МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Лабораторная работа № 3

По дисциплине «Компьютерная геометрия и графика»

Изучение алгоритмов псевдотонирования изображений

Выполнил студент группы №M3101
Семенов Георгий Витальевич

Преподаватель:
Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ

2020

Цель работы: изучить алгоритмы и реализовать программу, применяющий алгоритм дизеринга к изображению в формате PGM (P5) с учетом гамма-коррекции.

Описание: Программа должна быть написана на C/C++ и не использовать внешние библиотеки.

Аргументы передаются через командную строку:

program.exe <имя_входного_файла> <имя_выходного_файла> <градиент> <дизеринг> <битность> <гамма>

где

- <имя_входного_файла>, <имя_выходного_файла>: формат файлов: PGM P5; ширина и высота берутся из <имя_входного_файла>;
- <градиент>: 0 - используем входную картинку, 1 - рисуем горизонтальный градиент (0-255) (ширина и высота берутся из <имя_входного_файла>);
- <дизеринг> - алгоритм дизеринга:
    a. 0 – Нет дизеринга;
    b. 1 – Ordered (8x8);
    c. 2 – Random;
    d. 3 – Floyd–Steinberg;
    e. 4 – Jarvis, Judice, Ninke;
    f. 5 - Sierra (Sierra-3);
    g. 6 - Atkinson;
    h. 7 - Halftone (4x4, orthogonal);
- <битность> - битность результата дизеринга (1..8);
- <гамма>: 0 - sRGB гамма, иначе - обычная гамма с указанным значением.

Частичное решение:

- <градиент> = 1;
- <дизеринг> = 0..3;
- <битность> = 1..8;
- <гамма> = 1 (аналогично отсутствию гамма-коррекции)

+ корректно выделяется и освобождается память, закрываются файлы, есть обработка ошибок.

Теоретическая часть

При преобразовании изображения с уменьшением его битности возникает проблема в том, что нарушается цельность восприятия изображения, т.е. цвета преобразуются в новую, более узкую палитру единично, независимо друг от друга.
Данную проблему решают **алгоритмы дизеринга**. Они поддерживают цельность изображения с помощью двух фундаментальных методов: ordered dithering и error diffusion dithering, в которых квантование (сужение палитры) всегда происходит слева-направо, сверху-вниз.
Ordered dithering последовательно применяет к участкам данного изображения квадратную матрицу, позволяющую, вообще говоря, сохранить цельность воспринимаемого цвета. Этот алгоритм считает новое значение пикселя, основываясь исключительно на прежнем его значении.
Error diffusion dithering последовательно учитывает ошибку квантования (т.е. цветовую разность между новым оттенком и прежним) и передаёт её согласно определяемому алгоритмом закону распространения ошибки. Так, алгоритм как бы размывает ошибку.

Экспериментальная часть

Язык программирования: C++.

Изображение описано в классе pnmImage, хранящее в двумерном массиве (vector) указатели на экземпляры класса pnmBWColor, в котором реализованы переводы sRGB и gamma.

В методах horizontalGradient, dither, orderedDitherer и threshold реализованы отрисовка горизонтального градиента, общий дизеринг, ordered dithering и получение цвета для новой палитры.

Вывод

Алгоритмы дизеринга по времени зависят от размеров изображения и глубины цвета исходной палитры.

