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 №M3234:**

***Ота Никита Терентьевич***

**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).

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:

С: 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:**

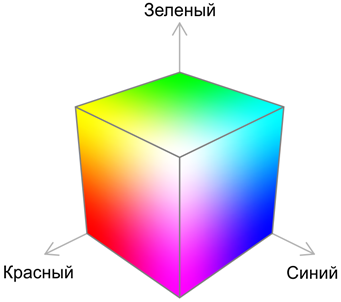
Цветовое пространство — модель представления цвета, основанная на использовании цветовых координат. Цветовое пространство строится таким образом, чтобы любой цвет был представлен точкой, имеющей определённые координаты. Чаще всего одному набору координат будет соответствовать один цвет, но для некоторых случаев это не так (например, для модели CMYK или, например, когда кодируется цветовой тон — данные по тону «закольцованы», и тона для максимального и минимального значений совпадают).

Наиболее популярными на сегодняшний день являются следующие цветовые модели: RGB (используется в основном в мониторах и камерах), CMY(K) (используется в полиграфии), HSI (широко используется в машинном зрении и дизайне). Существует множество других моделей. Например, CIE XYZ (стандартные модели), YCbCr и др.

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

**RGB**

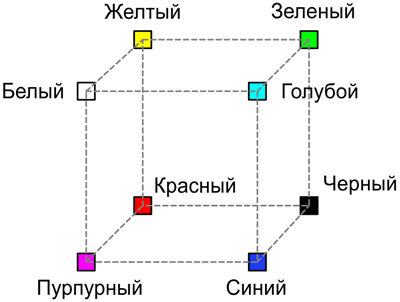
В модели RGB (от англ. red – красный, green – зелёный, blue – голубой) все цвета получаются путём смешения трёх базовых (красного, зелёного и синего) цветов в различных пропорциях. Доля каждого базового цвета в итоговом может восприниматься, как координата в соответствующем трёхмерном пространстве, поэтому данную модель часто называют цветовым кубом.



Модель цветового куба RGB

Чаще всего модель строится так, чтобы куб был единичным. Точки, соответствующие базовым цветам, расположены в вершинах куба, лежащих на осях: красный – (1;0;0), зелёный – (0;1;0), синий – (0;0;1). При этом вторичные цвета (полученные смешением двух базовых) расположены в других вершинах куба: голубой — (0;1;1), пурпурный — (1;0;1) и жёлтый – (1;1;0). Чёрный и белые цвета расположены в начале координат (0;0;0) и наиболее удалённой от начала координат точке (1;1;1).

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



Вершины куба

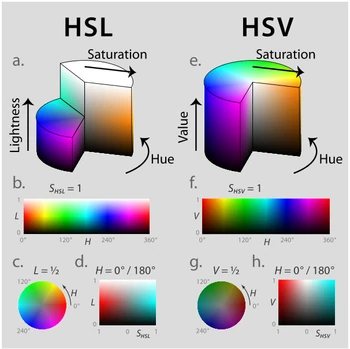
**HSL и HSV**

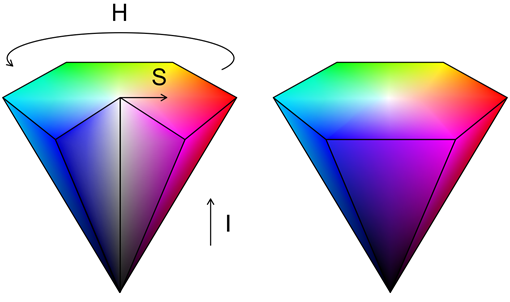
HSL, HLS или HSI (от англ. hue, saturation, lightness (intensity)) — цветовая модель, в которой цветовыми координатами являются тон, насыщенность и светлота.

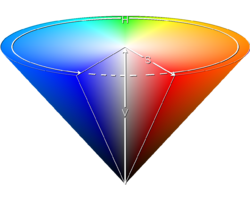
HSV (англ. Hue, Saturation, Value — тон, насыщенность, значение) или HSB (англ. Hue, Saturation, Brightness — тон, насыщенность, яркость) — цветовая модель, в которой координатами цвета являются тон, насыщенность и яркость.  
Следует отметить, что HSV и HSL — две разные цветовые модели (lightness — светлота, что отличается от яркости).

