МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Программирование»

Тема: Обработка BMP изображения

Студент гр. 3343 _____ Старков С.А

Преподаватель _____ Государкин Я.С.

Санкт-Петербург

2024

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент: Старков Савва

Группа: 3343

Тема: Обработка BMP изображения

Условие (вариант 3.2):

Программа должна иметь следующий функционал по обработке изображений:

1. Инверсия цвета на всём изображении. Флаг для выполнения данной операции: `--inverse`

2. Установить компоненту цвета, как сумму двух других. Флаг для выполнения данной операции: `--component_sum`.

Функционал определяется:

Какую компоненту требуется изменить. Флаг `--component_name`. Возможные значения `red`, `green` и `blue`.

Дата выдачи задания: 18.03.2024

Дата сдачи реферата: 22.05.2024

Дата защиты реферата: 22.05.2024

АННОТАЦИЯ

В ходе курсовой работы реализована программа, осуществляющая обработку изображений в формате BMP. Для взаимодействия с программой предусмотрен интерфейс командной строки (CLI). Программа реализует следующие функции:

1. Инверсия цвета: изменяет цвет каждого пикселя на его инверсный.

   Флаг: `--inverse`

2. Установка компоненты цвета как суммы двух других: изменяет значение одной из цветовых компонент на сумму двух других.

   Флаг: `--component_sum`

ВВЕДЕНИЕ

Цель работы:

Создание интерактивного консольного приложения для манипуляций с изображениями в формате BMP. Приложение предоставит пользователю следующие возможности:

- Загрузка и сохранение изображений в формате BMP.
- Редактирование изображений.
- Визуализация результатов обработки изображений.

ОПИСАНИЕ СТРУКТУРЫ ПРОГРАММЫ

Программа была реализована на языке C++. Код программы разбит на несколько файлов:

- *main.cpp* – содержит функцию main, в которой вызывается функция для обработки аргументов командной строки.
- *inverse.cpp* – содержит функции inverse и component_sum.
- *write.cpp* – содержит функции для чтения и записи изображения.
- *functions.h* – заголовочный файл, который содержит объявления некоторых функций для того, чтобы они могли вызываться в других .cpp файлах проекта.
- *bmp.h* – заголовочный файл, который содержит определения структур BMPFileHeader, BMPImage, OptionFlags, а также в этом файле содержатся директивы include для подключения библиотек.
- *helps*.h – заголовочный файл содержит всю выводимую информацию с объявлениями ошибок и string для вывода на экран.
- write.h – заголовочный файл с объявлениями функций и чтения и записи из write.h

ТЕСТИРОВАНИЕ

Рисунок 1 — исходное изображение

1. Тестирование инверс: --inverse --input ./INPUT.bmp --output ./OUTPUT.bmp

Рисунок 2 — работа функции inverse

2. Тестирование компоненты цвета : *--component_sum --component_name red --input ./INPUT.bmp --output ./OUTPUT.bmp*