Листинг

https://github.com/MrGeorgeous/ComputerGeometryAndGraphics/blob/master/README.md

Файл: Main.cpp

```
#define _CRT_SECURE_NO_WARNINGS
```

```cpp
#include<stdio.h>
#include <iostream>
#include <fstream>
#include <istream>
#include <iomanip>

#include<string>
#include<vector>
#include<set>
#include<queue>


#include <functional>
#include <algorithm>
#include <utility>


using namespace std;

const int UCHAR_SIZE = sizeof(unsigned char);

class pnmBWColor {

public:

	double color = 0.0;

	pnmBWColor() {

	}

	pnmBWColor(const pnmBWColor& c) {
		color = c.color;
	}

	pnmBWColor(double black) {
		color = black;
	}

	~pnmBWColor() {

	}

	void setColor(double r) {
		color = r;
	}

	void inverseColor(unsigned char depth = 255) {
		color = depth - color;
	}


	void gamma(double gamma = 1, size_t depth = 255, bool reverse = false) {
		if (!reverse) {
			gammaDecode(color, gamma, depth);
		}
		else {
			gammaEncode(color, gamma, depth);
		}
	}

	void srgb(size_t depth = 255, bool reverse = false) {

		if (!reverse) {
```

```
                srgbDecode(color, depth);
        }
        else {
                srgbEncode(color, depth);
        }

    }

    static void gammaEncode(double& a, double gamma = 1.0, size_t depth = 255) {
        a = double(depth) * pow(double(a) / depth, 1.0 / gamma);
    }

    static void gammaDecode(double& a, double gamma = 1.0, size_t depth = 255) {
        a = double(depth) * pow(double(a) / depth, gamma);
    }

    static void srgbEncode(double& a, size_t depth = 255) {
        double r = double(a) / double(depth);

        if (r <= 0.0031308) {
                a = double(depth) * (12.92 * r);
        }
        else {
                a = double(depth) * ((211.0 * pow(r, 5.0 / 12.0) - 11.0) /
200.0);
        }
    }

    static void srgbDecode(double& a, size_t depth = 255) {
        double r = double(a) / depth;

        if (r <= 0.04045) {
                a = depth * (25 * r / 323);
        }
        else {
                a = depth * pow((200.0 * r + 11.0) / 211, 12.0 / 5.0);
        }
    }


};



typedef vector<vector<double>> doubleMatrix;
typedef vector<vector<pnmBWColor*>> pnmMatrix;
typedef vector<char> chars;



class pnmBWImage {

public:

    int width = 0; // y
    int height = 0; // x
    int depth = 255; // d
    int gamma = -1; // gamma

    pnmMatrix m;
    chars errorEncounter;
```

```
pnmBWImage(size_t w, size_t h, pnmBWColor color = pnmBWColor(255)) {

        m = pnmMatrix(h, vector<pnmBWColor*>(w, nullptr));

        for (int j = 0; j < h; j++) {
                for (int i = 0; i < w; i++) {
                        m[j][i] = new pnmBWColor(color);
                }
        }

        width = w;
        height = h;

}

~pnmBWImage() {

        for (int i = 0; i < height; i++) {
                for (int j = 0; j < width; j++) {
                        delete m[i][j];
                }
        }

}

pnmBWImage(string filename, int g = -1) {

        if (!errorEncounter.empty()) {
                return;
        }

        FILE* file = fopen(filename.c_str(), "rb");
        if (!(file != NULL)) {
                errorEncounter.push_back(1);
                return;
        }

        char p1, p2 = ' ';
        int w = 0, h = 0, d = 0;

        fscanf(file, "%c%c\n%i %i\n%i\n", &p1, &p2, &w, &h, &d);

        width = w;
        height = h;
        depth = d;
        gamma = g;

        if (((w <= 0) || (h <= 0))) {
                errorEncounter.push_back(1);
                return;
        }

        m = pnmMatrix(h, vector<pnmBWColor*>(w, nullptr));

        if (p1 != 'P') {
                errorEncounter.push_back(1);
                return;
        }
        if (!((p2 == '5'))) {
                errorEncounter.push_back(1);
                return;
        }
        if (!(d == 255)) {
```

```
                errorEncounter.push_back(1);
                return;
        }

        unsigned char t;
        for (int j = 0; j < h; j++) {
                for (int i = 0; i < w; i++) {
                        size_t r = fread(&t, UCHAR_SIZE, 1, file);
                        if (r != UCHAR_SIZE) {
                                errorEncounter.push_back(1);
                                return;
                        }
                        m[j][i] = new pnmBWColor(t);
                }
        }

        fclose(file);

        correction(gamma, false);

}

