CSS Color Module Level 4

Editor's Draft, 8 September 2023

▼ More details about this document

This version:
https://drafts.csswg.org/css-color-4/

Latest published version:
https://www.w3.org/TR/css-color-4/

Previous Versions:
https://www.w3.org/TR/2022/CR-css-color-4-20221101/

Implementation Report:
https://wpt.fyi/results/css/css-color

Feedback:
CSSWG Issues Repository
Inline In Spec

Editors:
Tab Atkins Jr. (Google)
Chris Lilley (W3C)
Lea Verou (Invited Expert)

Former Editor:
L. David Baron (Mozilla)

Suggest an Edit for this Spec:
GitHub Editor

Test Suite:
https://wpt.fyi/results/css/css-color/

Chrome 117	Firefox 117	Safari 16	Edge 117
6306/6353	6197/6353	6102/6353	6301/6353

Abstract

This specification describes CSS <color> values, and properties for foreground color and group opacity.

CSS is a language for describing the rendering of structured documents (such as HTML and XML) on screen, on paper, etc.

Status of this document

This is a public copy of the editors' draft. It is provided for discussion only and may change at any moment. Its publication here does not imply endorsement of its contents by W3C. Don't cite this document other than as work in progress.

Please send feedback by filing issues in GitHub (preferred), including the spec code "css-color" in the title, like this: "[css-color] *…summary of comment…*". All issues and comments are archived. Alternately, feedback can be sent to the (archived) public mailing list www-style@w3.org.

This document is governed by the 12 June 2023 W3C Process Document.

The following features are at-risk, and may be dropped during the CR period:

- Equivalence of deprecated and un-deprecated system colors

"At-risk" is a W3C Process term-of-art, and does not necessarily imply that the feature is in danger of being dropped or delayed. It means that the WG believes the feature may have difficulty being interoperably implemented in a timely manner, and marking it as such allows the WG to drop the feature if necessary when transitioning to the Proposed Rec stage, without having to publish a new Candidate Rec without the feature first.

Table of Contents

§ 1. Introduction

This section is not normative.

This module describes CSS properties which allow authors to specify the foreground color and opacity of the text content of an element. This module also describes in detail the CSS <color> value type.

It not only defines the color-related properties and values that already exist in CSS1, CSS2, and CSS Color 3, but also defines new properties and values.

In particular, it allows specifying colors in other color spaces than sRGB; previously, the more saturated colors outside the sRGB gamut could not be used in CSS even if the display device supported them.

A draft implementation report is available.

§ 1.1. Value Definitions

This specification follows the CSS property definition conventions from [CSS2] using the value definition syntax from [CSS-VALUES-3]. Value types not defined in this specification are defined in CSS Values & Units [CSS-VALUES-3]. Combination with other CSS modules may expand the definitions of these value types.

In addition to the property-specific values listed in their definitions, all properties defined in this specification also accept the CSS-wide keywords as their property value. For readability they have not been repeated explicitly.

§ 2. Color Terminology

A ***color*** is a definition (numeric or textual) of the human visual perception of a light or a physical object illuminated with light. The objective study of human color perception is termed colorimetry.

The color of a physical object depends on how much light it reflects at each visible wavelength, plus the actual color of the light illuminating it (again, the amount of light at each wavelength). It is measured by a *spectrophotometer* .

The color of something that emits light (including colors on a computer screen) depends on how much light it emits at each visible wavelength. It is measured by a *spectroradiometer*.

If two objects have different spectra, but still produce the same physical sensation, we say they have the same color. We can calculate whether two colors are the same by converting the spectra to CIE XYZ (three numbers).

> EXAMPLE 1 ¶
> For example a green leaf, a photograph of that leaf displayed on a computer screen, and a print of that photograph, are all producing a green sensation by different means. If the screen and the printer are calibrated, the green in the leaf, and the photo, and the print will look the same.

A ***color space*** is an organization of colors with respect to an underlying colorimetric model, such that there is a clear, objectively-measurable meaning for any color in that color space. This also means that the same color can be expressed in multiple color spaces, or transformed from one color space to another, while still looking the same.

> EXAMPLE 2 ¶
>
> A leaf is measured with a spectrophotometer and found to have the color `lch(51.2345% 21.2 130)` which is `lab(51.2345% -13.6271 16.2401)`.
>
> This same color could be expressed in various color spaces:

```
color(sRGB 0.41587 0.503670 0.36664);
color(display-p3 0.43313 0.50108 0.37950);
color(a98-rgb 0.44091 0.49971 0.37408);
color(prophoto-rgb 0.36589 0.41717 0.31333);
color(rec2020 0.42210 0.47580 0.35605);
```

An ***additive color space*** means that the coordinate system is linear in light intensity. The CIE XYZ color space is an additive color space (in addition, the Y component of XYZ is the luminance).

In an additive color space, calculations can be done to accurately predict color mixing. Most RGB spaces are not additive, because the components are *gamma encoded*. Undoing this gamma encoding produces linear-light values.

EXAMPLE 3 ¶

For example, if a light fixture contains two identical colored lights, and only one is switched on, and the color is measured to be color(xyz 0.13 0.12 0.04), then the color when both are switched on will be exactly twice that, color(xyz 0.26 0.24 0.08).

If we have two differently colored spotlights shining on a stage, and one has the measured value color(xyz 0.15 0.24 0.17) while the other is color(xyz 0.11 0.06 0.06) then we can accurately predict that if the colored beams are made to overlap, the color of the mixture will be the sum of the XYZ component values, or color(xyz 0.26 0.30 0.23).

A ***chromaticity*** is a color measurement where the lightness component has been factored out. From the identical lights example above, the u',v' chromaticity with one light is (0.2537, 0.5 and the chromaticity is the same with both lights (they are the same color, it is just brighter).

Chromaticities are additive, so they accurately predict the chromaticity (but not the resulting lightness) of a mixture. Being two-dimensional, chromaticity is easily represented on a *chromaticity diagram* to predict the chromaticity of a color mixture. Any two colors can be mixed, and the resulting colors will lie on the line joining them on the diagram. Three colors form a plane, and the resulting colors will lie in the triangle they form on the diagram.

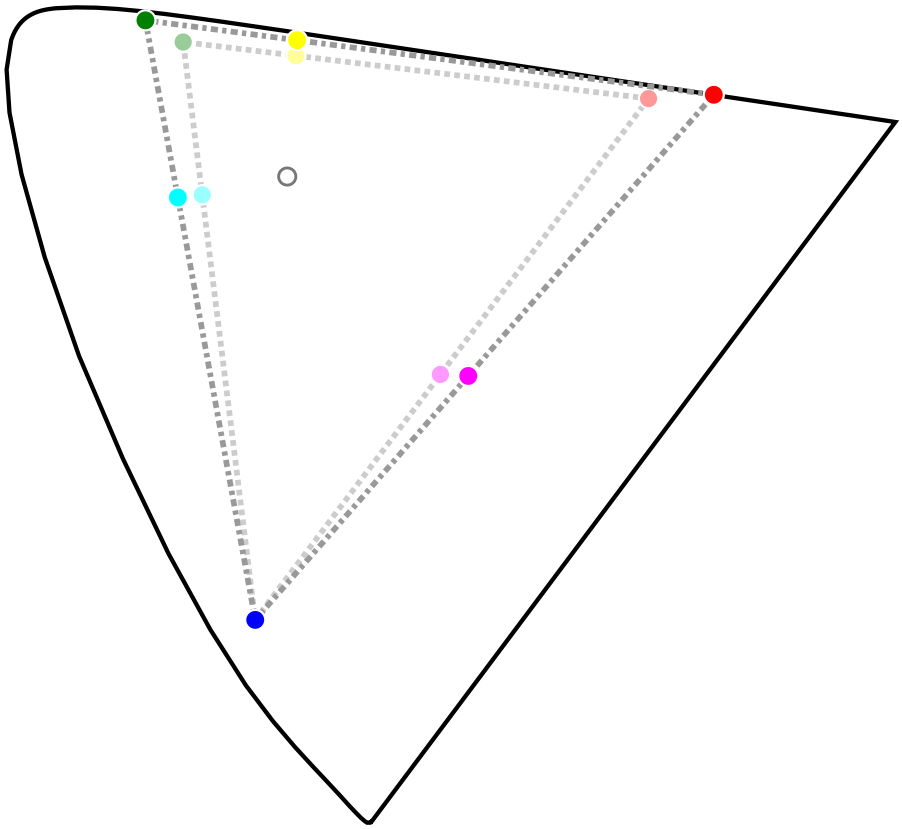


Figure 1 *A chromaticity diagram showing (in solid colors) the 'display-p3' color space and for comparison (faded) the 'sRGB' color space. The white point (D65) is also shown.*

RGB color spaces are additive, and their gamut is defined by the chromaticities of the red, green and blue primaries, plus the chromaticity of the ***white point*** (the color formed by all three primaries at full intensity).

Most color spaces use one of a few daylight-simulating white points, which are named by the color temperature of the corresponding black-body radiator. For example, D65 is a daylight whitepoint corresponding to a correlated color temperature of 6500 Kelvin (actually 6504, because the value of Plank's constant has changed since the color was originally defined).

To avoid cumulative round-trip errors, it is important that the identical chromaticity values are used consistently, at all places in a calculation. Thus, for maximum compatibility, for this specification, the following two standard daylight-simulating white points are defined:

Name	x	y	CCT
D50	0.345700	0.358500	5003K
D65	0.312700	0.329000	6504K

When the measured physical characteristics (such as the chromaticities of the primary colors it uses, or the colors produced in response to a given set of inputs) of a color space or a color-producing device are known, it is said to be ***characterized***.

If in addition adjustments have been made so that a device meets calibration targets such as white point, neutrality of greys, predictability and consistency of tone response, then it is said to be ***calibrated***.

Real physical devices cannot yet produce every possible color that the human eye can see. The range of colors that a given device can produce is termed the ***gamut*** *(not to be confused with gamma)*. Devices with a limited gamut cannot produce very saturated colors, like those found in a rainbow.

The gamuts of different color spaces may be compared by looking at the volume (in cubic Lab units) of colors that can be expressed. The following table examines the predefined color spaces available in CSS.

color space	Volume (million Lab units)
sRGB	0.820
display-p3	1.233
a98-rgb	1.310
prophoto-rgb	2.896
rec2020	2.042

A color in CSS is either an ***invalid color***, as described below for each syntactic form, or a valid color.

Any color which is not an invalid color is a ***valid color***.

A color may be a valid color but still be outside the range of colors that can be produced by an output device (a screen, projector, or printer)

It is said to be ***out of gamut***.

Each valid color is either ***in-gamut*** for a particular output device (screen, or printer) or it is out of gamut.

EXAMPLE 4

For example, given a screen which covers 100% of the display-p3 color space, but no more, the following color is out of gamut:

```
color(prophoto-rgb 0.88 0.45 0.10)
```

because, expressed in display-p3, one or more coordinates are either greater that 1.0 or less than 0.0:

```
color(display-p3 1.0844 0.43 0.1)
```

This color is valid, and could, for example, be used as a gradient stop, but would need to be CSS gamut mapped for display, producing a similar-looking but lower chroma (less saturated) color.

3. Applying Color in CSS

3.1. Accessibility and Conveying Information By Color

Although colors can add significant information to documents and make them more readable, color by itself should not be the sole means to convey important information. Authors should consider the W3C Web Content Accessibility Guidelines [WCAG21] when using color in their documents.

> *1.4.1 Use of Color:* Color is not used as the only visual means of conveying information, indicating an action, prompting a response, or distinguishing a visual element

3.2. Foreground Color: the 'color' property

Name:	*'color'*
Value:	<color>
Initial:	CanvasText
Applies to:	all elements and text
Inherited:	yes

Percentages:	N/A
Computed value:	computed color, see resolving color values
Canonical order:	per grammar
Animation type:	by computed value type

▼ TESTS

color-001.html	(live test) (source)
color-002.html	(live test) (source)
color-003.html	(live test) (source)
inheritance.html (4 tests)	(live test) (source)
color-interpolation.html (192 tests)	(live test) (source)
color-initial-canvastext.html (2 tests)	(live test) (source)
color-valid.html (16 tests)	(live test) (source)
color-invalid.html (10 tests)	(live test) (source)

This property specifies the primary foreground color of the element. This is used as the fill color of its text content, and in addition specifies the used value that 'currentcolor' resolves to, which allows indirect references to this foreground color and affects the initial values of various other color properties such as 'border-color' and 'text-emphasis-color'.

<color>

Sets the primary foreground color to the specified <color>.

✔ MDN

EXAMPLE 5 ¶

The <color> type provides multiple ways to syntactically specify a given color. For example, the following declarations all specify the sRGB color "lime":

```
em { color:      lime; }   /* color keyword  */
em { color:      rgb(0 255 0); } /* RGB range 0–255   */
em { color:      rgb(0% 100% 0%); } /* RGB range 0%–100% */
em { color:      color(sRGB 0 1 0); } /* sRGB range 0.0–1.0 */
```

When applied to text, this property, including its alpha component, has no effect on "color glyphs" (such as the emoji in some fonts), which are colored by a built-in palette. However, some colored fonts are able to refer to a contextual "foreground color", such as by palette entry

'0xFFFF' in the `COLR` table of OpenType, or by the 'context-fill' value in SVG-in-OpenType. In such cases, the foreground color is set by this property, identical to how it sets the 'currentcolor' value.

§ 3.3. Transparency: the 'opacity' property

Opacity can be thought of as a postprocessing operation. Conceptually, after the element (including its descendants) is rendered into an RGBA offscreen image, the opacity setting specifies how to blend the offscreen rendering into the current composite rendering. See simple alpha compositing for details.

Name:	*'opacity'*
Value:	<alpha-value>
Initial:	1
Applies to:	all elements
Inherited:	no
Percentages:	map to the range [0,1]
Computed value:	specified number, clamped to the range [0,1]
Canonical order:	per grammar
Animation type:	by computed value type

▼ TESTS

opacity-computed.html (8 tests)	(live test)	(source)
opacity-valid.html (16 tests)	(live test)	(source)
opacity-invalid.html (3 tests)	(live test)	(source)
composited-filters-under-opacity.html	(live test)	(source)
filters-under-will-change-opacity.html	(live test)	(source)
color-composition.html (10 tests)	(live test)	(source)
opacity-interpolation.html (120 tests)	(live test)	(source)
canvas-change-opacity.html	(live test)	(source)
opacity-animation-ending-correctly-001.html	(live test)	(source)

opacity-animation-ending-correctly-002.html (live test) (source)

<alpha-value>

The opacity to be applied to the element. It is interpreted identically to its definition in 'rgb()', except that the resulting opacity is applied to the entire element, rather than a particular color.

▼ TESTS

inline-opacity-float-child.html (live test) (source)

The 'opacity' property applies the specified opacity to the element *as a whole*, including its contents, rather than applying it to each descendant individually. This means that, for example, an opaque child occluding part of the element's background will continue to do so even when 'opacity' is less than 1, but the element and child as a whole will show the underlying page through themselves.

It also means that the glyphs corresponding to all characters in the element are treated *as a whole*; any overlapping portions do not increase the opacity.

▼ TESTS

opacity-overlapping-letters.html (live test) (source)

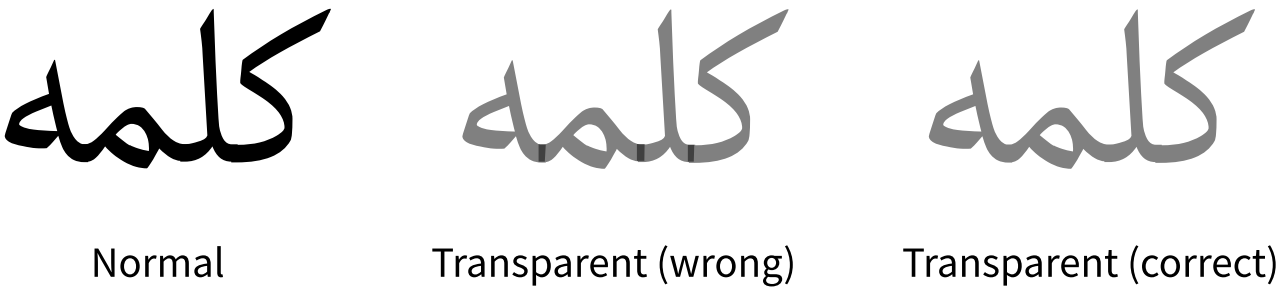


Figure 2 *Correct and incorrect rendering of text with an 'opacity' value of less than one, whose glyphs overlap.*

If separate opacity for each glyph is desired, it can be achieved by using a color value which includes opacity, rather than setting the 'opacity' property.

If a box has 'opacity' less than 1, it forms a stacking context for its children. (This prevents its contents from interleaving in the z-axis with content outside it.)

▼ TESTS

body-opacity-0-to-1-stacking-context.html (live test) (source)

Furthermore, if the 'z-index' property applies to the box, the 'auto' value is treated as '0' for the element; it is otherwise painted on the same layer within its parent stacking context as positioned elements with stack level 0 (as if it were a positioned element with 'z-index:0').

See section 9.9 and Appendix E of [CSS2] for more information on stacking contexts.

These rules about z-order do not apply to SVG elements, since SVG has its own rendering model ([SVG11], Chapter 3).

§ 3.4. Color Space of Tagged Images

An ***tagged image*** is an image that is explicitly assigned a color profile, as defined by the image format. This is usually done by including an International Color Consortium (ICC) profile [ICC].

For example JPEG [JPEG], PNG [PNG] and TIFF [TIFF] all specify a means to embed an ICC profile.

Image formats may also use other, equivalent methods, often for brevity.

For example, PNG specifies a means (the sRGB chunk) to explicitly tag an image as being in the sRGB color space, without including the sRGB ICC profile.

Tagged RGB images, and tagged images using a transformation of RGB such as YCbCr, if the color profile or other identifying information is valid, must be treated as being in the specified color space.

▼ TESTS

tagged-images-001.html	(live test)	(source)
tagged-images-002.html	(live test)	(source)
tagged-images-003.html	(live test)	(source)
tagged-images-004.html	(live test)	(source)

If the color profile or other identifying information is invalid, the image is treated as untagged images

For example, when a browser running on a system with a Display P3 monitor displays an JPEG image tagged as being in the ITU Rec BT.2020 [Rec.2020] color space, it must convert the colors from ITU Rec BT.2020 to Display P3 so that they display correctly. It must not treat the ITU Rec BT.2020 values as if they were Display P3 values, which would produce incorrect colors.

§ 3.5. Color Spaces of Untagged Colors

Colors specified in HTML, and untagged images must be treated as being in the sRGB color space ([SRGB]) unless otherwise specified.

▼ TESTS
untagged-images-001.html (live test) (source)

An ***untagged image*** is an image that is not explicitly assigned a color profile, as defined by the image format.

Note that this rule does not apply to untagged videos, since ***untagged video*** should be presumed to be in an ITU-defined color space.

- At below 720p, it is Recommendation ITU-R BT.601 [ITU-R-BT.601]
- At 720p, it is SMPTE ST 296 (same colorimetry as 709) [SMPTE296]
- At 1080p, it is Recommendation ITU-R BT.709 [ITU-R-BT.709]
- At 4k (UHDTV) and above, it is ITU-R BT.2020 [Rec.2020] for SDR video

§ 4. Representing Colors: the <color> type

Colors in CSS are represented as a list of color components, also sometimes called "channels", representing axises in the color space. Each channel has a minimum and maximum value, and can take any value between those two. Additionally, every color is accompanied by an ***alpha component***, indicating how transparent it is, and thus how much of the backdrop one can see through the color.

CSS has several syntaxes for specifying color values:

- the sRGB hex color notation which represents the RGB and alpha channels in hexadecimal notation
- the various color functions which can represent colors using a variety of color spaces and coordinate systems
- the constant named color keywords
- the variable <system-color> keywords and 'currentColor' keyword.

✔ MDN

The ***color functions*** use CSS functional notation to represent colors in a variety of color spaces by specifying their channel coordinates. Some of these use a ***cylindrical polar color*** model, specifying color by a <hue> angle, a central axis representing lightness (black-to-white), and a radius representing saturation or chroma (how far the color is from a neutral grey). The others use a ***rectangular orthogonal color*** model, specifying color using three orthogonal component axes.

The color functions available in Level 4 are

- 'rgb()' and its 'rgba()' alias, which (like the hex color notation) specify sRGB colors directly by their red/green/blue/alpha channels.
- 'hsl()' and its 'hsla()' alias, which specify sRGB colors by hue, saturation, and lightness using the HSL cylindrical coordinate model.
- 'hwb()', which specifies an sRGB color by hue, whiteness, and blackness using the HWB cylindrical coordinate model.
- 'lab()', which specifies a CIELAB color by CIE Lightness and its a- and b-axis hue coordinates (red/green-ness, and yellow/blue-ness) using the CIE LAB rectangular coordinate model.
- 'lch()' , which specifies a CIELAB color by CIE Lightness, Chroma, and hue using the CIE LCH cylindrical coordinate model
- 'oklab()', which specifies an Oklab color by Oklab Lightness and its a- and b-axis hue coordinates (red/green-ness, and yellow/blue-ness) using the Oklab rectangular coordinate model.
- 'oklch()' , which specifies an Oklab color by Oklab Lightness, Chroma, and hue using the Oklch cylindrical coordinate model.
- 'color()', which allows specifying colors in a variety of color spaces including sRGB, Linear-light sRGB, Display P3, A98 RGB, ProPhoto RGB, ITU-R BT.2020-2, and CIE XYZ.

For easy reference in other specifications, ***opaque black*** is defined as the color 'rgb(0 0 0 / 100%)'; ***transparent black*** is the same color, but fully transparent—i.e. 'rgb(0 0 0 / 0%)'.

§ 4.1. The <color> syntax

Colors in CSS are represented by the *'<color>'* type:

```
<color> = <absolute-color-base> | currentcolor | <system-color>

<absolute-color-base> = <hex-color> | <absolute-color-function> | <named
<absolute-color-function> = <rgb()> | <rgba()> |
                            <hsl()> | <hsla()> | <hwb()> |
                            <lab()> | <lch()> | <oklab()> | <oklch()> |
                            <color()>
```

The <hsl()>, <hsla()>, <hwb()>, <lch()>, and <oklch()> color functions are cylindrical polar color representations using a <hue> angle; the other color functions use rectangular orthogonal color representations.

4.1.1. Modern (Space-separated) Color Function Syntax

All of the <absolute-color-function> syntactic forms defined in this specification use the ***modern color syntax***, meaning:

- color components are separated by whitespace
- the optional alpha term is separated by a solidus ("/")
- minimum required precision when serializing is defined, and may be greater than 8 bits per component
- the 'none' value is allowed, to represent missing components
- components using <percentage> and <number> may be freely mixed

EXAMPLE 6

The following represents a saturated sRGB red that is 50% opaque:

```
rgb(100% 0% 0% / 50%)
```

✔ MDN

4.1.2. Legacy (Comma-separated) Color Function Syntax

For Web compatibility, the syntactic forms of 'rgb()', 'rgba()', 'hsl()', and 'hsla()', (those defined in earlier specifications) also support a ***legacy color syntax*** which has the following differences:

- color components are separated by commas (optionally preceded and/or followed by whitespace)
- non-opaque forms use a separate notation (for example 'hsla()' rather than 'hsl()') and the alpha term is separated by commas (optionally preceded and/or followed by whitespace)
- minimum required precision is lower, 8 bits per component
- the 'none' value is not allowed
- color components must be specified using either all-<percentage> or all-<number>, they can not be mixed.

EXAMPLE 7

The following represents a saturated sRGB red that is 50% opaque:

```
rgba(100%, 0%, 0%, 0.5)
```

For the color functions introduced in this or subsequent levels, where there is no Web compatibility issue, the legacy color syntax is invalid.

§ 4.2. Representing Transparency: the <alpha-value> syntax

```
<alpha-value> = <number> | <percentage>
```

Opacity in CSS is typically represented using the <alpha-value> syntax, for example in the 'opacity' property or as the alpha component in a color function. Represented as a <number>, the useful range of the value is '0' (representing full transparency) to '1' (representing full opacity). It can also be written as a <percentage>, which computes to the equivalent <number> ('0%' to '0', '100%' to '1'). Unless otherwise specified, an <alpha-value> component defaults to '100%' when omitted. Values outside the range [0,1] are not invalid, but are clamped to that range at parsed-value time.

§ 4.3. Representing Cylindrical-coordinate Hues: the <hue> syntax

Hue is represented as an angle of the color circle (the rainbow, twisted around into a circle, and with purple added between violet and red).

```
<hue> = <number> | <angle>
```

Because this value is so often given in degrees, the argument can also be given as a number, which is interpreted as a number of degrees and is the canonical unit.

This number is normalized to the range [0,360).

> NOTE: The angles and spacing corresponding to particular hues depend on the color space. For example, in HSL and HWB, which use the sRGB color space, sRGB green is 120 degrees. In LCH, sRGB green is 134.39 degrees, display-p3 green is 136.01 degrees, a98-rgb green is 145.97 degrees and prophoto-rgb green is 141.04 degrees (because these are all different shades of green).

<hue> components are the most common components to become powerless; any achromatic color will have a powerless hue component.

§ 4.4. “Missing” Color Components and the ‘none’ Keyword

✔ MDN

In certain cases, a color can have one or more ***missing color components***.

In this specification, this happens automatically due to hue-based interpolation for some colors (such as ‘white’); other specifications can define additional situations in which components are automatically missing.

It can also be specified explicitly, by providing the keyword ***‘none’*** for a component in a color function. All color functions (with the exception of those using the legacy color syntax) allow any of their components to be specified as ‘none’.

This should be done with care, and only when the particular effect of doing so is desired.

For handling of missing component in color interpolation, see § 12.2 Interpolating with Missing Components.

For all other purposes, a missing component behaves as a zero value, in the appropriate unit for that component: ‘0’, ‘0%’, or ‘0deg’. This includes rendering the color directly, converting it to another color space, performing computations on the color component values, etc.

If a color with a missing component is serialized or otherwise presented directly to an author, then for legacy color syntax it represents that component as a zero value; otherwise, it represents that component as being the ‘none’ keyword.

> EXAMPLE 8 ¶
>
> A missing hue is common when interpolating in cylindrical color spaces. For example, using the ‘color-mix()’ function specified in [CSS-COLOR-5] one could write ‘color-mix(in hsl, white 30%, green 70%)’. Since ‘white’ is an achromatic color, it has a missing hue when expressed in ‘hsl()’ (effectively ‘hsl(none 0% 100%)’, since *any* hue will produce the same color) which means that the color-mix function will treat it as having the same hue as ‘green’ (effectively ‘hsl(120deg 0% 100%)’), and then interpolate based on those components.
>
> The result will be a color that truly looks like a blend of green and white, rather than perhaps looking reddish (if ‘white’s hue was defaulted to ‘0deg’).

EXAMPLE 9

Explicitly specifying missing components can be useful to achieve an effect where you only *want* to interpolate certain components of a color.

For example, to animate a color to "grayscale", no matter what the color is, one can interpolate it with 'oklch(none 0 none)'. This will take the hue and lightness from the starting color, but animate its chroma down to 0, rendering it into an equal-lightness gray with a steady hue across the whole animation.

Doing this manually would require matching the hue and lightness of the starting color explicitly.

4.4.1. "Powerless" Color Components

Individual color syntaxes can specify that, in some cases, a given component of their syntax becomes a ***powerless color component***. This indicates that the value of the component doesn't affect the rendered color; any value you give it will result in the same color displayed in the screen.

For example, in 'hsl()', the hue component is powerless when the saturation component is '0%'; a '0%' saturation indicates a grayscale color, which has no hue at all, so '0deg' and '180deg', or any other angle, will give the exact same result.

If a powerless component is manually specified, it acts as normal; the fact that it's powerless has no effect. However, if a color is automatically produced by color space conversion, then any powerless components in the result must instead be set to missing, instead of whatever value was produced by the conversion process.

User agents *may* treat a component as powerless if the color is "sufficiently close" to the precise conditions specified. For example, a gray color converted into 'lch()' may, due to numerical errors, have an *extremely small* chroma rather than precisely '0%'; this can, at the user agent's discretion, still treat the hue component as powerless. It is intentionally unspecified exactly what "sufficiently close" means for this purpose.

5. sRGB Colors

CSS colors in the sRGB color space are represented by a triplet of values—red, green, and blue—identifying a point in the sRGB color space [SRGB]. This is an internationally-recognized, device-independent color space, and so is useful for specifying colors that will be displayed on a computer screen, but is also useful for specifying colors on other types of devices, like printers.

CSS also allows the use of non-sRGB color spaces, as described in § 10 Predefined Color Spaces.

CSS provides several methods of directly specifying an sRGB color: hex colors, 'rgb()'/'rgba()' color functions, 'hsl()'/'hsla()' color functions, 'hwb()' color function, named colors, and the 'transparent' keyword.

§ 5.1. The RGB functions: 'rgb()' and 'rgba()'

The 'rgb()' and 'rgba()' functions define an sRGB color by specifying the r, g and b (red, green, and blue) channels directly. Their syntax is:

```
rgb() = [ <legacy-rgb-syntax> | <modern-rgb-syntax> ]
rgba() = [ <legacy-rgba-syntax> | <modern-rgba-syntax> ]
<legacy-rgb-syntax> =   rgb( <percentage>#{3} , <alpha-value>? ) |
                        rgb( <number>#{3} , <alpha-value>? )
<legacy-rgba-syntax> = rgba( <percentage>#{3} , <alpha-value>? ) |
                        rgba( <number>#{3} , <alpha-value>? )
<modern-rgb-syntax> = rgb(
  [ <number> | <percentage> | none]{3}
  [ / [<alpha-value> | none] ]?  )
<modern-rgba-syntax> = rgba(
  [ <number> | <percentage> | none]{3}
  [ / [<alpha-value> | none] ]?  )
```

Percentages	Allowed for r, g and b
Percent reference range	For r, g and b: 0% = 0.0, 100% = 255.0 For alpha: 0% = 0.0, 100% = 1.0

▼ TESTS

rgb-001.html	(live test)	(source)
rgb-002.html	(live test)	(source)
rgb-003.html	(live test)	(source)
rgb-004.html	(live test)	(source)
rgb-005.html	(live test)	(source)
rgb-006.html	(live test)	(source)
rgb-007.html	(live test)	(source)
rgb-008.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
color-computed-rgb.html (77 tests)	(live test)	(source)
color-invalid-rgb.html (26 tests)	(live test)	(source)
color-valid-rgb.html (44 tests)	(live test)	(source)

The first three arguments specify the r, g and b (red, green, and blue) channels of the color, respectively. '0%' represents the minimum value for that color channel in the sRGB gamut, and '100%' represents the maximum value.

The percentage reference range of the color channels comes from the historical fact that many graphics engines stored the color channels internally as a single byte, which can hold integers between 0 and 255. Implementations should honor the precision of the channel as authored or calculated wherever possible. If this is not possible, the channel should be rounded towards +∞.

The final argument, the <alpha-value>, specifies the alpha of the color. If omitted, it defaults to '100%'.

▼ TESTS

background-color-rgb-001.html	(live test)	(source)
background-color-rgb-002.html	(live test)	(source)
background-color-rgb-003.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

Values outside these ranges are not invalid, but are clamped to the ranges defined here at parsed-value time.

For historical reasons, 'rgb()' and 'rgba()' also support a legacy color syntax.

▼ TESTS

rgba-001.html	(live test)	(source)
rgba-002.html	(live test)	(source)
rgba-003.html	(live test)	(source)
rgba-004.html	(live test)	(source)
rgba-005.html	(live test)	(source)
rgba-006.html	(live test)	(source)
rgba-007.html	(live test)	(source)
rgba-008.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

§ 5.2. The RGB Hexadecimal Notations: '#RRGGBB'

The CSS ***hex color notation*** allows an sRGB color to be specified by giving the channels as hexadecimal numbers, which is similar to how colors are often written directly in computer code. It's also shorter than writing the same color out in 'rgb()' notation.

The syntax of a *'<hex-color>'* is a <hash-token> token whose value consists of 3, 4, 6, or 8 hexadecimal digits. In other words, a hex color is written as a hash character, "#", followed by

some number of digits 0-9 or letters a-f (the case of the letters doesn't matter - '#00ff00' is identical to '#00FF00').

The number of hex digits given determines how to decode the hex notation into an RGB color:

6 digits

The first pair of digits, interpreted as a hexadecimal number, specifies the red channel of the color, where '00' represents the minimum value and 'ff' (255 in decimal) represents the maximum. The next pair of digits, interpreted in the same way, specifies the green channel, and the last pair specifies the blue. The alpha channel of the color is fully opaque.

