UNIVERSITÀ DI PISA

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Informatica

Tesi di Laurea Specialistica in Informatica

Extending Cairo with color space support

Candidato:

Andrea Canciani

Relatore: **Dott. Antonio Cisternino**

Controrelatore: **Prof. Francesco Romani**

Anno Accademico 2009/10

Abstract

Cairo is a 2D vector graphics library which can draw on multiple output targets, including PDF, PostScript, SVG, Xlib, Quartz and GDI.

Cairo provides an interface which exposes PDF-like operations, hence it can draw complex shapes and fill them with a colored pattern, but it lacks color management.

The purpose of this work is to design and implement an extension of the Cairo library which augments it with color management support, in accordance with the ICC and PDF standard specifications. This extension makes it possible to move the color handling from applications to the graphic library and provides a flexible and efficient drawing model.

Acknowledgements

This work would have been much harder, if not impossible, without the contributions and help I received from many people.

I would especially like to thank Cairo developers, in particular Adrian Johnson and Benjamin Otte, whose previous work on color spaces in Cairo has been a useful reference and starting point, and Chris Wilson and Joonas Pihlaja who helped me to learn most of the Cairo internals.

I believe that a valuable contribution to this work comes from the long thread about the extension I was proposing. I sent a mail on this topic on the Cairo development mailing list and many users replied with use cases and suggestions based on their experience with color management.

The chats with Øyvind Kolås (maintainer of the GEGL graphic library), have been very useful when trying to find out what the correct drawing model was and the requirements it had to satisfy.

I greatly appreciate the helpfulness of Lars Borg, who kindly permitted the use of his images to recreate the ICC color profile test in Cairo, and of Gernot Hoffmann, who let me use his beautiful PostScript images in this document.

My friends, my family and my advisor deserve special credits for supporting me through the development of this project.

Contents

List of Figures

Chapter 1

Introduction

Graphic systems and drawing libraries have been able to draw in color for a long time, but hardware constraints and limited scope of high-quality color management have pushed for faster software with poor color handling. Digital color has recently received more attention because of the diffusion of digital photography.

It is quite common for anybody to shoot digital photos and to edit them or to watch and print them using a browser. Some people do not care if the colors printed photo match those on screen, but most people is disappointed when this happens. Unless some additional care is taken when printing, this happens very frequently, because monitors and printers synthesize colors in different ways.

The same problem obviously applies to any digital visual content, therefore professional graphic workflows started tackling it a long time ago, because they require correct color processing and accurate color reproduction in order to get the same result no matter what output device is used.

Instead, most software represents colors with 3 numerical values, which indicate the intensity of red, green and blue respectively. When applications directly accessed the video memory of the computer, these values controlled the image produced on the screen. Nowadays application cannot communicate directly with the hardware, but the most graphic software stacks still do the same thing: they just feed the graphic card with the RGB values from the application.

As hardware speed and user expectations grow, more applications are implementing proper color handling, because it is needed in any software which must display and print documents and provide the same visual results across multiple computers and output devices.

A notable example can be found in browsers, whose latest versions are providing color management: Mozilla Firefox enables is by default since version 3.5 and Microsoft Internet Explorer provides it since version 9.

Although some applications are adding color management support, it is quite hard to write color managed software, because the burden of color space handling and color transformation is in the application instead of the graphic library.

Most graphic libraries have not been designed to perform color management internally and provide no way to use an external color management system. This matched the requirements of most applications, but is becoming insufficient for modern graphic interfaces, because application which need color management have no option but to perform color transformations explicitly.

This approach has several limitations and is not generally feasible, therefore another approach is proposed: if graphic libraries expose appropriate color managed operations, it is possible to completely delegate color handling to them. This moves the color management code from applications to graphic libraries, where it can be integrated with the drawing pipeline to get better quality and higher performance.

This change is independent but similar to the evolution from raster-only imaging libraries to vector based ones: higher level operations make it easier to get better-looking graphics.

The purpose of this work is to extend the Cairo graphic library with facilities to enable the development of color managed applications. Cairo has been chosen because it is a widely used graphic library, especially in open source projects (GTK and Mozilla being the two major ones), which is completely lacking color management. Moreover Cairo is a cross-platform library which supports many different output formats, therefore this extension makes color management support available for a very wide range of applications.

The thesis has the following structure:

- Chapter 2 introduces the theoretical foundations of color management and the standard technology which is generally used to implement it.

- Chapter 3 analyzes the availability of this technology or other alternative color handling solutions in software across the whole graphic stack.

- Chapter 4 describes the current Cairo drawing model and extends it to support color transformations, based on what is currently available in state-of-the-art graphic systems.

- In chapter 5 an implementation of this model is presented, along with shortcomings, problems, and solutions found while extending the library.

- Chapter 6 shows the results and points out some ideas to improve the proposed implementation of color management and to take further advantage of the possibilities it offers.

Chapter 2

Color spaces

2.1 Digital color

In order to perform computations on colored images, colors must be represented with numbers. The most common way to express a color in a digital environment has been to simply use the output intensity for devices which measure colors (like scanners and digital cameras) and the input intensity for devices which reproduce colors (like monitors, projectors, printers).

Although these values indicate a well-defined color on each device, they are not related not to human color perception, but just to the device behavior. In particular, the most common case has been to have three of these values, each controlling a different type of light-emitting element of a monitor. These elements are phosphors which cover the display surface and can emit red, blue or green light, depending on their type.

Because of this, colors are often expressed with three numbers, indicating a red, green and blue light intensity, also known as RGB. As explained in section 2.5, these values will not result in the same color if used on two different devices. In fact, these values do not indicate a color, but input levels of a color reproduction device. Nonetheless, they provide a color representation which has been widely used when the cost of getting accurate color reproduction was too high for commodity hardware.

Although these numbers correspond to a physical measure, they have often been indicated as pure numbers in the $[0, 1]$ interval, to be intended as a relative intensity, from the minimum to the maximum allowed by the monitor. Another common representation, which is used to improve the performance of the computations, indicates the values as bytes, i.e. integers in $[0, 255]$. This representation matches very well the capabilities of most video hardware, but it quantizes to only 256 different values. This is sufficient for some simple graphic operations, but if more complex computations are performed on data represented with this accuracy, the error can be amplified and become visible. To work around this problem, some programs use 16-bit color data and the output images of many digital photo cameras can have more than 8 bit of precision.

As the hardware improves and provides more processing power and input and output devices which can handle a higher number of quantization levels, the applications evolve and become able to perform slower computations on color data within reasonable times, therefore some modern programs can even handle color in floating point format.

Moreover, thanks to the available computational power, these programs started to implement color systems which can provide more accurate results than simple RGB values. This is required in order to satisfy the user expectation of the same visual result no matter what output device is used.

2.2 Color perception

A framework which can describe colors and operate on them preserving their appearance across multiple different physical devices needs to have foundations based on the human perception of colors. In this setting, a color is usually defined as the sensation produced in the brain by the stimulation of the eye with light.

From a physical point of view, this light is an electromagnetic wave, thus it can be described by its spectral power distribution, i.e. the intensity of each frequency composing the wave. Although spectra could be used to indicate colors, this is not effective representation, because two identical spectra will cause the same color to be perceived, but the human vision system perceives some different spectra as identical colors as well.

Two such spectral power distributions which produce the same color sensation are called *metamers*. Metamerism plays an important role in color reproduction, because it makes it possible to represent the color associated to a very wide range of spectra using a combination of just three primaries.

This property can be obtained by studying the physiology of the human eye. The human retina contains two types of cells: rods and cones. Cones can be of 3 different types, so one might expect the 4 cellular perceptions to form a 4-dimensional space in which each "color sensation" occupies a different point. As a matter of fact, the rods only contribute significant information in particular conditions (when viewing very dark images), so they can be ignored in image reproduction systems. The sensibility of each cone type to each wavelength of the visible spectrum provide sufficient information to determine with a reasonable accuracy [1] whether two spectra are metamers or not.

Most color reproduction systems rely on this fact to produce in the

[1]The vision system is actually not completely uniform and color perception depends on other factors, for example the angle under which the color is viewed. This is often ignored and a standard approximation is used.

observer color sensations using appropriate combinations of red, green and blue light, but this approach has some limits, because the range of colors that can be synthesized is smaller than the range of visible colors.

2.3 Reference color spaces

In order to provide a systematic way to identify metamers and represent colors, CIE (Commission Internationale d'Eclairage) defined the *standard observer*[5].

To create a model of the human vision based on the assumption that any color could be obtained from an appropriate combination of three monochromatic lights, CIE performed an experiment to determine an average viewer which could be used as reference.

