

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_palette_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_palette_iterator * 4;
    image.palette[offset + 2] = colors[i].red;
    image.palette[offset + 1] = colors[i].green;
    image.palette[offset] = colors[i].blue;

    new_palette_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_palette_index = 0;
        double min_dissimilarity = __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_dissimilarity) {
        min_dissimilarity = dissimilarity;
        nearest_color_palette_index = i;
    }
}

if (!(x & 1u)) {
    new_rastr[new_offset] = (new_rastr[new_offset] & 0x0F) | (nearest_color_palette_index
<< 4);
} else {
    new_rastr[new_offset] = (new_rastr[new_offset] & 0xF0) | (nearest_color_palette_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