

# Лабораторная работа №2. Цветовые Пространства

## Закрепление знаний

В первой лабораторной вы научились открывать изображение и пробегаться по нему циклом для создания различного рода фильтров. Прежде чем мы перейдем к работе с цветовыми пространствами, нам нужно закрепить навык **математического преобразования** данных. Мы реализуем три классических фильтра, каждый из которых демонстрирует свой тип математической операции над цветом.

### Сепия (Линейная алгебра).

Исторически сепия — это способ химического тонирования фотографий с использованием чернил каракатиц для увеличения долговечности отпечатков. Побочным эффектом данной обработки был коричнево-охряный цвет фотографий. В цифровой обработке этот эффект имитируется путем линейного смешивания цветовых каналов.

В отличие от перевода в оттенки серого которое мы делали в прошлой лабораторной работе, сепия требует вычисления новых значений на основе всех трех исходных компонентов (у серого полученный коэффициент использовался для всех каналов) . Здесь мы смешиваем информацию из соседних каналов с определенными весовыми коэффициентами.

*Формула №1. Формула эффекта сепии.*

$$\begin{aligned} pixelRedNew &= 0.393 * pixelRed + 0.769 * pixelGreen + 0.189 * pixelBlue \\ pixelGreenNew &= 0.349 * pixelRed + 0.686 * pixelGreen + 0.168 * pixelBlue \\ pixelBlueNew &= 0.272 * pixelRed + 0.534 * pixelGreen + 0.131 * pixelBlue \end{aligned}$$

где *pixelRed*, *pixelGreen*, *pixelBlue* - изначальное значение соответствующего цветового канала пикселя;

*pixelRedNew*, *pixelGreenNew*, *pixelBlueNew* - новое значение цветового канала пикселя.

Те цифры, которые мы используем в формуле — это **эмпирический стандарт** (W3C standard), который вывели инженеры Microsoft, чтобы имитировать тот самый химический оттенок на RGB мониторах. Это просто матрица смешивания, подобранная "на глаз", чтобы было похоже на старину.

### Постеризация (Квантование).

В цифровой обработке сигналов этот процесс называется **квантованием**, но в графике прижилось название **постеризация**. Термин пришел из печатной графики: при создании плакатов (постеров) использование большого количества красок было дорогим, поэтому художники ограничивали палитру несколькими цветами, превращая плавные градиенты в четкие, заметные переходы.

В формате RGB каждый цветовой канал может принимать 256 значений (от 0 до 255).

Постеризация искусственно уменьшает это число. Вместо 256 оттенков красного мы разрешаем, например, только 4. С математической точки зрения мы разбиваем непрерывный диапазон значений [0, 255] на *N* интервалов и приводим все значения внутри интервала к одному числу.

*Формула №2. Расчет интервала*

$$\text{Interval} = \frac{255}{\text{Level} - 1}$$

где *Interval* - ширина шага постерилизации;

*Level* - количество уровней постерилизации;

*Формула №3. Преобразование постерилизации.*

$$\text{pixelChannelNew} = \frac{\text{pixelChannel}}{\text{Interval}} * \text{Interval}$$

где *pixelChannelNew* - новое значение цветового канала пикселя;

*pixelChannel* - изначальное значение цветового канала пикселя;

*Interval* - ширина шага постерилизации полученная из предыдущей формулы;

При реализации важно следить за тем, чтобы знаменатель при вычислении интервала не был равен нулю (то есть *Levels > 1*).

### **Гамма-коррекция (Нелинейная яркость)**

Человеческий глаз и цифровая камера воспринимают свет по-разному. Матрица камеры линейна: если фотонов попадает в два раза больше, сигнал становится в два раза сильнее. Человеческий глаз же имеет нелинейную (логарифмическую) чувствительность: мы гораздо лучше различаем оттенки в тенях, чем в ярком свете.

Кроме того старые ЭЛТ-мониторы отображали сигнал нелинейно: при подаче 50% напряжения экран светился не на 50% яркости, а значительно темнее (примерно на 20%). Эта зависимость описывается степенной функцией. Чтобы компенсировать это затемнение, изображение необходимо предварительно "высветлить". Этот процесс и называется **гамма-коррекцией**.

*Формула №4. Гамма-коррекция.*

$$\text{pixelChannelNew} = 255 * \left( \frac{\text{pixelChannel}}{255} \right)^{\frac{1}{\gamma}}$$

где *pixelChannelNew* - новое значение цветового канала пикселя;

*pixelChannel* - изначальное значение цветового канала пикселя;

*γ* - коэффициент коррекции.