The standard observer is a mathematical model based on empirical measures which describes the color perception of the hypothetical average human. Any color is represented by the values X, Y, Z which can be computed as:

$$
\begin{aligned}
X &= \int_0^\infty I(\lambda)\,\overline{x}(\lambda)\,d\lambda \\
Y &= \int_0^\infty I(\lambda)\,\overline{y}(\lambda)\,d\lambda \\
Z &= \int_0^\infty I(\lambda)\,\overline{z}(\lambda)\,d\lambda
\end{aligned}
$$

In these equations, $I(\lambda)$ is the spectral power distribution of the color, $\overline{x}(\lambda)$, $\overline{y}(\lambda)$ and $\overline{z}(\lambda)$ are the color matching functions (shown in figure 2.1), which have been derived experimentally by CIE.

Two spectra which have the same XYZ coordinates are metamers for the standard observer, i.e. the average human would perceive them as the same color. For this reason these coordinates are called *tristimulus* and are the preferred representation of colors from a colorimetric perspective.

The X, Y and Z components define a point in the CIEXYZ color space, which corresponds to a color perception. In general, color spaces are coordinate systems in which a colors can be represented as an n-uple. In order to get a colorimetric meaning, color spaces are usually defined by specifying the relationship between each point with the corresponding tristimulus, i.e. using the CIEXYZ color space as a reference.

The main defect of CIEXYZ is its non-uniformity. The perceptual difference between two colors does not correspond in a simple way to the distance of their XYZ representation.

To address this problem, other color spaces have been constructed; the most widely used one is probably CIELAB, which has been defined, again, by CIE. The CIELAB coordinates can be computed from XYZ as follows:

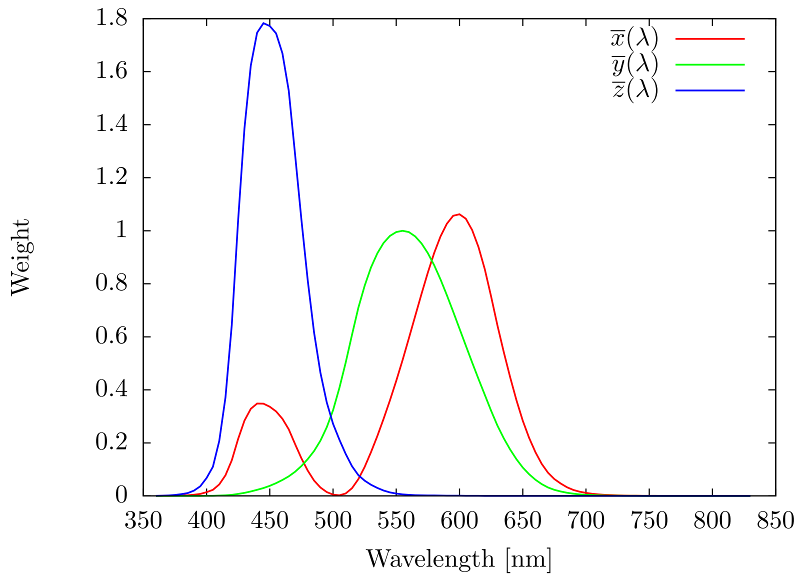

Figure 2.1: The CIEXYZ color matching functions

$$L^\star = 116 f\left(\frac{Y}{Y_n}\right) - 16$$
$$a^\star = 500\left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right]$$
$$b^\star = 200\left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right]$$

where (X_n, Y_n, Z_n) is the reference white point (usually one of the CIE standard illuminants, like D50 or D65) and

$$f(t) = \begin{cases} t^{\frac{1}{3}} & \text{if } t > (\frac{6}{29})^3 \\ \frac{1}{3}\left(\frac{29}{6}\right)^2 t + \frac{4}{29} & \text{otherwise} \end{cases}$$

CIEXYZ and CIELAB have two special roles that other color spaces don't usually have:

- They are the basis for other color spaces. Other color spaces are typically based either on CIEXYZ or on CIELAB.
- They are interchange color spaces. If the relationship between a color space and CIEXYZ is defined appropriately, it is possible to convert the tristimulus representation of a color to and from the representation in the new color space. The same consideration applies to CIELAB.

This makes it possible to convert colors between two arbitrary, otherwise unrelated, color spaces.

Another well-known CIEXYZ-based color space is CIExyY. It is projectively related to CIEXYZ and its purpose is to better separate luminance from chromaticity coordinates.

The Y component (luminance) of CIExyY is the same as that of CIEXYZ. The x, y components, also known as chromaticity coordinates, are defined by the following formulas:

$$x = \frac{X}{X + Y + Z}$$
$$y = \frac{Y}{X + Y + Z}$$

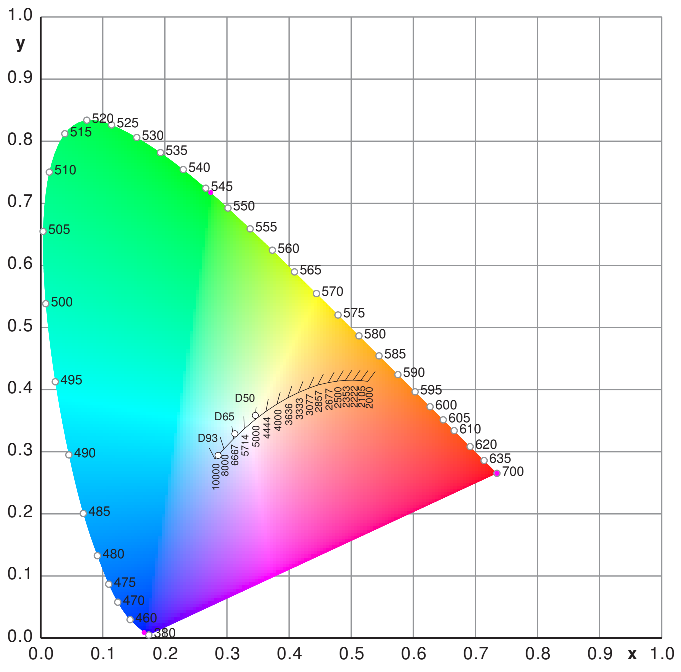

Figure 2.2: The CIExyY chromaticity chart. Image from [9], used with kind permission of the author, Gernot Hoffmann.

This color space is often used to compare the expressive power of other color spaces. In figure 2.2 a projection of the visible colors in the xy plane is shown (the colored area of the graph), delimited by the points corresponding to the visible spectrum. The plot shows some other well-known points: D50, D65 and D93, which are the most widely used CIE reference illuminants, and the Planckian locus, which is range of colors emitted by a black body when varying its temperature.

2.4 Standard color spaces

The CIEXYZ and CIELAB color spaces have been designed to describe human perception, but they are unrelated to image acquisition and reproduction devices. Other color spaces have been standardized and have been widely used for television, computer monitors, scanners, printers etc.

Monitors are physical devices which reproduce colors using additive color synthesis: they mix colored light to obtain the desired color. The standard observer model implies that three primaries would in theory be sufficient to compose any color, but in practice, this would require the light emitters to be able to produce any intensity, including the negative ones. This is not physically possible, so the range of colors which can be composed is limited to the combination of the primaries ranging from minimum to maximum intensity (usually represented respectively as 0 and 1).

The range of possible colors within these constraints is called *gamut*. Although the gamut is usually a three-dimensional subset of the CIEXYZ color space, for simplicity it is often represented by its bi-dimensional projection on the xy plane of CIExyY.

The color space which is currently being used as the standard color space for modern file formats and devices is sRGB. It is defined from the CIEXYZ color space, but it has been designed to match the behavior of computer monitors, therefore its primaries are the colors emitted by red, green and blue phosphors of typical CRT displays. Moreover, the relationship between the coordinates and the primary intensity (usually called *gamma curve* or *tone reproduction curve*) is non-linear, because the relationship between the input signal and the output intensity of a cathode ray tube is a power law.

