МИНИСТЕРСТВО ВЫСШЕГО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное образовательное учреждение высшего образования

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

Мегафакультет трансляционных информационных технологий

Факультет информационных технологий и программирования

Отчет по лабораторной работе №4

По дисциплине «Компьютерная геометрия и графика»

Изучение цветовых пространств

Выполнил студент группы №M3101:

Пантелеев Ярослав Кириллович

Преподаватель:

Скаков Павел Сергеевич

САНКТ-ПЕТЕРБУРГ

2020

Цель работы: реализовать программу, которая позволяет проводить преобразования между цветовыми пространствами.

Входные и выходные данные могут быть как одним файлом ppm, так и набором из 3 pgm.

Описание:

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

Аргументы передаются через командную строку:

lab4.exe -f <from_color_space> -t <to_color_space> -i <count> <input_file_name> -o <count> <output_file_name>,

где

- <color_space> - RGB / HSL / HSV / YCbCr.601 / YCbCr.709 / YCoCg / CMY
- <count> - 1 или 3
- <file_name>:
    - для count=1 просто имя файла; формат ppm
    - для count=3 шаблон имени вида <name.ext>, что соответствует файлам <name_1.ext>, <name_2.ext> и <name_3.ext> для каждого канала соответственно; формат pgm

Порядок аргументов (-f, -t, -i, -o) может быть произвольным.

Везде 8-битные данные и полный диапазон (**0..255, PC range**).

Полное решение: всё работает + корректно выделяется и освобождается память, закрываются файлы, есть обработка ошибок.

/* да, частичного решения нет */

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:

C: fprintf(stderr, "Error\n");

C++: std::cerr

Следующие параметры гарантировано не будут выходить за обусловленные значения:

- <count> = 1 или 3;
- width и height в файле - положительные целые значения;
- яркостных данных в файле ровно width * height;

Теоретическая часть:

Цветовые пространства

Цветовые пространства бывают аддитивные (например, RGB) и субтрактивные (например, CMY). В аддитивных пространствах 0 соответствует чёрному цвету, а 100% всех компонент – белому. Это отражает работу источников света, например, отображение информации на мониторе. В субтрактивных наоборот: отсутствие компонент – это белый, а полное присутствие – чёрный. Это соответствует смешению красок на бумаге

Пространство RGB

Пространство RGB – это самое широко используемое цветовое пространство. Его компоненты примерно соответствуют трём видам наших цветовых рецепторов: L, M, S.

R (Red) – красный
G (Green) – зелёный
B (Blue) – синий

Типичный диапазон значений: 0..255 для каждой компоненты, но возможны и другие значения, например, 0..1023 для 10-битных данных.

Пространства HSL и HSV

Пространства HSL (другие названия: HLS, HSI) и HSV (другое название: HSB) широко используются в интерфейсах выбора цвета. Предназначены для “интуитивно понятного” изменения таких характеристик цвета как: оттенок, насыщенность, яркость.

H (Hue) – оттенок: диапазон 0..360°, 0..100 или 0..1
S (Saturation) – насыщенность: 0..100 или 0..1
L/I (Lightness/Intensity) – “светлота”: 0..100 или 0..1
V/B (Value/Brightness) – “яркость”: 0..100 или 0..1

Пространство YUV / YCbCr

Пространство YUV (другое название: YCbCr) крайне широко используется для обработки и хранения графической и видео информации. Отдельные компоненты примерно соответствуют разложению нашей зрительной системой информации о цвете на яркость и две цветоразницы.

Y – яркость
U/Cb – цветоразность “хроматический синий”
V/Cr – цветоразность “хроматический красный”

Пространство YCgCo

Пространство YCgCo – недавно разработанная альтернатива YCbCr. Те же принципы, но более простое преобразование в/из RGB.

Y – яркость

Cg – цветоразность “хроматический зелёный”

Co – цветоразность “хроматический оранжевый”

Пространство CMY/CMYK

Пространства CMY и CMYK соответствуют устройству цветных принтеров. CMYK для улучшения эффективности использования красок добавляет компонент, соответствующий чёрной краске: без него получение широко востребованного чёрного, требует смешивания всех трёх красок.