Рисунок 3 — работа функции color_replace

ЗАКЛЮЧЕНИЕ

В ходе данного курсового проекта было создано приложение на языке программирования C++ для манипуляций с изображениями в формате BMP. Предоставляемый функционал программы может быть выбран пользователем через командную строку. Сборка приложения осуществляется при помощи инструмента make. После сборки приложение запускается из командной строки, где пользователь может выбрать одну или несколько из поддерживаемых функций для обработки изображения.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp

```
#include <queue>
#include <functional>
#include "../include/functions.h"
#include "../include/write.h"
#include <getopt.h>
#include "../include/helps.h"

using namespace std;



int main(int argc ,char *argv[] ){
deque <function<void()>> queuefunction;

OptionFlags flags;

int opt;
BMPImage image{};

const struct option opts[] =
{
   {"help",no_argument,NULL,'h'},

   {"shift",no_argument,NULL,269},

   {"step",required_argument,NULL,283},
```

```
    {"axis",required_argument,NULL,287},

    {"info",no_argument,NULL,'i'},

    {"inverse_image", no_argument, NULL, 258},

    {"input",required_argument,NULL,257},

    {"output",required_argument,NULL,'o'},

    {"component_name",required_argument,NULL,256},

    {"component_max", no_argument, NULL, 259},

    {0, 0, 0, 0}
};

   while((opt = getopt_long(argc,argv,"ho:i:c:",opts,NULL)) != -1){


      switch (opt)
      {
         case 287:{
            flags.axisFlag = true;
            flags.axisfield = optarg;
            break;
         }
         case 283:{
            flags.stepValue = stoi(optarg);
            break;
```

```
}
case 269:{
    flags.shiftFlag = true;
    break;
}
case 258:{
    flags.inverseFlag = true;
    break;
}
case 259:{
    flags.component_sumFlag = true;
    break;
}
case 'h':{
    flags.helpFlag = true;
    break;
}
case 256:{
    flags.color = optarg;
    break;
}
case 'o':{
    flags.filenameoutput = optarg;
    break;
}
case 257:{
    flags.filenameinput = optarg;
    break;
}
case 'i':{
```

```
            flags.infoFlag = true;
            break;
        }
        default:{
            std::cerr<<MESSAGE_UnknownFlag << std::endl;
            break;
        }
    }
}

if (flags.helpFlag == true)
{
    std::cout<<MESSAGE_HelpInfo<<std::endl;
    exit(0);
    /* code */
}

if(flags.filenameinput.empty()){
    flags.filenameinput = argv[argc-1];
}

if(flags.filenameoutput.empty()){
    flags.filenameoutput = "out.bmp";
}

if (flags.filenameinput == flags.filenameoutput){
    std::cerr<<"The names of the input and output files are the same"<<std::endl;
    exit(ERR_INPUTQUALS);
}
```

```
        auto read = [&flags, &image](){image = readBMP(flags.filenameinput.c_str());};
        queuefunction.emplace_front(read);



        if(flags.inverseFlag){
            auto inverse = [&flags,&image] (){
                inverse_all(image);
            };
            queuefunction.emplace_back(inverse);
        }
        if (flags.component_sumFlag != false){
            if(!flags.color.compare(RED) && !flags.color.compare(GREEN) && !flags.color.compare(BLUE)){
                std::cerr<< MESSAGE_WrongColor<<std::endl;
                exit(ERR_WRONGCOLOR);
            }
            auto component_sumfunc = [&image,&flags](){
                component_max(image,flags.color);
            };
            queuefunction.emplace_back(component_sumfunc);
        }



        if(flags.infoFlag == true){
            auto infofunc = [&image,&flags]{
                printInfo(image,flags.filenameinput);
```

```
        };
        queuefunction.emplace_back(infofunc);
    };

    auto write = [&image,&flags]{
        writeBMP(flags.filenameoutput,image);
    };
    queuefunction.emplace_back(write);









    while (!queuefunction.empty())
    {
        queuefunction.front()();
        queuefunction.pop_front();
    }

    if(!image.data.empty()){
        image.data.clear();
    }



}
```