In [16] the sRGB color space is defined as:

$$\begin{bmatrix} R_{linear} \\ G_{linear} \\ B_{linear} \end{bmatrix} = \begin{bmatrix} 3.2410 & -1.5374 & -0.4986 \\ -0.9692 & 1.8760 & 0.0416 \\ 0.0556 & -0.2040 & 1.0570 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

$$f(t) = \begin{cases} 1.055t^{\frac{1.0}{2.4}} - 0.055 & \text{if } t > 0.00304 \\ 12.92t & \text{otherwise} \end{cases}$$

$$R = f(R_{linear})$$
$$G = f(G_{linear})$$
$$B = f(B_{linear})$$

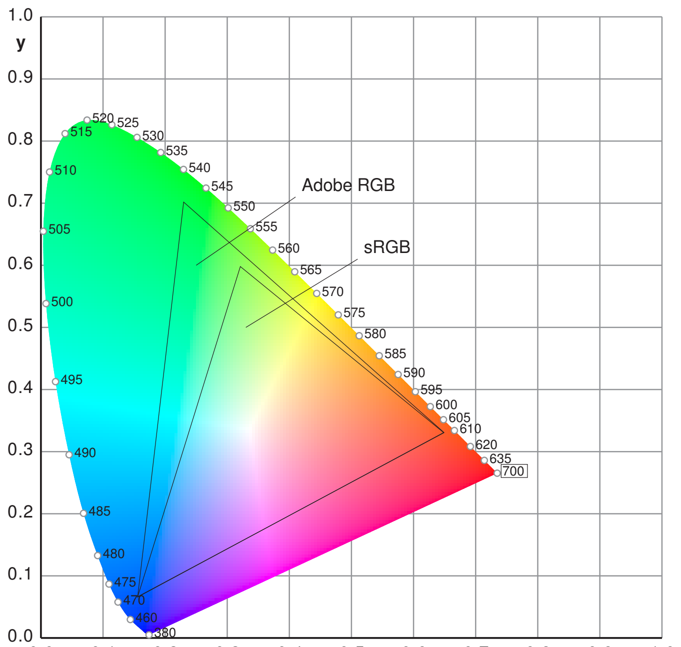

Figure 2.3: A projection of the gamuts of sRGB and of Adobe RGB in the CIExyY chromaticity chart. Image based on [9], with the authorization of the original author, Gernot Hoffmann.

Unfortunately the gamut of sRGB is quite small, so other color spaces with a wider gamut are often used, for example to handle photos (Adobe RGB, ProPhoto RGB) 2.3. The main disadvantage of these color spaces is that, since colors are represented with quantized components, the minimum color difference is bigger than with sRGB. This can be a problem if colors are represented with 8 bits per components or fewer because it might make it impossible to have a continuous color transitions.

Other standard color spaces have been created, for example as part of television standards ([13], [2], [17]). They all have three primaries, gamma curves and, although they have different coefficients, they share the basic structure with sRGB. This makes it possible to represent them in a simple way, as shown in section 2.9.

Luma-chroma color spaces, which are often employed in image and video compression to keep more detail in the image lightness at the expense of chromatic accuracy, retain the same structure as well, although the three components are not actually primaries. The JPEG YC_bC_r color space is the most common example of this family of color spaces.

2.5 Device-dependent color spaces

Color spaces in this family are probably the most widely used, because colors described by giving a n-uple of components (like RGB, Hue-Saturation-Lightness or Cyan-Magenta-Yellow-Black values) without specifying the relationship between the components and the tristimulus values of the color, are typically specified in a device-dependent color space.

Most software ignores color management and sends commands to the hardware without proper color handling. This happens when color is specified in a device-dependent color space, i.e. its components are directly passed to the hardware, without transforming them to take into account the color reproduction behavior of the devices.

This happens, for example, whenever the pixel components of a monitor are controlled directly. Two different monitors displaying the brightest available "red" will usually show two different shades of red.

The very wide use of RGB triples led to the misconception that RGB values are sufficient to describe a color, without the need of the reference system provided by device-independent color spaces, but in general, a point in a device-dependent color space does *not* represent a well-defined, unique color. The color can only be obtained by using the coordinates of that point as input for a device.

The coordinates of a point in a device-dependent color space are the intensities of the device components, therefore any point whose coordinates are in the valid range (i.e. between the minimum and maximum intensity that the device can handle) immediately defines a point in the gamut of a device, but the tristimulus (or any other colorimetric description) of the color cannot be computed, it can only be measured after the device has reproduced it.

This means that the same point can result in a different tristimulus on each device and conversely that a given tristimulus is in a point which depends on the device. For this reason device-dependent color spaces are not appropriate when color fidelity across multiple devices is desired.

2.6 Hue, Saturation, Lightness

Another family of color spaces has been designed to assist artists when they need to choose a color. Hue, saturation and lightness are easier to combine into the desired color than red, green and blue intensity, because they emulate the color mixing procedure performed on pigments. To create color spaces based on these properties, a relationship between them and RGB colors was established, defining HSV (Hue, Saturation, Value) and HSL (Hue, Saturation, Lightness) as depicted in figure 2.4.

These color spaces have been defined on top of device-dependent RGB,

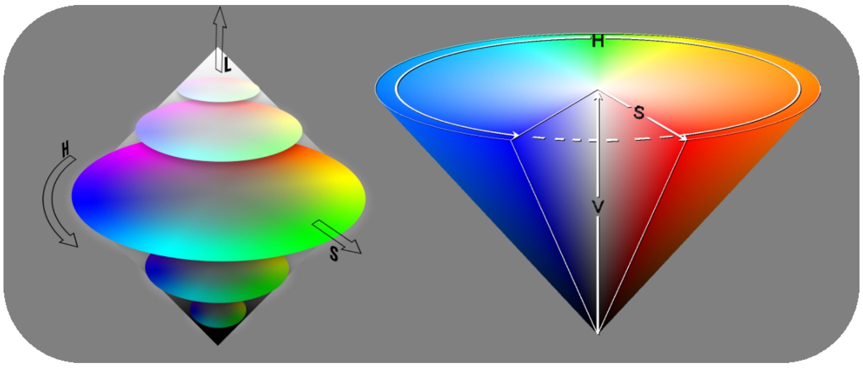

Figure 2.4: On the left, the color cone described by HSV; on the right, the HSL color double cone.

therefore they suffer from the same issues as any device-dependent color space. Despite this, they are important because they provide an intuitive way to express colors, thus alternative color spaces were defined with the same structure.

CIELCH is a Lightness-Chroma-Hue device-independent color space which provides the same perceptual uniformity properties as CIELAB but whose coordinates are in a cylindrical space which separates hue and chroma (saturation).

Its definition is a straightforward transformation of the CIELAB coordinates in cylindrical coordinates:

$$\begin{aligned} L^\star_{LCH} &= L^\star_{LAB} \\ C^\star_{LCH} &= \sqrt{(a^\star_{LAB})^2 + (b^*_{LAB})^2} \\ H^\star_{LCH} &= arctan\frac{b^*_{LAB}}{a^*_{LAB}} \end{aligned}$$

2.7 CMYK

The device-dependent CMYK color space (Cyan, Magenta, Yellow and Key black) is the printer-related equivalent of device-dependent RGB, i.e. to define a color, it directly specifies the intensities of ink to be sent to the the device.

There is a substantial difference in the color synthesis performed by display devices and that used in printing devices: monitors use additive color synthesis, where the spectra of multiple light emitters are summed; printers drop think ink layers on paper, which behave as filters for the light being reflected by the medium. This means that each ink can absorb a part of the light being reflected and this type of color synthesis is thus called subtractive.

Just like red, green and blue are three primaries which provide a reasonably wide gamut for additive color synthesis, their complements (cyan, magenta and yellow) can be used to compose colors in subtractive color synthesis, although the physical properties of inks and paper usually result in a smaller gamut than what most monitors provide. This is basically caused by the inherent impossibility to synthesize fully saturated colors.

Even if it would not be strictly needed, printers usually have a fourth ink, black. This provides a twofold advantage: since the black printed area being larger than colored areas on average, the black ink makes it possible to have a lower cost per page; in addition, from the colorimetrical point of view this ink provides a different shade of black, which is often more “colorless” than the shade which can be composed by mixing the same quantity of other inks.

Similar considerations led to exachrome printers, which try to widen the gamut by providing additional colors to be mixed, and to printing workflows which include specific inks for colors which cannot be synthesized. Sometimes these inks are not even colors, but are used for some special effects, like metallic or glittering inks.

2.8 Calibrated color spaces

The sRGB color space has been proposed as a standard color space to be adopted both for software and hardware color management. This would in theory make it possible to have a color managed workflow without the need of color profiles. However, even if modern hardware tries to comply with the sRGB standard, a device-specific color space is usually needed in order to achieve high color accuracy. Each physical device is unique and changes over time, thus in a professional graphic workflow both input and output devices must be calibrated.

A calibrated color space is a color space whose primaries and tone reproduction curves are determined by measuring the input/output behavior of the device with a colorimeter or with a spectrometer. This makes it possible to have a very accurate match between the expected color (modeled by the mapping between the device-specific color space and CIEXYZ) and the perceived color.