> EXAMPLE 10 ¶
>
> In other words, '#00ff00' represents the same color as 'rgb(0 255 0)' (a lime green).

8 digits

The first 6 digits are interpreted identically to the 6-digit notation. The last pair of digits, interpreted as a hexadecimal number, specifies the alpha channel of the color, where '00' represents a fully transparent color and 'ff' represent a fully opaque color.

> EXAMPLE 11 ¶
>
> In other words, '#0000ffcc' represents the same color as 'rgb(0 0 100% / 80%)' (a slightly-transparent blue).

3 digits

This is a shorter variant of the 6-digit notation. The first digit, interpreted as a hexadecimal number, specifies the red channel of the color, where '0' represents the minimum value and 'f' represents the maximum. The next two digits represent the green and blue channels, respectively, in the same way. The alpha channel of the color is fully opaque.

> EXAMPLE 12 ¶
>
> This syntax is often explained by saying that it's identical to a 6-digit notation obtained by "duplicating" all of the digits. For example, the notation '#123' specifies the same color as the notation '#112233'. This method of specifying a color has lower "resolution" than the 6-digit notation; there are only 4096 possible colors expressible in the 3-digit hex syntax, as opposed to approximately 17 million in 6-digit hex syntax.

4 digits

This is a shorter variant of the 8-digit notation, "expanded" in the same way as the 3-digit notation is. The first digit, interpreted as a hexadecimal number, specifies the red channel of the color, where '0' represents the minimum value and 'f' represents the maximum. The next three digits represent the green, blue, and alpha channels, respectively.

▼ TESTS

hex-001.html	(live test)	(source)
hex-002.html	(live test)	(source)
hex-003.html	(live test)	(source)
hex-004.html	(live test)	(source)
border-bottom-color.xht	(live test)	(source)
border-left-color.xht	(live test)	(source)
border-right-color.xht	(live test)	(source)
border-top-color.xht	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
color-computed-hex-color.html (6 tests)	(live test)	(source)
color-invalid-hex-color.html (10 tests)	(live test)	(source)

§ 6. Color Keywords

In addition to the various numeric syntaxes for <color>s, CSS defines several sets of color keywords that can be used instead—each with their own advantages or use cases.

§ 6.1. Named Colors

CSS defines a large set of ***named colors***, so that common colors can be written and read more easily. A ***'<named-color>'*** is written as an <ident>, accepted anywhere a <color> is. As usual for CSS-defined <ident>s, all of these keywords are ASCII case-insensitive.

The names resolve to colors in sRGB.

16 of CSS's named colors come from the VGA palette originally, and were then adopted into HTML: aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, and yellow. Most of the rest come from one version of the X11 color system, used in Unix-derived systems to specify colors for the console, and were then adopted into SVG.

NOTE: these color names are standardized here, *not because they are good*, but because their use and implementation has been widespread for decades and the standard needs to reflect reality. Indeed, it is often hard to imagine what each name will look like (hence the list below); the names are not evenly distributed throughout the sRGB color volume, the names are not even internally consistent ('darkgray' is lighter than 'gray', while 'lightpink' is darker than 'pink'), and some names (such as 'indianred', which was originally named after a red pigment from India), have been found to be offensive. Thus, their use is *not encouraged*.

(Two special color values, ‘transparent’ and ‘currentcolor’, are specially defined in their own sections.)

The following table defines all of the opaque named colors, by giving equivalent numeric specifications in the other color syntaxes.

Named	Numeric	Color name	Hex rgb	Decimal
		‘aliceblue’	#F0F8FF	240 248 255
		‘antiquewhite’	#FAEBD7	250 235 215
		‘aqua’	#00FFFF	0 255 255
		‘aquamarine’	#7FFFD4	127 255 212
		‘azure’	#F0FFFF	240 255 255
		‘beige’	#F5F5DC	245 245 220
		‘bisque’	#FFE4C4	255 228 196
		‘black’	#000000	0 0 0
		‘blanchedalmond’	#FFEBCD	255 235 205
		‘blue’	#0000FF	0 0 255
		‘blueviolet’	#8A2BE2	138 43 226
		‘brown’	#A52A2A	165 42 42
		‘burlywood’	#DEB887	222 184 135
		‘cadetblue’	#5F9EA0	95 158 160
		‘chartreuse’	#7FFF00	127 255 0
		‘chocolate’	#D2691E	210 105 30
		‘coral’	#FF7F50	255 127 80
		‘cornflowerblue’	#6495ED	100 149 237
		‘cornsilk’	#FFF8DC	255 248 220
		‘crimson’	#DC143C	220 20 60
		‘cyan’	#00FFFF	0 255 255
		‘darkblue’	#00008B	0 0 139
		‘darkcyan’	#008B8B	0 139 139
		‘darkgoldenrod’	#B8860B	184 134 11
		‘darkgray’	#A9A9A9	169 169 169
		‘darkgreen’	#006400	0 100 0
		‘darkgrey’	#A9A9A9	169 169 169
		‘darkkhaki’	#BDB76B	189 183 107
		‘darkmagenta’	#8B008B	139 0 139
		‘darkolivegreen’	#556B2F	85 107 47
		‘darkorange’	#FF8C00	255 140 0
		‘darkorchid’	#9932CC	153 50 204

Named	Numeric	Color name	Hex rgb	Decimal
		'darkred'	#8B0000	139 0 0
		'darksalmon'	#E9967A	233 150 122
		'darkseagreen'	#8FBC8F	143 188 143
		'darkslateblue'	#483D8B	72 61 139
		'darkslategray'	#2F4F4F	47 79 79
		'darkslategrey'	#2F4F4F	47 79 79
		'darkturquoise'	#00CED1	0 206 209
		'darkviolet'	#9400D3	148 0 211
		'deeppink'	#FF1493	255 20 147
		'deepskyblue'	#00BFFF	0 191 255
		'dimgray'	#696969	105 105 105
		'dimgrey'	#696969	105 105 105
		'dodgerblue'	#1E90FF	30 144 255
		'firebrick'	#B22222	178 34 34
		'floralwhite'	#FFFAF0	255 250 240
		'forestgreen'	#228B22	34 139 34
		'fuchsia'	#FF00FF	255 0 255
		'gainsboro'	#DCDCDC	220 220 220
		'ghostwhite'	#F8F8FF	248 248 255
		'gold'	#FFD700	255 215 0
		'goldenrod'	#DAA520	218 165 32
		'gray'	#808080	128 128 128
		'green'	#008000	0 128 0
		'greenyellow'	#ADFF2F	173 255 47
		'grey'	#808080	128 128 128
		'honeydew'	#F0FFF0	240 255 240
		'hotpink'	#FF69B4	255 105 180
		'indianred'	#CD5C5C	205 92 92
		'indigo'	#4B0082	75 0 130
		'ivory'	#FFFFF0	255 255 240
		'khaki'	#F0E68C	240 230 140
		'lavender'	#E6E6FA	230 230 250
		'lavenderblush'	#FFF0F5	255 240 245
		'lawngreen'	#7CFC00	124 252 0
		'lemonchiffon'	#FFFACD	255 250 205
		'lightblue'	#ADD8E6	173 216 230
		'lightcoral'	#F08080	240 128 128

Named	Numeric	Color name	Hex rgb	Decimal	
		'lightcyan'	#E0FFFF	224 255 255	
		'lightgoldenrodyellow'	#FAFAD2	250 250 210	
		'lightgray'	#D3D3D3	211 211 211	
		'lightgreen'	#90EE90	144 238 144	
		'lightgrey'	#D3D3D3	211 211 211	
		'lightpink'	#FFB6C1	255 182 193	
		'lightsalmon'	#FFA07A	255 160 122	
		'lightseagreen'	#20B2AA	32 178 170	
		'lightskyblue'	#87CEFA	135 206 250	
		'lightslategray'	#778899	119 136 153	
		'lightslategrey'	#778899	119 136 153	
		'lightsteelblue'	#B0C4DE	176 196 222	
		'lightyellow'	#FFFFE0	255 255 224	
		'lime'	#00FF00	0 255 0	
		'limegreen'	#32CD32	50 205 50	
		'linen'	#FAF0E6	250 240 230	
		'magenta'	#FF00FF	255 0 255	
		'maroon'	#800000	128 0 0	
		'mediumaquamarine'	#66CDAA	102 205 170	✔ MDN
		'mediumblue'	#0000CD	0 0 205	
		'mediumorchid'	#BA55D3	186 85 211	
		'mediumpurple'	#9370DB	147 112 219	
		'mediumseagreen'	#3CB371	60 179 113	
		'mediumslateblue'	#7B68EE	123 104 238	
		'mediumspringgreen'	#00FA9A	0 250 154	
		'mediumturquoise'	#48D1CC	72 209 204	
		'mediumvioletred'	#C71585	199 21 133	
		'midnightblue'	#191970	25 25 112	
		'mintcream'	#F5FFFA	245 255 250	✔ MDN
		'mistyrose'	#FFE4E1	255 228 225	
		'moccasin'	#FFE4B5	255 228 181	
		'navajowhite'	#FFDEAD	255 222 173	
		'navy'	#000080	0 0 128	
		'oldlace'	#FDF5E6	253 245 230	
		'olive'	#808000	128 128 0	
		'olivedrab'	#6B8E23	107 142 35	
		'orange'	#FFA500	255 165 0	

Named	Numeric	Color name	Hex rgb	Decimal
		‘orangered’	#FF4500	255 69 0
		‘orchid’	#DA70D6	218 112 214
		‘palegoldenrod’	#EEE8AA	238 232 170
		‘palegreen’	#98FB98	152 251 152
		‘paleturquoise’	#AFEEEE	175 238 238
		‘palevioletred’	#DB7093	219 112 147
		‘papayawhip’	#FFEFD5	255 239 213
		‘peachpuff’	#FFDAB9	255 218 185
		‘peru’	#CD853F	205 133 63
		‘pink’	#FFC0CB	255 192 203
		‘plum’	#DDA0DD	221 160 221
		‘powderblue’	#B0E0E6	176 224 230
		‘purple’	#800080	128 0 128
		‘rebeccapurple’	#663399	102 51 153
		‘red’	#FF0000	255 0 0
		‘rosybrown’	#BC8F8F	188 143 143
		‘royalblue’	#4169E1	65 105 225
		‘saddlebrown’	#8B4513	139 69 19
		‘salmon’	#FA8072	250 128 114
		‘sandybrown’	#F4A460	244 164 96
		‘seagreen’	#2E8B57	46 139 87
		‘seashell’	#FFF5EE	255 245 238
		‘sienna’	#A0522D	160 82 45
		‘silver’	#C0C0C0	192 192 192
		‘skyblue’	#87CEEB	135 206 235
		‘slateblue’	#6A5ACD	106 90 205
		‘slategray’	#708090	112 128 144
		‘slategrey’	#708090	112 128 144
		‘snow’	#FFFAFA	255 250 250
		‘springgreen’	#00FF7F	0 255 127
		‘steelblue’	#4682B4	70 130 180
		‘tan’	#D2B48C	210 180 140
		‘teal’	#008080	0 128 128
		‘thistle’	#D8BFD8	216 191 216
		‘tomato’	#FF6347	255 99 71
		‘turquoise’	#40E0D0	64 224 208
		‘violet’	#EE82EE	238 130 238

Named	Numeric	Color name	Hex rgb	Decimal
		'wheat'	#F5DEB3	245 222 179
		'white'	#FFFFFF	255 255 255
		'whitesmoke'	#F5F5F5	245 245 245
		'yellow'	#FFFF00	255 255 0
		'yellowgreen'	#9ACD32	154 205 50

NOTE: this list of colors and their definitions is a superset of the list of named colors defined by SVG 1.1.

For historical reasons, this is also referred to as the X11 color set.

NOTE: The history of the X11 color system is interesting, and was excellently summarized by Alex Sexton in their talk "Peachpuffs and Lemonchiffons".

▼ TESTS

named-001.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
color-computed-named-color.html (455 tests)	(live test)	(source)
color-invalid-named-color.html (184 tests)	(live test)	(source)

§ 6.2. System Colors

In general, the <system-color> keywords reflect *default* color choices made by the user, the browser, or the OS. They are typically used in the browser default stylesheet, for this reason.

To maintain legibility, the <system-color> keywords also respond to light mode or dark mode changes.

EXAMPLE 13

For example, traditional blue link text is legible on a white background (WCAG contrast 8.59:1, AAA pass) but would not be legible on a black background (WCAG contrast 2.44:1, AA fail). Instead, a lighter blue such as #81D9FE would be used in dark mode (WCAG contrast 13.28:1, AAA pass).

Legible link text

Illegible link text

Legible link text

However, in forced colors mode, most colors on the page are forced into a restricted, user-chosen palette. The *'<system-color>'* keywords expose these user-chosen colors so that the rest of the page can integrate with this restricted palette.

When the forced-colors media feature is 'active', authors *should* use the <system-color> keywords as color values in properties other than those listed in *CSS Color Adjustment 1* § 3.1 Properties Affected by Forced Colors Mode, to ensure legibility and consistency across the page and avoid an uncoordinated mishmash of user-forced and page-chosen colors.

▼ TESTS

system-color-consistency.html (27 tests) (live test) (source)

color-valid-system-color.html (19 tests) (live test) (source)

When the values of <system-color> keywords come from the browser, (as opposed to being OS defaults or user choices) the browser should ensure that matching foreground/background pairs have a minimum of WCAG AA contrast. However, user preferences (for higher or lower contrast), whether set as a browser preference, a user stylesheet, or by altering the OS defaults, must take precedence over this requirement.

Authors *may* also use these keywords at any time, but *should* be careful to use the colors in matching background-foreground pairs to ensure appropriate contrast, as any particular contrast relationship across non-matching pairs (e.g. 'Canvas' and 'ButtonText') is not guaranteed.

The <system-color> keywords are defined as follows:

'Canvas'

Background of application content or documents.

'CanvasText'

Text in application content or documents.

'LinkText'

Text in non-active, non-visited links. For light backgrounds, traditionally blue.

'VisitedText'

Text in visited links. For light backgrounds, traditionally purple.

'ActiveText'

Text in active links. For light backgrounds, traditionally red.

'ButtonFace'

The face background color for push buttons.

'ButtonText'

Text on push buttons.

'ButtonBorder'

The base border color for push buttons.

'Field'

Background of input fields.

'FieldText'

Text in input fields.

'Highlight'

Background of selected text, for example from ::selection.

'HighlightText'

Text of selected text.

'SelectedItem'

Background of selected items, for example a selected checkbox.

'SelectedItemText'

Text of selected items.

'Mark'

Background of text that has been specially marked (such as by the HTML `<mark>` element).

'MarkText'

Text that has been specially marked (such as by the HTML `<mark>` element).

'GrayText'

Disabled text. (Often, but not necessarily, gray.)

'AccentColor'

Background of accented user interface controls.

'AccentColorText'

Text of accented user interface controls.

▼ TESTS

color-valid.html (16 tests) (live test) (source)

system-color-compute.html (27 tests) (live test) (source)
system-color-hightlights-vs-getSelection-001.html (live test) (source)
system-color-hightlights-vs-getSelection-002.html (live test) (source)

NOTE: As with all other keywords, these names are ASCII case-insensitive. They are shown here with mixed capitalization for legibility.

For systems that do not have a particular system UI concept, the specified value should be mapped to the most closely related system color value that exists. The following ***system color pairings*** are expected to form legible background-foreground colors:

- 'Canvas' background with 'CanvasText', 'LinkText', 'VisitedText', 'ActiveText' foreground.
- 'Canvas' background with a 'ButtonBorder' border and adjacent color 'Canvas'
- 'ButtonFace' background with 'ButtonText' foreground.
- 'Field' background with 'FieldText' foreground.
- 'Mark' background with 'MarkText' foreground
- 'ButtonFace' or 'Field' background with a 'ButtonBorder' border and adjacent color 'Canvas'"
- 'Highlight' background with 'HighlightText' foreground.
- 'SelectedItem' background with 'SelectedItemText' foreground.
- 'AccentColor' background with 'AccentColorText' foreground.

Additionally, 'GrayText' is expected to be readable, though possibly at a lower contrast rating, over any of the backgrounds.

EXAMPLE 14

For example, the system color combinations in the browser you are currently using:

Canvas with CanvasText: CanvasText

Canvas with LinkText: LinkText

Canvas with VisitedText: VisitedText

Canvas with ActiveText: ActiveText

Canvas with GrayText: GrayText

Canvas with ButtonBorder and adjacent Canvas: CanvasText Adjacent

ButtonFace with ButtonText: ButtonText

ButtonFace with ButtonText and ButtonBorder: ButtonText

ButtonFace with GrayText: GrayText

Field with FieldText: FieldText

Field with GrayText: GrayText

Mark with MarkText: MarkText

Mark with GrayText: GrayText

Highlight with HighlightText: HighlightText

Highlight with GrayText: GrayText

SelectedItem with SelectedItemText: SelectedItemText

AccentColor with AccentColorText: AccentColorText

AccentColor with GrayText: GrayText

Earlier versions of CSS defined additional <system-color>s, which have since been deprecated. These are documented in Appendix A: Deprecated CSS System Colors.

NOTE: The <system-color>s incur some privacy and security risk, as detailed in § 20 Privacy Considerations and § 19 Security Considerations.

§ 6.3. The 'transparent' keyword

The keyword ***'transparent'*** specifies a transparent black. It is a type of <named-color>.

▼ TESTS

color-computed.html (16 tests)	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
t423-transparent-1-a.xht	(live test)	(source)
t423-transparent-2-a.xht	(live test)	(source)

§ 6.4. The 'currentcolor' keyword

The keyword ***'currentcolor'*** represents value of the 'color' property on the same element. Unlike <named-color>s, it is *not* restricted to sRGB; the value can be any <color>. Its used values is determined by resolving color values.

▼ TESTS

border-color-currentcolor.html	(live test)	(source)
color-mix-currentcolor-nested-for-color-property.html	(live test)	(source)
currentcolor-001.html	(live test)	(source)
currentcolor-002.html	(live test)	(source)
currentcolor-003.html	(live test)	(source)
currentcolor-004.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

EXAMPLE 15 ¶

Here's a simple example showing how to use the 'currentcolor' keyword:

```
.foo {
  color: red;
  background-color: currentcolor;
}
```

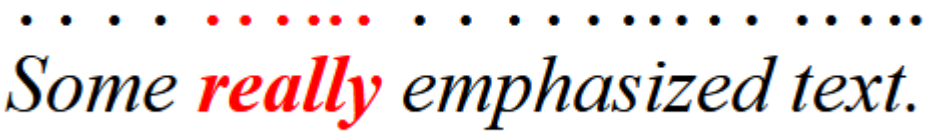
This is equivalent to writing:

```
.foo {
  color: red;
  background-color: red;
}
```

EXAMPLE 16

For example, the 'text-emphasis-color' property [CSS3-TEXT-DECOR], whose initial value is 'currentcolor', by default matches the text color even as the 'color' property changes across elements.

```
<p><em>Some <strong>really</strong> emphasized text.</em>
<style>
p { color: black; }
em { text-emphasis: dot; }
strong { color: red; }
</style>
```



In the above example, the emphasis marks are black over the text "Some" and "emphasized text", but red over the text "really".

NOTE: Multi-word keywords in CSS usually separate their component words with hyphens. 'currentcolor' doesn't, because (deep breath) it was originally introduced in SVG as a property value, "current-color" with the usual CSS spelling. It (along with all other properties and their values) then became presentation attributes and attribute values, as well as properties, to make generation with XSLT easier. Then all of the presentation attributes were changed from hyphenated to camelCase, because the DOM had an issue with hyphen meaning "minus". But then, they didn't follow CSS conventions anymore so all the properties and property values that were *already* part of CSS were changed back to hyphenated! 'currentcolor' was not a part of CSS at that time, so remained camelCased. Only later did CSS pick it up, at which point the capitalization stopped mattering, as CSS keywords are ASCII case-insensitive.

§ 7. HSL Colors: 'hsl()' and 'hsla()' functions

The RGB system for specifying colors, while convenient for machines and graphic libraries, is often regarded as very difficult for humans to gain an intuitive grasp on. It's not easy to tell, for example, how to alter an RGB color to produce a lighter variant of the same hue.

There are several other color schemes possible. One such is the HSL [HSL] color scheme, which is more intuitive to use, but still maps easily back to RGB colors.

'HSL' colors are specified as a triplet of hue, saturation, and lightness. The syntax of the 'hsl()' and 'hsla()' functions is:

```
hsl() = [ <legacy-hsl-syntax> | <modern-hsl-syntax> ]
hsla() = [ <legacy-hsla-syntax> | <modern-hsla-syntax> ]
<modern-hsl-syntax> = hsl(
    [<hue> | none]
    [<percentage> | <number> | none]
    [<percentage> | <number> | none]
    [ / [<alpha-value> | none] ]? )
<modern-hsla-syntax> = hsla(
    [<hue> | none]
    [<percentage> | <number> | none]
    [<percentage> | <number> | none]
    [ / [<alpha-value> | none] ]? )
<legacy-hsl-syntax> = hsl( <hue>, <percentage>, <percentage>, <alpha-val
<legacy-hsla-syntax> = hsla( <hue>, <percentage>, <percentage>, <alpha-v
```

✔ MDN

Percentages	Allowed for S and L
Percent reference range	for S and L: 0% = 0.0, 100% = 100.0

▼ TESTS

hsl-001.html	(live test)	(source)
hsl-002.html	(live test)	(source)
hsl-003.html	(live test)	(source)
hsl-004.html	(live test)	(source)
hsl-005.html	(live test)	(source)
hsl-006.html	(live test)	(source)
hsl-007.html	(live test)	(source)
hsl-008.html	(live test)	(source)
background-color-hsl-001.html	(live test)	(source)
background-color-hsl-002.html	(live test)	(source)
background-color-hsl-003.html	(live test)	(source)
background-color-hsl-004.html	(live test)	(source)
color-computed-hsl.html (3735 tests)	(live test)	(source)
color-invalid-hsl.html (25 tests)	(live test)	(source)
color-valid-hsl.html (26 tests)	(live test)	(source)

The first argument specifies the hue angle.

In HSL (and HWB) the angle '0deg' represents sRGB primary red (as does '360deg', '720deg', etc.), and the rest of the hues are spread around the circle, so '120deg' represents sRGB primary green,

'240deg' represents sRGB primary blue, etc.

The next two arguments are the saturation and lightness, respectively. For saturation, '100%' is a fully-saturated, bright color, and '0%' is a fully-unsaturated gray. For lightness, '50%' represents the "normal" color, while '100%' is white and '0%' is black.

The final argument specifies the alpha channel of the color. It's interpreted identically to the fourth argument of the 'rgb()' function. If omitted, it defaults to '100%'.

HSL colors resolve to sRGB.

If the saturation of an HSL color is '0%', then the hue component is powerless. If the lightness of an HSL color is '0%' or '100%', the hue component is powerless and the saturation is '0%'.

EXAMPLE 17 ¶

For example, an ordinary red, the same color you would see from the keyword 'red' or the hex notation '#f00', is represented in HSL as 'hsl(0deg 100% 50%)'.

An advantage of HSL over RGB is that it is more intuitive: people can guess at the colors they want, and then tweak.

EXAMPLE 18 ¶

For example, the following colors can all be generated off of the basic "green" hue, just by varying the other two arguments:

```
hsl(120deg 100% 50%) lime green
hsl(120deg 100% 25%) dark green
hsl(120deg 100% 75%) light green
hsl(120deg 75% 85%)  pastel green
```

A disadvantage of HSL over LCH is that hue manipulation changes the visual lightness, and that hues are not evenly spaced apart.

It is thus easier in HSL to create sets of matching colors (by keeping the hue the same and varying the saturation and lightness), compared to manipulating the sRGB component values; however, because the lightness is simply the mean of the gamma-corrected red, green and blue components it does not correspond to the visual perception of lightness across hues.

EXAMPLE 19

For example, 'blue' is represented in HSL as 'hsl(240deg 100% 50%)' while 'yellow' is 'hsl(60deg 100% 50%)'. Both have an HSL Lightness of 50%, but clearly the yellow looks much lighter than the blue.

In LCH, sRGB blue is 'lch(29.6% 131.2 301.3)' while sRGB yellow is 'lch(97.6% 94.7 99.6)'. The LCH Lightnesses of 29.6% and 97.6% clearly reflect the visual lightnesses of the two colors.

The hue angle in HSL is not perceptually uniform; colors appear bunched up in some areas and widely spaced in others.

EXAMPLE 20

For example, the pair of hues 'hsl(220deg 100% 50%)' and 'hsl(250deg 100% 50%)' have an HSL hue difference of 250-220 = 30deg and look fairly similar, while another pair of colors 'hsl(50deg 100% 50%)' and 'hsl(80deg 100% 50%)', which also have a hue difference of 80-50 = 30deg, look very different.

In LCH, the same pair of colors 'lch(42.1% 97.4 290.6)' and 'lch(30.8% 129.7 302.1)' have a hue difference of 302.1-290.6 = 11.5deg while the second pair 'lch(86.8% 86.2 87.3)' and 'lch(92.0% 98.8 119.1)' have a hue difference of 119.1-87.3 = 31.8deg, correctly reflecting the visual separation of hues.

For historical reasons, 'hsl()' and 'hsla()' also support a legacy color syntax.

▼ TESTS

hsla-001.html	(live test)	(source)
hsla-002.html	(live test)	(source)
hsla-003.html	(live test)	(source)
hsla-004.html	(live test)	(source)
hsla-005.html	(live test)	(source)
hsla-006.html	(live test)	(source)
hsla-007.html	(live test)	(source)
hsla-008.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

§ 7.1. Converting HSL Colors to sRGB

Converting an HSL color to sRGB is straightforward mathematically. Here's a sample implementation of the conversion algorithm in JavaScript. It returns an array of three numbers representing the red, green, and blue channels of the colors, normalized to the range [0, 1].

```
/**
 * @param {number} hue - Hue as degrees 0..360
 * @param {number} sat - Saturation in reference range [0,100]
 * @param {number} light - Lightness in reference range [0,100]
 * @return {number[]} Array of RGB components 0..1
 */
function hslToRgb(hue, sat, light) {
    hue = hue % 360;

    if (hue < 0) {
        hue += 360;
    }

    sat /= 100;
    light /= 100;

    function f(n) {
        let k = (n + hue/30) % 12;
        let a = sat * Math.min(light, 1 - light);
        return light - a * Math.max(-1, Math.min(k - 3, 9 - k, 1));
    }

    return [f(0), f(8), f(4)];
}
```

§ 7.2. Converting sRGB Colors to HSL

Conversion in the reverse direction proceeds similarly.

```
/**
 * @param {number} red - Red component 0..1
 * @param {number} green - Green component 0..1
 * @param {number} blue - Blue component 0..1
 * @return {number[]} Array of HSL values: Hue as degrees 0..360, Satura
 */
function rgbToHsl (red, green, blue) {
    let max = Math.max(red, green, blue);
```

```
    let min = Math.min(red, green, blue);
    let [hue, sat, light] = [NaN, 0, (min + max)/2];
    let d = max - min;

    if (d !== 0) {
        sat = (light === 0 || light === 1)
            ? 0
            : (max - light) / Math.min(light, 1 - light);

        switch (max) {
            case red:   hue = (green - blue) / d + (green < blue ? 6 : 0
            case green: hue = (blue - red) / d + 2; break;
            case blue:  hue = (red - green) / d + 4;
        }

        hue = hue * 60;
    }

    return [hue, sat * 100, light * 100];
}
```

§ 7.3. Examples of HSL Colors

The tables below illustrate a wide range of possible HSL colors. Each table represents one hue, selected at 30° intervals, to illustrate the common "core" hues: red, yellow, green, cyan, blue, magenta, and the six intermediary colors between these.

In each table, the X axis represents the saturation while the Y axis represents the lightness.

0° Reds					
	100%	80%	60%	40%	20%	0%
100%						
90%						
80%						
70%						
60%						
50%						
40%						
30%						
20%						
10%						

30° Reds-Yellows (=Oranges)					
	100%	80%	60%	40%	20%	0%
100%						
90%						
80%						
70%						
60%						
50%						
40%						
30%						
20%						
10%						

0° Reds

100% 80% 60% 40% 20% 0%

0%

30° Reds-Yellows (=Oranges)

100% 80% 60% 40% 20% 0%

0%

60° Yellows

100% 80% 60% 40% 20% 0%

100%

90%

80%

70%

60%

50%

40%

30%

20%

10%

0%

90° Yellow-Greens

100% 80% 60% 40% 20% 0%

100%

90%

80%

70%

60%

50%

40%

30%

20%

10%

0%

120° Greens

100% 80% 60% 40% 20% 0%

100%

90%

80%

70%

60%

50%

40%

30%

20%

10%

0%

150° Green-Cyans

100% 80% 60% 40% 20% 0%

100%

90%

80%

70%

60%

50%

40%

30%

20%

10%

0%

180° Cyans

100% 80% 60% 40% 20% 0%

100%

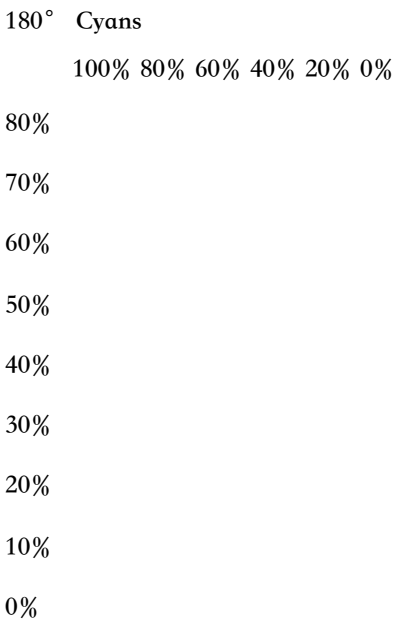
90%

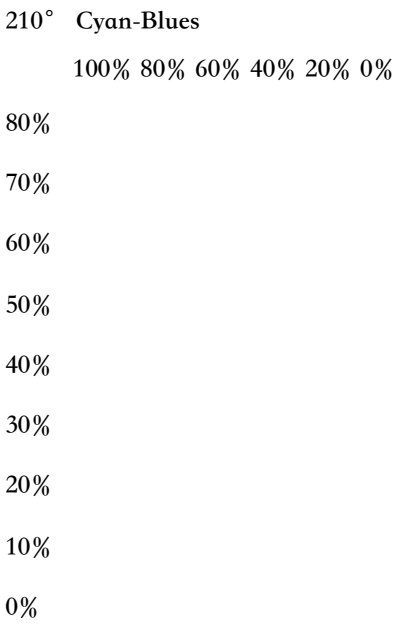
210° Cyan-Blues

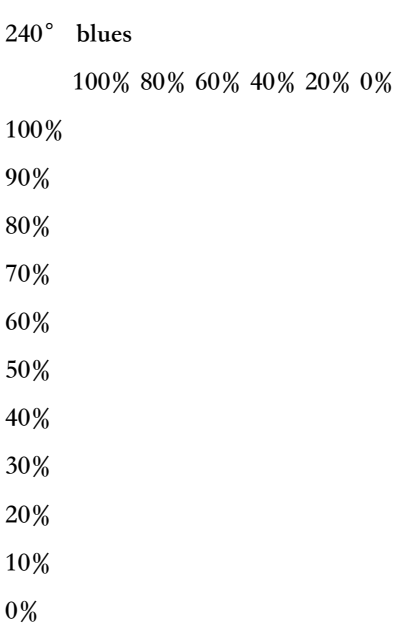
100% 80% 60% 40% 20% 0%

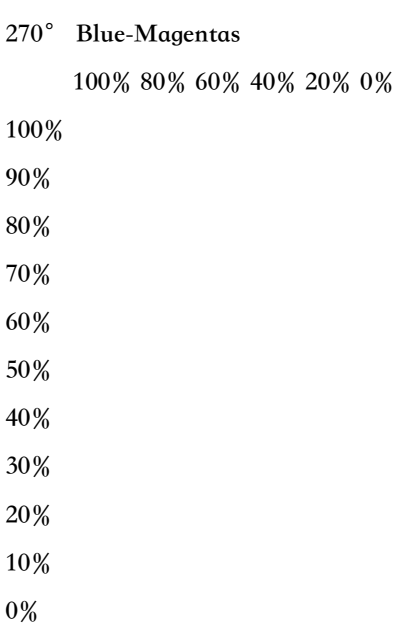
100%

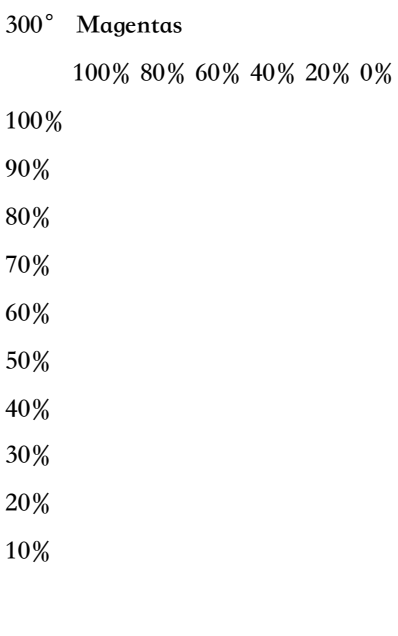
90%

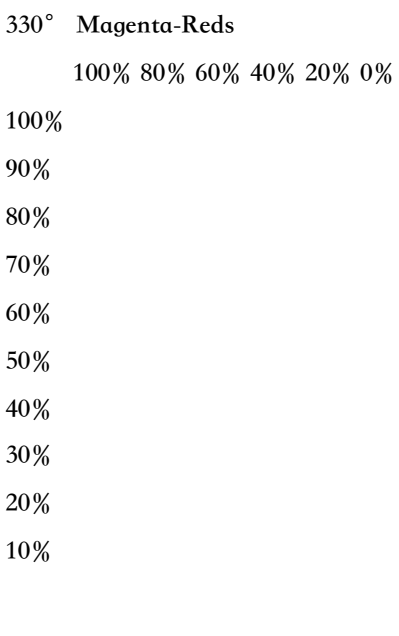

180° Cyans
100% 80% 60% 40% 20% 0%
80%
70%
60%
50%
40%
30%
20%
10%
0%


210° Cyan-Blues
100% 80% 60% 40% 20% 0%
80%
70%
60%
50%
40%
30%
20%
10%
0%


240° blues
100% 80% 60% 40% 20% 0%
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%


270° Blue-Magentas
100% 80% 60% 40% 20% 0%
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%


300° Magentas
100% 80% 60% 40% 20% 0%
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%


330° Magenta-Reds
100% 80% 60% 40% 20% 0%
100%
90%
80%
70%
60%
50%
40%
30%
20%
10%

300° Magentas
100% 80% 60% 40% 20% 0%
0%

330° Magenta-Reds
100% 80% 60% 40% 20% 0%
0%

§ 8. HWB Colors: 'hwb()' function

'HWB' (short for Hue-Whiteness-Blackness) [HWB] is another method of specifying sRGB colors, similar to 'HSL'', but often even easier for humans to work with. It describes colors with a starting hue, then a degree of whiteness and blackness to mix into that base hue.

