Color Topics for Programmers

Peter Occil

This version of the document is dated 2025-02-15.

Peter Occil

1 Introduction

This document presents an overview of many common color topics that are of general interest to programmers and that can be implemented in many different programming languages. **Sample Python code**[1] **that implements many of the methods in this document is available. Supplemental topics**[2] are listed in another open-source page.

Topics this document covers include:

- Red-green-blue (RGB) and other color models of practical interest.
- How to generate colors with certain properties.
- Color differences, color maps, and color mixing.
- Dominant colors of an image.
- Colors as spectral curves.

This document does not cover:

- How to change or set colors used—
  - in text, foregrounds, or backgrounds of user-interface elements (such as buttons, text boxes, and windows),
  - in text or backgrounds of documents (such as HTML documents), or
  - when generating graphics (such as plots and charts).
- Determining which colors are used, or used by default, in user-interface elements, documents, plots, or charts.
- Color pickers, including how to choose colors with them.
- Specifics on setting and getting pixel, palette, and other colors in images (including screenshots) with the exception of finding dominant colors.
- Colorization of command line outputs, or terminal or shell outputs. **"ANSI" graphic codes**[3] are discussed elsewhere.
- In general, topics that are specific to a programming language or application programming interface.

2 Contents

[1] https://peteroupc.github.io/colorutil.zip
[2] https://peteroupc.github.io/suppcolor.html
[3] https://peteroupc.github.io/suppcolor.html#Terminal_Graphics

3 Notation and Definitions

- The **pseudocode conventions**[4] apply to this document.
- **bpc.** Bits per color component, also known as bits per color channel.
- **CIE.** French initials for the International Commission on Illumination.
- **Color model.** Describes, in general terms, the relationship of colors in a theoretical space.
- **Color space.** A mapping from colors to numbers that follows a particular color model.
- **D50 illuminant, D65 illuminant.** CIE models of daylight at a correlated color temperature of about 5000 or 6500 kelvins respectively.[5]
- **D50/2 white point.** The white point determined by the D50 illuminant and the CIE 1931 standard observer.
- **D65/2 white point.** The white point determined by the D65 illuminant and the CIE 1931 standard observer.
- **Image color list.** Means either—
  - a list of colors (which can have duplicates), all of the same color space, or
  - the colors (which can have duplicates) used in a raster image's pixels, a vector image, a three-dimensional image, a digital video, or a digital document.
- **ISO.** International Organization for Standardization.
- **Light source.** Means a ***primary light source*** or an ***illuminant***, as defined by the CIE. Roughly means an emitter of light, or radiation describing an emitter of light.
- **RGB.** Red-green-blue.

4 Overview of Color Vision

Color[6] is possible only if three things exist, namely—

- *light*,
- an *object* receiving that light (a surface, for example), and
- an *observer* viewing that object and interpreting the light received from it.

Because of this, color does not exist in light, in objects receiving light, in light sources, or even in the signals generated by the eyes when they see things.[7] In the Opticks, I. Newton said, "the Rays to speak properly are not coloured."

Color appearance is subjective — since interpreting the light is required — and varies with the *light source* (sunlight, daylight, incandescent light, etc.), *object* (material), *observer*, viewing situation, or a combination of these.[8]

> **Note:** The three things that together make color possible — *light*, *object*, and *observer* — can be modeled by curves that span the *visible spectrum* (the part of the electromagnetic spectrum

[4] https://peteroupc.github.io/pseudocode.html

[5] The CIE publishes **tabulated data** for the D65 illuminant and the CIE 1931 and 1964 standard observers at its Web site. In some cases, the CIE 1931 standard observer can be approximated using the methods given in Wyman, Sloan, and Shirley, **"Simple analytic approximations to the CIE XYZ color matching functions"**, Journal of Computer Graphics Techniques 2(2), 2013, pp. 1-11.

[6] This overview has none of the heavy baggage from color teachings involving "red, yellow, and blue", "primary/secondary/tertiary" colors, or using a "color wheel" to "predict" color mixtures. Also deliberately missing are discussions on color psychology, "color forecasting", or color in natural language, all topics that are generally irrelevant in programming.

[7] It's not accurate to speak of "red light", "green light", "blue light", "white light", and so on.

[8] Color perception is influenced by the three things that make color possible: *Light.* For example, natural daylight and sunlight change how they render colors depending on time of day and year, place, and weather. *Objects.* A material's surface properties such as gloss, transparency, haze, and more affect color perception. *Observers.* Different observers "see" colors differently due to aging, culture, defective color vision, personal experience, kind of observer (human, camera, lens, animal, etc.), and more. B. MacEvoy documents the **wide observer variation** even among people with normal color vision. For a detailed overview on phenomena involving human color vision, see section 9 of Kirk, R., "Standard Colour Spaces", **FilmLight Technical Note**, version 4.0, 2004-2018. https://www.filmlight.ltd.uk/support/documents/other/legacy_tl.php

in which light is "seen"), as described in the section "**Spectral Color Functions**".

4.1 Human Color Vision

When a person views an object, the light it reflects reaches that person's eyes.

The human eye has an inner back lining (called the *retina*) filled with three kinds of *cones*, and each kind of cone is differently sensitive to light.

The human visual system compares the responses it receives from the cones and converts them to three kinds of signals, namely a light–dark signal and the two *opponent signals* red/green and blue/yellow. It's these signals, and not the cone responses, that are passed to the brain.[9]

The human brain interprets the signals from the eyes to judge color appearance, taking into account the visual situation. One process involved in this is called *adaptation*, in which the human visual system, roughly speaking, treats the brightest thing in the scene as "white" and mentally adjusts the rest of the colors it sees accordingly, to account for differences in lighting. Adaptation is thus similar to a digital camera's "auto white balance".

Notes:

1. The cone responses can be described by three overlapping "curves" that peak at different places in the visible spectrum — in fact, two of these curves span the entire visible spectrum. As a result, at least two of the three kinds of cones will react to light, not just one by itself.
2. Because there are three kinds of cones, three numbers are enough to uniquely identify a color humans can see — which is why many **color spaces**[10] are 3-dimensional, such as RGB or CIE XYZ spaces.

4.2 Defective and Animal Color Vision

Defective color vision, including so-called **"colorblindness"**[11], can make certain kinds of light harder to distinguish than is the case with normal color vision.[12]

In addition to humans, many other animals possess color vision to a greater or lesser extent. As an extreme example, the **mantis shrimp**[13] has at least twelve different cone types, making its color vision considerably sharper than humans'.

5 Specifying Colors

A color can be specified in one of two ways:

- **As a point in space**, that is, as a small set of numbers (usually three numbers) showing where the color lies in a color space. This is the usual practice. Some color spaces include the following:
  - **RGB** color spaces describe proportions of "red", "green", and "blue" dots of light.
  - **HSV**, **HSL**, and **HWB** color spaces transform RGB colors to make their presentation more intuitive, but are not perception-based.
  - **XYZ**, **CIELAB**, and **CIELUV** color spaces are based on human color perception.
  - **CMYK** color spaces are especially used to describe proportions of four specific kinds of ink.
  - $\mathbf{Y'C_BC_R}$ is especially used in video encoding.

[9] For example, the light–dark signal is roughly the sum of the three cone responses. The theory of opponent colors is largely due to E. Hering's work and was reconciled with the three-cone theory around the mid-20th century (for example, through work by Hurvich and Jameson).

[10] https://peteroupc.github.io/suppcolor.html#Kinds_of_Color_Spaces

[11] https://en.wikipedia.org/wiki/Color_blindness

[12] For information on how defective color vision can be simulated, see "**Color Blindness Simulation Research**", by "Jim".

[13] https://en.wikipedia.org/wiki/Mantis_shrimp

- **As a *spectral curve***, which gives the behavior of light across the electromagnetic spectrum (see "**Spectral Color Functions**"). Colors given as spectral curves, unlike colors in RGB or other color spaces, have the advantage that they are not specific to a lighting condition, whereas colors in a given color space assume a specific lighting, viewing, or printing condition.

6 RGB Color Model

The **red–green–blue (RGB) color model** is the most commonly seen color model in mainstream computer programming. The RGB model is ideally based on the intensity that "red", "green", and "blue" dots of light should have in order to reproduce certain colors on display devices.[14] The RGB model is a cube with one vertex set to "black", the opposite vertex set to "white", and the remaining vertices set to "red", "green", "blue", "cyan", "yellow", and "magenta".

RGB colors. An RGB color consists of three components in the following order: "red", "green", "blue".

RGBA colors. Some RGB colors also contain a fourth component, called the *alpha component*, which ranges from fully transparent to fully opaque. Such RGB colors are called *RGBA colors* in this document. RGB colors without an alpha component are generally considered fully opaque.

0-1 format. In this document, an RGB or RGBA color is in the **0-1 format** if all its components are 0 or greater and 1 or less. This document understands all RGB and RGBA colors to be in this format unless noted otherwise.

6.1 RGB Color Spaces

There are many **RGB color spaces**, not just one, and they generally differ in their red, green, blue, and white points and in their color component transfer functions (*"transfer functions"*):

- **Red, green, blue, and white points.** These are what a given RGB color space considers "red", "green", "blue", and "white", that is, what that space associates with the RGB colors (1, 0, 0), (0, 1, 0), (0, 0, 1), and (1, 1, 1), respectively. (The first three points are commonly called "primaries".) Each of these points need not be an actual color (this is illustrated by the **ACES2065-1 color space**, for example). Examples of "primaries" are Rec. 601 (NTSC), Rec. 709, and DCI-P3. Examples of white points are the D50/2 and D65/2 white points.
- **"Transfer function".** This is a function used to convert, component by component, a so-called ***linear RGB*** color to an ***encoded RGB*** (R' G' B') color in the same color space. Examples include the sRGB transfer function given **later**; *power-law* or *gamma* functions such as $c^{1/\gamma}$, where c is the red, green, or blue component and γ is a positive number; and the PQ and HLG functions.

In general, the same three numbers, such as (1, 0.5, 0.3), identify a different-appearing RGB color in different RGB color spaces. In this document, the only RGB color space described in detail is **sRGB**. (Lindbloom)[15] contains further information on many RGB color spaces.

Notes:

1. In this document, all techniques involving RGB colors apply to such colors in linear or encoded form, unless noted otherwise.

[14] Although most color display devices in the past used three dots per pixel ("red", "green", and "blue"), this may hardly be the case today. Nowadays, recent display devices and luminaires are likely to use more than three dots per pixel — such as "red", "green", "blue", and "white", or RGBW — and ideally, color spaces following the *RGBW color model*, or similar color models, describe the intensity those dots should have in order to reproduce certain colors. Such color spaces, though, are not yet of practical interest to most programmers outside the development of solid-state lighting, luminaires, or display devices, or of software to control them.

[15] B. Lindbloom, "**RGB Working Space Information**".

2. In the TV and film industries, some RGB color spaces, including sRGB, belong in the category of so-called *standard dynamic range* (*SDR*) color spaces, while others cover a wider range of colors (*wide color gamut* or *WCG*), a wider "brightness" range (*high dynamic range* or *HDR*), or both. (Mano 2018)[16] contains an introduction to WCG/HDR images. See also Rep. 2390-4, a more advanced overview, from the International Telecommunication Union.
3. RGB colors encoded in images and video or specified in documents are usually 8-bpc or 10-bpc *encoded RGB* colors.

6.2 sRGB

Among RGB color spaces, one of the most popular is the *sRGB color space*. In sRGB—

- the red, green, and blue points were chosen to cover the range of colors displayed by typical cathode-ray-tube displays (as in the high-definition standard **Rec. 709**[17]),
- the white point was chosen as the D65/2 white point, and
- the color component transfer function (implemented as `SRGBFromLinear` below) was based on the power-law (gamma) encoding used for cathode-ray-tube monitors.

For background, see the **sRGB proposal**[18], which recommends RGB image data in an unidentified RGB color space to be treated as sRGB.

The following methods convert colors between linear and encoded sRGB. (Note that the thresholds `0.0031308` and `0.4045` are those of IEC 61966-2-1, the official sRGB standard published by the International Electrotechnical Commission; the sRGB proposal has a different value for these thresholds.)

```
// Convert a color component from encoded to linear sRGB
// NOTE: This is not gamma decoding; it's similar to, but
// not exactly, c^2.2.  This function was designed "to
// allow for invertability in integer math", according to
// the sRGB proposal.
METHOD SRGBToLinear(c)
 // NOTE: Threshold here would more properly be
 // 12.92 * 0.0031308 = 0.040449936, but 0.04045
 // is what the IEC standard uses
  if c <= 0.04045: return c / 12.92
  return pow((0.055 + c) / 1.055, 2.4)
END METHOD

// Convert a color component from linear to encoded sRGB
// NOTE: This is not gamma encoding; it's similar to, but
// not exactly, c^(1/2.2).
METHOD SRGBFromLinear(c)
  if c <= 0.0031308: return 12.92 * c
  return pow(c, 1.0 / 2.4) * 1.055 - 0.055
END METHOD

// Convert a color from encoded to linear sRGB
METHOD SRGBToLinear3(c)
   return [SRGBToLinear(c[0]), SRGBToLinear(c[1]), SRGBToLinear(c[2])]
END METHOD
```

[16] Mano, Y., et al. "Enhancing the Netflix UI Experience with HDR", Netflix Technology Blog, Medium.com, Sep. 24, 2018.
[17] https://en.wikipedia.org/wiki/Rec._709
[18] https://www.w3.org/Graphics/Color/sRGB

```
// Convert a color from linear to encoded sRGB
METHOD SRGBFromLinear3(c)
   return [SRGBFromLinear(c[0]), SRGBFromLinear(c[1]), SRGBFromLinear(c[2])]
END METHOD
```

> **Note:** IEC 61966-2-1 defines a reference display where, among other things, encoded sRGB colors' components (c) are decoded using a power law of 2.2, that is, the decoding is $c2.2$. Indeed, this power law, and not an inverse of the sRGB color component transfer function, is what is **employed in practice**[19] by most computer displays today that can show, more or less, the range of colors supported by sRGB.

6.3 Representing RGB Colors

The following shows how linear or encoded RGB colors can be represented as integers or as text.

6.3.1 Binary Formats

RGB and RGBA colors are often expressed by packing their components as binary integers, as follows:

- **RGB colors:** With an RN-bit red component, a GN-bit green, and a BN-bit blue, resulting in an integer that's (RN + GN + BN) bits long.
- **RGBA colors:** With an RN-bit red component, a GN-bit green, a BN-bit blue, and an AN-bit alpha, resulting in an integer that's (RN + GN + BN + AN) bits long.

For both kinds of colors, the lowest value of each component is 0, and its highest value is $2^B - 1$, where B is that component's size in bits.

The following are examples of these formats:

- **5/6/5 RGB colors:** As 16-bit integers (5 bits each for red and blue, and 6 bits for green).
- **5-bpc:** As 15-bit integers (5 bpc [bits per color channel] RGB).
- **8-bpc:** As 24-bit integers (8 bpc RGB), or as 32-bit integers with an alpha component.
- **10-bpc:** As 30-bit integers (10 bpc RGB), or as 40-bit integers with an alpha component.
- **16-bpc:** As 48-bit integers (16 bpc RGB), or as 64-bit integers with an alpha component.

There are many ways to store RGB and RGBA colors in these formats as integers or as a sequence of 8-bit bytes. For example, the RGB color's components can be in "little-endian" or "big-endian" 8-bit byte order, or the order in which the color's components are packed into an integer can vary. This document does not seek to survey the RGB binary storage formats available.

