Федеральное государственное бюджетное образовательное учреждение высшего образования

«Сибирский государственный университет телекоммуникаций и информатики»

(СибГУТИ)

Кафедра прикладной математики и кибернетики

Расчетно-графическое задание  
по курсу программирование графической информации

Выполнил:  
 студент группы ИП-014

Обухов А.И.

Работу проверил:  
 доцент каф. ПМиК  
 Перцев И.В.

Новосибирск 2024 г.

**Задание:**

Написать программу-конвертор количества цветов в изображении.

Предлагаемый алгоритм. Для уменьшения количества цветов выбираются наиболее часто встречаемые цвета в исходном изображении. Причем эти цвета не должны быть слишком похожими друг на друга. Для сравнения цветов вычисляются разности между RGB составляющими.

*Delta=(R1-R2)2 + (G1-G2)2+ (B1-B2)2*

После формирования новой палитры цвета в заменяются на наиболее похожие из записанных в палитру.

Можно использовать любой другой алгоритм преобразования цветов (например медианного сечения) главное требование – алгоритм должен быть реализован самостоятельно.

Программа должны выводить изображение на экран до и после конвертирования.

Вариант 1: Преобразовать True Color BMP файл в 16-цветный BMP файл.

**Листинг программы:**

#include <stdbool.h>

#include <stdint.h>

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <string.h>

#pragma pack(1)

typedef struct {

uint16\_t signature;

uint32\_t filesize;

uint32\_t reserved;

uint32\_t offset;

uint32\_t header\_size;

uint32\_t width;

uint32\_t height;

uint16\_t planes;

uint16\_t bpp;

uint32\_t compression;

uint32\_t image\_size;

uint32\_t x\_pixels\_per\_m;

uint32\_t y\_pixels\_per\_m;

uint32\_t colors\_used;

uint32\_t colors\_important;

} Head;

#pragma(pop)

typedef struct {

Head head;

uint8\_t\* rastr;

uint8\_t\* palette;

} Image;

void read\_head(Head\* head, FILE\* file) { fread(head, sizeof(Head), 1, file); }

Image read\_image(FILE\* file)

{

Image image = { 0 };

read\_head(&image.head, file);

fseek(file, sizeof(image.head), SEEK\_SET);

if (image.head.bpp <= 8) {

int colors\_number = 1 << image.head.bpp;

image.palette = (uint8\_t\*)malloc(colors\_number \* 4 \* sizeof(uint8\_t));

fread(image.palette, sizeof(uint8\_t), (colors\_number \* 4), file);

}

image.rastr = (uint8\_t\*)malloc(image.head.filesize);

int bytes\_per\_pixel = image.head.bpp / 8;

if (bytes\_per\_pixel == 0) {

bytes\_per\_pixel = 1;

}

int row\_size = image.head.width \* bytes\_per\_pixel;

uint32\_t padding = (4 - (row\_size) % 4) % 4;

for (int i = 0; i < image.head.height; i++) {

fread(&image.rastr[i \* row\_size], sizeof(uint8\_t), row\_size, file);

fseek(file, padding, SEEK\_CUR);

}

return image;

}

void print\_head(Head head)

{

printf("signature: %s\n", (char\*)&head.signature);

printf("filesize: %d\n", head.filesize);

printf("reserved: %d\n", head.reserved);

printf("offset: %d\n", head.offset);

printf("header\_size: %d\n", head.header\_size);

printf("width: %d\n", head.width);

printf("height: %d\n", head.height);

printf("planes: %d\n", head.planes);

printf("bpp: %d\n", head.bpp);

printf("compression: %d\n", head.compression);

printf("image\_size: %d\n", head.image\_size);

printf("x\_pixels\_per\_m: %d\n", head.x\_pixels\_per\_m);

printf("y\_pixels\_per\_m: %d\n", head.y\_pixels\_per\_m);

printf("colors\_used: %d\n", head.colors\_used);

printf("colors\_important: %d\n", head.colors\_important);

}

typedef struct {

uint8\_t red;

uint8\_t green;

uint8\_t blue;

uint8\_t count;

} Pixel;

typedef struct {

int start, end;

Pixel min, max;

} Box;

int compare\_red(const void\* a, const void\* b) {

Pixel\* pixel\_a = (Pixel\*)a;

Pixel\* pixel\_b = (Pixel\*)b;

return pixel\_a->red - pixel\_b->red;

}

int compare\_green(const void\* a, const void\* b) {

Pixel\* pixel\_a = (Pixel\*)a;

Pixel\* pixel\_b = (Pixel\*)b;

return pixel\_a->green - pixel\_b->green;

}

int compare\_blue(const void\* a, const void\* b) {

Pixel\* pixel\_a = (Pixel\*)a;

Pixel\* pixel\_b = (Pixel\*)b;

return pixel\_a->blue - pixel\_b->blue;

}

void find\_min\_max(Pixel\* img, int start, int end, Pixel\* min, Pixel\* max) {

\*min = img[start];

\*max = img[start];

for (int i = start + 1; i <= end; i++) {

if (img[i].red < min->red) min->red = img[i].red;

if (img[i].green < min->green) min->green = img[i].green;

if (img[i].blue < min->blue) min->blue = img[i].blue;

if (img[i].red > max->red) max->red = img[i].red;

if (img[i].green > max->green) max->green = img[i].green;

if (img[i].blue > max->blue) max->blue = img[i].blue;

}

}

int longest\_side(Pixel\* min, Pixel\* max) {

int red\_range = max->red - min->red;

int green\_range = max->green - min->green;

int blue\_range = max->blue - min->blue;

if (red\_range >= green\_range && red\_range >= blue\_range) return 0;

if (green\_range >= red\_range && green\_range >= blue\_range) return 1;

return 2;

