

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 графика и геометрия»**

**Изучение алгоритмов псевдотонирования изображений**

**Выполнил студент группы М3101**

***Кузьмук Павел Юрьевич***

**Проверил:**

**Скаков Павел Сергеевич**

**САНКТ-ПЕТЕРБУРГ**

**2020**

## **ЦЕЛЬ РАБОТЫ**

Изучение алгоритмов и реализация программы, применяющей алгоритмы 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:
  - o 0 – Нет дизеринга;
  - o 1 – Ordered (8x8);
  - o 2 – Random;
  - o 3 – Floyd–Steinberg;
  - o 4 – Jarvis, Judice, Ninke;
  - o 5 - Sierra (Sierra-3);
  - o 6 - Atkinson;
  - o 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

### *Изучение алгоритмов псевдотонирования изображений*

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:

C: 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

### Определение и применение дизеринга:

*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».

При оптимизации изображений путём уменьшения количества цветов, применение дизеринга приводит к визуальному улучшению изображения.

### Виды дизеринга:

Все алгоритмы дизеринга условно можно поделить на 2 вида: алгоритмы с рассеиванием ошибки (Error diffusion) и упорядоченные алгоритмы (Ordered).

Для определения пороговых (threshold) цветов для разных битностей воспользуемся следующим алгоритмом: 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.

### Алгоритмы дизеринга, использующиеся в лабораторной работе:

#### 1. Нет дизеринга (no dithering)

Данный метод подразумевает лишь округление всех цветов до пороговых.

#### Алгоритмы с упорядоченным распределением ошибки (Ordered):

#### 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.

Для каждого пикселя производится смещение его значения цвета на соответствующее значение из карты порогов  $M$  в соответствии с его местоположением, в результате чего значение пикселя квантуется на другой цвет, если оно превышает пороговое значение.

Для большинства случаев сглаживания достаточно просто добавить пороговое значение к каждому пикселю или эквивалентно сравнить значение этого пикселя с порогом: если значение пикселя меньше, чем число в соответствующей ячейке матрицы, записать в пиксель черный цвет, в противном случае, белый в случае битности 1.

Поскольку алгоритм работает с одиночными пикселями и не имеет условных операторов, он очень быстрый и подходит для преобразований в реальном времени. Кроме того, расположение шаблонов сглаживания всегда остается одинаковым относительно кадра дисплея, что способствует улучшению сжатия изображения. Упорядоченное сглаживание больше подходит для линейной графики, так как приводит к более прямым линиям и меньшему количеству аномалий. Однако, результат, получаемый после работы данного метода, получаются хуже, чем после применения алгоритмов с рассеиванием ошибок, о которых будет изложено далее.

### **3. Halftone (4x4, orthogonal)**

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 случае обработки цифрового изображения Halftone представляет собой матрицы порогов, позволяющие воспроизводить “точки” как при печати изображения.

### **4. Random**

Данный алгоритм можно также отнести к типу ordered с тем условием, что для определения добавки к значению текущего пикселя берется не элемент матрицы, а случайное число.

Алгоритмы с рассеиванием ошибки (Error diffusion):

### **5. Floyd–Steinberg**

Первая и возможно самая известная формула рассеивания ошибок была опубликована Робертом Флойдом и Луисом Стейнбергом в 1976 году. Рассеивание ошибок происходит по следующей схеме:

- **x** - текущий пиксель, от которого распространяется ошибка
- **y,x** - строка/столбец изображения
- **ymx, xmx** - номер последние строки и столбца

### Изучение алгоритмов псевдотонирования изображений

y=0	x=0										xmx
y	x	7/16			x	7/16					x
	5/16	1/16		3/16	5/16	1/16				3/16	5/16
ymx											

Пример преобразования пиксельного значения 96: при окрашивании пикселя в темно-серый мы получаем ошибку 11. Мы распространяем эту ошибку окружающим пикселям, поделив 11 на 16 (= 0,6875), затем умножаем её на соответствующие значения, например:

	X	+7 * 0,6875
+3 * 0,6875	+5 * 0,6875	+1 * 0,6875

Если наш пиксель округляется в большее значение (например, 129 к 170), то ошибка будет отрицательной соответственно.

