МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНО
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
УНИВЕРСИТЕТ
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»
Факультет информационных технологий и программирования

Дисциплина: «Компьютерная графика»

Лабораторная работа № 3

Изучение алгоритмов псевдотонирования изображений

Выполнил: студент группы № M32342
Кубанцев Ярослав Максимович
Проверил: к.т.н., ассистент факультета
прикладной оптики
Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ
2020

Содержание

Цель работы

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

Теоретическая справка

Дизеринг — при обработке цифровых сигналов представляет собой подмешивание в первичный сигнал псевдослучайного шума со специально подобранным спектром.

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.

Среди алгоритмов дизеринга выделяют алгоритмы рассеивания ошибки. Их суть заключается в том, чтобы распределить *ошибку* – разность между оригинальным значением и новым – конкретного пикселя между его соседними.

Решение

Random

Самый простой из алгоритмов. В каждый пиксель добавляется случай шум, что в общей картине дает улучшение изображения.

Ordered

В отличии от случайного алгоритма, вносимый шум имеет заданную структуру и берется из специальной матрицы.

Floyd–Steinberg; Jarvis, Judice, Ninke; Sierra (Sierra-3); Atkinson;

В алгоритмах рассеивания ошибки все пиксели обрабатываются последовательно. Ошибка текущего пикселя распределяется между следующими (соседними) пикселями. Схема рассеивания и пропорции определяются конкретным алгоритмом.

Halftone

Суть алгоритма заключается в том, чтобы разбить изображение на зоны заданной площади, для каждой зоны подсчитать сумму значений пикселей внутри её и сконцентрировать это значение в виде круга.

Листинг программ

Dither.h

```
#ifndef HW3_DITHER_H
#define HW3_DITHER_H

#include <vector>
#include "Picture.h"

class Dither {
private:
    typedef unsigned char uchar;
    const static int orderedMatrix[8][8];

    static Picture basicDither(Picture const &oldPicture,
                               double (*ditherPixel)(double a, size_t i,
size_t j));

    static double orderedDP(double a, size_t i, size_t j);

    static double randomDP(double a, size_t i, size_t j);

    static double roundPixel(uchar pixel, int bitness);

    static void
    addError(std::vector<uchar> &data, int weight, int height, int error, int
sum, int value, int i, int j, int a,
             int b);

    static Picture errorDiffusionDither(Picture const &oldPicture, int
bitness,
                                        void (*setError)(std::vector<uchar>
&, int, int, int, int, int));

    static void floydSteinbergEDD(std::vector<uchar> &data, int weight, int
height, int error, int i, int j);

    static void jjnEDD(std::vector<uchar> &data, int weight, int height, int
error, int i, int j);

    static void sierra3EDD(std::vector<uchar> &data, int weight, int height,
int error, int i, int j);
```

```
    static void atkinsonEDD(std::vector<uchar> &data, int weight, int height,
int error, int i, int j);


    static void
    setBrightness(GammaCorrection const &gc, std::vector<uchar> &data, int
weight, int height, double value, int i,
                  int j);


public:
    static Picture round(Picture const &oldPicture, int bitness);


    static Picture ordered(Picture const &oldPicture);


    static Picture random(Picture const &oldPicture);


    static Picture floydSteinberg(Picture const &oldPicture, int bitness);


    static Picture jjn(Picture const &oldPicture, int bitness);


    static Picture sierra3(Picture const &oldPicture, int bitness);


    static Picture atkinson(Picture const &oldPicture, int bitness);


    static Picture halftone(Picture const &oldPicture);
};



#endif //HW3_DITHER_H
```