}

void median\_cut(Pixel\* img, int start, int end) {

if (end <= start) {

return;

}

Pixel min, max;

find\_min\_max(img, start, end, &min, &max);

int longest = longest\_side(&min, &max);

if (longest == 0) qsort(img + start, end - start + 1, sizeof(Pixel), compare\_red);

else if (longest == 1) qsort(img + start, end - start + 1, sizeof(Pixel), compare\_green);

else qsort(img + start, end - start + 1, sizeof(Pixel), compare\_blue);

int median = (start + end) / 2;

median\_cut(img, start, median);

median\_cut(img, median + 1, end);

}

double calculate\_dissimilarity(Pixel color1, Pixel color2)

{

double diff\_r = ((int)color1.red) - ((int)color2.red);

double diff\_g = ((int)color1.green) - ((int)color2.green);

double diff\_b = ((int)color1.blue) - ((int)color2.blue);

return sqrt(diff\_r \* diff\_r + diff\_g \* diff\_g + diff\_b \* diff\_b);

}

// Convert true color BMP image to 4-bit color depth

Image convert\_true\_color\_bmp\_to\_4\_bit(Image image)

{

Pixel \*colors = (Pixel \*)malloc(image.head.height \* image.head.width \* sizeof(Pixel));

if (colors == NULL) {

fprintf(stderr, "Memory allocation failed.\n");

exit(1);

}

// Copy true color pixels to colors array

for (int y = 0; y < image.head.height; y++) {

for (int x = 0; x < image.head.width; x++) {

uint64\_t index = y \* image.head.width + x;

colors[index].red = image.rastr[index \* 3 + 2];

colors[index].green = image.rastr[index \* 3 + 1];

colors[index].blue = image.rastr[index \* 3];

}

}

median\_cut(colors, 0, image.head.height \* image.head.width - 1);

uint64\_t step = (image.head.height \* image.head.width) / 16;

image.palette = malloc(16 \* 4 \*sizeof(uint8\_t));

uint8\_t new\_pallette\_iterator = 0;

for (int i = 0; i < image.head.width \* image.head.height; i += step) {

// printf("color %d: %d %d %d\n", i, colors[i].red, colors[i].green, colors[i].blue);

uint8\_t offset = new\_pallette\_iterator \* 4;

image.palette[offset + 2] = colors[i].red;

image.palette[offset + 1] = colors[i].green;

image.palette[offset] = colors[i].blue;

new\_pallette\_iterator++;

}

uint8\_t \*new\_rastr = calloc(image.head.height \* (image.head.width + 1) / 2, sizeof(uint8\_t));

// Update image raster with quantized colors

for (int y = 0; y < image.head.height; y++) {

for (int x = 0; x < image.head.width; x++) {

uint64\_t old\_offset = y \* image.head.width \* 3 + x \* 3;

uint64\_t new\_offset = y \* ((image.head.width + 1) / 2) + x / 2;

Pixel rastr\_color = {

image.rastr[old\_offset + 2],

image.rastr[old\_offset + 1],

image.rastr[old\_offset]

};

uint8\_t nearest\_color\_pallette\_index = 0;

double min\_dissimilirity = \_\_DBL\_MAX\_\_;

for (int i = 0; i < 16; i++) {

Pixel palette\_color = {

image.palette[i \* 4 + 2],

image.palette[i \* 4 + 1],

image.palette[i \* 4]

};

double dissimilarity = calculate\_dissimilarity(rastr\_color, palette\_color);

if (dissimilarity < min\_dissimilirity) {

min\_dissimilirity = dissimilarity;

nearest\_color\_pallette\_index = i;

}

}

if (!(x & 1u)) {

new\_rastr[new\_offset] = (new\_rastr[new\_offset] & 0x0F) | (nearest\_color\_pallette\_index << 4);

} else {

new\_rastr[new\_offset] = (new\_rastr[new\_offset] & 0xF0) | (nearest\_color\_pallette\_index & 0x0F);

}

}

}

free(image.rastr);

free(colors);

image.rastr = new\_rastr;

image.head.bpp = 4;

image.head.planes = 1;

image.head.colors\_used = 16;

image.head.colors\_important = 16;

image.head.offset = sizeof(Head) + (1 << image.head.bpp) \* 4;

image.head.filesize = image.head.height \* (image.head.width + 1) / 2 + image.head.offset;

return image;

}

// Write image to file

void write\_image(FILE\* file, Image image)

{

fwrite(&image.head, sizeof(Head), 1, file);

if (NULL != image.palette) {

fwrite(image.palette, sizeof(uint8\_t), (1 << image.head.bpp) \* 4, file); // Write palette

}

uint32\_t row\_size = ((image.head.width + 1) / 2);

uint32\_t padding = (4 - (row\_size) % 4) % 4;

for (int i = 0; i < image.head.height; i++) {

fwrite(&image.rastr[i \* row\_size], sizeof(uint8\_t), row\_size, file);

fwrite(&padding, sizeof(uint8\_t), padding, file);

}

}

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

{

if (argc <= 2) {

fprintf(stderr, "Usage: %s <input> <output>\n", argv[0]);

return -1;

}

char\* filename = argv[1];

FILE\* file = fopen(filename, "rb");

if (NULL == file) {

perror(filename);

return -1;

}

Image image = read\_image(file);

print\_head(image.head);

if (image.head.bpp != 24) {

fprintf(stderr, "Unsupported format");

return -1;

}

fclose(file);

Image converted\_image = convert\_true\_color\_bmp\_to\_4\_bit(image);

write\_image(fopen(argv[2], "wb"), converted\_image);

return 0;

}

**Результат работы:**

****

Рис. 1 Исходное изображение

****

Рис. 2 Результирующие изображение