**Министерство образования и науки Российской Федерации**

федеральное государственное автономное образовательное

учреждение высшего образования

**Санкт-Петербургский исследовательский университет**

**Информационных технологий, механики и оптики**

Мегафакультет Трансляционных информационных технологий

Факультет Информационных технологий и программирования

**Отчет**

по лабораторной работе № 3

По дисциплине «Компьютерная геометрия и графика»

Изучение алгоритмов псевдотонирования изображений

Выполнила: студент гр. M3101

Тарасов Денис Евгеньевич

Преподаватель: Скаков П.С.

Санкт-Петербург

2020

**Цель работы:**

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

**Описание работы**

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:
  + 0 – Нет дизеринга;
  + 1 – Ordered (8x8);
  + 2 – Random;
  + 3 – Floyd–Steinberg;
  + 4 – Jarvis, Judice, Ninke;
  + 5 - Sierra (Sierra-3);
  + 6 - Atkinson;
  + 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.

**Полное решение**: все остальное

**Примеры преобразования 2 (random) при разных значениях гаммы и битностей:** [dither\_random](https://drive.google.com/drive/u/0/folders/1W9LYSeSapdiuFvu2atWtaJ5YvWRLFiHX)

Если программе передано значение, которое не поддерживается – следует сообщить об ошибке.

Коды возврата:

0 - ошибок нет

1 - произошла ошибка

В поток вывода ничего не выводится (printf, cout).

Сообщения об ошибках выводятся в поток вывода ошибок:

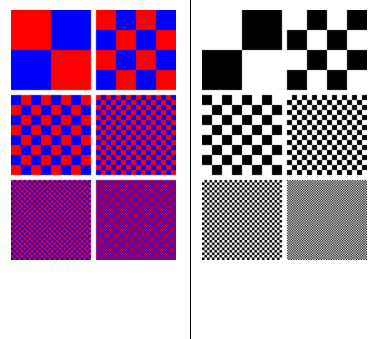
С: fprintf(stderr, "Error\n");

C++: std::cerr

Следующие параметры гарантировано не будут выходить за обусловленные значения:

* <градиент> = 0 или 1;
* <битность> = 1..8;
* width и height в файле - положительные целые значения;
* яркостных данных в файле ровно width \* height;
* <гамма> - вещественная неотрицательная;

**Теоретическая часть**

Дизеринг (англ. *dither*), псевдотонирование — при [обработке цифровых сигналов](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D1%84%D1%80%D0%BE%D0%B2%D0%B0%D1%8F_%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2) представляет собой подмешивание в первичный сигнал псевдослучайного шума со специально подобранным спектром. Применяется при обработке цифрового звука, видео и графической информации для уменьшения негативного эффекта от [квантования](https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%BD%D1%82%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%BE%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2)).

В компьютерной графике дизеринг используется для создания иллюзии [глубины цвета](https://ru.wikipedia.org/wiki/%D0%93%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D0%B0_%D1%86%D0%B2%D0%B5%D1%82%D0%B0) для изображений с относительно небольшим количеством цветов в палитре. Отсутствующие цвета составляются из имеющихся путем их «перемешивания».

Например, если необходимо получить отсутствующий в палитре фиолетовый цвет, его можно получить, разместив красные и синие пиксели в шахматном порядке; серый цвет может быть составлен из черных и белых точек

|  |
| --- |
|  |
| ***Пример для B=2*** |

Для начала разберем Алгоритм определения пороговых цветов для битностей. Вот сам алгоритм: для округления текущего значения цвета до ближайшего, который можно отобразить в задаваемой

битности **B**, из целочисленного значения цвета берутся **B** старших бит и дублируются сдвигами по **B** бит в текущее значение цвета.

Рассмотрим сами алгоритмы дизеринга. Их Можно разделить на два вида: Ordered и Error diffusion (рассеивание ошибок)

1) No dithering. Просто применяем Алгоритм определения пороговых цветов для битностей для каждого пикселя.

*Ordered dithering:*

2) ordered 8x8. Алгоритм уменьшает количество цветов, применяя карту порогов **M** (другое обозначение: Bayer matrix) к отображаемым пикселям, в результате чего некоторые пиксели меняют цвет в зависимости от расстояния исходного цвета от доступных записей цветов в уменьшенной палитре.

