МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение высшего образования

Санкт-Петербургский национальный 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**

**Выполнил студент группы №M3234:**

***Ота Никита Терентьевич***

**Преподаватель:**

***Скаков Павел Сергеевич***

**САНКТ-ПЕТЕРБУРГ**

**2020**

**Цель работы:**

Реализовать программу, которая позволяет проводить преобразования между цветовыми пространствами.

Входные и выходные данные могут быть как одним файлом ppm, так и набором из 3 pgm.

**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.

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:

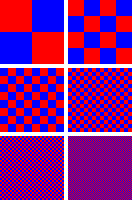
* <градиент> = 0 или 1;
* <битность> = 1..8;
* width и height в файле - положительные целые значения;
* яркостных данных в файле ровно width \* height;
* <гамма> - вещественная неотрицательная;

**Теоретическая часть:**

Дизеринг (англ. dither), псевдотонирование — при обработке цифровых сигналов 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.

В компьютерной графике дизеринг используется для создания иллюзии глубины цвета 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 получения нового цвета

Дизеринг изображения применяется при уменьшении её размерности. В обычном случае отбрасываются младшие, самые малозначимые биты. Зачастую это ведет к резкому ухудшению изображения. Дизеринг призван исправить это. Из-за вноса специального шума человеческий глаз меньше замечает переход к меньшей размерности.

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.

**Random**

Алгоритм добавляет в каждый пиксель случайный шум, что в общей картине дает улучшение изображения.

**Ordered**

Алгоритм добавляет какой-то определённый шум, исходя из конкретной матрицы.

**Floyd–Steinberg**

**Jarvis, Judice, Ninke**

**Sierra (Sierra-3)**

**Atkinson**

Эти алгоритмы используют ошибку (разность между оригинальным значением и новым) для получения шума. Проход по всем пикселям с выделением ошибки производится последовательно. Алгоритмы распределяют остаток квантования по соседним пикселям, которые еще не были обработаны.

**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.

**Практическая часть:**

Язык, на котором написана программа, решающая поставленную задачу: С++.

Ключи компиляции: -std=c++11 -fsanitize=address -fno-stack-limit.

Компилятор: g++.

**План работы программы:**

1. Чтение данных из командной строки, обработка неправильных введённых данных.
2. Чтение данных с введённого пользователем файла входа с обработкой ошибок.
3. Применение конкретного алгоритма дизеринга.
4. Вывод результата в файл с обработкой ошибок.

**Код программ:**

**main.cpp**

#include <iostream>

#include <fstream>

#include <cstring>

#include <string>

#include <math.h>

#include "dithering.h"

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

if (argc != 7) {

std::cerr << "Got " << argc << " arguments\n";

std::cerr << "Usage: <input file> <output file> <gradient> <dithering> <bitness> <gamma>\n";

return EXIT\_FAILURE;

}

// Guaranteed argv[3] == "0" or "1"

int const gradient = (strcmp(argv[3], "0") == 0 ? 0 : 1);

int dithering;

try {

dithering = std::stoi(argv[4]);

} catch (std::invalid\_argument const& e) {

std::cerr << "<dithering> must be an int value\n";

std::cerr << e.what() << '\n';

return EXIT\_FAILURE;

} catch (std::out\_of\_range const& e) {

std::cerr << "Got too big value in <dithering> to parse as int\n";

std::cerr << e.what() << '\n';

return EXIT\_FAILURE;

}

if (dithering < 0 || dithering > 7) {

std::cerr << "<dithering> must be an int value in [0..7]\n";

return EXIT\_FAILURE;

}

// Guaranteed argv[5] in [0..8]

int bitness = std::stoi(argv[5]);

// Guaranteed argv[6] is positive number

double gamma = std::stod(argv[6]);

bool sRGB = (strcmp(argv[6], "0") == 0);

std::string input = argv[1];

if (input.length() < 4 || input.substr(input.length() - 4, 4) != ".pgm") {

std::cerr << "Given input file is not pgm\n";

return EXIT\_FAILURE;

}

std::string output = argv[2];

if (output.length() < 4 || output.substr(output.length() - 4, 4) != ".pgm") {

std::cerr << "Given output file is not pgm\n";

return EXIT\_FAILURE;

}

std::ifstream in(argv[1], std::ios::binary);

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

std::cerr << "Couldn't open the file " << argv[1] << '\n';

return EXIT\_FAILURE;

}

char p;

int type;

in >> p >> type;

if (p != 'P' || type != 5) {

std::cerr << "The file " << input << " must have P5 format or couldn't read first line info\n";

in.close();

return EXIT\_FAILURE;

}

int width, height;

in >> width >> height;

