Работа с изображениями в языке C++

Бирюков В. А.

October 22, 2022

Шестнадцатеричная система

Система счисления по целочисленному основанию 16.

В качестве цифр этой системы обычно используются цифры от 0 до 9 и латинские буквы от A до F.

Примеры:

```
6 = 0x6
12 = 0xc
20 = 0x14
200 = 0xc8
```

```
255 = 0xff
256 = 0x100
1000 = 0x3E8
1024 = 0x400
```

Коды ASCII в шестнадцатеричной системе

Коды ASCII в шестнадцатеричной системе:

'\0' = 0x00	'0' = 0x30	'A' = 0x41	'a' = 0x61
'\n' = 0x0a	'1' = 0x31	'B' = 0x42	'b' = 0x62
'\r' = 0x0d	'2' = 0x32	'P' = 0x50	'p' = 0x70
' ' = 0x20	'3' = 0x33	'Z' = 0x5A	'z' = 0x7a

Сохраняем строку в файл

Текстовый режим открытия файла:

```
std::ofstream out{"my_file.txt"};
out << "Cat\nDog";
```

То в файл запишется (Linux):

```
43 61 74 0a 44 6f 67
```

Или в файл запишется (Windows):

```
43 61 74 0d 0a 44 6f 67
```

Сохраняем строку в файл

Бинарный режим открытия файла:

```
std::ofstream out{"my_file.txt", std::ios::binary};
out << "Cat\nDog";
```

То в файл запишется (Linux и Windows):

```
43 61 74 0a 44 6f 67
```

Сохраняем число в файл

```
int a = 12345678;
std::ofstream out{"my_file.txt"};
out << a;
```

То в файл запишется строка, представляющая число 12345678:

```
31 32 33 34 35 36 37 38
```

Сохраняем число в файл

```
int a = 12345678; // 12345678 == 0xBC614E
std::ofstream out{"my_file.txt", std::ios::binary};
out.write(reinterpret_cast<const char*>(&a), 4);
```

То в файл запишется байтовое представления числа в памяти:

```
4E 61 BC 00
```

Сохраняем `unsigned char` в файл

Числа типа `unsigned char` и `char` воспринимаются оператором `<<` как символы.

```
unsigned char a = 75; // 75 = 0x4b
std::ofstream out{"my_file.txt"};
out << a;
```

То в файл запишется символ `K`:

```
4B
```

Если вы откроете этот файл в текстовом редакторе, то увидите:

```
K
```

Сохраняем `unsigned char` в файл

Чтобы число типа `unsigned char` воспринимается оператором << как число, нужно привести его к другому целочисленному типу.

```
unsigned char a = 75; // 75 = 0x4b
std::ofstream out{"my_file.txt"};
out << static_cast<int>(a);
```

То в файл запишется символ `K`:

```
37 35
```

Если вы откроете этот файл в текстовом редакторе, то увидите:

```
75
```

Сохраняем `unsigned char` в файл

Числа типа `unsigned char` и `char` воспринимаются оператором `<<` как символы.

```
unsigned char a = 75; // 75 = 0x4b
std::ofstream out{"my_file.txt", std::ios::binary};
out.write(reinterpret_cast<const char*>(&a), 1);
```

То в файл запишется байтовое представление числа:

```
4B
```

Если вы откроете этот файл в текстовом редакторе, то увидите:

```
K
```

Хранение информации о цвете в памяти

Цветовая модель RGB

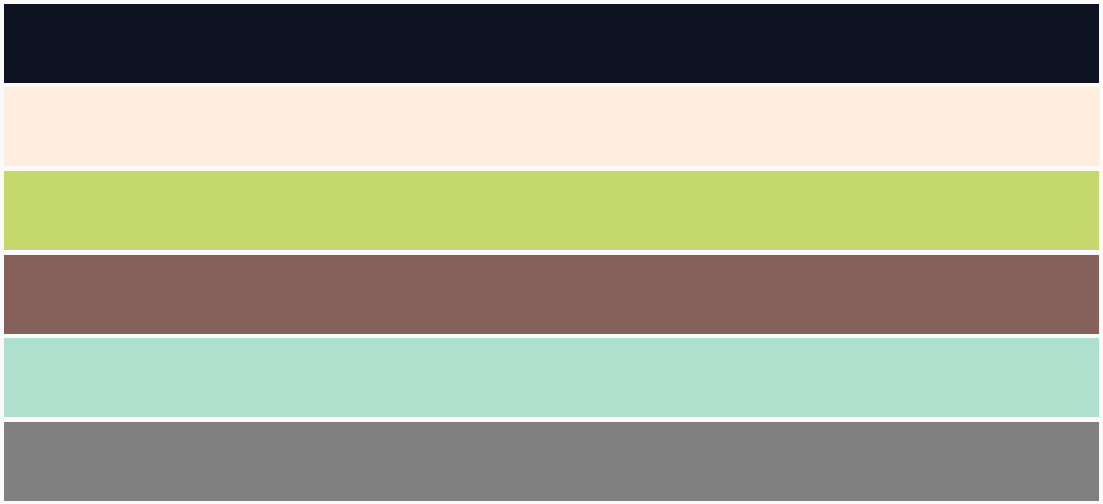
- RGB – цветовая модель, описывающая способ кодирования цвета с помощью трёх цветов: красного(R), зелёного(G) и синего(B).
- Чаще всего, в современных компьтерах, каждая компонента цвета кодируется одним байтом.
- Соответственно, значение каждой компоненты цвета кодируется числои из отрезка [0, 255].

Хранение информации о цвете в памяти

(255, 0, 0)
(0, 255, 0)
(0, 0, 255)
(0, 255, 255)
(255, 0, 255)
(255, 255, 0)
(0, 0, 0)
(255, 255, 255)

Хранение информации о цвете в памяти

```
   (13, 19, 33)
(255, 237, 223)
(197, 216, 109)
  (134, 97, 92)
(175, 224, 206)
(128, 128, 128)
```



Хранение информации о цвете в памяти

0D1321

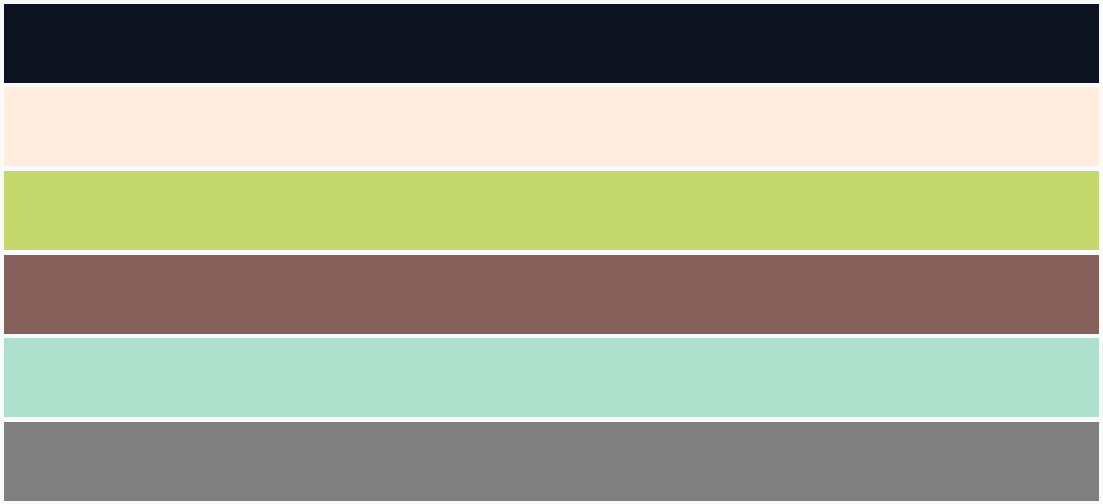
FFEDDF

C5D86D

86615C

AFE0CE

808080



Хранение информации о цвете в памяти

Предположим, что мы хотим хранить в памяти следующий цвет:

`rgb(40, 80, 120) = #285078`

Для этого можно создать массив из трёх элементов:

```
unsigned char a[3] = {40, 80, 120};
unsigned char b[3] = {0x28, 0x50, 0x78};
```

Хранение информации о цвете в памяти

Предположим, что мы хотим хранить в памяти следующий цвет:

`rgb(40, 80, 120) = #285078`

В языка C++ лучше воспользоваться контейнером `std::array`:

```
std::array<unsigned char, 3> a = {40, 80, 120};
std::array<unsigned char, 3> b = {0x28, 0x50, 0x78};
```

Хранение информации о цвете в памяти

Предположим, что мы хотим хранить в памяти следующий цвет:

`rgb(40, 80, 120) = #285078`

Можно создать структруру, которая будет хранить компоненты цвета:

```
struct Color
{
    unsigned char r, g, b;
};
//...
Color a = {40, 80, 120};
```

Печать компонент цвета

```
#include <iostream>

struct Color
{
    unsigned char r, g, b;
};

int main()
{
    Color a = {40, 80, 120};
    std::cout << a.r << " " << a.g << " "
              << a.b << std::endl;
}
```

На экран напечатается: `( P x`

Печать компонент цвета

```
#include <iostream>

struct Color
{
    std::uint8_t r, g, b;
};

int main()
{
    Color a = {40, 80, 120};
    std::cout << a.r << " " << a.g << " "
              << a.b << std::endl;
}
```

Всё равно напечатается: (P x

Печать компонент цвета

```
#include <iostream>

struct Color
{
    unsigned char r, g, b;
};

int main()
{
    Color a = {40, 80, 120};
    std::cout << (int)a.r << " " <<
                 (int)a.g << " " <<
                 (int)a.b << std::endl;
}
```