Dither.cpp

```
#include <cmath>

#include "Dither.h"

#include "Picture.h"


constexpr const int Dither::orderedMatrix[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}};

Picture Dither::ordered(Picture const &oldPicture) {
    return basicDither(oldPicture, orderedDP);
}

Picture Dither::round(Picture const &oldPicture, int bitness) {
    std::vector<uchar> newData(oldPicture.data);
    for (uchar &a:newData) {
        a = roundPixel(a, bitness);
    }
    return Picture(newData, oldPicture.getWeight(), oldPicture.getHeight(),
oldPicture.getGammaCorrection());
}

Picture Dither::basicDither(Picture const &oldPicture, double
(*ditherPixel)(double, size_t, size_t)) {
    std::vector<uchar> newData(oldPicture.data);
    GammaCorrection const &gc = oldPicture.getGammaCorrection();
    int weight = oldPicture.getWeight();
    int height = oldPicture.getHeight();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < weight; j++) {
```

```
            double a = gc.encode(newData[i * weight + j]);
            a = ditherPixel(a, i, j);
            a = std::min(255.0, std::max(0.0, a));
            a = gc.decode(a);
            newData[i * weight + j] = a;
        }
    }
    return Picture(newData, weight, height, gc);
}

double Dither::orderedDP(double a, size_t i, size_t j) {
    double r = 255.0 / 8.0;
    return (a + r * ((orderedMatrix[i % 8][j % 8] + 1) / 64.0 - 0.5));
}

Picture Dither::random(Picture const &oldPicture) {
    return basicDither(oldPicture, randomDP);
}

double Dither::randomDP(double a, size_t i, size_t j) {
    double r = 255.0 / 8.0;
    return a + r * (((double) rand() + 1) / RAND_MAX - 0.5);
}

double Dither::roundPixel(Dither::uchar pixel, int bitness) {
    if (bitness >= 4) {
        pixel = ((pixel >> (8 - bitness)) << (8 - bitness));

    }
    if (bitness == 1) {
        pixel = ((uchar) -1) * (pixel >> 7);
```

```
    }
    if (bitness == 2) {
        pixel = pixel >> 6;
        pixel = (pixel << 6) + (pixel << 4) + (pixel << 2) + pixel;

    }
    if (bitness == 3) {
        pixel = pixel >> 5;
        pixel = (pixel << 5) + (pixel << 2) + (pixel >> 1);
    }
    return pixel;
}

Picture Dither::floydSteinberg(Picture const &oldPicture, int bitness) {
    return errorDiffusionDither(oldPicture, bitness, floydSteinbergEDD);
}

Picture Dither::jjn(Picture const &oldPicture, int bitness) {
    return errorDiffusionDither(oldPicture, bitness, floydSteinbergEDD);
}

void
Dither::addError(std::vector<uchar> &data, int weight, int height, int error,
int sum, int value, int i, int j, int a,
                 int b) {
    if (i + a >= 0 && j + b >= 0 && i + a < height && j + b < weight) {
        data[(i + a) * weight + j + b] += error * value / sum;
    }
}

Picture Dither::sierra3(Picture const &oldPicture, int bitness) {
```

```
    return errorDiffusionDither(oldPicture, bitness, sierra3EDD);
}

Picture
Dither::errorDiffusionDither(Picture const &oldPicture, int bitness,
                             void (*setError)(std::vector<Dither::uchar> &,
int, int, int, int, int)) {
    std::vector<uchar> newData(oldPicture.data);
    GammaCorrection const &gc = oldPicture.getGammaCorrection();
    int weight = oldPicture.getWeight();
    int height = oldPicture.getHeight();
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < weight; j++) {
            uchar a = newData[i * weight + j];
            uchar b = roundPixel(a, bitness);
            int error = ((int) a) - b;
            setError(newData, weight, height, error, i, j);
            newData[i * weight + j] = b;
        }
    }
    return Picture(newData, weight, height, gc);
}

void Dither::floydSteinbergEDD(std::vector<uchar> &data, int weight, int
height, int error, int i, int j) {
    addError(data, weight, height, error, 16, 7, i, j, 0, 1);
    addError(data, weight, height, error, 16, 3, i, j, 1, -1);
    addError(data, weight, height, error, 16, 5, i, j, 1, 0);
    addError(data, weight, height, error, 16, 1, i, j, 1, 1);
}
```

```
void Dither::jjnEDD(std::vector<uchar> &data, int weight, int height, int
error, int i, int j) {
    addError(data, weight, height, error, 48, 7, i, j, 0, 1);
    addError(data, weight, height, error, 48, 5, i, j, 0, 2);

    addError(data, weight, height, error, 48, 3, i, j, 1, -2);
    addError(data, weight, height, error, 48, 5, i, j, 1, -1);
    addError(data, weight, height, error, 48, 7, i, j, 1, 0);
    addError(data, weight, height, error, 48, 5, i, j, 1, 1);
    addError(data, weight, height, error, 48, 3, i, j, 1, 2);