// Guaranteed height, width are ints and greater than 0

int brightness;

in >> brightness;

char line\_separaor = in.get();

if (brightness != MAX\_BRIGHTNESS || line\_separaor != '\n') {

std::cerr << "At the third line should be 255 or couldn't get that info in the file " << input << '\n';

in.close();

return EXIT\_FAILURE;

}

picture pic(height, width, brightness, gradient, in);

switch (dithering) {

case 0:

without\_dithering(pic, bitness, sRGB, gamma);

break;

case 1:

ordered\_dithering(pic, bitness, sRGB, gamma);

break;

case 2:

random\_dithering(pic, bitness, sRGB, gamma);

break;

case 3:

Floyd\_Steinberg\_dithering(pic, bitness, sRGB, gamma);

break;

case 4:

Jarvis\_Judice\_Ninke\_dithering(pic, bitness, sRGB, gamma);

break;

case 5:

Sierra\_3\_dithering(pic, bitness, sRGB, gamma);

break;

case 6:

Atkinson\_dithering(pic, bitness, sRGB, gamma);

break;

case 7:

Halftone\_dithering(pic, bitness, sRGB, gamma);

break;

}

std::ofstream out(output, std::ios::binary);

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

std::cerr << "Couldn't open the output file " << output << '\n';

// picture destructor successfully calls clear\_array()

return EXIT\_FAILURE;

}

out << "P5\n";

out << width << ' ' << height << '\n';

out << brightness << '\n';

double \*\*data = pic.get\_all();

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

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

if (!out || out.bad()) {

std::cerr << "Something happened with an output stream while writing in it data\n";

// picture destructor successfully calls clear\_array()

return EXIT\_FAILURE;

}

out << static\_cast<unsigned char>(floor(data[h][w]));

}

}

out.close();

return EXIT\_SUCCESS;

}

**dithering.h**

#include <fstream>

#include <cstdlib>

int const MAX\_BRIGHTNESS = 255;

double const S\_RGB\_BARRIER = 0.04045;

double const S\_RGB\_DIVISOR = 12.92;

double const S\_RGB\_DEGREE = 2.4;

double const S\_RGB\_BASE\_TERM = 0.055;

double const S\_RGB\_BASE\_DIVISOR = 1.055;

template <typename T>

void clear\_array(T \*\*arr, int const n);

struct picture {

picture() = default;

picture(int h, int w, int m, int g, std::ifstream &in);

~picture();

int get\_width() const;

int get\_height() const;

int get\_max\_brightness() const;

double\*\* get\_all();

double get(int const h, int const w);

void set(int const h, int const w, double const x);

private:

int width = 0;

int height = 0;

int max\_brightness = MAX\_BRIGHTNESS;

double \*\*data = nullptr;

int counter = 0;

};

int const ORDERED\_SIZE = 8;

int const ORDERED[ORDERED\_SIZE][ORDERED\_SIZE] = {

{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}

};

int const HALFTONE\_SIZE = 4;

int const HALFTONE[HALFTONE\_SIZE][HALFTONE\_SIZE] = {

{6, 12, 10, 3},

{11, 15, 13, 7},

{9, 14, 5, 1},

{4, 8, 2, 0}

};

double const ONE\_SIXTEENTH = 1.0 / 16;

double const ONE\_FORTY\_EIGHTH = 1.0 / 48;

double const ONE\_THIRTY\_SECOND = 1.0 / 32;

double const ONE\_EIGHTH = 1.0 / 8;

double const ONE\_SECOND = 1.0 / 2;

void without\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void ordered\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void random\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void Floyd\_Steinberg\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void Jarvis\_Judice\_Ninke\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void Sierra\_3\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void Atkinson\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

void Halftone\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma);

**dithering.cpp**

#include <vector>

#include <algorithm>

#include <numeric>

#include <math.h>

#include <iostream>

#include <random>

#include <chrono>

#include "dithering.h"

template <typename T>

void clear\_array(T \*\*arr, int n) {

if (arr == nullptr) {

return;

}

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

if (arr[i] != nullptr) {

delete[] arr[i];

}

arr[i] = nullptr;

}

delete[] arr;

arr = nullptr;

}

picture::picture(int h, int w, int m, int g, std::ifstream &in) : height(h), width(w), max\_brightness(max\_brightness) {

data = new double\*[height];

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

data[h] = new double[width];

}

// std::cout << "Initializing picture: " << width << " \* " << height << '\n';

if (g == 1) {

// std::cout << "Gradient mode chosen\n";

double gradient\_coefficient = static\_cast<double>(MAX\_BRIGHTNESS) / (width - 1);

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

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

data[h][w] = gradient\_coefficient \* w;

}

}

} else {

// std::cout << "Picture mode chosen\n";

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

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

data[h][w] = in.get();

// Guaranteed file has width \* height bytes

if (in.gcount() == 0 || in.bad()) {

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

delete[] data[i];