Алгоритм выполняет следующее преобразование для каждого цвета c каждого пикселя:

color'=findNearestPaletteColor(color + resizerM(x%n,y%n)),

где:

* color - старый цвет пикселя
* M(x%n,y%n) - элемент карты порогов
* findNearestPaletteColor - функция, возвращающая ближайший цвет к подаваемому, который можно [отобразить на текущей палитре](https://docs.google.com/document/d/1h8c5CUP9d0wpbYDe9jGUukHes-LLGX-ZyAL7WXz43G8/edit#heading=h.rj4j1g3h32y8)
* color' - новый цвет пикселя в текущей палитре

|  |
| --- |
|  |
| **Bayer matrix 8x8** |

3)Halftone Полутонирование - создание изображения со многими уровнями серого или цвета (т.е. слитный тон) на аппарате с меньшим количеством тонов, обычно чёрно-белый принтер. В принципе, задача в том чтобы уменьшить разрешение, увеличивая видимую глубину тона (так называемое пространственное полутонирование).

4) Random dithering. Это обычный ordered dithering, но вместо элемента карты порогов берется случайный элемент от 0 до 1

*Error diffusing dithering*

Все дизеринги с рассеиванием ошибок отличаются друг от друга лишь матрицей, так что я опишу алгоритм 1 раз и просто покажу матрицы.

Алгоритмы с распространением (рассеиванием) ошибок распределяют остаток квантования по соседним пикселям, которые еще не были обработаны.

В отличие от предыдущих алгоритмов, данные алгоритмы работают с некоторой окрестностью пикселя:то, что алгоритм делает в одном месте, влияет на то, что происходит в других местах. Это означает, что требуется буферизация и усложняет параллельную обработку.

Как вы могли предположить, рассеивание ошибок работает путем «рассеивания» (распространения) ошибки каждого вычисления в соседние пиксели. Если алгоритм находит серый пиксель со значением 96, он также определяет, что 96 ближе к 85, и поэтому делает пиксель темно-серым. Но тогда алгоритм учитывает «ошибку» в его преобразовании. В частности, ошибку в том, что преобразованный пиксель на самом деле был на расстоянии в 11 шагов от темно-серого.

Когда мы перемещаемся к следующему пикселю, алгоритм рассеивания ошибок добавляет ошибку предыдущего пикселя к текущему пикселю.

Это обычный алгоритм рассеивания ошибок.

Рассмотрим, например алгоритм Флойда-Штейнберга (Floyd-Stainberg).

В этом алгоритме все происходит почти так же, но ошибка не идет в один пиксель, а распределяется согласно матрице. И все похожие алгоритмы так же предлагают свою матрицу.

|  |
| --- |
|  |
| **Floyd-Stainberg** |

|  |
| --- |
|  |
| **Jarvis, Judice, Ninke** |

|  |
| --- |
|  |
| **Stucki** |

|  |
| --- |
|  |
| **Burkes** |

|  |
| --- |
|  |
| **Atkinson** |

|  |
| --- |
|  |
| **Sierra** |

**Экспериментальная часть**

Язык программирования: C++ 17

Читаем картинку, применяем к ней обратную гамма коррекцию. После этого используем один из способов дизеринга. Затем применяем гамма-коррекцию и выводим файл.

**Выводы**

Выполнение лабораторной работы позволило познакомиться с алгоритмами дизеринга и понять, как они работают. Были реализованы алгоритмы:

* No dithering
* Ordered 8x8
* Random dithering
* Halftone (4x4, orthogonal)
* Floyd-Steinberg
* Jarvis, Judice, Ninke
* Sierra (Sierra-3)
* Atkinson

**Листинг**

Source.cpp

#include <iostream>

#include <string>

#include "PGM.h"

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

if (argc != 7) {

cerr << "Command is invalid";

return 1;

}

string inpFileName=string(argv[1]);

string outFileName=string(argv[2]);

bool grad;

int numb\_dith;

int bit;

double gamma;

try {

grad = (string(argv[3]) == "1");

if (string(argv[3]) == "1") {

grad = true;

}

else if (string(argv[3]) == "0") {

grad = false;

}

else {

cerr << "Wrong gradient";

return 1;