На экран напечатается: 40 80 120

Печать компонент цвета

```
#include <iostream>

struct Color
{
    unsigned char r, g, b;
};

int main()
{
    Color a = {40, 80, 120};
    std::cout << static_cast<int>(a.r) << " " <<
                 static_cast<int>(a.g) << " " <<
                 static_cast<int>(a.b) << std::endl;
}
```

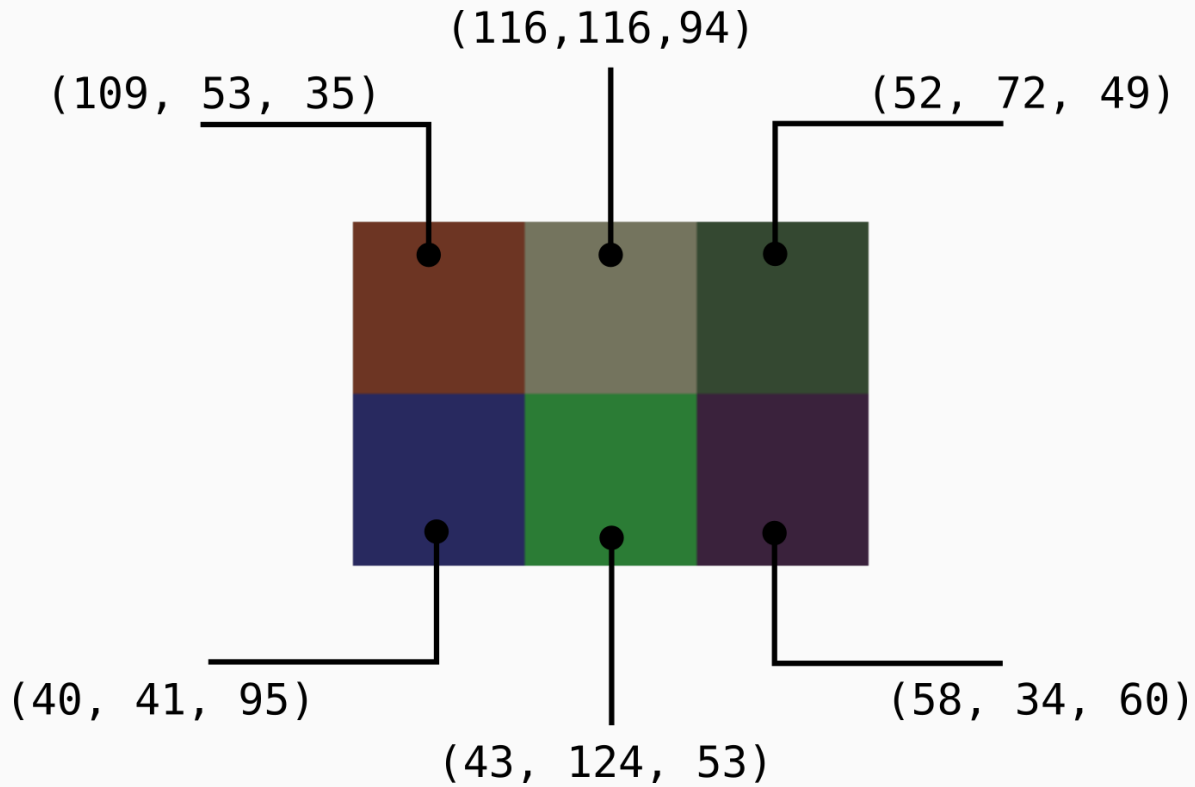
На экран напечатается: 40 80 120

Хранение изображения в памяти

Пример изображения

Ширина = 3 пикселя, высота = 2 пикселя

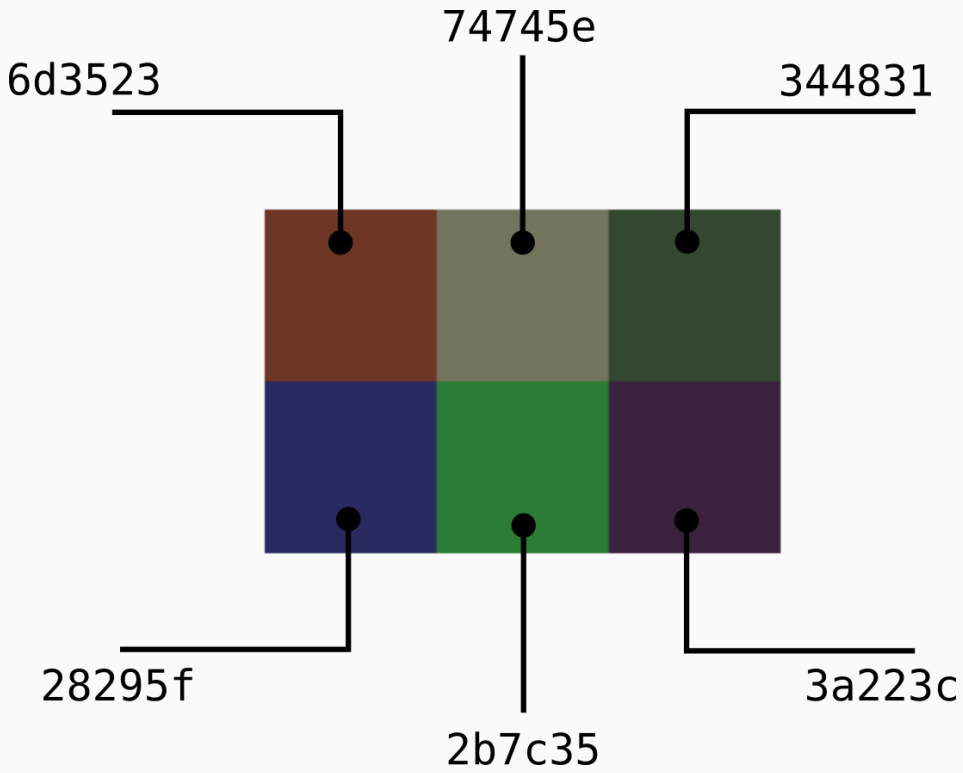
Цвет каждого пикселя кодируется 3-мя байтами



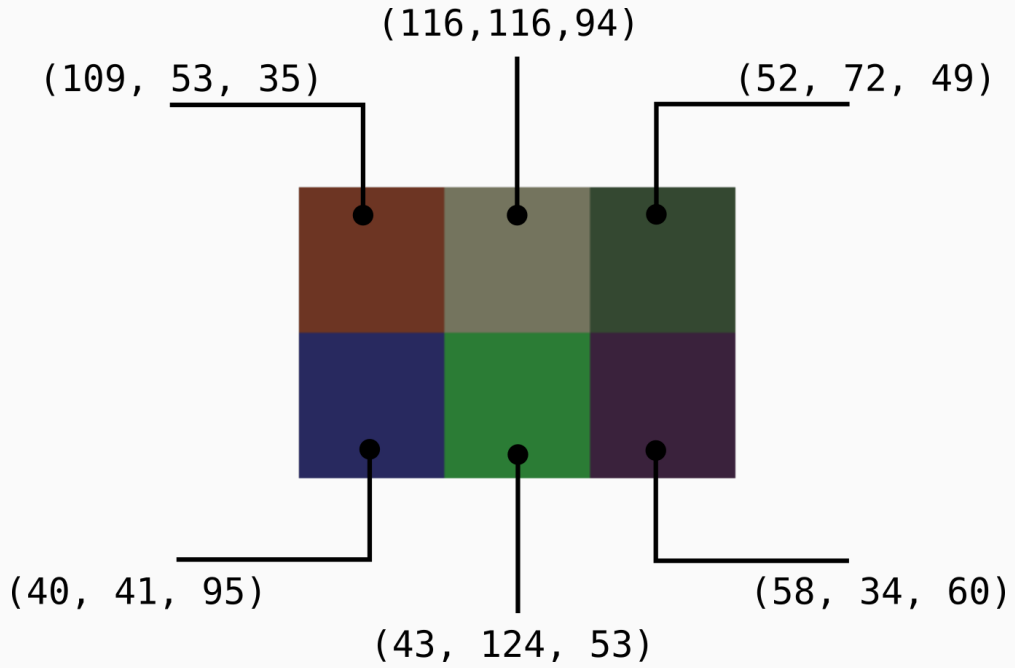
Пример изображения

Ширина = 3 пикселя, высота = 2 пикселя

Цвет каждого пикселя кодируется 3-мя байтами

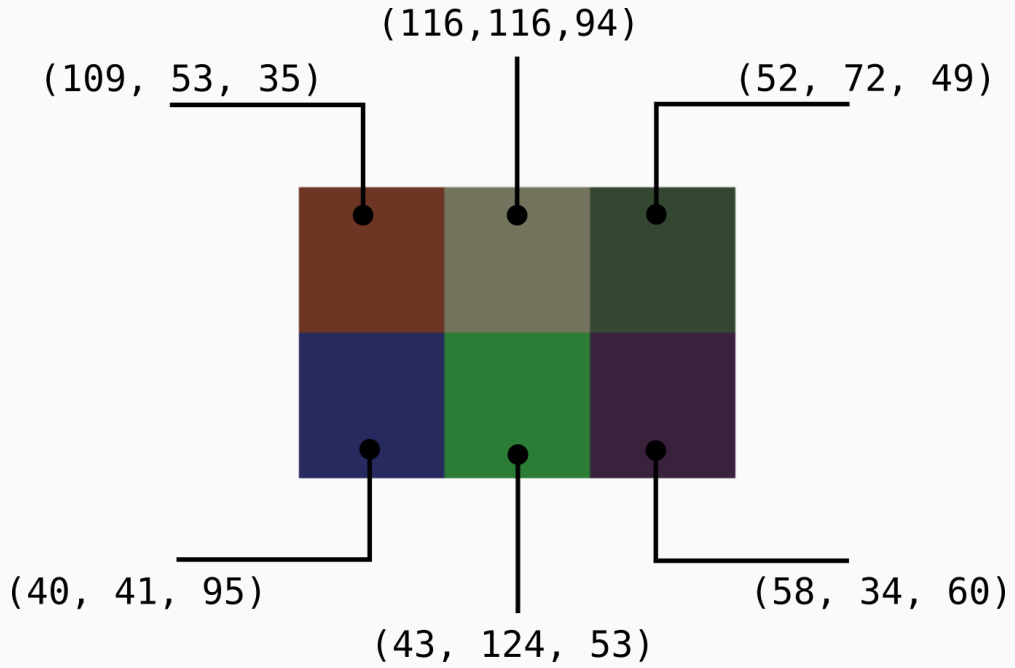


Хранение изображения в памяти(стиль C)



