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 № 2

Изучение алгоритмов отрисовки растровых линий с применением сглаживания и гамма-коррекции

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, рисующую линию на изображении в формате PGM (P5) с учетом гамма-коррекции sRGB.

Теоретическая справка

Гамма-коррекция — предыскажения яркости чёрно-белого или цветоделённых составляющих цветного изображения при его записи в телевидении и цифровой фотографии. В большинстве случаев представляется степенной функцией.

Особенность человеческого глаза заключается в том, что он лучше различает темные полутона, чем светлые. При стандартном линейном кодировании различие в светлых областях практически не различимо, в то время как в темных различаются переходы. Гамма-коррекция создана исправить это, перенеся информацию со светлых полутонов в темные.

sRBG – вариация на тему гамма-коррекции. В отличии оригинала в начале имеет линейный участок, после которого переходит в степенную функцию.

Решение

Линия, которую нужно нарисовать, представляется в виде прямоугольника, а каждый пиксель – виде квадрата. Результирующая яркость пикселя – сумма изначальной яркости и яркости линии, взяты пропорционально непокрытой и покрытой площадей соответственно:

$$x = x_0(1 - s) + x_1 s$$

где x – результирующая яркость,

x_0 – яркость пикселя на изображении,

x_1 – яркость линии,

s – площадь пересечения линии и пикселя.

Вычисление площади

Первый случай – когда пиксель полностью покрывается прямоугольником (здесь и далее под *прямоугольником* подразумевается линия, которую нужно нарисовать, а под *линией* – геометрическая прямая). Для это достаточно проверить крайние точки на принадлежность прямоугольника. В таком случае $s = 1$.

К сожалению, в противоположном случае работает не всегда, а именно при толщине меньше 1. Если все вершины пикселя не покрыты ≠ пиксель полностью не покрыт.

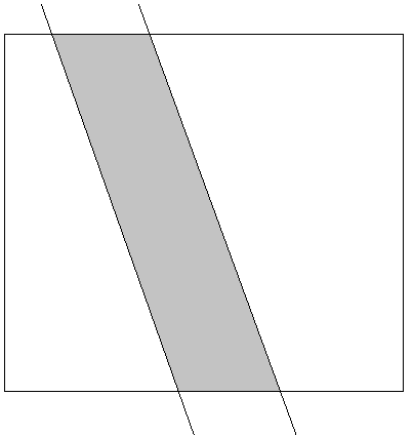

Рис.1. Пример покрытия пикселя

Подобных случаев очень много. Поэтому гораздо легче искать площадь непокрытой части. Идея заключается в том, чтобы начиная с каждой непокрытой вершины «вырезать» непокрытую часть.

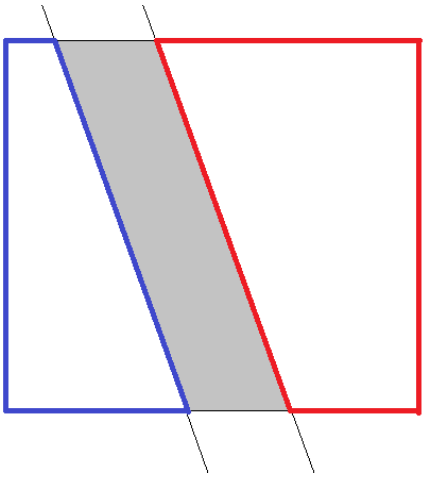

Рис. 2. Пример вырезанных частей

Для этого достаточно пройтись от каждой вершины по часовой стрелке, каждый раз переходя либо к следующей вершине пикселя, либо к точке пересечения стороны прямоугольника и стороны пикселя.

Хоть такой алгоритм и является затратным с точки зрения вычислений, но он обрабатывает все случаи, когда вырезаемая часть является выпуклым много угольником.