Many color-pickers are based on the HWB color system, due to its intuitiveness.

HWB colors resolve to sRGB.

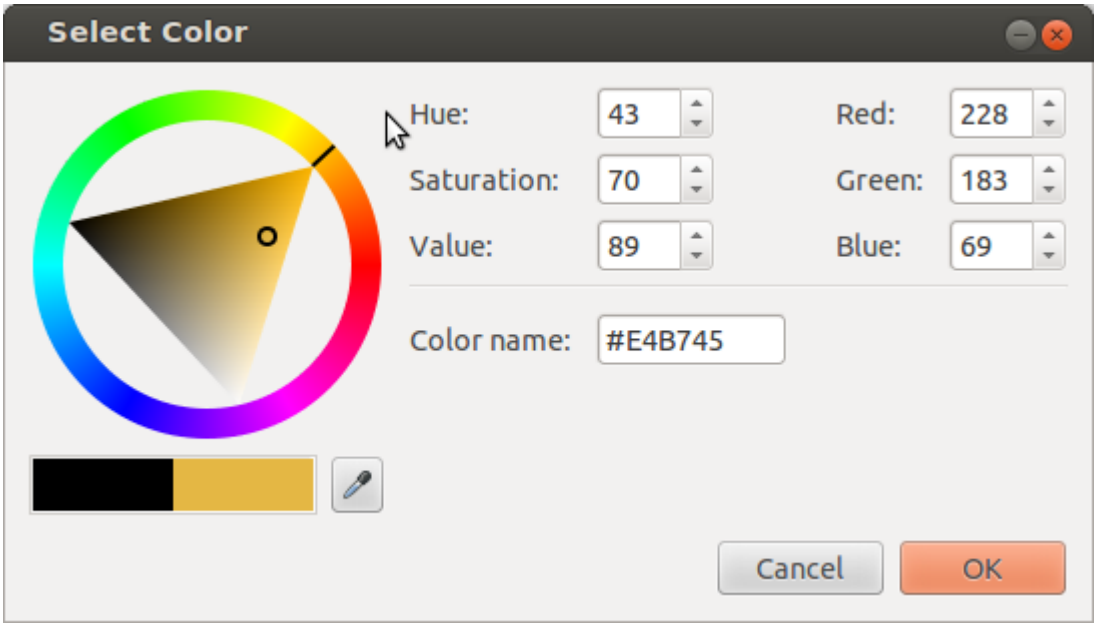


Figure 3 *This is a screenshot of Chrome's color picker, shown when a user activates an* `<input type="color">`*. The outer wheel is used to select the hue, then the relative amounts of white and black are selected by clicking on the inner triangle.*

The syntax of the 'hwb()' function is:

```
hwb() = hwb(
  [<hue> | none]
  [<percentage> | <number> | none]
  [<percentage> | <number> | none]
  [ / [<alpha-value> | none] ]? )
```

Percentages	Allowed for W and B
Percent reference range	for W and B: 0% = 0.0, 100% = 100.0

The first argument specifies the hue, and is interpreted identically to 'hsl()'.

The second argument specifies the amount of white to mix in, as a percentage from '0%' (no whiteness) to '100%' (full whiteness). Similarly, the third argument specifies the amount of black to mix in, also from '0%' (no blackness) to '100%' (full blackness).

EXAMPLE 21 ¶

For example, hwb(150 20% 10%) is the same color as hsl(150 77.78% 55%) and rgb(20% 90% 55%).

Values outside of these ranges are not invalid; hue angles outside the range [0,360) will be normalized to that range and values of white or black greater than 100% will produce achromatic colors as described below.

The resulting color can be thought of conceptually as a mixture of paint in the chosen hue, white paint, and black paint, with the relative amounts of each determined by the percentages.

If the sum white+black is greater than or equal to '100%', it defines an achromatic color, i.e. a shade of gray; when converted to sRGB the R, G and B values are identical and have the value white / (white + black).

EXAMPLE 22 ¶

For example, in the color hwb(45 40% 80%) white and black adds to 120, so this is an achromatic color whose R, G and B components are 40 / 40 + 80 = 0.33 rgb(33.33% 33.33% 33.33%).

Achromatic HWB colors no longer contain any hint of the chosen hue. In this case, the hue component is powerless.

The fourth argument specifies the alpha channel of the color. It's interpreted identically to the fourth argument of the 'rgb()' function. If omitted, it defaults to '100%'.

There is no Web compatibility issue with 'hwb', which is new in this level of the specification, and so 'hwb()' does *not* support a legacy color syntax that separates all of its arguments with commas. Using commas inside 'hwb()' is an error.

▼ TESTS

hwb-001.html	(live test) (source)
hwb-002.html	(live test) (source)
hwb-003.html	(live test) (source)

hwb-004.html	(live test)	(source)
hwb-005.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
color-computed-hwb.html (50 tests)	(live test)	(source)
color-invalid-hwb.html (12 tests)	(live test)	(source)
color-valid-hwb.html (23 tests)	(live test)	(source)

§ 8.1. Converting HWB Colors to sRGB

Converting an HWB color to sRGB is straightforward, and related to how one converts HSL to RGB. The following Javascript implementation of the algorithm first normalizes the white and black components, so their sum is no larger than 100%.

```
/**
 * @param {number} hue -  Hue as degrees 0..360
 * @param {number} white -  Whiteness in reference range [0,100]
 * @param {number} black -  Blackness in reference range [0,100]
 * @return {number[]} Array of RGB components 0..1
 */
function hwbToRgb(hue, white, black) {
    white /= 100;
    black /= 100;
    if (white + black >= 1) {
        let gray = white / (white + black);
        return [gray, gray, gray];
    }
    let rgb = hslToRgb(hue, 100, 50);
    for (let i = 0; i < 3; i++) {
        rgb[i] *= (1 - white - black);
        rgb[i] += white;
    }
    return rgb;
}
```

§ 8.2. Converting sRGB Colors to HWB

Conversion in the reverse direction proceeds similarly.

```
/**
 * @param {number} red - Red component 0..1
 * @param {number} green - Green component 0..1
```

```
 * @param {number} blue - Blue component 0..1
 * @return {number[]} Array of HWB values: Hue as degrees 0..360, Whiten
 */
function rgbToHwb(red, green, blue) {
    var hsl = rgbToHsl(red, green, blue);
    var white = Math.min(red, green, blue);
    var black = 1 - Math.max(red, green, blue);
    return([hsl[0], white*100, black*100]);
}
```

§ 8.3. Examples of HWB Colors

0° Reds

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						
80%						
100%						

30° Red-Yellows (Oranges)

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						
80%						
100%						

60° Yellows

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						
80%						
100%						

90° Yellow-Greens

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						
80%						
100%						

120° Greens

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						

150° Green-Cyans

W\B	0%	20%	40%	60%	80%	100%
0%						
20%						
40%						
60%						

80%
100%

80%
100%

180° Cyans

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

210° Cyan-Blues

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

240° Blues

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

270° Blue-Magentas

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

300° Magentas

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

330° Magenta-Reds

W\B 0% 20% 40% 60% 80% 100%
0%
20%
40%
60%
80%
100%

§ 9. Device-independent Colors: CIE Lab and LCH, Oklab and Oklch

§ 9.1. CIE Lab and LCH

This section is not normative.

Physical measurements of a color are typically expressed in the CIE L*a*b* [CIELAB] color space, created in 1976 by the CIE and commonly referred to simply as Lab. Color conversions from one device to another may also use Lab as an intermediate step. Derived from human vision experiments, Lab represents the entire range of color that humans can see.

Lab is a rectangular coordinate system with a central Lightness (L) axis. This value is usually written as a unitless number; for compatibility with the rest of CSS, it may also be written as a percentage. 100% means an L value of 100, not 1.0. L=0% or 0 is deep black (no light at all) while L=100% or 100 is a diffuse white.

Usefully, L=50% or 50 is mid gray, by design, and equal increments in L are evenly spaced visually: the Lab color space is intended to be *perceptually uniform*.

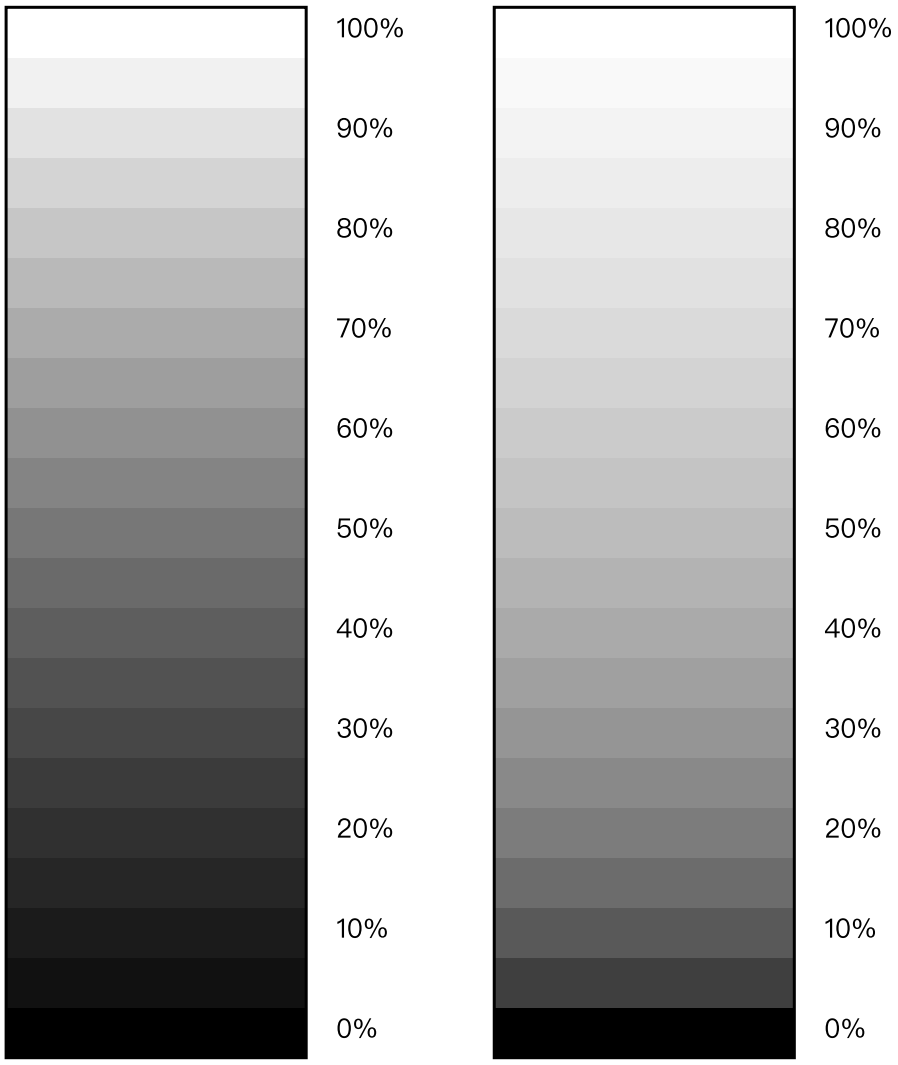


Figure 4 *This figure shows, to the left, the Lightness axis of the CIE Lab color space. Twenty-one neutral swatches are shown (L=0%, L=5%, to L=100%). The steps are equally spaced, visually. To the right, the same number of steps in luminance are equally spaced in light energy but* ***not*** *equally spaced visually.*

The a and b axes convey hue; positive values along the a axis are a purplish red while negative values are the complementary color, a green. Similarly, positive values along the b axis are yellow and negative are the complementary blue/violet. Desaturated colors have small values of a and b and are close to the L axis; saturated colors lie far from the L axis.

The illuminant is D50 white, a standardized daylight spectrum with a color temperature of 5000K, as reflected by a perfect diffuse reflector; it approximates the color of sunlight on a sunny day. D50 is also the whitepoint used for the profile connection space in ICC color interconversion, the whitepoint used in image editors which offer Lab editing, and the value used by physical measurement devices such as spectrophotometers and spectroradiometers, when they report measured colors in Lab.

Conversion from colors specified using other white points is called a ***chromatic adaptation transform***, which models the changes in the human visual system as we adapt to a new lighting condition. The Bradford algorithm [Bradford-CAT] is the industry standard chromatic adaptation transform, and is easy to calculate as it is a simple matrix multiplication.

CIE LCH has the same L axis as Lab, but uses polar coordinates C (chroma) and H (hue), making it a polar, cylindrical coordinate system. C is the geometric distance from the L axis and H is the angle from the positive a axis, towards the positive b axis.

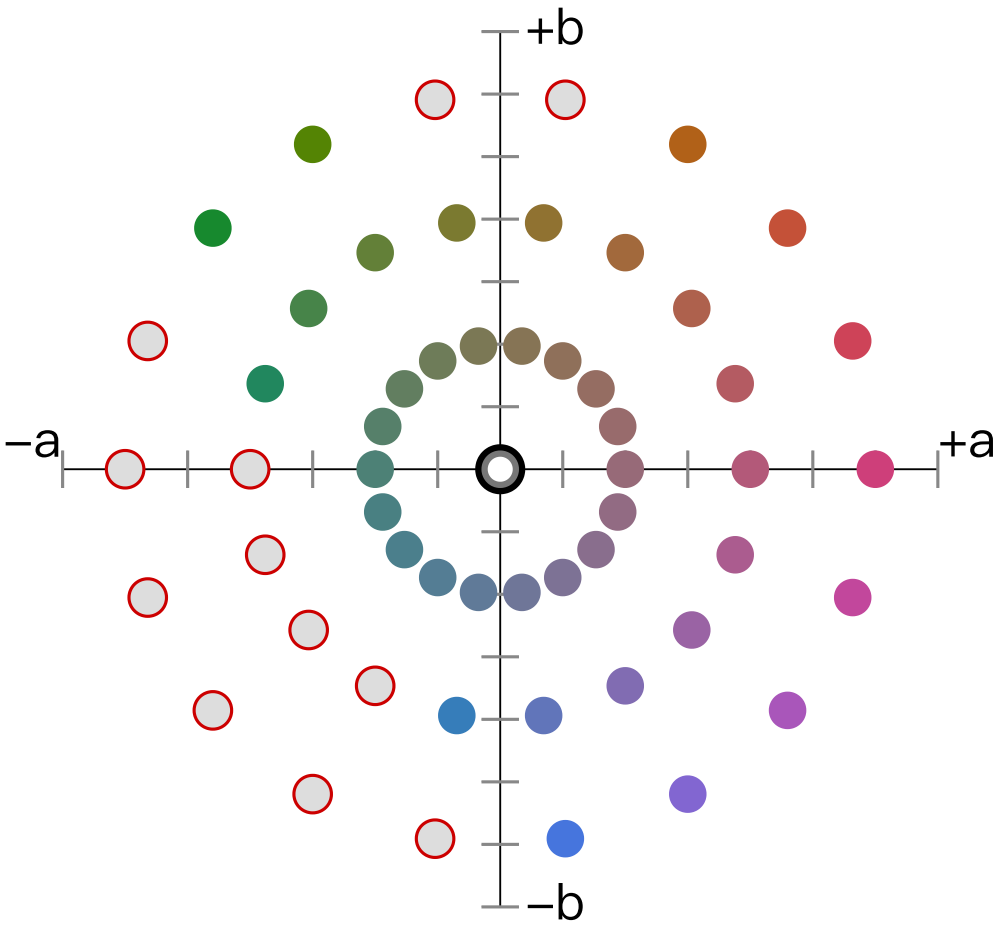


Figure 5 *This figure shows the L=50 plane of the CIE Lab color space. 20 degree increments in CIE LCH are displayed as circles at three levels of Chroma: 20, 40 and 60. All the 20 Chroma colors fit inside sRGB gamut, some of 40 and 60 Chroma are outside. These out of gamut colors are visualized as grey, with a red warning outer stroke.*

Note: The L axis in Lab and LCH is not to be confused with the L axis in HSL. For example, in HSL, the sRGB colors blue (#00F) and yellow (#FF0) have the same value of L (50%) even though visually, blue is much darker. This is much clearer in Lab: sRGB blue is lab(29.567% 68.298 -112.0294) while sRGB yellow is lab(97.607% -15.753 93.388). In Lab and LCH, if two colors have the same measured L value, they have identical visual lightness. HSL and related polar RGB models were developed in an attempt to give similar usability benefits for RGB that LCH gave to Lab, but are significantly less accurate.

Although the use of CIE Lab and LCH is widespread, it is known to have some problems. In particular:

Hue linearity

In the blue region (LCH Hue between 270° and 330°), visual hue departs from what LCH predicts. Plotting a set of blues of the same hue and differing Chroma, which should lie on a

straight line from the neutral axis, instead form a curve. Put another way, as a saturated blue has it's Chroma progressively reduced, it becomes noticeably purple.

Hue uniformity
: While hues in LCH are in general evenly spaced, (and far better than HSL or HWB), uniformity is not perfect.

Over-prediction of high Chroma differences
: For high Chroma colors, changes in Chroma are less noticeable than for more neutral colors.

These deficiencies affect, for example, creation of evenly spaced gradients, gamut mapping from one color space to a smaller one, and computation of the visual difference between two colors.

To compensate for this, formulae to predict the visual difference between two colors (delta E) have been made more accurate over time (but also, much more complex to compute). The current industry standard formula, delta E 2000, works well to mitigate some of the Lab and LCH problems. A sample implementation is given in § 18.1 ΔE2000.

This does not help with hue curvature, however.

§ 9.2. Oklab and Oklch

This section is not normative.

Recently, Oklab, an improved Lab-like space has been developed [Oklab]. The corresponding polar form is called Oklch. It was produced by numerical optimization of a large dataset of visually similar colors, and has improved hue linearity, hue uniformity, and chroma uniformity compared to CIE LCH.

Like CIE Lab, there is a central lightness L axis which is usually written as a unitless number in the range [0,1]; for compatibility with the rest of CSS, it may be written as a percentage. 100% means an L value of 1.0. L=0% or 0.0 is deep black (no light at all) while L=100% or 1.0 is a diffuse white.

> NOTE: Unlike CIE Lab, which assumes adaptation to the diffuse white, Oklab assumes adaptation to the color being defined, which is intended to make it scale invariant.

As with CIE Lab, the a and b axes convey hue; positive values along the a axis are a purpli while negative values are the complementary color, a green. Similarly, positive values along the b axis are yellow and negative are the complementary blue/violet.

The illuminant is D65, the same white point as most RGB color spaces.

Oklch has the same L axis as Oklab, but uses polar coordinates C (chroma) and H (hue).

NOTE: Unlike CIE LCH, where Chroma can reach values of 200 or more, Oklch Chroma ranges to 0.5 or so. The hue angles between CIE LCH and Oklch are broadly similar, but not identical.

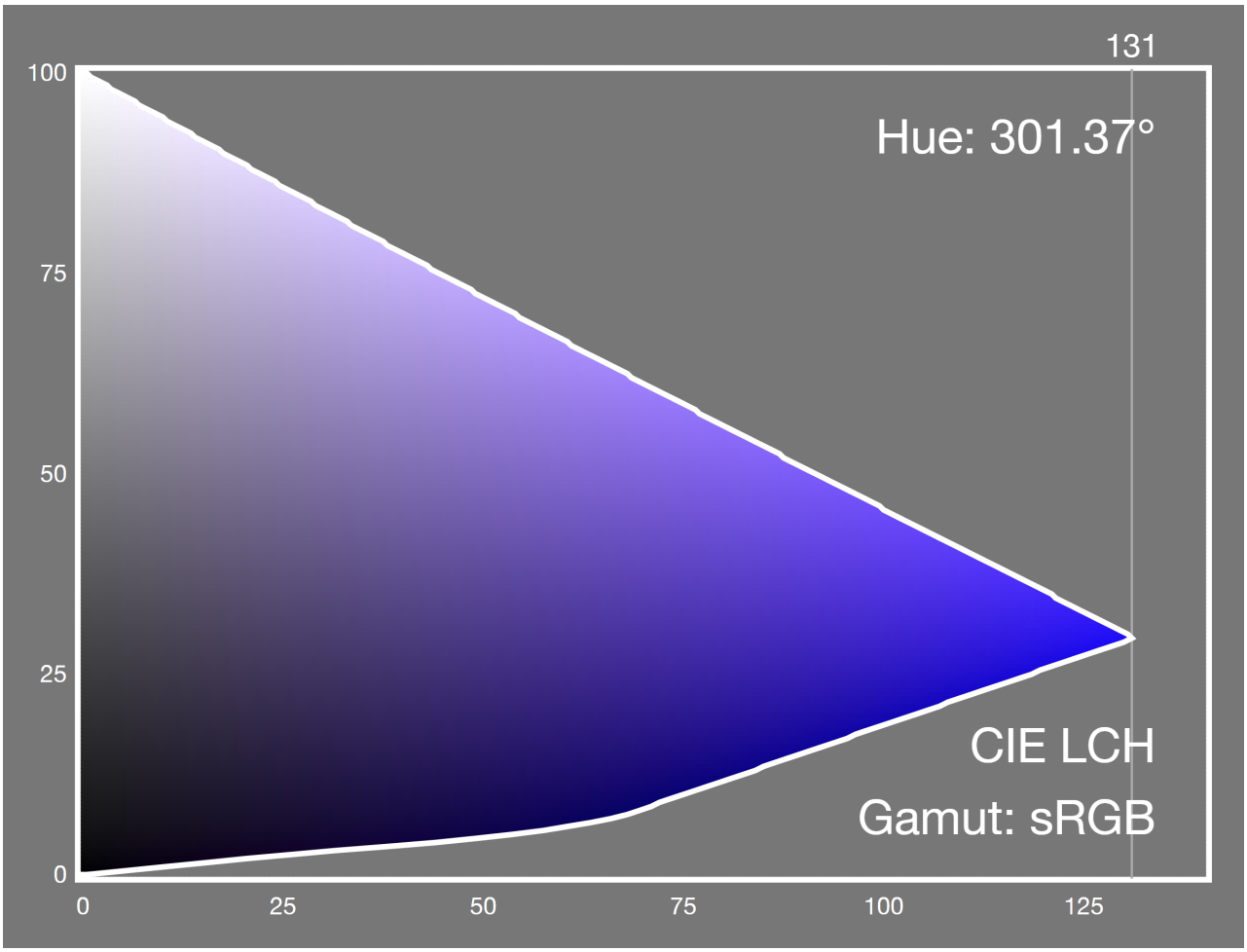


Figure 6 *A constant CIE LCH hue slice, showing the sRGB gamut around primary blue. A noticeable purpling is immediately evident.*

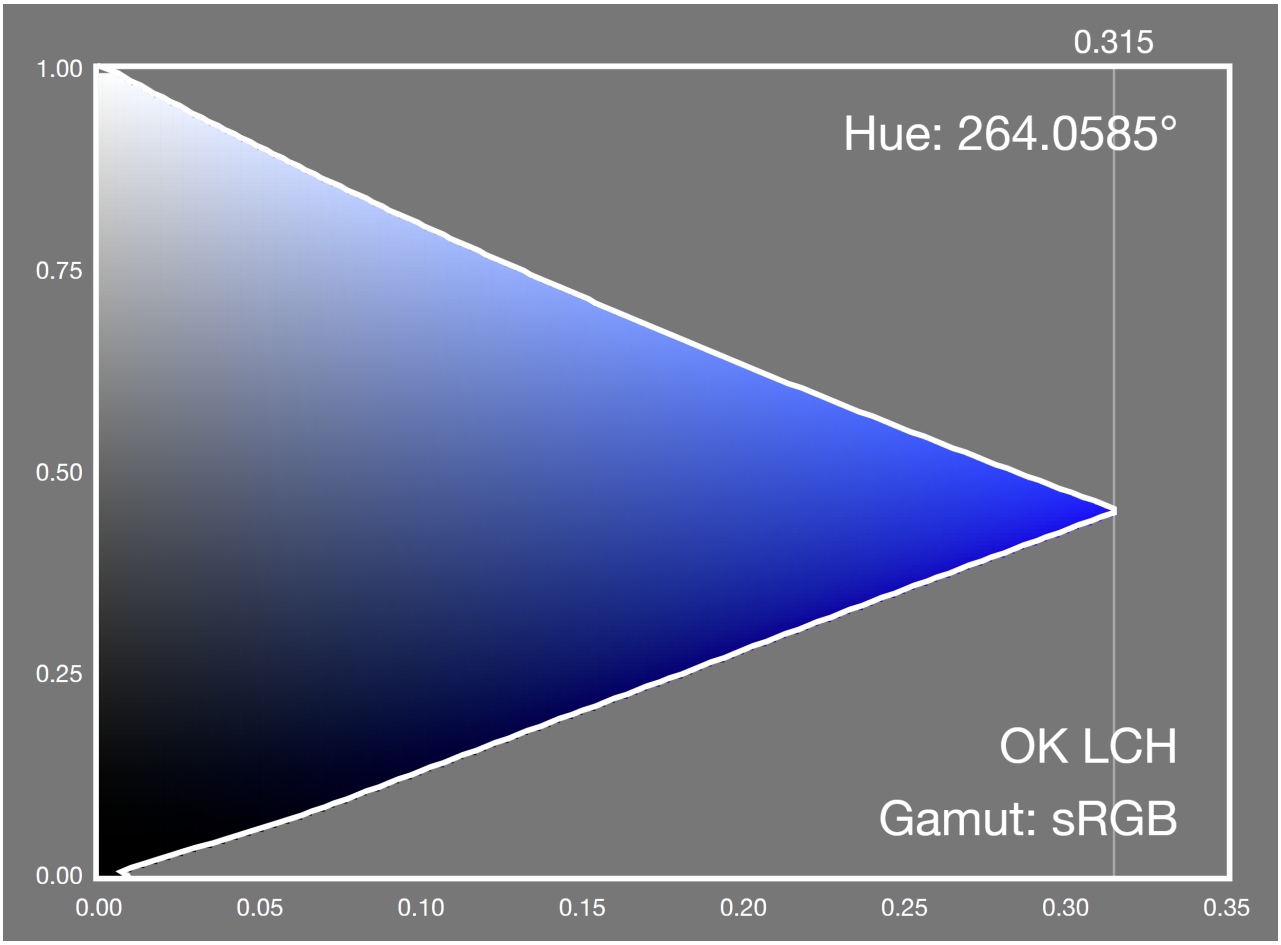


Figure 7 *A constant Oklch hue slice, showing the sRGB gamut around primary blue. The visual hue remains constant.*

Because Oklab is more perceptually uniform than CIE Lab, the color difference is a straightforward distance in 3D space (root sum of squares). Although trivial, a sample implementation is give in § 18.2 ΔEOK.

§ 9.3. Specifying Lab and LCH: the 'lab()' and 'lch()' functional notations

CSS allows colors to be directly expressed in Lab and LCH.

```
lab() = lab( [<percentage> | <number> | none]
      [ <percentage> | <number> | none]
      [ <percentage> | <number> | none]
      [ / [<alpha-value> | none] ]? )
```

Percentages	Allowed for L, a and b
Percent reference range	for L: 0% = 0.0, 100% = 100.0 for a and b: -100% = -125, 100% = 125

▼ TESTS

lab-001.html	(live test)	(source)
lab-002.html	(live test)	(source)
lab-003.html	(live test)	(source)
lab-004.html	(live test)	(source)
lab-005.html	(live test)	(source)
lab-006.html	(live test)	(source)
lab-007.html	(live test)	(source)
lab-008.html	(live test)	(source)
lab-l-over-100-1.html	(live test)	(source)
lab-l-over-100-2.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)
color-computed-lab.html (104 tests)	(live test)	(source)
color-invalid-lab.html (18 tests)	(live test)	(source)
color-valid-lab.html (116 tests)	(live test)	(source)

In ***'Lab'***, the first argument specifies the CIE Lightness, L. This is a number between '0%' or 0 and '100%' or 100 Values less than '0%' or 0 must be clamped to '0%' at parsed-value time; values greater than '100%' or 100 are clamped to '100%' at parsed-value time.

The second and third arguments are the distances along the "a" and "b" axes in the Lab color space, as described in the previous section. These values are signed (allow both positive and negative values) and theoretically unbounded (but in practice do not exceed ± 160).

There is an optional fourth <alpha-value> component, separated by a slash, representing the alpha component.

If the lightness of a Lab color (after clamping) is '0%', or '100%' the color will be displayed as black, or white, respectively due to gamut mapping to the display.

EXAMPLE 23 ¶

```
lab(29.2345% 39.3825 20.0664);
lab(52.2345 40.1645 59.9971);
lab(60.2345 -5.3654 58.956);
lab(62.2345% -34.9638 47.7721);
lab(67.5345 -8.6911 -41.6019);
lab(29.69% 44.888% -29.04%)
```

```
lch() = lch( [<percentage> | <number> | none]
      [ <percentage> | <number> | none]
      [ <hue> | none]
      [ / [<alpha-value> | none] ]? )
```

Percentages	Allowed for L and C
Percent reference range	for L: 0% = 0.0, 100% = 100.0 for C: 0% = 0, 100% = 150

▼ TESTS

lch-001.html	(live test) (source)
lch-002.html	(live test) (source)
lch-003.html	(live test) (source)
lch-004.html	(live test) (source)
lch-005.html	(live test) (source)
lch-006.html	(live test) (source)
lch-007.html	(live test) (source)
lch-008.html	(live test) (source)
lch-009.html	(live test) (source)
lch-010.html	(live test) (source)
lch-l-over-100-1.html	(live test) (source)
lch-l-over-100-2.html	(live test) (source)
color-valid.html (16 tests)	(live test) (source)

In CIE *'LCH'* the first argument specifies the CIE Lightness L, interpreted identically to the Lightness argument of 'lab()'.

The second argument is the chroma C, (roughly representing the "amount of color"). Its minimum useful value is '0', while its maximum is theoretically unbounded (but in practice does not exceed '230'). If the provided value is negative, it is clamped to '0' at parsed-value time.

The third argument is the hue angle H. It's interpreted similarly to the <hue> argument of 'hsl()', but doesn't map hues to angles in the same way because they are evenly spaced perceptually. Instead, '0deg' points along the positive "a" axis (toward purplish red), (as does '360deg', '720deg', etc.); '90deg' points along the positive "b" axis (toward mustard yellow), '180deg' points along the negative "a" axis (toward greenish cyan), and '270deg' points along the negative "b" axis (toward sky blue).

