**Министерство образования и науки Российской Федерации**

федеральное государственное автономное образовательное

учреждение высшего образования

**Санкт-Петербургский исследовательский университет**

**Информационных технологий, механики и оптики**

Факультет информационных технологий и программирования

Дисциплина: Компьютерная геометрия и графика

**Отчет**

по лабораторной работе №3

***Изучение алгоритмов псевдотонирования изображений***

Выполнила: студент гр. M3101

Семина А. Д.

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**

# *Определение*

**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. |  |  |

При оптимизации изображений путём уменьшения количества цветов, применение дизеринга приводит к визуальному улучшению изображения (однако для отдельных сжатых форматов (например, [PNG](https://ru.wikipedia.org/wiki/PNG)), увеличивает его размер).  Дизеринг также находит применение в веб-дизайне, где этот полезный метод используется для сокращения числа цветов изображения, что уменьшает размер файла (и трафик) без ущерба для качества. Он также используется при уменьшении цифровых фотографий в формате RAW в 48 или 64 бита на пиксель до RGB в 24 бита на пиксель для редактирования.

Однако уменьшение количества цветов практически всегда приводит к появлению специфических эффектов. Обычные фотографии могут иметь тысячи и даже миллионы различающихся цветов и оттенков, и преобразование их в индексированный формат с фиксированной палитрой приводит к потере огромного количества информации о цвете.

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

## *Определение пороговых цветов для битностей*

Но перед описанием самих алгоритм определим, как именно будут определяться наши пороговые (threshold) цвета для наших битностей: 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.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Порядок бит в значении цвета: | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Битность **В** | Маска изменения порядка бит |
| 8 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| 5 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 5 | 4 | 3 | 7 | 6 | 5 | |
| 2 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 6 | 7 | 6 | 7 | 6 | 7 | 6 | |
| 1 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 7 | 7 | 7 | 7 | 7 | 7 | 7 | 7 | |

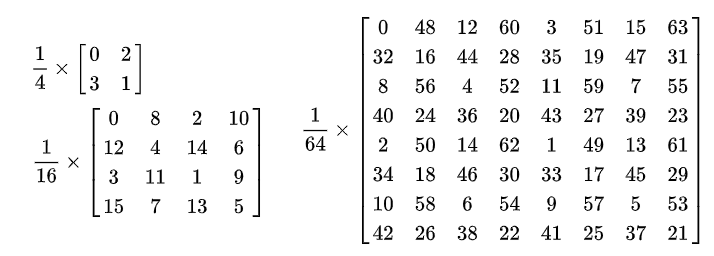
Рассмотрим пример для **В=2**

|  |  |  |  |
| --- | --- | --- | --- |
| Значение цвета в 10СС | Значение цвета в 2СС | Перестановка бит 2СС | Перестановка бит 10СС |
| 0 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **0** | 0 | 0 | 0 | 0 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 |
| 50 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **0** | 1 | 1 | 0 | 0 | 1 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | 0 |
| 100 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **0** | **1** | 1 | 0 | 0 | 1 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | | 85 |
| 150 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **0** | 0 | 1 | 0 | 1 | 1 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | | 170 |
| 200 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **1** | 0 | 0 | 1 | 0 | 0 | 0 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 255 |
| 255 | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | **1** | **1** | 1 | 1 | 1 | 1 | 1 | 1 | | |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | --- | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | | 255 |

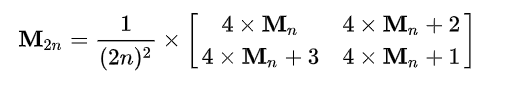
# *Ordered dithering*

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.

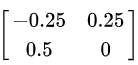
Пороговые карты **M** бывают разных размеров:



Здесь представлено 3 карты: 2х2, 4х4, 8х8. Причем, все элементы матрицы должны быть представлены в диапазоне **(0, 1]**. Для удобства, нормирующий множитель был вынесен непосредственно перед матрицей для удобства.