}

if (strlen(argv[4]) != 1 || !isdigit(argv[4][0])) {

cerr << "Algo should be int number from 0 to 7";

return 1;

}

numb\_dith = atoi(argv[4]);

if (numb\_dith < 0 || numb\_dith>7) {

cerr << "Wrong algo";

return 1;

}

if (strlen(argv[5]) != 1 || !isdigit(argv[5][0])) {

cerr << "Bits should be int number from 1 to 8";

return 1;

}

bit = atoi(argv[5]);

if (bit > 8 || bit < 1) {

cerr << "Wrong bits";

return 1;

}

gamma = stold(argv[6]);

}

catch (exception& ex) {

cerr << ex.what();

return 1;

}

PGM\* pict;

try {

pict = new PGM(inpFileName, grad, gamma);

}

catch (exception& ex) {

cerr << ex.what();

return 1;

}

if (numb\_dith == 0) {

pict->no\_dithering(bit);

}

else if (numb\_dith == 1) {

pict->ordered\_dithering(bit);

}

else if (numb\_dith == 2) {

pict->random\_dithering(bit);

}

else if (numb\_dith == 3) {

pict->floyd\_dithering(bit);

}

else if (numb\_dith == 4) {

pict->Jarvis\_dithering(bit);

}

else if (numb\_dith == 5) {

pict->Sierra3\_dithering(bit);

}

else if (numb\_dith == 6) {

pict->Atkinson\_dithering(bit);

}

else if (numb\_dith == 7) {

pict->Halftone(bit);

}

else {

delete pict;

cerr << "There is no such algorithm";

return 1;

}

try {

pict->write(outFileName, gamma, bit);

}

catch (exception& ex) {

delete pict;

cerr << ex.what();

return 1;

}

delete pict;

return 0;

}

PGM.h

#pragma once

#include <vector>

#include <iostream>

#include <fstream>

#include <cmath>

#include <ctime>

using namespace std;

class PGM {

private:

vector<char> vers;

int width;

int height;

int colorDepth;

vector<vector<double>> pict;

double find\_new\_color(double a\_color, int bit) {

int color = round(a\_color);

if (bit == 8) {

return color;

}

else if (bit == 7) {

return (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 5) + (color & 1 << 4) + (color & 1 << 3) + (color & 1 << 2) + (color & 1 << 1) + (color & 1 << 7) / 128;

}

else if (bit == 6) {

double new\_color = (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 5) + (color & 1 << 4) + (color & 1 << 3) + (color & 1 << 2) + (color & 1 << 7) / 64 + (color & 1 << 6) / 64;

return new\_color;

}

else if (bit == 5) {

double new\_color = (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 5) + (color & 1 << 4) + (color & 1 << 3) + (color & 1 << 7) / 32 + (color & 1 << 6) / 32 + (color & 1 << 5) / 32;

return new\_color;

}

else if (bit == 4) {

double new\_color = (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 5) + (color & 1 << 4) + (color & 1 << 7) / 16 + (color & 1 << 6) / 16 + (color & 1 << 5) / 16 + (color & 1 << 4) / 16;

return new\_color;

}

else if (bit == 3) {

double new\_color = (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 5) + (color & 1 << 7) / 8 + (color & 1 << 6) / 8 + (color & 1 << 5) / 8 + (color & 1 << 7) / 64 + (color & 1 << 6) / 64;

return new\_color;

}

else if (bit == 2) {

double new\_color = (color & 1 << 7) + (color & 1 << 6) + (color & 1 << 7) / 4 + (color & 1 << 6) / 4 + (color & 1 << 7) / 16 + (color & 1 << 6) / 16 + (color & 1 << 7) / 64 + (color & 1 << 6) / 64;

return new\_color;

}

else if (bit == 1) {

double new\_color = (color & 1 << 7) + (color & 1 << 7) / 2 + (color & 1 << 7) / 4 + (color & 1 << 7) / 8 + (color & 1 << 7) / 16 + (color & 1 << 7) / 32 + (color & 1 << 7) / 64 + (color & 1 << 7) / 128;

return new\_color;

}

}

double sum\_between(double a, double b) {

if (a + b <= 0.0) {

return 0.0;

}

else if (a + b >= 1.0) {

return 1.0;

}

else {

return a + b;

