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

Компьютерная графика и геометрия

Отчет

по лабораторной работе №3

Изучение алгоритмов псевдотонирования изображений

Выполнил: студент гр. М3101
Ершов Михаил Юрьевич
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

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

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».
Определение пороговых цветов для битностей заключается в 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.

Виды дизеринга:

1. **No dithering**
   Данный алгоритм подразумевает простое округление всех цветов до ближайших пороговых.
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 с каждого пикселя:
   color'=findNearestPaletteColor(color + resizerM(x%n,y%n)),

   где:

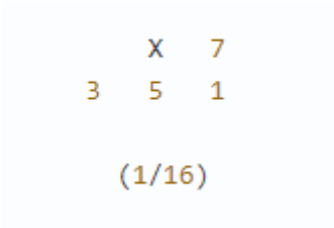
   - color - старый цвет пикселя
   - M(x%n,y%n) - элемент карты порогов
   - findNearestPaletteColor - функция, возвращающая ближайший цвет к подаваемому, который можно отобразить на текущей палитре
   - color' - новый цвет пикселя в текущей палитре

   Значения, считанные из карты порогов, должны масштабироваться в том же диапазоне, что и color. Для этого в формуле вводится resizer.
   Матрица M, которая использовалась в данной работе:

$$\frac{1}{64} \times \begin{bmatrix} 0 & 48 & 12 & 60 & 3 & 51 & 15 & 63 \\ 32 & 16 & 44 & 28 & 35 & 19 & 47 & 31 \\ 8 & 56 & 4 & 52 & 11 & 59 & 7 & 55 \\ 40 & 24 & 36 & 20 & 43 & 27 & 39 & 23 \\ 2 & 50 & 14 & 62 & 1 & 49 & 13 & 61 \\ 34 & 18 & 46 & 30 & 33 & 17 & 45 & 29 \\ 10 & 58 & 6 & 54 & 9 & 57 & 5 & 53 \\ 42 & 26 & 38 & 22 & 41 & 25 & 37 & 21 \end{bmatrix}$$

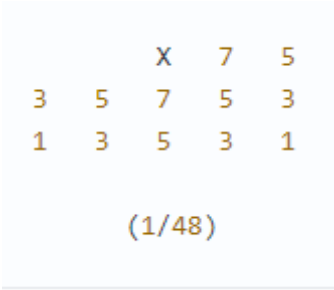
3. **Random**
   Аналогичен методу Ordered, однако вместо элементов матрицы M каждый раз выбирается случайное значение из диапазона (0; 1]
4. **Floyd-Steinberg**

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

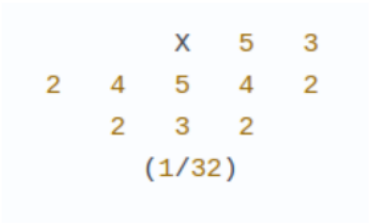


Где X – текущий рассматриваемый пиксель. Для него вычисляется значение ошибки, равное разнице между полученным в результате дизеринга значением и исходным, после чего полученная ошибка распространяется в соседние пиксели с коэффициентами, как на изображении выше, каждый из коэффициентов дополнительно делится на 16, о чём говорит запись (1/16).

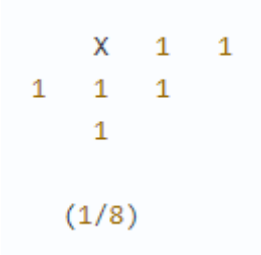
5. **Jarvis, Judice, Ninke**
   В год, когда Флойд и Стейнберг опубликовали свой знаменитый алгоритм дизеринга, был издан менее известный, но гораздо более мощный алгоритм. Фильтр Джарвиса, Джудиса и Нинке значительно сложнее, чем Флойда-Стейнберга. Ниже указаны коэффициенты, с которыми распространяется ошибка из пикселя X, в соседние.