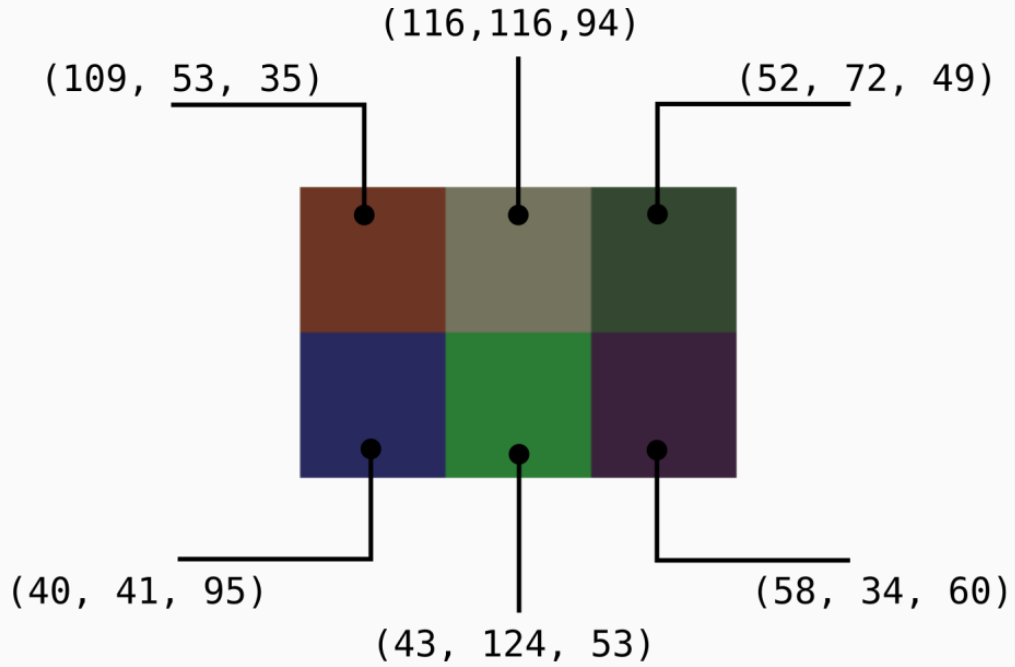

```
int width = 3, height = 2;
unsigned char data[18] = {109, 53, 35, 116, 116, 94,
                          52, 72, 49, 40, 41, 95,
                          43, 124, 53, 58, 34, 60};
```

Хранение изображения в памяти(стиль C++)




```
int width = 3, height = 2;
std::vector<unsigned char> v {109, 53, 35, 116, 116,94,
                              52, 72, 49, 40, 41, 95,
                              43, 124, 53, 58, 34, 60};
```

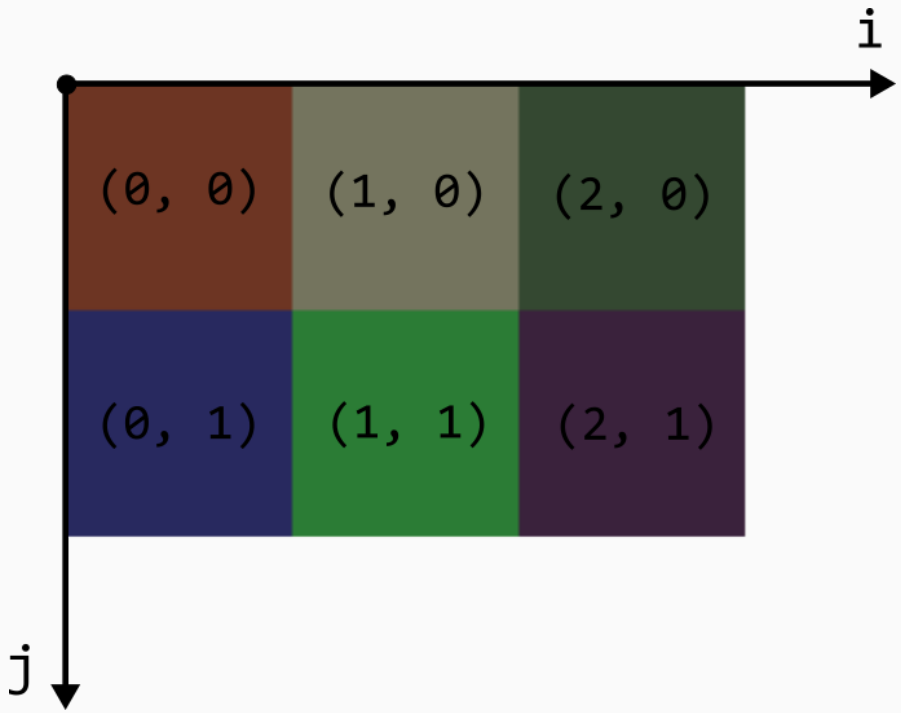
Хранение изображения в памяти



data

109	53	35	116	116	94	52	72	49	40	41	95	43	124	53	58	34	60

Координаты пикселей



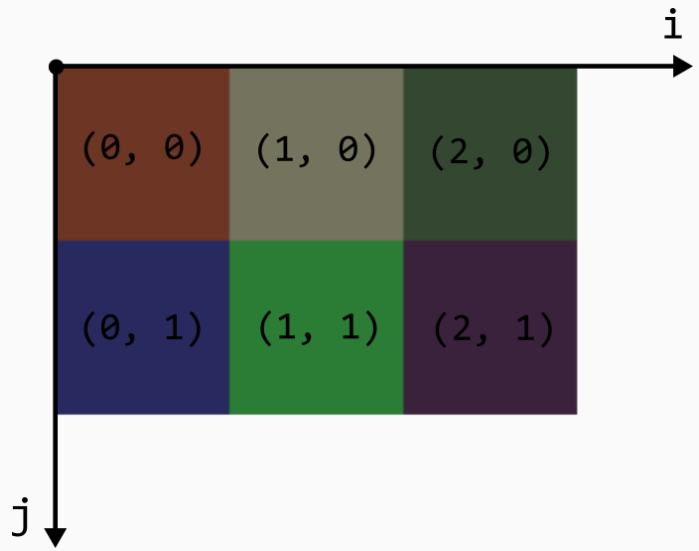
Индекс компоненты цвета пикселя в массиве

```
int width = getImageWidth();
int height = getImageHeight();
std::vector<unsigned char> data = getImageData();
```

Изменим k-ю компоненту пикселя с координатами (i, j):

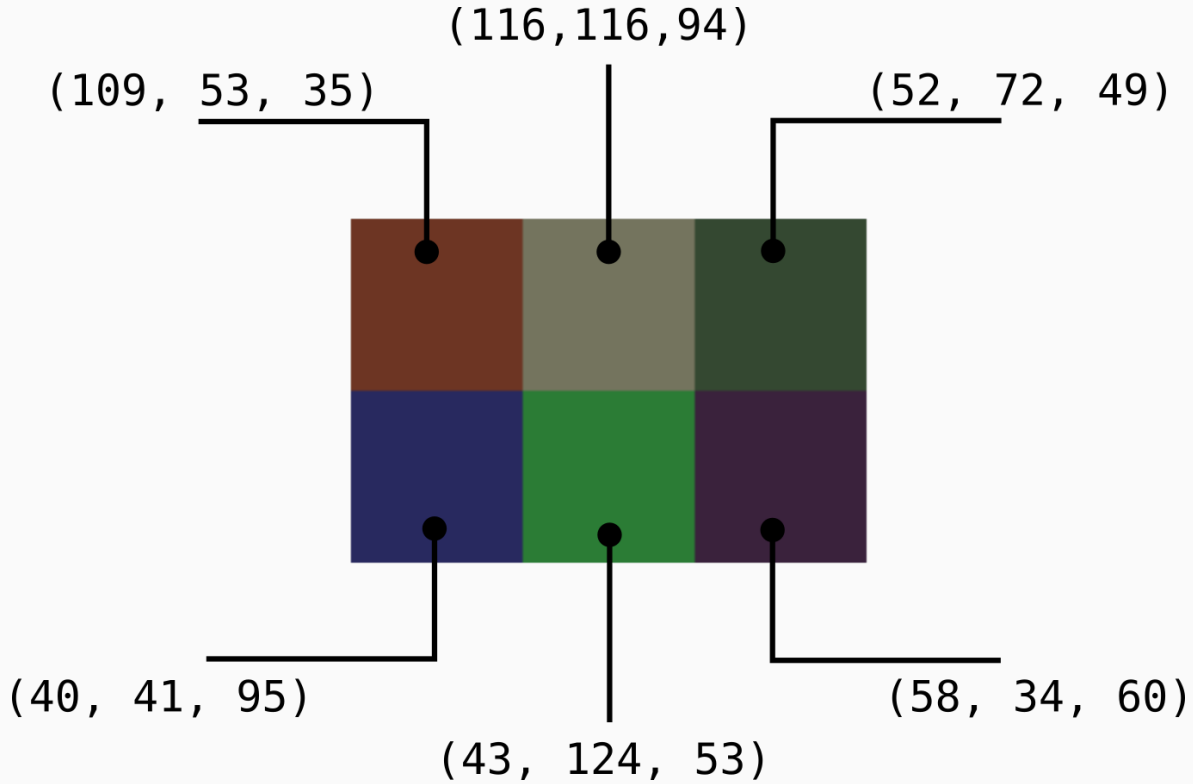
```
data[3 * (i + j * width) + k] = 100;
```