}

}

vector<vector<double>> OrderedDitheringMatrix = { {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} };

vector<vector<int>> HalftoneMatrix= { {7, 13, 11, 4},

{12, 16, 14, 8},

{10, 15, 6, 2},

{5, 9, 3, 1}};

public:

PGM(string inpFileName,bool gradient,double gamma) {

ifstream inpFile(inpFileName, ios::binary);

if (!inpFile.is\_open()) {

throw exception("Cant open input file");

}

char vers1,vers2;

inpFile >> vers1 >> vers2 >> width >> height >> colorDepth;

if (vers1 != 'P' || vers2 != '5') {

throw exception("Wrong format");

}

vers.push\_back(vers1);

vers.push\_back(vers2);

char read\_char[1];

double new\_pixel;

inpFile.read(read\_char, 1);

pict.assign(height, vector<double>(width));

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

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

if (!gradient) {

if (inpFile.eof()) {

throw exception("Too few pixels");

}

inpFile.read(read\_char, 1);

new\_pixel = double(unsigned char(read\_char[0]))/colorDepth;

if (gamma != 0) {

new\_pixel = pow(new\_pixel, gamma);

new\_pixel \*= colorDepth;

pict[i][j] = new\_pixel;

}

else {

if (new\_pixel < 0.04045) {

new\_pixel = new\_pixel / 12.92;

pict[i][j] = new\_pixel \* colorDepth;

}

else {

new\_pixel = pow((new\_pixel + 0.055) / 1.055, 2.2);

new\_pixel \*= colorDepth;

pict[i][j] = new\_pixel;

}

}

}

else {

new\_pixel = (j \* 1.0/(double(width)-1));

if (gamma != 0) {

new\_pixel = pow(new\_pixel, gamma);

new\_pixel \*= colorDepth;

pict[i][j] = new\_pixel;

}

else {

if (new\_pixel < 0.04045) {

new\_pixel = new\_pixel / 12.92;

pict[i][j] = new\_pixel \* colorDepth;

}

else {

new\_pixel = pow((new\_pixel + 0.055) / 1.055, 2.2);

new\_pixel \*= colorDepth;

pict[i][j] = new\_pixel;

}

}

}

}

}

inpFile.close();

}

void write(string outFileName,double gamma,int bit) {

ofstream outFile(outFileName, ios::binary);

if (!outFile.is\_open()) {

throw exception("Cant open output file");

}

int NewcolorDepth = int(pow(2, bit) - 1);

outFile << vers[0] << vers[1] << '**\n**' << width << ' ' << height << '**\n**' << colorDepth << '**\n**';

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

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

if (gamma != 0) {

outFile << unsigned char(int(round(pow(pict[i][j] / colorDepth, 1.0 / gamma) \* colorDepth)));

}

else {

if (pict[i][j] / colorDepth <= 0.0031308) {

outFile << unsigned char(round(pict[i][j] \* 12.92\*colorDepth));

}

else {

unsigned char kavo = unsigned char(round(((pow(pict[i][j]/colorDepth, 1.0 / 2.2) \* 1.055) - 0.055) \* colorDepth));

outFile << kavo;

}

}

}

}

outFile.close();

}

void no\_dithering(int bit) {

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

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

try {

pict[i][j] = find\_new\_color(pict[i][j], bit);

}

catch (exception& ex) {

throw ex;

}

}

}

}

void ordered\_dithering(int bit) {

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

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

double a = (OrderedDitheringMatrix[i % 8][j % 8] / 64) - 0.5;

pict[i][j] = find\_new\_color(sum\_between(pict[i][j] / colorDepth, a)\*colorDepth, bit);

}

}

}

void random\_dithering(int bit) {

srand(time(NULL));

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

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

pict[i][j] = find\_new\_color(sum\_between(pict[i][j]/colorDepth,rand()\*1.0/(RAND\_MAX-1)-0.5) \* colorDepth, bit);

}

}

}

void floyd\_dithering(int bit) {

vector<vector<double>> errors(height, vector<double>(width, 0));

double error;

double dErr;

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

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

pict[i][j] = sum\_between(pict[i][j] / colorDepth, errors[i][j]/colorDepth)\*colorDepth;

dErr = find\_new\_color(pict[i][j], bit);

error = (pict[i][j] - dErr) / 16.0;

pict[i][j] = dErr;

if (j +1< width) {

errors[i][j + 1] += (error \* 7);