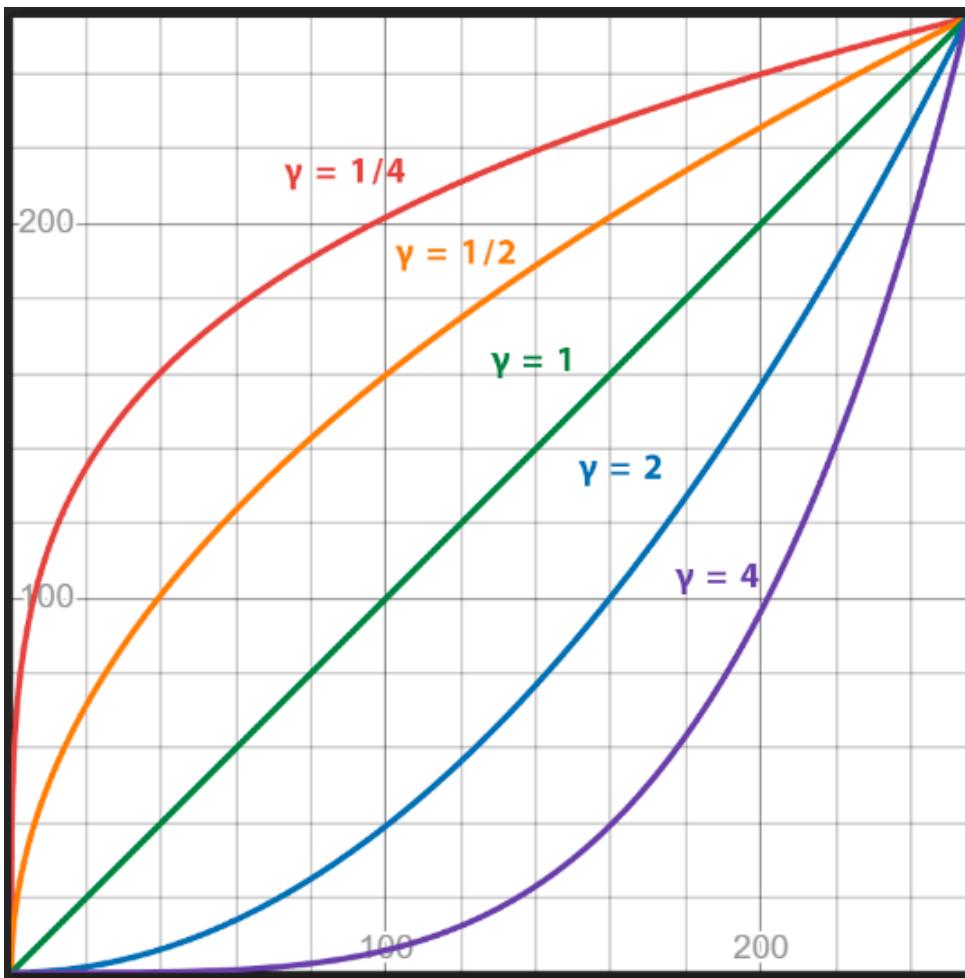


Рисунок 1 - Кривая изменения яркости при различных коэффициентах  $\gamma$

Степень  $1/\gamma$  это классическая формулировка коррекции. Мы компенсируем гамму монитора, поэтому степень обратная. Это делает формулу более интуитивной: больше гамма больше деталей в тенях.

### Задача №1.

Реализовать в программе следующие функции и фильтры:

- Фильтр сепии;
- Фильтр постеризации с настраиваемым количеством уровней;
- Функция гамма-коррекции изображения.

# Цветовые пространства

## RGB

Секрет системы RGB заключается в особенностях строения человека - человеческом зрении. Сетчатка человеческого глаза содержит три типа рецепторов (колбочек), чувствительных к длинным (красным), средним (зеленым) и коротким (синим) световым волнам. Любой цвет, который мы видим — это комбинация возбуждения этих трех рецепторов.