    addError(data, weight, height, error, 48, 1, i, j, 2, -2);
    addError(data, weight, height, error, 48, 3, i, j, 2, -1);
    addError(data, weight, height, error, 48, 5, i, j, 2, 0);
    addError(data, weight, height, error, 48, 3, i, j, 2, 1);
    addError(data, weight, height, error, 48, 1, i, j, 2, 2);
}

void Dither::sierra3EDD(std::vector<uchar> &data, int weight, int height, int
error, int i, int j) {
    addError(data, weight, height, error, 32, 5, i, j, 0, 1);
    addError(data, weight, height, error, 32, 3, i, j, 0, 2);

    addError(data, weight, height, error, 32, 2, i, j, 1, -2);
    addError(data, weight, height, error, 32, 4, i, j, 1, -1);
    addError(data, weight, height, error, 32, 5, i, j, 1, 0);
    addError(data, weight, height, error, 32, 4, i, j, 1, 1);
    addError(data, weight, height, error, 32, 2, i, j, 1, 2);

    addError(data, weight, height, error, 32, 2, i, j, 2, -1);
    addError(data, weight, height, error, 32, 3, i, j, 2, 0);
    addError(data, weight, height, error, 32, 2, i, j, 2, 1);
```

```
}

void Dither::atkinsonEDD(std::vector<uchar> &data, int weight, int height,
int error, int i, int j) {
    addError(data, weight, height, error, 8, 1, i, j, 0, 1);
    addError(data, weight, height, error, 8, 1, i, j, 0, 2);

    addError(data, weight, height, error, 8, 1, i, j, 1, -1);
    addError(data, weight, height, error, 8, 1, i, j, 1, 0);
    addError(data, weight, height, error, 8, 1, i, j, 1, 1);

    addError(data, weight, height, error, 8, 1, i, j, 2, 0);
}

Picture Dither::atkinson(Picture const &oldPicture, int bitness) {
    return errorDiffusionDither(oldPicture, bitness, atkinsonEDD);
}

Picture Dither::halftone(Picture const &oldPicture) {
    std::vector<uchar> newData(oldPicture.data);
    GammaCorrection const &gc = oldPicture.getGammaCorrection();
    int weight = oldPicture.getWeight();
    int height = oldPicture.getHeight();
    for (int i = 0; i < height; i += 4) {
        for (int j = 0; j < weight; j += 4) {
            int i1 = std::min(i + 4, height);
            int j1 = std::min(j + 4, height);
            double a = 0;
            double b = (i1 - i) * (j1 - j) * 255;
            for (int ii = i; ii < i1; ii++) {
                for (int jj = j; jj < j1; jj++) {
                    a += newData[ii * weight + jj];
```

```
                }
            }
            a = a / b;
            if (a <= 0.25) {
                for (int ii = i; ii < i1; ii++) {
                    for (int jj = j; jj < j1; jj++) {
                        newData[ii * weight + jj] = 0;
                    }
                }
                for (int ii = i + 1; ii < i + 3; ii++) {
                    for (int jj = j + 1; jj < j + 3; jj++) {
                        if (ii >= height || jj >= weight) {
                            continue;
                        }
                        newData[ii * weight + jj] = a * 4;
                    }
                }
            } else {
                setBrightness(gc, newData, weight, height, 1, i + 1, j + 1);
                setBrightness(gc, newData, weight, height, 1, i + 1, j + 2);
                setBrightness(gc, newData, weight, height, 1, i + 2, j + 1);
                setBrightness(gc, newData, weight, height, 1, i + 2, j + 2);

                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1,
i, j + 1);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1,
i, j + 2);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 1, j);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 1, j + 3);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 2, j);
```

```
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 2, j + 3);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 3, j + 1);
                setBrightness(gc, newData, weight, height, sqrt(a) * 2 - 1, i
+ 3, j + 2);


                setBrightness(gc, newData, weight, height, 4 * a - 4 *
sqrt(a) + 1, i, j);
                setBrightness(gc, newData, weight, height, 4 * a - 4 *
sqrt(a) + 1, i, j + 3);
                setBrightness(gc, newData, weight, height, 4 * a - 4 *
sqrt(a) + 1, i + 3, j);
                setBrightness(gc, newData, weight, height, 4 * a - 4 *
sqrt(a) + 1, i + 3, j + 3);
            }
        }
    }
    return Picture(newData, weight, height, gc);
}

void
Dither::setBrightness(GammaCorrection const &gc, std::vector<uchar> &data,
int weight, int height, double value, int i,
                      int j) {
    if (i < height && j < weight) {
        data[i * weight + j] = value * 255;
    }
}
```