* Hue — цветовой тон, (например, красный, зелёный или сине-голубой). Варьируется в пределах 0—360°, однако иногда приводится к диапазону 0—100 или 0—1.
* Saturation — насыщенность (цвета). Варьируется в пределах 0—100 или 0—1. Чем больше этот параметр, тем «чище» цвет, поэтому этот параметр иногда называют чистотой цвета. А чем ближе этот параметр к нулю, тем ближе цвет к нейтральному серому.
* Lightness (светлота) или Intensity (интенсивность) – визуальная оценка яркости. Задаётся в пределах 0—100 или 0—1.
* Value (значение цвета) или Brightness — яркость. Также задаётся в пределах 0—100 или 0—1.

Простейший способ отобразить HSV в трёхмерное пространство — воспользоваться цилиндрической системой координат. Здесь координата H определяется полярным углом, S — радиус-вектором, а V — Z-координатой. То есть, оттенок изменяется при движении вдоль окружности цилиндра, насыщенность — вдоль радиуса, а яркость — вдоль высоты. Несмотря на «математическую» точность, у такой модели есть существенный недостаток: на практике количество различимых глазом уровней насыщенности и оттенков уменьшается при приближении яркости (V) к нулю (то есть, на оттенках, близких к чёрному).



Другой способ визуализации цветового пространства — конус. Как и в цилиндре, оттенок изменяется по окружности конуса. Насыщенность цвета возрастает с отдалением от оси конуса, а яркость — с приближением к его основанию. Иногда вместо конуса используется шестиугольная правильная пирамида.



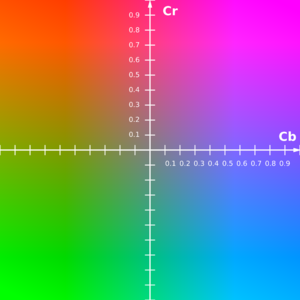
Предназначены для “интуитивно понятного” изменения таких характеристик цвета как: оттенок, насыщенность, яркость.

**YCbCr**

YCbCr, Y′CbCr, или Y Pb/Cb Pr/Cr, также пишется как Y’CBCR или YCBCR — семейство цветовых пространств, которые используются для передачи цветных изображений в компонентном видео и цифровой фотографии.

Y' — компонента яркости, CB и CR являются синей и красной цветоразностными компонентами.

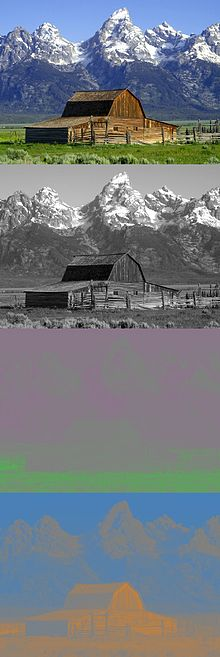
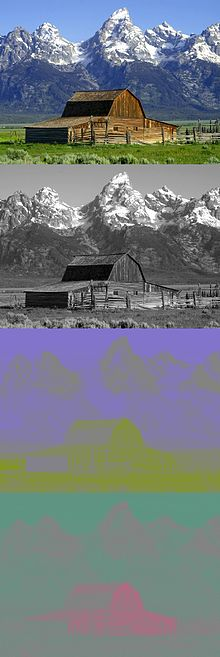
Y’CbCr не является абсолютным цветовым пространством, скорее, это способ кодирования информации сигналов RGB. Для систем отображения используются сигналы основных цветов RGB (красный, зелёный и синий). Эти сигналы не являются эффективными для хранения и передачи изображений, так как они имеют большую избыточность.



CbCr плоскость при постоянной яркости Y '= 0,5

**YCgCo**

YCgCo – цветовая модель, похожая на YСbCr, но с более простым преобразованием в или из RGB. Y – компонента яркости, Cg – chrominance green (цветоразность “хроматический зелёный”), Co – chrominance orange (цветоразность “хроматический оранжевый”). Также, как и YCbCr используется для обработки и хранения графической и видео информации.