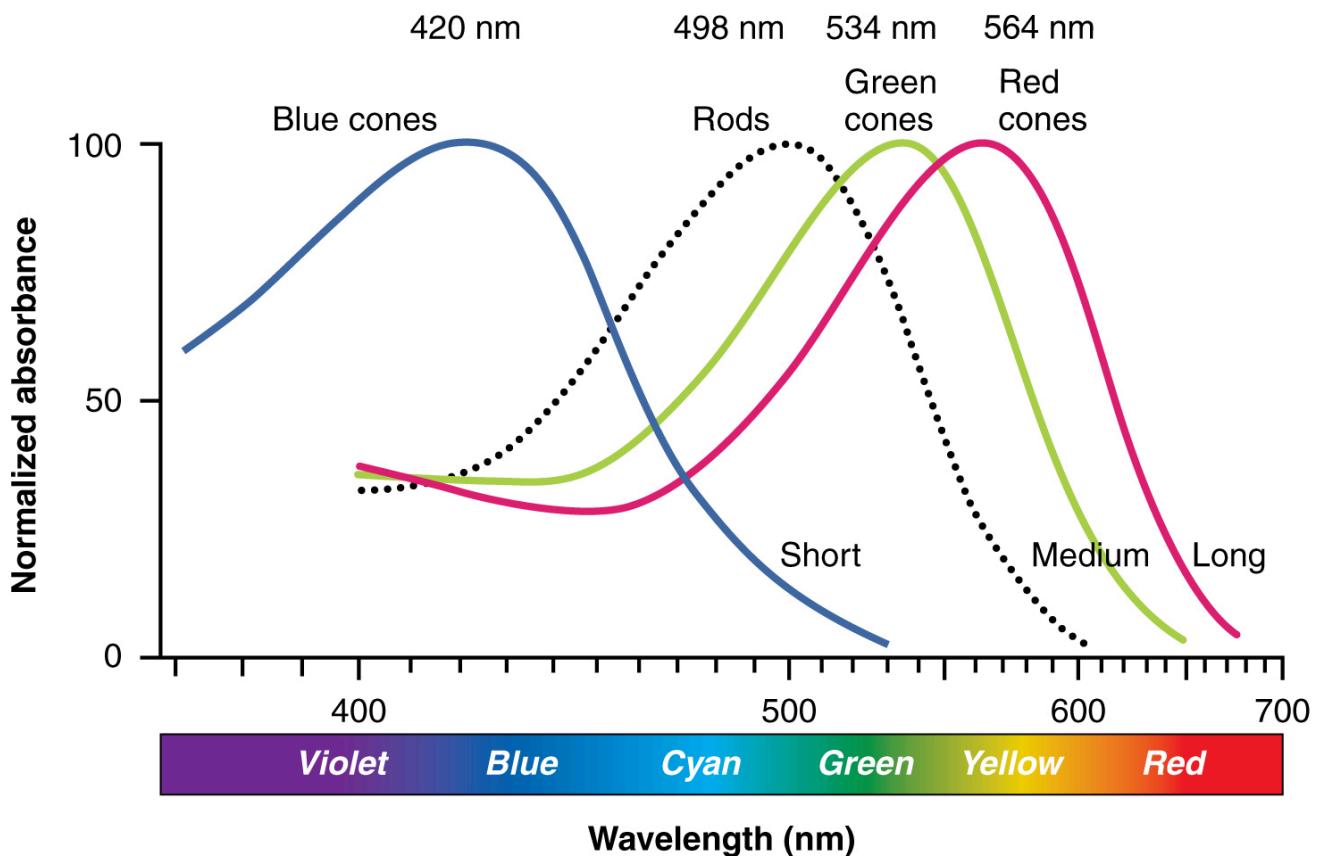


Рисунок 2 - График чувствительности колбочек к волнам разной длины.

Таким образом человеческий мозг сам по себе работает в режиме RGB. Любой цвет, который мы видим — это просто сумма сигналов с этих трех датчиков. Создавая систему RGB инженеры в первую очередь опирались на это свойство.

Второй важной особенностью в применении RGB в технологиях визуализации является то как в этой модели получается цвет. Существует два способа получить цвет:

- **Аддитивная модель** - все начинается с черного цвета в который добавляется свет для получения нового цвета. Максимум света приводит к белому цвету. **RGB является аддитивной моделью.**
- **Субтрактивная модель** - все начинается с белого цвета от которого отнимается свет. Вычитание всего света приводит к черному цвету. Таковой моделью является CMY, применяемая в основном в полиграфии.

Подавляющее большинство мониторов работает **излучая** свет, поэтому наиболее практичным решением было использовать аддитивную модель.

Так же один пиксель монитора физически состоит из трех субпикселей: красного, зеленого и синего. Хранение изображения в формате RGB позволяет передавать данные на видеокарту и монитор без дополнительных преобразований.