Предварительная работа с матрицей на примере матрицы **2х2**:

1. Пусть Mint - матрица, заполненная целыми числами [0, 2^2-1=3]
2. Для уменьшения дальнейших вычислений матрицу можно хранить, сразу поделив на нормирующий множитель: Mpre(i,j)=(Mint(i,j) + 1) / n2
3. Дополнительно, для улучшения визуального эффекта Mend(i,j)=Mpre(i,j) - 0.5. Тогда наша матрица примет следующий вид: 

После вычисления матрицы порогов (или же простого хардкода итоговой матрицы), можно переходить к алгоритму.

Алгоритм смещает для каждого пикселя его значение 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.

Алгоритм выполняет следующее преобразование для каждого цвета c каждого пикселя:

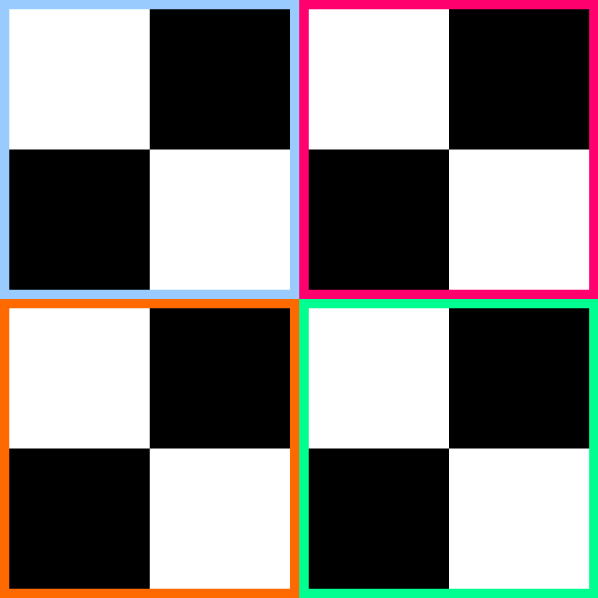
color'=findNearestPaletteColor(color + resizerM(x%n,y%n)),

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
* color' - новый цвет пикселя в текущей палитре

Значения, считанные из карты порогов, должны масштабироваться в том же диапазоне, что и color. Для этого в формуле вводится resizer.

По сути, наша квадратная матрица накладывается на изображение и для расчета значения цвета одного пикселя требуется взять соответствующий элемент матрицы. Так, на рисунке ниже, представляющем собой картинку 4х4 пикселей, продемонстрирован пример наложения матрицы 2х2 (для удобства все наложения выделены рамками разных цветов).



Пример применения дизеринга ordered 4x4 к горизонтальному градиенту:



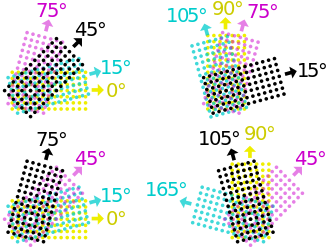
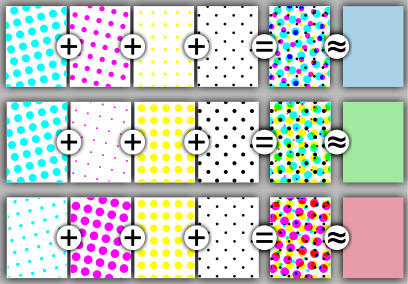
Поскольку алгоритм работает с одиночными пикселями и не имеет условных операторов, он очень быстрый и подходит для преобразований в реальном времени. Кроме того, расположение шаблонов сглаживания всегда остается одинаковым относительно кадра дисплея, что способствует улучшению сжатия изображения. Упорядоченное 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 рассеянием ошибок.

# *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).

Полутона широко используются для печати цветных изображений. Общая идея: изменяя плотность четырех вторичных цветов печати: голубого, пурпурного, желтого и черного (сокращение CMYK), можно воспроизвести любой конкретный оттенок.

В этом случае может возникнуть дополнительная проблема. В простом случае можно создать полутон, используя те же методы, что и для печати оттенков серого, но в этом случае разные цвета печати должны оставаться физически близко друг к другу, чтобы обмануть глаза и подумать, что они одного цвета. Чтобы сделать это, промышленность стандартизировала набор известных углов, которые приводят к образованию точек в виде маленьких кружков или розеток.



В случае обработки цифрового изображения halftone представляет собой матрицы порогов (для различных углов поворота), позволяющие воспроизводить это “точки” как при печать изображения.