The following pseudocode presents methods to convert RGB colors to and from different binary color formats (where RGB color integers are packed red/green/blue, in that order from lowest to highest bits):

```
METHOD round(x):
   if floor(x)<0.5: return floor(x)
   else: return ceil(b)
END METHOD

// Converts 0-1 format to N/N/N format as an integer.
METHOD ToNNN(rgb, scale)
   sm1 = scale - 1
   return round(rgb[2]*sm1) * scale * scale + round(rgb[1]*sm1) * scale +
         round(rgb[0]*sm1)
END METHOD
```

[19] https://github.com/dylanraga/win11hdr-srgb-to-gamma2.2-icm

```
// Converts N/N/N integer format to 0-1 format
METHOD FromNNN(rgb, scale)
   sm1 = scale - 1
   r = rem(rgb, scale)
   g = rem(floor(rgb / scale), scale)
   b = rem(floor(rgb / (scale * scale)), scale)
   return [ r / sm1, g / sm1, b / sm1]
END METHOD

METHOD To444(rgb): return ToNNN(rgb, 16)
METHOD To555(rgb): return ToNNN(rgb, 32)
METHOD To888(rgb): return ToNNN(rgb, 256)
METHOD To161616(rgb): return ToNNN(rgb, 65536)
METHOD From444(rgb): return FromNNN(rgb, 16)
METHOD From555(rgb): return FromNNN(rgb, 32)
METHOD From888(rgb): return FromNNN(rgb, 256)
METHOD From161616(rgb): return FromNNN(rgb, 65536)

METHOD To565(rgb)
   return round(rgb[2] * 31) * 32 * 64 + round(rgb[1] * 63) * 32 +
         round(rgb[0] * 31)
END METHOD

METHOD From565(rgb)
   r = rem(rgb, 32)
   g = rem(floor(rgb / 32.0), 64)
   b = rem(floor(rgb / (32.0 * 64.0)), 32)
   return [ r / 31.0, g / 63.0, b / 31.0]
END METHOD
```

6.3.2 HTML Format and Other Text Formats

A color string in the **HTML color format** (also known as "hex" format), which expresses 8-bpc RGB colors as text strings, consists of the character "#", two base-16 (hexadecimal) digits[20] for the red component, two for the green, and two for the blue, in that order.

For example, `#003F86` expresses the 8-bpc RGB color (0, 63, 134).

The following pseudocode presents methods to convert RGB colors to and from the HTML color format or the 3-digit variant described in note 1 to this section.

```
METHOD NumToHex(x)
    if hex < 0 or hex >= 16: return error
    hexlist=["0", "1", "2", "3", "4", "5", "6",
       "7", "8", "9", "A", "B", "C", "D", "E", "F"]
    return hexlist[x]
END METHOD

METHOD HexToNum(x)
    hexlist=["0", "1", "2", "3", "4", "5", "6",
       "7", "8", "9", "A", "B", "C", "D", "E", "F"]
    hexdown=["a", "b", "c", "d", "e", "f"]
```

[20]The base-16 digits, in order, are 0 through 9, followed by A through F. The digits A through F can be uppercase or lowercase.

```
    i = 0
    while i < 16
            if hexlist[i] == x: return i
            i = i + 1
    end
    i = 0
    while i < 6
            if hexdown[i] == x: return 10 + i
            i = i + 1
    end
    return -1
END METHOD

METHOD ColorToHtml(rgb)
   r = (rgb[0] * 255)
   g = (rgb[1] * 255)
   b = (rgb[2] * 255)
   if floor(r)<0.5: r=floor(r)
   else: r=ceil(r)
   if floor(g)<0.5: g=floor(g)
   else: g=ceil(g)
   if floor(b)<0.5: b=floor(b)
   else: b=ceil(b)
   return ["#",
     NumToHex(rem(floor(r/16),16)), NumToHex(rem(r, 16)),
     NumToHex(rem(floor(g/16),16)), NumToHex(rem(g, 16)),
     NumToHex(rem(floor(b/16),16)), NumToHex(rem(b, 16)),
   ]
END METHOD

METHOD HtmlToColor(colorString)
    if string[0]!="#": return error
    if size(colorString)==7
            r1=HexToNum(colorString[1])
            r2=HexToNum(colorString[2])
            g1=HexToNum(colorString[3])
            g2=HexToNum(colorString[4])
            b1=HexToNum(colorString[5])
            b2=HexToNum(colorString[6])
            if r1<0 or r2<0 or g1<0 or g2<0 or
                    b1<0 or b2<0: return error
            return [(r1*16+r2)/255.0,
                    (g1*16+g2)/255.0,
                    (b1*16+b2)/255.0]
    end
    if size(colorString)==4
            r=HexToNum(colorString[1])
            g=HexToNum(colorString[2])
            b=HexToNum(colorString[3])
            if r<0 or g<0 or b<0: return error
            return [(r*16+r)/255.0,
                    (g*16+g)/255.0,
```

```
                    (b*16+b)/255.0]
    end
    return error
END METHOD
```

Other text-based color formats include the following[21]:

- The **CSS Color Module Level 3**[22], which specifies this format, also mentions a **3-digit variant**, consisting of "#" followed by three base-16 digits, one each for the red, green, and blue components, in that order. Conversion to the 6-digit format involves replicating each base-16 component (for example, "#345" is the same as "#334455" in the 6-digit format).
- An **8-digit variant** used in the Android operating system consists of "#" followed by eight base-16 digits, two each for the alpha, red, green, and blue components, in that order. This variant thus describes 8-bpc RGBA colors.
- Additional formats are given in the **supplemental color topics**[23].

  Note: As used in the **CSS Color Module Level 3**, for example, colors in the HTML color format or its 3-digit variant are in the ***sRGB color space*** (as encoded RGB colors).

7 Transformations of RGB Colors

The following sections discuss popular color models for transforming RGB colors. The exact appearance of colors in these models varies by **RGB color space**.

7.1 HSV

HSV[24] (also known as HSB) is a color model that transforms RGB colors to make them easier to manipulate and reason with. An HSV color consists of three components, in the following order:

- *Hue* is an angle from red at 0 to yellow to green to cyan to blue to magenta to red.[25]
- A component called "saturation", the distance of the color from gray and white (but not necessarily from black), is 0 or greater and 1 or less.
- A component variously called "value" or "brightness" is the distance of the color from black and is 0 or greater and 1 or less.

The following pseudocode converts colors between RGB and HSV. The transformation is independent of RGB color space, but should be done using ***linear RGB* colors**.

```
METHOD RgbToHsv(rgb)
    mx = max(max(rgb[0], rgb[1]), rgb[2])
    mn = min(min(rgb[0], rgb[1]), rgb[2])
    // NOTE: "Value" is the highest of the
    // three components
    if mx==mn: return [0,0,mx]
    s=(mx-mn)/mx
    h=0
    if rgb[0]==mx
```

[21]The hue angle is in radians, and the angle is 0 or greater and less than 2 π . Radians can be converted to degrees by multiplying by `180 / pi`. Degrees can be converted to radians by multiplying by `pi / 180`.

[22]https://www.w3.org/TR/css3-color/#rgb-color

[23]https://peteroupc.github.io/suppcolor.html#Additional_Color_Formats

[24]https://en.wikipedia.org/wiki/HSL_and_HSV

[25]The hue angle is in radians, and the angle is 0 or greater and less than 2 π . Radians can be converted to degrees by multiplying by `180 / pi`. Degrees can be converted to radians by multiplying by `pi / 180`.

```
            h=(rgb[1]-rgb[2])/(mx-mn)
    else if rgb[1]==mx
            h=2+(rgb[2]-rgb[0])/(mx-mn)
    else
            h=4+(rgb[0]-rgb[1])/(mx-mn)
    end
    if h < 0: h = 6 - rem(-h, 6)
    if h >= 6: h = rem(h, 6)
    return [h * (pi / 3), s, mx]
END METHOD

METHOD HsvToRgb(hsv)
    hue=hsv[0]
    sat=hsv[1]
    val=hsv[2]
    if hue < 0: hue = pi * 2 - rem(-hue, pi * 2)
    if hue >= pi * 2: hue = rem(hue, pi * 2)
    hue60 = hue * 3 / pi
    hi = floor(hue60)
    f = hue60 - hi
    c = val * (1 - sat)
    a = val * (1 - sat * f)
    e = val * (1 - sat * (1 - f))
    if hi == 0: return [val, e, c]
    if hi == 1: return [a, val, c]
    if hi == 2: return [c, val, e]
    if hi == 3: return [c, a, val]
    if hi == 4: return [e, c, val]
    return [val, c, a]
END METHOD
```

Note: The HSV color model is not perception-based, as the HWB article acknowledges[26].

7.2 HSL

HSL[27] (also known as HLS), like HSV, is a color model that transforms RGB colors to ease intuition. An HSL color consists of three components, in the following order:

- *Hue* is the same for a given RGB color as in **HSV**.
- A component called "saturation" is the distance of the color from gray (but not necessarily from black or white), which is 0 or greater and 1 or less.
- A component variously called "lightness", "luminance", or "luminosity", is roughly the amount of black or white mixed with the color and is 0 or greater and 1 or less, where 0 is black, 1 is white, closer to 0 means closer to black, and closer to 1 means closer to white.

The following pseudocode converts colors between RGB and HSL. The transformation is independent of RGB color space, but should be done using ***linear RGB* colors**.

```
METHOD RgbToHsl(rgb)
    vmax = max(max(rgb[0], rgb[1]), rgb[2])
    vmin = min(min(rgb[0], rgb[1]), rgb[2])
```

[26]Smith, A.R. and Lyons, E.R., 1996. HWB—A more intuitive hue-based color model. Journal of graphics tools, 1(1), pp. 3-17.

[27]https://en.wikipedia.org/wiki/HSL_and_HSV

```
    vadd = vmax + vmin
    // NOTE: "Lightness" is the midpoint between
    // the greatest and least RGB component
    lt = vadd / 2.0
    if vmax==vmin: return [0, 0, lt]
    vd = vmax - vmin
    divisor = vadd
    if lt > 0.5: divisor = 2.0 - vadd
    s = vd / divisor
    h = 0
    hvd = vd / 2.0
    deg60 = pi / 3
    if rgb[0]==vmax
            h=((vmax-rgb[2])*deg60 + hvd) / vd
            h = h - ((vmax-rgb[1])*deg60+hvd) / vd
    else if rgb[2]==vmax
            h=pi * 4 / 3 + ((vmax-rgb[1])*deg60 + hvd) / vd
            h = h - ((vmax-rgb[0])*deg60+hvd) / vd
    else
            h=pi * 2 / 3 + ((vmax-rgb[0])*deg60 + hvd) / vd
            h = h - ((vmax-rgb[2])*deg60+hvd) / vd
    end
    if h < 0: h = pi * 2 - rem(-h, pi * 2)
    if h >= pi * 2: h = rem(h, pi * 2)
    return [h, s, lt]
END METHOD

METHOD HslToRgb(hsl)
    if hsl[1]==0: return [hsl[2],hsl[2],hsl[2]]
    lum = hsl[2]
    sat = hsl[1]
    bb = 0
    if lum <= 0.5: bb = lum * (1.0 + sat)
    if lum > 0.5: bb= lum + sat - (lum * sat)
    a = lum * 2 - bb
    hueval = hsl[0]
    if hueval < 0: hueval = pi * 2 - rem(-hueval, pi * 2)
    if hueval >= pi * 2: hueval = rem(hueval, pi * 2)
    deg60 = pi / 3
    deg240 = pi * 4 / 3
    hue = hueval + pi * 2 / 3
    hue2 = hueval - pi * 2 / 3
    if hue >= pi * 2: hue = hue - pi * 2
    if hue2 < 0: hue2 = hue2 + pi * 2
    rgb = [a, a, a]
    hues = [hue, hueval, hue2]
    i = 0
    while i < 3
       if hues[i] < deg60: rgb[i] = a + (bb - a) * hues[i] / deg60
       else if hues[i] < pi: rgb[i] = bb
       else if hues[i] < deg240
            rgb[i] = a + (bb - a) * (deg240 - hues[i]) / deg60
```

```
      end
      i = i + 1
    end
    return rgb
END METHOD
```

Notes:

- In some applications and specifications, especially where this color model is called HLS, the HSL color's "lightness" component comes before "saturation". This is not the case in this document, though.
- The HSL color model is not perception-based, as the HWB article acknowledges[28].

7.3 HWB

In 1996, the HWB model, which seeks to be more intuitive than HSV or HSL, was published[29]. An HWB color consists of three components in the following order:

- *Hue* is the same for a given RGB color as in **HSV**.
- *Whiteness*, the amount of white mixed to the color, is 0 or greater and 1 or less.
- *Blackness*, the amount of black mixed to the color, is 0 or greater and 1 or less.

The conversions given below are independent of RGB color space, but should be done using ***linear RGB* colors**.

- To convert an RGB color `color` to HWB, generate `[RgbToHsv(color)[0], min(min(color[0], color[1]), color[2]), 1 - max(max(color[0], color[1]), color[2])]`.
- To convert an HWB color `hwb` to RGB, generate `HsvToRgb([hwb[0], 1 - hwb[1]/(1-hwb[2]), 1 - hwb[2]])` if `hwb[2] < 1`, or `[hwb[0], 0, 0]` otherwise.

  Note: The HWB color model is not perception-based, as the HWB article acknowledges[30].

7.4 Y′ C_BC_R and Other Video Color Formats

An RGB color can be transformed to a specialized form to improve image and video encoding.

Y′ C_BC_R[31] (also known as YCbCr, YCrCb, or Y′ CrCb) is a family of color formats designed for this purpose. A Y′ C_BC_R color consists of three components in the following order:

- Y′, or *luma*, expresses an approximate "brightness".[32]
- C_B, or *blue chroma*, is based on the difference between blue and luma.
- C_R, or *red chroma*, is, based on the difference between red and luma.

The following pseudocode is an approximate conversion between RGB and Y′ C_BC_R (an approximation because the factors in the pseudocode are rounded off to a limited number of decimal places). There are three variants shown here, namely—

- the Rec. 601 variant (for standard-definition digital video), as the `YCbCrToRgb601` and `RgbToYCbCr601` methods,

[28] Smith, A.R. and Lyons, E.R., 1996. HWB—A more intuitive hue-based color model. Journal of graphics tools, 1(1), pp. 3-17.

[29] Smith, A.R. and Lyons, E.R., 1996. HWB—A more intuitive hue-based color model. Journal of graphics tools, 1(1), pp. 3-17.

[30] Smith, A.R. and Lyons, E.R., 1996. HWB—A more intuitive hue-based color model. Journal of graphics tools, 1(1), pp. 3-17.

[31] https://en.wikipedia.org/wiki/YCbCr

[32] The prime symbol appears near Y because the conversion from RGB usually involves **encoded RGB colors**, so that Y′ (*luma*) is not the same as luminance (Y). (See C. Poynton, **"*YUV* and *luminance* considered harmful"**.) However, that symbol is left out in function names and other names in the pseudocode for convenience only.

- the Rec. 709 variant (for high-definition video), as the `YCbCrToRgb709` and `RgbToYCbCr709` methods, and
- the **JPEG File Interchange Format**[33] variant, as the `YCbCrToRgbJpeg` and `RgbToYCbCrJpeg` methods.