The calibration process requires additional hardware and software, but its cost is nowadays quite reasonable. Additionally, some vendors provide color profiles for their hardware. These profiles are usually created by averaging the properties of some sampling of the products, so they cannot be as accurate as a device-specific profile, but they can improve the color accuracy of the device with no additional effort or cost for the user.

It should be noted that, although calibrated color profiles are usually created to match a specific device, they are device-independent color spaces, because they provide a mapping between the components and the corresponding tristimulus.

2.9 ICC color profiles

Even if color spaces are defined mathematically, they often need to be stored along with the color components to which they give a colorimetrical meaning.

This can be accomplished by encoding them as an color profile, which can represent the mathematical definition of the color space as a sequence of bytes.

The most widespread format for color profiles has been designed and specified by the International Color Consortium. ICC profiles [3] can represent color spaces with up to 15 components and with an almost arbitrary mapping to/from CIEXYZ or CIELAB.

To guarantee that colors can be converted between any color spaces, without having to define a mapping between each possible pair, the conversion is defined using a *Profile Connection Space*. ICC profiles contain a function which converts colors from the profile color space to the PCS (which is required to be either CIEXYZ or CIELAB) and vice versa. To convert a color between two arbitrary color spaces, it is converted from the source color space to the PCS, then from the PCS to the destination color space.

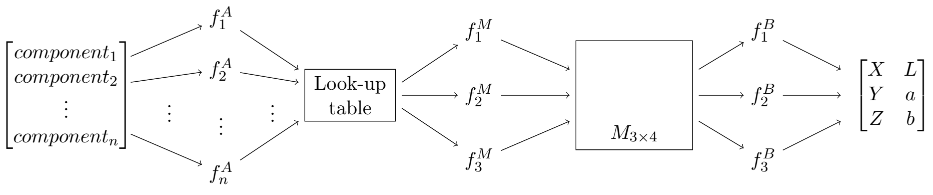

Figure 2.5: The general color space to PCS conversion

Figure 2.5 shows the general conversion from a color in the color space defined in a profile and the PCS. The multidimensional LUT makes arbitrary mappings possible even if the f_i^X functions are constrained to be unary. Simple color spaces can often be described by providing only a part of these components. For example, to describe sRGB, the M matrix and the f_i^M functions would be sufficient[2].

[2]There are some additional constraints on which components must be specified and the ICC specification makes it possible to avoid this complicated definition for simple color spaces. Most standard RGB color spaces and simple calibrated color spaces can be defined directly by specifying the primaries and the tone reproduction curves.

The transformation from PCS to the profile color space involves the same operations, but performed in the opposite direction. The opposite transformations is not, in general, the inverse of the direct transformation, therefore it is explicitly described in the profile.

An ICC color profile usually contains multiple color transformation functions between the color space and the PCS, which are selected depending on the *rendering intent.* Although in theory the association between a color space and CIEXYZ is unique and completely defines the color space, in practice color spaces have a limited gamut. Colors which are out-of-gamut needs to be mapped to in-gamut colors when a physical device is used to reproduce them.

The ICC specification describes four different rendering intents:

- the *absolute colorimetric* intent preserves the tristimulus of in-gamut colors
- the *relative colorimetric* intent rescales tristimulus values so that the white point of the color space to the white point of the PCS
- the *perceptual* intent is vendor specific, but it generally tries to provide the most pleasing perceptual result
- the *saturation* intent is vendor specific, but, as indicated by the name, it tries to preserve the saturation

The artist is supposed to choose a rendering intent from the available ones so that its result provides the best result, but unfortunately this depends both on the content of the image being converted and on its purpose. Usually the relative colorimetric and the perceptual rendering intents provide the best results.

Chapter 3

Color management in graphic software

3.1 Hardware support

3.1.1 Standard profiles

The color management model which has been proposed for the Web and is being adopted by Microsoft and Apple is centered on sRGB. This standard is based on the color reproduction capabilities of common CRT monitors, therefore it can provide a reasonable approximation for colors to be displayed on screen.

The adoption of sRGB on other hardware (scanners, printers and photo cameras) has been promoted to simplify the creation of color managed workflows, in which every image elaboration element handles both input and output in the standard color space.

This has helped in improving the consistency across multiple devices, but has some issues because of the limitations imposed by the sRGB gamut. They have been partially worked around by using bigger color spaces, usually Adobe RGB.

Having multiple standard color spaces requires the input hardware to tag the images with information about their color space and the software to handle the input color spaces and convert the image to the color space of the output devices. This is basically the same as a full color-managed workflow, therefore this approach defeats the purpose of a simple color managed workflow completely based on sRGB.

3.1.2 Embedded profiles

Even if some standard color profiles are designed to match the behavior of common hardware, they cannot provide a close match, because of their generality. To take advantage of the capabilities of specific hardware, vendors often provide color profiles for their products, usually by embedding them in the device.

These profiles describe the average behavior of a sampling of devices of the same model, therefore they provide valuable information about the device specifications. For example some hardware has primaries are more saturated than the sRGB ones, which result in a wider gamut. If no color transformation is performed, the visual result will not match the expected one. The embedded color profile defines the color transformation to be applied in order to obtain the expected result.

Even if embedded profiles do not provide the best possible color fidelity, they are useful because they improve the accuracy of color reproduction without additional operations by the user.

3.1.3 Calibration

The best possible color fidelity can be obtained by calibrating the devices, i.e. by measuring their behavior. Calibration is mainly used in professional color managed workflows, but its cost is becoming lower, hence non-professional users are beginning to perform it as well.

For input devices calibration is performed by using to acquire a standard reference, for example a scanner is usually calibrated by acquiring a page whose color content is known.

The inverse approach can be applied to output devices: they are used to synthesize some known component intensities and the resulting color is measured with a colorimeter or a spectrometer.

Calibration software can then convert the measured data to a color profile which describes the relationship between the values of the device components and the corresponding color. The quality of the profile obviously depends on the number of measures performed, so high quality profiles require more calibration time, but noticeable quality improvements can even be achieved with a relatively small number of samples.

An important limitation of calibrated profiles is that they only provide high accuracy for the specific conditions in which the device was calibrated. Different profiles need to be created if these conditions are modified. For example most printers mix 4 inks (cyan, magenta, yellow and black) on white paper to synthesize colors. Different ink cartridges and paper types result in different colors, so a calibration of the printer is needed in order to achieve accurate color reproduction whenever the combination of paper and ink cartridges changes.

3.2 Software

3.2.1 Professional graphic applications

People using graphic software professionally need color managed workflows to ensure that their images will look the same no matter what output device will be used to view or edit them. Color management is especially important when the images will used on different output devices (monitors vs. printers, different monitors), because images in device-dependent color spaces would cause the result to look different on each device.

Because of this, professional graphic software makes it possible to work and create documents using device-independent color spaces. In a color managed workflow every color is expressed in a device-independent color space, so that it can be converted in the output device color space when it has to be displayed or printed.

This issue is common to every representation of visual contents as both raster-based and vector-based editors have to handle this issue, thus most modern image formats have some support for color spaces. This means that usually the editor can embed in the image file an ICC color profile which describes the color space of the image.

Adobe Photoshop and Adobe Illustrator are well-known color managed applications for editing respectively raster and vector images.

GIMP, Inkscape and Scribus are examples of color managed image editors developed by the free software community. All of them use the LittleCMS library as color management system.

These applications probably need to communicate directly with the color management system to implement some of their most advanced operations. Nevertheless they use the Cairo graphic library to render their content, therefore the proposed extension would make it easier to display it correctly.

3.2.2 Document viewers

Most applications (browsers, image viewers, PDF readers, movie players, etc.) provide some document viewing capabilities and often have to be able to read and use correctly the color space information provided in the files they access. In order to provide a good user experience, documents displayed on screen and printed on paper are expected to look the same. For the same reason an image is supposed to look the same on any computer, no matter what monitor is displaying it.

This has two effects:

- there must be some “reasonably good” way to handle image data whose color space is not known
- image data in a given color space must be converted to the monitor color space

The first problem has been partially solved by defining a default color space [16]. On the Web, any image without an explicit color space is assumed to be expressed in sRGB. This is consistent with what some system already did and is being adopted for most parts of the modern graphic interfaces.

The solution of the first problem only reduces a very wide class of images to the state of being defined in a known color space, which is exactly the second problem. Different approaches have been adopted to handle this problem, but the main one currently is to leave it to applications.