$i \in [0, width - 1], j \in [0, height - 1], k \in [0, 2],$

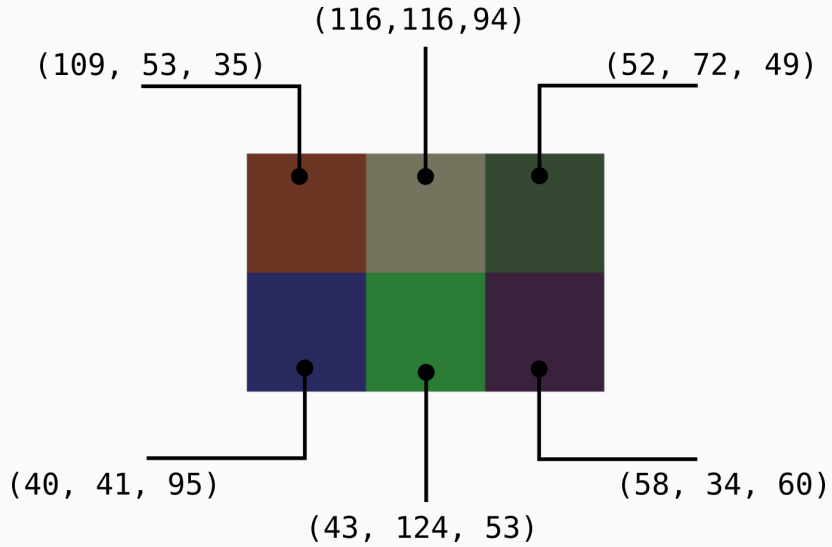


Формат `.ppm`

Пример изображения



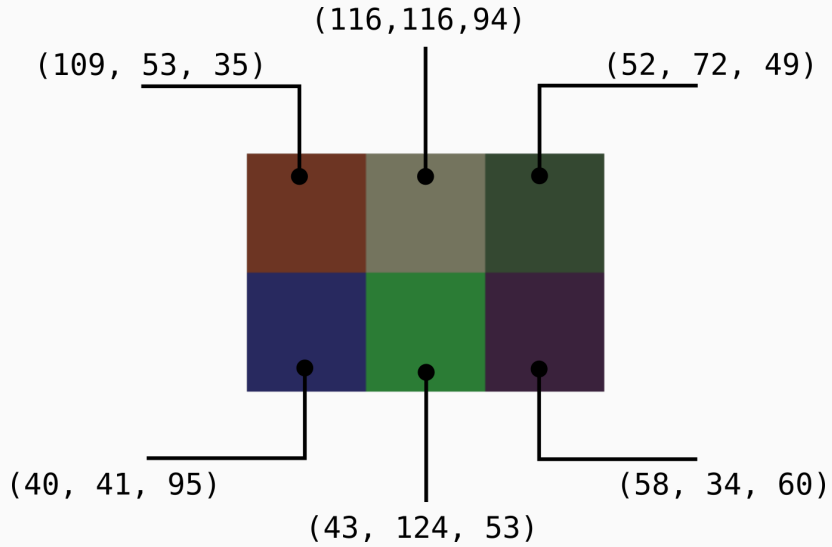
Текстовый формат .ppm изображения



Файл image.ppm

```
P3
3 2
255
109 53 35 116 116 94
52 72 49 40 41 95
43 124 53 58 34 60
```

Текстовый формат .ppm изображения. Побайтово.




```
50 33 0a 33 20 32 0a 32
35 35 0a 31 30 39 20 35
33 20 33 35 20 0a 31 31
36 20 31 31 36 20 39 34
20 0a 35 32 20 37 32 20
34 39 0a 34 30 20 34 31
20 39 35 20 0a 34 33 20
31 32 34 20 35 33 20 0a
35 38 20 33 34 20 36 30
```

Запись изображения формата ppm P3

```
const int width = 200;
const int height = 100;

std::ofstream out {"my_image.ppm"};
out << "P3\n" << width << " " << height << "\n255\n";

for (int j = 0; j < height; ++j)
{
    for (int i = 0; i < width; ++i)
        out << 40 << " " << 80 << " " << 120 << "\n";
}
```

Чтение изображения формата ppm P3

```
std::ifstream in {"my_file.ppm"};

std::string type;
in >> type;
if (type != "P3")
{
    std::cout << "Error. Format should be P3\n";
    std::exit(1);
}

int width, height, maxValue;
in >> width >> height >> maxValue;
...
```

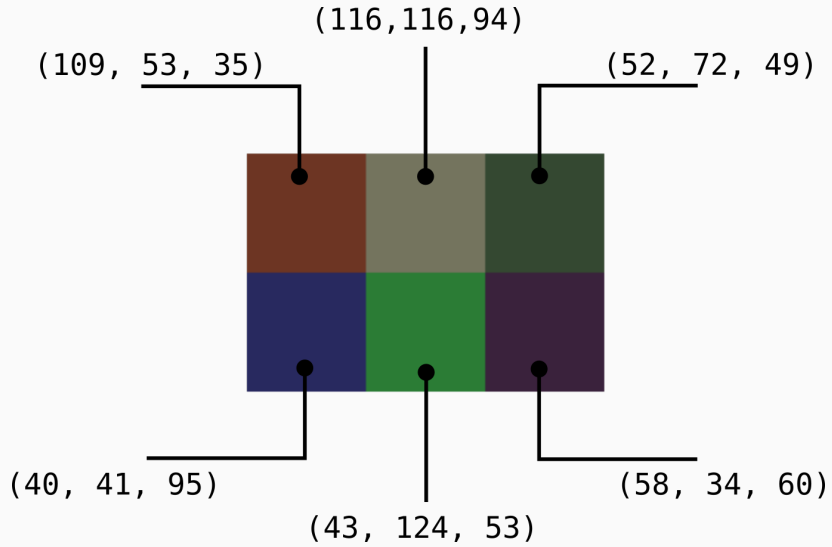
Чтение изображения формата ppm P3

```
...
std::vector<unsigned char> data(width * height);

for (int j = 0; j < height; ++j)
{
    for (int i = 0; i < width; ++i)
    {
        int ri, gi, bi;
        in >> r >> g >> b;
        unsigned char r = ri, g = gi, b = bi;
        data[3 * (j * width + i) + 0] = r;
        data[3 * (j * width + i) + 1] = g;
        data[3 * (j * width + i) + 2] = b;
     }
}
```

Варианты формата `ppm`.

- `P1`: На каждый пиксель приходится 1 бит информации
- `P2`: На каждый пиксель приходится 1 байт информации
- `P3`: На каждый пиксель приходится 3 байта информации
- `P4`: То же самое, что и `P1`, но в бинарном формате
- `P5`: То же самое, что и `P2`, но в бинарном формате
- `P6`: То же самое, что и `P3`, но в бинарном формате

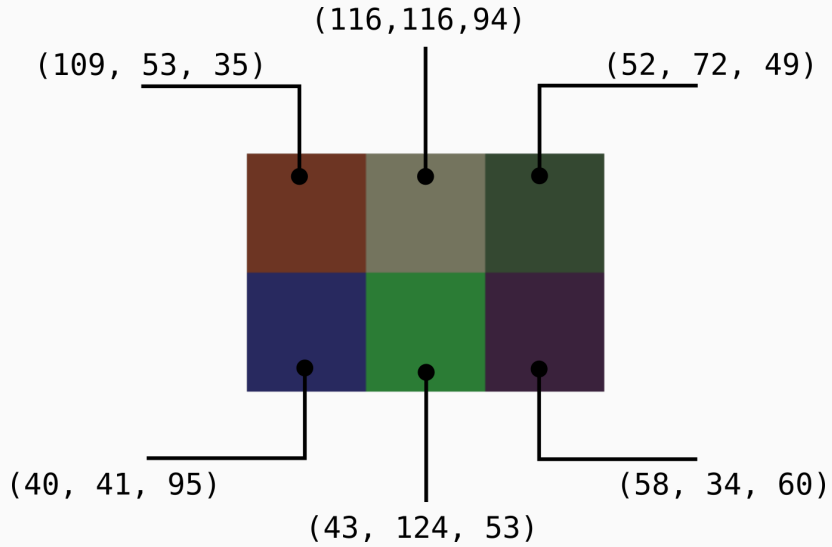
Бинарный формат .ppm изображения



Если изображение открыть в текстовом редакторе:

```
P6
3 2
255
m5#tt^4H1()_+|5:"<
```

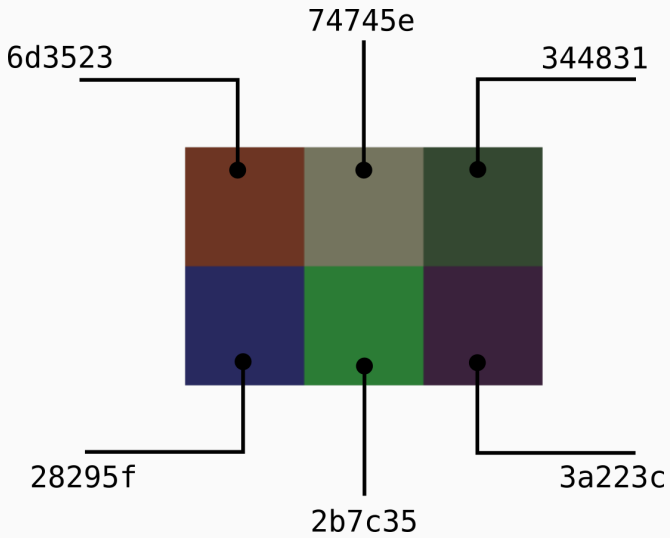
Бинарный формат .ppm изображения. Побайтово.