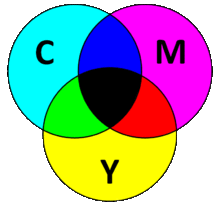
YCbCr

YCgCo

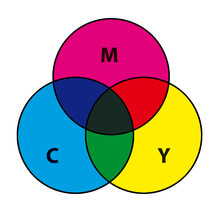
**CMY или CMYK**

CMY, CMYK – cубтрактивная модель (от англ. Cyan — голубой, Magenta — пурпурный, Yellow — жёлтый, blacK – чёрный). Используется для получения твёрдых копий (печати) изображений, и в некотором роде является антиподом цветового RGB-куба. Если в RGB модели базовые цвета – это цвета источников света, то модель CMY – это модель поглощения цветов.

Например, бумага, покрытая жёлтым красителем не отражает синий свет, т.е. можно сказать, что жёлтый краситель вычитает из отражённого белого света синий. Аналогично голубой краситель вычитает из отражённого света красный, а пурпурный краситель вычитает зелёный. Именно поэтому данную модель принято называть субтрактивной.

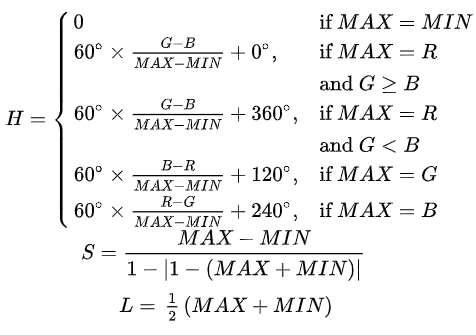


Легко заметить, что для получения чёрного цвета в модели CMY необходимо смешать голубой, пурпурный и жёлтый в равных пропорциях. Этот метод имеет два серьёзных недостатка: во-первых, полученный в результате смешения чёрный цвет будет выглядеть светлее «настоящего» чёрного, во-вторых, это приводит к существенным затратам красителя. Поэтому на практике модель СMY расширяют до модели CMYK, добавляя к трём цветам чёрный (англ. Black).