For this reason some application use color management libraries, but unfortunately these libraries are not integrated in the graphic API used to display the images. Each application must be able to detect the monitor color space and to perform the color conversion of the images it wants to display. This makes color management more complicated and more problematic performance-wise.

Color management is starting to be available in browsers, because the major ones added (or enabled) it in recent releases: Microsoft Internet Explorer 9 and Mozilla Firefox 3.5 use the information provided by ICC color profiles embedded in the images they display.

Other software is moving towards color management as well, for example Poppler, the default PDF viewer on GNOME-based desktops applies ICC color profiles using LittleCMS. Instead at the time of this writing, Eye of GNOME, the default image viewer in the same environment, is unable to handle ICC profiles embedded in the images it displays.

3.3 Support in documents

Even editors and viewers are sometimes missing color management, many graphic document formats have some support for color profiles. The following section summarizes the color management facilities available in some of the most common formats on the Web (SVG, JPEG, GIF and PNG images) and in printing (PostScript and PDF documents, TIFF images).

3.3.1 PostScript

PostScript [10] is the page description language used by most printing devices. It is a Turing-complete language, which means that in theory it is as expressive as a language can be, but in practice it has some major limitations when compared to modern graphic formats, like the absence of support for transparency.

Beside from transparency, the color support in PostScript is fairly complete, even if not directly support ICC profiles, because it has some support for color management both for the input and for the output.

Input colors, such as solid colors, shadings, embedded images, always have an associated color space, usually called CSA, Color Specification Array.

Colors can be expressed in the device-dependent RGB color spaces using red-green-blue intensities or hue-saturation-brightness values:

```
/DeviceRGB setcolorspace red green blue setcolor
red green blue setrgbcolor
hue saturation brightness sethsbcolor
```

Grayscale and CMYK components can be specified in a similar way:

```
/DeviceGray setcolorspace gray setcolor
gray setgray
/DeviceCMYK setcolorspace cyan magenta yellow black setcolor
cyan magenta yellow black setcmykcolor
```

Device-independent color spaces are described by specifying a conversion to CIEXYZ. For example, a color in sRGB can be set using these commands:

```
[ /CIEBasedABC
    << /DecodeLMN
      [  dup 0.03928 le
            12.92321 div
            0.055 add 1.055 div 2.4 exp
          ifelse
         bind dup dup
      ]
      /MatrixLMN [0.412457 0.212673 0.019334
                  0.357576 0.715152 0.119192
                  0.180437 0.072175 0.950301]
      /WhitePoint [0.9505 1.0 1.0890]
    >>
] setcolorspace
red green blue setcolor
```

Depending on the number of components and the formulation of the color space, it can be defined with:

- `CIEBasedA`, for color spaces with just one component,
- `CIEBasedABC`, for simple 3-components color spaces,
- `CIEBasedDEF`, for 3-components color spaces which need a color look-up table,
- `CIEBasedDEFG`, for 4-components color spaces

Transformation to output components can be expressed in a similar way, by specifying the mapping from CIEXYZ to the device-specific output color space, using a CRD (Color Rendering Dictionary). Just like CIE-based input color spaces, CRD describe the color transformation in a PostScript-specific way, not woth a standard ICC profile.

Because of limitations on CSA and CRD structures, the conversion between ICC color profiles and CSA/CRD color space representations is only possible for color spaces with 1, 3 or 4 components, but this is not a major problem, because these are the most widely used color spaces.

In addition to this, PostScript provides support for some special color spaces strictly related to the PostScript drawing model or to printing (for example support for special non-chromatic inks). Although this can be useful for some professional users, they have been considered out of the scope of this extension.

3.3.2 Portable Document Format

PDF, standardized in [8], is a page description language based on PostScript, but the absence of flow control commands makes it easier to parse and interpret. Conversely, the availability of modern graphic primitives, like transparent colors and blending, permits many graphic effects that PostScript was unable to express in a simple document.

Color management support in PDF is similar to that provided by the PostScript language. `/DeviceRGB`, `/DeviceGray` and `/DeviceCMYK` are available and serve the same purpose as in PostScript. Instead of the PostScript-specific CSA and CRD color space representations, PDF supports CIE-based color space using standard ICC color profiles:

```
100 0 obj
  [ /ICCBased 101 0 R ]
endobj

101 0 obj
  << /N 3 % 3 components
     /Length 1605 % length of the profile data stream
     /Filter /ASCIIHexDecode % filter used to decode the stream
  >>
stream
00 00 02 14 ... hex encoding of the ICC profile ... 03 90 >
endstream
endobj
```

In addition to this, PDF specifies the behavior of color conversion with respect to graphic operations, because in general color transformations do not commute with color blending and interpolation. PDF specifies that interpolation operations are to be performed in the color space of the object being interpolated, whereas blending of an object on a destination is to be performed in the color space of the destination. This is, in fact, the model adopted in the extension of the Cairo library, thus chapter 4 will analyze it in greater detail and will show that it can achieve the maximum possible generality.

Just like PostScript, PDF also has the possibility of print-oriented color spaces, to support special components, like golden or opaque inks, to be expressed in the document and used or emulated on the output device depending on their availability. These features are not supported by the proposed color space extension, but they might be added as future enhancements.

3.3.3 Scalable Vector Graphics

The Scalable Vector Graphics [7] standard specifies an XML-based graphic format. Although it is a vector format like PostScript and PDF, SVG has many differences with respect to them.

In particular, with regard to color support, SVG has no concept of device-dependent colors, as it implicitly defaults to using the sRGB color space. ICC profiles can be used to specify input colors in arbitrary color spaces, but color operations like interpolation and blending can only be performed in sRGB or in linear RGB (i.e. the sRGB color space with linear tone reproduction curves).

There is ongoing work to define some color management features as part of the SVG Color specification [4], which adds support for device-dependent color spaces and other predefined color spaces, like CIELAB and CIELCH. This specification also extends the color spaces that can be used for interpolation to sRGB, linearRGB, CIELAB and CIELCH, but it does not allow ICC-based color spaces to be used for this purpose. Although the allowed color spaces are usually sufficient, the impossibility of expressing operations performed in other color spaces is still quite limiting from the point of view of the proposed extension.

It should be noted that SVG is a recent format and is still evolving, hence it is definitely possible that it will provide the same (or even more extensive) color support as PDF.

3.3.4 Raster images

Even though raster images are usually stored in some compressed format which makes their structure more complex, they are conceptually described by a rectangular grid of color samples, each expressed as a n-uple of component values. This is typically accomplished by specifying the RGB values in each point, but this approach does not allow proper color management.

When no color space is specified, images are usually assumed to be in sRGB or, depending on their purpose, in linear RGB, but most modern raster file formats support the embedding of an ICC color profile as part of the image metadata, which make it possible to explicitly define the color space in which the sample components are.

In particular, the ICC specification [3] describes how to embed ICC profiles in JPEG, GIF and TIFF images, while the support for color profiles in PNG is explicitly described in its specification [6].

3.4 Graphic APIs

3.4.1 Quartz

Quartz is the core graphic API on MacOS X and iOS. It is a vector rendering API, based on the PDF standard, which exposes through a C interface most of the PDF features, including color management.

Specifically, Quartz uses ColorSync, the MacOS X color management system, to handle color profiles and perform color transformations.

Because of this, Quartz implements most of the useful parts of color management: it supports device-dependent color space and thanks to ColorSync it can also parse and use ICC color profiles both for input and output, under the additional constraint that PDF must have native support for the profile. This restricts the color profiles it can handle to those with 1, 3 or 4 components, but it has already been noted that this condition is satisfied by most common profiles. If this condition is satisfied, the creation of a color, given its components and the ICC profile is very simple:

```
CGFloat components[] = { /* the color components */ };
color_space = CGColorSpaceCreateWithICCProfile (icc_profile);
color = CGColorCreate (color_space, components);
CGColorSpaceRelease (color_space);
```

Colors in device-dependent color spaces can be created in a similar way, by using the appropriate color space creation functions:

- `CGColorSpaceCreateDeviceRGB ()`
- `CGColorSpaceCreateDeviceGray ()`
- `CGColorSpaceCreateDeviceCMYK ()`

A notable feature of Quartz is that none of its functions allow to create a colored graphic object (colors, gradients, images) without explicitly specifying its color space. This effectively guarantees that every object always has an associated color space, without needing a default color space.

Quartz is also quite flexible: even if it is based on PDF, it can be used to draw on raster images and it is, in fact, the graphic API used for the GUI of most MacOS X applications. Nonetheless it retains the ability to use the same sequence of commands to generate vector images in the form of PDF documents.