There is an optional fourth <alpha-value> component, separated by a slash, representing the alpha component.

If the chroma of an LCH color is '0%', the hue component is powerless. If the lightness of an LCH color (after clamping) is '0%', or '100%', the color will be displayed as black, or white, respectively due to gamut mapping to the display.

EXAMPLE 24 ¶

```
lch(29.2345% 44.2 27);
lch(52.2345% 72.2 56.2);
lch(60.2345 59.2 95.2);
lch(62.2345% 59.2 126.2);
lch(67.5345% 42.5 258.2);
lch(29.69% 45.553% 327.1)
```

There is no Web compatibility issue with 'lab' or 'lch'', which are new in this level of the specification, and so 'lab()' and 'lch()' do *not* support a legacy color syntax that separates all of their arguments with commas. Using commas inside these functions is an error.

§ 9.4. Specifying Oklab and Oklch: the 'oklab()' and 'oklch()' functional notations

CSS allows colors to be directly expressed in Oklab and Oklch.

```
oklab() = oklab( [ <percentage> | <number> | none]
    [ <percentage> | <number> | none]
    [ <percentage> | <number> | none]
    [ / [<alpha-value> | none] ]? )
```

Percentages	Allowed for L, a and b
Percent reference range	for L: 0% = 0.0, 100% = 1.0 for a and b: -100% = -0.4, 100% = 0.4

▼ TESTS

oklab-001.html	(live test)	(source)
oklab-002.html	(live test)	(source)
oklab-003.html	(live test)	(source)
oklab-004.html	(live test)	(source)
oklab-005.html	(live test)	(source)
oklab-006.html	(live test)	(source)
oklab-007.html	(live test)	(source)
oklab-008.html	(live test)	(source)
oklab-009.html	(live test)	(source)
oklab-l-over-1-1.html	(live test)	(source)
oklab-l-over-1-2.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

In *'Oklab'* the first argument specifies the Oklab Lightness. This is a number between '0%' or 0 and '100%' or 1.0.

Values less than '0%' or 0.0 must be clamped to '0%' at parsed-value time; values greater than '100%' or 1.0 are clamped to '100%' at parsed-value time.

The second and third arguments are the distances along the "a" and "b" axes in the Oklab color space, as described in the previous section. These values are signed (allow both positive and negative values) and theoretically unbounded (but in practice do not exceed ± 0.5).

There is an optional fourth <alpha-value> component, separated by a slash, representing the alpha component.

If the lightness of an Oklab color is '0%' or 0, or '100%' or 1.0, the color will be displayed as black, or white, respectively due to gamut mapping to the display.

EXAMPLE 25 ¶

```
oklab(40.101% 0.1147 0.0453);
oklab(59.686% 0.1009 0.1192);
oklab(0.65125 -0.0320 0.1274);
oklab(66.016% -0.1084 0.1114);
oklab(72.322% -0.0465 -0.1150);
oklab(42.1% 41% -25%)
```

```
oklch() = oklch( [ <percentage> | <number> | none]
      [ <percentage> | <number> | none]
      [ <hue> | none]
      [ / [<alpha-value> | none] ]? )
```

Percentages	Allowed for L and C
Percent reference range	for L: 0% = 0.0, 100% = 1.0 for C: 0% = 0.0 100% = 0.4

▼ TESTS

oklch-001.html	(live test)	(source)
oklch-002.html	(live test)	(source)
oklch-003.html	(live test)	(source)
oklch-004.html	(live test)	(source)
oklch-005.html	(live test)	(source)
oklch-006.html	(live test)	(source)
oklch-007.html	(live test)	(source)
oklch-008.html	(live test)	(source)
oklch-009.html	(live test)	(source)
oklch-010.html	(live test)	(source)
oklch-011.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

In *'Oklch'* the first argument specifies the Oklch Lightness L, interpreted identically to the Lightness argument of 'oklab()'.

The second argument is the chroma C. Its minimum useful value is '0', while its maximum is theoretically unbounded (but in practice does not exceed '0.5'). If the provided value is negative, it is clamped to '0' at parsed-value time.

The third argument is the hue angle H. It's interpreted similarly to the <hue> arguments of 'hsl()' and 'lch()', but doesn't map hues to angles in the same way. '0deg' points along the positive "a" axis (toward purplish red), (as does '360deg', '720deg', etc.); '90deg' points along the positive "b" axis (toward mustard yellow), '180deg' points along the negative "a" axis (toward greenish cyan), and '270deg' points along the negative "b" axis (toward sky blue).

There is an optional fourth <alpha-value> component, separated by a slash, representing the alpha component.

If the chroma of an Oklch color is '0%' or 0, the hue component is powerless. If the lightness of an Oklch color is '0%' or 0, or '100%' or 1.0, the color will be displayed as black, or white, respectively due to gamut mapping to the display.

EXAMPLE 26 ¶

```
oklch(40.101% 0.12332 21.555);
oklch(59.686% 0.15619 49.7694);
oklch(0.65125 0.13138 104.097);
oklch(0.66016 0.15546 134.231);
oklch(72.322% 0.12403 247.996);
oklch(42.1% 48.25% 328.4)
```

There is no Web compatibility issue with 'oklab' or 'oklch", which are new in this level of the specification, and so 'oklab()' and 'oklch()' do *not* support a legacy color syntax that separates all of their arguments with commas. Using commas inside these functions is an error.

§ 9.5. Converting Lab or Oklab colors to LCH or Oklch colors

Conversion to the polar form is trivial:

1. H = atan2(b, a)
2. C = sqrt(a^2 + b^2)
3. L is the same

For extremely small values of a and b (near-zero Chroma), although the visual color does not change from being on the neutral axis, small changes to the values can result in the reported hue angle swinging about wildly and being essentially random. In CSS, this means the hue is powerless, and treated as missing when converted into LCH or Oklch; in non-CSS contexts this might be reflected as a missing value, such as NaN.

§ 9.6. Converting LCH or Oklch colors to Lab or Oklab colors

Conversion to the rectangular form is trivial:

1. If H is missing, a = b = 0
2. Otherwise,
    1. a = C cos(H)
    2. b = C sin(H)
3. L is the same

§ 10. Predefined Color Spaces

CSS provides several predefined color spaces including 'display-p3' [Display-P3], which is a wide gamut space typical of current wide-gamut monitors, 'prophoto-rgb', widely used by photographers and 'rec2020' [Rec.2020], which is a broadcast industry standard, ultra-wide gamut space capable of representing almost all visible real-world colors.

§ 10.1. Specifying Predefined Colors: the 'color()' function

The 'color()' function allows a color to be specified in a particular, specified color space (rather than the implicit sRGB color space that most of the other color functions operate in). Its syntax is:

```
color() = color( <colorspace-params> [ / [ <alpha-value> | none ] ]? )
<colorspace-params> = [ <predefined-rgb-params> | <xyz-params>]
<predefined-rgb-params> = <predefined-rgb> [ <number> | <percentage> | n
<predefined-rgb> = srgb | srgb-linear | display-p3 | a98-rgb | prophoto-
<xyz-params> = <xyz-space> [ <number> | <percentage> | none ]{3}
<xyz-space> = xyz | xyz-d50 | xyz-d65
```

▼ TESTS

color-computed-color-function.html (401 tests)	(live test)	(source)
color-invalid-color-function.html (115 tests)	(live test)	(source)
color-valid-color-function.html (258 tests)	(live test)	(source)

The color function takes parameters specifying a color, in an explicitly listed color space.

It represents either an invalid color, as described below, or a valid color.

The parameters have the following form:

- An <ident> denoting one of the predefined color spaces (such as 'display-p3') Individual predefined color spaces may further restrict whether <number>s or <percentage>s or both, may be used.

  If the <ident> names a non-existent color space (a name that does not match one of the predefined color spaces), this argument represents an invalid color.

- The three parameter values that the color space takes (RGB or XYZ values).

An out of gamut color has component values less than 0 or 0%, or greater than 1 or 100%. These are not invalid, and are retained for intermediate computations; instead, for display, they are css gamut mapped using a relative colorimetric intent which brings the values (in the display color space) within the range 0/0% to 1/100% at actual-value time.

- An optional slash-separated <alpha-value>.

▼ TESTS

color-valid.html (16 tests) (live test) (source)

There is no Web compatibility issue with 'color()', which is new in this level of the specification, and so 'color()' does *not* support a legacy color syntax that separates all of its arguments with commas. Using commas inside this function is an error.

A color which is either an invalid color or an out of gamut color ***can't be displayed***.

If the specified color ***can be displayed***, (that is, it isn't an invalid color and isn't out of gamut) then this is the actual value of the 'color()' function.

If the specified color is a valid color but can't be displayed, the actual value is derived from the specified color, css gamut mapped for display.

If the color is an invalid color, the used value is opaque black.

EXAMPLE 27

This very intense lime color is in-gamut for rec.2020:

```
color(rec2020 0.42053 0.979780 0.00579);
```

in LCH, that color is

```
lch(86.6146% 160.0000 136.0088);
```

in display-p3, that color is

```
color(display-p3 -0.6112 1.0079 -0.2192);
```

and is out of gamut for display-p3 (red and blue are negative, green is greater than 1). If you have a display-p3 screen, that color is:

- *valid*
- *in gamut* (for rec.2020)
- *out of gamut* (for your display)
- and so *can't be displayed*

The color used for display will be a less intense color produced automatically by gamut mapping.

INVALID EXAMPLE28

This example has a typo! An intense green is provided in profoto-rgb space (which doesn't exist). This makes it invalid, so the used value is opaque black

```
color(profoto-rgb 0.4835 0.9167 0.2188)
```

§ 10.2. The Predefined sRGB Color Space: the 'sRGB' keyword

The ***sRGB*** predefined color space defined below is the same as is used for legacy sRGB colors, such as 'rgb()'.

'srgb'

The 'srgb' [SRGB] color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the range [0, 1].

The whitepoint is D65.

[SRGB] specifies two viewing conditions, *encoding* and *typical*. The [ICC] recommends using the *encoding* conditions for color conversion and for optimal viewing, which are the values in the table below.

sRGB is the default color space for CSS, used for all the legacy color functions.

It has the following characteristics:

	x	y
Red chromaticity	0.640	0.330
Green chromaticity	0.300	0.600
Blue chromaticity	0.150	0.060
White chromaticity	D65	
Transfer function	see below	
White luminance	80.0 cd/m^2	
Black luminance	0.20 cd/m^2	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

```
let sign = c < 0? -1 : 1;
let abs = Math.abs(c);

if (abs < 0.04045) {
  cl = c / 12.92;
}
else {
  cl = sign * (Math.pow((abs + 0.055) / 1.055, 2.4));
}
```

c is the gamma-encoded red, green or blue component. cl is the corresponding linear-light component.

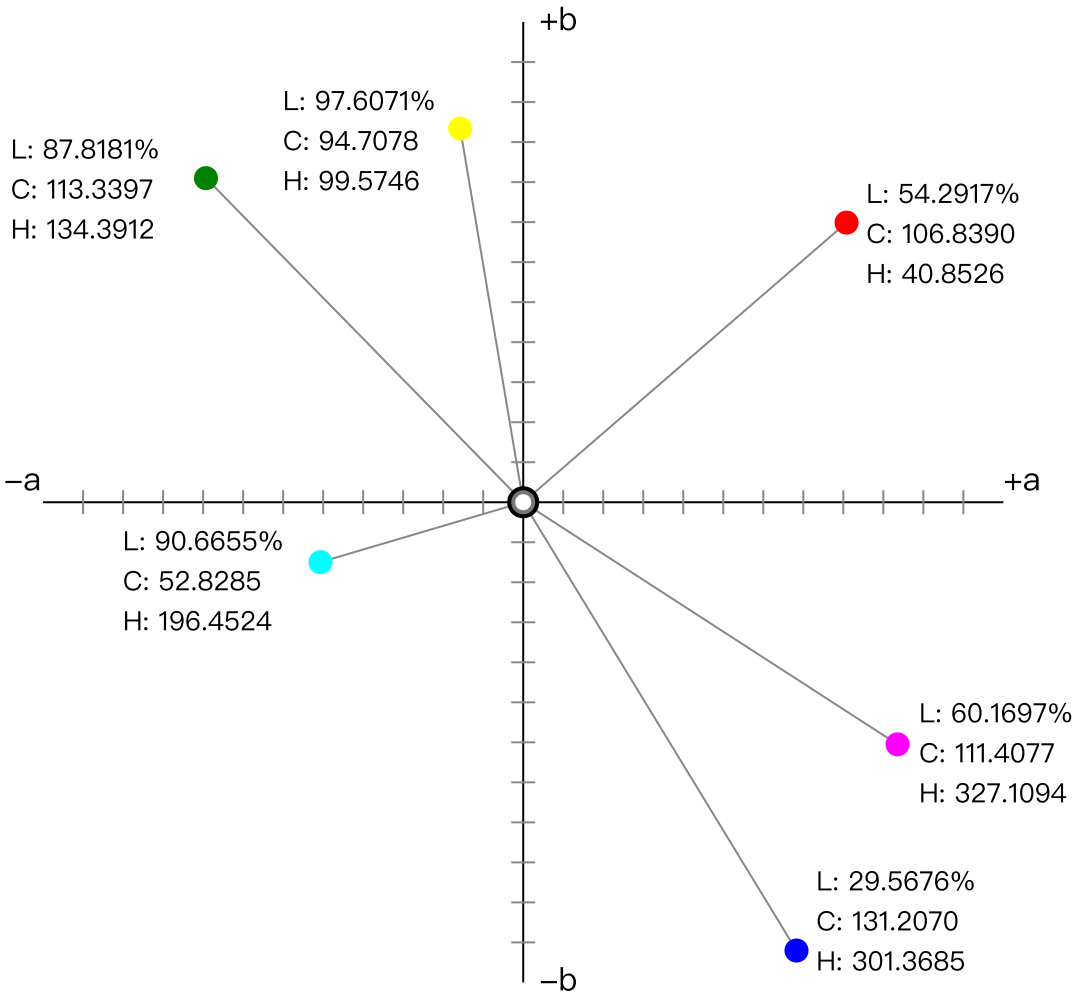


Figure 8 *Visualization of the sRGB color space in LCH. The primaries and secondaries are shown.*

▼ TESTS

predefined-001.html (live test) (source)
predefined-002.html (live test) (source)
color-valid.html (16 tests) (live test) (source)

§ 10.3. The Predefined Linear-light sRGB Color Space: the 'srgb-linear' keyword

The ***sRGB-linear*** predefined color space is the same as 'srgb' *except* that the transfer function is linear-light (there is no gamma-encoding).

'srgb-linear'

The 'srgb-linear' [SRGB] color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the range [0, 1]. The whitepoint is D65.

It has the following characteristics:

	x	y
Red chromaticity	0.640	0.330
Green chromaticity	0.300	0.600
Blue chromaticity	0.150	0.060
White chromaticity	D65	
Transfer function	unity, see below	
White luminance	80.0 cd/m^2	
Black luminance	0.20 cd/m^2	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

```
cl = c;
```

c is the red, green or blue component. cl is the corresponding linear-light component, which is identical.

▼ TESTS

srgb-linear-001.html	(live test) (source)
srgb-linear-002.html	(live test) (source)
srgb-linear-003.html	(live test) (source)
srgb-linear-004.html	(live test) (source)
color-valid.html (16 tests)	(live test) (source)

To avoid banding artifacts, a higher precision is required for 'srgb-linear' than for 'srgb'.

EXAMPLE 29 ¶

For example, these are the same color

```
color(srgb 0.691 0.139 0.259)
color(srgb-linear 0.435 0.017 0.055)
```

§ 10.4. The Predefined Display P3 Color Space: the 'display-p3' keyword

'display-p3'

The 'display-p3' [Display-P3] color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the

range [0, 1]. It uses the same primary chromaticities as [DCI-P3], but with a D65 whitepoint, and the same transfer curve as sRGB.

Modern displays, TVs, laptop screens and phone screens are able to display all, or nearly all, of the display-p3 gamut.

It has the following characteristics:

	x	y
Red chromaticity	0.680	0.320
Green chromaticity	0.265	0.690
Blue chromaticity	0.150	0.060
White chromaticity	D65	
Transfer function	same as srgb	
White luminance	80.0 cd/m^2	
Black luminance	0.80 cd/m^2	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

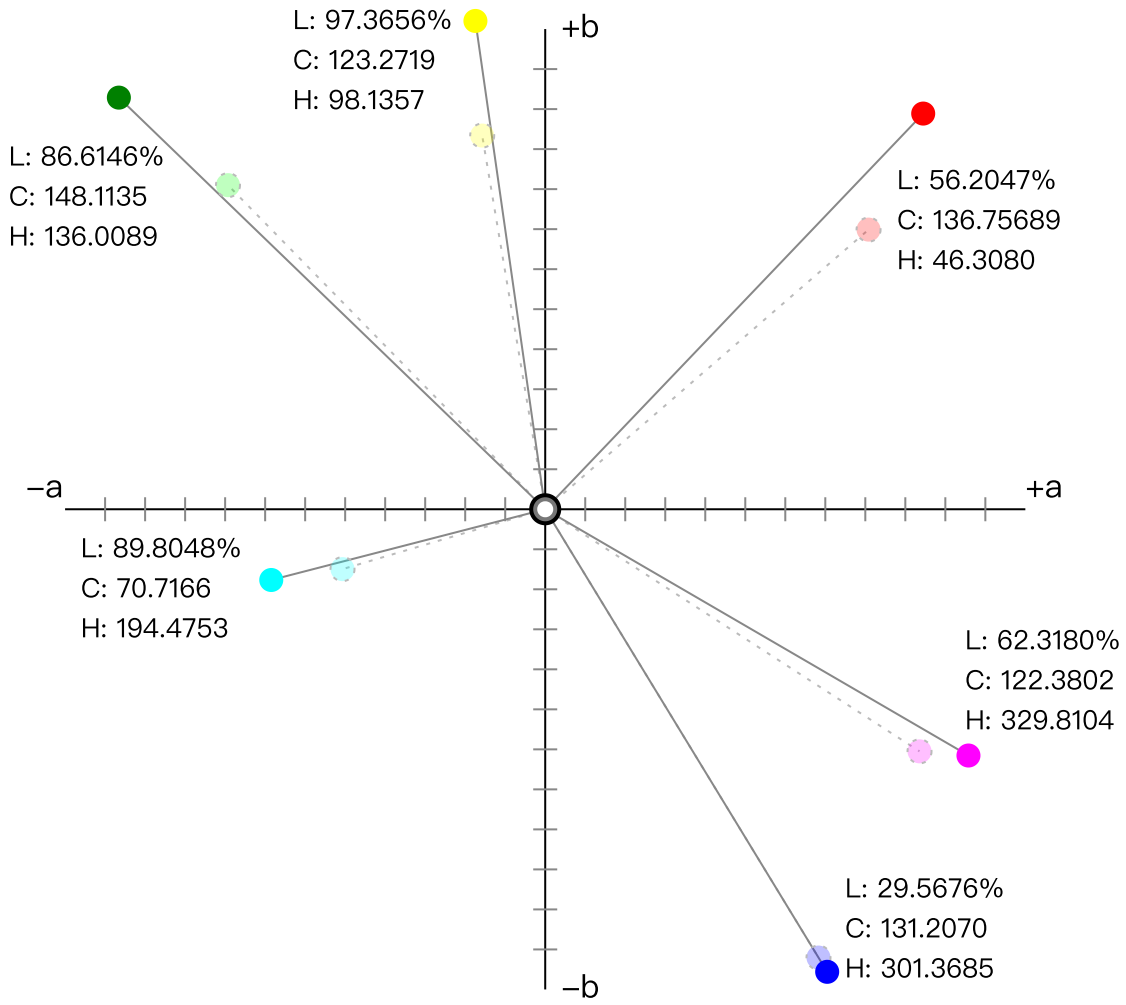


Figure 9 *Visualization of the P3 color space in LCH. The primaries and secondaries are shown (but in sRGB, not in the correct colors). For comparison, the sRGB primaries and secondaries are also shown, as dashed circles. P3 primaries have higher Chroma.*

▼ TESTS

predefined-005.html	(live test)	(source)
predefined-006.html	(live test)	(source)
display-p3-001.html	(live test)	(source)
display-p3-002.html	(live test)	(source)
display-p3-003.html	(live test)	(source)
display-p3-004.html	(live test)	(source)
display-p3-005.html	(live test)	(source)
display-p3-006.html	(live test)	(source)

§ 10.5. The Predefined A98 RGB Color Space: the 'a98-rgb' keyword

'a98-rgb'

The 'a98-rgb' color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the range [0, 1]. The transfer curve is a gamma function, close to but not exactly 1/2.2.

It has the following characteristics:

	x	y
Red chromaticity	0.6400	0.3300
Green chromaticity	0.2100	0.7100
Blue chromaticity	0.1500	0.0600
White chromaticity	D65	
Transfer function	256/563	
White luminance	160.0 cd/m^2	
Black luminance	0.5557 cd/m^2	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

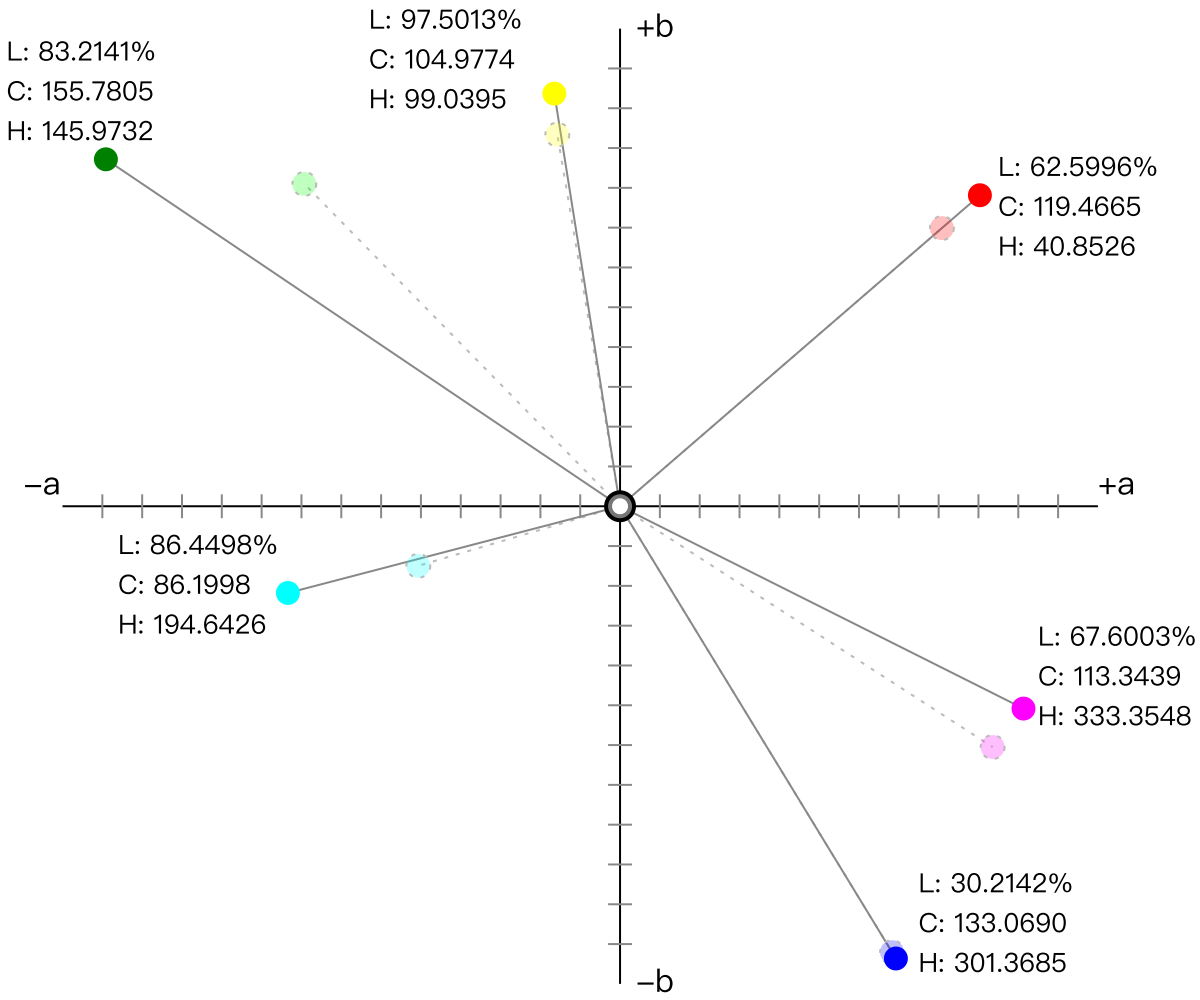


Figure 10 *Visualization of the A98 color space in LCH. The primaries and secondaries are shown (but in sRGB, not in the correct colors). For comparison, the sRGB primaries and secondaries are also shown, as dashed circles. a98 primaries have higher Chroma, especially the yellow, green and cyan.*

▼ TESTS

predefined-007.html	(live test) (source)
predefined-008.html	(live test) (source)
a98rgb-001.html	(live test) (source)
a98rgb-002.html	(live test) (source)
a98rgb-003.html	(live test) (source)
a98rgb-004.html	(live test) (source)
color-valid.html (16 tests)	(live test) (source)

§ 10.6. The Predefined ProPhoto RGB Color Space: the 'prophoto-rgb' keyword

'prophoto-rgb'

The 'prophoto-rgb' color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the range [0, 1]. The transfer curve is a gamma function with a value of 1/1.8, and a small linear portion near black. The white point is D50, the same as is used by CIE Lab. Thus, conversion to CIE Lab does not require the chromatic adaptation step.

The ProPhoto RGB space uses hyper-saturated, non physically realizable primaries. These were chosen to allow a wide color gamut and in particular, to minimize hue shifts under tonal manipulation. It is often used in digital photography as a wide gamut color space for the archival version of photographic images. The 'prophoto-rgb' color space allows CSS to specify colors that will match colors in such images having the same RGB values.

The ProPhoto RGB space was originally developed by Kodak and is described in [Wolfe]. It was standardized by ISO as [ROMM],[ROMM-RGB].

The white luminance is given as a range, and the viewing flare (and thus, the black luminance) is 0.5% to 1.0% of this.

It has the following characteristics:

	x	y
Red chromaticity	0.734699	0.265301
Green chromaticity	0.159597	0.840403
Blue chromaticity	0.036598	0.000105
White chromaticity	D50	
Transfer function	see below	
White luminance	160.0 to 640.0 cd/m^2	
Black luminance	See text	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

```
const E = 16/512;
let sign = c < 0? -1 : 1;
let abs = Math.abs(c);

if (abs <= E) {
  cl =  c / 16;
}
```

```
else {
  cl = sign * Math.pow(c, 1.8);
}
```

c is the gamma-encoded red, green or blue component. cl is the corresponding linear-light component.

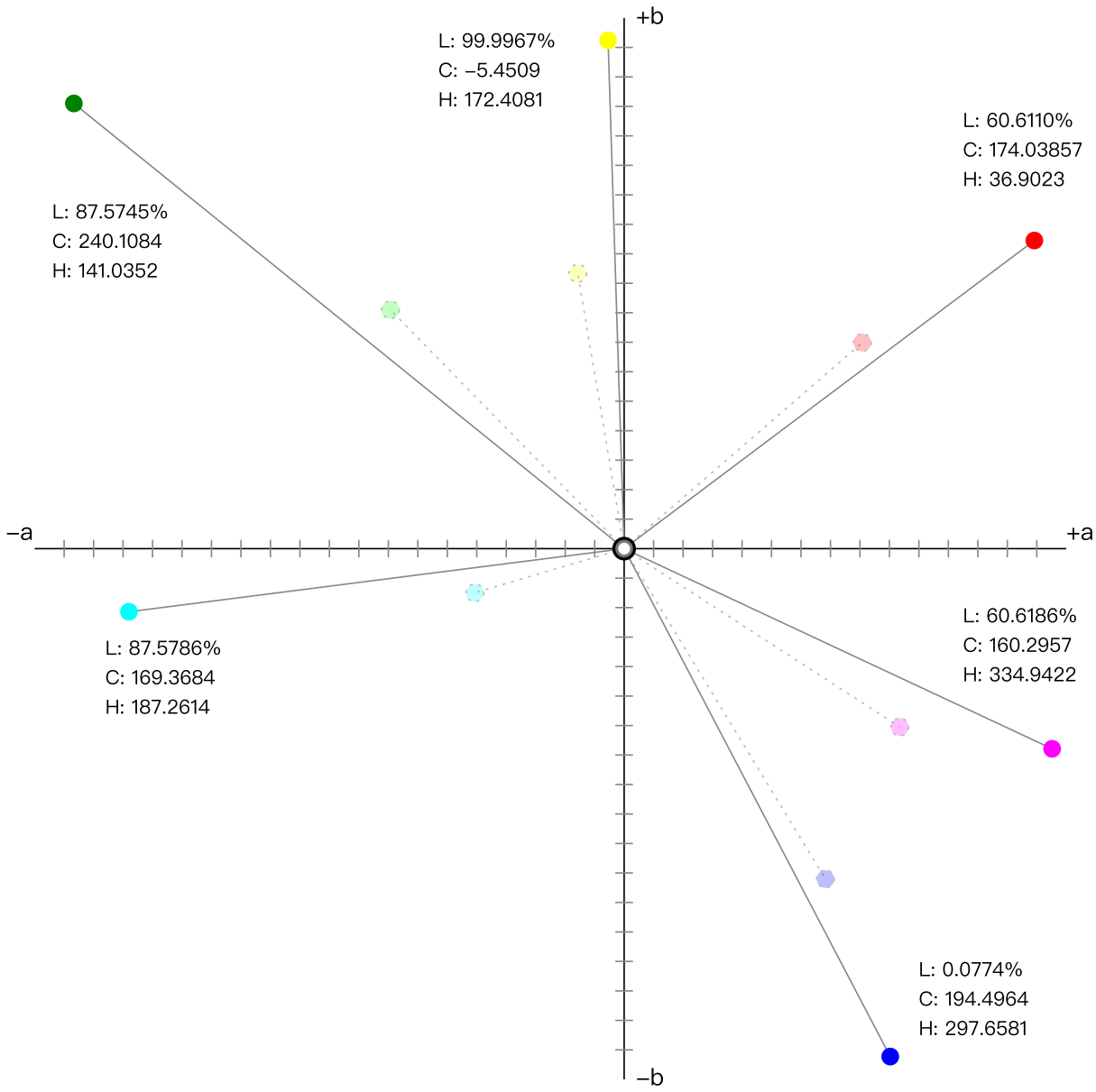


Figure 11 *Visualization of the prophoto-rgb color space in LCH. The primaries and secondaries are shown (but in sRGB, not in the correct colors). For comparison, the sRGB primaries and secondaries are also shown, as dashed circles. prophoto-rgb primaries and secondaries have much higher Chroma, but much of this ultrawide gamut does not correspond to physically realizable colors.*

▼ TESTS

predefined-009.html (live test) (source)

predefined-010.html	(live test)	(source)
prophoto-rgb-001.html	(live test)	(source)
prophoto-rgb-002.html	(live test)	(source)
prophoto-rgb-003.html	(live test)	(source)
prophoto-rgb-004.html	(live test)	(source)
prophoto-rgb-005.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

§ 10.7. The Predefined ITU-R BT.2020-2 Color Space: the 'rec2020' keyword

'rec2020'

The 'rec2020' [Rec.2020] color space accepts three numeric parameters, representing the red, green, and blue channels of the color. In-gamut colors have all three components in the range [0, 1], ("full-range", in video terminology). ITU Reference 2020 is used for Ultra High Definition, 4k and 8k television.

The primaries are physically realizable, but with difficulty as they lie very close to the spectral locus.

Current displays are unable to reproduce the full gamut of rec2020. Coverage is expected to increase over time as displays improve.

It has the following characteristics:

	x	y
Red chromaticity	0.708	0.292
Green chromaticity	0.170	0.797
Blue chromaticity	0.131	0.046
White chromaticity	D65	
Transfer function	see below, from [Rec.2020] table 4	
Image state	display-referred	
Percentages	Allowed for R, G and B	
Percent reference range	for R,G,B: 0% = 0.0, 100% = 1.0	

```
const α = 1.09929682680944 ;
const β = 0.018053968510807;

let sign = c < 0? -1 : 1;
let abs = Math.abs(c);

if (abs < β * 4.5 ) {
```

```
  cl = c / 4.5;
}
else {
  cl = sign * (Math.pow((abs + α −1 ) / α, 1/0.45));
}
```

c is the gamma-encoded red, green or blue component. cl is the corresponding linear-light component.