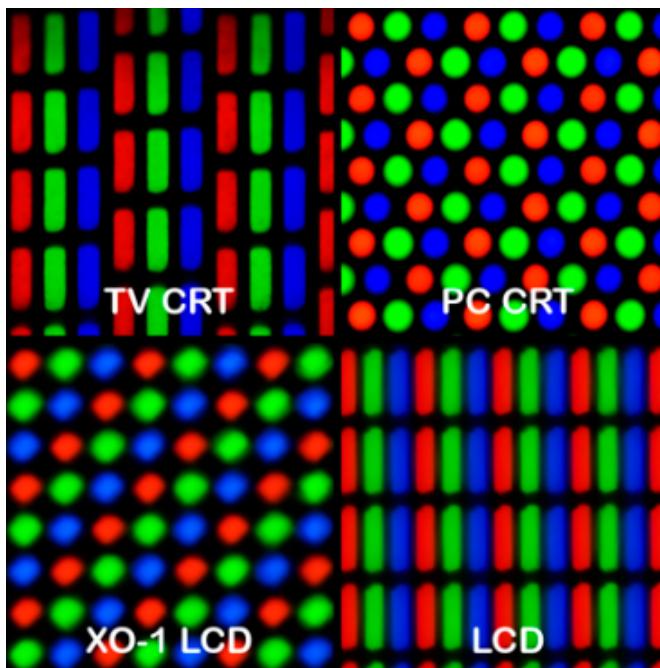


Рисунок №3. Расположение пикселей в различных стандартах пиксельных матриц .

**RGB** — это аппаратный формат. Он идеален для хранения и отображения, но, как вы узнаете далее, даже будучи основанным на человеческом зрении данный формат в действительности неудобен для редактирования цвета человеком.

## Контраст

Как было сказано в предыдущей лабораторной работе - та реализация что была ранее является наивной и не соответствует современным стандартам реализации контраста в изображениях.

Суть контраста заключается в том чтобы сделать светлое - светлее, а темное - темнее. В наивном варианте мы увеличивали и уменьшали все значения. Чтобы сделать правильный контраст, нам нужна **опорная точка** (середина диапазона). Для 8-битного изображения (0-255) серединой считается 128.

Формула №5. Контраст со смещением в средние тона.

$$pixelChannelNew = (pixelChannel - 128) * Contrast + 128$$

где  $pixelChannelNew$  - новое значение цветового канала пикселя;

$pixelChannel$  - изначальное значение цветового канала пикселя;

$Contrast$  - коэффициент контраста.

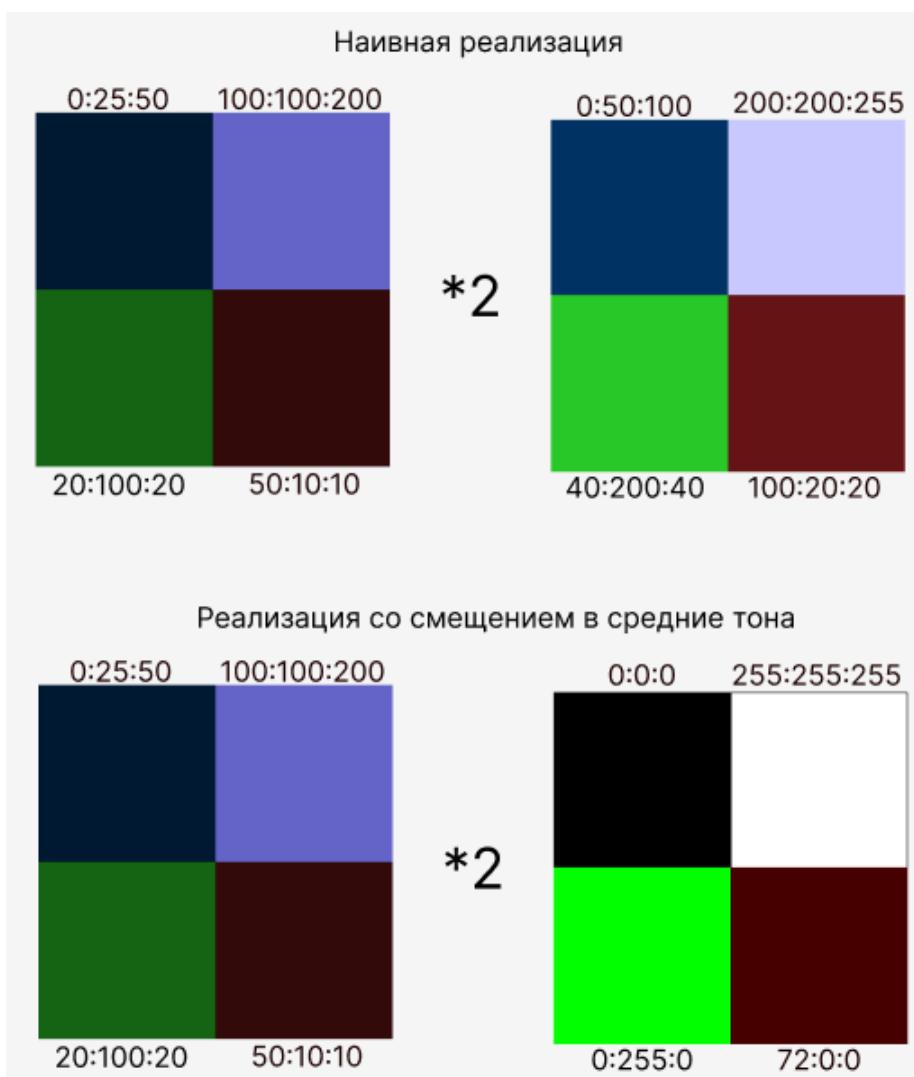


Рисунок №4. Сравнение результатов работы двух алгоритмов контраста.