Байты файла изображения:

```
50 36 0a 33 20 32 0a 32
35 35 0a 6d 35 23 74 74
5e 34 48 31 28 29 5f 2b
7c 35 3a 22 3c
```

Бинарный формат .ppm изображения. Побайтово.



Байты файла изображения:

```
50 36 0a 33 20 32 0a 32
35 35 0a 6d 35 23 74 74
5e 34 48 31 28 29 5f 2b
7c 35 3a 22 3c
```

Запись изображения формата ppm P6

```
const int width = getWidth();
const int height = getHeight();
std::vector<unsigned char> data = getData();

std::ofstream out {"result.ppm", std::ios::binary};

out << "P6\n" << width << " " << height << "\n255\n";

out.write(reinterpret_cast<const char*>(&data[0]),
    data.size());
```

Чтение изображения формата ppm P6

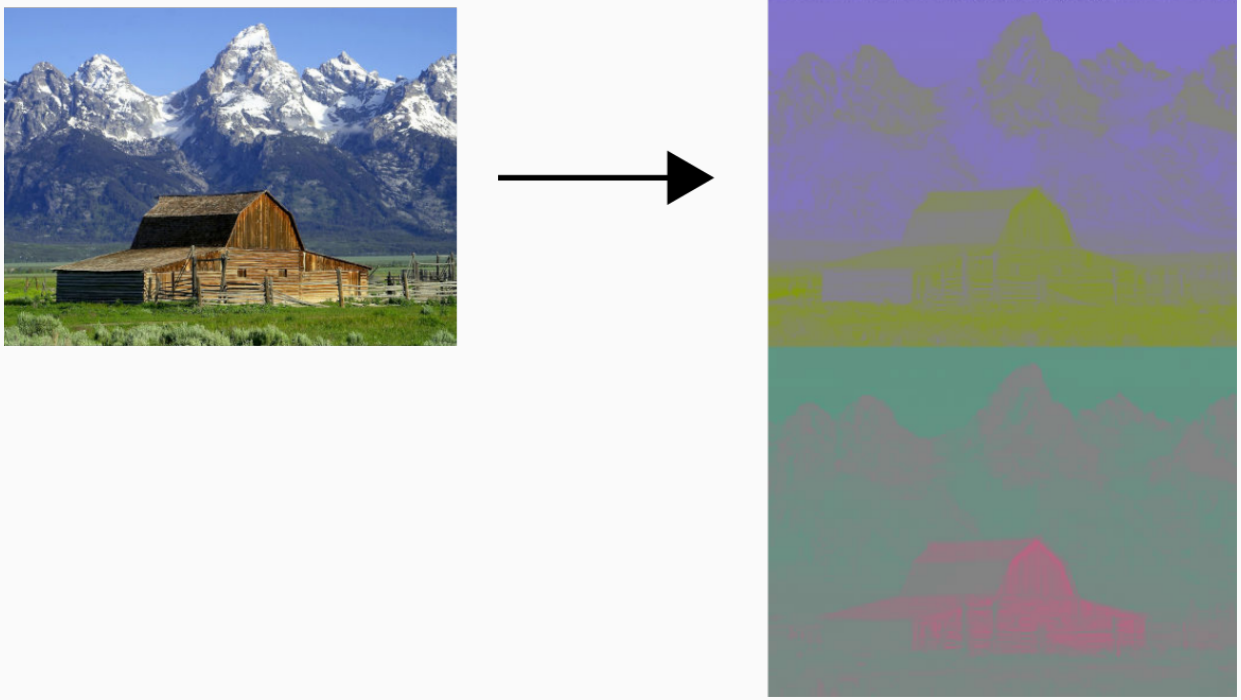
```
std::ifstream in {"image.ppm", std::ios::binary};
if (in.fail())
{
    std::cout << "Error. Can't open file!\n";
    std::exit(1);
}

std::string type;
in >> type;
if (type != "P6")
{
    std::cout << "Error. File should be type P6\n";
    std::exit(1);
}
...
```

Чтение изображения формата ppm P6

```
...
int width, height, maxValue;
in >> width >> height >> in >> maxValue;

char temp;
in >> std::noskipws >> temp;

std::vector<unsigned char> data(3 * width * height);
in.read(reinterpret_cast<char*>(&data[0]),
    data.size());
```

Формат jpeg

Конверсия из RGB в YCbCr



Конверсия из RGB в YCbCr

1. Перевод из RGB в YCbCr:

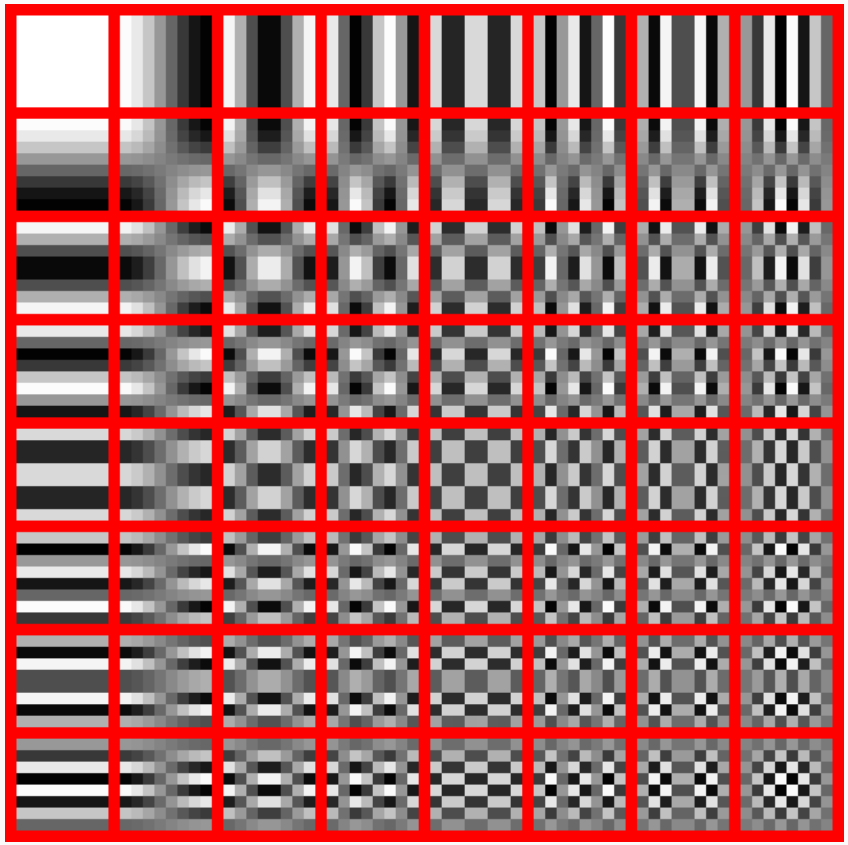
$$y = 0.299 \cdot r + 0.587 \cdot g + 0.114 \cdot b$$
$$cb = 128 - 0.169 \cdot r - 0.331 \cdot g + 0.5 \cdot b$$
$$cr = 128 + 0.5 \cdot r - 0.419 \cdot g - 0.081 \cdot b$$

2. Downsampling компонент Cb и Cr

Дискретное косинусное преобразование

3. Разбиваем изображение на блоки 8 на 8 пикселей.
4. Проводим дискретное косинусное преобразование для каждого блока.

Базис разложения:



❺ Для каждого блока 8 на 8 получится матрица примерно такого вида:

$$G = \begin{bmatrix} -415.38 & -30.19 & -61.20 & 27.24 & 56.12 & -20.10 & -2.39 & 0.46 \\ 4.47 & -21.86 & -60.76 & 10.25 & 13.15 & -7.09 & -8.54 & 4.88 \\ -46.83 & 7.37 & 77.13 & -24.56 & -28.91 & 9.93 & 5.42 & -5.65 \\ -48.53 & 12.07 & 34.10 & -14.76 & -10.24 & 6.30 & 1.83 & 1.95 \\ 12.12 & -6.55 & -13.20 & -3.95 & -1.87 & 1.75 & -2.79 & 3.14 \\ -7.73 & 2.91 & 2.38 & -5.94 & -2.38 & 0.94 & 4.30 & 1.85 \\ -1.03 & 0.18 & 0.42 & -2.42 & -0.88 & -3.02 & 4.12 & -0.66 \\ -0.17 & 0.14 & -1.07 & -4.19 & -1.17 & -0.10 & 0.50 & 1.68 \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \\ \downarrow v. \end{matrix}$$