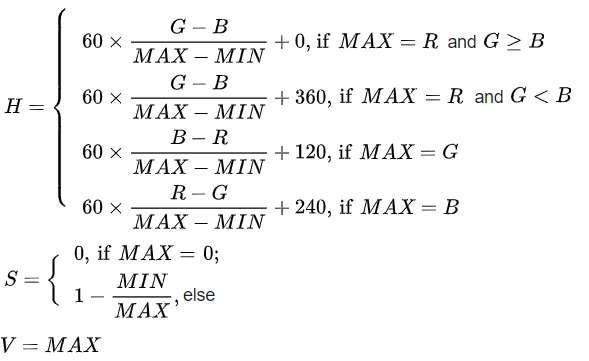


**Формулы перевода из одной модели в другую:**

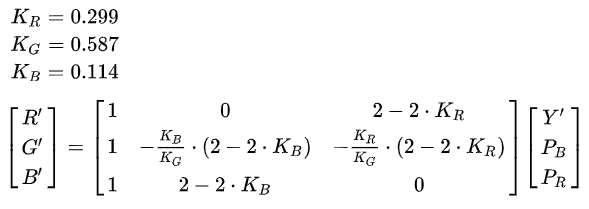
RGB → HSL



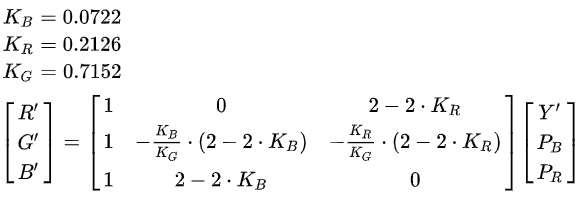
RGB → HSV



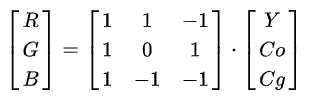
RGB → YCbCr.601



RGB → YCbCr.709

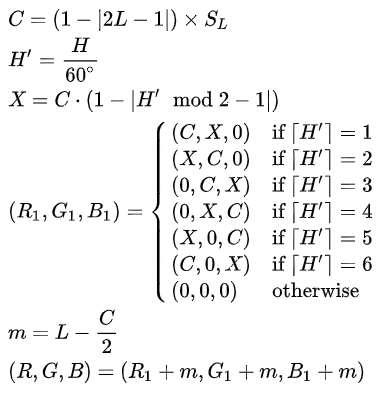


RGB → YCgCo

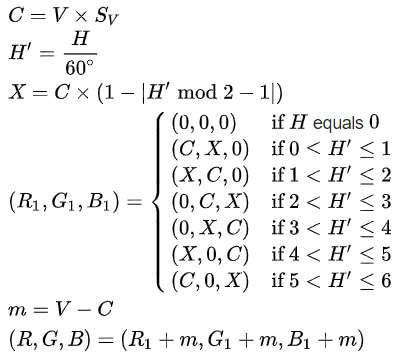


RGB → CMY

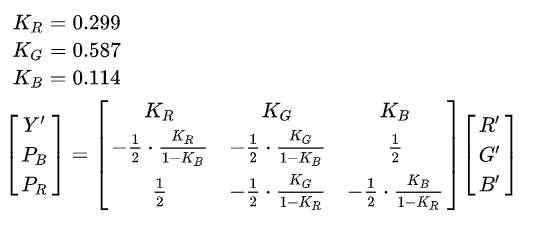
HSL → RGB



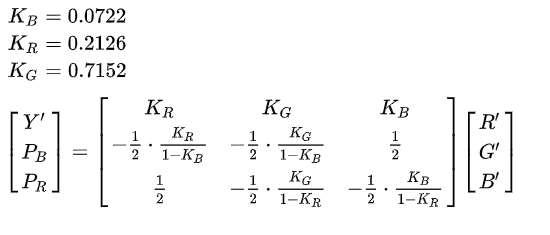
HSV → RGB

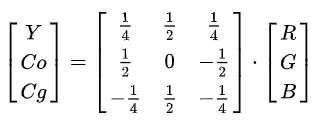


YCbCr.601 → RGB



YCbCr.709 → RGB





YCgCo → RGB CMY → RGB

**Практическая часть:**

Язык, на котором написана программа, решающая поставленную задачу: С++.

Ключи компиляции: -std=c++11 -fsanitize=address -fno-stack-limit.

Компилятор: g++.

**План работы программы:**

1. Чтение данных из командной строки, обработка неправильных введённых данных.
2. Чтение данных с введённого пользователем файла входа с обработкой ошибок.
3. Перевод данного цветового пространства в RGB.
4. Перевод из RGB в заданное цветовое пространство.
5. Вывод результата в файл с обработкой ошибок.

**Код программ:**

**main.cpp**

#include <iostream>

#include <string>

#include <cstring>

#include <fstream>

#include "colour\_space.h"

COLOUR\_SPACE check\_and\_get\_colour\_space(std::string const& s, std::map<std::string, COLOUR\_SPACE> &spaces) {

if (spaces[s] == ERROR) {

throw std::runtime\_error("Colour space " + s + " is not supported\n");

}

return spaces[s];

}

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

std::map<std::string, COLOUR\_SPACE> colour\_spaces = {

{"RGB", RGB},

{"HSL", HSL},

{"HSV", HSV},

{"YCbCr.601", YCbCr601},

{"YCbCr.709", YCbCr709},

{"YCoCg", YCoCg},

{"CMY", CMY}

};

if (argc != 11) {

std::cerr << "Got " << argc << " arguments\n";

std::cerr << "Usage: -f <from\_color\_space> -t <to\_color\_space> -i <count>"

" <input\_file\_name> -o <count> <output\_file\_name>";

return 1;

}

COLOUR\_SPACE colour\_space\_from = ERROR, colour\_space\_to = ERROR;

std::string input\_file\_name, output\_file\_name;

int inputs = 0, outputs = 0;

try {

for (int arg = 1; arg < argc; ++arg) {

if (argv[arg][0] == '-' && strlen(argv[arg]) == 2) {

switch (argv[arg][1]) {

case 'f':

colour\_space\_from = check\_and\_get\_colour\_space(argv[arg + 1], colour\_spaces);

break;

case 't':

colour\_space\_to = check\_and\_get\_colour\_space(argv[arg + 1], colour\_spaces);

break;

case 'i':

inputs = std::stoi(argv[arg + 1]);

input\_file\_name = std::string(argv[arg + 2]);

break;

case 'o':

outputs = std::stoi(argv[arg + 1]);

output\_file\_name = std::string(argv[arg + 2]);

break;

default:

std::cerr << argv[arg] << " is not supported\n";

return 1;