The $Y'C_BC_R$ transformation is independent of RGB color space, but the three variants given earlier should use ***encoded RGB* colors** rather than *linear RGB* colors.

```
// NOTE: Derived from scaled YPbPr using red/green/blue luminance factors
// in the NTSC color space
METHOD RgbToYCbCr601(rgb)
    y = (16.0/255.0+rgb[0]*0.25678824+rgb[1]*0.50412941+rgb[2]*0.097905882)
    cb = (128.0/255.0-rgb[0]*0.1482229-rgb[1]*0.29099279+rgb[2]*0.43921569)
    cr = (128.0/255.0+rgb[0]*0.43921569-rgb[1]*0.36778831-rgb[2]*0.071427373)
    return [y, cb, cr]
END METHOD

// NOTE: Derived from scaled YPbPr using red/green/blue Rec. 709 luminance factors
METHOD RgbToYCbCr709(rgb)
    y = (0.06200706*rgb[2] + 0.6142306*rgb[1] + 0.1825859*rgb[0] + 16.0/255.0)
    cb = (0.4392157*rgb[2] - 0.338572*rgb[1] - 0.1006437*rgb[0] + 128.0/255.0)
    cr = (-0.04027352*rgb[2] - 0.3989422*rgb[1] + 0.4392157*rgb[0] + 128.0/255.0)
    return [y, cb, cr]
END METHOD

// NOTE: Derived from unscaled YPbPr using red/green/blue luminance factors
// in the NTSC color space
METHOD RgbToYCbCrJpeg(rgb)
    y = (0.299*rgb[0] + 0.587*rgb[1] + 0.114*rgb[2])
    cb = (-0.1687359*rgb[0] - 0.3312641*rgb[1] + 0.5*rgb[2] + 128.0/255.0)
    cr = (0.5*rgb[0] - 0.4186876*rgb[1] - 0.08131241*rgb[2] + 128.0/255.0)
    return [y, cb, cr]
END METHOD

METHOD YCbCrToRgb601(yCbCr)
    cb = yCbCr[1] - 128/255.0
    cr = yCbCr[2] - 128/255.0
    yp = 1.1643836 * (yCbCr[0] - 16/255.0)
    r = yp + 1.5960268 * cr
    g = yp - 0.39176229 * cb - 0.81296765 * cr
    b = yp + 2.0172321 * cb
    return [min(max(r,0),1),min(max(g,0),1),min(max(b,0),1)]
END METHOD

METHOD YCbCrToRgb709(yCbCr)
    cb = yCbCr[1] - 128/255.0
    cr = yCbCr[2] - 128/255.0
    yp = 1.1643836 * (yCbCr[0] - 16/255.0)
    r = yp + 1.7927411 * cr
    g = yp - 0.21324861 * cb - 0.53290933 * cr
    b = yp + 2.1124018 * cb
    return [min(max(r,0),1),min(max(g,0),1),min(max(b,0),1)]
```

[33] https://www.w3.org/Graphics/JPEG/jfif3.pdf

```
END METHOD

METHOD YCbCrToRgbJpeg(yCbCr)
    cb = yCbCr[1] - 128/255.0
    cr = yCbCr[2] - 128/255.0
    yp = yCbCr[0]
    r = yp + 1.402 * cr
    g = yp - 0.34413629 * cb - 0.71413629 * cr
    b = yp + 1.772 * cb
    return [min(max(r,0),1),min(max(g,0),1),min(max(b,0),1)]
END METHOD
```

Notes:

1. This document does not seek to survey the various ways in which Y ′ C_BC_R and similar colors are built up into pixels in images and video. In general, such ways take into account the human eye's normally greater spatial sensitivity to luminance (Y, as approximated, for example, by Y ′ , luma) than chromatic sensitivity (for example, C_B, C_R).
2. Other video color formats include "BT.2020 constant luminance", in **Rec. 2020**[34], and IC_TC_P, mentioned in Rep. 2390-4 and detailed in a **Dolby white paper**[35].

8 Other Color Models

The following sections discuss other color models of practical interest.

8.1 CIE XYZ

The **CIE 1931 standard colorimetric system**[36] (called the *XYZ color model* in this document) describes a transformation of a spectral curve into a point in three-dimensional space, as further explained in "**Spectral Color Functions**". An XYZ color consists of three components, in the following order:

- X is a component without special meaning.
- Y is related to the color's ***luminance***.
- Z is a component without special meaning.

Conventions for XYZ colors include the following:

- **Absolute XYZ.** In this convention, the Y component represents an absolute *luminance* in candelas per square meter (cd/m^2).
- **Relative XYZ.** In this convention, the three components are divided by the luminance of a given white point. In this case, the Y component represents a *luminance factor*; the white point has a luminance factor of 1.[37] (In sRGB, the white point's luminance is 80 cd/m^2.)

The conversion between RGB and XYZ varies by **RGB color space**. For example, the pseudocode below shows two methods that convert a color between **encoded sRGB** (`rgb`) and relative XYZ:

- For `XYZFromsRGB(rgb)` and `XYZTosRGB(xyz)`, the white point is the D65/2 white point.
- For `XYZFromsRGBD50(rgb)` and `XYZTosRGBD50(xyz)`, the white point is the D50/2 white point[38].

[34] https://en.wikipedia.org/wiki/Rec._2020

[35] https://www.dolby.com/us/en/technologies/dolby-vision/ICtCp-white-paper.pdf

[36] https://en.wikipedia.org/wiki/CIE_1931_color_space

[37] In interior and architectural design, the luminance factor multiplied by 100 is also known as *light reflectance value* (LRV).

[38] Although the D65/2 white point is the usual one for sRGB, another white point may be more convenient in the following cases, among others: - Using the white point `[0.9642, 1, 0.8249]` can improve interoperability with applications color-managed with International Color Consortium (ICC) version 2 or 4 profiles (this corresponds to the D50/2 white point given in CIE Publication 15 **before it was corrected**). - The printing industry uses the D50 illuminant for historical reasons (see A.

Both methods are approximate conversions because the factors in the pseudocode are rounded off to a limited number of decimal places.

```
// Applies a 3 &amp;times; 3 matrix transformation
METHOD Apply3x3Matrix(xyz, xyzmatrix)
    r=xyz[0]*xyzmatrix[0]+xyz[1]*xyzmatrix[1]+xyz[2]*xyzmatrix[2]
    g=xyz[0]*xyzmatrix[3]+xyz[1]*xyzmatrix[4]+xyz[2]*xyzmatrix[5]
    b=xyz[0]*xyzmatrix[6]+xyz[1]*xyzmatrix[7]+xyz[2]*xyzmatrix[8]
    return [r,g,b]
END METHOD

METHOD XYZFromsRGBD50(rgb)
    lin=SRGBToLinear3(rgb)
    // D65/2 sRGB matrix adapted to D50/2
    return Apply3x3Matrix(lin, [
       0.436027535573195, 0.385097932872408, 0.143074531554397,
       0.222478677613186, 0.716902127457834, 0.0606191949289806,
       0.0139242392790820, 0.0970836931437703, 0.714092067577148])
END METHOD

METHOD XYZTosRGBD50(xyz)
    // D65/2 sRGB matrix adapted to D50/2
    rgb=Apply3x3Matrix(xyz, [
       3.13424933163426, -1.61717292521282, -0.490692377104512,
       -0.978746070339639, 1.91611436125945, 0.0334415219513205,
       0.0719490494816283, -0.228969853236611, 1.40540126012171])
    return SRGBFromLinear3(rgb)
END METHOD

METHOD XYZFromsRGB(rgb)
    lin=SRGBToLinear3(rgb)
    // D65/2 sRGB matrix
    return Apply3x3Matrix(lin, [
      0.4123907992659591, 0.35758433938387796, 0.18048078840183424
       0.21263900587151016, 0.7151686787677559, 0.0721923153607337
       0.01933081871559181, 0.11919477979462596, 0.9505321522496605])
END METHOD

METHOD XYZTosRGB(xyz)
    // D65/2 sRGB matrix
    rgb=Apply3x3Matrix(xyz, [
       3.2409699419045235, -1.5373831775700944, -0.49861076029300355,
        -0.9692436362808797, 1.8759675015077204, 0.0415550574071756,
        0.05563007969699365, -0.20397695888897652, 1.0569715142428786])
    return SRGBFromLinear3(rgb)
END METHOD
```

Notes:

Kraushaar, **"Why the printing industry is not using D65?"**, 2009). https://lists.w3.org/Archives/Public/public-colorweb/2018Apr/0003.html https://fogra.org/plugin.php?menuid=125&template=mv/templates/mv_show_front.html&mv_id=10&extern_meta=x&mv_content_id=140332&getlang=en

1. In the pseudocode just given, 3 × 3 matrices are used to transform a linear RGB color to or from XYZ form (see **"Conversion Matrices Between XYZ and RGB"**).
2. `XYZTosRGB` and `XYZTosRGBD50` can return sRGB colors with components less than 0 or greater than 1, to make out-of-range XYZ colors easier to identify. If that is not desired, then the sRGB color can be converted to an in-range one. There are many such *gamut mapping* conversions; for example, one such conversion involves clamping each component of the sRGB color using the idiom `min(max(compo,0), 1)`, where `compo` is that component.
3. XYZ colors that have undergone **black point compensation** (see also ISO 18619) can be expressed as `Lerp3(wpoint, xyz, (1.0 - blackDest) / (1.0 - blackSrc))`, where—
   - `wpoint` is the white point as an absolute or relative XYZ color,
   - `xyz` is a relative XYZ color (relative to `wpoint`), and
   - `blackSrc` and `blackDest` are the luminance factors of the source and destination black points.

8.1.1 Encoding XYZ Through RGB

The following summarizes the transformations needed to convert a color from (relative) XYZ through RGB to an encoding form suitable for images or video.

1. An XYZ-to-linear-RGB transform. This is usually a **matrix** generated using the **RGB color space**'s red, green, blue, and white points, but can also include a ***chromatic adaptation transform***[39] if the XYZ and RGB color spaces use different white points (see the `XYZFromsRGBD50` and `XYZTosRGBD50` methods above)[40].
2. A linear-to-encoded-RGB transform. This is the RGB color space's "transfer function". This can be left out if linear RGB colors are desired.
3. A pixel encoding transform. This transforms the RGB color into $\mathbf{Y}'\mathbf{C}_B\mathbf{C}_R$ or another form. This can be left out.
4. The final color form is serialized into a binary, text, or other representation (see also "**Representing RGB Colors**").

The corresponding conversions to XYZ are then the inverse of the conversions just given.

8.1.2 Conversion Matrices Between XYZ and RGB

The following methods calculate a 3 × 3 matrix to convert from a linear RGB color to XYZ form (`RGBToXYZMatrix`) and back (`XYZToRGBMatrix`), given the RGB color space's red, green, blue, and white points. Each point is expressed as a relative XYZ color with arbitrary X and Z components and a Y component of 1. For example, `xr` and `zr` are the red point's X and Z components, respectively. See **brucelindbloom.com** for more information.

```
METHOD RGBToXYZMatrix(xr,zr,xg,zg,xb,zb,xw,zw)
 s1=(xb*zg - xb*zw - xg*zb + xg*zw + xw*zb - xw*zg)
 s2=(xb*zg - xb*zr - xg*zb + xg*zr + xr*zb - xr*zg)
 s3=(-xb*zr + xb*zw + xr*zb - xr*zw - xw*zb + xw*zr)
 sz=(-xr*(zg - zr) + xw*(zg - zr) + zr*(xg - xr) -
    zw*(xg - xr)) /
    ((xb - xr)*(zg - zr) - (xg - xr)*(zb - zr))
 sx=s1/s2
 sy=s3/s2
```

[39] https://en.wikipedia.org/wiki/Chromatic_adaptation

[40] Chromatic adaptation transforms include linear Bradford transformations, but are not further detailed in this document. (See also E. Stone, "**The Luminance of an sRGB Color**", 2013.) https://ninedegreesbelow.com/photography/srgb-luminance.html

```
 return [xr*sx,xg*sy,xb*sz,sx,sy,sz,zr*sx,zg*sy,zb*sz]
END METHOD

METHOD XYZToRGBMatrix(xr,zr,xg,zg,xb,zb,xw,zw)
 // NOTE: Inverse of RGBToXYZMatrix
 d1=(xb*zg - xb*zw - xg*zb + xg*zw + xw*zb - xw*zg)
 d2=(xb*zr - xb*zw - xr*zb + xr*zw + xw*zb - xw*zr)
 d3=(xg*zr - xg*zw - xr*zg + xr*zw + xw*zg - xw*zr)
 return [(zb - zg)/d1,(xb*zg - xg*zb)/d1,
  (-xb + xg)/d1, (zb - zr)/d2,
  (xb*zr - xr*zb)/d2,(-xb + xr)/d2,
  (zg - zr)/d3,(xg*zr - xr*zg)/d3,
  (-xg + xr)/d3]
END METHOD
```

8.1.3 Chromaticity Coordinates

The chromaticity coordinates x, y, and z are each the ratios of the corresponding component of an XYZ color to the sum of those components; therefore, those three coordinates sum to 1.[41] "xyY" form consists of x then y then the Y component of an XYZ color. "Yxy" form consists of the Y component then x then y of an XYZ color.

The CIE 1976 uniform chromaticity scale diagram is drawn using coordinates u' and v'.[42] "u ′ v ′ Y" form consists of u' then v' then the Y component of an XYZ color. "Yu ′ v ′" form consists of the Y component then u' then v' of an XYZ color.

In the following pseudocode, `XYZToxyY` and `XYZFromxyY` convert XYZ colors to and from their "xyY" form, respectively, and `XYZTouvY` and `XYZFromuvY` convert XYZ colors to and from their "u ′ v ′ Y" form, respectively.

```
    METHOD XYZToxyY(xyz)
            sum=xyz[0]+xyz[1]+xyz[2]
            if sum==0: return [0,0,0]
            return [xyz[0]/sum, xyz[1]/sum, xyz[1]]
    END METHOD

    METHOD XYZFromxyY(xyy)
            // NOTE: Results undefined if xyy[1]==0
            return [xyy[0]*xyy[2]/xyy[1], xyy[2], xyy[2]*(1 - xyy[0] - xyy[1])/xyy[1]]
    END METHOD

    METHOD XYZTouvY(xyz)
            sum=xyz[0]+xyz[1]*15.0+xyz[2]*3.0
            if sum==0: return [0,0,0]
            return [4.0*xyz[0]/sum,9.0*xyz[1]/sum,xyz[1]]
    END METHOD

    METHOD XYZFromuvY(uvy)
            // NOTE: Results undefined if uvy[1]==0
```

[41] Chromaticity coordinates can be defined for any three-dimensional Cartesian color space, not just XYZ (for example, (r, g, b) chromaticity coordinates for RGB). Such coordinates are calculated analogously to (x, y, z) coordinates.

[42] **CIE Technical Note 001:2014** says the chromaticity difference ($\Delta u' v'$) should be calculated as the **Euclidean distance** between two $u' v'$ pairs and that a chromaticity difference of 0.0013 is just noticeable "at 50% probability". (u, v) coordinates, a former 1960 version of u' and v', are found by taking u as u' and v as (v' * 2.0 / 3).

```
        su=uvy[2]/(uvy[1]/9.0)
        x=u*su/4.0
        z=(su/3.0)-(x/3.0)-5.0*uvy[2]
        return [x,uvy[2],z]
END METHOD
```

8.2 CIELAB

CIELAB[43] (also known as CIE $L^*a^*b^*$ or CIE 1976 $L^*a^*b^*$) is a three-dimensional color model designed for color comparisons.[44] In general, CIELAB color spaces differ in their white points.

A color in CIELAB consists of three components, in the following order:

- L^*, or *lightness* of a color (how bright that color appears in comparison to white), is 0 or greater and 100 or less, where 0 is black and 100 is white.
- a^* is a coordinate of the red/green axis (positive points to red, negative to green).
- b^* is a coordinate of the yellow/blue axis (positive points to yellow, negative to blue).[45]

L^*C^*h form expresses CIELAB colors as cylindrical coordinates; the three components have the following order:

- Lightness (L^*) remains unchanged.
- *Chroma* (C^*) is the distance of the color from the "gray" line.[46]
- *Hue* (h, an angle)[47] ranges from magenta at roughly 0 to red to yellow to green to cyan to blue to magenta.