Именно так работает контраст в большинстве графических редакторов. Мы меняем динамический диапазон относительно среднего тона, не сбивая общую экспозицию кадра.

## Насыщенность

Чтобы понять теорию насыщенности, нужно представить цветовую модель не как набор цифр, а как геометрию. RGB можно представить как куб. В одном углу черный (0,0,0), в другом белый (255,255,255), в остальных углах лежат цвета. Проведя линию от черного угла к белому мы получим линию серых оттенков или **ахроматическую ось**.

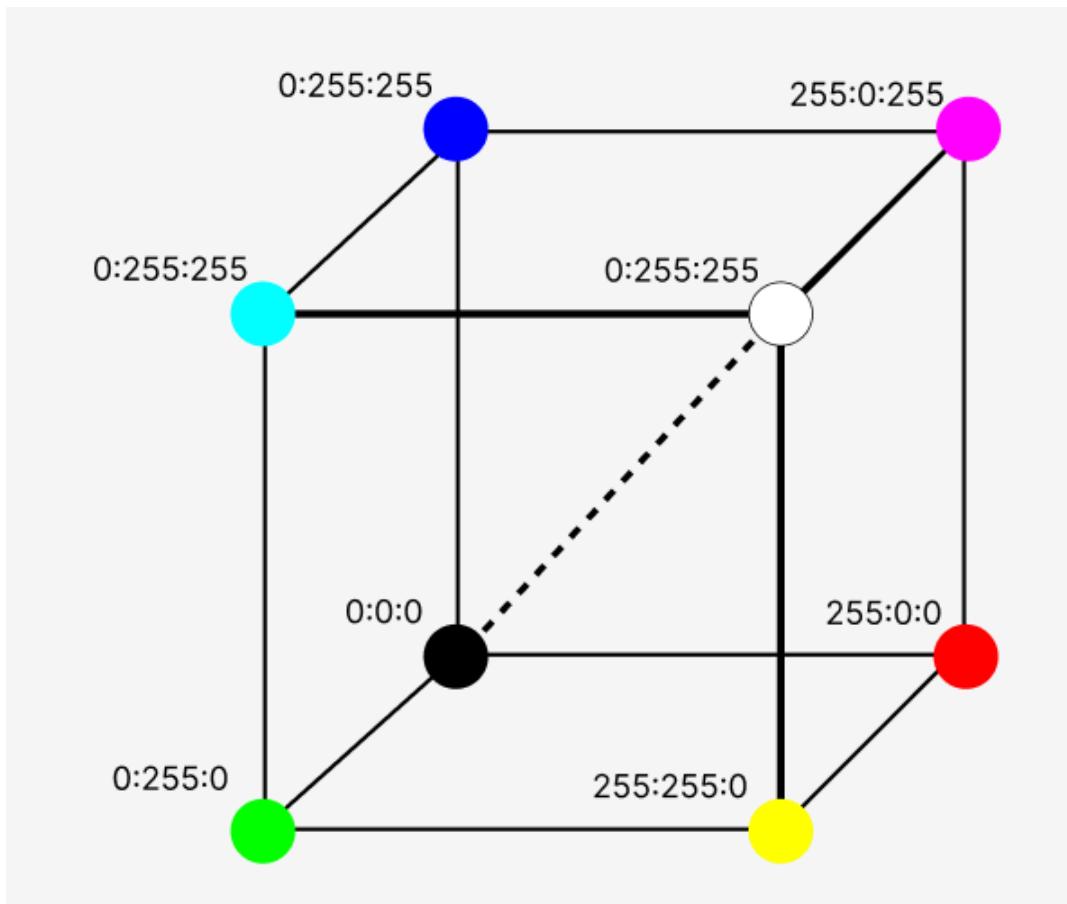


Рисунок №5. Куб RGB с ахроматической осью (пунктир).

**Насыщенность** - это **расстояние** от точки цвета до ахроматической оси. Чем ближе цвет к оси тем он менее насыщенный или "грязный", чем дальше - тем наоборот выглядит "чище".