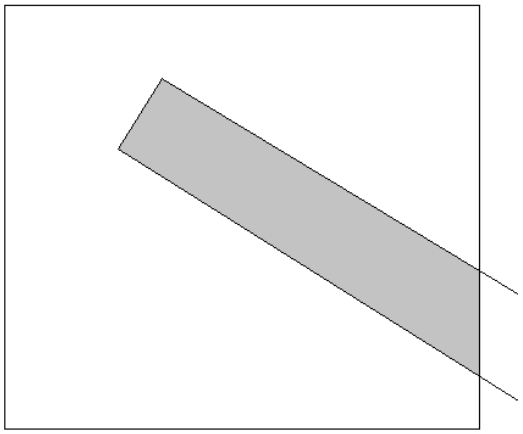

Рис. 3. Пример вырожденного случая

Когда вырезаемая фигура не является выпуклой – случай куда более сложный. Но для его появления должно быть выполнено условие: вершина прямоугольника должна лежать внутри данного прямоугольника. Таких случаев может быть не больше 4 на 1 линию, поэтому нецелесообразно их обрабатывать. ~~*Не баг, а фича.*~~

Листинг программ

GammaCorrection.h

```
#ifndef HW2_GAMMACORRECTION_H
#define HW2_GAMMACORRECTION_H


class GammaCorrection {
private:
    double gamma;

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

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

public:
    GammaCorrection(double gamma = -1.0);

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


#endif //HW2_GAMMACORRECTION_H
```

GammaCorrection.cpp

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

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

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

int GammaCorrection::getNewBrightness(int oldBrightness, int newBrightness,
double proportion) {
    if (gamma == -1.0) {
        return sRGB(oldBrightness, newBrightness, proportion);
    } else {
        return exponential(oldBrightness, newBrightness, proportion);
    }
}
```

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

Line.h

```
#ifndef HW2_LINE_H
#define HW2_LINE_H

#include <utility>

class Line {
public:
    typedef std::pair<double, double> Point;

    const Point &getVector() const;

    const Point &getPoint() const;

    Line();

    Line(Point point1, Point point2);

    friend Point intersectionPoint(Line line1, Line line2);

    friend bool isParallel(Line const &line1, Line const &line2);

    friend bool operator==(Line const &line1, Line const &line2);

    double distanceToPoint(Point point0) const;

    bool isPointOn(Point point0) const;

private:
    Point vector;
    Point point;
};

double distanceBetweenPoints(Line::Point point1, Line::Point point2);

bool dEqual(double a, double b);

int sign(double a);

#endif //HW2_LINE_H
```

Line.cpp

```
#include "Line.h"

#include <utility>
#include <cmath>

#define x first
#define y second

const Line::Point &Line::getVector() const {
    return vector;
}

const Line::Point &Line::getPoint() const {
    return point;
}
```

```
Line::Line(Line::Point point1, Line::Point point2)
        : point(std::move(point1)) {
    vector = std::make_pair(point2.x - point1.x, point2.y - point1.y);
    double length = sqrtl(this->vector.x * this->vector.x + this->vector.y *
this->vector.y);
    this->vector.x /= length;
    this->vector.y /= length;
}

Line::Point intersectionPoint(Line line1, Line line2) {
    if (isParallel(line1, line2)) {
        return std::make_pair(nan("0"), nan("0"));
    }
    if (line1.vector.x == 0 || line2.vector.y == 0) {
        std::swap(line1, line2);
    }
    auto vector = line1.vector;
    double a = line2.point.x - line1.point.x;
    double b = line2.point.y - line1.point.y;

    line2.vector.x /= line1.vector.x;
    a /= line1.vector.x;
    line1.vector.x = 1;

    line2.vector.y -= line2.vector.x * line1.vector.y;
    b -= a * line1.vector.y;
    line1.vector.y = 0;

    b /= line2.vector.y;
    a -= line2.vector.x * b;
    return std::make_pair(
            line1.point.x + vector.x * a,
            line1.point.y + vector.y * a);
}

bool isParallel(Line const &line1, Line const &line2) {
    return (line1.vector.x == 0 && line2.vector.x == 0)
           || (line1.vector.y == 0 && line2.vector.y == 0)
           || (line1.vector.x / line2.vector.x == line1.vector.y /
line2.vector.y);
}