C (Cyan) – голубой

M (Magenta) – пурпурный

Y(Yellow) – жёлтый

K (blacK) – чёрный

Модель зрения человека

Представление и обработка графической информации в вычислительных системах основаны на наших знаниях о модели зрения человека. Не только регистрация и отображение изображений стараются соответствовать системе зрения человека, но и алгоритмы кодирования и сжатия данных становятся намного эффективнее при учёте того, что видит и не видит человек.

Согласно современным представлениям, система зрения человека имеет 4 вида рецепторов:

- 3 вида “колбочек”: S (short), M (medium), L (long), отвечающих за цветное зрение. Работают только при высокой освещённости.
- 1 вид “палочек”: R (rods), позволяющих регистрировать яркость. Работают только при низкой освещённости.

Система цветного зрения человека трёхкомпонентная: воспринимаемый цвет описывается тремя значениями. Любые спектры излучения, приводящие к одинаковым этим трём значениям, неразличимы для человека. Регистрация спектра L, M и S рецепторами лежит в основе цветовой модели RGB, описывающей цвет как комбинацию красного, синего и зелёного. Однако, M и L рецепторы чувствительны далеко не только к чистым “зелёному” и “красному” цветам, а воспринимают довольно широкие спектры, которые ещё и значительно перекрываются. При

непосредственном восприятии S, M, L значений было бы очень трудно различать красно-зелёные оттенки.

Но система зрения человека решила эту проблему тем, что производится “предварительная обработка”. SML сигнал (что условно соответствует RGB) преобразуется следующим образом:

$$Y = S + M + L$$

$$A = L - M$$

$$B = (L + M) – S$$

То есть, представление красный-зелёный-синий превращается в яркость (Y) и две цветоразницы: красно-зелёную (A) и жёлто-синюю (B).В мозг передаётся обработанный сигнал: YAB. Кроме того, количество нейронов для компонент Y, A, B различна: о яркости передаётся гораздо больше информации, чем о цветоразностях.

Всё это послужило основой для различных цветоразностных систем представления цвета, например, YUV (альтернативное название: YCbCr), широко используемых при эффективном кодировании и сжатии графической информации.

Экспериментальная часть

Язык выполнения работы: C++17, компилятор Microsoft Visual C++.

Этапы работы программы:

1) Чтение файла картинки (хедера и информации о цвете каждого пикселя) с обработкой ошибок чтения (из 1 файла или из 3 файлов)
2) Выполнение заданного аргументом командной строки преобразования между цветовыми пространствами (из *from* в *to*)
3) Запись полученного изображения в выходной файл (из 1 файла или из 3 файлов)

Вывод:

Выполнение данной лабораторной работы позволило узнать о существовании различных цветовых пространств, об их особенностях, сферах применения и об алгоритмах преобразования изображения для перехода из одного цветового пространства в другое.

Листинг исходного кода:

main.cpp

```
1.  #include <iostream>
2.  #include <string>
3.  #include <vector>
4.  #include "Image.h"
5.
6.  int main(int argc, char* argv[]) {
7.      // Îáðàáîòêà àðãóìåíòîâ êîìàíäíîé ñòðîêè
8.      std::string from, to, input, output;
9.      int input_count = 0;
10.     int output_count = 0;
11.     for(int i = 1; i < argc; i++) {
12.         if(std::string(argv[i]) == "-f") {
13.             from = std::string(argv[i + 1]);
14.         }
15.         if(std::string(argv[i]) == "-t") {
16.             to = std::string(argv[i + 1]);
17.         }
18.         if(std::string(argv[i]) == "-i") {
19.             input_count = atoi(argv[i + 1]);
20.             input = std::string(argv[i + 2]);
21.         }
22.         if(std::string(argv[i]) == "-o") {
23.             output_count = atoi(argv[i + 1]);
24.             output = std::string(argv[i + 2]);
25.         }
26.     }
27.
28.     // Ïðîâåðêà êîððåêòíîñòè âñåõ àðãóìåíòîâ êîìàíäíîé ñòðîêè
29.     bool input_format_is_correct = false;
30.     bool output_format_is_correct = false;
31.     // Íîìåðà ôîðìàòîâ (óêàçàíû â êëàññå Pixel):
1     2     3        4           5          6       7
32.     std::vector <std::string> color_spaces = { "RGB", "HSL", "HSV", "YCbCr.601", "YCbCr.709", "
YCoCg", "CMY" };
33.     for (const auto& color_space : color_spaces) {
34.         if(color_space == from)
35.             input_format_is_correct = true;
36.         if(color_space == to)
37.             output_format_is_correct = true;
38.     }
39.     if (!input_format_is_correct ||
40.         !output_format_is_correct ||
41.         argc != 11 ||
42.         from.empty() || to.empty() || input.empty() || output.empty() ||
43.         input_count != 1 && input_count != 3 ||
44.         output_count != 1 && output_count != 3)
45.     {
46.         std::cerr << "command line arguments are invalid" << "\n";
47.         return 1;
48.     }
49.
50.     // Ðàáîòà ñ îáúåêòîì êëàññà Image
51.     Image* img;
52.     try {
53.         img = new Image(input, input_count);
54.     }
55.     catch(const std::exception& error) {
56.         std::cerr << error.what() << '\n';
57.         return 1;
58.     }
59.
60.     img->change_color_space(from, to);
61.     try {
```

```
62.         img->write(output, output_count);
63.     }
64.     catch(const std::exception& error) {
65.         std::cerr << error.what() << '\n';
66.         delete img;
67.         return 1;
68.     }
69.
70.     delete img;
71.     return 0;
72. }
```

