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

Компьютерная графика

Лабораторная работа № 2

**Изучение алгоритмов отрисовки растровых линий с применением сглаживания и**

**гамма-коррекции**

Выполнил: студент группы № М32342  
Кубанцев Ярослав Максимович

Проверил: ассистент факультета прикладной оптики Скаков Павел Сергеевич

САНКТ -ПЕТЕРБУРГ  
2020

**Содержание**

|  |  |
| --- | --- |
| Цель работы | 3 |
| Теоретическая справка | 4 |
| Решение | 5 |
| Листинг программ | 7 |
|  |  |

**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.

В современном цифровом телевидении и цифровой фотографии гамма-коррекция при кодировании позволяет учитывать особенности восприятия полутонов человеческим зрением для повышения эффективности использования глубины цвета. Известно, что в нормальных условиях глаз лучше различает перепады освещённости в тенях, чем в света́х[3]. При отсутствии гамма-коррекции, в светах цифрового изображения кодируется слишком много полутонов, которые неспособен различить зритель. Напротив, в тенях остаётся слишком мало информации, снижая качество изображения[4]. В случае линейного кодирования файлов JPEG по существующей 8-битной системе, неизбежна «постеризация» изображения, когда глаз различает ступенчатые тональные переходы.

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

**Решение**

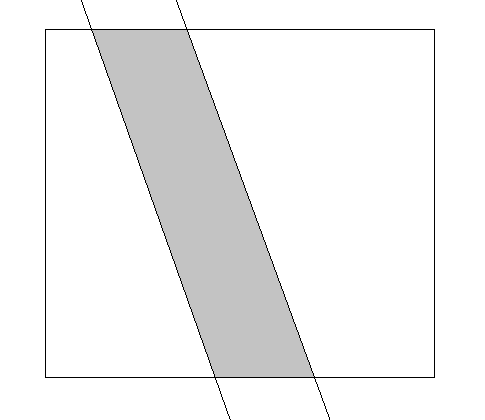
Линия, которую нужно нарисовать, представляется в виде прямоугольника, а каждый пиксель – виде квадрата. Результирующая яркость пикселя – сумма изначальной яркости и яркости линии, взяты пропорционально непокрытой и покрытой площадей соответственно:

где *x* – результирующая яркость, – яркость пикселя на изображении, – яркость линии, *s* – площадь пересечения линии и пикселя.

**Вычисление площади**

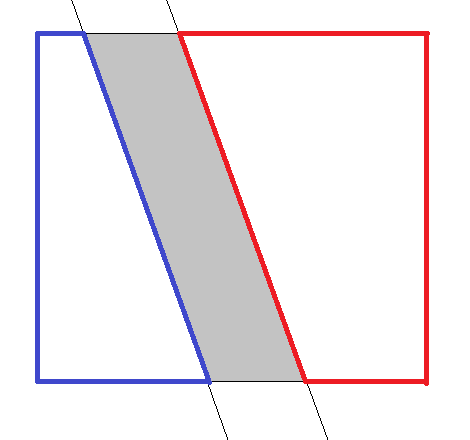
Первый случай – когда пиксель полностью покрывается прямоугольником (здесь и далее под *прямоугольником* подразумевается линия, которую нужно нарисовать, а под *линией* – геометрическая прямая). Для это достаточно проверить крайние точки на принадлежность прямоугольника. В таком случае *s* = 1.

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



*Пример покрытия пикселя*

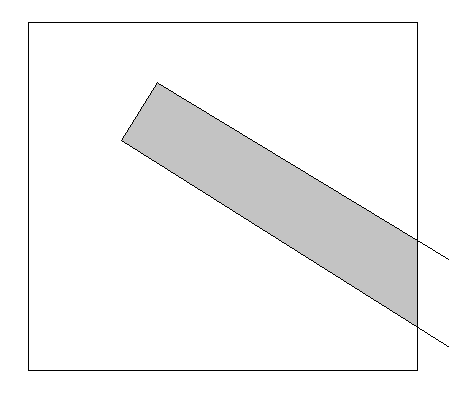
Подобных случаев очень много. Поэтому гораздо легче искать площадь непокрытой части. Идея заключается в том, чтобы начиная с каждой непокрытой вершины «вырезать» непокрытую часть.



*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.



*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:

long double gamma;

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

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

public:

GammaCorrection(long double gamma = -1.0);

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

};

#endif //HW2\_GAMMACORRECTION\_H

*GammaCorrection.cpp*

#include "GammaCorrection.h"

#include <cmath>

int GammaCorrection::sRGB(int oldBrightness, int newBrightness, long double proportion) {

long double a = oldBrightness / 255.0;

a = a <= 0.04045 ? a / 12.92 : powl((a + 0.055) / 1.055, 2.4);

long 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, long double proportion) {

long double a = oldBrightness / 255.0;

a = powl(a, gamma);

long 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, long double proportion) {

if (gamma == -1.0) {

return sRGB(oldBrightness, newBrightness, proportion);

} else {

return exponential(oldBrightness, newBrightness, proportion);

}

}

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

*Line.h*

#ifndef HW2\_LINE\_H

#define HW2\_LINE\_H

#include <utility>

class Line {

public:

typedef std::pair<long double, long 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);

long double distanceToPoint(Point point0) const;

bool isPointOn(Point point0) const;

private:

Point vector;

Point point;

};

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

bool dEqual(long double a, long double b);

int sign(long 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);

long 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;

long double a = line2.point.x - line1.point.x;

long 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);

}

long 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);

}

long 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(long double a, long double b) {

const long double eps = 1e-12;

return (a > b - eps && a < b + eps);

}

int sign(long double a) {

return (0 < a) - (a < 0);

}

*main.cpp*

#include <iostream>

#include "Picture.h"

#define x first

#define y second

typedef unsigned char uchar;

bool getLongDouble(char \*str, long 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;

}

long 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 "GammaCorrection.h"

#include "Rectangle.h"

class Picture {

public:

typedef unsigned char uchar;

bool read(std::ifstream &in);

bool write(std::ofstream &out);

void drawLine(long double x1, long double y1, long double x2, long double y2, long 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(long double x1, long double y1, long double x2, long double y2, long 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++) {

long double s = rectangle.intersectionArea({(long double) j, (long 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(long double x, long 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(long double x, long 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];

long double weight, height, thickness;

public:

Rectangle(long double x1, long double y1, long double x2, long double y2, long double thickness);

bool isPointInside(Line::Point point) const;

long double intersectionArea(Pixel pixel) const;

};

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

#endif //HW2\_RECTANGLE\_H

*Rectangle.h*

#include <algorithm>

#include <vector>

#include "Rectangle.h"

#define x first

#define y second

Rectangle::Rectangle(long double x1, long double y1, long double x2, long double y2, long 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();

long double borderDistance = std::min(0.5l, 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);

}

long 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;

}

}

long 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);

}

long double getArea(const std::vector<Line::Point> &pointList) {

if (pointList.empty()) {

return 0;

}

long 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);

}