Line::Line() {
    vector = std::make_pair(1.0, 0.0);
    point = std::make_pair(0.0, 0.0);
}

bool operator==(Line const &line1, Line const &line2) {
    if (line1.point == line2.point) {
        return isParallel(line1, line2);
    }
    Line line = Line(std::make_pair(line1.point.x - line2.point.x,
line1.point.y - line2.point.y), line1.point);
    return isParallel(line1, line2) && isParallel(line1, line);
}

double Line::distanceToPoint(Line::Point point0) const {
    if (isPointOn(point0)) {
        return 0;
    }
    Line line;
    line.point = point0;
```

```
    line.vector.x = vector.y;
    line.vector.y = -vector.x;
    return distanceBetweenPoints(point0, intersectionPoint(*this, line));
}

bool Line::isPointOn(Line::Point point0) const {
    return (*this) == Line(point, point0);
}

double distanceBetweenPoints(Line::Point point1, Line::Point point2) {
    return powl((point1.x - point2.x) * (point1.x - point2.x) + (point1.y -
point2.y) * (point1.y - point2.y), 0.5);
}

bool dEqual(double a, double b) {
    const double eps = 1e-12;
    return (a > b - eps && a < b + eps);
}

int sign(double a) {
    return (0 < a) - (a < 0);
}
```

main.cpp

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

#define x first
#define y second

typedef unsigned char uchar;

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

int main(int argc, char **argv) {
    if (argc != 10 && argc != 9) {
        std::cerr
                << "Use linedrawer <input file> <output file> <line
brightness> <line thickness> <x1> <y1> <x2> <y2> [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;
    }

    uchar lineBrightness;
    try {
        int intermediateValue = std::stoi(argv[3]);
        if (intermediateValue < 0 || intermediateValue > 255) {
            throw std::invalid_argument("");
        }
        lineBrightness = intermediateValue;
    } catch (std::invalid_argument &e) {
        std::cerr << "Incorrect args" << std::endl;
        in.close();
        out.close();
        return 1;
    }
    double lineThickness, x1, y1, x2, y2, gamma = -1;
    if (getLongDouble(argv[4], lineThickness) || lineThickness < 0 ||
        getLongDouble(argv[5], x1) || getLongDouble(argv[6], y1) ||
        getLongDouble(argv[7], x2) || getLongDouble(argv[8], y2)) {
        std::cerr << "Incorrect args" << std::endl;
        in.close();
        out.close();
        return 1;
    }
    if (argc == 10) {
        if (getLongDouble(argv[9], gamma)) {
            std::cerr << "Incorrect args" << std::endl;
            in.close();
            out.close();
            return 1;
        }
    }
    Picture picture;
    if (picture.read(in)) {
        std::cerr << "Can't read input file" << std::endl;
        in.close();
        out.close();
        return 1;
    }
    picture.drawLine(x1, y1, x2, y2, lineThickness, lineBrightness, {gamma});
    if (picture.write(out)) {
        std::cerr << "Can't write output file" << std::endl;
        in.close();
        out.close();
        return 1;
    }
    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"
#include "Rectangle.h"
```

```
class Picture {
public:
    typedef unsigned char uchar;

    bool read(std::ifstream &in);
    bool write(std::ofstream &out);
    void drawLine(double x1, double y1, double x2, double y2, double
thickness, int brightness, GammaCorrection gammaCorrection);

private:
    std::vector<uchar> data;
    size_t weight, height;
};


#endif //HW2_PICTURE_H
```

Picture.cpp

```
#include "Picture.h"

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;
}

void Picture::drawLine(double x1, double y1, double x2, double y2, double
thickness, int brightness, GammaCorrection gammaCorrection) {
    Rectangle rectangle = Rectangle(x1, y1, x2, y2, thickness);
    size_t iMin = std::max(0ll, (long long) (std::min(y1, y2) - thickness / 2
- 2));
    size_t jMin = std::max(0ll, (long long) (std::min(x1, x2) - thickness / 2
- 2));
    size_t iMax = std::min(height, (size_t) (std::max(y1, y2) + thickness / 2
+ 2));
    size_t jMax = std::min(weight, (size_t) (std::max(x1, x2) + thickness / 2
+ 2));
    for (size_t i = iMin; i < iMax; i++) {
        for (size_t j = jMin; j < jMax; j++) {
            double s = rectangle.intersectionArea({(double) j, (double) i});
            data[i * weight + j] = gammaCorrection.getNewBrightness(data[i *
weight + j], brightness, s);
        }
    }
}
```