Image.h

```
1.  #pragma once
2.  #include <vector>
3.  #include <fstream>
4.
5.  class Pixel {
6.  public:
7.      unsigned char a, b, c;
8.      Pixel(unsigned char, unsigned char, unsigned char);
9.      Pixel() = default;
10.     void RGB_to_HSL();       // 1 -> 2
11.     void HSL_to_RGB();       // 2 -> 1
12.     void RGB_to_HSV();       // 1 -> 3
13.     void HSV_to_RGB();       // 3 -> 1
14.     void RGB_to_YCbCr_601(); // 1 -> 4
15.     void YCbCr_601_to_RGB(); // 4 -> 1
16.     void RGB_to_YCbCr_709(); // 1 -> 5
17.     void YCbCr_709_to_RGB(); // 5 -> 1
18.     void RGB_to_YCoCg();     // 1 -> 6
19.     void YCoCg_to_RGB();     // 6 -> 1
20.     void RGB_to_CMY();       // 1 -> 7
21.     void CMY_to_RGB();       // 7 -> 1
22. };
23.
24. class Image {
25. public:
26.     Image(const std::string&, int);
27.     void write(std::string, int);
28.     void change_color_space(std::string, std::string);
29. private:
30.     int width;
31.     int height;
32.     int color_depth;
33.     std::vector <std::vector <Pixel>> image;
34. };
```