}

}

}

} catch (std::runtime\_error const& e) {

std::cerr << e.what();

return 1;

}

if (colour\_space\_from == ERROR) {

std::cerr << "Input file's colour space info not found\n";

return 1;

}

if (colour\_space\_to == ERROR) {

std::cerr << "Output file's colour space info not found\n";

return 1;

}

if (inputs == 0 || input\_file\_name.empty()) {

std::cerr << "Input file or files not found\n";

return 1;

}

if (outputs == 0 || output\_file\_name.empty()) {

std::cerr << "Output file or files not found\n";

}

try {

picture pic(inputs, input\_file\_name, colour\_space\_from);

pic.to\_RGB();

pic.RGB\_to(colour\_space\_to);

pic.write(outputs, output\_file\_name);

} catch (std::runtime\_error const& e) {

std::cerr << e.what();

return 1;

}

// all arguments are correct according to guarantee

return 0;

}

**colour\_space.h**

#include <map>

#include <string>

int const MAX\_BRIGHTNESS = 255;

int const TWO\_PIES = 360;

int const ONE\_THIRD\_PI = 60;

double const EPS = 1e-5;

double const ONE\_SECOND = 0.5;

double const K\_R\_601 = 0.299;

double const K\_G\_601 = 0.587;

double const K\_B\_601 = 0.114;

double const K\_R\_709 = 0.2126;

double const K\_G\_709 = 0.0722;

double const K\_B\_709 = 0.7152;

enum COLOUR\_SPACE {

ERROR,

RGB,

HSL,

HSV,

YCbCr601,

YCbCr709,

YCoCg,

CMY

};

struct picture {

picture() = default;

picture(int cnt, std::string const& name, COLOUR\_SPACE space);

~picture();

int get\_width() const;

int get\_height() const;

int get\_max\_brightness() const;

int\*\*\* get\_data();

void to\_RGB();

void RGB\_to(COLOUR\_SPACE);

void write(int cnt, std::string const& name);

private:

int width = 0;

int height = 0;

int max\_brightness = 255;

int \*\*\*data = nullptr;

COLOUR\_SPACE space = ERROR;

void update\_int(std::string const& name, int& x, int curr\_value);

void read\_meta\_data(std::ifstream &in, std::string const& name, int encoding);

void read\_data\_P5(std::string const& file);

void read\_data\_P6(std::string const& file);

void force\_close(std::string const& message);

void clear\_array();

void write\_to\_one\_P6(std::string const& file);

void write\_to\_three\_P5(std::string const& file);

void write\_meta\_data(std::ofstream &out, std::string const& name, int encoding);

};

**colour\_space.cpp**

#include <iostream>

#include <fstream>

#include <stdexcept>

#include <math.h>

#include "colour\_space.h"

// picture initialization

picture::picture(int cnt, std::string const& name, COLOUR\_SPACE space) : space(space) {

switch (cnt) {

case 1:

if (name.substr(name.length() - 4, 4) != ".ppm") {

force\_close("Expected ppm file\n");

}

read\_data\_P6(name);

break;

case 3:

if (name.substr(name.length() - 4, 4) != ".pgm") {

force\_close("Expected ppm file\n");

}

read\_data\_P5(name);

break;

}

}

void picture::clear\_array() {

if (data != nullptr) {

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

delete[] data[h][w];

}

delete[] data[h];

}

delete[] data;

data = nullptr;

}

}

picture::~picture() {

clear\_array();

}

int picture::get\_width() const {

return width;

}

int picture::get\_height() const {

return height;

}

int picture::get\_max\_brightness() const {

return max\_brightness;

}

int\*\*\* picture::get\_data() {

return data;