Матрица:

{{ 7, 13, 11, 4},

{12, 16, 14, 8},

{10, 15, 6, 2},

{ 5, 9, 3, 1}}

# *Error diffusion algorithms*

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, что усложняет параллельную обработку.

Для простоты расчетов все стандартные формулы дизеринга продвигают ошибку только вперед. Если обойти изображение попиксельно, начиная с верхнего левого угла и двигаясь вправо, учитывать ошибки назад (например, влево и/или вверх) необходимости не будет.

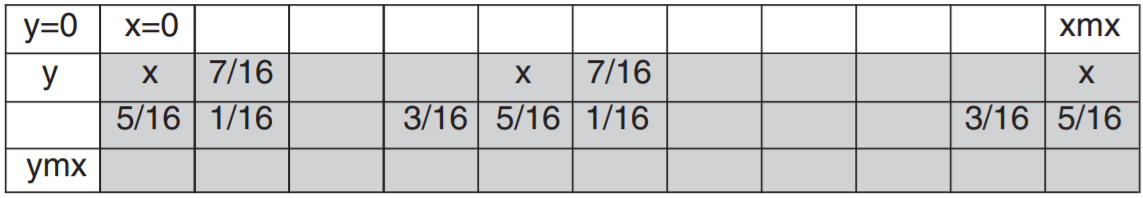
Таким образом, алгоритмы с рассеянием ошибки действуют следующим образом:

1. Обход по пикселям совершается слева направо, сверху вниз
2. При анализе текущего пикселя (color):
   1. определяется его новый цвет color' через findNearestPaletteColor
   2. рассчитывается ошибка как color-color'
   3. затем ошибка распространяется согласно используемому паттерну/алгоритму (см. далее)

# *Floyd–Steinberg*

Рассеивание ошибок происходит по следующей схеме:

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



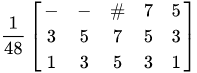
Путем распространения ошибки нескольким пикселям, каждому с различным значением, мы сводим к минимуму все отвлекающие полосы с точками, заметные в исходном примере алгоритма рассеивания ошибок. Вот изображение куба с применением алгоритма Флойда-Стейнберга:

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.

# *Jarvis, Judice, Ninke*

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



При таком алгоритме ошибка распределяется на в три раза больше пикселей, чем у Флойда-Стейнберга, что приводит к более гладкому и более тонкому результату.

# *Atkinson*

Формула Аткинсона немного отличается от других в этом списке, потому что она распространяет только часть ошибки, не всю целиком. В современных графических приложениях этот метод встречается под именем «уменьшить выцветание». Рассеивание только части ошибок помогает уменьшить зернистость, но непрерывные светлые и темные участки изображения могут потерять цвет.

Фильтр:

X 1 1

1 1 1

1

\*(1/8)

# *Sierra*

Фильтр для распространения ошибки:

X 5 3

2 4 5 4 2

2 3 2

(1/32)

# *Random dithering*

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 из диапазона (0.1].

**Экспериментальная часть**

Язык программирования: C++ 17

Данные считываются в формате, указанном в описании работы. Если данные не соответствуют формату, то выводится ошибка и программа завершает свою работу.

Я храню 3 массива: image1 – начальное изображение с цветами 0..255, image1\_ – начальное изображение или градиент с цветами 0..1, image2 – результат с цветами 0..255.

В зависимости от введенного номера делаю дизеринг. Если введен 0, то для каждого пикселя изменяю гамму, ограничиваю получившееся значение снизу 0 и сверху 255, меняю битность. Если введено 1, 2 или 7, то к исходному цвету прибавляю значение из соответствующей ячейки матрицы или рандомное число, а потом делаю все то же самое, что и для 0. Если введено 3, 4, 5, 6 – то это алгоритм с распространением ошибки, я создаю массив ошибок, прибавляю к обрабатываемому пикселю соответствующую ячейку этого массива, далее с этим пикселем делаю то же, что и для пикселя без дизеринга (значение 0) и распространяю ошибку в зависимости от фильтра.

**Выводы**

В ходе проделанной работы были получены следующие результаты:

Выделения дополнительной памяти можно было избежать.

Время работы каждого из представленных алгоритмов дизеринга - O(w \* h), где w и h – высота и ширина изображения. Сделать быстрее не получится, так как хотя бы раз мы должны пройти по всем пикселям изображения.