In the following pseudocode:

- The following methods convert an **encoded sRGB** color to and from CIELAB:
  - `SRGBToLab` and `SRGBFromLab` treat white as the D65/2 white point.
  - `SRGBToLabD50` and `SRGBFromLabD50` treat white as the D50/2 white point.[48]

  Both methods are approximate conversions because the values in the pseudocode are rounded off to a limited number of decimal places.
- `XYZToLab(xyz, wpoint)` and `LabToXYZ(lab, wpoint)` convert an XYZ color to or from CIELAB, respectively, treating `wpoint` (an XYZ color) as the white point.
- `LabToChroma(lab)` and `LabToHue(lab)` find a CIELAB color's *chroma* or *hue*, respectively.

[43] https://en.wikipedia.org/wiki/Lab_color_space

[44] Although the CIELAB color model is also often called "perceptually uniform"— - CIELAB "was not designed to have the perceptual qualities needed for gamut mapping", according to **B. Lindbloom**, and - such a claim "is really only the case for very low spatial frequencies", according to P. Kovesi (P. Kovesi, "**Good Colour Maps: How to Design Them**", arXiv:1509.03700 [cs.GR], 2015). https://arxiv.org/abs/1509.03700

[45] The placement of the L^*, a^*, and b^* axes is related to the light–dark signal and the two opponent signals red/green and blue/yellow. See also endnote 6.

[46] The terms *lightness* and *chroma* are relative to an area appearing white. The corresponding terms *brightness* and *saturation*, respectively, are subjective terms: *brightness* is the perceived degree of reflected or emitted light, and *saturation* is the perceived hue strength (*colorfulness*) of an area in proportion to its brightness. (See also the CIE's International Lighting Vocabulary.) CIELAB has no formal saturation formula, however (see the Wikipedia article on **colorfulness**). https://en.wikipedia.org/wiki/Colorfulness

[47] The hue angle is in radians, and the angle is 0 or greater and less than 2 π . Radians can be converted to degrees by multiplying by `180 / pi`. Degrees can be converted to radians by multiplying by `pi / 180`.

[48] Although the D65/2 white point is the usual one for sRGB, another white point may be more convenient in the following cases, among others: - Using the white point `[0.9642, 1, 0.8249]` can improve interoperability with applications color-managed with International Color Consortium (ICC) version 2 or 4 profiles (this corresponds to the D50/2 white point given in CIE Publication 15 **before it was corrected**). - The printing industry uses the D50 illuminant for historical reasons (see A. Kraushaar, **"Why the printing industry is not using D65?"**, 2009). https://lists.w3.org/Archives/Public/public-colorweb/2018Apr/0003.html https://fogra.org/plugin.php?menuid=125&template=mv/templates/mv_show_front.html&mv_id=10&extern_meta=x&mv_content_id=140332&getlang=en

- `LchToLab(lch)` finds a CIELAB color given a 3-item list of lightness, chroma, and hue (L^*C^*h), in that order.
- `LabHueDifference(lab1, lab2)` finds the *metric hue difference* (ΔH^*) between two CIELAB colors. The return value can be positive or negative, but in some cases, the absolute value of that return value can be important.
- `LabChromaHueDifference(lab1, lab2)` finds the *chromaticness difference* (ΔCh) between two CIELAB colors, as given, for example, in ISO 13655.

```
METHOD XYZToLab(xyzval, wpoint)
    xyz=[xyzval[0]/wpoint[0],xyzval[1]/wpoint[1],xyzval[2]/wpoint[2]]
    i=0
    while i < 3
       if xyz[i] > 216.0 / 24389 // See BruceLindbloom.com
          xyz[i]=pow(xyz[i], 1.0/3.0)
       else
           kappa=24389.0/27 // See BruceLindbloom.com
           xyz[i]=(16.0 + kappa*xyz[i]) / 116
       end
       i=i+1
    end
    return [116.0*xyz[1] - 16,
        500 * (xyz[0] - xyz[1]),
        200 * (xyz[1] - xyz[2])]
END METHOD

METHOD LabToXYZ(lab,wpoint)
    fy=(lab[0]+16)/116.0
    fx=fy+lab[1]/500.0
    fz=fy-lab[2]/200.0
    fxcb=fx*fx*fx
    fzcb=fz*fz*fz
    xyz=[fxcb, 0, fzcb]
    eps=216.0/24389 // See BruceLindbloom.com
    if fxcb <= eps: xyz[0]=(108.0*fx/841)-432.0/24389
    if fzcb <= eps: xyz[2]=(108.0*fz/841)-432.0/24389
    if lab[0] > 8 // See BruceLindbloom.com
            xyz[1]=pow(((lab[0]+16)/116.0), 3.0)
    else
            xyz[1]=lab[0]*27.0/24389 // See BruceLindbloom.com
    end
    xyz[0]=xyz[0]*wpoint[0]
    xyz[1]=xyz[1]*wpoint[1]
    xyz[2]=xyz[2]*wpoint[2]
    return xyz
END METHOD

METHOD SRGBToLab(rgb)
    return XYZToLab(XYZFromsRGB(rgb),
      [0.9504559270516716, 1, 1.0890577507598784])
END METHOD
```

```
METHOD SRGBFromLab(lab)
    return XYZTosRGB(LabToXYZ(lab,
      [0.9504559270516716, 1, 1.0890577507598784]))
END METHOD

METHOD SRGBToLabD50(rgb)
    return XYZToLab(XYZFromsRGBD50(rgb), [0.9642, 1, 0.8251])
END METHOD

METHOD SRGBFromLabD50(lab)
    return XYZTosRGBD50(LabToXYZ(lab, [0.9642, 1, 0.8251]))
END METHOD

   // -- Derived values from CIELAB colors

METHOD LabToChroma(lab)
        return sqrt(lab[1]*lab[1] + lab[2]*lab[2])
END METHOD

    METHOD LabToHue(lab)
            h = atan2(lab[2], lab[1])
            if h < 0: h = h + pi * 2
            return h
    END METHOD

    METHOD LchToLab(lch)
            return [lch[0], lch[1] * cos(lch[2]), lch[1] * sin(lch[2])]
    END METHOD

METHOD LabHueDifference(lab1, lab2)
  cmul=LabToChroma(lab1)*LabToChroma(lab2)
  h2=LabToHue(lab2)
  h1=LabToHue(lab1)
  hdiff=h2-h1
  if abs(hdiff)>pi
        if h2<=h1: hdiff=hdiff+math.pi*2
        else: hdiff=hdiff-math.pi*2
  end
  return sqrt(cmul)*sin(hdiff*0.5)*2
END METHOD

METHOD LabChromaHueDifference(lab1, lab2)
            da=lab1[1]-lab2[1]
            db=lab1[2]-lab2[2]
            return sqrt(da*da+db*db)
    END METHOD
```

Note: The difference in lightness, a^*, b^*, or chroma (ΔL^*, Δa^*, Δb^*, or ΔC^*, respectively) between two CIELAB colors is simply the difference between the corresponding value of the second CIELAB color and that of the first.

8.3 CIELUV

CIELUV (also known as CIE *L*u*v** or CIE 1976 *L*u*v**) is a second color model designed for color comparisons. A CIELUV color has three components, namely, *L**, or *lightness* (which is the same as in CIELAB), *u**, and *v**, in that order. As **B. MacEvoy explains**, "CIELUV represents the additive mixture of two lights as a straight line", so that this color model is especially useful for light sources.

In the following pseudocode—

- the `SRGBToLuv`, `SRGBFromLuv`, `SRGBToLuvD50`, `SRGBFromLuvD50`, `XYZToLuv`, and `LuvToXYZ` methods perform conversions involving CIELUV colors analogously to the similarly named methods for **CIELAB**, and
- the `LuvToSaturation` method finds the ***saturation***[49] (*s*uv) of a CIELUV color.

`SRGBToLuv` and `SRGBFromLuv` are approximate conversions because the values in the pseudocode are rounded off to a limited number of decimal places.

```
METHOD XYZToLuv(xyz, wpoint)
    lab=XYZToLab(xyz, wpoint)
    sum=xyz[0]+xyz[1]*15+xyz[2]*3
    lt=lab[0]
    if sum==0: return [lt, 0, 0]
    upr=4*xyz[0]/sum // U-prime
    vpr=9*xyz[1]/sum // V-prime
    sumwhite=wpoint[0]+15*wpoint[1]+wpoint[2]*3
    return [lt,
            lt*13*(upr - 4*wpoint[0]/sumwhite),
            lt*13*(vpr - 9.0*wpoint[1]/sumwhite)]
END METHOD

METHOD LuvToXYZ(luv, wpoint)
    if luv[0]==0: return [0, 0, 0]
    xyz=LabToXYZ([luv[0], 1, 1],wpoint)
    sumwhite=wpoint[0]+15*wpoint[1]+wpoint[2]*3
    u0=4*wpoint[0]/sumwhite
    v0=9.0*wpoint[1]/sumwhite
    lt=luv[0]
    a=(52*lt/(luv[1]+13*u0*lt)-1)/3.0
    d=xyz[1]*(39*lt/(luv[2]+13*v0*lt)-5)
    x=(d+5*xyz[1])/(a+1.0/3)
    z=x*a-5*xyz[1]
    return [x,xyz[1],z]
END METHOD

METHOD SRGBToLuv(rgb)
    return XYZToLuv(XYZFromsRGB(rgb),
      [0.9504559270516716, 1, 1.0890577507598784])
END METHOD

METHOD SRGBFromLuv(lab)
    return XYZTosRGB(LuvToXYZ(lab,
```

[49] https://en.wikipedia.org/wiki/Colorfulness

```
      [0.9504559270516716, 1, 1.0890577507598784]))
END METHOD

METHOD SRGBToLuvD50(rgb)
    return XYZToLuv(XYZFromsRGBD50(rgb), [0.9642, 1, 0.8251])
END METHOD

METHOD SRGBFromuvD50(lab)
    return XYZTosRGBD50(LuvToXYZ(lab, [0.9642, 1, 0.8251]))
END METHOD

METHOD LuvToSaturation(luv)
    if luv[0]==0: return 0
    return sqrt(luv[1]*luv[1]+luv[2]*luv[2])/luv[0]
END METHOD
```

Notes:

- Hue and chroma can be derived from a CIELUV color in a similar way as from a CIELAB color, with u^* and v^* used instead of a^* and b^*, respectively. The `LabToHue`, `LabToChroma`, `LabHueDifference`, `LabChromaHueDifference`, and `LchToLab` methods from the previous section work with CIELUV colors analogously to CIELAB colors.
- The difference in lightness, u^*, v^*, chroma, or saturation (ΔL^*, Δu^*, Δv^*, ΔC^*uv, or Δsuv, respectively) between two CIELUV colors is simply the difference between the corresponding value of the second CIELUV color and that of the first.

8.4 CMYK and Other Ink-Mixture Color Models

The *CMYK color model*, ideally, describes the proportion of cyan, magenta, yellow, and black (K) inks to use to reproduce certain colors on a surface.[50] However, since **color mixture** of inks or other colorants is very complex, the exact color appearance of any recipe of colorants (not just in the CMYK context) depends on the *printing condition* (as defined in ISO 12647-1), including what colorants are used, how the colorants are printed, and what surface (for example, paper) the printed output appears on.

Characterization tables. In printing industry practice, a given printing condition is characterized by finding out how it forms colors using different mixtures of inks. This is usually done by printing CMYK color "patches" and using a **color measurement device**[51] to measure their **CIELAB** colors under standardized lighting and measurement conditions.

The International Color Consortium maintains a **list of standardized conversions** of CMYK color "patches", usually to CIELAB colors, for different standardized printing conditions. Such conversions are generally known as *characterization data* or *characterization tables*.

Given a CMYK-to-CIELAB characterization table, a CMYK color can be converted to and from a CIELAB color by multidimensional interpolation of the table's "patches".[52]

Rough conversions. The following pseudocode shows *very rough* conversions between an RGB color (`color`) and a CMYK color (`cmyk`):

```
    // RGB to CMYK
```

[50] This section concerns mostly CMYK because printing systems that involve inks other than cyan, magenta, yellow, and black (notably "extended gamut" systems of five or more inks, and systems that use custom "spot" color inks) are not yet of general interest to programmers.

[51] https://peteroupc.github.io/suppcolor.html#Color_Measurement_Devices

[52] This page does not detail how multidimensional interpolation works, but an example is SciPy's `griddata` method. https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.griddata.html

```
k = min(min(1.0 - color[0], 1.0 - color[1]), 1.0 - color[2])
cmyk=[0, 0, 0, 1]
if k!=1:
   cmyk=[((1.0 - color[0]) - k) / (1 - k), ((1.0 - color[1]) - k) /
      (1 - k), ((1.0 - color[2]) - k) / (1 - k), k]
end
// CMYK to RGB
ik = 1 - cmyk[3]
color=[(1 - cmyk[0]) * ik, (1 - cmyk[1]) * ik, (1 - cmyk[2]) * ik]
```

9 Color Operations

This section goes over many of the operations that can be done on colors. Note that for best results, these operations need to be carried out with ***linear RGB* colors** rather than *encoded RGB* colors, unless noted otherwise.

9.1 Luminance Factor (Grayscale)

The ***luminance factor***—

- is a single number indicating a color's luminance relative to "white", that is, how much light reaches the eyes when that color is viewed, in comparison to "white",
- is called `Luminance(color)` in this document,
- is equivalent to the Y component of a relative **XYZ color**, and
- ranges from 0 for "black" to 1 for "white".

Finding a color's luminance factor depends on that color's color space.

A ***linear RGB* color**'s luminance factor is `(color[0] * r + color[1] * g + color[2] * b)`, where `r`, `g`, and `b` are the luminance factors (relative Y components) of the RGB color space's red, green, and blue points, respectively. (If a different white point than the RGB color space's usual white point should have a luminance factor of 1, then `r`, `g`, and `b` are the corresponding values after a ***chromatic adaptation transform***[53] from one white point to another.[54])

An ***encoded RGB* color** needs to be converted to linear RGB (in the same RGB color space) before finding its luminance factor. For example, the pseudocode below implements `Luminance(color)` for encoded sRGB colors (`LuminanceSRGB` and `LuminanceSRGBD50`)[55]. Both methods are approximate conversions because the factors in the pseudocode are rounded off to a limited number of decimal places.

```
// Convert encoded sRGB to luminance factor
METHOD LuminanceSRGB(color)
    // Convert to linear sRGB
    c = SRGBToLinear(color)
    // Find the linear sRGB luminance factor
    return c[0] * 0.2126 + c[1] * 0.7152 + c[2] * 0.0722
```

[53] https://en.wikipedia.org/wiki/Chromatic_adaptation

[54] Chromatic adaptation transforms include linear Bradford transformations, but are not further detailed in this document. (See also E. Stone, "**The Luminance of an sRGB Color**", 2013.) https://ninedegreesbelow.com/photography/srgb-luminance.html

[55] Although the D65/2 white point is the usual one for sRGB, another white point may be more convenient in the following cases, among others: - Using the white point `[0.9642, 1, 0.8249]` can improve interoperability with applications color-managed with International Color Consortium (ICC) version 2 or 4 profiles (this corresponds to the D50/2 white point given in CIE Publication 15 **before it was corrected**). - The printing industry uses the D50 illuminant for historical reasons (see A. Kraushaar, **"Why the printing industry is not using D65?"**, 2009). https://lists.w3.org/Archives/Public/public-colorweb/2018Apr/0003.html https://fogra.org/plugin.php?menuid=125&template=mv/templates/mv_show_front.html&mv_id=10&extern_meta=x&mv_content_id=140332&getlang=en

```
END METHOD

// Convert encoded sRGB (with D50/2 white point)
// to luminance factor
METHOD LuminanceSRGBD50(color)
    c = SRGBToLinear(color)
    return c[0] * 0.2225 + c[1] * 0.7169 + c[2] * 0.0606
END METHOD
```

Examples:

1. **Grayscale.** A color, `color`, can be converted to grayscale by calculating `[Luminance(color), Luminance(color), Luminance(color)]`.
2. An ***image color list***'s **average luminance factor** is often equivalent to the average `Luminance(color)` value among the colors in that image color list.
3. An application can consider a color **dark** if `Luminance(color)` is lower than some threshold, say, 15.
4. An application can consider a color **light** if `Luminance(color)` is greater than some threshold, say, 70.