}

void picture::read\_data\_P5(std::string const& file) {

int dot\_pos = file.find\_last\_of('.');

std::string name = file.substr(0, dot\_pos);

std::string extension = file.substr(dot\_pos);

for (int num = 0; num < 3; ++num) {

std::string file\_name = name + "\_" + std::to\_string(num + 1) + extension;

std::ifstream in(file\_name);

read\_meta\_data(in, file\_name, 5);

// File contains width \* height bytes of data

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

data[h][w][num] = in.get();

}

}

}

}

void picture::read\_data\_P6(std::string const& name) {

std::ifstream in(name);

read\_meta\_data(in, name, 6);

// File contains width \* height \* 3 bytes of data

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

for (int color = 0; color < 3; ++color) {

data[h][w][color] = in.get();

}

}

}

}

void picture::update\_int(std::string const& name, int &x, int curr\_value) {

if (x == 0) {

x = curr\_value;

} else if (x != curr\_value) {

force\_close(name + " in 2 files are different\n");

}

}

void picture::read\_meta\_data(std::ifstream &in, std::string const& name, int encoding) {

if (!in.is\_open()) {

force\_close("Couldn't find file" + name + '\n');

}

char p, line\_separator;

int type;

in >> p >> type;

line\_separator = in.get();

if (p != 'P' || type != encoding || line\_separator != '\n') {

force\_close("Expected P" + std::to\_string(encoding) + " encoding in " + name);

}

int this\_height, this\_width;

in >> this\_width >> this\_height;

line\_separator = in.get();

// guaranteed that width > 0 && height > 0 && they are ints

if (line\_separator != '\n') {

force\_close("Couldn't read width and height\n");

}

update\_int("Heights", height, this\_height);

update\_int("Width", width, this\_width);

in >> max\_brightness;

line\_separator = in.get();

if (max\_brightness != 255 || line\_separator != '\n') {

force\_close("Couldn't read max brightness or max brightness is not 255\n");

}

if (data == nullptr) {

data = new int\*\*[height];

for (int h = 0; h < height; ++h) {

data[h] = new int\*[width];

for (int w = 0; w < width; ++w) {

data[h][w] = new int[3];

}

}

}

}

void picture::force\_close(std::string const& message) {

clear\_array();

throw std::runtime\_error(message);

}

// end of picture initialization

// from ? to RGB

// a < b

inline bool less(double a, double b) {

return (b - a) > ((fabs(a) < fabs(b) ? fabs(b) : fabs(a)) \* EPS);

}

inline int correction(double val) {

return static\_cast<int>(floor(MAX\_BRIGHTNESS \* fmax(0, fmin(1, val))));

}

inline void update(int \*pixel, double x, double y, double z) {

pixel[0] = correction(x);

pixel[1] = correction(y);

pixel[2] = correction(z);

}

void HSL\_to\_RGB(int \*pixel) {

double H = (pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS)) \* TWO\_PIES;

double S = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double L = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double H\_ = H / ONE\_THIRD\_PI;

double C = (1 - fabs(2 \* L - 1)) \* S;

double X = C \* (1 - fabs(fmod(H\_, 2) - 1));

double m = L - C / 2;

double R = 0, G = 0, B = 0;

switch (static\_cast<int>(floor(H\_))) {

case 0:

R = C;

G = X;

B = 0;

break;

case 1:

R = X;

G = C;

B = 0;

break;

case 2:

R = 0;

G = C;

B = X;

break;

case 3:

R = 0;

G = X;

B = C;

break;

case 4:

R = X;

G = 0;

B = C;

break;

case 5:

R = C;

G = 0;

B = X;

break;

}

update(pixel, R + m, G + m, B + m);

}

void HSV\_to\_RGB(int \*pixel) {

double H = (pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS)) \* TWO\_PIES;

double S = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double V = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double C = V \* S;

double H\_ = H / ONE\_THIRD\_PI;

double X = C \* (1 - fabs(fmod(H\_, 2) - 1));

double m = V - C;

double R = 0, G = 0, B = 0;

// doesn't work if H\_ > 0 :(

if (H\_ >= 0 && H\_ <= 1) {

R = C;

G = X;

B = 0;

} else if (H\_ > 1 && H\_ <= 2) {

R = X;

G = C;