6. **Sierra-3**
   Точно так же аналогичен алгоритму Флойда-Стейнберга, но с другими коэффициентами:



7. **Atkinson**
   Тоже аналогичен алгоритму Флойда-Стейнберга, но с другими коэффициентами:



8. **Halftone(4x4, orthogonal)**
   Полутонирование - создание изображения со многими уровнями серого или цвета (т.е. слитный тон) на аппарате с меньшим количеством тонов, обычно чёрно-белый принтер.
   Аналогично алгоритму Ordered происходит умножение значения пикселей на значение таблицы:

$$\begin{matrix} 7 & 13 & 11 & 4 \\ 12 & 16 & 14 & 8 \\ 10 & 15 & 6 & 2 \\ 5 & 9 & 3 & 1 \end{matrix}$$

   Каждое значение матрицы также должно быть разделено на 17.

Экспериментальная часть

Язык программирования: C++14

Для начала происходит считывание изображение из файла, так как было реализовано частичное решение, то гамма-коррекция не учитывалась. После чего вызовом метода dither к изображению применяется один из методов дизеринга, приведённых выше, в зависимости от введённых данных. Затем изображение выводится в файл, указанный в введённых данных.

Выводы

Выполнение данной работы позволило изучить основные алгоритмы псевдотонирования изображений, как упорядоченные, так и с рассеиванием ошибки. Были реализованы следующие алгоритмы:

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

Листинг

main.cpp

```
#include <iostream>
#include <string>
#include <cstdlib>
#include "pgm.h"

using namespace std;

int main(int argc, char* argv[]) {
    if (argc != 7) {
        cerr << "Invalid number of arguments\n";
        return 1;
    }
    try {
        bool gradient = atoi(argv[3]);
        int typeOfDithering = atoi(argv[4]);
        int bit = atoi(argv[5]);
        double gamma = stod(argv[6]);
        PGM image(argv[1], gradient, gamma);
        image.dither(bit, typeOfDithering, gamma);
        image.print(argv[2], bit, gradient, gamma);
    }
    catch (const exception& e) {
        std::cerr << e.what();
        return 1;
    }

    return 0;
}
```

pgm.h

```
#ifndef LAB3_PGM_H
#define LAB3_PGM_H

#include <vector>
#include <cstdio>
#include <cstring>

typedef unsigned char uchar;

class PGM {
private:
    char header[2];
    int width, height;
    uchar maxValue;
    uchar* data;
    double* errors;
public:
    explicit PGM(char* fileName, bool gradient, double gamma = 0);
    ~PGM();
    void print(char* fileName, int bit, bool gradient, double gamma = 0);
    void dither(int bit, int typeOfDithering, double gamma = 0);
};

#endif //LAB3_PGM_H
```