**Листинг**

**Название исходного файла : Source.cpp**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <iostream>

#include <algorithm>

#include <vector>

#include <string>

#include <cmath>

using namespace std;

int grad, dithering, bitness;

double gamma\_;

char symbol;

int number, h, w, max\_color;

vector<unsigned char> image1, image2;

vector<double> image1\_;

FILE \*fin, \*fout;

double gamma\_correction(double color)

{

if (gamma\_ == 0)//sRGB

{

if (color <= 0.04045)

color = color / 12.92;

else

color = pow((color + 0.055) / 1.055, 2.4);

}

else

{

color = pow(color, gamma\_);

}

return color \* 255;

}

unsigned char find\_nearest\_palette\_color(double color)

{

color = gamma\_correction(color);

return (unsigned char)min(max(color, 0.), 255.);

}

unsigned char change\_bitness(unsigned char color)

{

unsigned char new\_ = (((1u << bitness) - 1) << (8 - bitness)) & color; //11111111->11...000->color...00

color = 0;

for (int i = 0; i <= 8 / bitness; i++)

color |= (unsigned char)(new\_ >> bitness \* i);

return color;

}

unsigned char change\_bitness\_for\_random(unsigned char color)

{

unsigned char color1 = color, color2;

unsigned char new\_ = (((1u << bitness) - 1) << (8 - bitness)) & color; //11111111->11...000->color...00

unsigned char new\_\_ = new\_;

color2 = 0;

for (int i = 0; i <= 8 / bitness; i++)

color2 |= (unsigned char)(new\_ >> bitness \* i);

if ((double)rand() / (double)RAND\_MAX > 0.5)

{

if (color2 > color1)

return color2;

else

{

if ((new\_\_ >> (8 - bitness)) != (1 << (bitness)) - 1)

{

new\_ = ((new\_\_ >> (8 - bitness)) + 1) << (8 - bitness);

color2 = 0;

for (int i = 0; i <= 8 / bitness; i++)

color2 |= (unsigned char)(new\_ >> bitness \* i);

}

return color2;

}

}

else

{

if (color2 > color1)

return color1;

else

{

if ((int)new\_\_ != 0)

{

new\_ = ((new\_\_ >> (8 - bitness)) - 1) << (8 - bitness);

color2 = 0;

for (int i = 0; i <= 8 / bitness; i++)

color2 |= (unsigned char)(new\_ >> bitness \* i);

}

return color2;

}

}

}

void dithering\_no()

{

//if(bitness != 8)

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

image2[y \* w + x] = change\_bitness(find\_nearest\_palette\_color(image1\_[y \* w + x]));

}

void dithering\_ordered()

{

const int n = 8;

double m[n][n] = {

{ 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 (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

image2[y \* w + x] = change\_bitness(

find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + ((m[x % n][y % n] + 1) / 64 - 0.5))

)

);

}

void dithering\_random()

{

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

{

image2[y \* w + x] = change\_bitness(

find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + 2 \* ((double)rand() / (double)RAND\_MAX - 0.5) \* ((1 << (8 - bitness)) \* 1. / 256.))

)

);

// image1\_[y \* w + x] += ((double)rand() / (double)RAND\_MAX - 0.5) \* ((1 << (8 - bitness)) \* 1. / 256.);

// image1\_[y \* w + x] = min(max(image1\_[y \* w + x], 0.), 1.);

// image1\_[y \* w + x] = change\_bitness\_for\_random(255 \* image1\_[y \* w + x]) / 255.;

// image2[y \* w + x] = find\_nearest\_palette\_color(image1\_[y \* w + x]);

// image1\_[y \* w + x] = gamma\_correction(image1\_[y \* w + x]);

// if((double)rand() / (double)RAND\_MAX > 0.5)

// image1\_[y \* w + x] = ceil(image1\_[y \* w + x]);

// else

// image1\_[y \* w + x] = floor(image1\_[y \* w + x]);

// image1\_[y \* w + x] = (unsigned char)min(max(image1\_[y \* w + x], 0.), 255.);

// image2[y \* w + x] = change\_bitness(image1\_[y \* w + x]);

}

}

void dithering\_floyd\_steinberg()