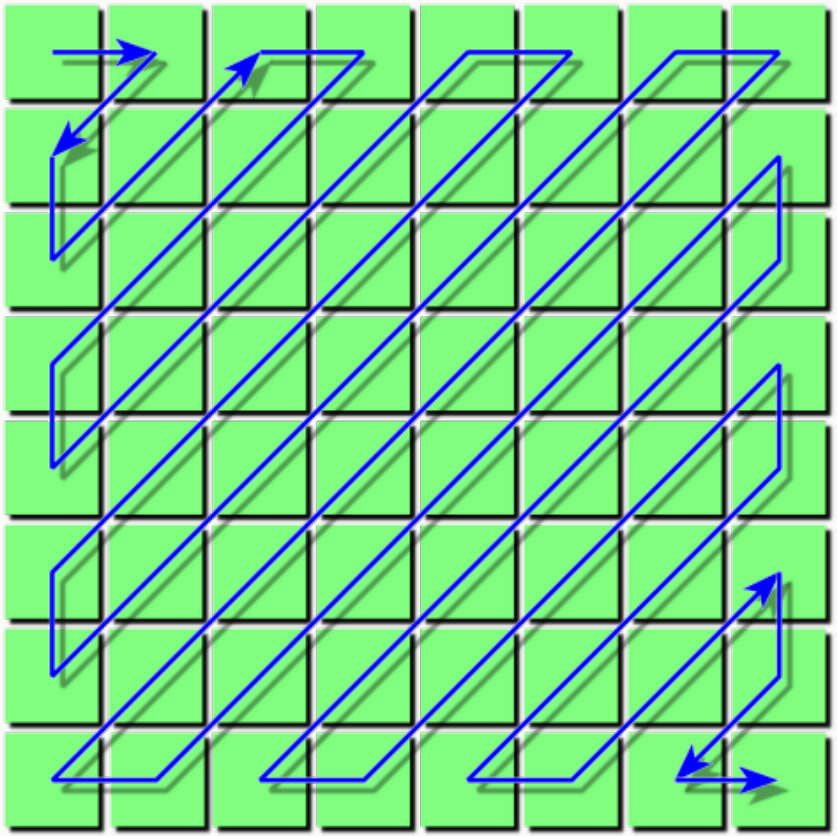
$u \longrightarrow$

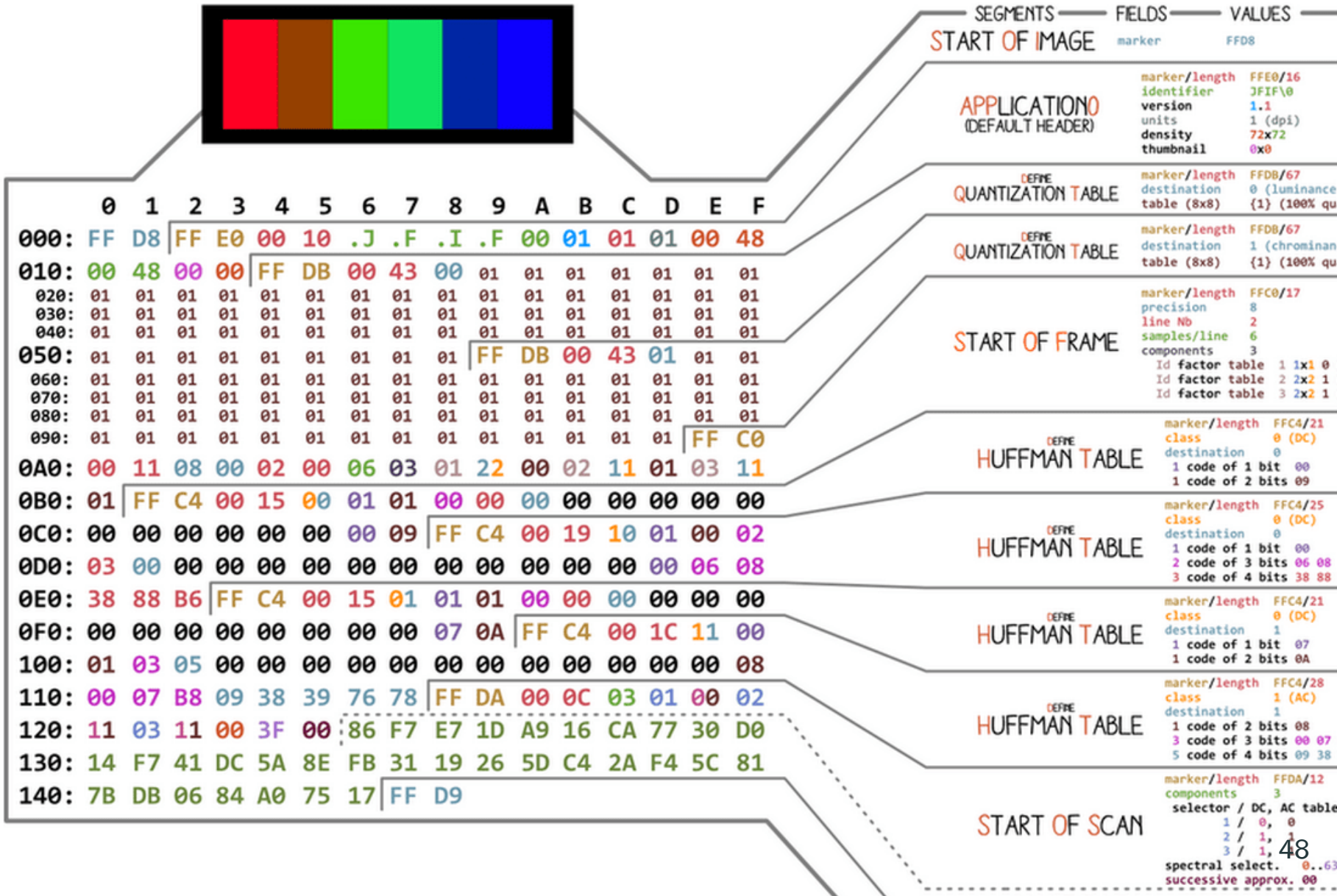
6. Делим каждый элемент матрицы на элемент матрицы квантизации Q. Получится матрица B.

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}.$$

$$B = \begin{bmatrix} -26 & -3 & -6 & 2 & 2 & -1 & 0 & 0 \\ 0 & -2 & -4 & 1 & 1 & 0 & 0 & 0 \\ -3 & 1 & 5 & -1 & -1 & 0 & 0 & 0 \\ -3 & 1 & 2 & -1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

7. Сохраняем матрицу B зигзагом, чтобы нули были сгруппированы.
8. Сжимаем полученные данные с помощью кода Хаффмана.




SEGMENTS
FIELDS
VALUES
START OF IMAGE
marker FFD8
APPLICATION0
(DEFAULT HEADER)
marker/length FFE0/16
identifier JFIF\0
version 1.1
units 1 (dpi)
density 72x72
thumbnail 0x0
DEFINE QUANTIZATION TABLE
marker/length FFDB/67
destination 0 (luminance
table (8x8) {1} (100% qu
DEFINE QUANTIZATION TABLE
marker/length FFDB/67
destination 1 (chrominan
table (8x8) {1} (100% qu
START OF FRAME
marker/length FFC0/17
precision 8
line Nb 2
samples/line 6
components 3
Id factor table 1 1x1 0
Id factor table 2 2x2 1
Id factor table 3 2x2 1
DEFINE HUFFMAN TABLE
marker/length FFC4/21
class 0 (DC)
destination 0
1 code of 1 bit 00
1 code of 2 bits 09
DEFINE HUFFMAN TABLE
marker/length FFC4/25
class 0 (DC)
destination 0
1 code of 1 bit 00
2 code of 3 bits 06 08
3 code of 4 bits 38 88
DEFINE HUFFMAN TABLE
marker/length FFC4/21
class 0 (DC)
destination 1
1 code of 1 bit 07
1 code of 2 bits 0A
DEFINE HUFFMAN TABLE
marker/length FFC4/28
class 1 (AC)
destination 1
1 code of 2 bits 08
3 code of 3 bits 00 07
5 code of 4 bits 09 38
START OF SCAN
marker/length FFDA/12
components 3
selector / DC, AC table
1 / 0, 0
2 / 1, 1
3 / 1, 1
spectral select. 0..63
successive approx. 00
0 1 2 3 4 5 6 7 8 9 A B C D E F
000: FF D8 FF E0 00 10 .J .F .I .F 00 01 01 01 00 48
010: 00 48 00 00 FF DB 00 43 00 01 01 01 01 01 01 01
020: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
030: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
040: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
050: 01 01 01 01 01 01 01 01 01 FF DB 00 43 01 01 01
060: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
070: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
080: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01 01
090: 01 01 01 01 01 01 01 01 01 01 01 01 01 01 FF C0
0A0: 00 11 08 00 02 00 06 03 01 22 00 02 11 01 03 11
0B0: 01 FF C4 00 15 00 01 01 00 00 00 00 00 00 00 00
0C0: 00 00 00 00 00 00 00 09 FF C4 00 19 10 01 00 02
0D0: 03 00 00 00 00 00 00 00 00 00 00 00 00 00 06 08
0E0: 38 88 B6 FF C4 00 15 01 01 01 00 00 00 00 00 00
0F0: 00 00 00 00 00 00 00 00 07 0A FF C4 00 1C 11 00
100: 01 03 05 00 00 00 00 00 00 00 00 00 00 00 00 08
110: 00 07 B8 09 38 39 76 78 FF DA 00 0C 03 01 00 02
120: 11 03 11 00 3F 00 86 F7 E7 1D A9 16 CA 77 30 D0
130: 14 F7 41 DC 5A 8E FB 31 19 26 5D C4 2A F4 5C 81
140: 7B DB 06 84 A0 75 17 FF D9

Библиотеку stb можно найти в `https://github.com/nothings/stb`

- `stb_image.h` библиотека для чтения изображения в форматах JPG, PNG, GIF, BMP и других.

```
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
```

- `stb_image_write.h` библиотека для чтения изображения в форматах JPG, PNG, GIF, BMP и других.

```
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"
```

