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 № 5

По дисциплине «Компьютерная геометрия и графика»

Изучение алгоритма настройки автояркости изображения

Выполнил студент группы №M3101
Семенов Георгий Витальевич

Преподаватель:
Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ

2020

Цель работы: реализовать программу, которая позволяет проводить настройку автояркости изображения в различных цветовых пространствах.

Описание:

Программа должна быть написана на C/C++ и не использовать внешние библиотеки.

Аргументы передаются через командную строку:

lab5.exe <имя_входного_файла> <имя_выходного_файла> <преобразование> [<смещение> <множитель>], где <преобразование>:

- 0 - применить указанные значения <смещение> и <множитель> в пространстве RGB к каждому каналу;
- 1 - применить указанные значения <смещение> и <множитель> в пространстве YCbCr.601 к каналу Y;
- 2 - автояркость в пространстве RGB: <смещение> и <множитель> вычисляются на основе минимального и максимального значений пикселей;
- 3 - аналогично 2 в пространстве YCbCr.601;
- 4 - автояркость в пространстве RGB: <смещение> и <множитель> вычисляются на основе минимального и максимального значений пикселей, после игнорирования 0.39% самых светлых и тёмных пикселей;
- 5 - аналогично 4 в пространстве YCbCr.601.

<смещение> - целое число, только для преобразований 0 и 1 в диапазоне [-255..255];

<множитель> - дробное положительное число, только для преобразований 0 и 1 в диапазоне [1/255..255].

Значение пикселя X изменяется по формуле: **(X-<смещение>)*<множитель>**.

YCbCr.601 в PC диапазоне: [0, 255]. Входные/выходные данные: PNM P5 или P6 (RGB).

Частичное решение: только преобразования 0-3 + корректно выделяется и освобождается память, закрываются файлы, есть обработка ошибок.

Полное решение: все остальное.

Теоретическая часть

Гистограмма — график статистического распределения элементов цифрового изображения с различной яркостью. Гистограмма является инструментом для работы с контрастом и экспозицией изображения.

Корректировать контрастность можно как работая в пространстве RGB, так и в других цветовых пространствах.
Например, широко используется корректировка контрастности в пространстве YCbCr. Здесь изменяются значения только канала Y, соответствующего яркости изображения, а каналы Cb и Cr остаются неизменными. Как правило, это даёт более контрастные, но менее насыщенные изображения, чем автоконтрастность в пространстве RGB.

Семейство пространств **YCbCr** определяется тремя координатами: Y — яркость цвета (люма), Cb — синяя цветоразносная компонента, Cr — красная цветоразностная компонента. Несмотря на то, что оно далеко от абсолютного цветового пространства, в силу того, что человеческий глаз менее чувствителен к перепадам цвета, пространство позволяет уменьшить поток видеоданных.

Формула, по которой применяется смещение-множитель к каналу:

$$y = (x - min) \cdot \frac{255}{max - min}, \qquad \text{где } min - \text{смещение,}$$

$$\frac{255}{max - min} - \text{множитель}$$

Экспериментальная часть

Язык программирования: C++.

Изображение описано в классе baseImage, хранящее в двумерном массиве (vector) указатели на экземпляры класса baseColor, в котором реализованы методы для работы с цветовыми пространствами.

В методе processLightness() реализованы операции для работы с яркостью изображения. В вспомогательной процедуре processDeltaMltpr() реализовано применение смещения и множителя к данному изображению.

Вывод

Яркость изображений может быть изменена методом применения смещения-множителя. Автояркость позволяет выровнять гистограмму цвета и улучшить восприятие изображения.

Листинг

https://github.com/MrGeorgeous/ComputerGeometryAndGraphics/blob/master/README.md
Файл: Source.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 <iostream>
#include <cmath>
#include <cfenv>

#pragma STDC FENV_ACCESS ON



using namespace std;

const int UCHAR_SIZE = sizeof(unsigned char);

typedef vector<vector<double>> matrix;

matrix matrixYCbCr601 = { {-100,0.0,0.0},{0.0,0.0,0.0},{0.0,0.0,0.0} };
matrix revMatrixYCbCr601 = { {-100,0.0,0.0},{0.0,0.0,0.0},{0.0,0.0,0.0} };