Note: `Luminance(color)` belongs to a family of functions that give out a single number that summarizes a color and ranges from 0 for "minimum intensity" through 1 for "maximum intensity". The following are other functions in this family.

1. **Single channel** of a multicomponent color; for example, `color[0]`, `color[1]`, or `color[2]` for an RGB color's red, green, or blue component, respectively. Examples of a color channel are a red component, a luminance factor, or a point on a spectral reflectance curve.
2. **Average** of the multicomponent color's components (see **Alpha Blending**).
3. **Maximum**; for example, `max(max(color[0], color[1]), color[2])` for three-component colors.
4. **Minimum**; for example, `min(min(color[0], color[1]), color[2])` for three-component colors. (For techniques 1-4, see also (Helland)[56].)
5. **Light/dark factor**: A **CIELAB** or **CIELUV** color's lightness (L^*) divided by 100 (or a similar ratio in other color spaces with a light/dark dimension, such as **HSL** "lightness" (Cook 2009)[57]).

9.2 Alpha Blending

An *alpha blend* is a linear interpolation of two multicomponent colors (such as two RGB colors) that works component-by-component. For example, the `Lerp3` function below[58] does an alpha blend of two three-component colors, where—

- `color1` and `color2` are the two colors, and
- `alpha`, the *alpha component*, is usually 0 or greater and 1 or less (from `color1` to `color2`), but need not be (Haeberli and Voorhees)[59].

[56] T. Helland, **"Seven grayscale conversion algorithms (with pseudocode and VB6 source code)"**.

[57] J. Cook, **"Converting color to grayscale"**, Aug. 24, 2009. https://www.johndcook.com/blog/2009/08/24/algorithms-convert-color-grayscale/

[58] `Lerp3` is equivalent to `mix` in OpenGL Shading Language (GLSL). Making `alpha` the output of a function (for example, `Lerp3(color1, color2, FUNC(...))`, where `FUNC` is an arbitrary function of one or more variables) can be done to achieve special nonlinear blends. Such blends (interpolations) are described in further detail **in another page**. https://peteroupc.github.io/html3dutil/MathUtil.html#MathUtil.vec3lerp

[59] P. Haeberli and D. Voorhees, "**Image Processing by Interpolation and Extrapolation**".

```
METHOD Lerp3(color1, color2, alpha)
    return [color1[0]+(color2[0]-color1[0])*alpha, color1[1]+(color2[1]-color1[1])*alpha,
        color1[2]+(color2[2]-color1[2])*alpha]
END METHOD
```

Alpha blends can support the following color operations.

- **Shade.** Generating a shade of a color (mixing with black) can be done by alpha blending that color with black (such as `[0, 0, 0]` in RGB).
- **Tint.** Generating a tint of a color (mixing with white) can be done by alpha blending that color with white (such as `[1, 1, 1]` in RGB).
- **Tone.** Generating a tone of a color (mixing with gray) can be done by alpha blending that color with gray (such as `[0.5, 0.5, 0.5]` in RGB).
- **Averaging.** Averaging two colors results by alpha blending with `alpha` set to 0.5.
- **Colorize.** `color1` is black, `color2` is the destination color, and `alpha` is a single number that summarizes the source color and ranges from 0 for "minimum intensity" through 1 for "maximum intensity". RGB example: `Lerp3([0, 0, 0], destinationColor, Luminance(srcColor))`, where `Luminance` is as described in **"Luminance Factor (Grayscale)"**. The destination color is usually the same for each pixel in an image.
- Converting an RGBA color to an RGB color on white can be done as follows: `Lerp3([color[0], color[1], color[2]], [1, 1, 1], color[3])`.
- Converting an RGBA color to an RGB color over `color2`, another RGB color, can be done as follows: `Lerp3([color[0], color[1], color[2]], color2, color[3])`.

9.3 Binarization

Binarization, also known as *thresholding*, involves classifying pixels or colors into one of two categories (usually black or white). It involves applying a function to a pixel or color and returning 1 if the result is greater than a threshold, or 0 otherwise. The following are examples of binarization with RGB colors in 0-1 format.

- **Black and white.** Generate `[1, 1, 1]` (white) if a *light–dark factor* (such as the color's **CIELAB** lightness, _L*_, divided by 100) is greater than 0.5, or `[0, 0, 0]` (black) otherwise.
- **Contrasting color.** Generate `[1, 1, 1]` (white) if a *light–dark factor* is less than 0.5, or `[0, 0, 0]` (black) otherwise.

Other forms of binarization may classify pixels based at least in part on their positions in the image.

9.4 Color Schemes and Harmonies

The following techniques generate new colors that are related to existing colors.

- **Color harmonies**[60] result by generating several colors that differ in hue (hue angle). For each color harmony given later, the following numbers are added to a hue angle[61] to generate the hues for the colors that make up that harmony:
  - **Analogous**: 0, Y, -Y, where Y is 2 π /3 or less. In general, *analogous colors* are two, four, or a higher even number of colors spaced at equal hue intervals from a central color.
  - **Complementary**: 0, π . This is the base hue with its opposite hue.
  - **Split complementary**: 0, π - Y, π + Y, where Y is greater than 0 and π /2 or less. The base hue and two hues close to the opposite hue.
  - **Triadic**: 0, 2 π /3, 4 π /3. Base hue and the two hues at 120 degrees from that hue.

[60] B. MacEvoy calls these ***hue harmonies***. See also his **summary of harmonious color relationships**.

[61] The hue angle is in radians, and the angle is 0 or greater and less than 2 π . Radians can be converted to degrees by multiplying by `180 / pi`. Degrees can be converted to radians by multiplying by `pi / 180`.

  - **Two-tone**: 0, Y, where Y is greater than - π /2 and less than π /2. This is the base hue and a close hue.
  - **Off-complementary**: 0, Y, where Y is - π /2 or less but greater than - π , or Y is π /2 or greater but less than π . B. MacEvoy mentions Y = 2 π /3.
  - **Double complementary**: 0, Y, π , π + Y, where Y is - π /2 or greater and π /2 or less. The base hue and a close hue, as well as their opposite hues.
  - **Tetradic**: Double complementary with Y = π /2.
  - **N-color**: 0, 2 π /N, 4 π /N, ..., (N - 1)2 π /N.
- **Monochrome colors**: Colors with the same hue; for example, different **shades, tints, or tones** of a given color are monochrome colors.
- **Achromatic colors**: Colors without hue; that is, black, white, and shades of gray.

9.5 Contrast Between Two Colors

There are several ways to find the contrast between two colors.

Luminance Contrast. Luminance contrast formulas quantify how differently a foreground (text) color appears over a background color or vice versa, in terms of the luminance of both colors. In general, the greater the difference, the higher the contrast.

> **Example:** The **Web Content Accessibility Guidelines 2.0 (WCAG)**[62] includes a contrast ratio formula implemented in the pseudocode below, where `RelLum(color)`— - is the "relative luminance" of a color as defined in the WCAG, and - is equivalent to **`Luminance(color)`** whenever WCAG conformity is not important.

```
METHOD ContrastRatioWCAG(color1, color2)
    rl1=RelLum(color1)
    rl2=RelLum(color2)
    return (max(rl1,rl2)+0.05)/(min(rl1,rl2)+0.05)
END METHOD
```

> For 8-bpc encoded sRGB colors, `RelLum(color)` is effectively equivalent to `LuminanceSRGB(color)`, but with the WCAG using a different version of `SRGBToLinear`, with 0.03928 (the value used in the sRGB proposal) rather than 0.04045, but this difference doesn't affect the result for such 8-bpc colors. In general, under the WCAG, a contrasting color is one whose contrast ratio with another color is 4.5 or greater (or 7 or greater for a stricter conformance level).

Opacity. In certain industries, a material's *contrast ratio* or *opacity* can be found by dividing the Y component of the material's **XYZ color** measured over a black surface by the Y component of the material's XYZ color measured over a white surface. Details of the measurement depend on the industry and material.

9.6 Porter–Duff Formulas

Porter and Duff (1984) define twelve formulas for combining (compositing) two RGBA colors[63]. In the formulas below, it is assumed that the two colors and the output are in the 0-1 format and have been *premultiplied* (that is, their red, green, and blue components have been multiplied beforehand by their alpha component). Given `src`, the source RGBA color, and `dst`, the destination RGBA color, the Porter–Duff formulas are as follows.

- **Source Over**: `[src[0]-dst[0]*(src[3] - 1), src[1]-dst[1]*(src[3] - 1), src[2]-dst[2]*(src[3] - 1), src[3]-dst[3]*(src[3] - 1)]`.

[62] https://www.w3.org/TR/2008/REC-WCAG20-20081211/#visual-audio-contrast-contrast

[63] Porter, T., and Duff. T. "Compositing Digital Images". Computer Graphics 18(3), p 253 ff., 1984.

- **Source In**: `[dst[3]*src[0], dst[3]*src[1], dst[3]*src[2], dst[3]*src[3]]`.
- **Source Held Out**: `[src[0]*(1 - dst[3]), src[1]*(1 - dst[3]), src[2]*(1 - dst[3]), src[3]*(1 - dst[3])]`.
- **Source Atop**: `[dst[3]*src[0] - dst[0]*(src[3] - 1), dst[3]*src[1] - dst[1]*(src[3] - 1), dst[3]*src[2] - dst[2]*(src[3] - 1), dst[3]]`.
- **Destination Over**: `[dst[0] - src[0]*(dst[3] - 1), dst[1] - src[1]*(dst[3] - 1), dst[2] - src[2]*(dst[3] - 1), dst[3] - src[3]*(dst[3] - 1)]`.
- **Destination In**: `[dst[0]*src[3], dst[1]*src[3], dst[2]*src[3], dst[3]*src[3]]`. Uses the destination color/alpha with the source alpha as the "mask".
- **Destination Held Out**: `[dst[0]*(1 - src[3]), dst[1]*(1 - src[3]), dst[2]*(1 - src[3]), dst[3]*(1 - src[3])]`.
- **Destination Atop**: `[dst[0]*src[3] - src[0]*(dst[3] - 1), dst[1]*src[3] - src[1]*(dst[3] - 1), dst[2]*src[3] - src[2]*(dst[3] - 1), src[3]]`.
- **Source**: `src`.
- **Destination**: `dst`.
- **Clear**: `[0, 0, 0, 0]`.
- **Xor**: `[-dst[3]*src[0] - dst[0]*src[3] + dst[0] + src[0], -dst[3]*src[1] - dst[1]*src[3] + dst[1] + src[1], -dst[3]*src[2] - dst[2]*src[3] + dst[2] + src[2], -2*dst[3]*src[3] + dst[3] + src[3]]`.

The same paper by Porter and Duff also mentioned a *plus operator*, which is a simple adding of the source and destination RGBA colors' components; however, the resulting color may have components greater than 1, which may lead to less than well-defined behavior.

9.7 Raster Operations

Raster operations define Boolean operations, or bit-by-bit combinations of an input or source color ("in") and an output or destination color ("out"). Unlike with most other color operations in this document, the input and output colors are nonnegative integers, rather than made of components with fractional numbers from 0 through 1, and, if the colors are RGB colors, they can be linear or encoded.

There are sixteen *binary raster operations*, each operation taking two bits (where each bit is either 0 or 1):

Code	Operation
0	0
1	NOT (in OR out)
2	out AND NOT in
3	NOT in
4	in AND NOT out
5	NOT out
6	in XOR out
7	NOT (in AND out)
8	in AND out
9	NOT (in XOR out)
10	out
11	NOT (in AND NOT out)
12	in
13	NOT (out AND NOT in)
14	in OR out
15	1

In the list of operations above:

- "NOT a" means 0 if a is 1, or 1 if a is 0.
- "a AND b" means 1 if a and b are both 1, or 0 otherwise.
- "a OR b" means 1 if a is 1 or b is 1 or both, or 0 otherwise.
- "a XOR b" means 1 if a does not equal b, or 0 otherwise.

The table below illustrates the result of some binary raster operations.

in	out	8: in AND out	6: in XOR out	3: NOT in
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	0	0

The result of a binary raster operation `rop` (where `rop` is one of the codes given in the table of binary raster operations), given bits `in` and `out`, equals `(rop>>((in * 2) + out)) AND 1`, where `>>` is a right-shift bit operation that involves the left-hand side `L` and the right-hand side `R` and is equivalent to `floor(L / pow(2, R))`.

There are also 256 *ternary raster operations*, involving bit-by-bit combinations of the input color ("in"), the output color ("out"), and a so-called *brush pattern* color ("pat"). Each operation has a code equal to `codeH * 16 + codeL`, and the corresponding operation has the form—

- (*opH* AND pat) XOR (*opL* AND NOT pat),

where *opH* is the binary raster operation for the input and output colors with the code *codeH*, and *opL*, with the code *codeL*. In other words, if the brush pattern bit is 1, use *opH*; if 0, use *opL*.

For example, code 28 is a ternary raster operation made up of binary raster operations *codeH* = 1, *opH* = NOT (in OR out), *codeL* = 12, and *opL* = in. (`1 * 16 + 12 = 28`.) This ternary operation can be expressed as `((NOT (in OR out)) AND pat) XOR (in AND NOT pat)`; that is, if the brush pattern bit is 1, use `NOT (in OR out)`; if 0, use `in`.

Binary and ternary raster operations are prevalent in bit block transfers, which copy or transfer parts of images onto other images.

> **Note:** Raster operations also function, in principle, when the input and output color values are interpreted as zero-based indices to a color palette (that is, color value 0 refers to the first entry in a palette of colors; color value 1, the second; and so on), rather than as intensities (such as RGB colors). But this is a more delicate case than the usual one, and functions best when—
>
> - the number of colors in the color palette is a power of two (for example, 2, 8, 16, 256), and
> - for each color index i, the color at index i is the same as (or at least "close" to) the "inversion" of the color at index $n - 1 - i$ (a less technical but less preferable alternative is: the colors in the palette are sorted by their intensity, so that, for example, the lowest-intensity color, the color closest to "black", appears first and the highest-intensity color, the color closest to "white", appears last).

9.8 Blend Modes

Blend modes[64] take two multicomponent colors, namely a source color and a destination color, and blend them to create a new color. The same blend mode, or different blend modes, can be applied to each component of a given color. In the idioms below, `src` is one component of the source color, `dst` is the same component of the destination color (for example, `src` and `dst` can both be two RGB colors' red components),

[64] https://en.wikipedia.org/wiki/Blend_modes

and both components are assumed to be 0 or greater and 1 or less. The following are examples of blend modes.

- **Normal**: `src`.
- **Lighten**: `max(src, dst)`.
- **Darken**: `min(src, dst)`.
- **Add**: `min(1.0, src + dst)`.
- **Subtract**: `max(0.0, src - dst)`.
- **Multiply**: `(src * dst)`.
- **Screen**: `1 - (1 - dst) * (1 - src)`.
- **Average**: `src + (dst - src) * 0.5`.
- **Difference**: `abs(src - dst)`.
- **Exclusion**: `src - 2 * src * dst + dst`.