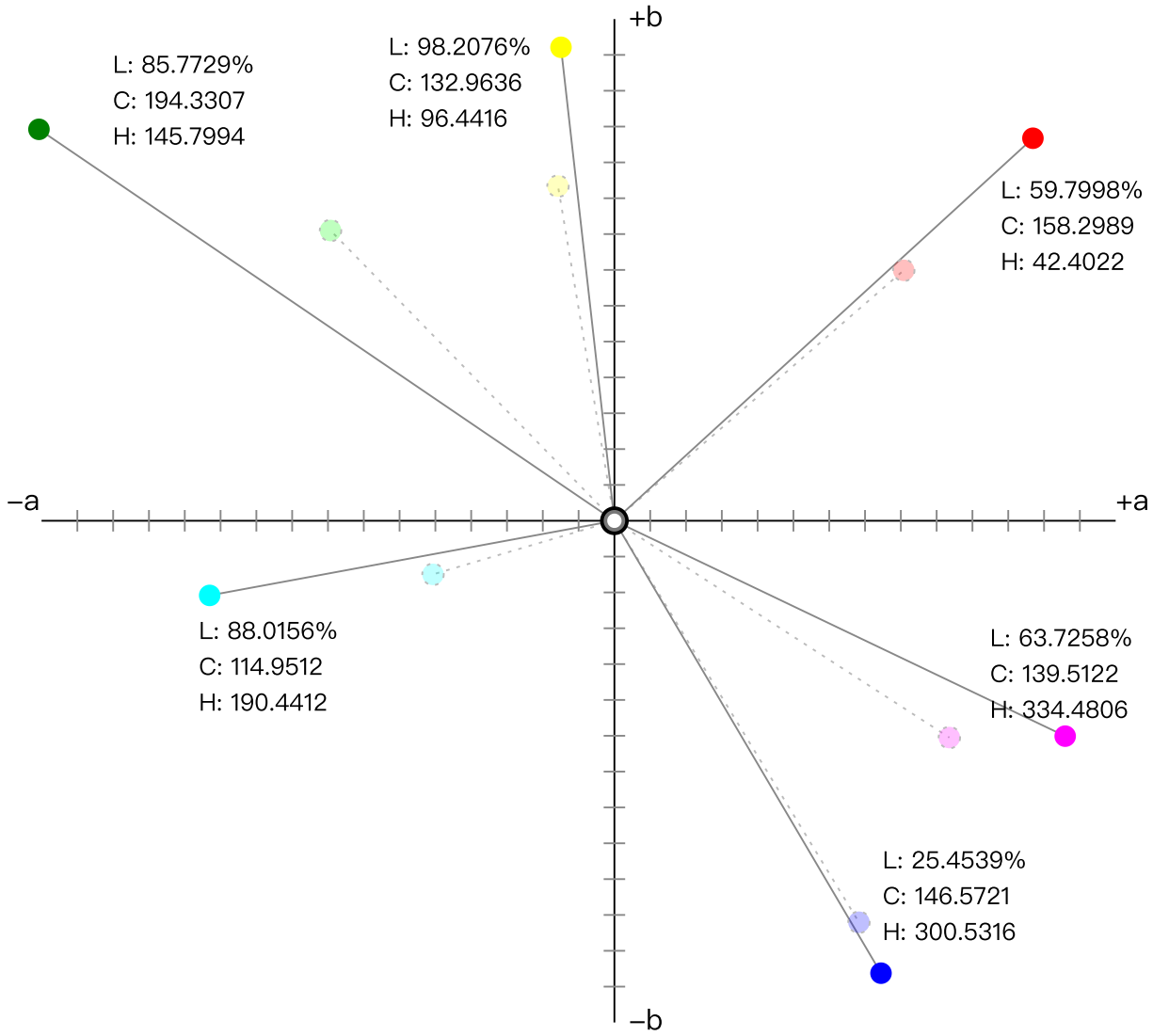


Figure 12 *Visualization of the rec2020 color space in LCH. The primaries and secondaries are shown (but in sRGB, not in the correct colors). For comparison, the sRGB primaries and secondaries are also shown, as dashed circles. rec2020 primaries have much higher Chroma.*

▼ TESTS

predefined-011.html	(live test) (source)
predefined-012.html	(live test) (source)
rec2020-001.html	(live test) (source)

rec2020-002.html	(live test)	(source)
rec2020-003.html	(live test)	(source)
rec2020-004.html	(live test)	(source)
rec2020-005.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

§ 10.8. The Predefined CIE XYZ Color Spaces: the 'xyz-d50', 'xyz-d65', and 'xyz' keywords

'xyz-d50' , ***'xyz-d65'*** , ***'xyz'***

The 'xyz' color space accepts three numeric parameters, representing the X,Y and Z values. It represents the CIE XYZ [COLORIMETRY] color space, scaled such that diffuse white has a luminance (Y) of 1.0. and, if necessary, chromatically adapted to the reference white.

The reference white for 'xyz-d50' is D50, while the reference white for 'xyz-d65' and 'xyz' is D65.

Values greater than 1.0/100% are allowed and must not be clamped; they represent colors brighter than diffuse white. Values less than 0/0% are uncommon, but can occur as a result of chromatic adaptation, and likewise must not be clamped.

It has the following characteristics:

Percentages	Allowed for X,Y,Z
Percent reference range	for X,Y,Z: 0% = 0.0, 100% = 1.0

EXAMPLE 30 ¶

These are exactly equivalent:

```
#7654CD
rgb(46.27% 32.94% 80.39%)
lab(44.36% 36.05 -58.99)
color(xyz-d50 0.2005 0.14089 0.4472)
color(xyz-d65 0.21661 0.14602 0.59452)
```

▼ TESTS

predefined-016.html	(live test)	(source)
xyz-001.html	(live test)	(source)
xyz-002.html	(live test)	(source)
xyz-003.html	(live test)	(source)

xyz-004.html	(live test)	(source)
xyz-005.html	(live test)	(source)
xyz-d50-001.html	(live test)	(source)
xyz-d50-002.html	(live test)	(source)
xyz-d50-003.html	(live test)	(source)
xyz-d50-004.html	(live test)	(source)
xyz-d50-005.html	(live test)	(source)
xyz-d65-001.html	(live test)	(source)
xyz-d65-002.html	(live test)	(source)
xyz-d65-003.html	(live test)	(source)
xyz-d65-004.html	(live test)	(source)
xyz-d65-005.html	(live test)	(source)
color-valid.html (16 tests)	(live test)	(source)

§ 10.9. Converting Predefined Color Spaces to Lab or Oklab

For all predefined RGB color spaces, conversion to Lab requires several steps, although in practice all but the first step are linear calculations and can be combined.

1. Convert from gamma-encoded RGB to linear-light RGB (undo gamma encoding)
2. Convert from linear RGB to CIE XYZ
3. If needed, convert from a D65 whitepoint (used by 'sRGB', 'display-p3', 'a98-rgb' and 'rec2020') to the D50 whitepoint used in Lab, with the Bradford transform. 'prophoto-rgb' already has a D50 whitepoint.
4. Convert D50-adapted XYZ to Lab

Conversion to Oklab is similar, but the chromatic adaptation step is only needed for 'prophoto-rgb'.

1. Convert from gamma-encoded RGB to linear-light RGB (undo gamma encoding)
2. Convert from linear RGB to CIE XYZ
3. If needed, convert from a D50 whitepoint (used by 'prophoto-rgb') to the D65 whitepoint used in Oklab, with the Bradford transform.
4. Convert D65-adapted XYZ to Oklab

There is sample JavaScript code for these conversions in § 17 Sample code for Color Conversions.

§ 10.10. Converting Lab or Oklab to Predefined RGB Color Spaces

Conversion from Lab to predefined spaces like 'display-p3' or 'rec2020' also requires multiple steps, and again in practice all but the last step are linear calculations and can be combined.

1. Convert Lab to (D50-adapted) XYZ
2. If needed, convert from a D50 whitepoint (used by Lab) to the D65 whitepoint used in sRGB and most other RGB spaces, with the Bradford transform. 'prophoto-rgb'' does not require this step.
3. Convert from (D65-adapted) CIE XYZ to linear RGB
4. Convert from linear-light RGB to RGB (do gamma encoding)

Conversion from Oklab is similar, but the chromatic adaptation step is only needed for 'prophoto-rgb'.

1. Convert Oklab to (D65-adapted) XYZ
2. If needed, convert from a D65 whitepoint (used by Oklab) to the D50 whitepoint used in 'prophoto-rgb', with the Bradford transform.
3. Convert from (D65-adapted) CIE XYZ to linear RGB
4. Convert from linear-light RGB to RGB (do gamma encoding)

There is sample JavaScript code for these conversions in § 17 Sample code for Color Conversions.

Implementations may choose to implement these steps in some other way (for example, using an ICC profile with relative colorimetric rendering intent) provided the results are the same for colors inside both the source and destination gamuts.

§ 10.11. Converting Between Predefined RGB Color Spaces

Conversion from one predefined RGB color space to another requires multiple steps, one of which is only needed when the whitepoints differ. To convert from *src* to *dest*:

1. Convert from gamma-encoded *src*RGB to linear-light *src*RGB (undo gamma encoding)
2. Convert from linear *src*RGB to CIE XYZ
3. If *src* and *dest* have different whitepoints, convert the XYZ value from *src*White to *dest*White with the Bradford transform.
4. Convert from CIE XYZ to linear *dest*RGB
5. Convert from linear-light *dest*RGB to *dest*RGB (do gamma encoding)

There is sample JavaScript code for this conversion for the predefined RGB color spaces, in § 17 Sample code for Color Conversions.

§ 10.12. Simple Alpha Compositing

When drawing, implementations must handle alpha according to the rules in Section 5.1 Simple alpha compositing of [Compositing].

§ 11. Converting Colors

Colors may be converted from one color space to another and, provided that there is no gamut mapping and that each color space can represent out of gamut colors, (for RGB spaces, this means that the transfer function is defined over the extended range) then (subject to numerical precision and round-off error) the two colors will look the same and represent the same color sensation.

To convert a color *col1* in a source color space *src* with white point *src-white* to a color *col2* in destination color space *dest* with white point *dest-white*:

1. If *src* is in a cylindrical polar color representation, first convert *col1* to the corresponding rectangular orthogonal color representation and let this be the new *col1*. Replace any missing component with zero.
2. If *src* is not a linear-light representation, convert it to linear light (undo gamma-encoding) and let this be the new *col1*.
3. Convert *col1* to CIE XYZ with a given whitepoint *src-white* and let this be *xyz*.
4. If *dest-white* is not the same as *src-white*, chromatically adapt *xyz* to *dest-white* using a linear Bradford chromatic adaptation transform, and let this be the new *xyz*.
5. If *dest* is a cylindrical polar color representation, let *dest-rect* be the corresponding rectangular orthogonal color representation. Otherwise, let *dest-rect* be *dest*.
6. Convert *xyz* to *dest*, followed by applying any transfer function (gamma encoding), producing *col2*.
7. If *dest* is a physical output color space, such as a display, then *col2* must be css gamut mapped so that it can be displayed.
8. If *dest-rect* is not the same as *dest*, in other words *dest* is a cylindrical polar color representation, convert from *dest-rect* to *dest*, and let this be *col2*. This may produce missing components.

§ 12. Color Interpolation

Color interpolation happens with gradients, compositing, filters, transitions, animations, and color mixing and color modification functions.

Interpolation between <color> values occurs by first checking the two colors for analogous components which will be carried forward; then (if required) converting them to a given color space which will be referred to as the ***interpolation color space*** below, and then linearly interpolating each component of the computed value of the color separately.

Interpolating to or from 'currentcolor' is possible. The numerical value used for this purpose is the used value.

ISSUE 1 Computed value needs to be able to represent combinations of 'currentColor' and an actual color. Consider the value of 'text-emphasis-color' in `div { text-emphasis: circle; transition: all 2s; }`
`div:hover { text-emphasis-color: lime; }`
`em { color: red; }` See Issue 445.

§ 12.1. Color Space for Interpolation

Various features in CSS depend on interpolating colors.

EXAMPLE 31
Examples include:

- <gradient>
- 'filter'
- 'animation'
- 'transition'

Mixing or otherwise combining colors has different results depending on the interpolation color space used. Thus, different color spaces may be more appropriate for each interpolation use case.

- In some cases, the result of physically mixing two colored lights is desired. In that case, the CIE XYZ or srgb-linear color space is appropriate, because they are linear in light intensity.
- If colors need to be evenly spaced perceptually (such as in a gradient), the Oklab color space (and the older Lab), are designed to be perceptually uniform.

- If avoiding graying out in color mixing is desired, i.e. maximizing chroma throughout the transition, Oklch (and the older LCH) work well for that.
- Lastly, compatibility with legacy Web content may be the most important consideration. The sRGB color space, which is neither linear-light nor perceptually uniform, is the choice here, even though it produces poorer results (overly dark or greyish mixes).

These features are collectively termed the ***host syntax***. They are not used by this specification itself, only exposed so that other specifications can use them; see e.g. use in *CSS Images 4* § 3.1 Linear Gradients: the linear-gradient() notation. The host syntax should define what the default interpolation color space should be for each case, and optionally provide syntax for authors to override this default. If such syntax is part of a property value, it should use a <color-interpolation-method> token, defined below for easy reference from other specifications. This ensures consistency across CSS, and that further customizations on how color interpolation is performed can automatically percolate across all of CSS.

```
<color-space> = <rectangular-color-space> | <polar-color-space>
<rectangular-color-space> = srgb | srgb-linear | display-p3 | a98-rgb |
<polar-color-space> = hsl | hwb | lch | oklch
<hue-interpolation-method> = [ shorter | longer | increasing | decreasin
<color-interpolation-method> = in [ <rectangular-color-space> | <polar-c
```

The keywords in the definitions of <rectangular-color-space> and <polar-color-space> each refer to their corresponding color space, represented in CSS either by the functional syntax with the same name, or (if no such function is present), by the corresponding <ident> in the 'color()' function.

▼ TESTS

color-mix-percents-01.html	(live test)	(source)
color-mix-percents-02.html	(live test)	(source)
gradient-interpolation-method-valid.html	(live test)	(source)
gradient-interpolation-method-invalid.html	(live test)	(source)
gradient-interpolation-method-computed.html	(live test)	(source)

If the host syntax does not define what color space interpolation should take place in, it defaults to Oklab.

Authors that prefer interpolation in sRGB in a particular instance can opt-in to the the old behavior by explicitly specifying sRGB as the interpolation color space, for example on a particular gradient where that result is desired.

If the colors to be interpolated are outside the gamut of the interpolation color space , then once converted to that space, they will contain out of range values.

These are not clipped, but the values are interpolated as-is.

§ 12.2. Interpolating with Missing Components

In the course of converting the two colors to the interpolation color space, any missing components will be replaced with the value 0.

Thus, the first stage in interpolating two colors is to classify any missing components in the input colors, and compare them to the components of the interpolation color space. If any analogous components which are missing components are found, they will be **carried forward** and re-inserted in the converted color before linear interpolation takes place.

The ***analogous components*** are as follows:

Category	Components
Reds	r,x
Greens	g,y
Blues	b,z
Lightness	L
Colorfulness	C, S
Hue	H
Opponent a	a
Opponent b	b

> NOTE: for the purposes of this classification, the XYZ spaces are considered super-saturated RGB spaces. Also, despite Saturation being Lightness-dependent, it falls in the same category as Chroma here. The Whiteness and Blackness components of HWB have no analogs in other color spaces.

EXAMPLE 32

For example, if these two colors are to be interpolated in Oklch, the missing hue in the CIE LCH color is analogous to the hue component of Oklch and will be carried forward while the missing blue component in the second color is not analogous to any Oklch component and will not be carried forward:

```
lch(50% 0.02 none)
color(display-p3 0.7 0.5 none)
```

which convert to

```
oklch(56.897% 0.0001 0)
oklch(63.612% 0.1522 78.748)
```

and with carried forward missing component re-inserted, the two colors to be interpolated are:

```
oklch(56.897% 0.0001 none)
oklch(63.612% 0.1522 78.748)
```

If a color with a carried forward missing component is interpolated with another color which is not missing that component, the missing component is treated as having the *other color's* component value.

Therefore, the carrying-forward step must be performed *before* any powerless component handling.

EXAMPLE 33

For example, if these two colors are interpolated, the second of which has a missing hue:

```
oklch(78.3% 0.108 326.5)
oklch(39.2% 0.4 none)
```

Then the actual colors to be interpolated are

```
oklch(78.3% 0.108 326.5)
oklch(39.2% 0.4 326.5)
```

and not

```
oklch(78.3% 0.108 326.5)
oklch(39.2% 0.4 0)
```

If the carried forward missing component is alpha, the color must be premultiplied with this carried forward value, not with the zero value that would have resulted from color conversion.

EXAMPLE 34

For example, if these two colors are interpolated, the second of which has a missing alpha:

```
oklch(0.783 0.108 326.5 / 0.5)
oklch(0.392 0.4 0 / none)
```

Then the actual colors to be interpolated are

```
oklch(78.3% 0.108 326.5 / 0.5)
oklch(39.2% 0.4 0 / 0.5)
```

giving the premultiplied Oklch values [0.3915, 0.054, 326] and [0.196, 0.2, 0].

If both colors are missing a given component, the interpolated color will also be missing that component.

§ 12.3. Interpolating with Alpha

When the colors to be interpolated are not fully opaque, they are transformed into ***premultiplied color values*** as follows:

- If the alpha value is 'none', the premultiplied value is the un-premultiplied value. Otherwise,
- If any component value is 'none', the premultiplied value is also 'none'.
- For rectangular orthogonal color coordinate systems, all component values are multiplied by the alpha value.
- For cylindrical polar color coordinate systems, the hue angle is *not* premultiplied, but the other two axes are premultiplied.

To obtain a color value from a premultipled color value,

- If the interpolated alpha value is zero or 'none', the un-premultiplied value is the premultiplied value. Otherwise,
- If any component value is 'none', the un-premultiplied value is also 'none'.
- otherwise, each component which had been premultiplied is divided by the interpolated alpha value.

▼ TESTS

color-transition-premultiplied.html (live test) (source)

▶ **Why is premultiplied alpha useful?**

EXAMPLE 35 ¶

For example, to interpolate, in the sRGB color space, the two sRGB colors rgb(24% 12% 98% / 0.4) and rgb(62% 26% 64% / 0.6) they would first be converted to premultiplied form [9.6% 4.8% 39.2%] and [37.2% 15.6% 38.4%] before interpolation.

The midpoint of linearly interpolating these colors would be [23.4% 10.2% 38.8%] which, with an alpha value of 0.5, is rgb(46.8% 20.4% 77.6% / 0.5) when premultiplication is undone.

EXAMPLE 36 ¶

To interpolate, in the Lab color space, the two colors rgb(76% 62% 03% / 0.4) and color(display-p3 0.84 0.19 0.72 / 0.6) they are first converted to lab lab(66.927% 4.873 68.622 / 0.4) lab(53.503% 82.672 -33.901 / 0.6) then the L, a and b coordinates are premultiplied before interpolation [26.771% 1.949 27.449] and [32.102% 49.603 -20.341].

The midpoint of linearly interpolating these would be [29.4365% 25.776 3.554] which, with an alpha value of 0.5, is lab(58.873% 51.552 7.108) / 0.5) when premultiplication is undone.

EXAMPLE 37 ¶

To interpolate, in the chroma-preserving LCH color space, the same two colors rgb(76% 62% 03% / 0.4) and color(display-p3 0.84 0.19 0.72 / 0.6) they are first converted to LCH lch(66.93% 68.79 85.94 / 0.4) lch(53.5% 89.35 337.7 / 0.6) then the L and C coordinates (but not H) are premultiplied before interpolation [26.771% 27.516 85.94] and [32.102% 53.61 337.7].

The midpoint of linearly interpolating these, along the 'shorter' hue arc (the default) would be [29.4365% 40.563 31.82] which, with an alpha value of 0.5, is lch(58.873% 81.126 63.64) / 0.5) when premultiplication is undone.

There is sample JavaScript code for alpha premultiplication and un-premultiplication, for both polar and rectangular color spaces, in § 17 Sample code for Color Conversions.

§ 12.4. Hue Interpolation

For color functions with a hue angle (LCH, HSL, HWB etc), there are multiple ways to interpolate. As arcs greater than 360° are rarely desirable, hue angles are fixed up prior to interpolation so that per-component interpolation is done over less than 360°, often less than 180°.

Host syntax can specify any of the following algorithms for hue interpolation (angles in the following are in degrees, but the logic is the same regardless of how they are specified). Specifying a hue interpolation strategy is already part of the <color-interpolation-method> syntax via the <hue-interpolation-method> token.

Both angles and their difference need to be constrained to [0, 360) prior to interpolation. To do this, the minimum number of turns that fit in the lesser angle is added or subtracted from both angles, bringing the lesser angle into the range [0,360); and if the difference between them is

greater than or equal to 360° then the minimum number of turns to bring the difference into the range [0,360) is further subtracted from the greater angle.

Unless otherwise specified, if no specific hue interpolation algorithm is selected by the host syntax, the default is 'shorter'.

▼ TESTS

color-mix-percents-01.html (live test) (source)

color-mix-percents-02.html (live test) (source)

NOTE: As a reminder, if the interpolating colors were not already in the specified interpolation color space, then converting them will turn any powerless components into missing components.

12.4.1. *shorter*

Hue angles are interpolated to take the *shorter* of the two arcs between the starting and ending hues.

EXAMPLE 38 ¶

For example, the midpoint when interpolating in Oklch from a red oklch(0.6 0.24 30) to a yellow oklch(0.8 0.15 90) would be at a hue angle of (30 + 90) / 2 = 60 degrees, along the shorter arc between the two colors, giving a deep orange oklch(0.7 0.195 60)

Angles are adjusted so that $\theta_2 - \theta_1 \in [-180, 180]$. In pseudo-Javascript:

```
if (θ₂ - θ₁ > 180) {
  θ₁ += 360;
}
else if (θ₂ - θ₁ < -180) {
  θ₂ += 360;
}
```

12.4.2. *longer*

Hue angles are interpolated to take the *longer* of the two arcs between the starting and ending hues.

EXAMPLE 39

For example, the midpoint when interpolating in Oklch from a red oklch(0.6 0.24 30) to a yellow oklch(0.8 0.15 90) would be at a hue angle of (30 + 360 + 90) / 2 = 240 degrees, along the longer arc between the two colors, giving a sky blue oklch(0.7 0.195 240)

Angles are adjusted so that $\theta_2 - \theta_1 \in \{(-360, -180], [180, 360)\}$. In pseudo-Javascript:

```
if (0 < θ₂ − θ₁ < 180) {
  θ₁ += 360;
}
else if (−180 < θ₂ − θ₁ <= 0) {
  θ₂ += 360;
}
```

12.4.3. *increasing*

Hue angles are interpolated so that, as they progress from the first color to the second, the angle is always *increasing*.

Depending on the difference between the two angles, this will either look the same as *shorter* or as *longer*. However, if one of the hue angles is being animated, and the hue angle difference passes through 180 degrees, the interpolation will not flip to the other arc.

EXAMPLE 40

For example, the midpoint when interpolating in Oklch from a deep brown oklch(0.5 0.1 30) to a turquoise oklch(0.7 0.1 190) would be at a hue angle of (30 + 190) / 2 = 110 degrees, giving a khaki oklch(0.6 0.1 110).

However, if the hue of second color is animated to oklch(0.7 0.1 230), the midpoint of the interpolation will be (30 + 230) / 2 = 130 degrees, continuing in the same increasing direction, giving another green oklch(0.6 0.1 130) rather than flipping to the opponent color part-way through the animation.

Angles are adjusted so that $\theta_2 - \theta_1 \in [0, 360)$. In pseudo-Javascript:

```
if (θ₂ < θ₁) {
  θ₂ += 360;
}
```

12.4.4. *decreasing*

Hue angles are interpolated so that, as they progress from the first color to the second, the angle is always *decreasing*.

Depending on the difference between the two angles, this will either look the same as *shorter* or as *longer*. However, if one of the hue angles is being animated, and the hue angle difference passes through 180 degrees, the interpolation will not flip to the other arc.

EXAMPLE 41

For example, the midpoint when interpolating in Oklch from a deep brown oklch(0.5 0.1 30) to a turquoise oklch(0.7 0.1 190) would be at a hue angle of (30 + 360 + 190) / 2 = 290 degrees, giving a purple oklch(0.6 0.1 290).

However, if the hue of second color is animated to oklch(0.7 0.1 230), the midpoint of the interpolation will be (30 +360 + 230) / 2 = 310 degrees, continuing in the same decreasing direction, giving another purple oklch(0.6 0.1 310) rather than flipping to the opponent color part-way through the animation.

Angles are adjusted so that $\theta_2 - \theta_1 \in (-360, 0]$. In pseudo-Javascript:

```
if (θ₁ < θ₂) {
  θ₁ += 360;
}
```

13. Gamut Mapping

13.1. An Introduction to Gamut Mapping

NOTE: This section provides important context for the specific requirements described elsewhere in the document.

This section is non-normative

When a color in an origin color space is converted to another, destination color space which has a smaller gamut, some colors will be outside the destination gamut.

For intermediate color calculations, these out of gamut values are preserved. However, if the destination is the display device (a screen, or a printer) then out of gamut values must be converted to an in-gamut color.

Gamut mapping is the process of finding an in-gamut color with the least objectionable change in visual appearance.

13.1.1. Clipping

The simplest and least acceptable method is simply to clip the component values to the displayable range. This changes the proportions of the three primary colors (for an RGB display), resulting in a hue shift.

> EXAMPLE 42
> For example, consider the color `color(srgb-linear 0.5 1 3)`. Because this is a linear-light color space, we can compare the intensities of the three components and see that the amount of blue light is three times the amount of green, while the amount of red light is half that of green. There is six times as much blue primary as red. In Oklch, this color has a hue angle of 265.1°
>
> If we now clip this color to bring it into gamut for sRGB, we get `color(srgb-linear 0.5 1 1)`. The amount of blue light is the same as green. In Oklch, this color has a hue angle of 196.1°, a substantial change of 69°.

13.1.2. Closest Color (MINDE)

A better method is to map colors, in a perceptually uniform color space, by finding the closest in-gamut color (so-called minimum ΔE or ***MINDE***). Clearly, the success of this technique depends on the degree of uniformity of the gamut mapping color space and the predictive accuracy of the deltaE function used.

However, when doing gamut mapping changes in Hue are *particularly* objectionable; changes in Chroma are more tolerable, and small changes in Lightness can also be acceptable especially if the alternative is a larger Chroma reduction. MINDE weights changes in each dimension equally, and thus gives suboptimal results.

13.1.3. Chroma Reduction

To improve on MINDE algorithms, colors are mapped in a perceptually uniform, *polar* color space by holding the hue constant, and reducing the chroma until the color falls in gamut.

EXAMPLE 43

In this example, Display P3 primary yellow (`color(display-p3 1 1 0)`) is being mapped to an sRGB display. The gamut mapping color space is Oklch.

```
color(display-p3 1 1 0)
```

is

```
color(srgb 1 1 -0.3463)
```

which is

```
color(oklch 0.96476 0.24503 110.23)
```

By progressively reducing the chroma component until the resulting color falls inside the sRGB gamut (has no components negative, or greater than one) a gamut mapped color is obtained.

```
color(oklch 0.96476 0.21094 110.23)
```

which is

```
color(srgb 0.99116 0.99733 0.00001)
```

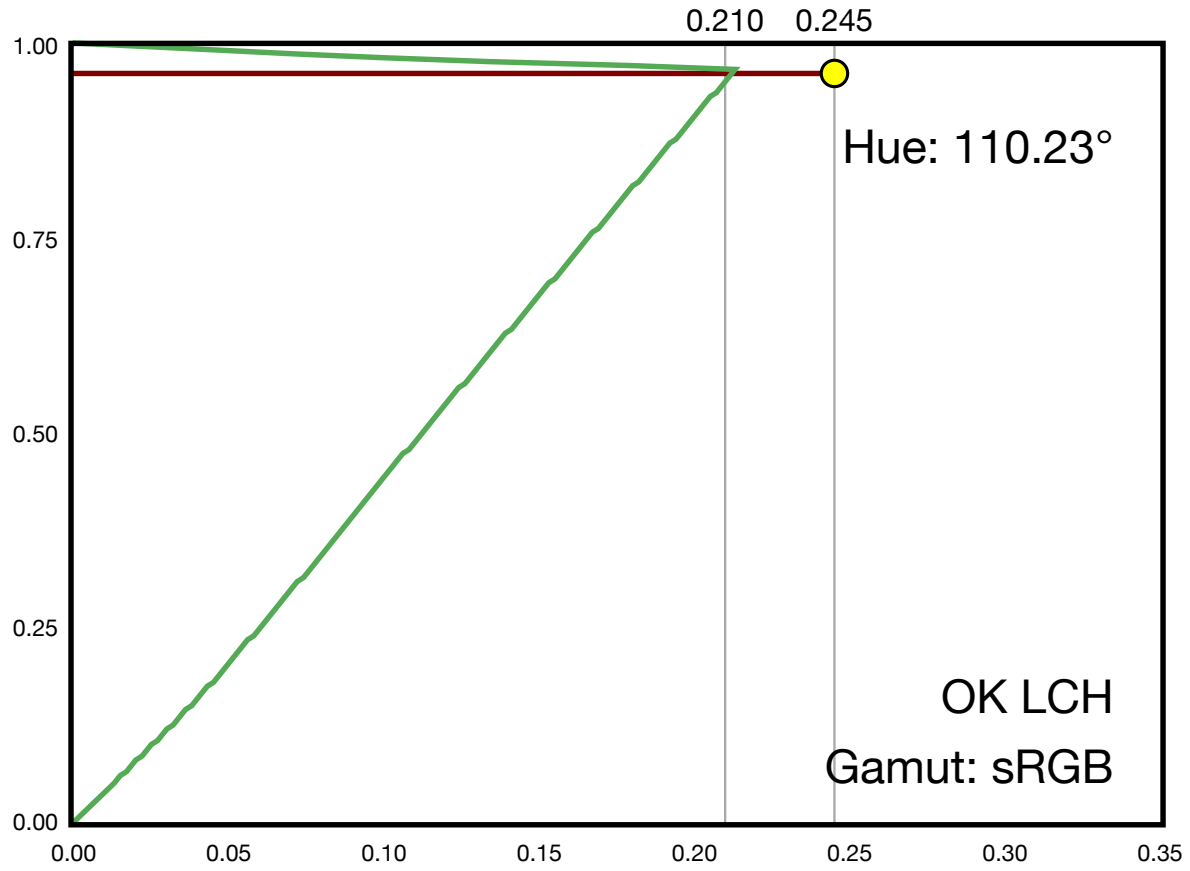


Figure 13 *A constant-hue slice of Oklch color space. The vertical axis represents lightness, the horizontal axis is chroma. The color to be mapped, shown as a yellow circle, has the chroma reduced while keeping hue and lightness constant. The color therefore moves along the maroon line in the diagram, towards the neutral axis on the left. The gamut boundary of sRGB is shown in green.*

A practical implementation will converge more quickly than a linear reduction; either by binary search, or by computing an analytical intersection of the line of constant hue and lightness with the gamut boundary.

§ 13.1.4. Excessive Chroma Reduction

Also, this simple approach will give sub-optimal results for certain colors, principally very light colors like yellow and cyan, if the upper edge of the gamut boundary is shallow, or even slightly concave. The line of constant lightness can skim just above the gamut boundary, resulting in an excessively low chroma in those cases.

The choice of color space will affect the acceptability of the gamut mapped colors.

EXAMPLE 44

In this example, Display P3 primary yellow (`color(display-p3 1 1 0)`) has the chroma progressively reduced in CIE LCH color space.

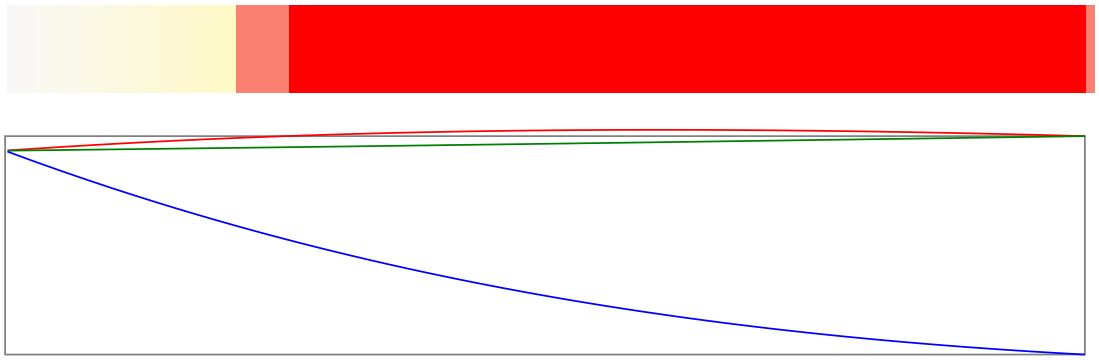


Figure 14 *In the upper part of this diagram, colors which are inside the gamut of sRGB are displayed as-is. Colors inside the gamut of Display P3 (but outside sRGB) are in salmon. Colors outside the gamut of Display P3 are in red. The lower part of the diagram shows the linear-light intensities of the Display P3 red, green and blue components.*

It can be seen that reduction in CIE LCH chroma makes the red intensity curve up, out of Display P3 gamut; by the time it falls again the chroma is very low. Simple gamut mapping in CIE LCH would give unsatisfactory results.