Алгоритм даёт достаточно хорошее качество, а также требует только один передний массив (одномерный массив шириной в изображение, где хранятся значения ошибок, распространяемые к следующей строке). Кроме того, поскольку его делитель 16, вместо деления можно использовать битовые сдвиги. Так алгоритм достигает высокой скорости работы даже на старом оборудовании.

Что касается значений 1/3/5/7, используемых для распространения ошибки – они были выбраны специально, потому что они создают равномерный клетчатый узор для серого изображения.

## 6. Jarvis, Judice, Ninke

В год, когда Флойд и Стейнберг опубликовали свой знаменитый алгоритм дithering, был издан менее известный, но гораздо более мощный алгоритм. Фильтр Джарвиса, Джудиса и Нинке значительно сложнее, чем Флойда-Стейнберга:

$$\frac{1}{48} \begin{bmatrix} - & - & \# & 7 & 5 \\ 3 & 5 & 7 & 5 & 3 \\ 1 & 3 & 5 & 3 & 1 \end{bmatrix}$$

Алгоритм Джарвиса, Джудиса и Нинке аналогичен алгоритму Флойда-Стейнберга с точностью до матрицы распределения ошибки. При данном алгоритме ошибка распределяется на в три раза больше пикселей, чем у Флойда-Стейнберга, что приводит к более гладкому и более тонкому результату.

## 7. Sierra (Sierra-3)

Аналогично алгоритму Флойда-Стейнберга, но с другой матрицей:

$$\frac{1}{32} \begin{bmatrix} -\textcolor{red}{1} & -\textcolor{red}{1} & \textcolor{red}{5} & 3 \\ 2 & 4 & \textcolor{red}{4} & 2 \end{bmatrix}$$



## 8. Atkinson

Аналогично алгоритму Флойда-Стейнберга, но с другой матрицей:

$$\frac{1}{8} \begin{bmatrix} -\textcolor{red}{i} & \textcolor{red}{i} & 1 \\ 1 & 1 & \textcolor{red}{i} \end{bmatrix} \begin{bmatrix} 1 & -\textcolor{red}{i} & -\textcolor{red}{i} & 1 & -\textcolor{red}{i} & -\textcolor{red}{i} \\ -\textcolor{red}{i} & -\textcolor{red}{i} & -\textcolor{red}{i} & -\textcolor{red}{i} & -\textcolor{red}{i} & -\textcolor{red}{i} \end{bmatrix}$$

## **ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ**

Лабораторная работа выполнена на языке C++. Стандарт языка C++14.

Процесс псевдотонирования выглядит следующим образом: читаем изображение из файла, сразу же применяя обратную гамма-коррекцию. Вызываем метод `dither` у объекта изображения, где происходит сжатие палитры цветов и изменение цветов определенных пикселей (возможно также рассеивание ошибок), после чего на этапе вывода изображения применяем гамма-коррекцию.

## **ВЫВОД**

Выполнение данной лабораторной работы позволило изучить алгоритмы псевдотонирования изображений как с упорядоченным распределением ошибок, так и с рассеиванием ошибок. Были реализованы следующие алгоритмы для псевдотонирования изображений:

1. Ordered (8x8)
2. Random
3. Floyd–Steinberg
4. Jarvis, Judice, Ninke
5. Sierra (Sierra-3)
6. Atkinson
7. Halftone (4x4, orthogonal)

При реализации чтения и записи изображения была изучена гамма-коррекция значений пикселей изображения. Реализована гамма-коррекция для вещественного значения гаммы, а также sRGB гамма-коррекция.