Image.cpp

```
1.  #include "Image.h"
2.  #include <algorithm>
3.  #include <stdexcept>
4.  #include <cmath>
5.  #include <string>
6.  #include <vector>
7.  #include <fstream>
8.
9.  // Ïàðñèì íàçâàíèå øàáëîíà îáëàñ÷èòü íàçâàíèÿ òðåõ îáëîâ
10. std::vector <std::string> parse_pattern(std::string pattern) {
11.     int separator_index = -1;
12.     for (size_t i = 0; i < pattern.size(); i++)
13.         if (pattern[i] == '.')
14.             separator_index = i;
15.     if(separator_index == -1)
```

```
        throw std::runtime_error("pattern has no dots, so doesn't match name.ext");

    const std::string name = pattern.substr(0, separator_index);
    const std::string format = pattern.substr(separator_index);
    return { name + "_1" + format,
             name + "_2" + format,
             name + "_3" + format };
}

// Êîíñòðóêòîð
Pixel::Pixel(unsigned char a, unsigned char b, unsigned char c) : a(a), b(b), c(c) {}

// Êîíñòðóêòîð
Image::Image(const std::string& filename, int count) {
    if(count == 1) {
        // Îòêðûâàåì ôàéë
        std::ifstream fin(filename, std::ios::binary);
        if (!fin.is_open())
            throw std::runtime_error("failed to open file");

        // ×èòàåì õåäåð
        char ch[2];
        fin >> ch[0] >> ch[1];
        if (ch[0] != 'P' || ch[1] != '6')
            throw std::runtime_error("expected P6 format");
        fin >> this->width >> this->height >> this->color_depth;
        if(this->color_depth != 255)
            throw std::runtime_error("only 255 color depth is supported");
        this->image.assign(this->height, std::vector <Pixel>(this->width));

        // ×èòàåì ïèêñåëè
        char color;
        fin.read(&color, 1);
        for (int i = 0; i < this->height; i++) {
            for (int j = 0; j < this->width; j++) {
                this->image[i][j] = Pixel();

                fin.read(&color, sizeof(unsigned char));
                this->image[i][j].a = color;

                fin.read(&color, sizeof(unsigned char));
                this->image[i][j].b = color;

                fin.read(&color, sizeof(unsigned char));
                this->image[i][j].c = color;
            }
        }
        fin.close();
    }
    else {
        // Íàõîäèì íàçâàíèÿ âõîäíûõ ôàéëîâ
        std::vector <std::string> files = parse_pattern(filename);
        if(files.size() != 3)
            throw std::runtime_error("unknown error during name generation");

        for(int k = 0; k < 3; k++) {
            // Îòêðûâàåì ôàéë
            std::ifstream fin(files[k], std::ios::binary);
            if(!fin.is_open())
                throw std::runtime_error("failed to open file " + files[k]);

            // ×èòàåì õåäåð
            char ch[2];
            fin >> ch[0] >> ch[1];
            if (ch[0] != 'P' || ch[1] != '5')
                throw std::runtime_error("expected P5 format");
            fin >> this->width >> this->height >> this->color_depth;
            if (this->color_depth != 255)
```

```
                throw std::runtime_error("only 255 color depth is supported");
            if (k == 0) {
                this->image.assign(this->height, std::vector <Pixel>(this->width));
            }
            // ×èòàåì ïèêñåëè
            char color;
            fin.read(&color, 1);
            for (int i = 0; i < this->height; i++) {
                for (int j = 0; j < this->width; j++) {
                    fin.read(&color, sizeof(unsigned char));
                    if (k == 0)
                        this->image[i][j].a = color;
                    if (k == 1)
                        this->image[i][j].b = color;
                    if (k == 2)
                        this->image[i][j].c = color;
                    // fin.read(&color, sizeof(unsigned char));
                    // fin.read(&color, sizeof(unsigned char));
                }
            }
            fin.close();
        }
    }
}

// Çàïèñü èçîáðàæåíèÿ â ôàéë(-û)
void Image::write(std::string filename, int count) {
    if(count == 1) {
        std::ofstream fout(filename, std::ios::binary);
        if(!fout.is_open()) {
            throw std::runtime_error("cannot open output file");
        }

        fout << "P6\n" << width << ' ' << height << '\n' << color_depth << '\n';
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                fout << image[i][j].a;
                fout << image[i][j].b;