EXAMPLE 45

In this example, Display P3 primary yellow (`color(display-p3 1 1 0)`) has the chroma progressively reduced, but this time in Oklch color space.

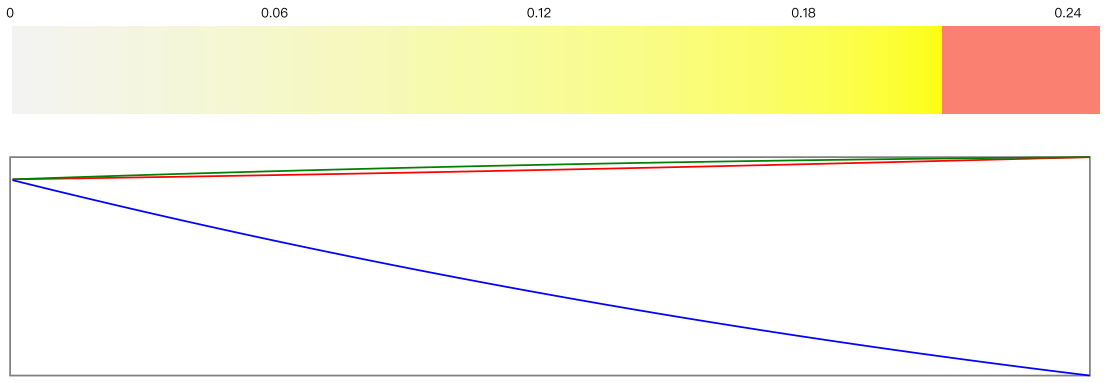


Figure 15 *In the upper part of this diagram, colors which are inside the gamut of sRGB are displayed as-is. Colors inside the gamut of Display P3 (but outside sRGB) are in salmon. Colors outside the gamut of Display P3 are in red. The lower part of the diagram shows the linear-light intensities of the Display P3 red, green and blue components.*

It can be seen that reduction in Oklch chroma is better behaved. Colors do not go outside the Display P3 gamut, and the resulting gamut-mapped yellow has good chroma. Simple gamut mapping in OK LCH would give acceptable results.

13.1.5. Chroma Reduction with Local Clipping

The simple chroma-reduction algorithm can be improved: at each step, the color difference is computed between the current mapped color and a clipped version of that color. If the current color is outside the gamut boundary, but the color difference between it and the clipped version is below the threshold for a *just noticeable difference* (JND), the clipped version of the color is returned as the mapped result. Effectively, this is doing a MINDE mapping at each stage, but constrained so the hue and lightness changes are very small, and thus are not noticeable.

EXAMPLE 46

In this example, Display P3 primary yellow (`color(display-p3 1 1 0)`) has the chroma progressively reduced in CIE LCH color space, with the local clip modification.

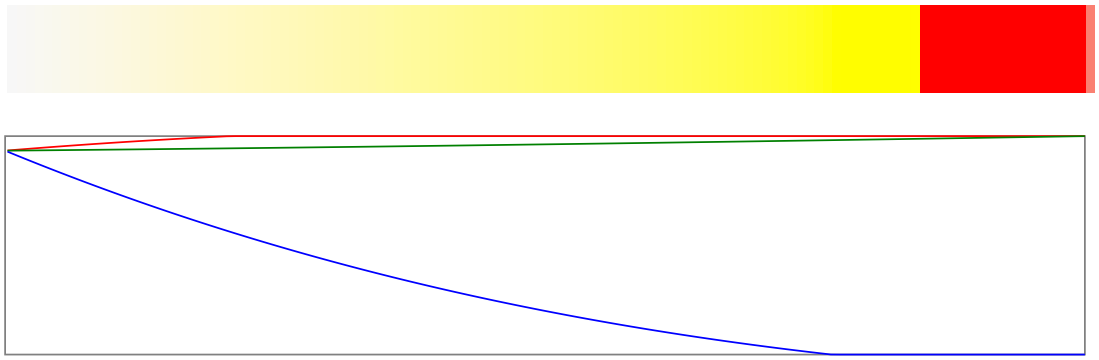


Figure 16 *In the upper part of this diagram, colors which are inside the gamut of sRGB are displayed as-is. Colors inside the gamut of Display P3 (but outside sRGB) are in salmon. Colors outside the gamut of Display P3 are in red. The lower part of the diagram shows the linear-light intensities of the Display P3 red, green and blue components.*

It can be seen that reduction in CIE LCH chroma still makes the red intensity curve up, out of Display P3 gamut; but less than before and the sRGB boundary is found much more quickly. Gamut mapping in CIE LCH with local clip would give acceptable results.

EXAMPLE 47

In this example, Display P3 primary yellow (`color(display-p3 1 1 0)`) has the chroma progressively reduced, but this time in Oklch color space and with the local clip modification.

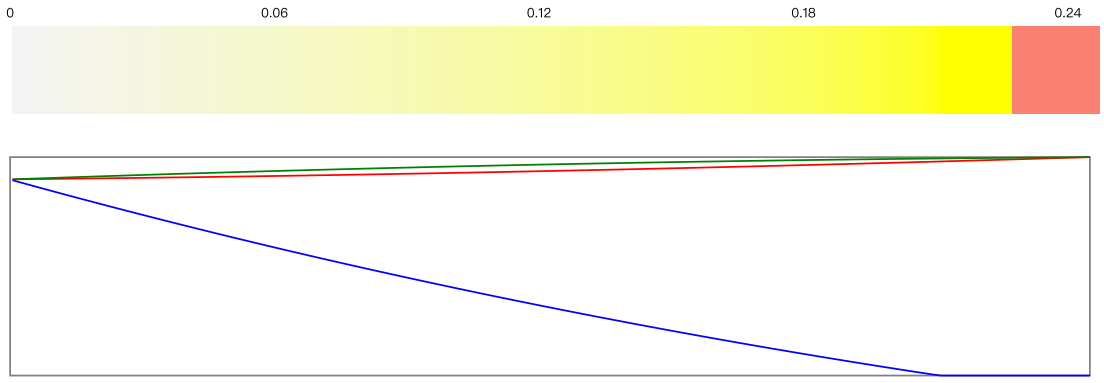


Figure 17 *In the upper part of this diagram, colors which are inside the gamut of sRGB are displayed as-is. Colors inside the gamut of Display P3 (but outside sRGB) are in salmon. Colors outside the gamut of Display P3 are in red. The lower part of the diagram shows the linear-light intensities of the Display P3 red, green and blue components.*

It can be seen that reduction in Oklch chroma, which was already good, is further improved by the local clip modification. Simple gamut mapping in CIE LCH with local clip would give excellent results.

13.1.6. Deviations from Perceptual Uniformity: Hue Curvature

Using the CIE LCH color space and deltaE2000 distance metric, is known to give suboptimal results with significant hue shifts, for colors in the hue range 270° to 330° .

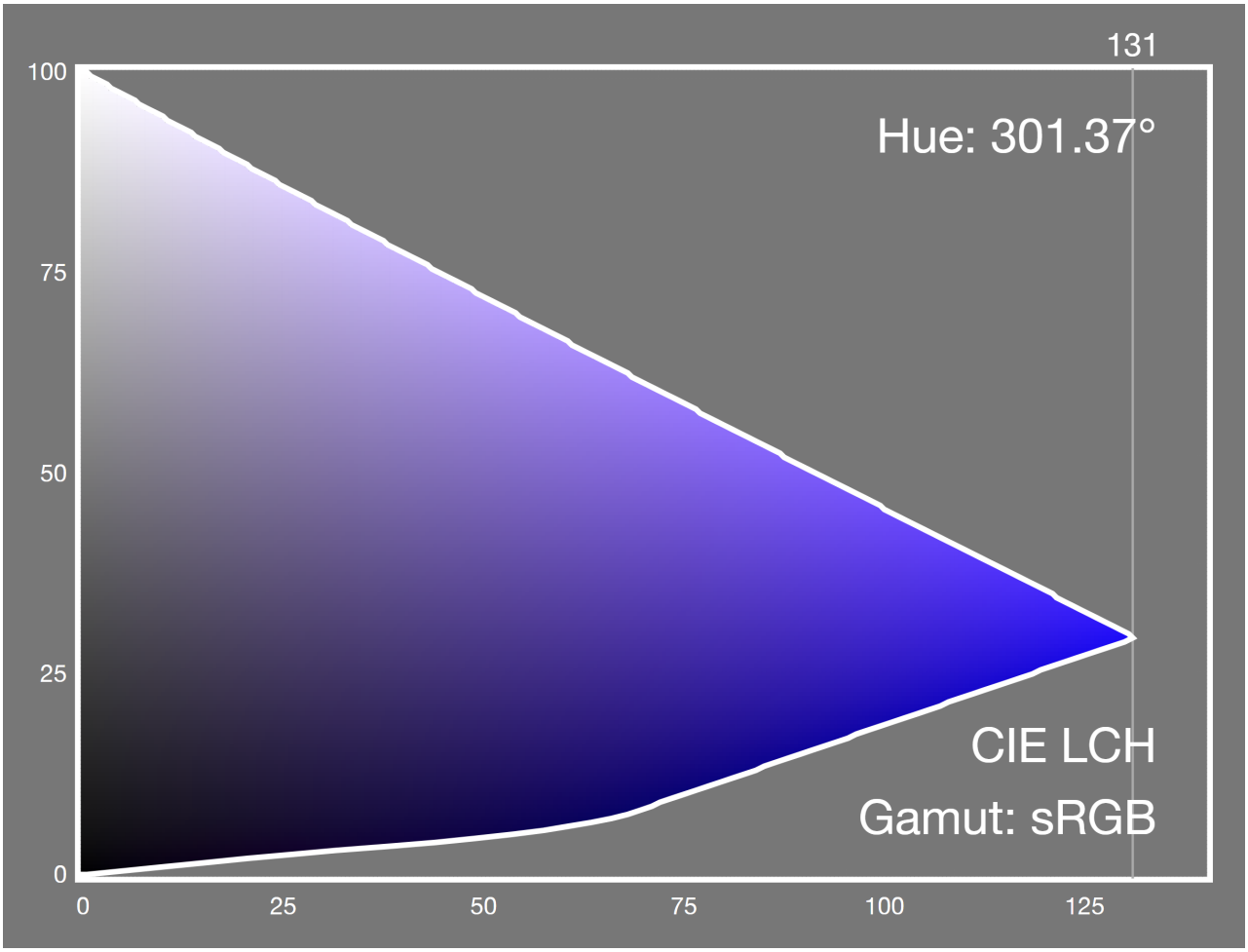


Figure 18 *A constant-hue slice of CIE LCH color space, at a hue angle of 301.37° corresponding to sRGB primary blue. The vertical axis is Lightness, the horizontal axis is Chroma. Between chroma of 25 and 75, the hue is visibly purple, becoming more blue between 100 and 131. The same phenomenon continues past 131, but cannot be shown on an sRGB display.*

Using Oklch color space and deltaEOK distance metric avoids this issue at all hue angles.

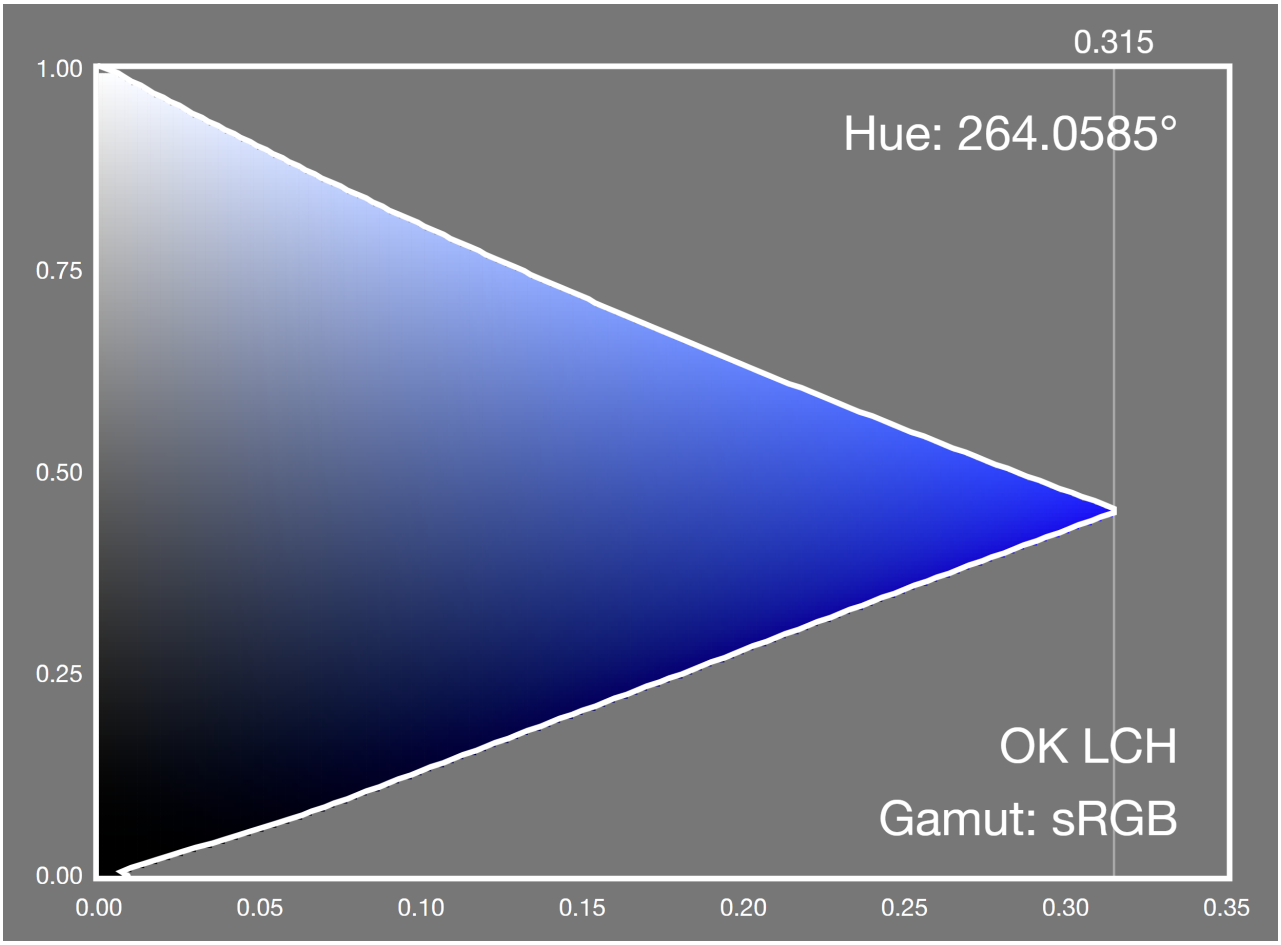


Figure 19 *A constant-hue slice of Oklch color space, at a hue angle of 264.06° corresponding to sRGB primary blue. The vertical axis is Lightness, the horizontal axis is Chroma. The hue is visibly the same at all values of chroma, up to 0.315 (the sRGB limit at this hue). It continues to be constant beyond this point, although that cannot be shown on an sRGB diagram.*

§ 13.2. CSS Gamut Mapping to an RGB Destination

The ***CSS gamut mapping algorithm*** applies to individual, Standard Dynamic Range (SDR) CSS colors which are out of gamut of an RGB display and thus require to be ***css gamut mapped***.

It implements a relative colorimetric intent, and colors inside the destination gamut are unchanged.

> NOTE: other situations, in particular mapping to printer gamuts where the maximum black level is significantly above zero, will require different algorithms which align the respective black and white points, which will result in lightness changes for very light and very dark colors as chroma is reduced..

> NOTE: this algorithm is for individual, distinct colors; for color images, where relationships between neighboring pixels are important and the aim is to preserve detail and texture, a perceptual rendering intent is more appropriate and in that case, colors inside the destination gamut could be changed.

CSS gamut mapping occurs in the Oklch color space, and the color difference formula used is deltaEOK. The local-MINDE improvement is used.

For colors which are out of range on the Lightness axis, white is returned in the destination color space if the Lightness is greater than or equal to 1.0, while black is returned in the destination color space if the Lightness is less than or equal to 0.0.

For the binary search implementation, at each step in the search, the deltaEOK is computed between the current mapped color and a clipped version of that color. If the current color is *outside* the gamut boundary, but the deltaEOK between it and the clipped version is below a threshold for a *just noticeable difference* (JND), the clipped version of the color is returned as the mapped result.

For the analytical implementation, having found the exact intersection, project outwards (towards higher chroma) along the line of constant lightness until the deltaEOK between the projected point and a clipped version of that point exceeds one JND. Then returned the clipped version of the color as the mapped result.

For the Oklch color space, one JND is is an Oklch difference of 0.02.

§ 13.2.1. Sample Pseudocode for the Binary Search Gamut Mapping Algorithm with Local MINDE

To ***CSS gamut map*** a color *origin* in color space *origin color space* to be in gamut of a destination color space *destination*:

1. if *destination* has no gamut limits (XYZ-D65, XYZ-D50, Lab, LCH, Oklab, Oklch) return *origin*
2. let *origin_Oklch* be *origin* converted from *origin color space* to the Oklch color space
3. if the Lightness of *origin_Oklch* is greater than or equal to 100%, return { 1 1 1 origin.alpha } in *destination*
4. if the Lightness of *origin_Oklch* is less than than or equal to 0%, return { 0 0 0 origin.alpha } in *destination*
5. let inGamut(*color*) be a function which returns true if, when passed a color, that color is inside the gamut of *destination*. For HSL and HWB, it returns true if the color is inside the gamut of sRGB.

6. if inGamut(*origin_Oklch*) is true, convert *origin_Oklch* to *destination* and return it as the gamut mapped color
7. otherwise, let delta(*one*, *two*) be a function which returns the deltaEOK of color *one* compared to color *two*
8. let *JND* be 0.02
9. let *epsilon* be 0.0001
10. let clip(*color*)| be a function which converts *color* to *destination*, converts all negative components to zero, converts all components greater that one to one, and returns the result
11. set *min* to zero
12. set *max* to the Oklch chroma of *origin_Oklch*
13. let *min_inGamut* be a boolean that represents when *min* is still in gamut, and set it to true
14. while (*max* - *min* is greater than *epsilon*) repeat the following steps
    1. set *chroma* to (*min* + *max*) /2
    2. set *current* to *origin_Oklch* and then set the chroma component to *chroma*
    3. if *min_inGamut* is true and also if inGamut(*current*) is true, set *min* to *chroma* and continue to repeat these steps
    4. otherwise, if inGamut(*current*) is false carry out these steps:
        1. set *clipped* to clip(*current*)
        2. set *E* to delta(*clipped*, *current*)
        3. if *E* < *JND*
            1. if (*JND* - *E* < *epsilon*) return *clipped* as the gamut mapped color
            2. otherwise,
                1. set *min_inGamut* to false
                2. set *min* to *chroma*
        4. otherwise, set *max* to *chroma* and continue to repeat these steps
15. return *current* as the gamut mapped color

§ 14. Resolving <color> Values

Unless otherwise specified for a particular property, specified colors are resolved to ***computed colors*** and then further to ***used colors*** as described below.

The resolved value of a <color> is its used value.

▼ TESTS

color-computed-hex-color.html (6 tests) (live test) (source)
color-computed-named-color.html (455 tests) (live test) (source)
color-invalid-hex-color.html (10 tests) (live test) (source)
color-invalid-named-color.html (184 tests) (live test) (source)
system-color-compute.html (27 tests) (live test) (source)

§ 14.1. Resolving sRGB values

This applies to:

- hex colors
- 'rgb()' and 'rgba()' values
- 'hsl()' and 'hsla()' values
- 'hwb()' values
- named colors
- system colors
- deprecated-colors

If the sRGB color was explicitly specified by the author as a named color, or as a system color, the specified value is that named or system color, converted to ASCII lowercase. The computed and used value is the corresponding sRGB color, paired with the specified alpha channel (after clamping to [0.0, 1.0]) and defaulting to opaque if unspecified).

EXAMPLE 48 ¶

The author-provided mixed-case form below has a specified value in all lowercase.

```
pUrPlE
purple
```

Otherwise, the specified, computed and used value is the corresponding sRGB color, paired with the specified alpha channel (after clamping to [0.0, 1.0]) and defaulting to opaque if unspecified).

▼ TESTS
color-computed-rgb.html (77 tests) (live test) (source)

EXAMPLE 49

For example, the computed value of

```
hsl(38.824 100% 50%)
```

is

```
rgb(255, 165, 0)
```

14.2. Resolving Lab and LCH values

This applies to 'lab()' and 'lch()' values.

The specified, computed and used value is the corresponding CIE Lab or LCH color (after clamping of L, C and H) paired with the specified alpha channel (as a <number>, not a <percentage>; and defaulting to opaque if unspecified).

EXAMPLE 50

For example, the computed value of

```
lch(52.2345% 72.2 56.2 / 1)
```

is

```
lch(52.2345% 72.2 56.2)
```

14.3. Resolving Oklab and Oklch values

This applies to 'oklab()' and 'oklch()' values.

The specified, computed and used value is the corresponding Oklab or Oklch color (after clamping of L, C and H) paired with the specified alpha channel (as a <number>, not a <percentage>; and defaulting to opaque if unspecified).

EXAMPLE 51

For example, the computed value of

```
oklch(42.1% 0.192 328.6 / 1)
```

is

```
oklch(42.1% 0.192 328.6)
```

§ 14.4. Resolving values of the 'color()' function

The specified, computed and used value is the color in the specified color space, paired with the specified alpha channel (as a <number>, not a <percentage>; and defaulting to opaque if unspecified).

EXAMPLE 52

For example, the computed value of

```
color(display-p3 0.823 0.6554 0.2537 /1)
```

is

```
color(display-p3 0.823 0.6554 0.2537)
```

For colors specified in the 'xyz' color space, which is an alias of the 'xyz-d65' color space, the computed and used value is in the 'xyz-d65' color space.

EXAMPLE 53

For example, the computed value of

```
color(xyz 0.472 0.372 0.131)
```

is

```
color(xyz-d65 0.472 0.372 0.131)
```

§ 14.5. Resolving other colors

This applies to system colors (including the <deprecated-color>s), 'transparent', and 'currentcolor'.

The specified value for each <system-color> keyword is the corresponding color in its color space. However, such colors must not be altered by 'forced colors mode'.

The specified value of 'transparent' is "transparent" while the computed and used value is transparent black.

The 'currentcolor' keyword computes to itself.

In the 'color' property, the used value of 'currentcolor' is the inherited value. In any other property, its used value is the used value of the 'color' property on the same element.

NOTE: This means that if the 'currentcolor' value is inherited, it's inherited as a keyword, not as the value of the 'color' property, so descendants will use their own 'color' property to resolve it.

EXAMPLE 54

For example, given this html:

```
<div>
  <p>Assume this example text is long enough
    to wrap on multiple lines.
  </p>
</div>
```

and this css:

```
div {
  color:  forestgreen;
  text-shadow: currentColor;
}
p {
  color:  mediumseagreen;
}
p::firstline {
  color:  yellowgreen;
}
```

The used value of the inherited property text-shadow on the first line fragment would be yellowgreen.

▼ TESTS

currentcolor-001.html (live test) (source)
currentcolor-002.html (live test) (source)
currentcolor-003.html (live test) (source)

§ 15. Serializing <color> Values

This section updates and replaces that part of CSS Object Model, section Serializing CSS Values, which relates to serializing <color> values.

In this section, the strings used in the specification and the corresponding characters are as follows.

String	Character(s)

" "	U+0020 SPACE
","	U+002C COMMA
"-"	U+002D HYPHEN-MINUS
"."	U+002E FULL STOP
"/"	U+002F SOLIDUS
"none"	U+006E LATIN SMALL LETTER N U+006F LATIN SMALL LETTER O U+006E LATIN SMALL LETTER N U+0065 LATIN SMALL LETTER E

The string "." shall be used as a decimal separator, regardless of locale, and there shall be no thousands separator.

For syntactic forms which support missing color components, the value 'none' (equivalently NONE, nOnE, etc), shall be serialized in all-lower case as the string "none".

§ 15.1. Serializing alpha values

This applies to any <color> value which can take an optional alpha value. It does not apply to the 'opacity' property.

If, after clamping to the range [0.0, 1.0] the alpha is exactly 1, it is omitted from the serialization; an implicit value of 1 (fully opaque) is the default.

If the alpha is any other value than 1, it is explicitly included in the serialization, as a <number>, not a <percentage>. The value is expressed in base ten, with the "." character as decimal separator. The leading zero must not be omitted. Trailing zeroes must be omitted.

> EXAMPLE 55 ¶
>
> For example, an alpha value of 70% will be serialized as the string "0.7" which has a leading zero before the decimal separator, "." as decimal separator (even if the current locale would use some other character, such as ","), and all digits after the "7" would be "0" and are omitted.

The precision with which alpha values are retained, and thus the number of decimal places in the serialized value, is not defined in this specification, but must at least be sufficient to round-trip

integer percentage values. Thus, the serialized value must contain at least two decimal places (unless trailing zeroes have been removed). Values must be rounded towards +∞, not truncated.

> EXAMPLE 56 ¶
>
> For example, an alpha value of 12.3456789% could be serialized as the strings "0.12" or "0.123" or "0.1234" or "0.12346" (rounding the value of 5 towards +∞ because the following digit is 6) or any longer, rounded serialization of the same form.

Because <alpha-value>s which were specified outside the valid range are clamped at parse time, the specified value will be clamped. However, per *CSS Values 4* § 10.12 Range Checking, <alpha-value>s specified using calc() are not clamped when the specified form is serialized; but the computed values are clamped.

> EXAMPLE 57 ¶
>
> For example an alpha value which was specified directly as 120% would be serialized as the string "1". However, if it was specified as calc(2*60%) the specified value would be serialized as the string "calc(1.2)".

§ 15.2. Serializing sRGB values

The serialized form of the following sRGB values:

- hex colors
- 'rgb()' and 'rgba()' values
- 'hsl()' and 'hsla()' values
- 'hwb()' values
- named colors
- system colors
- deprecated-colors
- 'transparent'

is derived from the specified value.

When serializing the value of a property which was set by the author to a CSS named color, a system color, a deprecated-color, or 'transparent' therefore, for the specified value, the ASCII

lowercase keyword value is retained. For the computed and used value, the corresponding sRGB value is used.

Thus, the serialized specified value of ‘transparent’ is the string "transparent", while the serialized computed value of ‘transparent’ is the string "rgba(0, 0, 0, 0)".

For all other sRGB values, the specified, computed and used value is the corresponding sRGB value.

Corresponding sRGB values use either the ‘rgb()’ or ‘rgba()’ form (depending on whether the (clamped) alpha is exactly 1, or not), with all ASCII lowercase letters for the function name.

During serialization, any missing values are converted to 0.

For compatibility, the sRGB component values are serialized in <number> form, not <percentage>. Also for compatibility, the component values are serialized in base 10, with a range of [0-255], regardless of the bit depth with which they are stored.

As noted earlier, unitary alpha values are not explicitly serialized. Also, for compatibility, if the alpha is exactly 1, the ‘rgb()’ form is used, with an implicit alpha; otherwise, the ‘rgba()’ form is used, with an explicit alpha value.

For compatibility, the legacy form with comma separators is used; exactly one ASCII space follows each comma. This includes the comma (not slash) used to separate the blue component of ‘rgba()’ from the alpha value.

EXAMPLE 58 ¶

For example, the serialized value of

```
rgb(29 164 192 / 95%)
```

is the string "rgba(29, 164, 192, 0.95)"

EXAMPLE 59

For example, an author-supplied value:

```
hwb(740deg 20% 30% / 50%)
```

Would be normalized first to

```
hwb(20 20% 30% / 50%)
```

and then converted to sRGB and serialized as

```
rgba(178.5, 93.5, 51, 0.5)
```

The precision of the returned result is described below.

NOTE: contrary to CSS Color 3, the parameters of the 'rgb()' function are of type <number>, not <integer>. Thus, any higher precision than eight bits is indicated with a fractional part.

The precision with which sRGB component values are retained, and thus the number of significant figures in the serialized value, is not defined in this specification, but must at least be sufficient to round-trip eight bit values. Values must be rounded towards $+\infty$, not truncated.

NOTE: authors of scripts which expect color values returned from `getComputedStyle` to have <integer> component values, are advised to update them to also cope with <number>.

EXAMPLE 60

For example,

```
rgb(146.064 107.457 131.223)
```

is now valid, and equal to

```
rgb(57.28% 42.14% 51.46%)
```

A conformant serialized form for both, is the string "rgb(146.06, 107.46, 131.2)".

Trailing fractional zeroes in any component values must be omitted; if the fractional part consists of all zeroes, the decimal point must also be omitted. This means that sRGB colors specified with integer component values will serialize with backwards-compatible integer values.

EXAMPLE 61 ¶

The serialized computed value of

```
'goldenrod'
```

is the string "rgb(218, 165, 32)" and not the string "rgb(218.000, 165.000, 32.000)"

§ 15.3. Serializing Lab and LCH values

The serialized form of 'lch()' and 'lab()' values is derived from the computed value and uses the 'lab()' or 'lch()' forms, with ASCII lowercase letters for the function name.

The component values are serialized in base 10; the L, a, b and C component values are serialized as <number>, using the Lab percentage reference ranges or the LCH percentage reference ranges as appropriate to perform percentage to number conversion; thus 0% L maps to 0 and 100% L maps to 100. A single ASCII space character " " must be used as the separator between the component values.

▼ TESTS

color-computed.html (16 tests) (live test) (source)

EXAMPLE 62 ¶

The serialized value of

```
lab(56.200% 0.000 83.600)
```

is the string "lab(56.2 0 83.6)"

EXAMPLE 63

The serialized value of

```
lab(56.200% 0.000 66.88%)
```

is the string "lab(56.2 0 83.6)"

Trailing fractional zeroes in any component values must be omitted; if the fractional part consists of all zeroes, the decimal point must also be omitted.

EXAMPLE 64

The serialized value of

```
lch(37% 105.0 305.00)
```

is the string "lch(37 105 305)", not "lch(37 105.0 305.00)".

The precision with which 'lab()' component values are retained, and thus the number of significant figures in the serialized value, is not defined in this specification, but due to the wide gamut must be sufficient to round-trip L values between 0 and 100, and a and b values between ±127, with at least sixteen bit precision; this will result in at least three decimal places unless trailing zeroes have been omitted. (half float or float, is recommended for internal storage). Values must be rounded towards +∞, not truncated.

NOTE: a and b values outside ±125 are possible with ultrawide gamut spaces. For example, *all* of the 'prophoto-rgb' primaries and secondaries exceed this range, but are within ±200.

As noted earlier, unitary alpha values are not explicitly serialized. Non-unitary alpha values must be explicitly serialized, and the string " / " (an ASCII space, then forward slash, then another space) must be used to separate the b component value from the alpha value.

EXAMPLE 65

The serialized value of

```
lch(56.2% 83.6 357.4 /93%)
```

is the string "lch(56.2 83.6 357.4 / 0.93)" not "lch(56.2% 83.6 357.4 / 0.93)"

§ 15.4. Serializing Oklab and Oklch values

The serialized form of 'oklch()' and 'oklab()' values is derived from the computed value and uses the 'oklab()' or 'oklch()' forms, with ASCII lowercase letters for the function name.

The component values are serialized in base 10; the L, a, b and C component values are serialized as <number> using the Oklab percentage reference ranges or the Oklch percentage reference ranges as appropriate to perform percentage to number conversion; thus 0% L maps to 0 and 100% L maps to 1.0. A single ASCII space character " " must be used as the separator between the component values.