}

delete[] data;

in.close();

throw std::runtime\_error("Something happened with an input stream while reading info\n");

}

}

}

}

in.close();

// std::cout << "Created\n";

// std::cout << "data[0][0] = " << data[0][0] << '\n';

}

picture::~picture() {

clear\_array(data, height);

}

int picture::get\_height() const {

return height;

}

int picture::get\_width() const {

return width;

}

int picture::get\_max\_brightness() const {

return max\_brightness;

}

double\*\* picture::get\_all() {

return data;

}

double picture::get(int const h, int const w) {

// std::cout << "Getting data[" << h << "][" << w << "]: ";

return data[h][w];

}

void picture::set(int const h, int const w, double const x) {

// std::cout << "Setting data[" << h << "][" << w << "] = " << x << std::endl;

data[h][w] = x;

}

std::vector<int> get\_even\_scatter(int bitness) {

std::vector<int> res(1 << bitness);

if (bitness == 8) {

std::iota(res.begin(), res.end(), 0);

return res;

}

int mid\_vals = res.size() - 2;

int sum\_up = MAX\_BRIGHTNESS - (1 << bitness) + 1;

int stride = sum\_up / (mid\_vals + 1), remnant = sum\_up % (mid\_vals + 1);

int pos = 1, val = 1;

while (mid\_vals--) {

val += stride;

if (remnant-- > 0) {

++val;

}

res[pos++] = val;

}

res.back() = MAX\_BRIGHTNESS;

return res;

}

int get\_less\_or\_equal(std::vector<int> const& arr, int const x) {

std::vector<int>::const\_iterator pos = std::lower\_bound(arr.begin(), arr.end(), x);

if (\*pos == x) {

// equals

return x;

}

return \*(pos - 1);

}

int get\_greater\_or\_equal(std::vector<int> const& arr, int const x) {

std::vector<int>::const\_iterator pos = std::upper\_bound(arr.begin(), arr.end(), x);

if (pos == arr.end()) {

return \*arr.rbegin();

}

return \*pos;

}

double correction(double val, bool const sRGB, double const gamma) {

val /= MAX\_BRIGHTNESS;

if (sRGB) {

if (val <= S\_RGB\_BARRIER) {

val /= S\_RGB\_DIVISOR;

} else {

val = pow((val + S\_RGB\_BASE\_TERM) / S\_RGB\_BASE\_DIVISOR, S\_RGB\_DEGREE);

}

} else {

val = pow(val, gamma);

}

val \*= MAX\_BRIGHTNESS;

return val;

}

inline int correct\_bounds(double val) {

return static\_cast<int>(floor(fmin(MAX\_BRIGHTNESS, fmax(0, val))));

}

double get\_nearest(double const val, int const bitness, bool const sRGB, double const gamma,

bool const without\_dist, bool const with\_dist, double const add) {

static std::vector<int> const scatter = get\_even\_scatter(bitness);

// static int cnt = 0;

// static std::ofstream out("test.log", std::ios::app);

double left = correction(get\_less\_or\_equal(scatter, val), sRGB, gamma);

// std::cout << "Got left = " << left << std::endl;

double right = correction(get\_greater\_or\_equal(scatter, val), sRGB, gamma);

// std::cout << "Got right = " << right << std::endl;

double middle = correction(val, sRGB, gamma);

// std::cout << "Got middle = " << middle << std::endl;

// if (++cnt % 65536 <= 10) {

// out << "=== " << cnt << " ===" << std::endl;

// out << "val = " << val << std::endl;

// out << "left = " << left << ", right = " << right << std::endl << "middle = " << middle;

// }

if (without\_dist) {

middle += add;

// if (cnt % 65536 <= 10) {

// out << ", add = " << add << std::endl;

// }

}

if (with\_dist) {

middle += add \* (right - left);

// if (cnt % 65536 <= 10) {

// out << ", add = " << add << std::endl;

// }

}

// if (cnt % 65536 <= 10) {

// out << "middle + add = " << middle << std::endl;

// out.flush();

// }

double res;

if (fabs(middle - left) < fabs(middle - right)) {

res = left;

} else {

res = right;

}

// if (cnt % 65536 <= 10) {

// out << "correct\_bounds(res) = " << correct\_bounds(res) << '\n';

// out.flush();

// }

return res;

}

int round\_pixel(double const val, int const bitness, bool const sRGB, double const gamma,

bool const without\_dist, bool const with\_dist, double const add) {

return correct\_bounds(get\_nearest(val, bitness, sRGB, gamma, without\_dist, with\_dist, add));

}

void without\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

for (int h = 0; h < pic.get\_height(); ++h) {

for (int w = 0; w < pic.get\_width(); ++w) {

pic.set(h, w, round\_pixel(pic.get(h, w), bitness, sRGB, gamma, false, false, 0));