{

// X 7

//3 5 1

//\*(1/16)

vector<double> error;

error.resize(h \* w, 0);

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

{

unsigned char tmp = find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + error[y \* w + x])

);

image2[y \* w + x] = change\_bitness(tmp);

double quant\_error = (tmp - image2[y \* w + x]) / 255.;

if (x < w - 1)

error[y \* w + x + 1] += quant\_error \* 7. / 16.;

if (y < h - 1)

{

if (x < w - 1)

error[(y + 1) \* w + x + 1] += quant\_error / 16.;

error[(y + 1) \* w + x] += quant\_error \* 5. / 16.;

if (x > 0)

error[(y + 1) \* w + x - 1] += quant\_error \* 3. / 16.;

}

}

}

void dithering\_jarvis\_judice\_ninke()

{

// X 7 5

//3 5 7 5 3

//1 3 5 3 1

// (1/48)

vector<double> error;

error.resize(h \* w, 0);

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

{

unsigned char tmp = find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + error[y \* w + x])

);

image2[y \* w + x] = change\_bitness(tmp);

double quant\_error = (tmp - image2[y \* w + x]) / 255.;

if (x < w - 1)

{

if (x < w - 2)

error[(y)\* w + x + 2] += quant\_error \* 5. / 48.;

error[(y)\* w + x + 1] += quant\_error \* 7. / 48.;

}

if (y < h - 1)

{

if (x < w - 2)

error[(y + 1) \* w + x + 2] += quant\_error \* 3. / 48.;

if (x < w - 1)

error[(y + 1) \* w + x + 1] += quant\_error \* 5. / 48.;

error[(y + 1) \* w + x] += quant\_error \* 7. / 48.;

if (x > 0)

error[(y + 1) \* w + x - 1] += quant\_error \* 5. / 48.;

if (x > 1)

error[(y + 1) \* w + x - 2] += quant\_error \* 3. / 48.;

if (y < h - 2)

{

if (x < w - 2)

error[(y + 2) \* w + x + 2] += quant\_error / 48.;

if (x < w - 1)

error[(y + 2) \* w + x + 1] += quant\_error \* 3. / 48.;

error[(y + 2) \* w + x] += quant\_error \* 5. / 48.;

if (x > 0)

error[(y + 2) \* w + x - 1] += quant\_error \* 3. / 48.;

if (x > 1)

error[(y + 2) \* w + x - 2] += quant\_error / 48.;

}

}

}

}

void dithering\_sierra()

{

// X 5 3

//2 4 5 4 2

// 2 3 2

// (1/32)

vector<double> error;

error.resize(h \* w, 0);

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

{

unsigned char tmp = find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + error[y \* w + x])

);

image2[y \* w + x] = change\_bitness(tmp);

double quant\_error = (tmp - image2[y \* w + x]) / 255.;

if (x < w - 1)

{

if (x < w - 2)

error[(y)\* w + x + 2] += quant\_error \* 3. / 32.;

error[(y)\* w + x + 1] += quant\_error \* 5. / 32.;

}

if (y < h - 1)

{

if (x < w - 2)

error[(y + 1) \* w + x + 2] += quant\_error \* 2. / 32.;

if (x < w - 1)

error[(y + 1) \* w + x + 1] += quant\_error \* 4. / 32.;

error[(y + 1) \* w + x] += quant\_error \* 5 / 32.;

if (x > 0)

error[(y + 1) \* w + x - 1] += quant\_error \* 4. / 32.;

if (x > 1)

error[(y + 1) \* w + x - 2] += quant\_error \* 2. / 32.;

if (y < h - 2)

{

if (x < w - 1)

error[(y + 2) \* w + x + 1] += quant\_error \* 2. / 32.;

error[(y + 2) \* w + x] += quant\_error \* 3. / 32.;

if (x > 0)

error[(y + 2) \* w + x - 1] += quant\_error \* 2. / 32.;

}

}

}

}

void dithering\_atkinson()

{

// X 1 1

//1 1 1

// 1

// (1/8)

vector<double> error;

error.resize(h \* w, 0);

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

{

unsigned char tmp = find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + error[y \* w + x])

);

image2[y \* w + x] = change\_bitness(tmp);

double quant\_error = (tmp - image2[y \* w + x]) / 255.;

if (x < w - 1)