▼ TESTS

color-computed.html (16 tests) (live test) (source)

EXAMPLE 66

The serialized value of

```
oklab(54.0% -0.10 -0.02)
```

is the string "oklab(0.54 -0.1 -0.02)" not "oklab(54 -0.1 -0.02)" or "oklab(54% -0.1 -0.02)"

EXAMPLE 67

The serialized value of

```
oklab(54.0 -25% -5%)
```

is the string "oklab(0.54 -0.1 -0.02)" not "oklab(54 -0.25 -0.05)"

Trailing fractional zeroes in any component values must be omitted; if the fractional part consists of all zeroes, the decimal point must also be omitted.

EXAMPLE 68 ¶

The serialized value of

```
oklch(56.43% 0.0900 123.40)
```

is the string "oklch(0.5643 0.09 123.4)", not "oklch(0.5643 0.0900 123.40)".

The precision with which ‘oklab()’ component values are retained, and thus the number of significant figures in the serialized value, is not defined in this specification, but due to the wide gamut must be sufficient to round-trip L values between 0 and 1 (0% and 100%), and a, b and C values between ±0.5, with at least sixteen bit precision; this will result in at least five decimal places unless trailing zeroes have been omitted. (half float or float, is recommended for internal storage). Values must be rounded towards +∞, not truncated.

NOTE: a, b and C values outside ±0.5 are possible with ultrawide gamut spaces. For example, the ‘prophoto-rgb’ green and blue primaries exceed this range, with C of 0.526 and 1.413 respectively.

As noted earlier, unitary alpha values are not explicitly serialized. Non-unitary alpha values must be explicitly serialized, and the string " / " (an ASCII space, then forward slash, then another space) must be used to separate the final color component (b, or C) value from the alpha value.

EXAMPLE 69 ¶

The serialized value of

```
oklch(53.85% 0.1725 320.67 / 70%)
```

is the string "oklch(0.5385 0.1725 320.67 / 0.7)"

§ 15.5. Serializing values of the ‘color()’ function

The serialized form of ‘color()’ values is derived from the computed value and uses the ‘color()’ form, with ASCII lowercase letters for the function name and the color space name.

The component values are serialized in base 10, as <number>. A single ASCII space character " " must be used as the separator between the component values, and also between the color space name and the first color component.

▼ TESTS
color-computed.html (16 tests) (live test) (source)

EXAMPLE 70 ¶

The serialized value of

```
color(dIsPlAy-P3  0.964  0.763  0.787)
```

is the string "color(display-p3 0.96 0.76 0.79)", if two decimal places are retained. Notice that 0.787 has rounded up to 0.79, rather than being truncated to 0.78.

Trailing fractional zeroes in any component values must be omitted; if the fractional part consists of all zeroes, the decimal point must also be omitted.

EXAMPLE 71 ¶

The serialized value of

```
color(rec2020 0.400 0.660 0.340)
```

is the string "color(rec2020 0.4 0.66 0.34)", not "color(rec2020 0.400 0.660 0.340)".

If the color space is sRGB, the color space is still explicitly required in the serialized result.

For the predefined color spaces, the *minimum* precision for round-tripping is as follows:

color space	Minimum bits
'srgb'	10
'srgb-linear'	12
'display-p3'	10
'a98-rgb'	10

‘prophoto-rgb’	12
‘rec2020’	12
‘xyz’, ‘xyz-d50’, ‘xyz-d65’	16

(16bit, half-float, or float *per component* is recommended for internal storage). Values must be rounded towards +∞, not truncated.

NOTE: compared to the legacy forms such as ‘rgb()’, ‘hsl()’ and so on, ‘color(srgb)’ has a higher minimum precision requirement. Stylesheet authors who prefer higher precision are thus encouraged to use the ‘color(srgb)’ form.

As noted earlier, unitary alpha values are not explicitly serialized. Non-unitary alpha values must be explicitly serialized, and the string " / " (an ASCII space, then forward slash, then another space) must be used to separate the final color component value from the alpha value.

EXAMPLE 72 ¶

The serialized value of

```
color(prophoto-rgb 0.2804 0.40283 0.42259/85%)
```

is the string "color(prophoto-rgb 0.28 0.403 0.423 / 0.85)", if three decimal places are retained.

§ 15.6. Serializing other colors

This applies to ‘currentcolor’.

The serialized form of this value is derived from the computed value and uses ASCII lowercase letters for the color name.

The serialized form of ‘currentColor’ is the string "currentcolor".

§ 16. Default Style Rules

The following stylesheet is informative, not normative. This style sheet could be used by an implementation as part of its default styling of HTML documents.

```
/* traditional desktop user agent colors for hyperlinks */
:link { color: LinkText; }
:visited { color: VisitedText; }
:active { color: ActiveText; }
```

§ 17. Sample code for Color Conversions

This section is not normative.

For clarity, a library is used for matrix multiplication. (This is more readable than inlining all the multiplies and adds). The matrices are in column-major order.

```
// Sample code for color conversions
// Conversion can also be done using ICC profiles and a Color Management
// For clarity, a library is used for matrix multiplication (multiply-ma

// standard white points, defined by 4-figure CIE x,y chromaticities
const D50 = [0.3457 / 0.3585, 1.00000, (1.0 - 0.3457 - 0.3585) / 0.3585]
const D65 = [0.3127 / 0.3290, 1.00000, (1.0 - 0.3127 - 0.3290) / 0.3290]

// sRGB-related functions

function lin_sRGB(RGB) {
        // convert an array of sRGB values
        // where in-gamut values are in the range [0 - 1]
        // to linear light (un-companded) form.
        // https://en.wikipedia.org/wiki/SRGB
        // Extended transfer function:
        // for negative values,  linear portion is extended on reflectio
        // then reflected power function is used.
        return RGB.map(function (val) {
                let sign = val < 0? -1 : 1;
                let abs = Math.abs(val);

                if (abs < 0.04045) {
                        return val / 12.92;
                }
```

```
                return sign * (Math.pow((abs + 0.055) / 1.055, 2.4));
        });
}

function gam_sRGB(RGB) {
        // convert an array of linear-light sRGB values in the range 0.0
        // to gamma corrected form
        // https://en.wikipedia.org/wiki/SRGB
        // Extended transfer function:
        // For negative values, linear portion extends on reflection
        // of axis, then uses reflected pow below that
        return RGB.map(function (val) {
                let sign = val < 0? -1 : 1;
                let abs = Math.abs(val);

                if (abs > 0.0031308) {
                        return sign * (1.055 * Math.pow(abs, 1/2.4) - 0.
                }

                return 12.92 * val;
        });
}

function lin_sRGB_to_XYZ(rgb) {
        // convert an array of linear-light sRGB values to CIE XYZ
        // using sRGB's own white, D65 (no chromatic adaptation)

        var M = [
                [ 506752 / 1228815,  87881 / 245763,   12673 /   70218 ]
                [  87098 /  409605, 175762 / 245763,   12673 /  175545 ]
                [   7918 /  409605,  87881 / 737289, 1001167 / 1053270 ]
        ];
        return multiplyMatrices(M, rgb);
}

function XYZ_to_lin_sRGB(XYZ) {
        // convert XYZ to linear-light sRGB

        var M = [
                [   12831 /   3959,    -329 /    214, -1974 /   3959 ],
                [ -851781 / 878810, 1648619 / 878810, 36519 / 878810 ],
                [     705 /  12673,   -2585 /  12673,   705 /    667 ],
        ];

        return multiplyMatrices(M, XYZ);
}
```

```
//  display-p3-related functions


function lin_P3(RGB) {
        // convert an array of display-p3 RGB values in the range 0.0 -
        // to linear light (un-companded) form.

        return lin_sRGB(RGB);   // same as sRGB
}

function gam_P3(RGB) {
        // convert an array of linear-light display-p3 RGB  in the range
        // to gamma corrected form

        return gam_sRGB(RGB);   // same as sRGB
}

function lin_P3_to_XYZ(rgb) {
        // convert an array of linear-light display-p3 values to CIE XYZ
        // using  D65 (no chromatic adaptation)
        // http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.h
        var M = [
                [ 608311 / 1250200, 189793 / 714400,  198249 / 1000160 ]
                [  35783 /  156275, 247089 / 357200,  198249 / 2500400 ]
                [      0 /       1,  32229 / 714400, 5220557 / 5000800 ]
        ];

        return multiplyMatrices(M, rgb);
}

function XYZ_to_lin_P3(XYZ) {
        // convert XYZ to linear-light P3
        var M = [
                [ 446124 / 178915, -333277 / 357830, -72051 / 178915 ],
                [ -14852 /  17905,   63121 /  35810,    423 /  17905 ],
                [  11844 / 330415,  -50337 / 660830, 316169 / 330415 ],
        ];

        return multiplyMatrices(M, XYZ);
}

// prophoto-rgb functions

function lin_ProPhoto(RGB) {
        // convert an array of prophoto-rgb values
```

```
		// where in-gamut colors are in the range [0.0 - 1.0]
		// to linear light (un-companded) form.
		// Transfer curve is gamma 1.8 with a small linear portion
		// Extended transfer function
		const Et2 = 16/512;
		return RGB.map(function (val) {
			let sign = val < 0? -1 : 1;
			let abs = Math.abs(val);

			if (abs <= Et2) {
				return val / 16;
			}

			return sign * Math.pow(abs, 1.8);
		});
	}

	function gam_ProPhoto(RGB) {
		// convert an array of linear-light prophoto-rgb  in the range 0
		// to gamma corrected form
		// Transfer curve is gamma 1.8 with a small linear portion
		// TODO for negative values, extend linear portion on reflection
		const Et = 1/512;
		return RGB.map(function (val) {
			let sign = val < 0? -1 : 1;
			let abs = Math.abs(val);

			if (abs >= Et) {
				return sign * Math.pow(abs, 1/1.8);
			}

			return 16 * val;
		});
	}

	function lin_ProPhoto_to_XYZ(rgb) {
		// convert an array of linear-light prophoto-rgb values to CIE X
		// using  D50 (so no chromatic adaptation needed afterwards)
		// http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.h
		var M = [
			[ 0.7977604896723027,  0.13518583717574031,  0.031349349
			[ 0.2880711282292934,  0.7118432178101014,   0.000085653
			[ 0.0,                 0.0,                  0.825104602
		];

		return multiplyMatrices(M, rgb);
```

```
}

function XYZ_to_lin_ProPhoto(XYZ) {
        // convert XYZ to linear-light prophoto-rgb
        var M = [
                [  1.3457989731028281,  -0.25558010007997534,  -0.051106
                [ -0.5446224939028347,   1.5082327413132781,    0.020536
                [  0.0,                  0.0,                   1.211967
        ];

        return multiplyMatrices(M, XYZ);
}

// a98-rgb functions

function lin_a98rgb(RGB) {
        // convert an array of a98-rgb values in the range 0.0 - 1.0
        // to linear light (un-companded) form.
        // negative values are also now accepted
        return RGB.map(function (val) {
                let sign = val < 0? -1 : 1;
                let abs = Math.abs(val);

                return sign * Math.pow(abs, 563/256);
        });
}

function gam_a98rgb(RGB) {
        // convert an array of linear-light a98-rgb  in the range 0.0-1.
        // to gamma corrected form
        // negative values are also now accepted
        return RGB.map(function (val) {
                let sign = val < 0? -1 : 1;
                let abs = Math.abs(val);

                return sign * Math.pow(abs, 256/563);
        });
}

function lin_a98rgb_to_XYZ(rgb) {
        // convert an array of linear-light a98-rgb values to CIE XYZ
        // http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.h
        // has greater numerical precision than section 4.3.5.3 of
        // https://www.adobe.com/digitalimag/pdfs/AdobeRGB1998.pdf
        // but the values below were calculated from first principles
        // from the chromaticity coordinates of R G B W
```

```
        // see matrixmaker.html
        var M = [
                [ 573536 /  994567,  263643 / 1420810,  187206 /  994567
                [ 591459 / 1989134, 6239551 / 9945670,  374412 / 4972835
                [  53769 / 1989134,  351524 / 4972835, 4929758 / 4972835
        ];

        return multiplyMatrices(M, rgb);
}

function XYZ_to_lin_a98rgb(XYZ) {
        // convert XYZ to linear-light a98-rgb
        var M = [
                [ 1829569 /  896150, -506331 /  896150, -308931 /  89615
                [ -851781 /  878810, 1648619 /  878810,   36519 /  87881
                [   16779 / 1248040, -147721 / 1248040, 1266979 / 124804
        ];

        return multiplyMatrices(M, XYZ);
}

//Rec. 2020-related functions

function lin_2020(RGB) {
        // convert an array of rec2020 RGB values in the range 0.0 - 1.0
        // to linear light (un-companded) form.
        // ITU-R BT.2020-2 p.4

        const α = 1.09929682680944 ;
        const β = 0.018053968510807;

        return RGB.map(function (val) {
                let sign = val < 0? -1 : 1;
                let abs = Math.abs(val);

                if (abs < β * 4.5 ) {
                        return val / 4.5;
                }

                return sign * (Math.pow((abs + α -1 ) / α, 1/0.45));
        });
}

function gam_2020(RGB) {
        // convert an array of linear-light rec2020 RGB  in the range 0.
        // to gamma corrected form
```

```
		// ITU-R BT.2020-2 p.4

		const α = 1.09929682680944 ;
		const β = 0.018053968510807;


		return RGB.map(function (val) {
			let sign = val < 0? -1 : 1;
			let abs = Math.abs(val);

			if (abs > β ) {
				return sign * (α * Math.pow(abs, 0.45) - (α - 1)
			}

			return 4.5 * val;
		});
	}

	function lin_2020_to_XYZ(rgb) {
		// convert an array of linear-light rec2020 values to CIE XYZ
		// using  D65 (no chromatic adaptation)
		// http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.h
		var M = [
			[ 63426534 / 99577255,  20160776 / 139408157,  47086771
			[ 26158966 / 99577255, 472592308 / 697040785,   8267143
			[        0 /        1,  19567812 / 697040785, 295819943
		];
		// 0 is actually calculated as  4.994106574466076e-17

		return multiplyMatrices(M, rgb);
	}

	function XYZ_to_lin_2020(XYZ) {
		// convert XYZ to linear-light rec2020
		var M = [
			[  30757411 / 17917100, -6372589 / 17917100, -4539589 /
			[ -19765991 / 29648200, 47925759 / 29648200,   467509 /
			[    792561 / 44930125, -1921689 / 44930125, 42328811 /
		];

		return multiplyMatrices(M, XYZ);
	}

	// Chromatic adaptation

	function D65_to_D50(XYZ) {
```

```
        // Bradford chromatic adaptation from D65 to D50
        // The matrix below is the result of three operations:
        // - convert from XYZ to retinal cone domain
        // - scale components from one reference white to another
        // - convert back to XYZ
        // http://www.brucelindbloom.com/index.html?Eqn_ChromAdapt.html
        var M =  [
                [  1.0479298208405488,    0.022946793341019088,  -0.0501
                [  0.029627815688159344,  0.990434484573249,     -0.0170
                [ -0.009243058152591178,  0.015055144896577895,   0.7518
        ];

        return multiplyMatrices(M, XYZ);
}

function D50_to_D65(XYZ) {
        // Bradford chromatic adaptation from D50 to D65
        var M = [
                [  0.9554734527042182,   -0.023098536874261423,  0.06325
                [ -0.028369706963208136,  1.0099954580058226,    0.02104
                [  0.012314001688319899, -0.020507696433477912,  1.33036
        ];

        return multiplyMatrices(M, XYZ);
}

// CIE Lab and LCH

function XYZ_to_Lab(XYZ) {
        // Assuming XYZ is relative to D50, convert to CIE Lab
        // from CIE standard, which now defines these as a rational frac
        var ε = 216/24389;  // 6^3/29^3
        var κ = 24389/27;   // 29^3/3^3

        // compute xyz, which is XYZ scaled relative to reference white
        var xyz = XYZ.map((value, i) => value / D50[i]);

        // now compute f
        var f = xyz.map(value => value > ε ? Math.cbrt(value) : (κ * val

        return [
                (116 * f[1]) - 16,       // L
                500 * (f[0] - f[1]), // a
                200 * (f[1] - f[2])  // b
        ];
        // L in range [0,100]. For use in CSS, add a percent
```

```
}

function Lab_to_XYZ(Lab) {
        // Convert Lab to D50-adapted XYZ
        // http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.h
        var κ = 24389/27;   // 29^3/3^3
        var ε = 216/24389;  // 6^3/29^3
        var f = [];

        // compute f, starting with the luminance-related term
        f[1] = (Lab[0] + 16)/116;
        f[0] = Lab[1]/500 + f[1];
        f[2] = f[1] - Lab[2]/200;

        // compute xyz
        var xyz = [
                Math.pow(f[0],3) > ε ?   Math.pow(f[0],3)            : (
                Lab[0] > κ * ε ?         Math.pow((Lab[0]+16)/116,3) : L
                Math.pow(f[2],3)  > ε ?  Math.pow(f[2],3)            : (
        ];

        // Compute XYZ by scaling xyz by reference white
        return xyz.map((value, i) => value * D50[i]);
}

function Lab_to_LCH(Lab) {
        // Convert to polar form
        var hue = Math.atan2(Lab[2], Lab[1]) * 180 / Math.PI;
        return [
                Lab[0], // L is still L
                Math.sqrt(Math.pow(Lab[1], 2) + Math.pow(Lab[2], 2)), //
                hue >= 0 ? hue : hue + 360 // Hue, in degrees [0 to 360)
        ];
}

function LCH_to_Lab(LCH) {
        // Convert from polar form
        return [
                LCH[0], // L is still L
                LCH[1] * Math.cos(LCH[2] * Math.PI / 180), // a
                LCH[1] * Math.sin(LCH[2] * Math.PI / 180) // b
        ];
}

// OKLab and OKLCH
// https://bottosson.github.io/posts/oklab/
```

```
// XYZ <-> LMS matrices recalculated for consistent reference white
// see https://github.com/w3c/csswg-drafts/issues/6642#issuecomment-9435

function XYZ_to_OKLab(XYZ) {
        // Given XYZ relative to D65, convert to OKLab
        var XYZtoLMS = [
                [ 0.8190224432164319,    0.3619062562801221,   -0.128873
                [ 0.0329836671980271,    0.9292868468965546,    0.03614
                [ 0.048177199566046255,  0.26423952494422764,   0.63354
        ];
        var LMStoOKLab = [
                [  0.2104542553,   0.7936177850,  -0.0040720468 ],
                [  1.9779984951,  -2.4285922050,   0.4505937099 ],
                [  0.0259040371,   0.7827717662,  -0.8086757660 ]
        ];

        var LMS = multiplyMatrices(XYZtoLMS, XYZ);
        return multiplyMatrices(LMStoOKLab, LMS.map(c => Math.cbrt(c)));
        // L in range [0,1]. For use in CSS, multiply by 100 and add a p
}

function OKLab_to_XYZ(OKLab) {
        // Given OKLab, convert to XYZ relative to D65
        var LMStoXYZ =  [
                [  1.2268798733741557,  -0.5578149965554813,   0.2813910
                [ -0.04057576262431372,  1.1122868293970594,  -0.0717110
                [ -0.07637294974672142, -0.4214933239627914,   1.5869240
        ];
        var OKLabtoLMS = [
        [ 0.9999999845051981432,  0.3963377921737678 5678,   0.215803758
        [ 1.0000000088817607767,  -0.1055613423236563494,   -0.063854174
        [ 1.0000000546724109177,  -0.08948418209496575 9684, -1.291485537
    ];

        var LMSnl = multiplyMatrices(OKLabtoLMS, OKLab);
        return multiplyMatrices(LMStoXYZ, LMSnl.map(c => c ** 3));
}

function OKLab_to_OKLCH(OKLab) {
        var hue = Math.atan2(OKLab[2], OKLab[1]) * 180 / Math.PI;
        return [
                OKLab[0], // L is still L
                Math.sqrt(OKLab[1] ** 2 + OKLab[2] ** 2), // Chroma
                hue >= 0 ? hue : hue + 360 // Hue, in degrees [0 to 360)
        ];
```

```
}

function OKLCH_to_OKLab(OKLCH) {
        return [
                OKLCH[0], // L is still L
                OKLCH[1] * Math.cos(OKLCH[2] * Math.PI / 180), // a
                OKLCH[1] * Math.sin(OKLCH[2] * Math.PI / 180)  // b
        ];
}

// Premultiplied alpha conversions

function rectangular_premultiply(color, alpha) {
// given a color in a rectangular orthogonal colorspace
// and an alpha value
// return the premultiplied form
        return color.map((c) => c * alpha)
}

function rectangular_un_premultiply(color, alpha) {
// given a premultiplied color in a rectangular orthogonal colorspace
// and an alpha value
// return the actual color
        if (alpha === 0) {
                return color; // avoid divide by zero
        }
        return color.map((c) => c / alpha)
}

function polar_premultiply(color, alpha, hueIndex) {
        // given a color in a cylindicalpolar colorspace
        // and an alpha value
        // return the premultiplied form.
        // the index says which entry in the color array corresponds to
        // for example, in OKLCH it would be 2
        // while in HSL it would be 0
        return color.map((c, i) => c * (hueIndex === i? 1 : alpha))
}

function polar_un_premultiply(color, alpha, hueIndex) {
        // given a color in a cylindicalpolar colorspace
        // and an alpha value
        // return the actual color.
        // the hueIndex says which entry in the color array corresponds
        // for example, in OKLCH it would be 2
        // while in HSL it would be 0
```

```
        if (alpha === 0) {
                return color; // avoid divide by zero
        }
        return color.map((c, i) => c / (hueIndex === i? 1 : alpha))
}

// Convenience functions can easily be defined, such as
function hsl_premultiply(color, alpha) {
        return polar_premultiply(color, alpha, 0);
}
```

§ 18. Sample Code for ΔE2000 and ΔEOK Color Differences

This section is not normative.

§ 18.1. ΔE2000

The simplest color difference metric, ΔE76, is simply the Euclidean distance in Lab color space. While this is a good first approximation, color-critical industries such as printing and fabric dyeing soon developed improved formulae. Currently, the most widely used formula is ΔE2000. It corrects a number of known asymmetries and non-linearities compared to ΔE76. Because the formula is complex, and critically dependent on the sign of various intermediate calculations, implementations are often incorrect [Sharma].

The sample code below has been validated to five significant figures against the test suite of paired Lab values and expected ΔE2000 published by [Sharma] and is correct.

```
// deltaE2000 is a statistically significant improvement
// over deltaE76 and deltaE94,
// and is recommended by the CIE and Idealliance
// especially for color differences less than 10 deltaE76
// but is wicked complicated
// and many implementations have small errors!

/**
 * @param {number[]} reference – Array of CIE Lab values: L as 0..100, a
 * @param {number[]} sample – Array of CIE Lab values: L as 0..100, a an
 * @return {number} How different a color sample is from reference
 */

function deltaE2000 (reference, sample) {
```

```
// Given a reference and a sample color,
// both in CIE Lab,
// calculate deltaE 2000.

// This implementation assumes the parametric
// weighting factors kL, kC and kH
// (for the influence of viewing conditions)
// are all 1, as seems typical.

let [L1, a1, b1] = reference;
let [L2, a2, b2] = sample;
let C1 = Math.sqrt(a1 ** 2 + b1 ** 2);
let C2 = Math.sqrt(a2 ** 2 + b2 ** 2);

    let Cbar = (C1 + C2)/2; // mean Chroma

    // calculate a-axis asymmetry factor from mean Chroma
    // this turns JND ellipses for near-neutral colors back into cir
    let C7 = Math.pow(Cbar, 7);
    const Gfactor = Math.pow(25, 7);
    let G = 0.5 * (1 - Math.sqrt(C7/(C7+Gfactor)));

    // scale a axes by asymmetry factor
    // this by the way is why there is no Lab2000 color space
    let adash1 = (1 + G) * a1;
    let adash2 = (1 + G) * a2;

    // calculate new Chroma from scaled a and original b axes
    let Cdash1 = Math.sqrt(adash1 ** 2 + b1 ** 2);
    let Cdash2 = Math.sqrt(adash2 ** 2 + b2 ** 2);

    // calculate new hues, with zero hue for true neutrals
    // and in degrees, not radians
    const π = Math.PI;
    const r2d = 180 / π;
    const d2r = π / 180;
    let h1 = (adash1 === 0 && b1 === 0)? 0: Math.atan2(b1, adash1);
    let h2 = (adash2 === 0 && b2 === 0)? 0: Math.atan2(b2, adash2);

    if (h1 < 0) {
            h1 += 2 * π;
    }
    if (h2 < 0) {
            h2 += 2 * π;
    }
```

```
h1 *= r2d;
h2 *= r2d;

// Lightness and Chroma differences; sign matters
let ΔL = L2 - L1;
let ΔC = Cdash2 - Cdash1;

// Hue difference, taking care to get the sign correct
let hdiff = h2 - h1;
let hsum = h1 + h2;
let habs = Math.abs(hdiff);
let Δh;

if (Cdash1 * Cdash2 === 0) {
        Δh = 0;
}
else if (habs <= 180) {
        Δh = hdiff;
}
else if (hdiff > 180) {
        Δh = hdiff - 360;
}
else if (hdiff < -180) {
        Δh = hdiff + 360;
}
else {
        console.log("the unthinkable has happened");
}

// weighted Hue difference, more for larger Chroma
let ΔH = 2 * Math.sqrt(Cdash2 * Cdash1) * Math.sin(Δh * d2r / 2)

// calculate mean Lightness and Chroma
let Ldash = (L1 + L2)/2;
let Cdash = (Cdash1 + Cdash2)/2;
let Cdash7 = Math.pow(Cdash, 7);

// Compensate for non-linearity in the blue region of Lab.
// Four possibilities for hue weighting factor,
// depending on the angles, to get the correct sign
let hdash;
if (Cdash1 == 0 && Cdash2 == 0) {
        hdash = hsum;   // which should be zero
}
else if (habs <= 180) {
```

```
        hdash = hsum / 2;
}
else if (hsum < 360) {
        hdash = (hsum + 360) / 2;
}
else {
        hdash = (hsum - 360) / 2;
}

// positional corrections to the lack of uniformity of CIELAB
// These are all trying to make JND ellipsoids more like spheres

// SL Lightness crispening factor
// a background with L=50 is assumed
let lsq = (Ldash - 50) ** 2;
let SL = 1 + ((0.015 * lsq) / Math.sqrt(20 + lsq));

// SC Chroma factor, similar to those in CMC and deltaE 94 formu
let SC = 1 + 0.045 * Cdash;

// Cross term T for blue non-linearity
let T = 1;
T -= (0.17 * Math.cos((     hdash - 30)  * d2r));
T += (0.24 * Math.cos(  2 * hdash        * d2r));
T += (0.32 * Math.cos(((3 * hdash) + 6)  * d2r));
T -= (0.20 * Math.cos(((4 * hdash) - 63) * d2r));

// SH Hue factor depends on Chroma,
// as well as adjusted hue angle like deltaE94.
let SH = 1 + 0.015 * Cdash * T;

// RT Hue rotation term compensates for rotation of JND ellipses
// and Munsell constant hue lines
// in the medium-high Chroma blue region
// (Hue 225 to 315)
let Δθ = 30 * Math.exp(-1 * (((hdash - 275)/25) ** 2));
let RC = 2 * Math.sqrt(Cdash7/(Cdash7 + Gfactor));
let RT = -1 * Math.sin(2 * Δθ * d2r) * RC;

// Finally calculate the deltaE, term by term as root sum of squ
let dE = (ΔL / SL) ** 2;
dE += (ΔC / SC) ** 2;
dE += (ΔH / SH) ** 2;
dE += RT * (ΔC / SC) * (ΔH / SH);
```

```
        return Math.sqrt(dE);
        // Yay!!!
};
```

§ 18.2. ΔEOK

Because Oklab does not suffer from the hue linearity, hue uniformity, and chroma non-linearities of CIE Lab, the color difference metric does not need to correct for them and so is simply the Euclidean distance in Oklab color space.

```
// Calculate deltaE OK
// simple root sum of squares
/**
 * @param {number[]} reference – Array of OKLab values: L as 0..1, a and
 * @param {number[]} sample – Array of OKLab values: L as 0..1, a and b
 * @return {number} How different a color sample is from reference
 */
function deltaEOK (reference, sample) {
    let [L1, a1, b1] = reference;
        let [L2, a2, b2] = sample;
        let ΔL = L1 - L2;
        let Δa = a1 - a2;
        let Δb = b1 - b2;
        return Math.sqrt(ΔL ** 2 + Δa ** 2 + Δb ** 2);
}
```

§ Appendix A: Deprecated CSS System Colors

Earlier versions of CSS defined several additional system colors. These color keywords have been **deprecated**, however, as they are insufficient for their original purpose (making website elements look like their native OS counterparts), represent a security risk by making it easier for a webpage to "spoof" a native OS dialog, and increase fingerprinting surface, compromising user privacy.

User agents must support these keywords, and to mitigate fingerprinting must map them to the (undeprecated) system colors as listed below. **Authors must not use these keywords.**

The deprecated system colors are represented as the *'<deprecated-color>'* sub-type, and are defined as:

ActiveBorder
Active window border. Same as 'ButtonBorder'.

ActiveCaption
Active window caption. Same as 'CanvasText'.

AppWorkspace
Background color of multiple document interface. Same as 'Canvas'.

Background
Desktop background. Same as 'Canvas'.

ButtonHighlight
The color of the border facing the light source for 3-D elements that appear 3-D due to one layer of surrounding border. Same as 'ButtonFace'.

ButtonShadow
The color of the border away from the light source for 3-D elements that appear 3-D due to one layer of surrounding border. Same as 'ButtonFace'.

CaptionText
Text in caption, size box, and scrollbar arrow box. Same as 'CanvasText'.

InactiveBorder
Inactive window border. Same as 'ButtonBorder'.

InactiveCaption
Inactive window caption. Same as 'Canvas'.

InactiveCaptionText
Color of text in an inactive caption. Same as 'GrayText'.

InfoBackground
Background color for tooltip controls. Same as 'Canvas'.

InfoText
Text color for tooltip controls. Same as 'CanvasText'.

Menu
Menu background. Same as 'Canvas'.

MenuText
Text in menus. Same as 'CanvasText'.

Scrollbar
Scroll bar gray area. Same as 'Canvas'.

ThreeDDarkShadow
The color of the darker (generally outer) of the two borders away from the light source for 3-D elements that appear 3-D due to two concentric layers of surrounding border. Same as 'ButtonBorder'.

ThreeDFace
The face background color for 3-D elements that appear 3-D due to two concentric layers of surrounding border. Same as 'ButtonFace'.

ThreeDHighlight
: The color of the lighter (generally outer) of the two borders facing the light source for 3-D elements that appear 3-D due to two concentric layers of surrounding border. Same as 'ButtonBorder'.

ThreeDLightShadow
: The color of the darker (generally inner) of the two borders facing the light source for 3-D elements that appear 3-D due to two concentric layers of surrounding border. Same as 'ButtonBorder'.

ThreeDShadow
: The color of the lighter (generally inner) of the two borders away from the light source for 3-D elements that appear 3-D due to two concentric layers of surrounding border. Same as 'ButtonBorder'.

Window
: Window background. Same as 'Canvas'.

WindowFrame
: Window frame. Same as 'ButtonBorder'.

WindowText
: Text in windows. Same as 'CanvasText'.

▼ TESTS

Results on these 'same as' equivalence tests are not great, which is why the feature is at-risk

deprecated-sameas-001.html	(live test)	(source)
deprecated-sameas-002.html	(live test)	(source)
deprecated-sameas-003.html	(live test)	(source)
deprecated-sameas-004.html	(live test)	(source)
deprecated-sameas-005.html	(live test)	(source)
deprecated-sameas-006.html	(live test)	(source)
deprecated-sameas-007.html	(live test)	(source)
deprecated-sameas-008.html	(live test)	(source)
deprecated-sameas-009.html	(live test)	(source)
deprecated-sameas-010.html	(live test)	(source)
deprecated-sameas-011.html	(live test)	(source)
deprecated-sameas-012.html	(live test)	(source)
deprecated-sameas-013.html	(live test)	(source)
deprecated-sameas-014.html	(live test)	(source)
deprecated-sameas-015.html	(live test)	(source)
deprecated-sameas-016.html	(live test)	(source)
deprecated-sameas-017.html	(live test)	(source)
deprecated-sameas-018.html	(live test)	(source)
deprecated-sameas-019.html	(live test)	(source)

deprecated-sameas-020.html	(live test)	(source)
deprecated-sameas-021.html	(live test)	(source)
deprecated-sameas-022.html	(live test)	(source)
deprecated-sameas-023.html	(live test)	(source)

§ Appendix B: Deprecated Quirky Hex Colors

When CSS is being parsed in quirks mode, *'<quirky-color>'* is a type of <color> that is only valid in certain properties:

- 'background-color'
- 'border-color'
- 'border-top-color'
- 'border-right-color'
- 'border-bottom-color'
- 'border-left-color'
- 'color'

It is *not* valid in properties that include or reference these properties, such as the 'background' shorthand, or inside functional notations such as 'color-mix()'

Additionally, while <quirky-color> must be valid as a <color> when parsing the affected properties in the '@supports' rule, it is *not* valid for those properties when used in the `CSS.supports()` method.