}

}

}

void ordered\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

for (int h = 0; h < pic.get\_height(); ++h) {

for (int w = 0; w < pic.get\_width(); ++w) {

double add = ORDERED[h % ORDERED\_SIZE][w % ORDERED\_SIZE];

add = (add + ONE\_SECOND) / ORDERED\_SIZE / ORDERED\_SIZE - ONE\_SECOND;

pic.set(h, w, round\_pixel(pic.get(h, w), bitness, sRGB, gamma, false, true, add));

}

}

}

void random\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

unsigned seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

std::mt19937 rnd(seed);

int bound = static\_cast<int>(1e7);

for (int h = 0; h < pic.get\_height(); ++h) {

for (int w = 0; w < pic.get\_width(); ++w) {

long long curr = rnd();

double add = ((curr % (2 \* bound + 1)) - bound) / static\_cast<double>(bound);

pic.set(h, w, round\_pixel(pic.get(h, w), bitness, sRGB, gamma, false, true, add / 2));

}

}

}

void update\_error(std::vector<double> &error, std::vector<int> const& coefficients,

int const pos, int const barrier, double const error\_val) {

int shift = (coefficients.size() == 2 ? 3 : 0);

if (pos + 1 < barrier) {

error[pos + 1] += error\_val \* coefficients[3 - shift];

}

if (pos + 2 < barrier) {

error[pos + 2] += error\_val \* coefficients[4 - shift];

}

if (coefficients.size() == 2) {

return;

}

if (pos - 2 >= 0) {

error[pos - 2] += error\_val \* coefficients[0];

}

if (pos - 1 >= 0) {

error[pos - 1] += error\_val \* coefficients[1];

}

error[pos] += error\_val \* coefficients[2];

}

void dithering\_with\_errors(picture &pic, int const bitness, bool const sRGB, double const gamma,

double const fraction, std::vector<int> const& prev,

std::vector<int> const& curr, std::vector<int> const& next) {

std::vector<double> prev\_error(pic.get\_width());

std::vector<double> curr\_error(pic.get\_width());

std::vector<double> next\_error(pic.get\_width());

if (prev.size() != 2) {

throw std::runtime\_error("prev.size() must be 2");

}

if (curr.size() != 5) {

throw std::runtime\_error("curr.size() must be 5");

}

if (next.size() != 5) {

throw std::runtime\_error("next.size() must be 5");

}

for (int h = 0; h < pic.get\_height(); ++h) {

for (int w = 0; w < pic.get\_width(); ++w) {

double curr\_val = get\_nearest(pic.get(h, w), bitness, sRGB, gamma, true, false, prev\_error[w]);

double middle = correction(pic.get(h, w), sRGB, gamma) + prev\_error[w];

double error = (middle - curr\_val) \* fraction;

update\_error(prev\_error, prev, w, pic.get\_width(), error);

update\_error(curr\_error, curr, w, pic.get\_width(), error);

update\_error(next\_error, next, w, pic.get\_width(), error);

pic.set(h, w, curr\_val);

}

prev\_error.swap(curr\_error);

curr\_error.swap(next\_error);

next\_error.assign(pic.get\_width(), 0);

}

}

void Floyd\_Steinberg\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

dithering\_with\_errors(pic, bitness, sRGB, gamma, ONE\_SIXTEENTH, {7, 0}, {0, 3, 5, 1, 0}, {0, 0, 0, 0, 0});

}

void Jarvis\_Judice\_Ninke\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

dithering\_with\_errors(pic, bitness, sRGB, gamma, ONE\_FORTY\_EIGHTH, {7, 5}, {3, 5, 7, 5, 3}, {1, 3, 5, 3, 1});

}

void Sierra\_3\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

dithering\_with\_errors(pic, bitness, sRGB, gamma, ONE\_THIRTY\_SECOND, {5, 3}, {2, 4, 5, 4, 2}, {0, 2, 3, 2, 0});

}

void Atkinson\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

dithering\_with\_errors(pic, bitness, sRGB, gamma, ONE\_EIGHTH, {1, 1}, {0, 1, 1, 1, 0}, {0, 0, 1, 0, 0});

}

void Halftone\_dithering(picture &pic, int const bitness, bool const sRGB, double const gamma) {

for (int h = 0; h < pic.get\_height(); ++h) {

for (int w = 0; w < pic.get\_width(); ++w) {

double add = HALFTONE[h % HALFTONE\_SIZE][w % HALFTONE\_SIZE];

add = (add + ONE\_SECOND) / HALFTONE\_SIZE / HALFTONE\_SIZE - ONE\_SECOND;

pic.set(h, w, round\_pixel(pic.get(h, w), bitness, sRGB, gamma, false, true, add));

}

}

}