inverse.cpp

```
#include "../include/functions.h"
```

```
#include "../include/bmp.h"
#include <string>
#include "../include/helps.h"

using namespace std;

void inverse_all(BMPImage &image_data) {

    for (uint32_t i = 0; i < image_size; i++) {
        image_data.data.at(i) = ~image_data.data[i];
        /* code */
    }

}


void component_max(BMPImage &image_data, std::string color) {

    if (!(color == RED) && !(color== GREEN) && !(color==BLUE)){
        std::cerr<<MESSAGE_WrongColor<<endl;
        exit(ERR_WRONGCOLOR);
        return;
    }
    //bgr
    for (size_t i = 0; i < image_size-3; i+=3)
    {
        uint8_t max = 255;
        if (color == BLUE ){
            int sum = image_data.data[i+1] + image_data.data[i+2];
            image_data.data[i] = (sum > max) ? max : sum;
```

```
            }
            if (color == GREEN){
                int sum = image_data.data[i] + image_data.data[i+2];
                image_data.data[i+1] = (sum > max) ? max : sum;

            }
            if (color == RED){
                int sum = image_data.data[i] + image_data.data[i+1];
                image_data.data[i+2] = (sum > max) ? max: sum;

            }
            /* code */
        }
    }

    Pixel get_pixel(BMPImage &info, int y, int x) {
        int bytesPerPixel = info.file_header.bit_count / 8;
        int bytesPerRow = (bytesPerPixel * info.file_header.width + 3) & ~3;
        int index = ((info.file_header.height - 1 - y) * bytesPerRow) + (x * bytesPerPixel);

        return Pixel{&(info.data.at(index + 2)), &(info.data.at(index + 1)), &(info.data.at(index))};
    }

    void set_pixel(BMPImage &info, int y, int x, Pixel pixel) {
        if (y >= info.file_header.height || y < 0 || x < 0 || x >= info.file_header.width) {
```

```
            return;
        }

        int bytesPerPixel = info.file_header.bit_count / 8;
        int bytesPerRow = (bytesPerPixel * info.file_header.width + 3) & ~3;
        int index = ((info.file_header.height - 1 - y) * bytesPerRow) + (x * bytesPerPixel);
        info.data.at(index + 2) = *pixel.red;
        info.data.at(index + 1) = *pixel.green;
        info.data.at(index) = *pixel.blue;
    }

    void set_pixel(BMPImage &info, int y, int x, Pixel pixel, std::vector<uint8_t> &new_data) {
        if (y >= info.file_header.height || y < 0 || x < 0 || x >= info.file_header.width) {
            return;
        }

        int bytesPerPixel = info.file_header.bit_count / 8;
        int bytesPerRow = (bytesPerPixel * info.file_header.width + 3) & ~3;
        int index = ((info.file_header.height - 1 - y) * bytesPerRow) + (x * bytesPerPixel);
        new_data.at(index + 2) = *pixel.red;
        new_data.at(index + 1) = *pixel.green;
        new_data.at(index) = *pixel.blue;
    }


    void shift_x(BMPImage &info, int shiftvalue) {
```

```
    vector<uint8_t> data_new(info.data);
    for (int y = 0; y < info.file_header.height; ++y) {
      for (int x = 0; x < info.file_header.width; ++x) {
        if (x >= shiftvalue) {

          Pixel pixel_new = get_pixel(info, y, x - shiftvalue);
          set_pixel(info, y, x, pixel_new, data_new);

        } else {

          Pixel pixel_new = get_pixel(info, y, info.file_header.width - shiftvalue
+ x);
          set_pixel(info, y, x, pixel_new, data_new);
        }


      }
    }
    info.data = data_new;

  }

  void shift_y(BMPImage &info, int shiftvalue) {
    vector<uint8_t> data_new(info.data);
    for (int y = 0; y < info.file_header.height; ++y) {
      for (int x = 0; x < info.file_header.width; ++x) {
        if (y>=shiftvalue) {

          Pixel pixel_new = get_pixel(info, y - shiftvalue, x);
          set_pixel(info, y, x, pixel_new, data_new);
```

```
            } else {
                Pixel pixel_new = get_pixel(info, info.file_header.height-shiftvalue+y,
x);
                set_pixel(info, y, x, pixel_new, data_new);

            }


        }
      }
      info.data = data_new;
    }

    void shift_func(BMPImage &info, int shiftvalue, std::string axisfield) {
      if (axisfield == "x") {
        vector<uint8_t> data_new(info.data);
        for (int y = 0; y < info.file_header.height; ++y) {
          for (int x = 0; x < info.file_header.width; ++x) {
            if (x >= shiftvalue) {

              Pixel pixel_new = get_pixel(info, y, x - shiftvalue);
              set_pixel(info, y, x, pixel_new, data_new);

            } else {

              Pixel pixel_new = get_pixel(info, y, info.file_header.width -
shiftvalue + x);
              set_pixel(info, y, x, pixel_new, data_new);
            }
```

```
            }
        }
        info.data = data_new;

    }

    if (axisfield == "y") {
        vector<uint8_t> data_new(info.data);
        for (int y = 0; y < info.file_header.height; ++y) {
            for (int x = 0; x < info.file_header.width; ++x) {
                if (y>=shiftvalue) {

