.c

```
//
// Created by Sergey Fedorov on 11/6/20.
//

#include "bmp.h"
#include "image.h"

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <float.h>
#include <math.h>

struct bmp_pixel {
    uint8_t b, g, r;
};

enum bmp_read_result bmp_image_read(struct bmp_image** bmp_image, FILE * file) {
    struct bmp_image* image = malloc(sizeof(struct bmp_image));
    int32_t row, rowOffset;

    size_t read_count = fread(&(image->header), sizeof(struct bmp_header), 1, file);

    if (read_count < 1) {
        free(image);
        return READ_INVALID_HEADER;
    } else if (!(image->header.bfType[0] == 'B' && image->header.bfType[1] == 'M')) {
        free(image);
        return READ_INVALID_SIGNATURE;
    } else if (
        (image->header.biSizeImage != 0 && (image->header.bfSize != image->header.bfOffBits +
image->header.biSizeImage))
        || (image->header.biCompression != 0)
        || (image->header.biPlanes != 1)) {
        free(image);
        return READ_INVALID_BMP_FORMAT;
    } else if (image->header.biBitCount != 24) {
        free(image);
        return READ_INVALID_BAD_PIXEL;
    }

    image->pixels = malloc(sizeof(struct bmp_pixel) * image->header.biWidth * image-
>header.biHeight);

    rowOffset = image->header.biWidth % 4;
    fseek(file, image->header.bfOffBits, SEEK_SET);
    for (row = image->header.biHeight - 1; row >= 0; --row) {
        read_count = fread(
            image->pixels + row * image->header.biWidth,
            sizeof(struct bmp_pixel),
            image->header.biWidth, file
        );

        if (read_count < image->header.biWidth) {
            free_bmp_image(image);
            return READ_INVALID_BITS;
        }

        if (fseek(file, rowOffset, SEEK_CUR) != 0) {
            free_bmp_image(image);
```

```c
            return READ_INVALID_BITS;
        }
    }

    *bmp_image = image;
    return READ_OK;
}

void free_bmp_image(struct bmp_image* image) {
    if (image != NULL) {
        free(image->pixels);
        free(image);
    }
}

void bmp_to_image(struct bmp_image* bmp_image, struct image** new_image_p, int free_bmp) {
    struct image* new_image = malloc(sizeof(struct image));

    new_image->height = bmp_image->header.biHeight;
    new_image->width = bmp_image->header.biWidth;
    new_image->data = malloc(sizeof(struct pixel) * new_image->height * new_image->width);

    int32_t row, pos, index;

    for(row = 0; row < new_image->height; row++) {
        for(pos = 0; pos < new_image->width; pos++) {
            index = row * new_image->width + pos;
            struct pixel converted_pixel = {
                bmp_image->pixels[index].b,
                bmp_image->pixels[index].g,
                bmp_image->pixels[index].r
            };

            new_image->data[index] = converted_pixel;
        }
    }

    *new_image_p = new_image;

    if (free_bmp) {
        free_bmp_image(bmp_image);
    }
}

void image_to_bmp(struct image* image, struct bmp_image** new_bmp_image_p, int free_prev_image) {
    struct bmp_image* new_image = malloc(sizeof(struct bmp_image));

    new_image->header.biHeight = image->height;
    new_image->header.biWidth = image->width;

    new_image->header.bfType[0] = 'B';
    new_image->header.bfType[1] = 'M';
    new_image->header.bfOffBits = sizeof(struct bmp_header);

    new_image->header.biSize = 40;
    new_image->header.biPlanes = 1;
    new_image->header.biBitCount = 24;
    new_image->header.biCompression = 0;

    new_image->header.biSizeImage = new_image->header.biHeight *
            (new_image->header.biWidth * sizeof(struct bmp_pixel) + new_image->header.biWidth %
4);
    new_image->header.bfSize = new_image->header.bfOffBits + new_image->header.biSizeImage;

    new_image->pixels = malloc(sizeof(struct bmp_pixel) * new_image->header.biWidth * new_image-
>header.biHeight);

    int32_t row, pos, index;
    for(row = 0; row < new_image->header.biHeight; row++) {
        for(pos = 0; pos < new_image->header.biWidth; pos++) {
            index = row * new_image->header.biWidth + pos;

            struct bmp_pixel converted_pixel = {
                image->data[index].b,
                image->data[index].g,
                image->data[index].r
            };

            new_image->pixels[index] = converted_pixel;
        }
    }

    *new_bmp_image_p = new_image;

    if (free_prev_image) {
        free_image(image);
    }

}


enum bmp_write_result bmp_image_write(const struct bmp_image* image, FILE* file) {
    static uint8_t offsetBuffer[] = { 0, 0, 0 };
    int32_t row, rowOffset;

    if (fwrite(&(image->header), sizeof(struct bmp_header), 1, file) == 0) {
        return WRITE_BAD_HEADER;
    }

    rowOffset = image->header.biWidth % 4;
    for (row = image->header.biHeight - 1; row >= 0; --row) {
        if (fwrite(image->pixels + row * image->header.biWidth,
                   sizeof(struct bmp_pixel), image->header.biWidth, file) < image-
>header.biWidth) {
            return WRITE_BAD_BODY;
        }

        if (fwrite(offsetBuffer, 1, rowOffset, file) < rowOffset) {
            return WRITE_BAD_OFFSET;
        }
    }

    return WRITE_OK;
}
```

Вывод:

На самом деле не особо много мыслей по поводу данной лабораторной работе, но вот следующая ремарка найдется:

Понравился способ того как мы читаем bmp картинку и то что по факту в бинарном файле у нас записана наша же структура указанная в коде, что, до этого момента, я не видел ни в каком другом языке.

Еще во время выполнения работы, приходилось часто дебажить код из-за неявных кастов типов, что достаточно сильно отражалось на финальном качестве картинки.