}

if (i + 1<height) {

errors[i + 1][j] += (error \* 5);

if (j - 1 >= 0) {

errors[i + 1][j - 1] += (error \* 3);

}

if (j + 1 < width) {

errors[i + 1][j + 1] += (error);

}

}

}

}

errors.clear();

}

void Jarvis\_dithering(int bit) {

vector<vector<double>> errors(height, vector<double>(width, 0));

double error;

double dErr;

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

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

pict[i][j] = sum\_between(pict[i][j] \* 1.0 / colorDepth, errors[i][j]/colorDepth) \* colorDepth;

dErr = find\_new\_color(pict[i][j], bit);

error = (pict[i][j] - dErr) / 48.0;

pict[i][j] = dErr;

if (j + 1 < width) errors[i][j + 1] += error \* 7;

if (j + 2 < width) errors[i][j + 2] += error \* 5;

if (i + 1 < height)

{

if (j - 2 >= 0) errors[i + 1][j - 2] += error \* 3;

if (j - 1 >= 0) errors[i + 1][j - 1] += error \* 5;

errors[i + 1][j] += error \* 7;

if (j + 1 < width) errors[i + 1][j + 1] += (error \* 5);

if (j + 2 < width) errors[i + 1][j + 2] += (error \* 3);

}

if (i + 2 < height)

{

if (j - 2 >= 0) errors[i + 2][j - 2] += (error \* 1);

if (j - 1 >= 0) errors[i + 2][j - 1] += (error \* 3);

errors[i + 2][j] += (error \* 5);

if (j + 1 < width) errors[i + 2][j + 1] += (error \* 3);

if (j + 2 < width) errors[i + 2][j + 2] += (error \* 1);

}

}

}

errors.clear();

}

void Sierra3\_dithering(int bit) {

vector<vector<double>> errors(height, vector<double>(width, 0));

double error;

double dErr;

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

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

pict[i][j] = sum\_between(pict[i][j] \* 1.0 / colorDepth, errors[i][j]/colorDepth) \* colorDepth;

dErr = find\_new\_color(pict[i][j], bit);

error = (pict[i][j] - dErr) / 32.0;

pict[i][j] = dErr;

if (j + 1 < width) errors[i][j + 1] += (error \* 5);

if (j + 2 < width) errors[i][j + 2] += (error \* 3);

if (i + 1 < height)

{

if (j - 2 >= 0) errors[i + 1][j - 2] += (error \* 2);

if (j - 1 >= 0) errors[i + 1][j - 1] += (error \* 4);

errors[i + 1][j] += (error \* 5);

if (j + 1 < width) errors[i + 1][j + 1] += (error \* 4);

if (j + 2 < width) errors[i + 1][j + 2] += (error \* 2);

}

if (i + 2 < height)

{

if (j - 1 >= 0) errors[i + 2][j - 1] += (error \* 2);

errors[i + 2][j] += (error \* 3);

if (j + 1 < width) errors[i + 2][j + 1] += (error \* 2);

}

}

}

errors.clear();

}

void Atkinson\_dithering(int bit) {

vector<vector<double>> errors(height, vector<double>(width, 0));

double error;

double dErr;

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

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

pict[i][j] = sum\_between(pict[i][j] \* 1.0 / colorDepth, errors[i][j]/colorDepth) \* colorDepth;

dErr = find\_new\_color(pict[i][j], bit);

error = (pict[i][j] - dErr) / 8.0;

pict[i][j] = dErr;

if (j + 1 < width) errors[i][j + 1] += error;

if (j + 2 < width) errors[i][j + 2] += error;

if (i + 1 < height)

{

if (j - 1 >= 0) errors[i + 1][j - 1] += error;

errors[i + 1][j] += error;

if (j + 1 < width) errors[i + 1][j + 1] += error;

}

if (i + 2 < height)

{

errors[i + 2][j] += error;

}

}

}

errors.clear();

}

void Halftone(int bit) {

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

for (int j = 0; j < width; j++)

{

double a = HalftoneMatrix[i % 4][j % 4] / 17.0 - 0.5;

pict[i][j] = find\_new\_color(sum\_between(pict[i][j]/colorDepth, a)\*colorDepth,bit);

}

}

}

};