9.9 Color Matrices

A *color matrix* is a 9-item (3 × 3) list for transforming a three-component color. The following are examples of color matrices:

- **Sepia.** Sepia matrices can have the form `[r*sw[0], g*sw[0], b*sw[0], r*sw[1], g*sw[1], b*sw[1], r*sw[2], g*sw[2], b*sw[2]]`, where `r`, `g`, and `b` are as defined in the section "**Luminance Factor (Grayscale)**", and `sw` is the RGB color for "sepia white" (an arbitrary choice). An example for linear sRGB is: `[0.207,0.696,0.07,0.212,0.712,0.072,0.16,0.538,0.054]`.
- **Saturate.** `[s+(1-s)*r, (1-s)*g, (1-s)*b, (1-s)*r, s+(1-s)*g,(1-s)*b,(1-s)*r,(1-s)*g,s+(1-s)*b]`, where `s` ranges from 0 through 1 (the greater `s` is, the less saturated), and `r`, `g`, and `b` are as defined in the section "**Luminance Factor (Grayscale)**"[65].
- **Hue rotate.** `[-0.37124*sr + 0.7874*cr + 0.2126, -0.49629*sr - 0.7152*cr + 0.7152, 0.86753*sr - 0.0722*cr + 0.0722, 0.20611*sr - 0.2126*cr + 0.2126, 0.08106*sr + 0.2848*cr + 0.7152, -0.28717*sr - 0.072199*cr + 0.0722, -0.94859*sr - 0.2126*cr + 0.2126, 0.65841*sr - 0.7152*cr + 0.7152, 0.29018*sr + 0.9278*cr + 0.0722]`, where `sr = sin(rotation)`, `cr = cos(rotation)`, and `rotation` is the hue rotation angle.[66][67] This is an approximate hue rotation because the constant factors in the pseudocode are rounded off to a limited number of decimal places.

In the following pseudocode, `TransformColor` transforms an RGB color (`color`) with a color matrix (`matrix`).

```
METHOD TransformColor(color, matrix)
   return [
      min(max(color[0]*matrix[0]+color[1]*matrix[1]+color[2]*matrix[2],0),1),
      min(max(color[0]*matrix[3]+color[1]*matrix[4]+color[2]*matrix[5],0),1),
      min(max(color[0]*matrix[6]+color[1]*matrix[7]+color[2]*matrix[8],0),1) ]
END METHOD
```

More generally—

[65] P. Haeberli, **"Matrix Operations for Image Processing"**, 1993. The hue rotation matrix given was generated using the technique in the section "Hue Rotation While Preserving Luminance", with constants rounded to five significant digits and with `rwgt=0.2126`, `gwgt=0.7152`, and `bwgt = 0.0722`, the sRGB luminance factors for the red, green, and blue points. For the saturation and hue rotation matrices, the sRGB luminance factors are used rather than the values recommended by the source.

[66] P. Haeberli, **"Matrix Operations for Image Processing"**, 1993. The hue rotation matrix given was generated using the technique in the section "Hue Rotation While Preserving Luminance", with constants rounded to five significant digits and with `rwgt=0.2126`, `gwgt=0.7152`, and `bwgt = 0.0722`, the sRGB luminance factors for the red, green, and blue points. For the saturation and hue rotation matrices, the sRGB luminance factors are used rather than the values recommended by the source.

[67] This is often called the "CMY" ("cyan–magenta–yellow") version of the RGB color (although the resulting color is not necessarily based on a proportion of cyan, magenta, and yellow inks; see also "**CMYK and Other Ink-Mixture Color Models**"). If such an operation is used, the conversions between "CMY" and RGB are exactly the same.

- an N×N matrix can be used to transform an N-component color, and
- an (N+1)×(N+1) matrix can be used to transform a color consisting of N components followed by the number 1; if this is done, the first N components of the transformed color are divided by its last component.

9.10 Lighten/Darken

The following approaches can generate a lighter or darker version of a color. In the examples, `color` is an RGB color in 0-1 format, and `value` is positive to lighten a color, or negative to darken a color, and -1 or greater and 1 or less.

- **RGB additive.** `[min(max(color[0]+value,0),1), min(max(color[1]+value,0),1), min(max(color[2]+value,0`
- **HSL "lightness" additive.** `HslToRgb(hsl[0], hsl[1], min(max(hsl[2] + value, 0), 1))`, where `hsl = RgbToHsl(color)`.
- **CIELAB lightness additive.** Adds a number to the L^* component of the color's CIELAB version. For example, given a CIELAB color `lab`, this is: `[min(max(lab[0] + (value * 100), 0), 100), lab[1], lab[2]]`.
- **Tints and shades.** A "tint" is a lighter version, and a "shade" is a darker version. See "**Alpha Blending**".

9.11 Saturate/Desaturate

The following approaches can generate a saturated or desaturated version of a color. In the examples, `color` is an RGB color in 0-1 format, and `value` is positive to saturate a color, or negative to desaturate a color, and -1 or greater and 1 or less.

- **HSV "saturation" additive.** `HsvToRgb(hsv[0], min(max(hsv[1] + color, 0), 1), hsv[2])`, where `hsv = RgbToHsv(color)`. (Note that HSL's "saturation" is inferior here.)
- **Tones, or mixtures of gray.** A "tone" is a desaturated version. A color can be desaturated by **alpha blending** that color with either its **grayscale** version or an arbitrary shade of gray.
- **Saturate matrix.** See "**Color Matrices**".

9.12 Miscellaneous

1. An RGB color—

   - is white, black, or a shade of gray (***achromatic***) if it has equal red, green, and blue components, and
   - is in the "safety palette" if its red, green, and blue components are each a multiple of 0.2.[68]

   An ***image color list*** is achromatic if all its colors are achromatic.

2. Background removal algorithms, including ***chroma key***[69], can replace "background" pixels of a raster image with other colors. Such algorithms are outside the scope of this document unless they use only a pixel's color to determine whether that pixel is a "background" pixel (for example, by checking whether

[68]The "safety palette", also known as the "Web safe" colors, consists of 216 colors that are uniformly spaced in the red–green–blue color cube. Robert Hess's article "**The Safety Palette**)", 1996, described the advantage that images that use only colors in this palette won't dither when displayed by Web browsers on displays that can show up to 256 colors at once. (See also **Wikipedia**. Dithering is the scattering of colors in a limited set to simulate colors outside that set.) The definition of the "safety palette", though not the name, dates as early as 1994, when Microsoft's WinG API provided a *halftone palette* to "allo[w] applications to simulate true 24-bit color on 8-bit devices" (*WinG Programmer's Reference*, 1994). WinG was an early attempt by Microsoft to bring high-performance graphics operations to the Windows platform and was superseded by DirectX. https://learn.microsoft.com/en-us/previous-versions/ms976419(v=msdn.10

[69]https://en.wikipedia.org/wiki/Chroma_key

the **color difference** between that color and a predetermined background color is small enough) and, if so, what color that pixel uses instead.

3. An application can **apply a function** to each component of an RGB or other multicomponent color, including a power function (of the form $base^{exponent}$), an inversion (an example is `[1.0 - color[0], 1.0 - color[1], 1.0 - color[2]]` for RGB colors in 0-1 format[70]), or a tone mapping curve. The function can be one-to-one, but need not be, as long as it maps numbers from 0 through 1 to numbers from 0 through 1.
4. An application can **swap** the values of any two components of an RGB or other multicomponent color to form new colors. The following example swaps the blue and red channels of an RGB color: `[color[2], color[1], color[0]]`.
5. Raster image processing techniques that process each pixel depending on neighboring pixels or the image context are largely out of scope of this document. These include pixel neighborhood filters (including Gaussian blur and other convolutions), morphological processing (including erosion and dilation), and image segmentation beyond individual pixels (including some clustering and background removal algorithms).

10 Color Differences

Color difference algorithms are used to determine if two colors are similar.

In this document, `COLORDIFF(color1, color2)` is a function that calculates a ***color difference***[71] (also known as "color distance") between two colors in the same color space, where the lower the number, the closer the two colors are. In general, however, color differences calculated using different color spaces or formulas cannot be converted to each other. This section gives some ways to implement `COLORDIFF`.

Euclidean distance. The following pseudocode implements the Euclidean distance of two multicomponent colors. This color difference formula is independent of color model.

```
// Euclidean distance for multicomponent colors
METHOD COLORDIFF(color1, color2)
    ret = 0
    for i in 0...len(color1)
       ret=ret+(color2[i]-color1[i])*(color2[i]-color1[i])
    end
    return sqrt(ret)
END METHOD
```

Notes:

- For CIELAB or CIELUV, the 1976 ΔE^*ab ("delta E a b") or ΔE^*uv color difference method, respectively[72], is the Euclidean distance between two CIELAB or two CIELUV colors, respectively.
- If Euclidean distances are merely being compared (so that, for example, two distances are not added or multiplied), then the square root operation can be omitted.

Riemersma's method. (Riemersma)[73] suggests an algorithm for color difference, to be applied to **encoded RGB colors**.

[70] This is often called the "CMY" ("cyan–magenta–yellow") version of the RGB color (although the resulting color is not necessarily based on a proportion of cyan, magenta, and yellow inks; see also "**CMYK and Other Ink-Mixture Color Models**"). If such an operation is used, the conversions between "CMY" and RGB are exactly the same.

[71] https://en.wikipedia.org/wiki/Color_difference

[72] The "E" here stands for the German word *Empfindung*.

[73] T. Riemersma, **"Colour metric"**, section "A low-cost approximation". https://www.compuphase.com/cmetric.htm

CMC. The following pseudocode implements the Color Measuring Committee color difference formula published in 1984, used above all in the textile industry. Note that in this formula, the order of the two **CIELAB** colors is important (the first color is the reference, and the second color is the test). Here, the formula is referred to as CMC(`LPARAM`:`CPARAM`) where—

- `LPARAM` is a lightness tolerance and is usually either 2 or 1, and
- `CPARAM` is a chroma tolerance and is usually 1.

```
METHOD COLORDIFF(lab1, lab2)
    c1=LabToChroma(lab1)
    c2=LabToChroma(lab2)
    h1=LabToHue(lab1)
    dl=0.511
    if lab1[0]>=16: dl=0.040975*lab1[0]/(1+0.01765*lab1[0])
    dc=0.0638+(0.0638*c1/(0.0131*c1+1))
    f4=pow(c1,4)
    f4=sqrt(f4/(f4+1900))
    dt=0
    if h1>=41*pi/45 and h1<=23*pi/12
       dt=0.56+abs(0.2*cos(h1+14*pi/15))
    else
       dt=0.36+abs(0.4*cos(h1+7*pi/36))
    end
    dh=(dt*f4+1-f4)*dc
    dl=dl*LPARAM
    dc=dc*CPARAM
    da=lab2[1]-lab1[1]
    db=lab2[2]-lab1[2]
    dchr=c2-c1
    dhue=sqrt(max(0,da*da+db*db-dchr*dchr))
    dl=((lab2[0]-lab1[0])/dl)
    dc=(dchr/dc)
    dh=(dhue/dh)
    return sqrt(dl*dl+dc*dc+dh*dh)
END METHOD
```

CIE94. This **CIELAB**-specific formula is detailed on the **supplemental color topics**[74] page.

CIEDE2000. The following pseudocode implements the color difference formula published in 2000 by the CIE, called CIEDE2000 or ΔE^*_{00}, between two **CIELAB** colors.

```
METHOD COLORDIFF(lab1, lab2)
    dl=lab2[0]-lab1[0]
    hl=lab1[0]+dl*0.5
    sqb1=lab1[2]*lab1[2]
    sqb2=lab2[2]*lab2[2]
    c1=sqrt(lab1[1]*lab1[1]+sqb1)
    c2=sqrt(lab2[1]*lab2[1]+sqb2)
    hc7=pow((c1+c2)*0.5,7)
    trc=sqrt(hc7/(hc7+6103515625.0))
    t2=1.5-trc*0.5
```

[74] https://peteroupc.github.io/suppcolor.html#Additional_Color_Formulas

```
    ap1=lab1[1]*t2
    ap2=lab2[1]*t2
    c1=sqrt(ap1*ap1+sqb1)
    c2=sqrt(ap2*ap2+sqb2)
    dc=c2-c1
    hc=c1+dc*0.5
    hc7=pow(hc,7)
    trc=sqrt(hc7/(hc7+6103515625.0))
    h1=atan2(lab1[2],ap1)
    if h1<0: h1=h1+pi*2
    h2=atan2(lab2[2],ap2)
    if h2<0: h2=h2+pi*2
    hdiff=h2-h1
    hh=h1+h2
    if abs(hdiff)>pi
      hh=hh+pi*2
      if h2<=h1: hdiff=hdiff+pi*2
      else: hdiff=hdiff-pi*2
    end
    hh=hh*0.5
    t2=1-0.17*cos(hh-pi/6)+0.24*cos(hh*2)
    t2=t2+0.32*cos(hh*3+pi/30)
    t2=t2-0.2*cos(hh*4-pi*63/180)
    dh=2*sqrt(c1*c2)*sin(hdiff*0.5)
    sqhl=(hl-50)*(hl-50)
    fl=dl/(1+(0.015*sqhl/sqrt(20+sqhl)))
    fc=dc/(hc*0.045+1)
    fh=dh/(t2*hc*0.015+1)
    dt=30*exp(-pow(36*hh-55*pi,2)/(25*pi*pi))
    r=0-2*trc*sin(2*dt*pi/180)
    return sqrt(fl*fl+fc*fc+fh*fh+r*fc*fh)
END METHOD
```

> **Note:** An improvement to CIEDE2000 (Huang et al. 2015)[75], recently recommended in CIE 230:2019 for small color differences, is not yet in common use.

Commercial factors. A *commercial factor* (`cf`) is an additional parameter to CMC and other color difference formulas. The `COLORDIFF` result is divided by `cf` (which is usually 1) to get the final color difference.

10.1 Nearest Colors

The **nearest color algorithm** is used, for example, to categorize colors or to reduce the number of colors used by an image.

In the pseudocode below, the method `NearestColorIndex` finds, for a given color (`color`), the index of the color nearest it in a given list (`list`) of colors, all in the same color space as `color`. `NearestColorIndex` is independent of color model.

```
METHOD NearestColorIndex(color, list)
   if size(list) == 0: return error
   if size(list) == 1: return 0
```

[75] Huang, M., Cui, G., et al. (2015). "Power functions improving the performance of color-difference formulas." Optical Society of America, 23(1), 597–610.

```
   i = 0
   best = -1
   bestIndex = 0
   while i < size(list)
       dist = COLORDIFF(color,list[i])
       if i == 0 or dist < best
          best = dist
          bestIndex = i
       end
       i = i + 1
   end
   return bestIndex
END METHOD
```

Examples:

- To find the nearest color to `color` in a list of colors (`list`), generate `nearestColor = list[NearestColorIndex(color, list)]`.
- Sorting colors into **color categories** can be done by a so-called "hard clustering" algorithm such as ***k*-means clustering** (see also the **Wikipedia article**[76]), which involves—
  1. defining a list (`repColors`) of k color points (which, for example, can be representative colors for red, blue, black, white, and so on, or can be colors chosen at random), then
  2. for each color (`color`) to be categorized, finding the nearest color to that color among the k color points (for example, by calling `NearestColorIndex(color, repColors)`), then
  3. replacing each color point in `repColors` with its new average color (based on the colors that point categorizes), then
  4. repeating steps 2 and 3 until the changes in all color points are negligible.

  If representative colors were used, steps 3 and 4, or step 4 itself, can be omitted. Otherwise, color points in `repColors` that end up categorizing no colors should be omitted.

11 Dominant Colors of an Image

There are several methods of finding the kind or kinds of colors that appear most prominently in an ***image color list***. For best results, these techniques need to be carried out with ***linear RGB*** rather than encoded RGB colors.

1. **Color quantization**[77]**.** In this technique, the image color list's colors are reduced to a small set of colors (for example, ten to twenty). Quantization algorithms include k-means clustering (see the previous section), recursive subdivision, and octrees.
2. **Histogram binning.** To find the dominant colors using this technique (which is independent of color model):
   - Generate or furnish a list of colors that cover the space of colors well. This is the *color palette*. A good example is the **"safety palette"** [78].