                fout << image[i][j].c;
            }
        }
        fout.flush();
        fout.close();
    } else {
        std::vector<std::string> files = parse_pattern(filename);
        if(files.size() != 3)
            throw std::runtime_error("unknown error during name generation");

        for(int k = 0; k < 3; k++) {
            std::ofstream fout(files[k], std::ios::binary);
            if(!fout.is_open()) {
                throw std::runtime_error("cannot open output file " + files[k]);
            }

            fout << "P5\n" << width << ' ' << height << '\n' << color_depth << '\n';
            if (k == 0) {
                for (int i = 0; i < height; i++) {
                    for (int j = 0; j < width; j++) {
                        fout << image[i][j].a;
                        // fout << image[i][j].a;
                        // fout << image[i][j].a;
                    }
                }
            }
            else if (k == 1) {
                for (int i = 0; i < height; i++) {
                    for (int j = 0; j < width; j++) {
                        fout << image[i][j].b;
```

```
                        // fout << image[i][j].b;
                        // fout << image[i][j].b;
                    }
                }
            }
            else if (k == 2) {
                for (int i = 0; i < height; i++) {
                    for (int j = 0; j < width; j++) {
                        fout << image[i][j].c;
                        // fout << image[i][j].c;
                        // fout << image[i][j].c;
                    }
                }
            }
            fout.flush();
            fout.close();
        }
    }

}

void Image::change_color_space(std::string from, std::string to) {
    // Îïåðàöèÿ èç from â RGB
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (from == "HSL")
                image[i][j].HSL_to_RGB();
            if (from == "HSV")
                image[i][j].HSV_to_RGB();
            if (from == "YCbCr.601")
                image[i][j].YCbCr_601_to_RGB();
            if (from == "YCbCr.709")
                image[i][j].YCbCr_709_to_RGB();
            if (from == "YCoCg")
                image[i][j].YCoCg_to_RGB();
            if (from == "CMY")
                image[i][j].CMY_to_RGB();
        }
    }

    // Îïåðàöèÿ èç RGB â to
    for (int i = 0; i < height; i++) {
        for (int j = 0; j < width; j++) {
            if (to == "HSL")
                image[i][j].RGB_to_HSL();
            if (to == "HSV")
                image[i][j].RGB_to_HSV();
            if (to == "YCbCr.601")
                image[i][j].RGB_to_YCbCr_601();
            if (to == "YCbCr.709")
                image[i][j].RGB_to_YCbCr_709();
            if (to == "YCoCg")
                image[i][j].RGB_to_YCoCg();
            if (to == "CMY")
                image[i][j].RGB_to_CMY();
        }
    }
}

void Pixel::HSL_to_RGB() {
    // H S L
    const double h = a / 255.0;
    const double s = b / 255.0;
    const double l = c / 255.0;

    double q;
    if (l < 0.5)
        q = l * (s + 1.0);
```

```
220.     else
221.         q = l + s - (l * s);
222.     const double p = l * 2 - q;
223.
224.     // R
225.     double r;
226.     double tr = h + 1.0 / 3.0;
227.     if (tr < 0)
228.         tr += 1.0;
229.     else if (tr > 1.0)
230.         tr -= 1.0;
231.     if (tr < 1.0 / 6.0)
232.         r = p + (q - p) * 6.0 * tr;
233.     else if (tr >= 1.0 / 6.0 && tr < 0.5)
234.         r = q;
235.     else if (tr >= 0.5 && tr < 2.0 / 3.0)
236.         r = p + (q - p) * (2.0 / 3.0 - tr) * 6.0;
237.     else
238.         r = p;
239.
240.     // G
241.     double g;
242.     double tg = h;
243.     if (tg < 0)
244.         tg += 1.0;
245.     else if (tg > 1.0)
246.         tg -= 1.0;
247.     if (tg < 1.0 / 6.0)
248.         g = p + (q - p) * 6.0 * tg;
249.     else if (tg >= 1.0 / 6.0 && tg < 0.5)
250.         g = q;
251.     else if (tg >= 0.5 && tg < 2.0 / 3.0)
252.         g = p + (q - p) * (2.0 / 3.0 - tg) * 6.0;
253.     else
254.         g = p;
255.
256.     // B
257.     double b;
258.     double tb = h - 1.0 / 3.0;
259.     if (tb < 0)
260.         tb += 1.0;
261.     else if (tb > 1.0)
262.         tb -= 1.0;
263.     if(tb < 1.0 / 6.0)
264.         b = p + (q - p) * 6.0 * tb;
265.     else if(tb >= 1.0 / 6.0 && tb < 0.5)
266.         b = q;