GammaCorrection.h

```
#ifndef HW2_GAMMACORRECTION_H
#define HW2_GAMMACORRECTION_H
```

```
class GammaCorrection {
private:
    double gamma;

    int sRGB(int oldBrightness, int newBrightness, double proportion) const;

    int exponential(int oldBrightness, int newBrightness, double proportion)
const;

public:
    GammaCorrection(double gamma = 0);

    int getNewBrightness(int oldBrightness, int newBrightness, double
proportion) const;

    double encode(int brightness) const;

    int decode(double brightness) const;
};


#endif //HW2_GAMMACORRECTION_H
```

GammaCorrection.cpp

```
#include "GammaCorrection.h"
#include <cmath>

int GammaCorrection::sRGB(int oldBrightness, int newBrightness, double
proportion) const {
    double a = oldBrightness / 255.0;
    a = a <= 0.04045 ? a / 12.92 : pow((a + 0.055) / 1.055, 2.4);
    double b = newBrightness / 255.0;
    b = b <= 0.04045 ? b / 12.92 : pow(((b + 0.055) / 1.055), 2.4);
    a = (1.0 - proportion) * a + proportion * b;
    a = a <= 0.0031308 ? 12.92 * a : pow(a, 0.416) * 1.055 - 0.055;
    return round(a * 255.0);
}

int GammaCorrection::exponential(int oldBrightness, int newBrightness, double
proportion) const {
    double a = oldBrightness / 255.0;
    a = pow(a, gamma);
    double b = newBrightness / 255.0;
    b = pow(b, gamma);
    a = (1.0 - proportion) * a + proportion * b;
    a = pow(a, 1 / gamma);
    return round(a * 255.0);
}

int GammaCorrection::getNewBrightness(int oldBrightness, int newBrightness,
double proportion) const{
    if (gamma == 0) {
        return sRGB(oldBrightness, newBrightness, proportion);
    } else {
        return exponential(oldBrightness, newBrightness, proportion);
    }
}


GammaCorrection::GammaCorrection(double gamma) : gamma(gamma) {}
```

```
int GammaCorrection::decode(double brightness) const{
    brightness /= 255.0;
    if (gamma == 0) {
        return round((brightness <= 0.0031308 ? 12.92 * brightness :
pow(brightness, 0.416) * 1.055 - 0.055) * 255.0);
    } else {
        return round(pow(brightness, 1 / gamma) * 255.0);
    }
}

double GammaCorrection::encode(int brightness) const{
    double b = brightness / 255.0;
    if (gamma == 0) {
        return (b <= 0.04045 ? b / 12.92 : pow((b + 0.055) / 1.055, 2.4)) *
255.0;
    } else {
        return pow(b, gamma) * 255.0;
    }
};
```

main.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include "Picture.h"
#include "Dither.h"

bool getInt(char *str, int &value) {
    try {
        value = std::stoi(str);
        return false;
    } catch (std::invalid_argument &e) {
        return true;
    }
}

bool getDouble(char *str, double &value) {
    try {
        value = std::stod(str);
        return false;
    } catch (std::invalid_argument &e) {
        return true;
    }
}

int main(int argc, char **argv) {
    if (argc != 7) {
        std::cerr
                << "Use dither <input file> <output file> <gradient>
<dithering> <bitness> <gamma>"
                << std::endl;
        return 1;
    }

    std::ifstream in(argv[1], std::ios_base::binary);
    if (!in.is_open()) {
        std::cerr << "Can't open input file" << std::endl;
        return 1;
    }
```