{

if (x < w - 2)

error[(y)\* w + x + 2] += quant\_error / 8.;

error[(y)\* w + x + 1] += quant\_error / 8.;

}

if (y < h - 1)

{

if (x < w - 1)

error[(y + 1) \* w + x + 1] += quant\_error / 8.;

error[(y + 1) \* w + x] += quant\_error / 8.;

if (x > 0)

error[(y + 1) \* w + x - 1] += quant\_error / 8.;

if (y < h - 2)

error[(y + 2) \* w + x] += quant\_error / 8.;

}

}

}

void dithering\_halftone()

{

const int n = 4;

double m[n][n] = {

{ 7, 13, 11, 4},

{12, 16, 14, 8},

{10, 15, 6, 2},

{ 5, 9, 3, 1},

};

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

image2[y \* w + x] = change\_bitness(

find\_nearest\_palette\_color

(

(image1\_[y \* w + x] + ((m[x % n][y % n]) - 8.5) / 16)

)

);

}

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

{

//printf("PROCESSING\n");

//program.exe <имя\_входного\_файла> <имя\_выходного\_файла> <градиент> <дизеринг> <битность> <гамма>

if (argc == 7)

{

try

{

grad = stoi(argv[3]);

if (grad != 0 && grad != 1)

{

cerr << "Gradient != 1 and gradient != 0";

return 1;

}

dithering = stoi(argv[4]);

if (dithering < 0 || dithering > 7)

{

cerr << "Dithering should be in [0..7]";

return 1;

}

bitness = stoi(argv[5]);

if (bitness < 1 || bitness > 8)

{

cerr << "Bit should be in [1..8]";

return 1;

}

gamma\_ = stod(argv[6]);

if (gamma\_ < 0)

{

cerr << "Gamma should be >= 0";

return 1;

}

}

catch (...)

{

cerr << "Invalid arguments";

return 1;

}

}

else

{

cerr << "Invalid number of arguments";

return 1;

}

fin = fopen(argv[1], "rb");

if (fin == NULL)

{

cerr << "Input file can't be open";

return 1;

}

int quantity = fscanf(fin, "%c%d\n%d%d\n%d\n", &symbol, &number, &w, &h, &max\_color);

if (quantity != 5 || symbol != 'P' || number != 5 || h <= 0 || w <= 0 || max\_color <= 0 || max\_color > 255)

{

fclose(fin);

cerr << "Invalid pnm file";

return 1;

}

if (grad == 0)

{

image1.resize(h \* w, 0);

image1\_.resize(h \* w, 0.);

image2.resize(h \* w, 0);

quantity = fread(&image1[0], sizeof(unsigned char), image1.size(), fin);

if (quantity != h \* w)

{

fclose(fin);

cerr << "Quantity != width \* height";

return 1;

}

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

image1\_[y \* w + x] = (double)image1[y \* w + x] / 255.;

}

else

{

image1.resize(h \* w, 0);

image1\_.resize(h \* w, 0.);

image2.resize(h \* w, 0);

//рисую градиент

for (int y = 0; y < h; ++y)

for (int x = 0; x < w; ++x)

image1\_[y \* w + x] = (double)x / (double)w \* 256. / 255.;

}

fclose(fin);

switch (dithering)

{

case 0:

dithering\_no();

break;

case 1:

dithering\_ordered();

break;

case 2:

dithering\_random();

break;

case 3:

dithering\_floyd\_steinberg();

break;

case 4:

dithering\_jarvis\_judice\_ninke();

break;

case 5:

dithering\_sierra();

break;

case 6:

dithering\_atkinson();

break;

case 7:

dithering\_halftone();

break;

default:

cerr << "Dithering should be in [0..7]";

return 1;

}

fout = fopen(argv[2], "wb");

if (fout == NULL)

{

cerr << "Output file can't be open";

return 1;

}

fprintf(fout, "%c%d\n%d %d\n%d\n", symbol, number, w, h, max\_color);

quantity = fwrite(&image2[0], sizeof(unsigned char), image2.size(), fout);

fclose(fout);

if (quantity != h \* w)

{

cerr << "Quantity of outputfile != width \* height";

return 1;

}

//printf("DONE");

//system("pause");

return 0;

}