void fillYCbCr(matrix& m, double a, double b, double c) {
        // a - K_r, b - K_g, c - K_b
        m[0][0] = a;
        m[0][1] = b;
        m[0][2] = c;
        m[1][0] = -0.5 * a / (1.0 - c);
        m[1][1] = -0.5 * b / (1.0 - c);
        m[1][2] = 0.5;
        m[2][0] = 0.5;
        m[2][1] = -0.5 * b / (1.0 - a);
        m[2][2] = -0.5 * c / (1.0 - a);
}

void fillRevYCbCr(matrix& m, double a, double b, double c) {
        // a - K_r, b - K_g, c - K_b
        m[0][0] = 1.0;
        m[0][1] = 0;
        m[0][2] = 2.0 - 2.0 * a;
        m[1][0] = 1.0;
        m[1][1] = -c / b * (2.0 - 2.0 * c);
        m[1][2] = -a / b * (2.0 - 2.0 * a);
        m[2][0] = 1.0;
        m[2][1] = 2.0 - 2.0 * c;
        m[2][2] = 0;
}
```

```
class baseColor {

public:

	double red = 1; // 0 ... 1
	double green = 1; // 0 ... 1
	double blue = 1; // 0 ... 1

	baseColor() {

	}

	baseColor(baseColor * c) {
		red = c->red;
		green = c->green;
		blue = c->blue;
	}

	baseColor(double r, double g, double b) : red(r), green(g), blue(b) {

	}

	void matrixMultiply(matrix& m) {
		if (m[0][0] == -100) {
			cout << "Fatality.";
		}

		double r = m[0][0] * this->red + m[0][1] * this->green + m[0][2] * this-
>blue;
		double g = m[1][0] * this->red + m[1][1] * this->green + m[1][2] * this-
>blue;
		double b = m[2][0] * this->red + m[2][1] * this->green + m[2][2] * this-
>blue;

		this->red = r;
		this->green = g;
		this->blue = b;
	}

	void fromRGBtoYCbCr601() {
		matrixMultiply(matrixYCbCr601);
		this->green += 128;
		this->blue += 128;
	}

	void fromYCbCr601toRGB() {
		this->green -= 128;
		this->blue -= 128;
		matrixMultiply(revMatrixYCbCr601);
	}

	//void red(double)

};


typedef vector<vector<double>> doubleMatrix;
typedef vector<vector<baseColor*>> pnmMatrix;
typedef vector<char> chars;

enum palette {
	RGB,
	HSL,
	HSV,
	YCbCr601,
	YCbCr709,
	YCoCg,
	CMY,
	NO_PALETTE
};
```

```
class baseImage {

public:

        int width = 0; // y
        int height = 0; // x
        palette colorSpace = RGB;

        bool bw = false;

        pnmMatrix m;
        chars errorEncounter;


        baseImage(size_t w, size_t h, baseColor color = baseColor()) {

                m = pnmMatrix(h, vector<baseColor*>(w, nullptr));
                for (int i = 0; i < height; i++) {
                        for (int j = 0; j < width; j++) {
                                m[i][j] = new baseColor(color);
                        }
                }

                bw = false;
                width = w;
                height = h;

        }

        ~baseImage() {

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

        }

        bool loadChannelsFromFile(string filename) {

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

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

                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);


                if (((w <= 0) || (h <= 0)) || (p1 != 'P') || (!((p2 == '5') || (p2 == '6')))
|| !(d == 255)) {
                        errorEncounter.push_back(1);
                        return false;
                }


                {
                        width = w;
```

```
                height = h;
                m = pnmMatrix(h, vector<baseColor*>(w, nullptr));
                for (int i = 0; i < height; i++) {
                        for (int j = 0; j < width; j++) {
                                m[i][j] = new baseColor();
                        }
                }
                bw = false;
        }

        unsigned char t;
        unsigned char r, g, b;

        for (int j = 0; j < min(h, height); j++) {
                for (int i = 0; i < min(w, width); i++) {

                        if (p2 == '6') {

                                bw = false;

                                size_t res = 0;
                                res += fread(&r, UCHAR_SIZE, 1, file);
                                res += fread(&g, UCHAR_SIZE, 1, file);
                                res += fread(&b, UCHAR_SIZE, 1, file);

                                if (res != 3 * UCHAR_SIZE) {
                                        errorEncounter.push_back(1);
                                        return false;
                                }

                                m[j][i]->red = double(r);
                                m[j][i]->green = double(g);
                                m[j][i]->blue = double(b);

                        } else {

                                bw = true;

                                size_t r = fread(&t, UCHAR_SIZE, 1, file);
                                if (r != UCHAR_SIZE) {
                                        errorEncounter.push_back(1);
                                        return false;
                                }