Pixel.h

```
#ifndef HW2_PIXEL_H
#define HW2_PIXEL_H


#include <utility>
#include "Line.h"

class Pixel {
private:
    Line::Point center;
    Line::Point borderPoints[4];
    Line borders[4];
public:
    Line::Point getRightUp() const;

    Line::Point getRightDown() const;

    Line::Point getLeftUp() const;

    Line::Point getLeftDown() const;

    Pixel(double x, double y);

    const Line &getUpLine() const;

    const Line &getDownLine() const;

    const Line &getLeftLine() const;

    const Line &getRightLine() const;

    const Line::Point *getBorderPoints() const;

    const Line *getBorders() const;

    bool isPointInside(Line::Point point) const;
};


#endif //HW2_PIXEL_H
```

Pixel.cpp

```
#include "Pixel.h"

#define x first
#define y second

Line::Point Pixel::getRightUp() const {
    return borderPoints[1];
}

Line::Point Pixel::getRightDown() const {
    return borderPoints[2];
}

Line::Point Pixel::getLeftUp() const {
    return borderPoints[0];
}
```

```
Line::Point Pixel::getLeftDown() const {
    return borderPoints[3];
}

Pixel::Pixel(double x, double y) {
    center = std::make_pair(x, y);
    borderPoints[0] = std::make_pair(center.x - 0.5, center.y - 0.5);
    borderPoints[1] = std::make_pair(center.x + 0.5, center.y - 0.5);
    borderPoints[2] = std::make_pair(center.x + 0.5, center.y + 0.5);
    borderPoints[3] = std::make_pair(center.x - 0.5, center.y + 0.5);
    borders[0] = Line(getLeftUp(), getRightUp());
    borders[1] = Line(getRightUp(), getRightDown());
    borders[2] = Line(getRightDown(), getLeftDown());
    borders[3] = Line(getLeftDown(), getLeftUp());
}

const Line &Pixel::getUpLine() const {
    return borders[0];
}

const Line &Pixel::getDownLine() const {
    return borders[2];
}

const Line &Pixel::getLeftLine() const {
    return borders[3];
}

const Line &Pixel::getRightLine() const {
    return borders[1];
}

const Line::Point *Pixel::getBorderPoints() const {
    return borderPoints;
}

const Line *Pixel::getBorders() const {
    return borders;
}

bool Pixel::isPointInside(Line::Point point) const {
    return dEqual(getUpLine().distanceToPoint(point) +
getDownLine().distanceToPoint(point), 1)
           && dEqual(getLeftLine().distanceToPoint(point) +
getRightLine().distanceToPoint(point), 1);
}
```

Rectangle.h

```
#ifndef HW2_RECTANGLE_H
#define HW2_RECTANGLE_H


#include "Line.h"
#include "Pixel.h"

class Rectangle {
private:
    Line::Point points[4];
    Line sides[4];
    double weight, height, thickness;
public:
    Rectangle(double x1, double y1, double x2, double y2, double thickness);
```

```
    bool isPointInside(Line::Point point) const;

    double intersectionArea(Pixel pixel) const;
};

double getArea(std::vector<Line::Point> const &pointsList);

#endif //HW2_RECTANGLE_H
```