Сложность алгоритма контраста в формате RGB заключается в том что сначала нужно определить на сколько пиксель приближен к оси серого:

1. **Поиск оси.** На первом этапе требуется понять каким серым цветом станет пиксель. Для этого воспользуемся формулой Luma из предыдущей лабораторной работы:

*Формула №6. Конвертация цветного изображения в черно-белое по формуле Luma Rec. 601.*

$$\text{grayPixel} = \text{originalPixel}.Red * 0.299 + \text{originalPixel}.Green * 0.587 + \text{originalPixel}.Blue * 0.114$$

где *grayPixel* - результат в формате byte,

*originalPixel* - оригинальный пиксель изображения;

2. Смешивание. Теперь когда у нас есть две известные точки (оригинальный пиксель и серый) можно найти результат смешения пикселя с коэффициентом насыщенности:

*Формула №7. Насыщенность со смешением к ахроматической оси.*

$$\text{pixelChannelNew} = (\text{pixelChannel} - \text{grayPixel}) * \text{Saturation} + \text{grayPixel}$$

где *pixelChannelNew* - новое значение цветового канала пикселя;

*pixelChannel* - изначальное значение цветового канала пикселя;

*grayPixel* - значение серого пикселя полученное на предыдущем шаге;

*Saturation* - коэффициент контраста.

Можно заметить что формула насыщенности и контраста являются идентичными, с той разницей что в контрасте изображение приводится к единому среднему, а в случае с насыщенностью требуется вычислить искомое среднее для каждого отдельного пикселя.

## **Задача №2.**

Реализовать следующий функции:

- Изменение контрастности изображение методом смещения к среднему;
- Изменение насыщенности изображения.