void print(string filename) {

        if (!errorEncounter.empty()) {
                return;
        }

        correction(gamma, true);

        FILE* file = fopen(filename.c_str(), "wb");
        if (!(file != NULL)) {
                errorEncounter.push_back(1);
                return;
        }

        fprintf(file, "P5\n");
        fprintf(file, "%i %i\n%i\n", width, height, depth);

        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {

                        unsigned char t = m[j][i]->color;
                        fwrite(&(t), sizeof(unsigned char), 1, file);

                }
        }

        fclose(file);

}


unsigned char colorToCurrentScheme(unsigned char c) {
        pnmBWColor color(c);
        if (gamma == -1) {
                color.srgb(depth, true);
        }
        else {
                color.gamma(gamma, depth, true);
        }
        return color.color;
}
```

```
protected:
	void correction(double g = -1, bool reverse = false) {

		if (!errorEncounter.empty()) {
			return;
		}

		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				if (g == -1) {
					m[i][j]->srgb(depth, reverse);
				}
				else {
					m[i][j]->gamma(g, depth, reverse);
				}
			}
		}

	}


public:



	int bit = 8;

	void dither(int mode = 0, int b = 8) {

		if (!errorEncounter.empty()) {
			return;
		}

		bit = b;

		if (mode == 0) {
			dither0();
		}

		if (mode == 1) {
			ditherOrdered8x8();
		}

		if (mode == 2) {
			ditherRandom();
		}

		if (mode == 3) {
			ditherFloydSteinberg();
		}

		if (mode == 4) {
			ditherJarvisJudiceNinke();
		}

		if (mode == 5) {
			ditherSierra();
		}

		if (mode == 6) {
			ditherAtkinson();
		}
```

```
        if (mode == 7) {
                ditherHalftone();
        }


}


void horizontalGradient() {
        for (int i = 0; i < width; i++) {
                double c = 255 * double(i) / double(width - 1.0);
                for (int j = 0; j < height; j++) {
                        m[j][i]->setColor(threshold(c));
                }
        }
}

void horizontalGradient8bit() {
        for (int i = 0; i < width; i++) {
                double c = 255 * double(i) / double(width - 1.0);
                for (int j = 0; j < height; j++) {
                        m[j][i]->setColor(round(c));
                }
        }
}

// x и y
void copyColorPlus(int j, int i, double value) {
        if ((0 <= i) && (i < width)) {
                if ((0 <= j) && (j < height)) {
                        double v = double(m[j][i]->color + value);
                        if (v < 0.0) {
                                m[j][i]->setColor(0.0);
                                return;
                        }
                        if (v > 255.0) {
                                m[j][i]->setColor(255.0);
                                return;
                        }
                        m[j][i]->setColor(v);
                }
        }
}


bool getBit(unsigned char byte, int position) {
        return (byte >> position) & 1U;
}

void setBit(unsigned char& byte, int position, bool value) {
        byte ^= (-value ^ byte) & (1UL << position);
}


double threshold(double color) {
        if (color >= 255.0) {
                color = 255;
                return 255;
        }
        if (color <= 0.0) {
                color = 0;
                return 0;
        }
```

```
        unsigned char c = int(color);

        if (bit == 8) {
                return round(color);
        }

        if (bit == 2) {
                setBit(c, 0, getBit(color, 6));
                setBit(c, 1, getBit(color, 7));
                setBit(c, 2, getBit(color, 6));
                setBit(c, 3, getBit(color, 7));
                setBit(c, 4, getBit(color, 6));
                setBit(c, 5, getBit(color, 7));
        }

        if (bit == 3) {
                setBit(c, 0, getBit(color, 6));
                setBit(c, 1, getBit(color, 7));
                setBit(c, 2, getBit(color, 5));
                setBit(c, 3, getBit(color, 6));
                setBit(c, 4, getBit(color, 7));
        }

        if (bit == 4) {
                setBit(c, 0, getBit(color, 4));
                setBit(c, 1, getBit(color, 5));
                setBit(c, 2, getBit(color, 6));
                setBit(c, 3, getBit(color, 7));
        }

        if (bit == 5) {
                setBit(c, 0, getBit(color, 5));
                setBit(c, 1, getBit(color, 6));