pgm.cpp

```cpp
#include "pgm.h"
#include <stdexcept>
#include <set>
#include <iostream>
#include <random>
#include <cmath>

PGM::PGM(char* fileName, bool gradient, double gamma) {
    FILE* fin = fopen(fileName, "rb");
    if (fin == nullptr) {
        throw std::runtime_error("File can't be opened\n");
    }
    int tmp;
    if (fscanf(fin, "%s%d%d%d", header, &width, &height, &tmp) < 4) {
        throw std::runtime_error("Invalid header\n");
    }
    if (strcmp(header, "P5") != 0) {
        throw std::runtime_error("File is not a PGM image\n");
    }
    if (tmp > 255) {
        throw std::runtime_error("Unsupported colours\n");
    }
    maxValue = (uchar)tmp;
    if (width < 0 || height < 0) {
        throw std::runtime_error("Invalid header\n");
    }
    if (fgetc(fin) == EOF) {
        throw std::runtime_error("Invalid header\n");
    }
    data = new uchar[width * height];
    errors = new double[width * height];
    for (int i = 0; i < width * height; i++) {
        errors[i] = 0;
    }
    if (!gradient) {
        fread(data, 1, width * height, fin);
        for (int i = 0; i < width * height; i++) {
            double relativeBrightness = (double)data[i] / maxValue;
            if (gamma == 0) {
                relativeBrightness = relativeBrightness <= 0.04045 ?
                    relativeBrightness / 12.92 :
                    std::pow((relativeBrightness + 0.055) / 1.055, 2.4);
            }
            else {
                relativeBrightness = std::pow(relativeBrightness, gamma);
            }
            data[i] = maxValue * relativeBrightness;
        }
    }
    else {
        for (int i = 0; i < height * width; i++) {
            data[i] = (i % width) * 256 / width;
        }
    }
    if (fclose(fin) != 0) {
        throw std::runtime_error("File can't be closed\n");
    }
}

PGM::~PGM() {
    delete[] data;
    delete[] errors;
}
```

```
void PGM::print(char* fileName, int bit, bool gradient, double gamma) {
    FILE* fout = fopen(fileName, "wb");
    if (fout == nullptr) {
        throw std::runtime_error("File can't be opened or created\n");
    }
    if (!gradient) {
        for (int i = 0; i < width * height; i++) {
            double relativeBrightness = (double)data[i] / maxValue;
            if (gamma == 0) {
                relativeBrightness = relativeBrightness <= 0.0031308 ?
                    relativeBrightness * 12.92 :
                    std::pow(1.055 * relativeBrightness, 1 / 2.4) - 0.055;
            }
            else {
                relativeBrightness = std::pow(relativeBrightness, 1 / gamma);
            }
            data[i] = maxValue * relativeBrightness;
        }
    }
    fprintf(fout, "%s\n%i %i\n%i\n", header, width, height, maxValue);
    fwrite(data, 1, width * height, fout);
    fclose(fout);
}
int BayerMatrix[8][8] = {
        {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 HalftoneMatrix[4][4] = {
        {7, 13, 11, 4},
        {12, 16, 14, 8},
        {10, 15, 6, 2},
        {5, 9, 3, 1}
};

uchar nearestColor(const uchar& color, const unsigned int& bit) {
    uchar borderColor = ((1u << bit) - 1);
    borderColor &= color >> (8u - bit);
    uchar additionalColor = borderColor >> (bit - (8 % bit));;
    uchar newColor = 0;
    for (int i = 0; i < 8 / bit; i++) {
        newColor += borderColor << i * bit;
    }
    return (newColor << (8 % bit)) + additionalColor;
}

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

void PGM::dither(int bit, int typeOfDithering, double gamma) {
    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(0.0, 1.0);
```

```
switch (typeOfDithering) {
case 0:
    for (int i = 0; i < width * height; i++) {
        data[i] = nearestColor(data[i], bit);
    }
    break;
case 1:
    for (int i = 0; i < width * height; i++) {
        int row = i / width;
        int col = i % width;
        double tmp = (double)BayerMatrix[row % 8][col % 8] / 64 - 0.5;
        double color = (double)data[i] / 255.0;
        data[i] = nearestColor(sum(color, tmp) * 255, bit);
    }
    break;
case 2:
    for (int i = 0; i < width * height; i++) {
        double tmp = dist(mt) - 0.5;
        double color = (double)data[i] / 255.0;
        data[i] = nearestColor(sum(color, tmp) * 255, bit);
    }
    break;
case 3:
    for (int i = 0; i < width * height; i++) {
        double color = (double)data[i] / 255.;
        color = sum(color, errors[i]);
        data[i] = nearestColor(color * 255, bit);
        double newColor = (double)data[i] / 255.0;
        double error = (color - newColor) / 16.0;
        int row = i / width;
        int col = i % width;
        if (col + 1 < width) {
            if (row + 1 < height) {
                errors[(row + 1) * width + col + 1] += error * 1.0;
            }
            errors[row * width + col + 1] += error * 7.0;
        }
        if (row + 1 < height) {
            if (col - 1 >= 0) {
                errors[(row + 1) * width + col - 1] += error * 3.0;
            }
            errors[(row + 1) * width + col] += error * 5.0;
        }
    }
    break;
case 4:
    for (int i = 0; i < width * height; i++) {