[76] https://en.wikipedia.org/wiki/K-means_clustering

[77] https://en.wikipedia.org/wiki/Color_quantization

[78] The "safety palette", also known as the "Web safe" colors, consists of 216 colors that are uniformly spaced in the red–green–blue color cube. Robert Hess's article "**The Safety Palette**)", 1996, described the advantage that images that use only colors in this palette won't dither when displayed by Web browsers on displays that can show up to 256 colors at once. (See also **Wikipedia**. Dithering is the scattering of colors in a limited set to simulate colors outside that set.) The definition of the "safety palette", though not the name, dates as early as 1994, when Microsoft's WinG API provided a *halftone palette* to "allo[w] applications to simulate true 24-bit color on 8-bit devices" (*WinG Programmer's Reference*, 1994). WinG was an early attempt by Microsoft to bring high-performance

- Create a list with as many zeros as the number of colors in the palette. This is the *histogram*.
- For each color in the image color list, find its **nearest color** in the color palette, and add 1 to the nearest color's corresponding value in the histogram.
- Find the color or colors in the color palette with the highest histogram values, and return those colors as the dominant colors.

3. **Posterization.** This involves rounding each component of a multicomponent color to the nearest multiple of $1/n$, where n is 1 plus the desired number of levels per channel. The rounding can be up, down, or otherwise.

Notes:

1. **Scale down**: For all these techniques, in the case of a raster image, an implementation can scale down that image before proceeding to find its dominant colors. Algorithms to resize or "resample" images are out of scope for this page, however.
2. **Color reduction**: Reducing the number of colors in an image usually involves finding that image's dominant colors and either—
   - applying a "nearest neighbor" approach (replacing that image's colors with their **nearest dominant colors**), or
   - applying a *dithering* technique (especially to reduce undesirable color "banding" in certain cases).[79]
3. **Unique colors**: Finding the number of unique colors in an image color list can be done by storing those colors as keys in a hash table, then counting the number of keys stored this way.[80]
4. **Disqualifying dominant colors**: An application can disqualify certain kinds of colors from being dominant, and use a substitute color as the dominant color if no dominant color remains. For example, the application can ignore colors in the background or near the image's edges, can ignore certain kinds of colors (for example, gray or nearly gray colors) while sampling the image color list, or can delete certain colors from the dominant color list.
5. Averaging the colors of an image, component-by-component, can lead to a meaningless result, especially if there is a wide color variety represented in the image (see `stackoverflow.com/questions/43111029`).
6. **Extracting a scene's "true colors"**: For applications where matching colors from the real world is important, colors need to be measured using a **color measurement device**[81], or be calculated from ***scene-referred* image data**.[82] PNG and many other image formats store image data commonly interpreted as **sRGB** by default; however, sRGB is an ***output-referred*** color space, not a scene-referred one (it's based on the color output of cathode-ray-tube monitors), making sRGB images unsuitable for real-world color-matching without more.Getting scene-referred image data from a digital camera, including a smartphone camera, is not trivial and is not discussed in detail in this document. It requires knowing, among other things, whether the camera offers access to raw image data, the format of that raw data, and possibly whether the camera does color rendering (which happens before generating output-referred image data). A raw image's colors can be estimated by

graphics operations to the Windows platform and was superseded by DirectX. https://learn.microsoft.com/en-us/previous-versions/ms976419(v=msdn.10

[79] *Dithering* is the scattering of colors in a limited set to simulate colors outside that set. Detailing the various dithering techniques is outside the scope of this article, but see the **Wikipedia article on dithering** as will as **Joel Yliluoma's algorithm and his review of other dithering algorithms**. Another way to implement dithering is mentioned in C. Peters, "**Free blue noise textures**", *Moments in Graphics*, Dec. 22, 2016. https://en.wikipedia.org/wiki/Dither https://bisqwit.iki.fi/story/howto/dither/jy/

[80] This document does not cover how to implement hash tables.

[81] https://peteroupc.github.io/suppcolor.html#Color_Measurement_Devices

[82] An example of scene-referred image data is a raw image from a digital camera after applying an input device transform as defined in Academy Procedure P-2013-001. Scene-referred image data have not undergone operations such as look modification transforms (as defined in P-2013-001), tone mapping, gamut mapping, or other color rendering.

the use of a raw image of a color calibration chart (test target) or by another technique. The ISO 17321 series and IEC 61966-9 touch on this subject.

12 Color Maps

A *color map* (or *color palette*) is a list of colors, which are usually related. All the colors in a color map can be in any one color space, but unless noted otherwise, ***linear RGB* colors** should be used rather than encoded RGB colors.

> **Example:** A **grayscale color map** consists of the encoded RGB colors `[[0, 0, 0], [0.5, 0.5, 0.5], [1, 1, 1]]`.

12.1 Kinds of Color Maps

The ***ColorBrewer 2.0*** Web site's suggestions for color maps are designed above all for visualizing data on land maps. For such purposes, C. Brewer, the creator of *ColorBrewer 2.0*, has identified **three kinds** of appropriate color maps:

- **Sequential color maps** for showing "ordered data that progress from low to high". Those found in *ColorBrewer 2.0* use varying tints of the same hue or of two close hues.
- **Diverging color maps** for showing continuous data with a clearly defined midpoint (the "critical value") and where the distinction between low and high is also visually important. Those found in *ColorBrewer 2.0* use varying tints of two "contrasting hues", one hue at each end, with lighter tints closer to the middle. Where such color maps are used in 3D visualizations, K. Moreland **recommends** "limiting the color map to reasonably bright colors".
- **Qualitative color maps** for showing discrete categories of data (see also "**Visually Distinct Colors**"). Those found in *ColorBrewer 2.0* use varying hues.

  Note: The fact that *ColorBrewer 2.0* identifies some of its color maps as being "print friendly"[83], "**color blind friendly**", or both suggests that these two factors can be important when generating color maps of the three kinds just mentioned.

12.2 Color Collections

If each color in a color map has a name, number, or code associated with it, the color map is also called a *color collection*. Examples of names are "red", "vivid green", "orange", "lemonchiffon", and "5RP 5/6"[84]. A survey of color collections or color atlases is not covered in this document, but some of them are discussed in some detail in my **colors tutorial for the HTML 3D Library**[85].

Converting a color (such as an RGB color) to a color name can be done by—

- retrieving the name keyed to that color in a hash table (or returning an error if that color doesn't exist in the hash table)[86], or
- finding the **nearest color** to that color among the named colors, and returning the name of the color found this way.

Converting a color name to a color can be done by retrieving the color keyed to that name (or optionally, its lowercase form) in a hash table, or returning an error if no such color exists.[87]

[83] In general, a color can be considered "print friendly" if it lies within the extent of colors (*color gamut*) that can be reproduced under a given or standardized printing condition (see also "**CMYK and Other Ink-Mixture Color Models**").

[84] Many color collections are represented by printed or dyed color swatches, are found in printed "fan decks", or both. Most color collections of this kind, however, are proprietary. "5RP 5/6" is an example from a famous color system and color space from the early 20th century.

[85] https://peteroupc.github.io/html3dutil/tutorial-colors.html#What__Do__Some__Colors__Look__Like

[86] This document does not cover how to implement hash tables.

[87] This document does not cover how to implement hash tables.

If each name, number, or code in a color map is associated with one or several colors, optionally with a weighting factor for each color, then the color map is also known as a *color dictionary* (Venn et al.)[88].

Notes:

- As used in the **CSS Color Module Level 3**, named colors defined in that module are expressed as encoded RGB colors in the ***sRGB color space***.
- If the color names identify points in a color space (as in the "5RP 5/6" example), converting a color name with a similar format (for example, "5.6PB 7.1/2.5") to a color can be done by multidimensional interpolation of the known color points.[89]

12.3 Visually Distinct Colors

Color maps can list colors used to identify different items. Because of this use, many applications need to use colors that are easily distinguishable by humans. In this respect—

- K. Kelly (1965) proposed a list of "twenty two colors of maximum contrast"[90], the first nine of which were intended for readers with normal and **defective** color vision, and
- B. Berlin and P. Kay, in a work published in 1969, identified eleven basic color terms: black, white, gray, purple, pink, red, green, blue, yellow, orange, and brown.

In general, the greater the number of colors used, the harder it is to distinguish them from each other. Any application that needs to distinguish many items (especially more than 22 items, the number of colors in Kelly's list) should use other visual means in addition to color (or rather than color) to help users identify them, such as numbered labels, text labels, different shapes, different shading, different dash patterns, or a combination of these. (Note that under the **Web Content Accessibility Guidelines 2.0**[91] level A, color may not be **"the only visual means of conveying information"**.)

In general, any method that seeks to choose colors that are maximally distant in a particular color space (that is, where the smallest **color difference** [`COLORDIFF`] between them is maximized as much as feasible) can be used to select visually distinct colors. Such colors can be generated in advance or while the program runs, and such colors can be limited to those in a particular *color gamut*. Here, the color difference method should be ΔE^*ab or another color difference method that takes human color perception into account. (See also (Tatarize)[92].)

12.4 Linear Gradients

A *linear gradient* is a smooth transition of two or more colors. A linear gradient consists of two or more *gradient stops*, which each consist of a point on the number line and a color located at that point. The remaining colors on the number line are linearly *interpolated* between these points, that is, the colors between any two nearby points go from one color to another at an unchanging rate.

The following pseudocode, `LinearGradientPoint`, gets the color at the specified point on the linear gradient. It takes a list, `stops`, consisting of one or more gradient stops, and `point`, the desired point on the gradient. Each gradient stop is a list containing the point and the color, in that order, and the gradient stops are sorted in ascending order by point. The method is independent of color space, but all colors passed to the

[88] Venn, A., et al. "Das Farbwörterbuch / The Colour Dictionary".

[89] This page does not detail how multidimensional interpolation works, but an example is SciPy's `griddata` method. https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.griddata.html

[90] An approximation of the colors, in order, to encoded sRGB in **HTML color format**, is as follows: "#F0F0F1", "#181818", "#F7C100", "#875392", "#F78000", "#9EC9EF", "#C0002D", "#C2B280", "#838382", "#008D4B", "#E68DAB", "#0067A8", "#F99178", "#5E4B97", "#FBA200", "#B43E6B", "#DDD200", "#892610", "#8DB600", "#65421B", "#E4531B", "#263A21". The list was generated by converting the Munsell renotations (and a similar renotation for black) to sRGB using the Python `colour` package.

[91] https://www.w3.org/TR/2008/REC-WCAG20-20081211/

[92] Tatarize, "**Color Distribution Methodology**".

method must be in the same color space and ***linear RGB* colors** should be used rather than encoded RGB colors.

```
METHOD LinearGradientPoint(stops, point)
    if size(stops)==0: return error
    if size(stops)==1: return stops[0][1]
    if point <= stops[0][0]: return stops[0][1]
    lastStop=stops[size(stops)-1]
    if point >= lastStop[0]: return lastStop[1]
    i = 0
    while i < size(stops) - 1
        i = i + 1
       s = stops[i][0]
       e = stops[i + 1][0]
        if point == s: return stops[i][1]
        if point == e: return stops[i + 1][1]
        if point < e
          interpPoint=(point - s) / (e - s)
          return Lerp3(stops[i][1],stops[i+1][1],
                       interpPoint)
        end
        i = i + 1
    end
    return lastStop[1]
end
```

> **Note:** Linear gradients are often the basis for 2-dimensional gradients such as radial gradients, or even gradients in higher dimensions. They can generally be described in terms of a *contouring function*, which returns a point on a linear gradient given an N-dimensional point. (The name comes from U.S. patent 6879327B1, "Creating gradient fills", which expired in March 2022.) For instance, a *radial gradient* can be implemented by using the following contouring function: `sqrt(x*x+y*y)`, where `x` and `y` are the coordinates of an arbitrary point in 2-dimensional space. The value of the radial gradient function can then be passed to `LinearGradientPoint` to generate the appropriate color at the specified 2-dimensional point. Note, however, that generating multidimensional gradients can cause undesirable "banding" of colors (see the notes in **"Dominant_Colors_of_an_Image"**). Ways to reduce banding include either dithering techniques[93] or adding/subtracting a small random offset ("noise") to the value of the contouring function for each 2-dimensional point.

12.5 Pseudocode

In the following pseudocode—

- `ColorMapContinuous` extracts a **continuous color** (blended color) from a color map (`colormap`), and
- `ColorMapDiscrete` extracts a **discrete color** (nearest color) from a color map (`colormap`),

where `value` is a number 0 or greater and 1 or less (0 and 1 are the start and end of the color map, respectively).

```
    METHOD ColorMapContinuous(colormap, value)
```

[93] *Dithering* is the scattering of colors in a limited set to simulate colors outside that set. Detailing the various dithering techniques is outside the scope of this article, but see the **Wikipedia article on dithering** as will as **Joel Yliluoma's algorithm and his review of other dithering algorithms**. Another way to implement dithering is mentioned in C. Peters, "**Free blue noise textures**", *Moments in Graphics*, Dec. 22, 2016. https://en.wikipedia.org/wiki/Dither https://bisqwit.iki.fi/story/howto/dither/jy/

```
    nm1 = size(colormap) - 1
    index = (value * nm1) - floor(value * nm1)
    if index >= nm1: return colormap[index]
    fac = (value * nm1) - index)
    list1 = colormap[index]
    list2 = colormap[index + 1]
    return [list1[0]+(list2[0]-list1[0])*fac, list1[1]+(list2[1]-list1[1])*fac,
        list1[2]+(list2[2]-list1[2])*fac]
END METHOD

METHOD ColorMapDiscrete(colormap, value)
   vn1=value*(N-1)
   if floor(vn1)<0.5: return colormap[floor(vn1)]
   return colormap[ceil(vn1)]
END METHOD
```

Example: The idiom `ColorMapContinuous(colormap, 1 - value)` gets a continuous color from the reversed version of a color map.

13 Generating a Random Color

The following techniques can be used to generate random colors. In this section:

- `RNDRANGEMinMaxExc`, `RNDINT`, and `RNDINTEXC` are methods defined in my article on **random number generation methods**[94].
- Some techniques here refer to a ***light–dark factor***. This factor can be implemented by any of the following, in order of preference from most to least.
  1. The color's **CIELAB** lightness (_L*_) divided by 100, or another value from 0 through 1 that expresses a color's lightness (in terms of human perception).
  2. `Luminance(color)`.
  3. Any other single number that summarizes a color and ranges from 0 ("minimum intensity") to 1 ("maximum intensity").
- For best results, these techniques need to use ***linear RGB* colors** rather than encoded RGB colors, unless noted otherwise.

The techniques follow.