                setBit(c, 2, getBit(color, 7));
        }

        if (bit == 6) {
                setBit(c, 0, getBit(color, 6));
                setBit(c, 1, getBit(color, 7));
        }

        if (bit == 7) {
                setBit(c, 0, getBit(color, 7));
        }

        if (bit == 1) {
                if (color <= 127) {
                        return 0;
                }
                else {
                        return 255;
                }
        }

        return c;
}


void dither0() {

        for (int i = 0; i < width; i++) {
                for (int j = 0; j < height; j++) {
                        m[j][i]->setColor(threshold(m[j][i]->color));
```

```
                        //cout << int(m[j][i]->color) << " ";
                    }
                }

        }


        void orderedDitherer(doubleMatrix pattern) {

                const int order = pattern.size();

                for (auto& v : pattern) {
                        for (auto& u : v) {
                                u = 255.0 * ((u) / double(order * order) - 0.5) /
double(bit);
                        }
                }

                for (int i = 0; i < width; i++) {
                        for (int j = 0; j < height; j++) {
                                double mapKey = pattern[i % order][j % order];
                                m[j][i]->setColor(threshold(double(m[j][i]->color) +
double(mapKey)));
                        }
                }

        }

        void ditherOrdered8x8() {

                doubleMatrix pattern = {
                        {0, 48, 12, 60, 3, 51, 15, 63},
                        {32, 16, 44, 28, 35, 19, 47, 31},
                        {8, 56, 4, 52, 11, 59, 7, 55},
                        {40, 24, 36, 20, 43, 27, 39, 23},
                        {2, 50, 14, 62, 1, 49, 13, 61},
                        {34, 18, 46, 30, 33, 17, 45, 29},
                        {10, 58, 6, 54, 9, 57, 5, 53},
                        {42, 26, 38, 22, 41, 25, 37, 21},
                };

                for (auto& v : pattern) {
                        for (auto& u : v) {
                                u += 1;
                        }
                }

                return orderedDitherer(pattern);

        }

        void ditherRandom() {

                for (int i = 0; i < width; i++) {
                        for (int j = 0; j < height; j++) {
                                m[j][i]->setColor(threshold(double(m[j][i]->color) +
double(rand() % 256 - 128)));
                        }
                }

        }

        void ditherFloydSteinberg() {
```

```
        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {


                        double oldPixel = double(m[j][i]->color);
                        double newPixel = threshold(oldPixel);
                        m[j][i]->setColor(newPixel);
                        double quant_error = oldPixel - newPixel;

                        copyColorPlus(j, i + 1, quant_error * 7 / 16);
                        copyColorPlus(j + 1, i, quant_error * 5 / 16);
                        copyColorPlus(j + 1, i + 1, quant_error * 1 / 16);
                        copyColorPlus(j + 1, i - 1, quant_error * 3 / 16);

                }
        }

}

void ditherJarvisJudiceNinke() {

        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {

                        double oldPixel = double(m[j][i]->color);
                        double newPixel = threshold(oldPixel);
                        m[j][i]->setColor(newPixel);
                        double quant_error = oldPixel - newPixel;

                        copyColorPlus(j, i + 1, quant_error * 7 / 48);
                        copyColorPlus(j, i + 2, quant_error * 5 / 48);
                        copyColorPlus(j + 1, i + 1, quant_error * 5 / 48);
                        copyColorPlus(j + 1, i + 2, quant_error * 3 / 48);
                        copyColorPlus(j + 2, i + 1, quant_error * 3 / 48);
                        copyColorPlus(j + 2, i + 2, quant_error * 1 / 48);
                        copyColorPlus(j + 1, i, quant_error * 7 / 48);
                        copyColorPlus(j + 2, i, quant_error * 5 / 48);
                        copyColorPlus(j + 1, i - 1, quant_error * 5 / 48);
                        copyColorPlus(j + 1, i - 2, quant_error * 3 / 48);
                        copyColorPlus(j + 2, i - 1, quant_error * 3 / 48);