267.     else if(tb >= 0.5 && tb < 2.0 / 3.0)
268.         b = p + (q - p) * (2.0 / 3.0 - tb) * 6.0;
269.     else
270.         b = p;
271.
272.     // Output
273.     this->a = static_cast<unsigned char>(r * 255);
274.     this->b = static_cast<unsigned char>(g * 255);
275.     this->c = static_cast<unsigned char>(b * 255);
276. }
277.
278. void Pixel::RGB_to_HSL() {
279.     // R G B
280.     const double r = a * 1.0 / 255;
281.     const double g = b * 1.0 / 255;
282.     const double b = c * 1.0 / 255;
283.     const double max = std::max(r, std::max(g, b));
284.     const double min = std::min(r, std::min(g, b));
285.
286.     // H S L
287.     double h;
```

```
288.     if(max == min)
289.         h = 0;
290.     else if(r == max && g >= b)
291.         h = (g - b) / (max - min) * 60;
292.     else if(r == max && g < b)
293.         h = (g - b) / (max - min) * 60 + 360;
294.     else if(max == g)
295.         h = (b - r) / (max-min) * 60 + 120;
296.     else
297.         h = (r - g) / (max-min) * 60 + 240;
298.     const double l = (max + min) / 2;
299.     const double s = (max - min) / (1 - abs(1 - (max + min)));
300.
301.     // Output
302.     this->a = static_cast<unsigned char>(round(h * 255 / 360));
303.     this->b = static_cast<unsigned char>(round(s * 255));
304.     this->c = static_cast<unsigned char>(round(l * 255));
305. }
306.
307. void Pixel::HSV_to_RGB() {
308.     // H S V
309.     const double h = a / 255.0 * 360.0;
310.     const double s = b / 255.0;
311.     const double v = c / 255.0;
312.
313.     const double c = v * s;
314.     const double x = c * (1 - abs((static_cast<int>(h / 60)) % 2 + (h / 60 - static_cast<int>(h
/ 60)) - 1));
315.     const double m = v - c;
316.
317.     // R G B
318.     double r, g, b;
319.     if(h >= 0 && h <= 60)
320.         r = c, g = x, b = 0;
321.     else if(h >= 60 && h <= 120)
322.         r = x, g = c, b = 0;
323.     else if(h >= 120 && h <= 180)
324.         r = 0, g = c, b = x;
325.     else if(h >= 180 && h <= 240)
326.         r = 0, g = x, b = c;
327.     else if(h >= 240 && h <= 300)
328.         r = x, g = 0, b = c;
329.     else
330.         r = c, g = 0, b = x;
331.
332.     // Output
333.     this->a = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>((r + m) * 255))));
334.     this->b = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>((g + m) * 255))));
335.     this->c = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>((b + m) * 255))));
336. }
337.
338. void Pixel::RGB_to_HSV() {
339.     // R G B
340.     const double r = a * 1.0 / 255;
341.     const double g = b * 1.0 / 255;
342.     const double b = c * 1.0 / 255;
343.
344.     // H S V
345.     const double v = std::max(r, std::max(g, b));
346.     const double diff = v - std::min(r, std::min(g, b));
347.     double h;
348.     if(diff == 0)
349.         h = 0;
350.     else if(v == r)
351.         h = (g - b) / diff;
```

```
352.     else if(v == g)
353.         h = 2 + (b - r) / diff;
354.     else
355.         h = 4 + (r - g) / diff;
356.     h *= 60.0;
357.     if(h < 0)
358.         h += 360;
359.     const double s = (v == 0 ? 0 : diff / v);
360.
361.     // Output
362.     this->a = static_cast<unsigned char>(round(h / 360.0 * 255.0));
363.     this->b = static_cast<unsigned char>(round(s * 255.0));
364.     this->c = static_cast<unsigned char>(round(v * 255.0));
365. }
366.
367. void Pixel::CMY_to_RGB() {
368.     this->a = static_cast<unsigned char>(255 - a);
369.     this->b = static_cast<unsigned char>(255 - b);
370.     this->c = static_cast<unsigned char>(255 - c);
371. }
372.
373. void Pixel::RGB_to_CMY() {
374.     this->a = static_cast<unsigned char>(255 - a);
375.     this->b = static_cast<unsigned char>(255 - b);
376.     this->c = static_cast<unsigned char>(255 - c);
377. }
378.
379. void Pixel::RGB_to_YCoCg() {
380.     // R G B
381.     const double r = a * 1.0 / 255;
382.     const double g = b * 1.0 / 255;
383.     const double b = c * 1.0 / 255;
384.
385.     // Y Co Cg
386.     double y = std::min(1.0, std::max(0.0, static_cast<double>(r / 4 + g / 2 + b / 4)));
387.     double co = std::min(1.0, std::max(0.0, static_cast<double>(r / 2 - b / 2 + 0.5)));
388.     double cg = std::min(1.0, std::max(0.0, static_cast<double>(-
r / 4 + g / 2 - b / 4 + 0.5)));
389.
390.     // Output
391.     this->a = static_cast<unsigned char>(y * 255);