B = 0;

} else if (H\_ > 2 && H\_ <= 3) {

R = 0;

G = C;

B = X;

} else if (H\_ > 3 && H\_ <= 4) {

R = 0;

G = X;

B = C;

} else if (H\_ > 4 && H\_ <= 5) {

R = X;

G = 0;

B = C;

} else if (H\_ > 5 && H\_ <= 6) {

R = C;

G = 0;

B = X;

}

update(pixel, R + m, G + m, B + m);

}

void YCbCr\_to\_RGB(int \*pixel, double K\_R, double K\_G, double K\_B) {

double Y = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double Cb = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS) - ONE\_SECOND;

double Cr = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS) - ONE\_SECOND;

double R = Y + Cr \* (2 - 2 \* K\_R);

double G = Y - (K\_B / K\_G) \* (2 - 2 \* K\_B) \* Cb - (K\_R / K\_G) \* (2 - 2 \* K\_R) \* Cr;

double B = Y + (2 - 2 \* K\_B) \* Cb;

update(pixel, R, G, B);

}

void YCoCg\_to\_RGB(int \*pixel) {

double Y = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double Co = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS) - ONE\_SECOND;

double Cg = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS) - ONE\_SECOND;

double R = Y + Co - Cg;

double G = Y + Cg;

double B = Y - Co - Cg;

update(pixel, R, G, B);

}

void picture::to\_RGB() {

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

switch (space) {

case RGB:

return;

case HSL:

HSL\_to\_RGB(data[h][w]);

break;

case HSV:

HSV\_to\_RGB(data[h][w]);

break;

case YCbCr601:

YCbCr\_to\_RGB(data[h][w], K\_R\_601, K\_G\_601, K\_B\_601);

break;

case YCbCr709:

YCbCr\_to\_RGB(data[h][w], K\_R\_709, K\_G\_709, K\_B\_709);

break;

case YCoCg:

YCoCg\_to\_RGB(data[h][w]);

break;

case CMY:

data[h][w][0] = MAX\_BRIGHTNESS - data[h][w][0];

data[h][w][1] = MAX\_BRIGHTNESS - data[h][w][1];

data[h][w][2] = MAX\_BRIGHTNESS - data[h][w][2];

break;

}

}

}

}

// end of ? to RGB

// RGB to ?

void RGB\_to\_HSL(int \*pixel) {

int max\_i = std::max(pixel[0], std::max(pixel[1], pixel[2]));

int min\_i = std::min(pixel[0], std::min(pixel[1], pixel[2]));

double R = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double G = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double B = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double max\_d = fmax(R, fmax(G, B));

double min\_d = fmin(R, fmin(G, B));

double V = max\_d;

double C = max\_d - min\_d;

double L = V - C / 2;

double H = 0;

if (max\_i != min\_i) {

if (max\_d == R) {

H = ONE\_THIRD\_PI \* ((G - B) / C);

} else if (max\_d == G) {

H = ONE\_THIRD\_PI \* (2 + (B - R) / C);

} else if (max\_d == B) {

H = ONE\_THIRD\_PI \* (4 + (R - G) / C);

}

}

if (H < 0) {

H += TWO\_PIES;

}

double S = 0;

// V != 0 && V != 1

if (max\_i != 0 && max\_i != 2 \* MAX\_BRIGHTNESS - min\_i) {

S = (V - L) / std::min(L, 1 - L);

}

update(pixel, H / TWO\_PIES, S, L);

}

void RGB\_to\_HSV(int \*pixel) {

double R = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double G = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double B = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double max\_d = fmax(R, fmax(G, B));

double min\_d = fmin(R, fmin(G, B));

double C = max\_d - min\_d;

double V = max\_d;

double H = 0;

if (C != 0) {

if (max\_d == R) {

H = ONE\_THIRD\_PI \* (G - B) / C;

} else if (max\_d == G) {

H = ONE\_THIRD\_PI \* (2 + (B - R) / C);

} else if (max\_d == B) {

H = ONE\_THIRD\_PI \* (4 + (R - G) / C);

}

}

if (H < 0) {

H += TWO\_PIES;

}

double S = 0;

if (V != 0) {

S = C / V;