Чтение изображения формата jpeg с помощью библиотеки stb

```
int width, height, channels;
unsigned char* img = stbi_load("russian_peasants.jpg",
    &width, &height, &channels, 0);

if(img == NULL)
{
     std::cout << "Error while loading image\n";
     exit(1);
}
```

Запись изображения формата jpeg с помощью библиотеки stb

```
int width = 400, height = 300, channels = 3;
std::vector<unsigned char> v = ...;

int success = stbi_write_jpg("test.jpg", width,
    height, channels, v.data, 100);
if(success == 0)
{
    std::cout << "Error while saving image\n";
    exit(1);
}
```

Создаём класс изображения

```
class Image
{
private:
    int mWidth  {0};
    int mHeight {0};
    std::vector<unsigned char> mData {};
public:
    Image() {}
    Image(const std::string& filename);
    Image(int width, int height);
    Image(int width, int height, Color c);

    int getWidth();
    int getHeight();
    unsigned char* getData();
    ...
```

```
    ...
    void setPixel(int i, int j, Color c);
    Color getPixel(int i, int j) const;

    Color& operator()(int i, int j);

    void loadPPMText(const std::string& fn);
    void savePPMText(const std::string& fn) const;
    void loadPPMBinary(const std::string& fn);
    void savePPMBinary(const std::string& fn) const;
    void loadPPM(const std::string& fn);
    void loadJPEG(const std::string& fn);
    void saveJPEG(const std::string& fn) const;
}
```

Класс Image. Вспомательный класс Color

```
class Image
{
public:

struct Color
{
    unsigned char r, g, b;

    Color& operator+=(const Color& c);
    Color operator+(const Color& c) const;
};

...

}
```

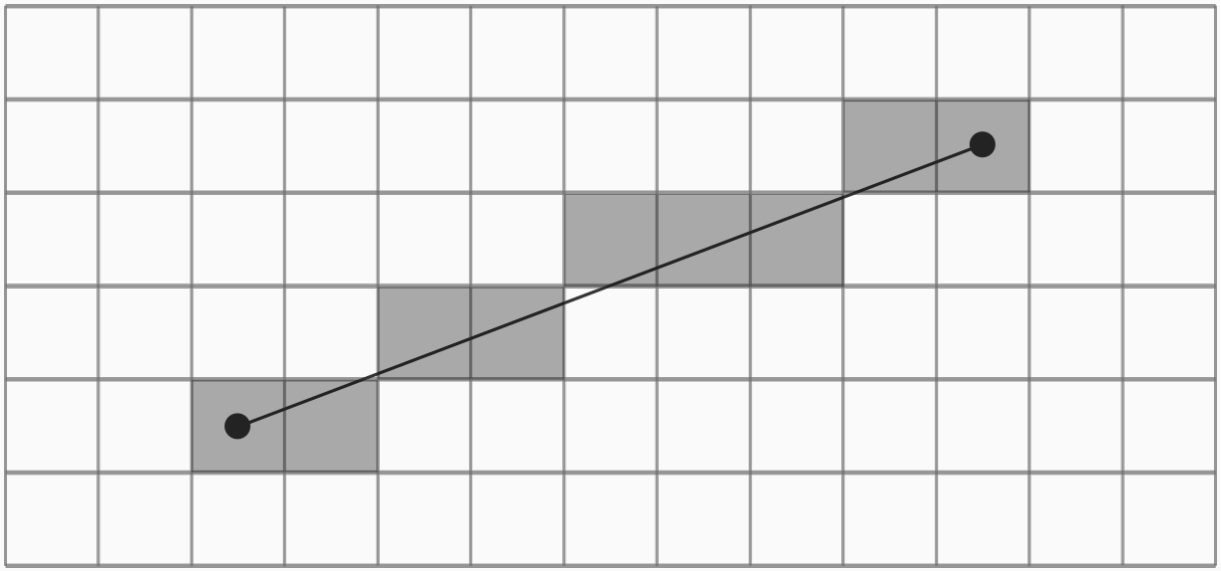
```
Image a {"input.ppm"};

for (int j = 0; j < a.getHeight(); ++j)
{
    for (int i = 0; i < a.getWidth(); ++i)
        a(i, j) += Image::Color{20, 20, 20};
}

b.savePPMBinary("output.ppm");
```

Рисование на изображении. Алгоритм Брезенхема.

Рисуем прямоугольник

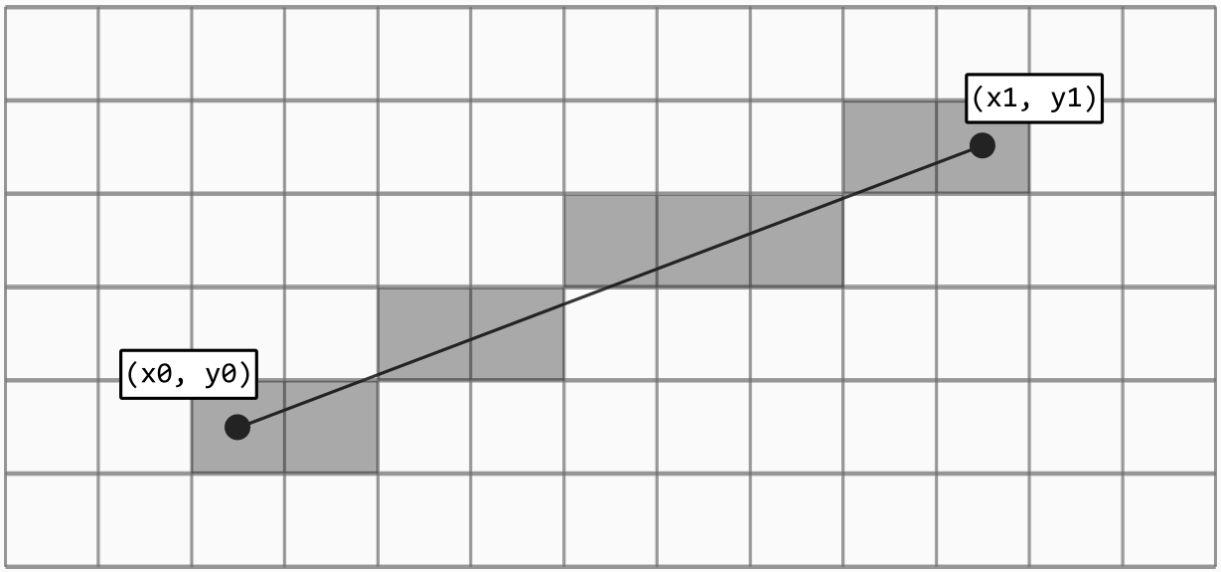
```
Image a {"input.ppm"};

for (int j = 300; j < 400; ++j)
{
    for (int i = 100; i < 300; ++i)
        a(i, j) = Image::Color{255, 0, 0};
}

b.savePPMBinary("output.ppm");
```

Как нарисовать линию?



Как нарисовать линию?

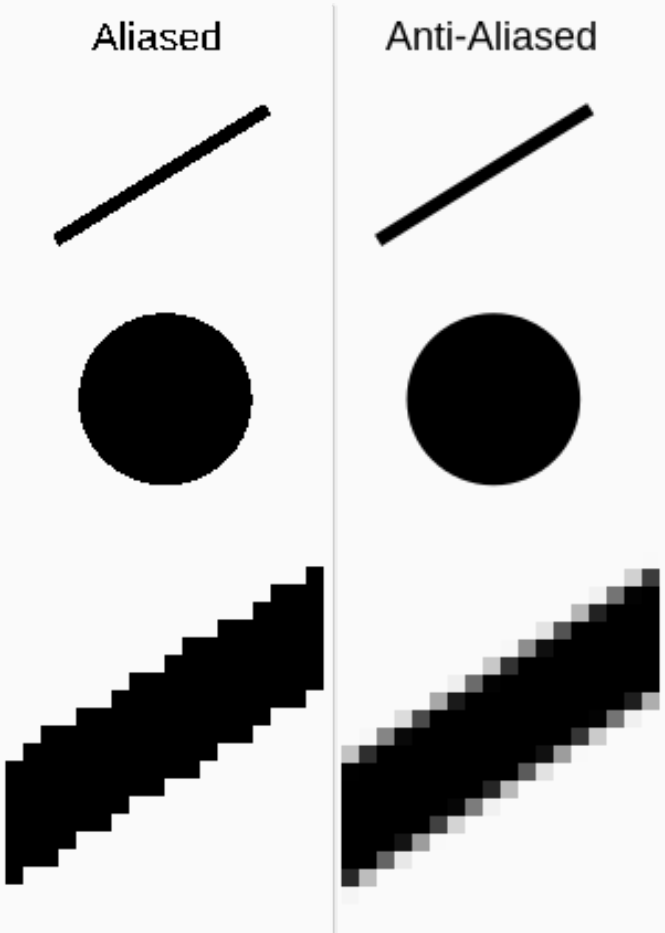
$$y = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) + y_0$$



Рисуем линию (углы от -45 до +45 градусов с Ox)

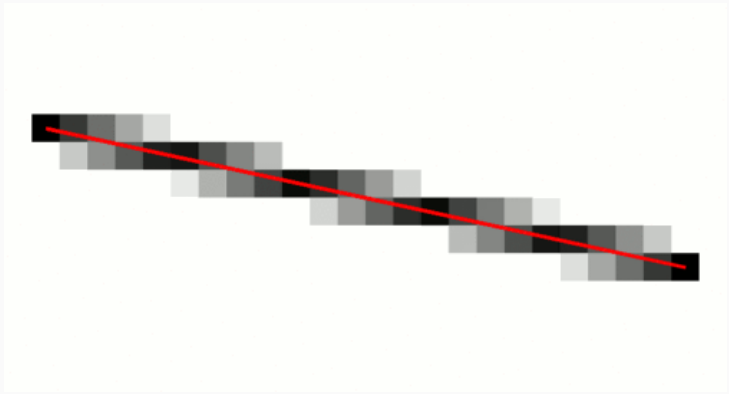
```
void line(Image& im, int x0, int y0, int x1, int y1) {
    int dx = abs(x1 - x0), dy = abs(y1 - y0);
    int diry = y1 - y0;
    if (diry > 0) diry = 1;
    if (diry < 0) diry = -1;

    int y = y0, error = 0;
    for (int x = x0; x < x1; ++x) {
        im(x, y) = Image::Color{0, 0, 0};
        error += dy;
        if (error >= dx) {
            y += diry;
            error -= dx;
        }
    }
}
```