3.4.2 GDI+/WPF

The graphic rendering API on Microsoft Windows has evolved over the years, but it does not yet provide the same level of color support as Quartz. Both GDI+ (Graphics Device Interface), the legacy API, and WPF (Windows Presentation Foundation) provide extensive vector rendering operations, including filling and stroking complex shapes with "brushes" which can represent both solid colors and more complex pattern, but they offer limited support for color management.

These APIs supports both standard ICC profiles and WCS profiles, which are a different way to describe color conversions. Although they have some additional capabilities, they are not widely used, because they are only supported by the Windows Color System [11] and cannot be embedded in documents without being converted to ICC profiles.

The drawing model provided by GDI+/WPF and WCS is quite similar to the SVG one with respect to colors, because it imposes strict limitations on what color spaces can be used and which objects can be color managed:

- GDI+ usually works in device-dependent RGB. It can enable color management only on device contexts, i.e. output images. It can set an output color profile and an input color space, with some additional restrictions: input color spaces can only have a simple structure (like sRGB), while output profiles must satisfy different constraints depending on the OS version.

- WPF makes it possible to specify colors in device-dependent RGB, in sRGB or in a color space defined by an ICC profile, but it does not allow choosing the color space in which operations are performed.

The constraints imposed by these APIs are about as tight as the SVG ones, hence the same consideration applies: artists will probably have no trouble working in the allowed color spaces, but implementing color managed applications on top of this drawing model might require some explicit color conversions. Although these color transformations can be performed using the color management APIs (Windows Image Color Management or Windows Color System) explicitly, they do not allow the flexibility which can be obtained performing drawing operations in arbitrary color spaces.

3.4.3 OpenGL

OpenGL is a cross-platform rendering API which abstracts device-specific and OS-dependent capabilities and exposes simple drawing primitives, like triangles. OpenGL closely matches the hardware functions, therefore its first versions only allowed for a fixed rendering pipeline, but its flexibility has greatly increased thanks to the introduction of a programmable pipeline.

OpenGL offers no direct support for ICC color profiles or for any color space besides device-dependent RGB. This turns out to actually cause some practical problems, because improper color handling has some issues, mainly related to the mismatch between the linear color space and the non-linear tone response curves of common monitors.

Two extensions have been proposed in order to fix this problems:

- `EXT_texture_sRGB` allows sRGB to be used on textures, i.e. as input color space

- `EXT_framebuffer_sRGB` allows sRGB to be used on framebuffers, i.e. as output color space

When these extensions are not available or when the desired color space is not sRGB, simple color transformations can be performed using the programmable pipeline, as described in [12], but this is likely to have a negative performance impact. Although the same approach would make it possible to support generic ICC profiles, it is far from practical, except for simple profiles.

3.4.4 Qt

Qt is a cross-platform application and UI framework. It provides a rendering API which wraps multiple possible targets (platform-specific graphic libraries, raster images, SVG files) and exposes an unified interface to draw on them.

The QPainter rendering API resembles GDI, as it can draw complex shapes, but its color patterns have no color management at all. Solid colors can be specified in many different device-dependent color spaces (RGB, CMYK, HSL and HSV), but operations are always performed in device-dependent RGB.

Part of these limitations are caused by the portability requirements of the Qt framework: the compositing operations are implemented using platform-specific capabilities, which usually do not include color management.

3.4.5 Cairo

Cairo is an cross-platform graphic library with support for multiple output devices. It officially supports rendering to Xlib, Quartz, GDI, PDF, PostScript and SVG and has experimental backends for OpenGL, OpenVG, XCB, Qt and other graphic APIs.

Cairo abstracts backend-specific operations and exposes PDF-like drawing commands. These commands are converted to native operations when possible or performed by a fallback system if the backend does not support the operation.

The observations in section 3.3, make it apparent that all the officially supported vector document formats have some color space support, with PostScript and PDF already providing extensive color management features and SVG evolving in that direction. Conversely, GUI-related backends have limited color management capabilities and Cairo is no exception.

Even if the drawing model exposed by Cairo is based on PDF, the library originates from an abstraction of low-level X operations. The X server can perform vector-based drawing through the RENDER extension, but it only supports simple shapes (triangles and trapezoids) and it can only operate in device-dependent RGB.

Cairo generalizes these operations to PDF drawing operations, like filling and stroking complex shapes, but does not yet provide color management features. There have been some attempts at improving this by adding CMYK, which is of particular interest for printing, and extending the support for planar YUV surfaces, but they didn't get merged in the official library.

A detailed explanation of the Cairo drawing model, with specific focus on color is given in section 4.1.

3.5 Desired features

The implementation of color management in Cairo would solve some issues that are, in fact, special cases. The following features have been reported multiple times as possible enhancements and each has a working or a partial implementation, but so far they did not get merged in the Cairo library, mainly because the problem of extending the Cairo drawing model was not addressed properly.

3.5.1 CMYK color spaces

Even if it has the same limitations and defects of every device-dependent color space, the support for CMYK has been repeatedly requested as new feature in Cairo. This in not surprising, because without support for device-dependent CMYK, it is basically impossible to implement a color managed application on top of Cairo, even if the application calls into the color management system directly.

A device-dependent CMYK color space would make it possible to implement color management on top of Cairo because CMYK images let the application have accurate control over the intensity of the inks used by the output device. This is needed in order to respect the color computed by the color management system. If the CMYK components were converted to RGB before being sent to the output device, the ink intensities used when printing would likely not be the same as those originally computed, hence the resulting color would likely be different.

In addition to this, some standard printing workflows (like PDF/X-1A) require all the color information to be expressed in device-dependent CMYK, thus the ability to create and handle documents in CMYK would allow to use Cairo in these workflows.

3.5.2 YC_bC_r color spaces

The another attempt at adding some color space support to Cairo aimed at improving the interaction between Cairo and video formats. To improve the compression, video frames are commonly represented in a YC_bC_r color space, which Cairo is unable to handle. Because of this, using Cairo to impose text, video controls or a logo on it requires a conversion of the video frames to RGB; then the application can draw on them with Cairo and either display the RGB frames or, depending on what was the destination, it might need to convert back to YC_bC_r.

Color space support makes it possible to avoid the (potentially double) color conversion between YC_bC_r and RGB. When the color conversion cannot be avoided, it will be performed by Cairo using the appropriate facilities if they are provided by the destination backend. For example this means that hardware-accelerated color transformations might be possible using GL shaders for most color spaces commonly used in video formats. This is likely to improve the performance of the conversion because it avoids moving the video data between video memory and main memory and performs the computation in parallel on the GPU.

3.5.3 Gamma handling

Another issue which is often ignored but is actually very important for a high-quality imaging system is proper gamma handling. This affects a very wide range of applications, because basically every graphic element besides opaque solid colors is affected.

Most physical output devices are non-linear, so the data sent to them must be transformed to compensate their behavior. This is sometimes performed implicitly by performing the color computations in the output color space, for example by using sRGB input images, composing them in sRGB and sending them to a monitor (whose color space is assumed to approximately be sRGB). If the “compose” step ignores the non-linearity of the sRGB color space, the resulting image will often look incorrect.

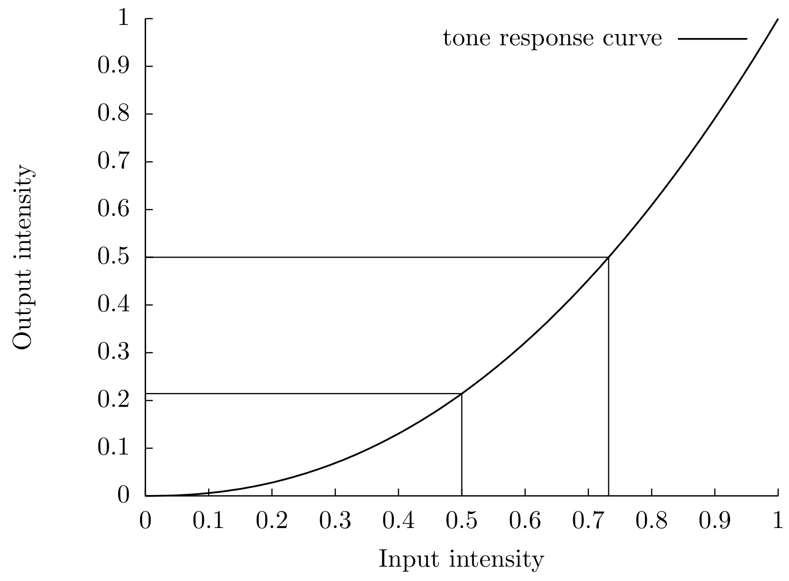

Figure 3.1: The relationship between linear and sRGB intensities

This problem is especially evident, for example, when scaling images [1]. If a black and white checkerboard image is scaled down 2:1, the expected result is a perceptually medium gray, but most programs will instead compute a “numerical” medium gray, which looks darker. On a typical display, assuming the sRGB color space, the a 50% input intensity would result in an output whose lightness is about 20% of the white. In order to produce a 50% output gray the input intensity should be about the 75%, as shown in figure 3.1.