                                m[j][i]->red = double(t);
                                m[j][i]->green = double(t);
                                m[j][i]->blue = double(t);

                        }

                }

        }

        fclose(file);
        return true;

}


baseImage(string filename) {

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

        if (!loadChannelsFromFile(filename)) {
                return;
        }
```

```
	}


	bool writeChannels(string filename) {

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

		unsigned char channelDepth = 255;

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

		if (bw) {

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

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

					t = max(0, min(255, int(m[j][i]->red)));
					fwrite(&t, sizeof(unsigned char), 1, file);

				}
			}

		}
		else {

			fprintf(file, "P6\n");
			fprintf(file, "%i %i\n%i\n", width, height, channelDepth);

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

					unsigned char r = max(0, min(255, int(m[j][i]-
>red)));
					unsigned char g = max(0, min(255, int(m[j][i]-
>green)));
					unsigned char b = max(0, min(255, int(m[j][i]-
>blue)));

					fwrite(&r, sizeof(unsigned char), 1, file);
					fwrite(&g, sizeof(unsigned char), 1, file);
					fwrite(&b, sizeof(unsigned char), 1, file);

				}
			}

		}

		fclose(file);
		return true;

	}


	void processDeltaMtlprRed(double delta, double mltpr) {
		for (int j = 0; j < height; j++) {
			for (int i = 0; i < width; i++) {
				m[j][i]->red = (m[j][i]->red - delta) * mltpr;
			}
		}
	}
```

```
void processDeltaMtlprGreen(double delta, double mltpr) {
        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {
                        m[j][i]->green = (m[j][i]->green - delta) * mltpr;
                }
        }
}

void processDeltaMtlprBlue(double delta, double mltpr) {
        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {
                        m[j][i]->blue = (m[j][i]->blue - delta) * mltpr;
                }
        }
}

void processDeltaMtlpr(double delta, double mltpr) {

        processDeltaMtlprRed(delta, mltpr);

        if (!bw) {
                processDeltaMtlprGreen(delta, mltpr);
                processDeltaMtlprBlue(delta, mltpr);
        }

}

void RGBtoYCbCr() {
        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {
                        m[j][i]->fromRGBtoYCbCr601();
                }
        }
}

void YCbCrtoRGB() {
        for (int j = 0; j < height; j++) {
                for (int i = 0; i < width; i++) {
                        m[j][i]->fromYCbCr601toRGB();
                }
        }
}

std::pair<size_t, size_t> getLimits(size_t(&count)[256]) {
        const int limit = (0.0039 * double(width) * double(height));

        size_t bound_min = 0;
        size_t count_min = 0;

        for (size_t i = 0; i < 256; i++) {
                count_min += count[i];
                bound_min = i;
                if (count_min >= limit) {
                        break;
                }
        }

        size_t bound_max = 0;
        size_t count_max = 0;

        for (size_t i = 255; i >=0; i--) {
                count_max += count[i];
                bound_max = i;
                if (count_max >= limit) {
                        break;
                }
        }

        return std::pair<size_t, size_t>(bound_min, bound_max);

}
```

```
	void processLightness(size_t operation, double delta, double mltpr) {

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

		fillRevYCbCr(revMatrixYCbCr601, 0.299, 0.587, 0.114);
		fillYCbCr(matrixYCbCr601, 0.299, 0.587, 0.114);

		if (operation == 0) {
			processDeltaMtlpr(delta, mltpr);
		}

		if (operation == 1) {
			RGBtoYCbCr();
			processDeltaMtlprRed(delta, mltpr);
			YCbCrtoRGB();
		}


		if (operation == 2) {

			double min_r = 255, max_r = 0;
			double min_g = 255, max_g = 0;
			double min_b = 255, max_b = 0;

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

					min_r = min(min_r, m[j][i]->red);
					max_r = max(max_r, m[j][i]->red);

				}
			}

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

						min_g = min(min_g, m[j][i]->green);
						max_g = max(max_g, m[j][i]->green);
						min_b = min(min_b, m[j][i]->blue);
						max_b = max(max_b, m[j][i]->blue);

					}
				}
			}

			if (!bw) {
				int min_ch = min(min_r, min(min_g, min_b));
				int max_ch = max(max_r, max(max_g, max_b));
				processDeltaMtlpr(min_ch, 255.0 / (max_ch - min_ch));
				cout << min_ch << " " << std::setprecision(3) << 255.0 /
(max_ch - min_ch) << "\n";