# HSV

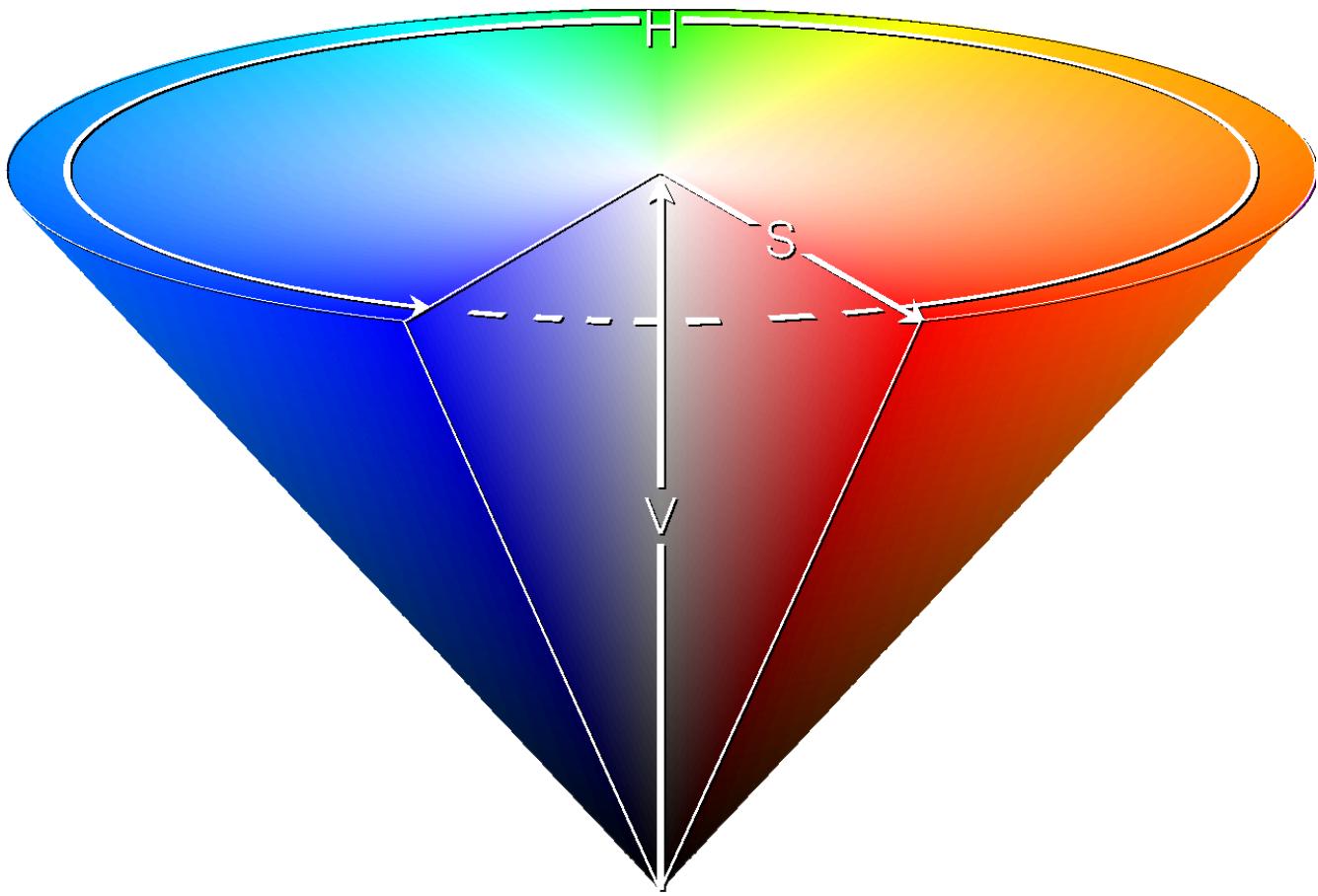
## Зачем он нужен и как это работает

Представьте такую ситуацию - вы художник который работает с изображениями. Вам дали задачу: "сделайте закат на изображении более оранжевым". Если мы работаем с изображением в формате RGB мы должны добавить побольше красного и убавить зеленый, а синий канал не трогать. Когда вы работаете с RGB вы работаете не с цветом, вы работаете с каналами, что для человеческого восприятия является неудобным и неинтуитивным.

Как было показано ранее RGB - это куб. В углах этого куба находятся чистые цвета, а что внутри? Внутри находятся смеси чистых цветов и чтобы понимать какой цвет получить нужно мыслить о цвете как о трехмерном пространстве. Люди думают о цвете иными категориями: "какой цвет?", "насколько он сочный?" и "насколько он яркий?". Таким образом нам нужна цветовая система координат которая будет соответствовать человеческому мышлению.

Таковой системой координат является цветовое пространство **HSV**. Если RGB это куб, то HSV это конус:

- По кругу идет **H (Hue)** — сам цвет (0 - красный, 120 - зеленый и т.д.). Так как цвет расположен в кругу значения для цвета имеют значения от 0 до 360. Но зачастую значение сокращается до 180 для того чтобы вмещаться в значение `byte`, а так же может быть записано от 0.0 до 1.0 если запись идет в `float` или `double`.
- От центра к краю идет **S (Saturation)** — насыщенность цвета, значение записывается от 0.0 до 1.0 или от 0 до 255 в зависимости от типа данных.
- Снизу вверх идет **V (Value)** — яркость, значение записывается от 0.0 до 1.0 или от 0 до 255 в зависимости от типа данных.



*Рисунок №6. Модель HSV представлена в виде конуса.*

Теперь задача "сделать закат оранжевым" решается изменением одного ползунка **Hue**, а задача "сделать фото черно-белым" — обнулением одного ползунка **Saturation**.

## Работа с HSV в Emgu.CV

Первое что нам потребуется это конвертация из формата `Image<Bgr, byte>` в формат `Image<Hsv, byte>` и обратно. Для этого у класса `Image` существует встроенная функция `Convert<TColor, TDepth>` который позволяет преобразовать изображение в любое цветовое пространство определенное в `Emgu.CV`.

*Блок кода №1. Конвертация изображений встроенным методом.*

```
//Конвертация BGR (byte) -> HSV (byte)
Image<Hsv, byte> hsvImage = sourceImage.Convert<Hsv, byte>();

//Конвертация BGR (byte) -> Gray (float)
//Значения яркости будут в диапазоне 0.0 - 1.0
Image<Gray, float> grayFloat = sourceImage.Convert<Gray, float>();
```

Важно знать что большинство конвертаций между сложными пространствами (например из `Hsv` в `Lab`) выполняется через промежуточное пространство `Bgr`. Это связано с тем что нет методик прямого перевода из одного формата в другой и исключением является `RGB` на основе которого и

создаются другие цветовые пространства. Такая конвертация может приводить к незначительным погрешностям округления и потере данных.

Так же при смене типа данных ( TDepth ) происходит автоматическое масштабирование значений:

- byte в float : Значения делятся на 255 (диапазон становится 0.0 - 1.0)
- float в byte : Значения умножаются на 255 и округляются.

Если в формате Bgr обращение к Data имело зависимость индексов: 0-Синий, 1-Зеленый, 2-Красный, то в формате Hsv данные индексы отведены под соответствующие каналы:

1. Data[y, x, 0] — Hue (Тон);
2. Data[y, x, 1] — Saturation (Насыщенность);
3. Data[y, x, 2] — Value (Яркость/Интенсивность).