This problem can be avoided by working in linear space, but both input and output images are usually expected to be in the sRGB color space, therefore a color conversion is needed.

Chapter 4

Design

4.1 Cairo drawing model

4.1.1 Cairo objects

Cairo has few public types, which basically allow the user to choose a source, a destination and how to transfer the color from the source to the destination.

Surfaces represent vector or raster images in Cairo. They can be used both as destinations and as sources, but they don't usually receive commands directly from the user.

To draw on a surface, a *context* must be created on it. The context contains the drawing status and it is the object which supports the drawing commands.

To choose the source which will be painted on the destination, a *pattern* must be created and set in the context. Patterns can represent a solid color, a gradient or an image.

The source is not entirely transferred to the destination. Each context can have a *clip*, which limits the portions of the destination surface changed by any graphic operation.

In addition to it, each graphic operation defines a *mask*, which influences "how much" color will be transferred from the source from the destination.

The implementation is usually more efficient, but conceptually all the drawing commands have a source, which is modulated by a mask, cut by the clip and then composited on the destination.

4.1.2 Contexts

Cairo drawing operations are performed on a context, which represents the state of the rendering device. The context augments the destination surface with a state, which makes it possible to describe a complex drawing operation using a sequence of simple commands.

These commands include functions to set up the rendering options, like antialiasing and error tolerance, to transform the coordinate system in which the commands are executed and to set up the source pattern.

In addition to these, contexts have commands to define a path composed of linear segments and Bezier curves, which can describe complex shapes.

A path can be used used to `clip` the context, further drawing operations on the context will only affect the regions which are inside the path.

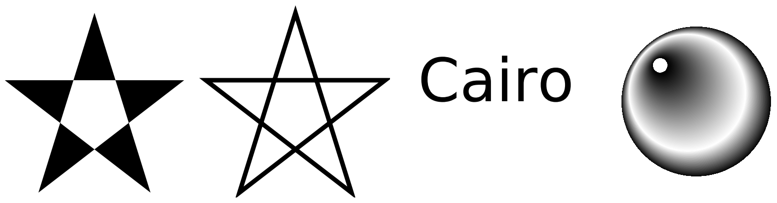

Figure 4.1: Some examples showing Cairo drawing operations. From left to right: fill, stroke, text, mask (through a radial pattern).

A path can also be used in drawing operations. If a path is `fill`ed, the color of the source pattern will transfer on the destination surface in the region inside the path. If a path is `stroke`d, its segments will be replaced with thick lines and the source will transfer through them.

There are three more notable drawing operations, which ignore the path: an operation to write `text`[1], an operation to `paint` the source over the whole destination and an operation to use the alpha channel of a pattern as the transfer `mask`.

Conceptually every drawing operation defines a transfer mask which controls the amount of color transferred to the destination surface. For a `mask` operation, this is obvious, but other operations can be seen as special cases of the `mask` operation:

- `paint` uses a solid opaque mask
- `fill` uses a surface mask whose alpha channel is the coverage of the path, i.e. 0 outside the path and 1 inside the path
- `stroke` uses a surface mask whose alpha is the coverage of the stroked path (the path whose linear elements have been replaced with thick segments)
- `text` uses a mask whose alpha is computed according to the font engine rendering, in a way which is equivalent to that of a fill mask

[1]Technically there are two text operations: `show_text` and `show_glyphs`. The second one allows the calling application to layout each symbol composing a text string, therefore is considered the true text API, whereas the first one is a "toy" API, useful for testing and demos.

4.1.3 Patterns

Cairo supports four types of patterns[2], which can be used to describe the color of every point of the plane. They are mostly based on the shading patterns defined in the PDF specification [8]. An example for each pattern type has been `painted` in figure 4.2.

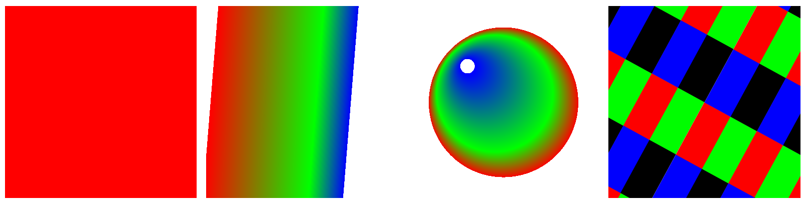

Figure 4.2: The four pattern types supported by Cairo. From left to right: solid, linear, radial and surface pattern.

The simplest patterns are *solid* patterns. They have the same color everywhere.

Cairo *linear* patterns (known as axial shadings in the PDF specification) are described by two points and a piecewise linear color function. The color function defines the color of every point on the line connecting the two points. The color of every other point of the plane is the same as that of the nearest point on the line.

The appearance of *radial* patterns has changed over time, but the latest definition matches that of PDF radial shading. The pattern is defined by two circles and a piecewise linear color function. The color function defines the color of every circle in the family of circles obtained by linear interpolation of the two reference circles. Any point on the plane which does not belong to any of the circles of the family is transparent; any point which belongs to more than one circle has the color of the circle with the highest interpolation parameter.

Surface patterns use surfaces, i.e. arbitrary images, to color every point of the plane. Surfaces usually only describe a bounded rectangle, therefore some additional information is needed to define the color of points outside the surface. This information is provided by an *extend rule*, which map each point either to the color of a point of the image, or to the transparent color. It should be noted that in general interpolation is needed to compute the color of points inside the surface boundaries, because raster surfaces sample the color on a discrete set of points.

[2]Recent development snapshots of the Cairo library contain a fifth pattern type, which corresponds to the tensor-product patch mesh defined in PDF and PostScript.

4.1.4 Surfaces

One of the important features of Cairo is its ability to draw on many different output targets. The officially supported targets include raster image buffers, standard vector formats like PDF, PostScript and Scalable Vector Graphics, and platform-specific drawing systems, like GDI, Quartz and X.

In Cairo these output targets are supported using a system of surface backends. Each backend can translate the Cairo graphic operations to commands which are appropriate for the target. If this is not possible because the operation is not supported by the target, it falls back to a generic implementation which performs the operation in Cairo. The result is then passed to the target.

The target is exposed through the Cairo API as a surface object, therefore different backends provide different functions to create surfaces. When a surface object is created, its output target and the appropriate conversions between Cairo and native graphic operations are set up. In addition to this, some backends provide additional functions to access some target-specific operations.

It is worth noting that besides from being used as destinations for drawing, surfaces can also be used to create patterns, which makes it possible to use them as sources for graphic operations. It is quite common to create temporary surfaces to be drawn on the real destination.

4.1.5 Cairo graphic pipeline

A simplified drawing pipeline for Cairo can be imagined as performing for each pixel the following steps:

- Compute the *clip*. The clip is defined as the intersection of geometric shapes and carries no color information, so the result of this computation is a coverage value. If the pixel is outside the clip (i.e. its coverage is 0), the destination will not be affected.

- Compute the *mask*. The mask can be defined as a geometric shape or as a pattern. It provides another "coverage" value, whose effect depends on the operator.

- Compute the *source*. The source is a pattern which provides a color. This color can have any opacity.

- Read the *destination*. Conceptually no computation is performed, because the whole pipeline is working in the destination coordinate space.

- Composite the values according to the *operator* definition.

- Store the *result* in the destination.

Most of the complexity of the rendering is in the the computations performed in the first three steps. When they transform a shape into coverage values, they involve rasterization of the shape. Otherwise they evaluate the color of a pattern for each pixel.

4.1.6 Compositing in Cairo

Just like most graphic libraries, Cairo defines several different ways to transfer color from a source to a destination. These functions are called *operators* and include Porter-Duff compositing operators and PDF blend modes.

In [15], Porter and Duff define the compositing operators to be the functions which use the alpha component of a pixel as a "shape". This makes it possible to define the result as the union of the regions where source and destination do or do not overlap, giving the 12 classic compositing operators.

The PDF standard [8] defines other operators, which all have the same behavior for the alpha component, but allow more complicated interactions between the color components of source and destination.