			} else {
				processDeltaMtlpr(min_r, 255.0 / (max_r - min_r));
				cout << min_r << " " << std::setprecision(3) << 255.0 /
(max_r - min_r) << "\n";
			}

		}

		if (operation == 3) {

			RGBtoYCbCr();

			double min_r = 255, max_r = 0;

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

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

                                        min_r = min(min_r, m[j][i]->red);
                                        max_r = max(max_r, m[j][i]->red);

                                }
                        }

                        processDeltaMtlprRed(min_r, 255.0 / (max_r - min_r));

                        cout << min_r << " " << std::setprecision(3) << 255.0 / (max_r -
min_r) << "\n";

                        YCbCrtoRGB();

                }


                if (operation == 4) {

                        size_t count_r[256];
                        size_t count_g[256];
                        size_t count_b[256];

                        for (int j = 0; j < 256; j++) {
                                count_r[j] = 0;
                                count_g[j] = 0;
                                count_b[j] = 0;
                        }

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

                                        count_r[(unsigned char)(m[j][i]->red)]++;

                                }
                        }

                        if (!bw) {
                                for (int j = 0; j < height; j++) {
                                        for (int i = 0; i < width; i++) {
                                                count_g[(unsigned char)(m[j][i]->green)]++;
                                                count_b[(unsigned char)(m[j][i]->blue)]++;
                                        }
                                }
                        }

                        if (!bw) {

                                auto limits_r = getLimits(count_r);
                                auto limits_g = getLimits(count_g);
                                auto limits_b = getLimits(count_b);

                                int min_ch = min(limits_r.first, min(limits_g.first,
limits_b.first));
                                int max_ch = max(limits_r.second, max(limits_g.second,
limits_b.second));

                                processDeltaMtlprRed(min_ch, 255.0 / (max_ch - min_ch));
                                processDeltaMtlprGreen(min_ch, 255.0 / (max_ch - min_ch));
                                processDeltaMtlprBlue(min_ch, 255.0 / (max_ch - min_ch));

                                cout << min_ch << " " << std::setprecision(3) << 255.0 /
(max_ch - min_ch) << "\n";

                        } else {

                                auto limits_r = getLimits(count_r);

                                int min_ch = limits_r.first;
                                int max_ch = limits_r.second;
```

```
                                processDeltaMtlprRed(min_ch, 255.0 / (max_ch - min_ch));

                                cout << min_ch << " " << std::setprecision(3) << 255.0 /
(max_ch - min_ch) << "\n";

                        }


                }

                if (operation == 5) {

                        RGBtoYCbCr();

                        size_t count_r[256];
                        for (int j = 0; j < 256; j++) {
                                count_r[j] = 0;
                        }

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

                                        count_r[(unsigned char)(m[j][i]->red)]++;

                                }
                        }

                        auto limits_r = getLimits(count_r);

                        processDeltaMtlprRed(limits_r.first, 255.0 / (limits_r.second -
limits_r.first));

                        cout << limits_r.first << " " << std::setprecision(3) << 255.0 /
(limits_r.second - limits_r.first) << "\n";

                        YCbCrtoRGB();

                }


        }
};







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

        string in = "";
        string out = "";
        size_t operation = -1;
        double delta = 0;
        double mltpr = 1;

        for (int i = 0; i < argc; i++) {

                if (i == 0) {
                        continue;
                }
                if (i == 1) {
                        in = argv[i];
                }
                if (i == 2) {
```

```
                        out = argv[i];
                }
                if (i == 3) {
                        operation = atoi(argv[i]);
                }
                if (i == 4) {
                        delta = atof(argv[i]);
                }
                if (i == 5) {
                        mltpr = atof(argv[i]);
                }

        }

        if (operation == -1) {
                cerr << "Operation was not specified.";
                return 1;
        }

        //for (size_t i = 0; i < 6; i++) {
        //      baseImage im(in);
        //      im.processLightness(i, delta, mltpr);
        //      im.writeChannels(to_string(i) + out);
        //      if (!im.errorEncounter.empty()) {
        //              cerr << "Some errors encountered.";
        //              return 1;
        //      }
        //}

        size_t i = operation;
        /*for (size_t i = 0; i < 6; i++) {*/
                baseImage im(in);
                im.processLightness(i, delta, mltpr);
                im.writeChannels(/*to_string(i) +*/ out);
                if (!im.errorEncounter.empty()) {
                        cerr << "Some errors encountered.";
                        return 1;
                }
        //}



        return 0;

}
```