                    Pixel pixel_new = get_pixel(info, y - shiftvalue, x);
                    set_pixel(info, y, x, pixel_new, data_new);

                } else {
                    Pixel pixel_new = get_pixel(info, info.file_header.height-
shiftvalue+y, x);
                    set_pixel(info, y, x, pixel_new, data_new);

                }


            }
        }
        info.data = data_new;

    }
```

```
    if (axisfield == "xy") {
        shift_x(info,shiftvalue);
        shift_y(info,shiftvalue);


    }


}

bool validateShift(OptionFlags &flags) {
    if ((flags.axisFlag || flags.shiftFlag || flags.stepFlag) == 1) {
        return true;
    } else {
        return false;
        std::cerr << "shift goes wrong!" << std::endl;
        exit(40);
    }

}




//void ImageProcessor::gray(Picture &picture, Coordinate leftUp, Coordinate rightDown){
//    std::vector<uint8_t> data =  picture.data;
//
```

```
//    for (int y = leftUp.y; y <= rightDown.y; ++y) {
//        for (int x = leftUp.x; x <= rightDown.x; ++x) {
//            Color pixel = picture.getPixel(x, y);
//            uint8_t grayValue = std::round(0.299 * pixel.red + 0.587 * pixel.green
+ 0.114 * pixel.blue);
//            picture.setPixel(x, y, Color(grayValue, grayValue, grayValue));
//        }
//    }
//}
//
write.cpp
#ifndef WRITE_H
#define WRITE_H

#include <iostream>
#include <fstream>
#include <vector>
#include "../include/write.h"

uint32_t image_size;

uint32_t calculateImageSize(const BMPFileHeader &bmpInfoHeader) {
    // Размер изображения можно вычислить на основе ширины, высоты и
бит на пиксель
    // В BMP-файлах данные изображения выровнены по 4 байтам, поэтому
необходимо учесть это при вычислении
    const uint32_t bytesPerPixel = bmpInfoHeader.bit_count / 8;

    const uint32_t padding = (4 - (bmpInfoHeader.width * bytesPerPixel) % 4) %
4;
```

```
        const uint32_t rowSize = (bmpInfoHeader.width * bytesPerPixel) + padding;

        // Размер изображения в байтах
        uint32_t imageSize = rowSize * abs(bmpInfoHeader.height);

        // Если в заголовке уже указан размер изображения, используем его
        if (bmpInfoHeader.image_size != 0) {
            imageSize = bmpInfoHeader.image_size;
        }

        return imageSize;
    }

    void writeBMP(std::string &filename, BMPImage &image) {
        std::ofstream file(filename, std::ios::binary);
        if (!file) {
            std::cerr << "Error opening file for writing: " << filename << std::endl;
            exit(41);
        }

        // Запись заголовка файла
        file.write(reinterpret_cast<const char *>(&image.file_header), sizeof(image.file_header));

        file.seekp(image.file_header.offset, std::ios::beg);
        // Запись данных изображения
        const uint32_t bytes_per_pixel = image.file_header.bit_count / 8;
        const uint32_t bytes_per_row = (((image.file_header.width * bytes_per_pixel) + 3) / 4) * 4;
        for (int i = 0; i < image.file_header.height; ++i) {
```

```
            file.write(reinterpret_cast<const char *>(image.data.data() + i * bytes_per_row), bytes_per_row);
            /* code */
        }


        file.close();
    }

    BMPImage readBMP(const char *filename) {
        BMPFileHeader header;
        std::ifstream file(filename, std::ios::binary);
        if (!file) {
            std::cerr << "Error opening file: " << filename << std::endl;
            exit(46);
        }
        // Чтение заголовка файла
        file.read(reinterpret_cast<char *>(&header), sizeof(header));
        if (header.signature != 0x4D42) { // 'BM'
            std::cerr << "Not a BMP file." << std::endl;
            exit(46);
        }


        // const uint32_t bytes_per_pixel = header.bit_count / 8;
        // const uint32_t bytes_per_row = (((header.width * bytes_per_pixel) + 3) / 4) * 4;

        // image_size = header.height * bytes_per_row;
```

```
    image_size = calculateImageSize(header);

    std::vector<uint8_t> dataNew;

    dataNew.resize(image_size);
    if (dataNew.empty()) {
        std::cerr << "Error allocating memory for image data" << std::endl;
        exit(45);
    }


    // Чтение данных изображения
    file.seekg(header.offset, std::ios::beg);
    file.read(reinterpret_cast<char *>(dataNew.data()), image_size);


    BMPImage image{header, dataNew};


    file.close();

    return image;

}


void printInfo(BMPImage &image, std::string filename) {
    std::cout << "File name: " << filename << std::endl;
    std::cout << "File type: " << image.file_header.signature << std::endl;
    std::cout << "Height: " << image.file_header.height << std::endl;
```

```
    std::cout << "Width: " << image.file_header.width << std::endl;
    std::cout << "Size: " << image.file_header.file_size << " bytes" << std::endl;
    std::cout << "Color depth: " << image.file_header.bit_count << std::endl;
    exit(0);

}

#endif
```