        double color = (double)data[i] / 255.;
        color = sum(color, errors[i]);
        data[i] = nearestColor(color * 255, bit);
        double newColor = (double)data[i] / 255.0;
        double error = (color - newColor) / 48.0;
        int row = i / width;
        int col = i % width;
        if (row + 1 < height) {
            if (row + 2 < height) {
                errors[(row + 2) * width + col] += error * 5.0;
            }
            errors[(row + 1) * width + col] += error * 7.0;
        }
        if (col + 1 < width) {
            if (col + 2 < width) {
                if (row + 1 < height) {
                    if (row + 2 < height) {
                        errors[(row + 2) * width + col + 2] += error * 1.0;
```

```
                    }
                    errors[(row + 1) * width + col + 2] += error * 3.0;
                }
                errors[row * width + col + 2] += error * 5.0;
            }
            if (row + 1 < height) {
                if (row + 2 < height) {
                    errors[(row + 2) * width + col + 1] += error * 3.0;
                }
                errors[(row + 1) * width + col + 1] += error * 5.0;
            }
            errors[row * width + col + 1] += error * 7.0;
        }
        if (col - 1 >= 0) {
            if (col - 2 >= 0) {
                if (row + 1 < height) {
                    if (row + 2 < height) {
                        errors[(row + 2) * width + col - 2] += error * 1.0;
                    }
                    errors[(row + 1) * width + col - 2] += error * 3.0;
                }
            }
            if (row + 1 < height) {
                if (row + 2 < height) {
                    errors[(row + 2) * width + col - 1] += error * 3.0;
                }
                errors[(row + 1) * width + col - 1] += error * 5.0;
            }
        }
    }
    break;
case 5:
    for (int i = 0; i < width * height; i++) {
        double color = (double)data[i] / 255.;
        color = sum(color, errors[i]);
        data[i] = nearestColor(color * 255, bit);
        double newColor = (double)data[i] / 255.0;
        double error = (color - newColor) / 32.0;
        int row = i / width;
        int col = i % width;
        if (row + 1 < height) {
            if (row + 2 < height) {
                errors[(row + 2) * width + col] += error * 3.0;
            }
            errors[(row + 1) * width + col] += error * 5.0;
        }
        if (col + 1 < width) {
            if (col + 2 < width) {
                if (row + 1 < height) {
                    errors[(row + 1) * width + col + 2] += error * 2.0;
                }
                errors[row * width + col + 2] += error * 3.0;
            }
            if (row + 1 < height) {
                if (row + 2 < height) {
                    errors[(row + 2) * width + col + 1] += error * 2.0;
                }
                errors[(row + 1) * width + col + 1] += error * 4.0;
            }
            errors[row * width + col + 1] += error * 5.0;
        }
        if (col - 1 >= 0) {
            if (col - 2 >= 0) {
                if (row + 1 < height) {
                    errors[(row + 1) * width + col - 2] += error * 2.0;
```

```
                    }
                }
                if (row + 1 < height) {
                    if (row + 2 < height) {
                        errors[(row + 2) * width + col - 1] += error * 2.0;
                    }
                    errors[(row + 1) * width + col - 1] += error * 4.0;
                }
            }
        }
        break;
    case 6:
        for (int i = 0; i < width * height; i++) {
            double color = (double)data[i] / 255.;
            color = sum(color, errors[i]);
            data[i] = nearestColor(color * 255, bit);
            double newColor = (double)data[i] / 255.0;
            double error = (color - newColor) / 8.0;
            int row = i / width;
            int col = i % width;
            if (row + 1 < height) {
                if (row + 2 < height) {
                    errors[(row + 2) * width + col] += error;
                }
                errors[(row + 1) * width + col] += error;
            }
            if (col + 1 < width) {
                if (col + 2 < width) {
                    errors[row * width + col + 2] += error;
                }
                if (row + 1 < height) {
                    errors[(row + 1) * width + col + 1] += error;
                }
                errors[row * width + col + 1] += error;
            }
            if (col - 1 >= 0) {
                if (row + 1 < height) {
                    errors[(row + 1) * width + col - 1] += error;
                }
            }
        }
        break;
    case 7:
        for (int i = 0; i < height * width; i++) {
            int row = i / width;
            int col = i % width;
            double tmp = (double)1.0 * HalftoneMatrix[row % 4][col % 4] / 17.0 - 0.5;
            double color = (double)data[i] / 255.0;
            data[i] = nearestColor(sum(color, tmp) * 255, bit);
        }
        break;
    default: throw std::runtime_error("Invalid type of dithering!\n");
    }
}
```