Антиалиасинг

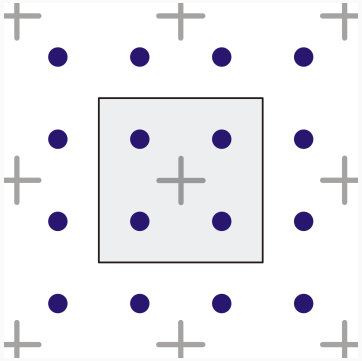


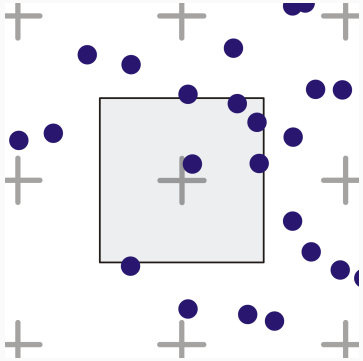
Алгоритмы для устранения антиалиасинга

- Алгоритм By



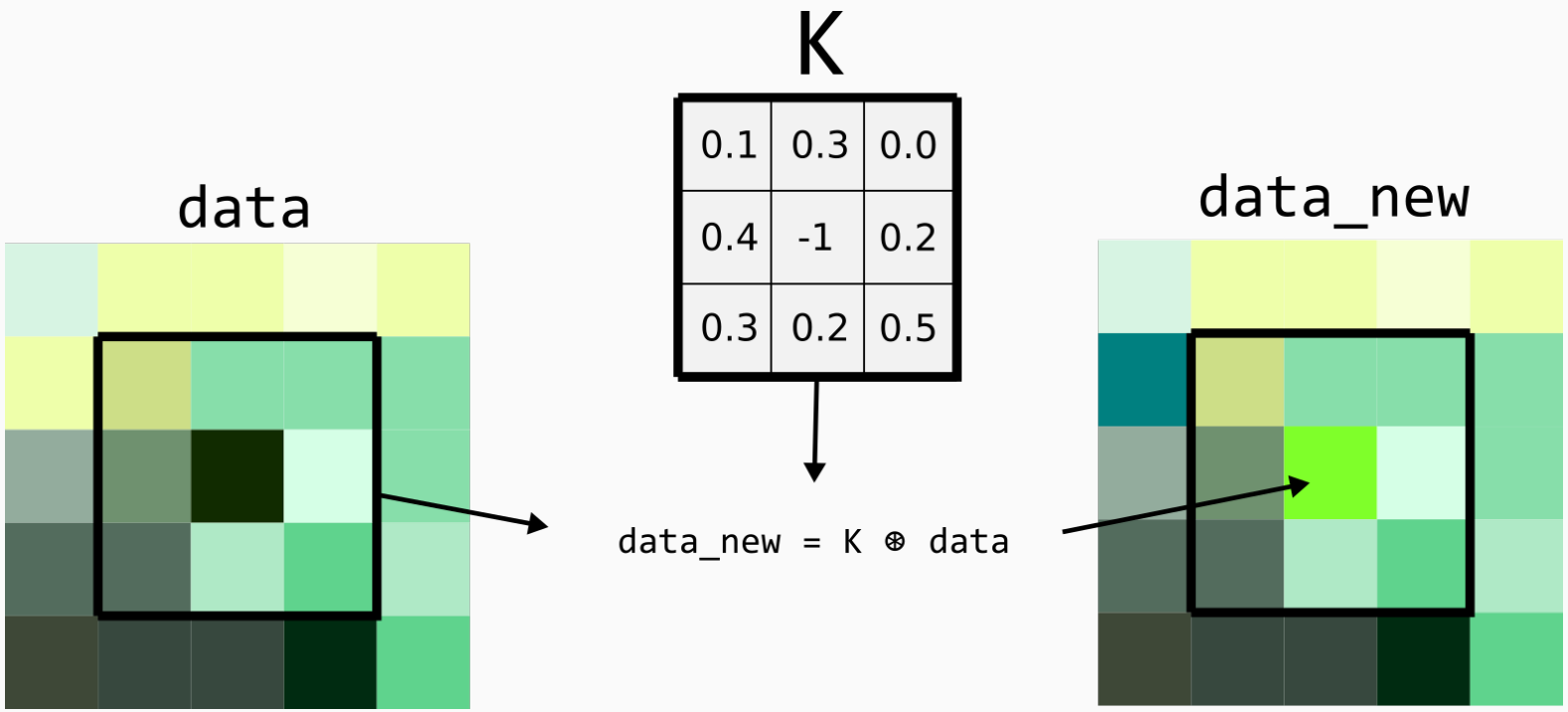
- Supersampling





Другие алгоритмы.

Свёртка



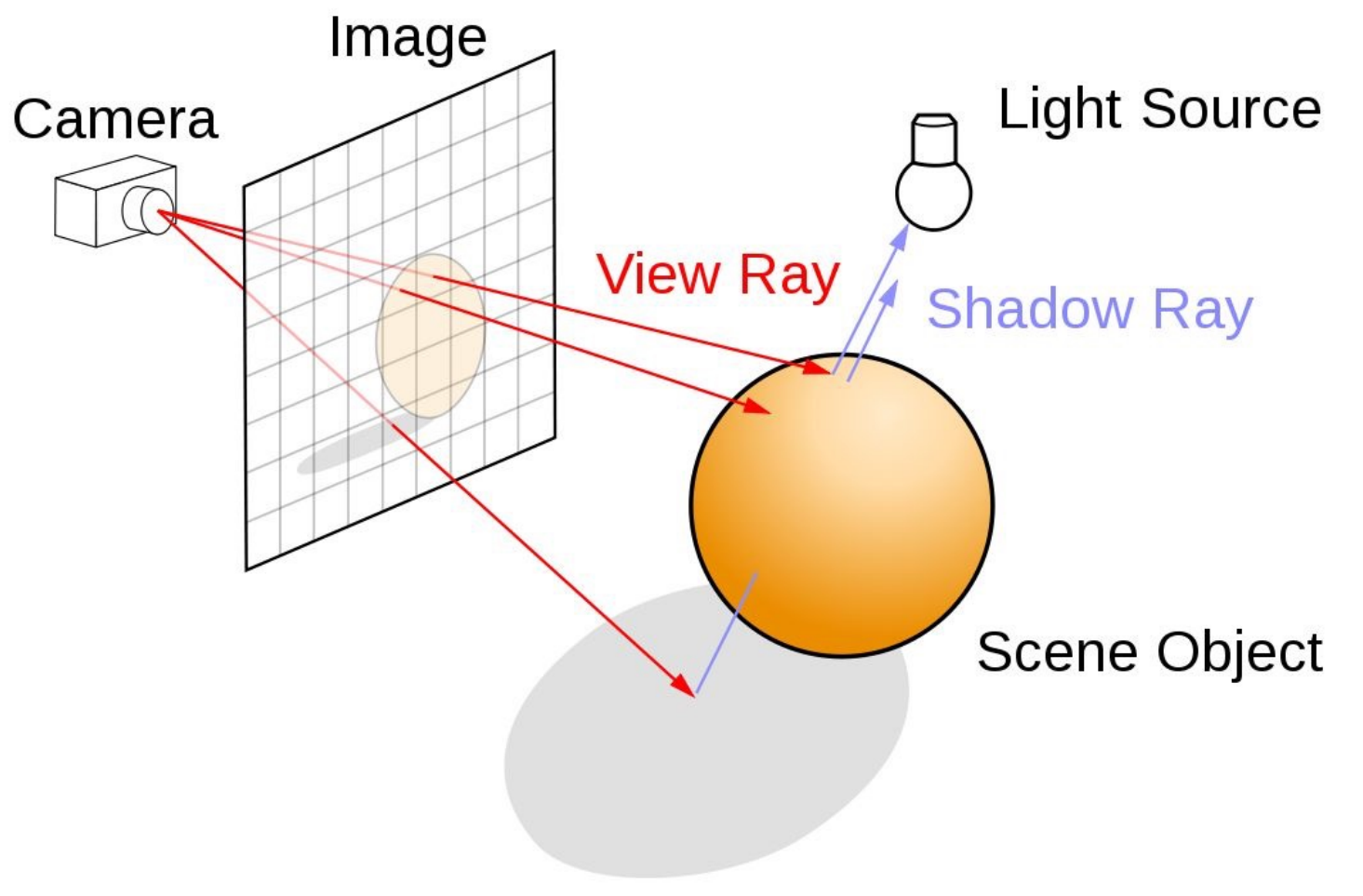


Image
Camera
Light Source
View Ray
Shadow Ray
Scene Object

Создание видео с помощью ffmpeg.

Создание видео с помощью ffmpeg

- Сначала создадим все кадры видео в виде jpeg картинок. Одна картинка на один кадр.
- Потом используем консольную программу `ffmpeg`:

```
ffmpeg -framerate 30 -pattern_type glob -i '*.jpg'
       -c:v libx264 -pix_fmt yuv420p out.mp4
```