Как отмечалось ранее в теории Ние измеряется в **градусах** от 0 до 360. Байт вмещает максимум **255**. 360 в 255 не влезает, поэтому каждая единица в записи соответствует двум градусам в кругу тона.

*Блок кода №2. Работа с каналами HSV через свойство Data.*

```
//Создание изображения в формате HSV
Image<Hsv, byte> hsvImage = new Image<Hsv, byte>(640, 480);

//Обращение к каналу Тона (Hue)
hsvImage.Data[y, x, 0] = (byte)(128);

//Это значение некорректно для OpenCV, так как Hie > 179. Визуальный результат будет
//непредсказуем.
hsvImage.Data[y, x, 0] = (byte)(255);

//Обращение к каналу Насыщенности (Saturation)
hsvImage.Data[y, x, 1] = (byte)(200);

//Обращение к каналу Яркости (Value)
hsvImage.Data[y, x, 2] = (byte)(128);
```

Другим вариантом обращения к каналам является работа с конкретными пикселями. Мы можем получить пиксель в качестве объекта класса `Hsv` у которого есть свойства `Hue`, `Saturation` и `Value`.

*Блок кода №3. Работа с пикселем в формате HSV.*

```
//Получение пикселя в виде объекта класса Hsv
Hsv pixel = hsvImage[y, x];

//Изменение тона
pixel.Hue = (byte)(128);

//Изменение насыщенности.
pixel.Satuation = (byte)(200);

//Изменение яркости
pixel.Value = (byte)(128);

//Запись пикселя обратно в изображение
hsvImage[y, x] = pixel;
```

### **Задача №3.**

Реализовать следующий функции:

- Работа с изображением в формате HSV, возможность изменять каждый канал. Тон должен быть зациклен (как например при 179+5 в результате должно стать 4, так и при 4-5 результат должен стать 179).
- Функции изменения контрастности. Для этого используйте формулу контрастности со сдвигом применяя к каналу яркости (`Value`).

## **Задание на лабораторную работу №2. Редактор цветовых пространств.**

В данной лабораторной работе требуется доработать редактор созданный в предыдущей лабораторной работе добавив в него следующие функции:

1. Фильтры сепии и постерилизации;
2. Функция гамма-коррекции изображения;
3. Изменить алгоритм контраста на алгоритм со сдвигом к среднему;
4. Функция изменения насыщенности в RGB формате.;
5. Функции изменения каналов тона, насыщенности и яркости в HSV изображении. Канал тона должен быть зацикленным (как например при  $179+5$  в результате должно стать 4, так и при  $4-5$  результат должен стать 179);
6. Реализовать функцию изменения контраста в формате HSV. Для этого используйте формулу контрастности со сдвигом применяя к каналу яркости (Value).

### **Полезные ссылки**

1. GitHub репозиторий с учебным проектом: <https://github.com/TheSkyEye1/AOCI-Lab2-Color-Space-Editor>
2. GitHub релиз проекта демонстрирующий весь функционал реализуемый в работе: <https://github.com/TheSkyEye1/AOCI-Lab2-Color-Space-Editor/releases/tag/v1.0.0>
3. Документация Emgu.CV: [https://www.emgu.com/wiki/index.php?title>Main\\_Page](https://www.emgu.com/wiki/index.php?title>Main_Page)
4. Структура Bgr в Emgu.CV: <https://www.emgu.com/wiki/files/4.4.0/document/html/fe167025-bd25-405e-7891-af1efecf39b8.htm>
5. Структура Hsv в Emgu.CV: <https://www.emgu.com/wiki/files/4.1.0/document/html/f2702b16-9a20-93bb-6821-56aa452955b8.htm>

### **Дополнительная информация**

#### **Оптимизация с помощью LUT (Look-up Table)**

Операции вроде гамма-коррекции или экспоненты очень дороги для процессора. Однако, так как значения каналов ограничены диапазоном  $[0, 255]$ , мы можем вычислить результат для каждого возможного значения заранее (всего 256 раз) и записать их в массив.

Вместо того чтобы считать формулу внутри цикла по пикселям, мы просто берем готовое значение из массива по индексу: `newVal = lut[oldVal]`. Это ускоряет обработку в десятки раз.

*Блок кода №4. Подготовка и применение Lut для гамма-коррекции.*

```
// Пример подготовки LUT для гаммы
byte[] gammaLut = new byte[256];
for (int i = 0; i < 256; i++) {
    double normalized = i / 255.0;
    double result = Math.Pow(normalized, 1.0 / gamma) * 255.0;
    gammaLut[i] = (byte) Math.Max(0, Math.Min(255, result));
}
```

```
// Использование внутри цикла по картинке  
pixel.Red = gammaLut[pixel.Red]; // Мгновенный доступ!
```