- Generating a random string in the **HTML color format** can be done by generating a **random hexadecimal string**[95] with length 6, then inserting the string "#" at the beginning of that string.
- Generating a random three-component color in the **0-1 format** can be done as follows: `[RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1)]`.
- Generating a random **8-bpc encoded RGB color** can be done as follows: `From888(RNDINT(16777215))`.
- To generate a random **dark RGB color**, either—
  - generate `color = [RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1)]` until a *light–dark factor* is less than a given threshold, for example, 0.5, or
  - generate `color = [RNDRANGEMinMaxExc(0, maxComp), RNDRANGEMinMaxExc(0, maxComp), RNDRANGEMinMaxExc(0, maxComp)]`, where `maxComp` is the maximum value of each color component, for example, 0.5.
- To generate a random **light RGB color**, either—
  - generate `color = [RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1)]` until a *light–dark factor* is greater than a given threshold, for example, 0.5, or

[94] https://peteroupc.github.io/randomfunc.html
[95] https://peteroupc.github.io/randomfunc.html#Creating_a_Random_Character_String

  - generate `color = [minComp + RNDRANGEMinMaxExc(0, 1) * (1.0 - minComp), minComp + RNDRANGEMinMaxExc(0, 1) * (1.0 - minComp), minComp + RNDRANGEMinMaxExc(0, 1) * (1.0 - minComp)]`, where `minComp` is the minimum value of each color component, for example, 0.5.
- One way to generate a random **pastel RGB color** is to generate `color = [RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1), RNDRANGEMinMaxExc(0, 1)]` until a *light–dark factor* is greater than 0.75 and less than 0.9.
- To generate a **random three-component color at or between two others** (`color1` and `color2`), generate `Lerp3(color1, color2, RNDRANGEMinMaxExc(0, 1))`.
- To generate a **random shade** of a given RGB color, generate `Lerp3(color1, [0, 0, 0], RNDRANGEMinMaxExc(0.2, 1.0))`.
- To generate a **random tint** of a given RGB color, generate `Lerp3(color1, [1, 1, 1], RNDRANGEMinMaxExc(0.0, 0.9))`.
- To generate a **random tone** of a given RGB color, generate `Lerp3(color1, [0.5, 0.5, 0.5], RNDRANGEMinMaxExc(0.0, 0.9))`.
- To generate a **random monochrome RGB color**, generate `HslToRgb(H, RNDRANGEMinMaxExc(0, 1),RNDRANGEMinMaxExc(0, 1))`, where `H` is an arbitrary **hue**.
- **Random color sampling:**
  - To select a random continuous color from a color map (`colormap`): `ColorMapContinuous(colormap, RNDRANGEMinMaxExc(0, 1))`.
  - To select one random color from a color map (`colormap`): `colormap[RNDINTEXC(size(colormap))]`. See also **"Sampling With Replacement: Choosing a Random Item from a List"**[96].
  - To select several random colors from a color map: See **"Sampling Without Replacement: Choosing Several Unique Items"**[97].
- **Similar random colors:** Generating a random color that's similar to another can be done by generating a random color (`color1`) until `COLORDIFF(color1, color2)` (defined **earlier**) is less than a predetermined threshold, where `color2` is the color to compare.
- **Image noise:** This alters a color using random numbers, such as by adding or multiplying random numbers to that color. For example, in *uniform noise*, each component of a multicomponent color is changed to `min(1,max(0,c+RNDRANGEMinMaxExc(-level, level)))`, where `c` is the value of the previous component and `level` is the noise level. Other kinds of image noise include noise following a Gaussian, Poisson, or other **probability distribution**[98], and *salt-and-pepper noise* that involves replacing each pixel by black or white at a predetermined probability each.

  Note: The methods in this section can also be implemented by using a ***hash function***[99] to convert arbitrary data to "random" bits which can be used either directly or to initialize a pseudorandom number generator which can generate further "random" bits. For example, `From888(MD5_24("Hello World"))`, where `MD5_24()` is the first 24 bits of the MD5 hash, can be interpreted as an 8-bpc encoded RGB color.

14 Spectral Color Functions

As mentioned earlier, color requires the existence of *light*, an *object*, and an *observer*. These three things can be specified as follows:

- **Light.** A light source can be specified as a *spectral power distribution* (SPD), a "curve" that describes the intensity of a light source across the electromagnetic spectrum.
- **Object.** There are two kinds of "objects": **reflective** (opaque) and **transmissive** (translucent or transparent). A *reflectance curve* or *transmittance curve*, respectively, describes the fraction of light

[96] https://peteroupc.github.io/randomfunc.html#Sampling_With_Replacement_Choosing_a_Random_Item_from_a_List
[97] https://peteroupc.github.io/randomfunc.html#Sampling_Without_Replacement_Choosing_Several_Unique_Items
[98] https://peteroupc.github.io/randomfunc.html#Specific_Non_Uniform_Distributions
[99] https://peteroupc.github.io/random.html#Hash_Functions

that is reflected by or passes through the object, respectively.

- **Observer.** An observer's visual response can be modeled by three *color-matching functions*.

The SPD, the reflectance or transmittance curve, and the color-matching functions, are converted to three numbers (called *tristimulus values*) that uniquely identify a perceived color.

The pseudocode below includes a `SpectrumToTristim` method for computing tristimulus values. In the method:

- `lightFunc(wl)`, `reflFunc(wl)`, and `cmfFunc(wl)` are arbitrary functions described next. All three take a ***wavelength*** (`wl`) in nanometers (nm) and return the corresponding values at that wavelength. (*See also note 1 later in this section.*)
- `lightFunc(wl)` models a **light source's SPD**; it returns the source's relative intensity at the wavelength `wl`. Choices for `lightFunc` include—
  - a CIE daylight illuminant such as the D65 or D50 illuminant (see the **Python sample code**[100] for implementation),
  - the `BlackbodySPD` method given in "**Color Temperature**", and
  - the SPD for a light-emitting diode (LED), fluorescent, or other artificial light source.
- `reflFunc(wl)` models the **reflectance or transmittance curve** and returns the value of that curve at the wavelength `wl`; the value is 0 or greater and usually 1 or less. (For optically brightened and other photoluminescent and fluorescent materials, the curve can have values greater than 1.)
- `cmfFunc(wl)` models three **color-matching functions** and returns a list of those functions' values at the wavelength `wl`. The choice of `cmfFunc` determines the kind of tristimulus values returned by `SpectrumToTristim`. Choices for `cmfFunc` include the CIE 1931 or 1964 *standard observer*, which is used to generate **XYZ colors** based on color stimuli seen at a 2-degree or 10-degree field of view, respectively.[101]

```
METHOD SpectrumToTristim(reflFunc, lightFunc, cmfFunc)
    i = 360 // Start of relevant part of spectrum
    xyz=[0,0,0]
    weight = 0
    // Sample at 5 nm intervals
    while i <= 830 // End of relevant part of spectrum
             cmf=cmfFunc(i)
             refl=reflFunc(i)
             specification=lightFunc(i)
             weight=weight+cmf[1]*specification*5
             xyz[0]=xyz[0]+refl*specification*cmf[0]*5
             xyz[1]=xyz[1]+refl*specification*cmf[1]*5
             xyz[2]=xyz[2]+refl*specification*cmf[2]*5
             i = i + 5
    end
    if weight==0: return xyz
    // NOTE: Note that `weight` is constant for a given
    // color-matching function set and light source together,
    // so that `weight` can be precomputed if they will
    // not change.
    // NOTE: If `weight` is 1/683, `cmfFunc` outputs XYZ
```

[100] https://peteroupc.github.io/colorutil.zip

[101] The CIE publishes **tabulated data** for the D65 illuminant and the CIE 1931 and 1964 standard observers at its Web site. In some cases, the CIE 1931 standard observer can be approximated using the methods given in Wyman, Sloan, and Shirley, **"Simple analytic approximations to the CIE XYZ color matching functions"**, Journal of Computer Graphics Techniques 2(2), 2013, pp. 1-11.

```
    // values, and `reflFunc` always returns 1, then SpectrumToTristim
    // will give out XYZ values where Y is a value in cd/m^2.
    xyz[0] = xyz[0] / weight
    xyz[1] = xyz[1] / weight
    xyz[2] = xyz[2] / weight
    return xyz
END METHOD

// Models a perfect reflecting diffuser or
// perfect transmitting diffuser
METHOD PerfectWhite(wavelength)
    return 1
END METHOD
```

Notes:

1. Although `lightFunc`, `reflFunc`, and `cmfFunc` are actually continuous functions, in practice tristimulus values are calculated based on measurements at discrete wavelengths. For example, CIE Publication 15 recommends a 5-nm wavelength interval. For spectral data at 10-nm and 20-nm intervals, the practice described in ISO 13655 or in ASTM International E308 and E2022 can be used to compute tristimulus values (in particular, E308 includes tables of weighting factors for common combinations of `cmfFunc` and `lightFunc`). For purposes of color reproduction, only wavelengths within the range 360-780 nm (0.36-0.78 m) are relevant in practice.
2. **Metamerism** occurs when two materials match the same color under one viewing situation (such as light source, `lightFunc`, or viewer, `cmfFunc`, or both), but not under another. If this happens, the two materials' reflectance or transmittance curves (`reflFunc`) are called *metamers*. For applications involving real-world color matching, metamerism is why reflectance and transmittance curves (`reflFunc`) can be less ambiguous than colors in the form of three tristimulus values (such as XYZ or RGB colors). (See also **B. MacEvoy's principle 38**.)

Examples: In these examples, `D65` is the D65 illuminant, `D50` is the D50 illuminant, `CIE1931` is the CIE 1931 standard observer, and `refl` is an arbitrary reflectance curve.

1. `SpectrumToTristim(refl, D65, CIE1931)` computes the reflectance curve's **XYZ color** (where a Y of 1 is the D65/2 white point).
2. `SpectrumToTristim(refl, D50, CIE1931)` is the same, except white is the D50/2 white point.
3. `SpectrumToTristim(PerfectWhite, light, cmf)` computes the white point for the specified illuminant `light` and the color matching functions `cmf`.
4. `SpectrumToTristim(PerfectWhite, D65, CIE1931)` computes the D65/2 white point.
5. `XYZTosRGB(SpectrumToTristim(refl, D65, CIE1931))` computes the reflectance curve's **encoded sRGB** color.
6. `XYZTosRGB(CIE1931(wl))` computes the encoded sRGB color of a light source that emits light only at the wavelength `wl` (a *monochromatic stimulus*), where the wavelength is expressed in nm.

14.1 Color Temperature

A *blackbody* is an idealized material that emits light based only on its temperature. As a blackbody's temperature goes up, its chromaticity changes from red to orange to pale yellow up to sky blue.

The `Planckian` method shown next models the spectral power distribution (SPD) of a blackbody with the specified temperature in kelvins (its **color temperature**). The `BlackbodySPD` method below uses that

method (where `TEMP` is the desired color temperature).[102]. Note that such familiar light sources as sunlight, daylight, candlelight, and incandescent lamps can be closely described by the appropriate blackbody SPD.

```
METHOD Planckian(wl, temp)
    num = pow(wl, -5)
    // NOTE: 0.014... was calculated based on
    // 2017 versions of Planck and Boltzmann constants, and
    // is rounded off to a limited number of decimal places.
    return num / (exp(0.0143877687750393/(wl*pow(10, -9)*temp)) - 1)
END METHOD

METHOD BlackbodySPD(wl) # NOTE: Relative only
    t=TEMP
    if t<60: t=60 # For simplicity, in very low temperature
    return Planckian(wl, t) * 100.0 /
        Planckian(560, wl)
END METHOD
```

> **Note:** If `TEMP` is 2856, the `BlackbodySPD` function above is substantially equivalent to the CIE illuminant A.

The concept "color temperature" properly applies only to blackbody chromaticities. For chromaticities close to a blackbody's, the CIE defines ***correlated color temperature*** (CCT) as the temperature of the blackbody with the closest (u, v) coordinates[103] to those of the specified color. The CCT calculation uses the CIE 1931 standard observer. (According to the CIE, however, CCT is not meaningful if the straight-line distance between the two (u, v) points is more than 0.05.)

The following method (`XYZToCCT`), which computes an approximate CCT from an **XYZ color**, is based on McCamy's formula from 1992.

```
METHOD XYZToCCT(xyz)
    xyy = XYZToxyY(xyz)
    c = (xyy[0] - 0.332) / (0.1858 - xyy[1])
    return ((449*c+3525)*c+6823.3)*c+5520.33
END METHOD
```

> **Note:** Color temperature, as used here, is not to be confused with the division of colors into *warm* (usually red, yellow, and orange) and *cool* (usually blue and blue green) categories, a subjective division which admits of much variation. But in general, in the context of light sources, the lower the light's CCT, the "warmer" the light appears, and the higher the CCT, the "cooler". However, CCT (or any other single number associated with a light source) is generally inadequate by itself to describe how a light source renders colors.

14.2 Color Mixture

The mixture of two colorants can be complex, and there are several approaches to simulating this kind of color mixture.

- As **S. A. Burns indicates**, two or more ***reflectance curves***, each representing a **pigment or colorant**, can be mixed by calculating their *weighted geometric mean*, which takes into account the

[102] See also J. Walker, "**Colour Rendering of Spectra**".

[103] **CIE Technical Note 001:2014** says the chromaticity difference ($\Delta u' v'$) should be calculated as the **Euclidean distance** between two $u' v'$ pairs and that a chromaticity difference of 0.0013 is just noticeable "at 50% probability". (u, v) coordinates, a former 1960 version of u' and v', are found by taking u as u' and v as (v' * 2.0 / 3).

relative proportions of those colorants in the mixture; the result is a new reflectance curve that can be converted into an RGB color.[104]

- As **B. MacEvoy indicates**, two or more spectral curves for **transmissive materials** can be mixed simply by multiplying them; the result is a new spectral curve for the mixed material.
- An alternative method of color formulation, based on the **Kubelka–Munk theory**, uses two curves for each colorant: an *absorption coefficient* curve (K curve) and a *scattering coefficient* curve (S curve). The ratio of absorption to scattering (K/S) has a simple relationship to reflectance factors in the Kubelka–Munk theory. The Python sample code implements the Kubelka–Munk equations. One way to predict a color formula using this theory is described by E. Walowit in 1985[105]. ISO 18314-2 is also a relevant document.

For convenience, the `WGM` method below computes the weighted geometric mean of one or more numbers, where—

- `values` is a list of values (for example, single values of several reflectance curves at the same point), and
- `weights` is a list of those values' corresponding weights (for example, mixing proportions of those curves).

```
METHOD WGM(values, weights)
    if size(values)!=size(weights): return error
    if size(values)==0: return values[0]
    sum=0
    i=0
    while i < size(weights)
      sum=sum+weights[i]
      i=i+1
    end
    if sum<=0: return error
    ret=1
    while i < size(values)
      ret=ret*pow(values[i],weights[i]/sum)
      i=i+1
    end
    return ret
END METHOD
```

15 Conclusion

This page discussed many topics on color that are generally relevant in programming.

Feel free to send comments. They may help improve this page. In particular, corrections to any method given on this page are welcome.

I acknowledge—

[104] As **B. MacEvoy explains** (at "Other Factors in Material Mixtures"), things that affect the mixture of two colorants include their "refractive index, particle size, crystal form, hiding power and tinting strength" (see also his **principles 39 to 41**), and "the material attributes of the support [for example, the paper or canvas] and the paint application methods" are also relevant here. These factors, to the extent the reflectance curves don't take them into account, are not dealt with in this method.

[105] Walowit, E. "Spectrophotometric color formulation based on two-constant Kubelka-Munk theory". Thesis, Rochester Institute of Technology, 1985. The following reference may also be of interest: Furferi, R., Carfagni, R., "An As-Short-as-Possible Mathematical Assessment of Spectrophotometric Color Matching", *Journal of Applied Sciences*, 2010.

- the CodeProject user Mike-MadBadger, who suggested additional clarification on color spaces and color models,
- "RawConvert" from the pixls.us discussion forum,
- Elle Stone, and
- Thomas Mansencal.

The following topics may be added in the future based on reader interest:

- The CAM02 color appearance model.
- The perception-based color spaces OkLab, OkLch, and HSLuv.
- Color rendering metrics for light sources, including color rendering index (CRI) and the metrics given in TM-30-15 by the Illuminating Engineering Society.

Descriptions on the following methods would greatly enhance this document, as long as the methods are not covered by any active patents or pending patent applications and can be implemented by public-domain source code (usable for any purpose):

- A method for performing color calibration and color matching using a smartphone's camera and, possibly, a color calibration card or white balance card.
- A method to convert two RGB colors into an RGB color that closely matches how the mixture of two pigments of the input colors would appear on paper.[106]
- A method to match a desired color on paper given spectral reflectance curves of the paper and of the inks being used in various concentrations.

16 Notes

17 License

[106] Mixbox appears to satisfy this, but the **repository's source code** is under a noncommercial license; whether the algorithm itself is so is uncertain. https://github.com/scrtwpns/mixbox

[107] https://creativecommons.org/publicdomain/zero/1.0/