Rectangle.cpp

```
#include <algorithm>
#include <vector>
#include "Rectangle.h"

#define x first
#define y second

Rectangle::Rectangle(double x1, double y1, double x2, double y2, double
thickness) : thickness(
        thickness) {
    if (x1 == x2 && y1 == y2) {
        return;
    }
    Line line = Line(std::make_pair(x1, y1), std::make_pair(x2, y2));
    Line::Point vector = line.getVector();
    double borderDistance = std::min(0.5, thickness / 2);

    points[1] = std::make_pair(x1 - vector.x * borderDistance, y1 - vector.y
* borderDistance);
    points[3] = std::make_pair(x2 + vector.x * borderDistance, y2 + vector.y
* borderDistance);

    points[0] = std::make_pair(points[1].x - vector.y * thickness / 2,
points[1].y + vector.x * thickness / 2);
    points[1] = std::make_pair(points[1].x + vector.y * thickness / 2,
points[1].y - vector.x * thickness / 2);
    points[2] = std::make_pair(points[3].x + vector.y * thickness / 2,
points[3].y - vector.x * thickness / 2);
    points[3] = std::make_pair(points[3].x - vector.y * thickness / 2,
points[3].y + vector.x * thickness / 2);

    sides[0] = Line(points[0], points[1]);
    sides[1] = Line(points[1], points[2]);
    sides[2] = Line(points[2], points[3]);
    sides[3] = Line(points[3], points[0]);

    weight = distanceBetweenPoints(points[0], points[1]);
    height = distanceBetweenPoints(points[0], points[3]);
}

bool Rectangle::isPointInside(Line::Point point) const {
    return dEqual(sides[0].distanceToPoint(point) +
sides[2].distanceToPoint(point), height) &&
           dEqual(sides[1].distanceToPoint(point) +
sides[3].distanceToPoint(point), weight);
}

double Rectangle::intersectionArea(Pixel pixel) const {
    if (thickness >= 1
        && !isPointInside(pixel.getLeftUp()) &&
!isPointInside(pixel.getRightUp())
```

```
        && !isPointInside(pixel.getLeftDown()) &&
!isPointInside(pixel.getRightDown())) {
        return 0;
    }
    if (isPointInside(pixel.getLeftUp()) && isPointInside(pixel.getRightUp())
        && isPointInside(pixel.getLeftDown()) &&
isPointInside(pixel.getRightDown())) {
        return 1;
    }
    for (Line::Point point : points) {
        if (pixel.isPointInside(point)) {
            return 0;
        }
    }
    double s = 0;
    bool used[] = {false, false, false, false};
    const Line::Point *borderPoints = pixel.getBorderPoints();
    const Line *borders = pixel.getBorders();
    for (int i = 0; i < 4; i++) {
        if (used[i] || isPointInside(borderPoints[i])) {
            continue;
        }
        std::vector<Line::Point> pointList;
        int j = i;
        do {
            used[j] = true;
            pointList.push_back(borderPoints[j]);
            Line::Point nextPoint = borderPoints[(j + 1) % 4];
            Line nextLine = borders[j];
            for (Line side : sides) {
                Line::Point interPoint = intersectionPoint(side, borders[j]);
                if (borderPoints[j] != interPoint
                    && pixel.isPointInside(interPoint)
                    && distanceBetweenPoints(borderPoints[j], nextPoint) >
                       distanceBetweenPoints(borderPoints[j], interPoint)) {
                    nextPoint = interPoint;
                    nextLine = side;
                }
            }
            if (nextPoint == borderPoints[(j + 1) % 4]) {
                j = (j + 1) % 4;
            } else {
                pointList.push_back(nextPoint);
                for (int t = 0; t < 4; t++) {
                    if (t == j) {
                        continue;
                    }
                    Line::Point interPoint = intersectionPoint(nextLine,
borders[t]);
                    if (pixel.isPointInside(interPoint) && interPoint !=
borderPoints[(t + 1) % 4]) {
                        pointList.push_back(interPoint);
                        j = (t + 1) % 4;
                        break;
                    }
                }
            }
        } while (!used[j]);
        s += getArea(pointList);
    }
    return 1 - std::abs(s);
}
```

```
double getArea(const std::vector<Line::Point> &pointList) {
    if (pointList.empty()) {
        return 0;
    }
    double s = 0;
    for (size_t i = 0; i < pointList.size() - 1; i++) {
        s += (pointList[i + 1].x - pointList[i].x) * (pointList[i].y +
pointList[i + 1].y) / 2;
    }
    s += (pointList[0].x - pointList.back().x) * (pointList[0].y +
pointList.back().y) / 2;
    return std::abs(s);
}
```