МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение высшего образования

Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Лабораторная работа № 2

По дисциплине «Компьютерная геометрия и графика»

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

Выполнил студент группы №M3101
Семенов Георгий Витальевич

Преподаватель:
Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ

2020

Цель работы: изучить алгоритмы и реализовать программу, рисующую линию на изображении в формате PGM (P5) с учетом гамма-коррекции sRGB.

Описание:

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> <яркость_линии> <толщина_линии> <x_начальный> <y_начальный> <x_конечный> <y_конечный> <гамма>

где

- <яркость_линии>: целое число 0..255;
- <толщина_линии>: положительное дробное число;
- <x,y>: координаты внутри изображения, (0;0) соответствует левому верхнему углу, дробные числа (целые значения соответствуют центру пикселей).
- <гамма>: (optional)положительное вещественное число: гамма-коррекция с введенным значением в качестве гаммы. При его отсутствии используется sRGB.

Частичное решение: <толщина линии>=1, <гамма>=2.0, координаты начала и конца – целые числа, чёрный фон вместо данных исходного файла (размеры берутся из исходного файла).

Полное решение: всё работает (гамма + sRGB, толщина не только равная 1, фон из входного изображения) + корректно выделяется и освобождается память, закрываются файлы, есть обработка ошибок.

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.

Следующие параметры гарантировано не будут выходить за обусловленные значения:

- <яркость_линии> = целое число 0..255;
- <толщина_линии> = положительное вещественное число;
- width и height в файле - положительные целые значения;

- яркостных данных в файле ровно width * height;
- <x_начальный> <x_конечный> = [0..width];
- <y_начальный> <y_конечный> = [0..height];

Теоретическая часть

Прямая на плоскости задаётся двумя точками и цветом.

Один из методов отрисовки растровых отрезков представлен в алгоритме Брезенхема, в котором цвет пикселя (окрашивается он или остаётся прежним) определяется последовательно передающейся **ошибкой** — метрической характеристикой, описывающей отклонение математической прямой от рисуемой пиксельно.

Иной метод отрисовки растрового отрезка с размытием представлен в алгоритме Ву, где соседние пиксели частично закрашиваются согласно их отклонению от математической прямой.

При отрисовке отрезков производят предварительное декодирование из gamma или sRGB представления, т.к. восприятие цвета человеческим глазом зависит от количества фотонов на единицу площади не линейно. Законы gamma и sRGB представлений позволяют переводить изображения из представления для человеческого субъективного восприятия в нестрогое физическое представление.

В данной лабораторной работе не была решена проблема отрисовки толстых линий со сглаживанием (anti-alias thick lines).

Экспериментальная часть

Язык программирования: C++.

Изображение описано в классе pnmImage, хранящее в двумерном массиве (vector) указатели на экземпляры класса pnmBWColor, в котором реализованы переводы sRGB и gamma.

В методах drawSingularLine, drawBWPoint, bresenham и wu реализован функционал отрисовки растровых линий толщиной 1.

Вывод

Алгоритмы отрисовки прямой по времени зависят линейно от площади покрытия пикселей, т.е. координат отрезка и его толщины.