## ЛИСТИНГ КОДА

main.cpp:

```
#include <iostream>

#include <string>
#include <cmath>
#include <ctime>
#include "pgm_image.h"

using namespace std;

int main(int argc, char *argv[]) {
    srand(time(NULL));
    if(argc != 7) {
        cerr << "command line arguments are invalid" << endl;
        return 1;
    }
    string fin = string(argv[1]);
    string fout = string(argv[2]);
    bool gradient;
    int algo, bit;
    bool srgb = false;
    double gamma;
    try {
        gradient = (string(argv[3]) == "1");
        algo = atoi(argv[4]);
        bit = atoi(argv[5]);
        if(string(argv[6]) == "0" || string(argv[6]) == "0.0")
        {
            gamma = 2.4;
            srgb = true;
        } else {
            gamma = stold(argv[6]);
        }
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
        return 1;
    }
    PGM_Image* image;
    try {
        image = new PGM_Image(fin, gradient, gamma, srgb);
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
        return 1;
    }
```

```
}

image -> dither(bit, algo, gamma, srgb);

try {
image -> drop(fout, gamma, srgb, bit);
delete(image);
}
catch (const exception& e) {
cerr << e.what() << endl;
return 1;
}
}
```

pgm\_image.h:

```
#pragma once

#include <vector>
#include <algorithm>
#include <stdexcept>
#include <fstream>

using namespace std;

class PGM_Image{
private:
int width, height, color_depth;
vector<vector<double>> image;
vector<vector<double>> err;
public:
PGM_Image(string, bool, double, bool);
void drop(string, double, bool, int);
void dither(int, int, double, bool);
};
```