}

update(pixel, H / TWO\_PIES, S, V);

}

void RGB\_to\_YCbCr(int \*pixel, double K\_R, double K\_G, double K\_B) {

double R = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double G = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double B = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double Y = K\_R \* R + K\_G \* G + K\_B \* B;

double Cb = ONE\_SECOND \* (B - Y) / (1 - K\_B);

double Cr = ONE\_SECOND \* (R - Y) / (1 - K\_R);

pixel[0] = correction(Y);

pixel[1] = correction(Cb + ONE\_SECOND);

pixel[2] = correction(Cr + ONE\_SECOND);

}

void RGB\_to\_YCoCg(int \*pixel) {

double R = pixel[0] / static\_cast<double>(MAX\_BRIGHTNESS);

double G = pixel[1] / static\_cast<double>(MAX\_BRIGHTNESS);

double B = pixel[2] / static\_cast<double>(MAX\_BRIGHTNESS);

double Y = R / 4 + G / 2 + B / 4;

double Co = R / 2 - B / 2;

double Cg = -R / 4 + G / 2 - B / 4;

pixel[0] = correction(Y);

pixel[1] = correction(Co + ONE\_SECOND);

pixel[2] = correction(Cg + ONE\_SECOND);

}

void picture::RGB\_to(COLOUR\_SPACE space) {

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

switch (space) {

case RGB:

return;

case HSL:

RGB\_to\_HSL(data[h][w]);

break;

case HSV:

RGB\_to\_HSV(data[h][w]);

break;

case YCbCr601:

RGB\_to\_YCbCr(data[h][w], K\_R\_601, K\_G\_601, K\_B\_601);

break;

case YCbCr709:

RGB\_to\_YCbCr(data[h][w], K\_R\_709, K\_G\_709, K\_B\_709);

break;

case YCoCg:

RGB\_to\_YCoCg(data[h][w]);

break;

case CMY:

data[h][w][0] = MAX\_BRIGHTNESS - data[h][w][0];

data[h][w][1] = MAX\_BRIGHTNESS - data[h][w][1];

data[h][w][2] = MAX\_BRIGHTNESS - data[h][w][2];

break;

}

}

}

}

// end of RGB to ?

// write info to file

void picture::write(int cnt, std::string const& name) {

switch (cnt) {

case 1:

if (name.substr(name.length() - 4, 4) != ".ppm") {

force\_close("Output file must be ppm\n");

}

write\_to\_one\_P6(name);

break;

case 3:

if (name.substr(name.length() - 4, 4) != ".pgm") {

force\_close("Output file must be pgm\n");

}

write\_to\_three\_P5(name);

break;

}

}

void picture::write\_to\_one\_P6(std::string const& file) {

std::ofstream out(file);

if (!out.is\_open()) {

force\_close("Couldn't open file " + file + " to write\n");

}

write\_meta\_data(out, file, 6);

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

for (int color = 0; color < 3; ++color) {

if (!out || out.bad()) {

force\_close("Output stream was damaged\n");

}

out << static\_cast<unsigned char>(data[h][w][color]);

}

}

}

out.close();

}

void picture::write\_to\_three\_P5(std::string const& file) {

int dot\_pos = file.find\_last\_of('.');

std::string name = file.substr(0, dot\_pos);

std::string extension = file.substr(dot\_pos);

for (int num = 0; num < 3; ++num) {

std::string file\_name = name + "\_" + std::to\_string(num + 1) + extension;

// std::cout << "Writing info into " << file\_name << '\n';

std::ofstream out(file\_name);

if (!out.is\_open()) {

force\_close("Could't open file " + file\_name + " to write\n");

}

write\_meta\_data(out, file\_name, 5);

for (int h = 0; h < height; ++h) {

for (int w = 0; w < width; ++w) {

if (!out || out.bad()) {

force\_close("Output stream was damaged\n");

}

out << static\_cast<unsigned char>(data[h][w][num]);

}

}

out.close();

}

}

void picture::write\_meta\_data(std::ofstream &out, std::string const& name, int encoding) {

out << 'P' << encoding << '\n';

out << width << ' ' << height << '\n';

out << max\_brightness << '\n';

}