```
    std::ofstream out(argv[2], std::ios_base::binary);
    if (!out.is_open()) {
        std::cerr << "Can't open output file" << std::endl;
        in.close();
        return 1;
    }
    int gradient, dithering, bitness;
    double gamma;
    if (getInt(argv[3], gradient) || !(gradient == 0 || gradient == 1)
        || getInt(argv[4], dithering) || !(dithering >= 0 && dithering <= 7)
        || getInt(argv[5], bitness) || !(bitness >= 1 && bitness <= 8)
        || getDouble(argv[6], gamma) || gamma < 0) {
        std::cerr << "Incorrect args" << std::endl;
        in.close();
        out.close();
        remove(argv[2]);
        return 1;
    }
    Picture picture((GammaCorrection(gamma)));
    try {
        if (gradient) {
            picture.readGradient(in);
        } else {
            picture.read(in);
        }
    } catch (std::bad_alloc &e) {
        std::cerr << "Could not alloc memory" << std::endl;
        in.close();
        out.close();
        remove(argv[2]);
        return 1;
    }
    if (dithering == 1) {
        picture = Dither::ordered(picture);
    }
    if (dithering == 2) {
        picture = Dither::random(picture);
    }
    if (dithering == 3) {
        picture = Dither::floydSteinberg(picture, bitness);
    }
    if (dithering == 4) {
        picture = Dither::jjn(picture, bitness);
    }
    if (dithering == 5) {
        picture = Dither::sierra3(picture, bitness);
    }
    if (dithering == 6) {
        picture = Dither::atkinson(picture, bitness);
    }
    if (dithering == 7) {
        picture = Dither::halftone(picture);
    }
    picture = Dither::round(picture, bitness);
    picture.write(out);
    in.close();
    out.close();
    return 0;
}
```

Picture.h

```
#ifndef HW2_PICTURE_H
#define HW2_PICTURE_H


#include <vector>
#include <fstream>
#include <algorithm>
#include "GammaCorrection.h"

class Picture {
public:
    typedef unsigned char uchar;

    bool read(std::ifstream &in);

    bool write(std::ofstream &out);

    bool readGradient(std::ifstream &in);

    Picture(const GammaCorrection &gammaCorrection);

    std::vector<uchar> data;

    size_t getWeight() const;

    size_t getHeight() const;

    const GammaCorrection &getGammaCorrection() const;

    Picture(std::vector<uchar> data, size_t weight, size_t height, const
GammaCorrection &gammaCorrection);

private:
    size_t weight, height;
    GammaCorrection gammaCorrection;
};


#endif //HW2_PICTURE_H
```

Picture.cpp

```
#include "Picture.h"

#include <utility>

bool Picture::read(std::ifstream &in) {
    std::string format;
    size_t maxValue;
    if (!(in >> format >> weight >> height >> maxValue) ||
        format != "P5" || maxValue != 255) {
        return true;
    }
    in.get();
    data.resize(weight * height);
    in.read((char *) data.data(), weight * height);
    return false;
}

bool Picture::write(std::ofstream &out) {
    out << "P5" << std::endl;
    out << weight << " " << height << std::endl;
    out << 255 << std::endl;
```

```
    out.write((char *) data.data(), data.size());
    return false;
}

bool Picture::readGradient(std::ifstream &in) {
    std::string format;
    size_t maxValue;
    if (!(in >> format >> weight >> height >> maxValue) ||
        format != "P5" || maxValue != 255) {
        return true;
    }
    in.get();
    data.resize(weight * height);
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < weight; j++) {
            data[i * weight + j] = ((double) i + j) / (height + weight) *
255.0;
        }
    }
    return false;
}

Picture::Picture(const GammaCorrection &gammaCorrection) :
gammaCorrection(gammaCorrection) {}

size_t Picture::getWeight() const {
    return weight;
}

size_t Picture::getHeight() const {
    return height;
}

const GammaCorrection &Picture::getGammaCorrection() const {
    return gammaCorrection;
}

Picture::Picture(std::vector<uchar> data, size_t weight, size_t height, const
GammaCorrection &gammaCorrection)
        : data(std::move(data)), weight(weight), height(height),
gammaCorrection(gammaCorrection) {}
```