392.     this->b = static_cast<unsigned char>(co * 255);
393.     this->c = static_cast<unsigned char>(cg * 255);
394. }
395.
396. void Pixel::YCoCg_to_RGB() {
397.     // Y Co Cg
398.     const double y = a * 1.0 / 255;
399.     const double co = b * 1.0 / 255 - 0.5;
400.     const double cg = c * 1.0 / 255 - 0.5;
401.
402.     // R G B
403.     const double r = std::min(1.0, std::max(0.0, static_cast<double>(y + co - cg)));
404.     const double g = std::min(1.0, std::max(0.0, static_cast<double>(y + cg)));
405.     const double b = std::min(1.0, std::max(0.0, static_cast<double>(y - co - cg)));
406.
407.     // Output
408.     this->a = static_cast<unsigned char>(r * 255);
409.     this->b = static_cast<unsigned char>(g * 255);
410.     this->c = static_cast<unsigned char>(b * 255);
411. }
412.
413. void Pixel::RGB_to_YCbCr_601() {
414.     // R G B
415.     const double r = a / 255.0;
416.     const double g = b / 255.0;
417.     const double b = c / 255.0;
418.
```

```
419.     // Coefficients for R G B
420.     const double coef_r = 0.299;
421.     const double coef_g = 0.587;
422.     const double coef_b = 0.114;
423.
424.     // Y Cb Cr
425.     const double Y = coef_r * r + coef_g * g + coef_b * b;
426.     const double Cb = (b - Y) / (2.0 * (1.0 - coef_b));
427.     const double Cr = (r - Y) / (2.0 * (1.0 - coef_r));
428.
429.     // Output
430.     this->a = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(Y * 255)))));
431.     this->b = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round((Cb + 0.5) * 255)))));
432.     this->c = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round((Cr + 0.5) * 255)))));
433. }
434.
435. void Pixel::YCbCr_601_to_RGB() {
436.     // Y Cb Cr
437.     const double Y = a / 255.0;
438.     const double Cb = b / 255.0 - 0.5;
439.     const double Cr = c / 255.0 - 0.5;
440.
441.     // Coefficients for R G B
442.     const double coef_r = 0.299;
443.     const double coef_g = 0.587;
444.     const double coef_b = 0.114;
445.
446.     // R G B
447.     const double r = Y + (2.0 - 2.0 * coef_r) * Cr;
448.     const double g = Y - coef_b * (2.0 - 2.0 * coef_b) * Cb / coef_g - coef_r * (2 - 2.0 * coef
_r) * Cr / coef_g;
449.     const double b = Y + (2.0 - 2.0 * coef_b) * Cb;
450.
451.     // Output
452.     this->a = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(r * 255)))));
453.     this->b = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(g * 255)))));
454.     this->c = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(b * 255)))));
455. }
456.
457. void Pixel::RGB_to_YCbCr_709() {
458.     // R G B
459.     const double r = a / 255.0;
460.     const double g = b / 255.0;
461.     const double b = c / 255.0;
462.
463.     // Coefficients for R G B
464.     const double coef_r = 0.2126;
465.     const double coef_g = 0.7152;
466.     const double coef_b = 0.0722;
467.
468.     // Y Cb Cr
469.     const double Y = coef_r * r + coef_g * g + coef_b * b;
470.     const double Cb = (b - Y) / (2.0 * (1.0 - coef_b));
471.     const double Cr = (r - Y) / (2.0 * (1.0 - coef_r));
472.
473.     // Output
474.     this->a = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(Y * 255)))));
475.     this->b = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round((Cb + 0.5) * 255)))));
476.     this->c = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round((Cr + 0.5) * 255)))));
```

```
477. }
478.
479. void Pixel::YCbCr_709_to_RGB() {
480.     // Y Cb Cr
481.     const double Y = a / 255.0;
482.     const double Cb = b / 255.0 - 0.5;
483.     const double Cr = c / 255.0 - 0.5;
484.
485.     // Coefficients for R G B
486.     const double coef_r = 0.2126;
487.     const double coef_g = 0.7152;
488.     const double coef_b = 0.0722;
489.
490.     // R G B
491.     const double r = Y + (coef_r * -2.0 + 2.0) * Cr;
492.     const double g = Y - coef_b * (2.0 - 2.0 * coef_b) * Cb / coef_g - coef_r * (2.0 - 2.0 * co
ef_r) * Cr / coef_g;
493.     const double b = Y + (coef_b * -2.0 + 2.0) * Cb;
494.
495.     // Output
496.     this->a = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(r * 255)))));
497.     this->b = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(g * 255)))));
498.     this->c = static_cast<unsigned char>(std::min(255,
std::max(0, static_cast<int>(std::round(b * 255)))));
499. }
```