Листинг

https://github.com/MrGeorgeous/ComputerGeometryAndGraphics/blob/master/README.md

Файл: Main.cpp

```
#define _CRT_SECURE_NO_WARNINGS

#include<stdio.h>
#include <iostream>
#include <fstream>
#include <istream>
#include <iomanip>

#include<string>
#include<vector>
#include<set>
#include<queue>


#include <functional>
#include <algorithm>
#include <utility>


using namespace std;

const int UCHAR_SIZE = sizeof(unsigned char);

class pnmBWColor {

public:

	unsigned char color = 0;

	pnmBWColor() {

	}

	pnmBWColor(const pnmBWColor& c) {
		color = c.color;
	}

	pnmBWColor(unsigned char black) {
		color = black;
	}

	~pnmBWColor() {

	}

	void setColor(unsigned char r) {
		color = r;
	}
```

```
	void inverseColor(unsigned char depth = 255) {
		color = depth - color;
	}


	void gamma(double gamma = 1, size_t depth = 255, bool reverse = false) {
		if (!reverse) {
			gammaDecode(color, gamma, depth);
		}
		else {
			gammaEncode(color, gamma, depth);
		}
	}

	void srgb(size_t depth = 255, bool reverse = false) {

		if (!reverse) {
			srgbDecode(color, depth);
		}
		else {
			srgbEncode(color, depth);
		}

	}

	static void gammaEncode(unsigned char& a, double gamma = 1.0, size_t depth = 255) {
		a = depth * pow(double(a) / depth, 1.0 / gamma);
	}

	static void gammaDecode(unsigned char& a, double gamma = 1.0, size_t depth = 255) {
		a = depth * pow(double(a) / depth, gamma);
	}

	static void srgbEncode(unsigned char& a, size_t depth = 255) {
		double r = double(a) / depth;

		if (r <= 0.0031308) {
			a = depth * (12.92 * r);
		}
		else {
			a = depth * ((211.0 * pow(r, 5.0 / 12) - 11) / 200);
		}
	}

	static void srgbDecode(unsigned char& a, size_t depth = 255) {
		double r = double(a) / depth;

		if (r <= 0.04045) {
			a = depth * (25 * r / 323);
		}
		else {
			a = depth * pow((200.0 * r + 11.0) / 211, 12.0 / 5);
		}
	}

};




typedef vector<vector<pnmBWColor*>> pnmMatrix;
typedef vector<char> chars;
```

```
class pnmBWImage {

public:

	int width = 0; // y
	int height = 0; // x
	int depth = 255; // d
	int gamma = -1; // gamma

	pnmMatrix m;
	chars errorEncounter;

	set<pnmBWColor*> manuallyEdited;


	pnmBWImage(size_t w, size_t h, pnmBWColor color = pnmBWColor(255)) {

		m = pnmMatrix(h, vector<pnmBWColor*>(w, nullptr));

		for (int j = 0; j < h; j++) {
			for (int i = 0; i < w; i++) {
				m[j][i] = new pnmBWColor(color);
			}
		}

		width = w;
		height = h;

	}

	~pnmBWImage() {

		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				delete m[i][j];
			}
		}

	}

	pnmBWImage(string filename, int g = -1) {

		if (!errorEncounter.empty()) {
			return;
		}

		FILE* file = fopen(filename.c_str(), "rb");
		if (!(file != NULL)) {
			errorEncounter.push_back(1);
			return;
		}

		char p1, p2 = ' ';
		int w = 0, h = 0, d = 0;

		fscanf(file, "%c%c\n%i %i\n%i\n", &p1, &p2, &w, &h, &d);

		width = w;
		height = h;
		depth = d;
		gamma = g;
```

```
        if (((w <= 0) || (h <= 0))) {
                errorEncounter.push_back(1);
                return;
        }

        m = pnmMatrix(h, vector<pnmBWColor*>(w, nullptr));

        if (p1 != 'P') {
                errorEncounter.push_back(1);
                return;
        }
        if (!((p2 == '5'))) {
                errorEncounter.push_back(1);
                return;
        }
        if (!(d == 255)) {
                errorEncounter.push_back(1);
                return;
        }

        unsigned char t;
        for (int j = 0; j < h; j++) {
                for (int i = 0; i < w; i++) {
                        size_t r = fread(&t, UCHAR_SIZE, 1, file);
                        if (r != UCHAR_SIZE) {
                                errorEncounter.push_back(1);
                                return;
                        }
                        m[j][i] = new pnmBWColor(t);
                }
        }

        fclose(file);

        correction(gamma, false);

}

void print(string filename) {

        if (!errorEncounter.empty()) {
                return;
        }

        correction(gamma, true);

        FILE* file = fopen(filename.c_str(), "wb");
        if (!(file != NULL)) {
                errorEncounter.push_back(1);
                return;
        }

        fprintf(file, "P5\n");
        fprintf(file, "%i %i\n%i\n", width, height, depth);

        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {

                        unsigned char t = m[j][i]->color;
                        fwrite(&(t), sizeof(unsigned char), 1, file);

                }
        }

        fclose(file);
```

```
	}


	unsigned char colorToCurrentScheme(unsigned char c) {
		pnmBWColor color(c);
		if (gamma == -1) {
			color.srgb(depth, true);
		}
		else {
			color.gamma(gamma, depth, true);
		}
		return color.color;
	}

protected:
	void correction(double g = -1, bool reverse = false) {

		if (!errorEncounter.empty()) {
			return;
		}

		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				if (g == -1) {
					m[i][j]->srgb(depth, reverse);
				}
				else {
					m[i][j]->gamma(g, depth, reverse);
				}
			}
		}

	}


public:

	void drawSingularLine(float x0, float y0, float x1, float y1, int lineColor = 0, float
lineWidth = 1) {

		if (!errorEncounter.empty()) {
			return;
		}

		if (lineWidth <= 1) {
			drawSingularLine(x0, y0, x1, y1, lineColor * lineWidth);
		}
		else {
			if ((x0 == x1) || (y0 == y1)) {
				bresenham(x0, y0, x1, y1, lineColor);
			}
			else {
				wu(x0, y0, x1, y1, lineColor);
			}
		}

	}

	void drawBWPoint(int x, int y, double t, float lineColor = 0) {
		if ((x >= width) || (y >= height) || (x < 0) || (y < 0)) {
			return;
		}
		m[y][x]->setColor(m[y][x]->color + (lineColor - m[y][x]->color) * t);
	}
```

```
void bresenham(int x1, int y1, int x2, int y2, int color) {

	int step = 2 * (y2 - y1);
	int err = step - (x2 - x1);

	for (int x = x1, y = y1; x <= x2; x++) {

		drawBWPoint(x, y, 1, color);
		err += step;

		if (err >= 0) {
			y++;
			err -= 2 * (x2 - x1);
		}
	}

}

int ipart(float x) {
	return int(std::floor(x));
}

float round(float x) {
	return std::round(x);
}

float fpart(float x) {
	return x - std::floor(x);
}

float rfpart(float x) {
	return 1 - fpart(x);
}


void wu(float x0, float y0, float x1, float y1, int lineColor = 0) {


	// Universing the direction
	const bool incline = abs(y1 - y0) > abs(x1 - x0);
	if (incline) {
		std::swap(x0, y0);
		std::swap(x1, y1);
	}
	if (x0 > x1) {
		std::swap(x0, x1);
		std::swap(y0, y1);
	}

	// getting incline
	const float dx = x1 - x0;
	const float dy = y1 - y0;
	const float alpha = (dx == 0) ? 1 : dy / dx;

	int xpx11;
	float intery;
	{

		const float xend = round(x0);
		const float yend = y0 + alpha * (xend - x0);
		const float xgap = rfpart(x0 + 0.5);

		const int ypx11 = ipart(yend);
```

```
                xpx11 = int(xend);

                if (incline) {
                        drawBWPoint(ypx11, xpx11, rfpart(yend) * xgap, lineColor);
                        drawBWPoint(ypx11 + 1, xpx11, fpart(yend) * xgap, lineColor);
                }
                else {
                        drawBWPoint(xpx11, ypx11, rfpart(yend) * xgap, lineColor);
                        drawBWPoint(xpx11, ypx11 + 1, fpart(yend) * xgap, lineColor);
                }

                intery = yend + alpha;
        }

        int xpx12;
        {
                const float xend = round(x1);
                const float yend = y1 + alpha * (xend - x1);
                const float xgap = rfpart(x1 + 0.5);
                xpx12 = int(xend);
                const int ypx12 = ipart(yend);
                if (incline) {
                        drawBWPoint(ypx12, xpx12, rfpart(yend) * xgap, lineColor);
                        drawBWPoint(ypx12 + 1, xpx12, fpart(yend) * xgap, lineColor);
                }
                else {
                        drawBWPoint(xpx12, ypx12, rfpart(yend) * xgap, lineColor);
                        drawBWPoint(xpx12, ypx12 + 1, fpart(yend) * xgap, lineColor);
                }
        }

        if (incline) {
                for (int x = xpx11 + 1; x < xpx12; x++) {
                        drawBWPoint(ipart(intery), x, rfpart(intery), lineColor);
                        drawBWPoint(ipart(intery) + 1, x, fpart(intery), lineColor);
                        intery += alpha;
                }
        }
        else {
                for (int x = xpx11 + 1; x < xpx12; x++) {
                        drawBWPoint(x, ipart(intery), rfpart(intery), lineColor);
                        drawBWPoint(x, ipart(intery) + 1, fpart(intery), lineColor);
                        intery += alpha;
                }
        }

}




};



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

        string fn, in, out;
        unsigned char bw_color = 255;
        float width = 1;
        float x0 = 0, y0 = 0, x1 = 0, y1 = 0;
```

```
	float gamma = -1;


	for (int i = 0; i < argc; i++) {
		if (i == 0) {
			fn = argv[i];
		}
		if (i == 1) {
			in = argv[i];
		}
		if (i == 2) {
			out = argv[i];
		}
		if (i == 3) {
			bw_color = atoi(argv[i]);
		}
		if (i == 4) {
			width = atof(argv[i]);
		}
		if (i == 5) {
			x0 = atof(argv[i]);
		}
		if (i == 6) {
			y0 = atof(argv[i]);
		}
		if (i == 7) {
			x1 = atof(argv[i]);
		}
		if (i == 8) {
			y1 = atof(argv[i]);
		}
		if (i == 9) {
			gamma = atof(argv[i]);
		}
	}

	pnmBWImage im(in, gamma);
	im.drawSingularLine(x0, y0, x1, y1, im.colorToCurrentScheme(bw_color), width);
	im.print(out);

	if (!im.errorEncounter.empty()) {
		cerr << "Some errors encountered.";
		return 1;
	}

	return 0;

}
```