A <quirky-color> can be represented as a <number-token>, <dimension-token>, or <ident-token>, according to the following rules:

- If it's an <ident-token>, the token's representation must contain exactly 3 or 6 characters, all hexadecimal digits. It represents a <hex-color> with the same value.
- If it's a <number-token>, it must have its integer flag set.

  Serialize the integer's value. If the serialization has less than 6 characters, prepend "0" characters to it until it is 6 characters long. It represents a <hex-color> with the same value.
- If it's a <dimension-token>, it must have its integer flag set.

  Serialize the integer's value, and append the representation of the token's unit. If the result has less than 6 characters, prepend "0" characters to it until it is 6 characters long. It represents a <hex-color> with the same value.

(In other words, Quirks Mode allows hex colors to be written without the leading "#", but with weird parsing rules.)

§ Acknowledgments

In addition to those who contributed to CSS Color 3, the editors would like to thank Emilio Cobos Álvarez, Chris Bai, Amelia Bellamy-Royds, Lars Borg, Mike Bremford, Andreu Botella, Dan Burzo, Max Derhak, fantasai, Simon Fraser, Devon Govett, Phil Green, Dean Jackson, Andreas Kraushaar, Pierre-Anthony Lemieux, Cameron McCormack, Romain Menke, Chris Murphy, Isaac Muse, Jonathan Neal, Chris Needham, Christoph Päper, Brad Pettit, Xidorn Quan, Craig Revie, Melanie Richards, Florian Rivoal, Jacob Rus, Joseph Salowey, Simon Sapin, Igor Snitkin, Lea Verou, Mark Watson, Sam Weinig, and Natalie Weizenbaum.

§ Changes

§ Changes since the Candidate Recommendation Draft of 1 November 2022

- Added descriptions and examples for hue interpolation keywords
- Use normative prose for achromatic HWB colors
- Corrected hue interpolation angle range; [0,360) not [0,360]
- Expressed that displaying as black or white when L=0% or 100% is due to gamut mapping. Removed incorrect assertions of powerlessness
- Dropped the confusing "representing black" and "representing white" comments
- Clarified that opponent a and b are analogous
- Specified RGB channels using reference ranges rather than prose, for consistency
- Explicitly referenced percent reference ranges for percentage to number conversion when serializing Lab, LCH, Oklab, Oklch
- Required Oklab interpolation, remove previous "may", describe explicit opt-out
- Labelled the Lab, LCH, Oklab and Oklch tutorial sections as non-normative. Moved some definitions out of the non-normative section.
- Clarified that, when interpolating, checking for analogous components happens before color space conversion
- Back-ported hwb() syntax changes and reference ranges from CSS Color 5
- Defined carry-forward operations must happen before powerless operations
- Clarified it is *color* components which must be all-number or all-percentage, in legacy rgb() syntax

- Clarified for legacy syntax that color components must be all-percentage or all-number
- Added examples of specified out of range alpha, with and without calc()
- Placed examples of serializing with trimmed trailing zeroes colorer to the relevant text
- clarified example, used value of text-shadow
- Clarified resolving currentColor
- Updated acknowledgments
- Stop claiming that achromatic colors have missing a,b, or chroma
- HSL and HWB changed to unbounded gamut, to promote round-tripping
- Defined percentage reference range for HSL
- Modern color syntax hsl() and hsla() allow mixed number and percentage components
- Modern color syntax rgb() and rgba() allow mixed number and percentage components
- Define the term "modern color syntax" (legacy color syntax already defined).
- Consistently use the term "analogous components"
- Changed to allow all predefined color spaces for interpolation
- Clarified that for color(), three parameters (RGB or XYZ) are required
- Clarified serialization of named colors, system colors, and transparent
- Define specified value for Lab, LCH, Oklab, Oklch
- Define specified value for other sRGB colors
- Define specified values for named and system colors
- Clamp alpha, Lightness, Chroma and Hue at parsed-value time
- Remove passing mention of specular white and CIE Lightness
- No longer require as-specified Hue to be retained; clamp to [0, 360]
- Consistent serialization of Lightness and number in examples
- Minor typos and editorial clarifications

§ Changes since the Candidate Recommendation of 5 July 2022

- Removed hue interpolation "specified" value
- Defined hue interpolation angle more precisely, maintaining differences of 360deg
- Added example of carried forward alpha for premultiplication
- Clarified a,b and C,h powerless at L=100% representing white.
- Removed handwavy mention of L=400 which applies to hdr-CIELAB not CIE Lab

- Consistent capitalization of Oklab and Oklch
- Moved definitions of valid color, invalid color, out of gamut and in gamut to terminology section
- Fixed definition of "longer" hue interpolation
- Further clarified the concept of a host syntax
- Accessibility improvements for color swatches
- Made explicit that legacy forms do not support "none"
- Remove "none" from the hue production, as it is not allowed in legacy syntax
- Removed some dangling references to CMYK and CMYKOGV, moved to CSS Color5
- Clarified how missing values in colors to be interpolated are carried forward
- Updated syntax of xyz-params so they take numbers and percentage, to align with prose
- Ensure all examples and figures have IDs, self-links
- Clarified importance to implementors of reading the gamut mapping introduction
- Removed left-over mention of custom color spaces (feature was moved to CSS Color 5)
- Refactor syntax of <color> and <alpha-value>
- Editorial refactoring for better reading order.
- Updated pseudocode for gamut mapping algorithm, remove un-needed deltaE calls

§ Changes since the Working Draft of 28 June 2022

- Updated status for Candidate Recommendation

§ Changes since the Working Draft of 28 April 2022

- Moved opacity property up to the top of the module, next to color property, before getting into details.
- Improved description of the color property, in particular effect on other properties
- Corrected longer hue adjust equation, for equal-modulo-360 colors
- Added two new System colors: AccentColor and AccentColorText
- Described overall color space conversion steps in new section
- Accounted for ‘none’ alpha in premultiplication and un-premultiplication

§ Changes since the Working Draft of 15 December 2021

- Made system colors fully resolve, but forbid their alteration in forced colors mode
- Removed forgiveness for incorrect number of parameters in color() function
- Changed serialization of CIE Lightness and OK Lightness to number rather than percentage.
- Marked deprecated system color equivalences as at-risk
- Added reference ranges to percentage values for CIE and OK L,a,b,C
- Noted that there is sample code for performing and undoing premultiplication, for both rectangular and polar color spaces.
- Added out of range clamping to the gamut mapping prose, as well as the pseudocode
- Added normative reference for ProPhoto RGB / ROMM
- Corrected sRGB and Display P3 black point value for reference surround
- Added normative reference for Display P3
- Avoided an infinite loop in gamut reduction, with colors whiter than white or darker than black
- Clarified serialization of the 'none' value
- Clarified the opt-in to interpolation in Oklab, for non-legacy colors
- Defined how premultiplication works, with the 'none' value
- Clarified that missing values in rgb serialize as 0
- Clarified the use of calc() with the 'none' value
- Typo, inconsistent casing on System Colors
- Added example of SelectedItem with SelectedItemText
- Explicitly noted the presence or absence of legacy colors
- Added normative reference for CIE XYZ
- Added normative reference for HWB and HSL
- Clarified that 'hwb()' is not a legacy syntax so does not support the older, comma-separated syntactic form
- Clarified that only legacy colors will gamut map, the others are unbounded
- Use distinct terms, spectrophotometer and spectroradiometer
- Assorted minor typos fixed, and grammatical improvements

§ Changes since the Working Draft of 1 June 2021

- Added gamut mapping section and defined a CSS gamut mapping algorithm as chroma reduction in Oklch with local MINDE.
- Computed value of color(xyz ...) is color(xyz-d65 ...)
- Added srgb-linear to interpolation color spaces
- Updated Changes from Colors 3 section
- Added Resolving Oklab and Oklch values section
- Added srgb-linear color space
- Moved @color-profile and device-cmyk to level 5 per CSSWG resolution
- Defined interpolation color space
- Clarified that matrices are row-major and linked to the matrix multiplication library
- Split old Security & Privacy section into separate sections
- Defined quirks-mode quirky hex colors
- Removed fallback colors from device-cmyk
- Host syntax that does not declare a default now uses Oklab by default
- Added sample code for deltaE OK
- Added sample conversion code for OKlab and Oklch
- Added oklab() and oklch() functions *Added description of Oklab and Oklch*
- Added description of CIE LCH deficiencies
- Allowed all components of a color to be "missing" via the 'none' keyword, defined when components are "powerless" and automatically become missing in some cases, and fixed all references to "NaN" channels to use the "missing" concept.
- Defined explicit x,y whitepoint values, use consistently throughout
- Defined the term host syntax
- Defined context for resolving override-color colors
- Added a new pair of system colors
- Corrected HSL and HWB sample code
- Replaced table of HSL values with error-free version
- Added Lea Verou as co-editor by WG resolution
- Clarified that hue angle is unbounded
- MarkText example corrected
- Added diagrams, corrected examples

- Some editorial clarifications
- Minor typos corrected, markup corrections

§ Changes since the Working Draft of 12 November 2020

- Noted indeterminate hue ssue on near-neutral Lab values converted to LCH
- Clarified which steps are linear combinations in RGB Lab interconversion
- Added components descriptor to @color-profile, for use in CSS Color 5
- All predefined RGB color spaces are defined over the extended range
- Clarified that there is no gamut mapping or gamut clipping step prior to color interpolation
- Clarified interpolation of legacy sRGB syntaxes
- Removed the lab option from 'color()'
- List steps to interconvert between predefined color spaces
- Consistent use of the term color space (two words)
- Provided more guidance on selecting color space for mixing
- Recalculated an example to increase precision
- Added hue interpolation example
- Simplified 'color()' syntax by removing the fallback options
- Clarified the types of ICC profile that may be linked from @color-profile
- Support for the rare ICC Named Colors was removed
- Improved precision of standard whitepoint chromaticities
- Removed a trademark from description of one predefined color space
- Rephrased interpolation to be more generic wrt to interpolation space
- Corrected Accessibility Considerations section
- Clarified that the color space argument for 'color()' is mandatory, even for sRGB
- Clarified that currentColor is not restricted to sRGB
- Small correction to the sRGB to XYZ to sRGB matrices, improve round-tripping
- Clarified the rec2020 transfer function, citing the correct ITU Rec BT.2020-2 reference
- Correct fallback examples to use the correct syntax
- Don't force non-legacy colors to interpolate in a gamma-encoded space
- Define premultiplied alpha interpolation
- Start to address interpolation to and from currentColor

- Define hue interpolation with NaN
- Generalize color interpolation
- Define interpolation to be in Lab, with override to LCG
- Corrections to hue interpolation
- Defined hue angle interpolation
- Added interpolation section
- Corrected syntax in some examples
- Clarify exactly which components are allowed percentages, in 'color()'
- Change to serialize 'lch()' as itself rather than as 'lab()'
- Minimum 10 bits per component precision for non-legacy sRGB in 'color()'
- color space no longer optional in 'color()'
- Consistent minimum precision between lab() and color(lab)
- Clarified fallback procedure for the color() function – first valid in-gamut color, else first valid color which is then gamut mapped, else transparent black
- Clarified difference between opacity property and colors with opacity, notably for rendering overlapping text glyphs
- Added sample (but verified correct) code for ΔE2000
- Added definition of previously-undefined term chromaticity, with examples; define chromaticity diagram.
- Added explanation of color additivity, with examples
- Added source links to WPT tests
- Export definition of color, and valid color, for other specifications to reference
- Define minimum number of bits per component, for serialization
- Updated "applies to" definitions (CSS-wide change)
- Added image state (display referred or scene referred) for predefined color spaces
- Listed white point correlated color temperature (e.g. D65) alongside chromaticity coordinates, for clarity
- Clarified that rounding is towards +∞
- Correction of typos, markup corrections, link fixes

§ Changes since Working Draft of 5 November 2019

- Export some terms for use in other specifications

- Update requirement from WCAG 2.0 to 2.1
- Fully specify Unicode characters used for serialization
- Define serialization of special named colors
- Define serialization of device-cmyk()
- Define serialization of 'color()'
- Fully define RGB serialization, in maximally web-compatible way
- Define serialization of Lab and LCH
- Fully define serialization of alpha values
- Consistency pass to avoid accidental RFC2119
- Add IDs to all the examples, to enable referencing
- Separate resolved color and serialized color sections
- (Security) ICC profiles have no executable code
- Define what out-of-range means for profiled colors
- Clarify out-of-range clamping
- Add examples of specified values
- Clarify computed values
- Resist fingerprinting, with mandatory mappings for deprecated system colors
- Added explanatory note on history and reason for standardizing X11 colors
- Correct hwb sample code
- Add table of DeltaE2000 values for MacBeth patches
- Add note on ICC profile Internet Media type
- Add reference to PNG sRGB chunk
- Clarify CMYK to Lab interconversion
- Clarify RGB to Lab interconversion
- More comparison of HSL vs. LCH
- More description for Rec BT.2020 color space
- Updated description of prophoto-rgb
- Removed duplicate "keywords" from Value Definitions section
- Added an example of an invalid color
- Added example with multiple fallbacks
- Assorted typos and markup fixes

- Clarify handling for undeclared custom color spaces
- Clarify some examples and explanatory notes
- Handling for valid and invalid ICC profiles
- Define handling for images with explicit tagged color space
- Define color space for 4k, SDR video
- State that user contrast settings mst take precedence
- Clarify meaning of system colors outside for forced-color mode
- Update default style rules
- Add CIE XYZ color space to 'color()'
- Greater clarity on hue angles, NaN explicitly allowed
- Improve section on system color pairings, require AA accessible contrast
- Warn of interaction between overlapping glyphs and the opacity property
- Correct grammar in color definition
- Improve description of Highlight/HighlightText
- Correct prophoto-rgb transfer function
- More precision for prophoto-rgb primaries
- Started to define "can't be displayed"
- Removed paragraph about canvas surface
- Added the buttonborder, mark, and marktext system colors
- Added reverse conversion, sRGB to HWB
- Clarified polar spaces are cylindrical, not spherical
- Added an Accessibility Considerations section
- Started to describe chroma-reduction gamut mapping rather than per-component clipping
- Corrected white chromaticity for rec2020
- Made device-cmyk available by @color-profile; updated CMYK to color algorithm to only use naive conversion as a last resort
- Added print-oriented CMYK and KCMYOGV examples
- User-defined color spaces now dashed-ident, making predefined color spaces extensible without clashes
- Added lab option to the color() function
- Added normative reference for CIE Lab
- Clarified that prophoto-rgb uses D50 whitepoint so does not require adaptation

- Clarified direction of increasing angle in LCH
- Clarified that color names are ASCII case insensitive
- Initial value of the "color" property is now CanvasText
- Removed confusing gray() function per CSS WG resolution
- Collect scattered definitions into new Color terminology section
- Add helpful figures and more examples
- Minor editorial clarifications, spell check, fixing typos, bikeshed markup fixes

§ Changes since Working Draft of 05 July 2016

- Changed Lightness in Lab and LCH to be a percentage, for CSS compatibility
- Clamping of color values clarified
- Percentage opacity is now allowed
- Define terms sRGB and linear-light sRGB, for use by other specs
- Add new list of CSS system colors; renaming Text to CanvasText
- Make system color keywords compute to themselves
- Add computed/used entry for system colors
- Rewrite intro to non-deprecated system colors to center their use around forced-colors mode rather than generic use
- Consistent hyphenation of predefined color spaces
- Restore text about non-opaque elements painting at layers even when not positioned
- Initial value of the "color" property is now black
- Clarify hue in LCH is modulo 360deg (change now reverted)
- Clarify allowed range of L in LCH and Lab, and meaning of L=100
- Update references for color spaces used in video
- Add prophoto-rgb predefined color space
- Correct black and white luminance levels for display-p3
- Clarify display-p3 transfer function
- Add a98-rgb color space, correct table of primary chromaticities
- Clarify that currentColor's computed value is not the resolved color
- Update syntax is examples to conform to latest specification
- Remove the color-mod() function

- Drop the "media" from propdef tables
- Export, and consistently use, "transparent black" and "opaque black"
- Clarify calculated values such as percents
- Clarify required precision and rounding behavior for color channels
- Clarify "color" property has no effect on color font glyphs (unless specifically referenced, e.g. with currentColor)
- Clarify how color values are resolved
- Clarify that HSL, HWB and named colors resolve to sRGB
- Simplify conversion from device-cmyk to sRGB
- Describe previous, comma-using color syntaxes as "legacy"; change examples to commaless form
- Remove superfluous requirement that displayed colors be restricted to device gamut (like there was any other option!)
- Rename P3 to display-p3; avoid claiming this is DCI P3, as these are not the same
- Improved description of the parameters to the "color()" function
- Disallow predefined spaces from "@color-profile" identifier
- Add canonical order to "color", "color-adjust" and "opacity" property definitions
- Switch definition of alpha compositing from SVG11 to CSS Compositing
- Clarify sample conversion code is non-normative
- Add Security and Privacy Considerations
- Update several references to most current versions
- Convert inline issues to links to GitHub issues
- Minor editorial clarifications, formatting and markup improvements

§ Changes from Colors 3

The primary change, compared to CSS Color 3, is that CSS colors are no longer restricted to the narrow gamut of sRGB.

To support this, several brand new features have been added:

1. predefined, wide color gamut RGB color spaces
2. 'lab()', 'lch()', 'oklab()' and 'oklch()' functions, for device-independent color

Other technical changes:

1. Serialization of <color> is now specified here, rather than in the CSS Object Model
2. 'hwb()' function, for specifying sRGB colors in the HWB notation.
3. Addition of named color 'rebeccapurple'.

In addition, there have been some syntactic changes:

1. 'rgb()' and 'rgba()' functions now accept <number> rather than <integer>.
2. 'hsl()' and 'hsla()' functions now accept <angle> as well as <number> for hues.
3. 'rgb()' and 'rgba()', and 'hsl()' and 'hsla()' are now aliases of each other (all of them have an optional alpha).
4. 'rgb()', 'rgba()', 'hsl()', and 'hsla()' have all gained a new syntax consisting of space-separated arguments and an optional slash-separated opacity. All the color functions use this syntax form now, in keeping with CSS's functional-notation design principles.
5. All uses of <alpha-value> now accept <percentage> as well as <number>.
6. 4 and 8-digit hex colors have been added, to specify transparency.
7. The 'none' value has been added, to represent powerless components.

§ 19. Security Considerations

The system colors, if they actually correspond to the user's system colors, pose a security risk, as they make it easier for a malware site to create user interfaces that appear to be from the system. However, as several system colors are now defined to be "generic", this risk is believed to be mitigated.

§ 20. Privacy Considerations

This specification defines "system" colors, which theoretically can expose details of the user's OS settings, which is a fingerprinting risk.

§ 21. Accessibility Considerations

This specification encourages authors to not use color alone as a distinguishing feature.

This specification encourages browsers to ensure adequate contrast for specific system color foreground/background pairs. A harder requirement with specific AA or AAA contrast ratios was considered, but since browsers are often just passing along color choices made by the OS, or selected by users (who may have particular requirements, including lower contrast for people

living with migraines or epileptic seizures), the CSSWG was unable to require a specific contrast level.

§ Conformance

§ Document conventions

Conformance requirements are expressed with a combination of descriptive assertions and RFC 2119 terminology. The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in the normative parts of this document are to be interpreted as described in RFC 2119. However, for readability, these words do not appear in all uppercase letters in this specification.

All of the text of this specification is normative except sections explicitly marked as non-normative, examples, and notes. [RFC2119]

Examples in this specification are introduced with the words “for example” or are set apart from the normative text with `class="example"`, like this:

EXAMPLE 73 ¶

This is an example of an informative example.

Informative notes begin with the word “Note” and are set apart from the normative text with `class="note"`, like this:

Note, this is an informative note.

Advisements are normative sections styled to evoke special attention and are set apart from other normative text with `<strong class="advisement">`, like this:

UAs MUST provide an accessible alternative.

▼ TESTS

Tests relating to the content of this specification may be documented in “Tests” blocks like this one. Any such block is non-normative.

§ Conformance classes

Conformance to this specification is defined for three conformance classes:

style sheet
: A CSS style sheet.

renderer
: A UA that interprets the semantics of a style sheet and renders documents that use them.

authoring tool
: A UA that writes a style sheet.

A style sheet is conformant to this specification if all of its statements that use syntax defined in this module are valid according to the generic CSS grammar and the individual grammars of each feature defined in this module.

A renderer is conformant to this specification if, in addition to interpreting the style sheet as defined by the appropriate specifications, it supports all the features defined by this specification by parsing them correctly and rendering the document accordingly. However, the inability of a UA to correctly render a document due to limitations of the device does not make the UA non-conformant. (For example, a UA is not required to render color on a monochrome monitor.)

An authoring tool is conformant to this specification if it writes style sheets that are syntactically correct according to the generic CSS grammar and the individual grammars of each feature in this module, and meet all other conformance requirements of style sheets as described in this module.

§ Partial implementations

So that authors can exploit the forward-compatible parsing rules to assign fallback values, CSS renderers **must** treat as invalid (and ignore as appropriate) any at-rules, properties, property values, keywords, and other syntactic constructs for which they have no usable level of support. In particular, user agents **must not** selectively ignore unsupported component values and honor supported values in a single multi-value property declaration: if any value is considered invalid (as unsupported values must be), CSS requires that the entire declaration be ignored.

§ Implementations of Unstable and Proprietary Features

To avoid clashes with future stable CSS features, the CSSWG recommends following best practices for the implementation of unstable features and proprietary extensions to CSS.

§ Non-experimental implementations

Once a specification reaches the Candidate Recommendation stage, non-experimental implementations are possible, and implementors should release an unprefixed implementation of any CR-level feature they can demonstrate to be correctly implemented according to spec.

To establish and maintain the interoperability of CSS across implementations, the CSS Working Group requests that non-experimental CSS renderers submit an implementation report (and, if necessary, the testcases used for that implementation report) to the W3C before releasing an unprefixed implementation of any CSS features. Testcases submitted to W3C are subject to review and correction by the CSS Working Group.

Further information on submitting testcases and implementation reports can be found from on the CSS Working Group's website at http://www.w3.org/Style/CSS/Test/. Questions should be directed to the public-css-testsuite@w3.org mailing list.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CSS-ANIMATIONS-1] defines the following terms:

- animation

[CSS-BACKGROUNDS-3] defines the following terms:

- background
- background-color

[CSS-BORDERS-4] defines the following terms:

- border-bottom-color
- border-color
- border-left-color
- border-right-color
- border-top-color

[CSS-CASCADE-5] defines the following terms:

- computed value
- inherited value
- specified value
- used value

[CSS-COLOR-5] defines the following terms:

- color-mix()

[CSS-COLOR-ADJUST-1] defines the following terms:

- forced colors mode

[CSS-CONDITIONAL-3] defines the following terms:

- @supports
- supports(conditionText)

[CSS-IMAGES-4] defines the following terms:

- <gradient>

[CSS-SYNTAX-3] defines the following terms:

- <dimension-token>
- <hash-token>
- <ident-token>
- <number-token>

[CSS-TEXT-DECOR-4] defines the following terms:

- text-emphasis-color

[CSS-TRANSITIONS-1] defines the following terms:

- transition

[CSS-VALUES-4] defines the following terms:

- #
- ,
- <angle>
- <ident>
- <integer>
- <number>
- <percentage>
- ?
- canonical unit
- css-wide keywords
- functional notation
- keyword
- {a}
- |

[CSS2] defines the following terms:

- z-index

[CSS22] defines the following terms:

- auto
- stacking context

[CSSOM-1] defines the following terms:

- resolved value

[DOM] defines the following terms:

- quirks mode

[FILTER-EFFECTS-1] defines the following terms:

- filter

[HTML] defines the following terms:

- mark

[MEDIAQUERIES-5] defines the following terms:

- media feature

§ References

§ Normative References

[Bradford-CAT]

Ming R. Luo; R. W. G. Hunt. *A Chromatic Adaptation Transform and a Colour Inconstancy Index. Color Research & Application 23(3) 154-158.* June 1998.

[CIELAB]
*ISO/CIE 11664-4:2019(E): Colorimetry — Part 4: CIE 1976 L*a*b* colour space*. 2019. Published. URL: http://cie.co.at/publications/colorimetry-part-4-cie-1976-lab-colour-space-1

[COLORIMETRY]
Colorimetry, Fourth Edition. CIE 015:2018. 2018. URL: http://www.cie.co.at/publications/colorimetry-4th-edition

[Compositing]
Rik Cabanier; Nikos Andronikos. *Compositing and Blending Level 1*. URL: https://drafts.fxtf.org/compositing-1/

[CSS-BACKGROUNDS-3]
Bert Bos; Elika Etemad; Brad Kemper. *CSS Backgrounds and Borders Module Level 3*. URL: https://drafts.csswg.org/css-backgrounds/

[CSS-BORDERS-4]
CSS Borders and Box Decorations Module Level 4. Editor's Draft. URL: https://drafts.csswg.org/css-borders-4/

[CSS-CASCADE-5]
Elika Etemad; Miriam Suzanne; Tab Atkins Jr.. *CSS Cascading and Inheritance Level 5*. URL: https://drafts.csswg.org/css-cascade-5/

[CSS-COLOR-ADJUST-1]
Elika Etemad; et al. *CSS Color Adjustment Module Level 1*. URL: https://drafts.csswg.org/css-color-adjust-1/

[CSS-CONDITIONAL-3]
David Baron; Elika Etemad; Chris Lilley. *CSS Conditional Rules Module Level 3*. URL: https://drafts.csswg.org/css-conditional-3/

[CSS-SYNTAX-3]
Tab Atkins Jr.; Simon Sapin. *CSS Syntax Module Level 3*. URL: https://drafts.csswg.org/css-syntax/

[CSS-TEXT-DECOR-4]
Elika Etemad; Koji Ishii. *CSS Text Decoration Module Level 4*. URL: https://drafts.csswg.org/css-text-decor-4/

[CSS-VALUES-3]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 3*. URL: https://drafts.csswg.org/css-values-3/

[CSS-VALUES-4]
Tab Atkins Jr.; Elika Etemad. *CSS Values and Units Module Level 4*. URL: https://drafts.csswg.org/css-values-4/

[CSS2]
Bert Bos; et al. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification*. URL: https://drafts.csswg.org/css2/

[CSS22]
Bert Bos. *Cascading Style Sheets Level 2 Revision 2 (CSS 2.2) Specification*. URL: https://drafts.csswg.org/css2/

[CSSOM-1]
Daniel Glazman; Emilio Cobos Álvarez. *CSS Object Model (CSSOM)*. URL: https://drafts.csswg.org/cssom/

[Display-P3]
Apple, Inc. *Display P3*. 2022-02. URL: https://www.color.org/chardata/rgb/DisplayP3.xalter

[DOM]
Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[HSL]
George H. Joblove, Donald Greenberg. *Color spaces for computer graphics*. August 1978. URL: https://doi.org/10.1145/965139.807362

[HTML]
Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[HWB]
Alvy Ray Smith. *HWB — A More Intuitive Hue-Based Color Model*. 1996. URL: http://alvyray.com/Papers/CG/HWB_JGTv208.pdf

[ICC]
ICC.1:2022 (Profile version 4.4.0.0). May 2022. URL: http://www.color.org/specification/ICC.1-2022-05.pdf

[ITU-R-BT.601]
Recommendation ITU-R BT.601-7 (03/2011), Studio encoding parameters of digital television for standard 4:3 and wide screen 16:9 aspect ratios. 8 March 2011. Recommendation. URL: https://www.itu.int/rec/R-REC-BT.601/

[ITU-R-BT.709]
Recommendation ITU-R BT.709-6 (06/2015), Parameter values for the HDTV standards for production and international programme exchange. 17 June 2015. Recommendation. URL: https://www.itu.int/rec/R-REC-BT.709/

[MEDIAQUERIES-5]
Dean Jackson; et al. *Media Queries Level 5*. URL: https://drafts.csswg.org/mediaqueries-5/

[Oklab]
Björn Ottosson. *A perceptual color space for image processing*. December 2020. URL: https://bottosson.github.io/posts/oklab/

[Rec.2020]
Recommendation ITU-R BT.2020-2: Parameter values for ultra-high definition television systems for production and international programme exchange. October 2015. URL: http://www.itu.int/rec/R-REC-BT.2020/en

[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. March 1997. Best Current Practice. URL: https://datatracker.ietf.org/doc/html/rfc2119

[ROMM]
ISO 22028-2:2013 Photography and graphic technology — Extended colour encodings for digital image storage, manipulation and interchange — Part 2: Reference output medium metric RGB colour image encoding (ROMM RGB). 2013-04. URL: https://www.iso.org/standard/56591.html

[SMPTE296]
ST 296:2012, 1280 × 720 Progressive Image 4:2:2 and 4:4:4 Sample Structure — Analog and Digital Representation and Analog Interface. 17 May 2012. Standard. URL: https://doi.org/10.5594/SMPTE.ST296.2012

[SRGB]
Multimedia systems and equipment - Colour measurement and management - Part 2-1: Colour management - Default RGB colour space - sRGB. URL: https://webstore.iec.ch/publication/6169

[SVG11]
Erik Dahlström; et al. *Scalable Vector Graphics (SVG) 1.1 (Second Edition)*. 16 August 2011. REC. URL: https://www.w3.org/TR/SVG11/

§ Informative References

[CSS-ANIMATIONS-1]
David Baron; et al. *CSS Animations Level 1*. URL: https://drafts.csswg.org/css-animations/

[CSS-COLOR-5]
Chris Lilley; et al. *CSS Color Module Level 5*. URL: https://drafts.csswg.org/css-color-5/

[CSS-IMAGES-4]
Tab Atkins Jr.; Elika Etemad; Lea Verou. *CSS Images Module Level 4*. URL: https://drafts.csswg.org/css-images-4/

[CSS-TRANSITIONS-1]
David Baron; et al. *CSS Transitions*. URL: https://drafts.csswg.org/css-transitions/

[CSS3-TEXT-DECOR]
Elika Etemad; Koji Ishii. *CSS Text Decoration Module Level 3*. URL: https://drafts.csswg.org/css-text-decor-3/

[DCI-P3]
SMPTE Recommended Practice - D-Cinema Quality — Reference Projector and Environment. 2011. URL: http://ieeexplore.ieee.org/document/7290729/

[FILTER-EFFECTS-1]
Dirk Schulze; Dean Jackson. *Filter Effects Module Level 1*. URL: https://drafts.fxtf.org/filter-effects-1/

[JPEG]
Eric Hamilton. *JPEG File Interchange Format*. September 1992. URL: https://www.w3.org/Graphics/JPEG/jfif3.pdf

[PNG]
Tom Lane. *Portable Network Graphics (PNG) Specification (Second Edition)*. URL: https://w3c.github.io/PNG-spec/

[ROMM-RGB]
ROMM RGB. URL: https://www.color.org/chardata/rgb/rommrgb.xalter

[Sharma]
G. Sharma; W. Wu; E. N. Dalal. *The CIEDE2000 Color-Difference Formula: Implementation Notes, Supplementary Test Data, and Mathematical Observations*. February 2005. URL: http://www2.ece.rochester.edu/~gsharma/ciede2000/

[TIFF]
TIFF Revision 6.0. 3 June 1992. URL: https://www.loc.gov/preservation/digital/formats/fdd/fdd000022.shtml

[WCAG21]
Andrew Kirkpatrick; et al. *Web Content Accessibility Guidelines (WCAG) 2.1*. URL: https://w3c.github.io/wcag/21/guidelines/

[Wolfe]
Geoff Wolfe. *Design and Optimization of the ProPhoto RGB Color Encodings*. 2011-12-21. URL: http://www.realtimerendering.com/blog/2011-color-and-imaging-conference-part-vi-special-session/

§ Property Index

Name	Value	Initial	Applies to	Inh.	%ages	Animation type	Canonical order	Computed value
'color'	<color>	CanvasText	all elements and text	yes	N/A	by computed value type	per grammar	computed color, see resolving color values
'opacity'	<alpha-value>	1	all elements	no	map to the range [0,1]	by computed value type	per grammar	specified number, clamped to the range [0,1]

§ Issues Index

ISSUE 1 Computed value needs to be able to represent combinations of 'currentColor' and an actual color. Consider the value of 'text-emphasis-color' in `div { text-emphasis: circle; transition: all 2s; }`
`div:hover { text-emphasis-color: lime; }`
`em { color: red; }` See Issue 445.