pgm\_image.cpp:

```
#include "pgm_image.h"

#include <cmath>
#include <iostream>

const double ld1 = 1;

vector<vector<int> > 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},
},
HalftoneMatrix = {
{7, 13, 11, 4},
{12, 16, 14, 8},
{10, 15, 6, 2},
{5, 9, 3, 1},
};

double sum_with_of(double a, double b) {
double res = a + b;
if(res <= 0.0)
return 0.0;
if(res >= 1.0)
return 1.0;
return res;
}

PGM_Image::PGM_Image(string filename, bool gradient, double gamma,
bool srgb) {
ifstream fin(filename, ios::binary);
if(!fin.is_open())
throw runtime_error("failed to open file");

char cc[2];
fin >> cc[0] >> cc[1];
if(cc[0] != 'P' || cc[1] != '5')
throw runtime_error("expected P5 format");
fin >> width >> height >> color_depth;
image.assign(height, vector<double>(width));
char pixel;
fin.read(&pixel, 1);
for(int i = 0; i < height; i++)
for(int j = 0; j < width; j++)
{
if(!gradient) {
fin.read(&pixel, sizeof(unsigned char));
double old = (double)((unsigned char)pixel) / (double)color_depth;
if(srgb)
old = (old < 0.04045 ? old / 12.92 : pow((old + 0.055) / 1.055,
gamma));
else
```

```
old = pow(old, gamma);
image[i][j] = old;
} else {
image[i][j] = j*ld1 / (width - 1);
}
}
fin.close();
}

void PGM_Image::drop(string filename, double gamma, bool srgb, int
bit) {
color_depth = (1 << bit) - 1;
ofstream fout(filename, ios::binary);
if(!fout.is_open()) {
throw runtime_error("cannot open output file");
}
fout << "P5\n" << width << ' ' << height << '\n' << color_depth << '\n';

for(int i = 0; i < height; i++)
for(int j = 0; j < width; j++)
{
double old = image[i][j];
if(srgb)
old = (old <= 0.0031308 ? old * 12.92 : pow(old, ld1/gamma)*1.055 -
0.055);
else
old = pow(old, ld1 / gamma);
int color = round(old * (double)color_depth);
fout << (unsigned char)color;
}
fout.flush();
fout.close();
}

void PGM_Image::dither(int bit, int algo, double gamma, bool srgb) {
auto nearest_color = [&bit, this](double pixel_color){
return round(pixel_color * ((1 << bit) - 1)) / ((1 << bit) - 1);
};

// No dithering
if(algo == 0) {
for(int i = 0; i < height; i++)
for(int j = 0; j < width; j++)
image[i][j] = nearest_color(image[i][j]);
return;
}
```

```
// Ordered 8x8
if(algo == 1) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
double potpl = (OrderedDitheringMatrix[i%8][j%8] / 64.0) - 0.5;
image[i][j] = nearest_color(sum_with_of(image[i][j], potpl));
}
return;
}
// Random
if(algo == 2) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
double potpl = rand() * 1.0 / (RAND_MAX-1) - 0.5;
image[i][j] = nearest_color(sum_with_of(image[i][j], potpl));
}
return;
}
// Halftone 4x4
if(algo == 7) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
double potpl = HalftoneMatrix[i%4][j%4]/16.0 - 0.5;
image[i][j] = nearest_color(sum_with_of(image[i][j], potpl));
}
return;
}

err.assign(height, vector<double>(width, 0));
// Floyd-Steinberg
if(algo == 3) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
image[i][j] = sum_with_of(image[i][j], err[i][j]);
double nc = nearest_color(image[i][j]);
double error = (image[i][j] - nc) / 16.0;
image[i][j] = nc;
if(j + 1 < width)
err[i][j+1] += error * 7;
if(i + 1 < height) {
if(j - 1 >= 0)
err[i+1][j-1] += error * 3;
err[i+1][j] += error * 5;
}
```



```
if(j + 1 < width)
err[i+1][j+1] += error;
}
}
return;
}
// Jarvis, Judice, Ninke
if(algo == 4) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
image[i][j] = sum_with_of(image[i][j], err[i][j]);
double nc = nearest_color(image[i][j]);
double error = (image[i][j] - nc) / 48.0;
image[i][j] = nc;
if(j + 1 < width) err[i][j+1] += error * 7;
if(j + 2 < width) err[i][j+2] += error * 5;
if(i + 1 < height)
{
if(j - 2 >= 0) err[i+1][j-2] += error * 3;
if(j - 1 >= 0) err[i+1][j-1] += error * 5;
err[i+1][j] += error * 7;
if(j + 1 < width) err[i+1][j+1] += (error * 5);
if(j + 2 < width) err[i+1][j+2] += (error * 3);
}
if(i + 2 < height)
{
if(j - 2 >= 0) err[i+2][j-2] += (error * 1);
if(j - 1 >= 0) err[i+2][j-1] += (error * 3);
err[i+2][j] += (error * 5);
if(j + 1 < width) err[i+2][j+1] += (error * 3);
if(j + 2 < width) err[i+2][j+2] += (error * 1);
}
}
return;
}
// Sierra-3
if(algo == 5) {
for(int i = 0; i < height; i ++)
for(int j = 0; j < width; j ++)
{
image[i][j] = sum_with_of(image[i][j], err[i][j]);
double nc = nearest_color(image[i][j]);
double error = (image[i][j] - nc) / 32.0;
image[i][j] = nc;
if(j + 1 < width) err[i][j+1] += (error * 5);
if(j + 2 < width) err[i][j+2] += (error * 3);
```

```
if(i + 1 < height)
{
if(j - 2 >= 0) err[i+1][j-2] += (error * 2);
if(j - 1 >= 0) err[i+1][j-1] += (error * 4);
err[i+1][j] += (error * 5);
if(j + 1 < width) err[i+1][j+1] += (error * 4);
if(j + 2 < width) err[i+1][j+2] += (error * 2);
}
if(i + 2 < height)
{
if(j - 1 >= 0) err[i+2][j-1] += (error * 2);
err[i+2][j] += (error * 3);
if(j + 1 < width) err[i+2][j+1] += (error * 2);
}
}
return;
}
// Atkinson
if(algo == 6) {
for(int i = 0; i < height; i++)
for(int j = 0; j < width; j++)
{
image[i][j] = sum_with_of(image[i][j], err[i][j]);
double nc = nearest_color(image[i][j]);
double error = (image[i][j] - nc) / 8.0;
image[i][j] = nc;
if(j + 1 < width) err[i][j+1] += error;
if(j + 2 < width) err[i][j+2] += error;
if(i + 1 < height)
{
if(j - 1 >= 0) err[i+1][j-1] += error;
err[i+1][j] += error;
if(j + 1 < width) err[i+1][j+1] += error;
}
if(i + 2 < height)
{
err[i+2][j] += error;
}
}
}
return;
}
}
```