                        copyColorPlus(j + 2, i - 2, quant_error * 1 / 48);

                }
        }

}

void ditherSierra() {

        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {

                        double oldPixel = double(m[j][i]->color);
                        double newPixel = threshold(m[j][i]->color);
                        m[j][i]->setColor(newPixel);
                        double quant_error = oldPixel - newPixel;

                        copyColorPlus(j, i + 1, quant_error * 5 / 32);
                        copyColorPlus(j, i + 2, quant_error * 3 / 32);
                        copyColorPlus(j + 1, i + 1, quant_error * 1 / 8);
                        copyColorPlus(j + 1, i + 2, quant_error * 1 / 16);
                        copyColorPlus(j + 2, i + 1, quant_error * 1 / 16);
                        copyColorPlus(j + 1, i, quant_error * 5 / 32);
                        copyColorPlus(j + 2, i, quant_error * 3 / 32);
```

```
                                copyColorPlus(j + 1, i - 1, quant_error * 1 / 8);
                                copyColorPlus(j + 1, i - 2, quant_error * 1 / 16);
                                copyColorPlus(j + 2, i - 1, quant_error * 1 / 16);

                        }
                }

        }

        void ditherAtkinson() {

                for (int j = 0; j < height; j++) {
                        for (int i = 0; i < width; i++) {

                                double oldPixel = double(m[j][i]->color);
                                double newPixel = threshold(oldPixel);
                                m[j][i]->setColor(newPixel);
                                double quant_error = oldPixel - newPixel;

                                copyColorPlus(j, i + 1, quant_error * 1 / 8);
                                copyColorPlus(j, i + 2, quant_error * 1 / 8);
                                copyColorPlus(j + 1, i + 1, quant_error * 1 / 8);
                                copyColorPlus(j + 1, i, quant_error * 1 / 8);
                                copyColorPlus(j + 2, i, quant_error * 1 / 8);
                                copyColorPlus(j + 1, i - 1, quant_error * 1 / 8);

                        }
                }

        }


        void ditherHalftone() {

                const int order = 4;

                doubleMatrix pattern = {
                                {7 , 13,  11,   4},
                                { 12,  16,  14,   8 },
                                {10,  15,   6,   2},
                                {5,   9,   3,   1}
                };

                return orderedDitherer(pattern);

        }

};




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

        string fn = "", in = "lena512.pgm", out = "lena.pgm";
```

```
float gamma = -1;
int gradient = 0;
int mode = 0;
int bit = 8;

for (int i = 0; i < argc; i++) {
        if (i == 0) {
                fn = argv[i];
        }
        if (i == 1) {
                in = argv[i];
        }
        if (i == 2) {
                out = argv[i];
        }
        if (i == 3) {
                gradient = atoi(argv[i]);
        }
        if (i == 4) {
                mode = atof(argv[i]);
        }
        if (i == 5) {
                bit = atof(argv[i]);
        }
        if (i == 6) {
                gamma = atof(argv[i]);
        }
}

pnmBWImage im(in, gamma);

if (gradient == 1) {
        im.horizontalGradient8bit();
}

im.dither(mode, bit);
im.print(out);

if (!im.errorEncounter.empty()) {
        cerr << "Some errors encountered.";
        return 1;
}


//pnmBWImage im2(in, 1);
//for (int i = 0; i < 256; i++) {
//      cout << i << " ||| ";
//      for (int j = 1; j <= 8; j++) {
//              im2.bit = j;
//              cout << im2.threshold(i) << " ";
//      }
//      cout << "\n";
//}


// Testing all modes and bits
//gamma = 1.0;
//for (int i = 0; i <= 7; i++) {

//      for (int j = 1; j <= 8; j++) {
//              pnmBWImage im(in, gamma);

//              im.horizontalGradient8bit();
//              im.dither(i, j);
```

```
        //              im.print("pics/" + to_string(i) + "_bit" + to_string(j) + "_"
+ out);

        //              if (!im.errorEncounter.empty()) {
        //                      cerr << "Some errors encountered.";
        //                      return 1;
        //              }
        //      }

        //}

        return 0;

}
```