Cairo adds two more operators to these, borrowing them from the RENDER specification [14]. In addition to this, Cairo defines some rules to generalize the operators to partially opaque clips and masks.

Most of the operators can be defined mathematically in a simple way:

$$\begin{aligned} result_\alpha &= op_\alpha(clip, mask, source_\alpha, destination_\alpha) \\ result_c &= op_c(clip, mask, source_\alpha, destination_\alpha, source_c, destination_c) \end{aligned}$$

where:

- $clip$ is the shape value provided by the clip path
- $mask$ is the shape value provided by the mask path or pattern
- $source_\alpha$ is the transparency of the source pattern
- $source_c$ is the c color component of the source pattern
- $destination_\alpha$ is the transparency of the destination surface
- $destination_c$ is the c color component of the destination surface
- $result_\alpha$ is the transparency of the destination (which will replace the old $destination_\alpha$ value in the destination surface after the whole computation has been performed)
- $result_c$ is the c color component of the result (which will replace the old $destination_c$ value in the destination surface after the whole computation has been performed)
- op_α and op_c are the operator compositing functions

Operators which can be expressed with this formulas (using appropriate op_α and op_c functions) are called *separable* operators, because each color component can be computed independently. This property provides a very simple way to apply separable operators to any source and destination pair, as long as their components match (i.e. they are in the same color space).

There are some operators which cannot be defined with such a formula, because they work on the color as a whole. These operators are defined in the PDF standard as *nonseparable* blend modes and conceptually represent the source and destination in HSL and compute the result taking some HSL components from the source and the other from the destination.

The following table summarizes the definition of the PDF nonseparable blend modes:

Operator	Hue	Saturation	Luminosity
HUE	source	destination	destination
SATURATION	destination	source	destination
COLOR	source	source	destination
LUMINOSITY	destination	destination	source

Nonseparable operators are defined on the color as a whole, thus they do not have a straightforward generalization to arbitrary color spaces. The PDF standard specifies the behavior for RGB and CMYK (and, implicitly, for grayscale), but it does not extend them to other color spaces.

4.2 Support for color spaces

4.2.1 Drawing model

Considering the graphic pipeline, there are three main stages where color-related operations happen:

- computation of the source
- compositing
- storing the result to the destination

The first assumption is that every image is specified in an appropriate color space, so that it is possible to associate a color to the n-uple of component intensities.

The computation of the source involves interpolating between color samples. Color interpolation is not invariant with respect to the color space it is performed into, so the color space in which it is performed must be specified. Without any loss of generality, the color space of the source can be assumed to be the one in which we want to perform the interpolation. This means that applications should create patterns with an appropriate color space.

Compositing takes colors from source and destination, which can be in different color spaces and computes a result. Even if it looks like the compositing space must be specified, a reasonable choice is to always use the destination color space. This makes the conversion from the destination color space to the compositing color space an identity function.

Moreover, since the result is already in the destination color space, it can be stored with no further conversion.

The very common situation of images in device-dependent color spaces is handled by performing the operations in the device-dependent color space if color conversion can be avoided. This is needed in order to be consistent with the previous behavior.

If color conversion from a device-dependent color space is used, the results are backend-dependent (i.e. they can be different depending on the output being an SVG document, a PostScript file or a raster image). The reason for this behavior is that some backends allow device-dependent color spaces to be used together with device-independent color spaces.

4.2.2 Advantages and limitations

The apparently very limited freedom in color space choice does not actually stop the application from performing source interpolation and compositing in any color space.

If a pattern must be interpolated in a color space which is not the same as the one in which the samples are specified, the samples can be converted to the interpolation color space. These new samples can be used to create another pattern which performs the interpolation in the desired color space.

If the application wants to composite in a color space which is not the same as the desired output color space, it can simply use a temporary surface. The temporary surface will have the "composite color space" (although it will actually be a destination color space). It will receive all the drawing operations, then it will be drawn on the "true" destination, to convert its colors to the desired color space.

These choices provide several advantages:

- The components which form the output of a source do not depend on the color space in which the interpolation occurs, but only on the source samples. This means that the functions which performs the interpolation can be color space agnostic.
- Destinations are always used in their own color space. If the color was repeatedly converted to another color space and back to the destination color space, it would accumulate rounding error.
- The number of color conversions performed is at most the same as those which would have to be performed if the interpolation space

and/or the compositing color spaces were independent from the source and destination color spaces. In some cases this performance advantage comes at a memory cost, because there is a trade-off between additional computations and the additional memory required by temporary surfaces.

- There is only one place where color conversions happen, i.e. in the compositing operations. This is important in order to control the complexity of the system. Moreover it means that a single parameter (the *rendering intent*) is sufficient to specify the desired color mapping between the two color spaces.
- This design is consistent with the color handling model defined in the PDF specification. This makes it inconsistent with the drawing model used in the SVG specification, but it is still possible to implement the latter using temporary surfaces.

4.2.3 Public API

The proposed API extension adds a color space type and functions to use it; this section contains an overview of the new functions and of their compliance with the design which has been chosen. Appendix 6.4 contains the prototypes of the new functions and their technical documentation.

The `cairo_color_space_t` type represents color spaces. It can be instantiated to get objects representing device-dependent or device-independent color spaces:

- `cairo_color_space_create_device_rgb()` creates a device-dependent RGB color space. This is the color space in which Cairo performs all of its computations by default.
- `cairo_color_space_create_device_gray()` creates a device-dependent grayscale color space. Some Cairo backends were implicitly using this color space in order to perform some optimizations of their output.
- `cairo_color_space_create_device_cmyk()` creates a device-dependent CMYK color space.
- `cairo_color_space_create_icc()` creates a device-independent color space from an ICC profile.
- the `NULL` value represents the 0-components device-dependent color space. Although this color space is unable to express any color, it is useful when a pattern or a surface carries no color content and is completely defined by its transparency.

Most public Cairo types are reference counted and `cairo_color_space_t` is no exception. Consistently with other Cairo types, color spaces have a reference count of 1 upon creation and are immediately destroyed as soon as they reference count becomes 0. Instances of `cairo_color_space_t` cannot be modified, therefore a copy or clone operation is not provided nor needed, as it is equivalent to `cairo_color_space_reference()` .

Since they cannot be modified, their status is only set upon creation and it can only indicate a failure in the instantiation, either because of an invalid input profile or because the available memory is insufficient.

In addition to these functions, new functions to create patterns and surfaces with an arbitrary color space have been provided.

The functions to create a pattern which performs interpolation in a given color space are the following:

- `cairo_pattern_create_color()` creates a solid pattern. The components of the pattern color are expressed in the interpolation color space.
- `cairo_pattern_create_{linear|radial}_with_color_space()` creates a gradient with a given interpolation color space. The color function of the gradient pattern must be defined by adding stops whose color is represented by components in this color space.
- `cairo_pattern_add_color_stop_color()` adds a color stop to the color function of a gradient. It does not provide an input parameter for a color space because the components must be specified in the pattern interpolation color space, which is chosen at pattern creation time.
- `cairo_pattern_create_surface()` can be used to create a surface pattern. This is not a new function and it does not accept any color space as input. The interpolation color space of the resulting surface pattern will be the same as the color space of the surface.

Surface creation functions in Cairo are usually backend-dependent, but there are some special functions which create a new surface from an existing one, trying to preserve the backend type. Two of these functions have been extended in order to support a color space parameter:

- `cairo_surface_create_similar_with_color_space` creates a new surface with a given size, color space and content (color, transparency or both).
- `cairo_push_group_with_color_space` replaces the target of the operations performed on a context with a new temporary surface, which is created with the specified color space and content.

Backend-specific surface creation APIs should be extended to support the creation of surfaces in arbitrary color spaces. Cairo surface formats should be extended accordingly, to permit surfaces whose color space has more than 3 components.

Finally the extension provides a function to control the color transformation, by specifying the rendering intent: `cairo_set_rendering_intent()` . This function makes it possible to select one of the ICC-specified rendering intents. This controls the transformation between two different color spaces, in particular with respect to the gamut and the white point mapping.

4.3 Usage

The new functions can be used together with the old API, which always implicitly use device-dependent RGB, but can also replace it in order to have the same code for any color space.

For example a solid half-transparent yellow pattern can be created with the old API:

```
cairo_pattern_t *yellow;
yellow = cairo_pattern_create_rgba (1.0, 1.0, 0.0, 0.5);
```

or the same object can be created with the new API:

```
cairo_color_space_t *deviceRGB;
cairo_pattern_t *yellow;
double components[3] = { 1.0, 1.0, 0.0 };
deviceRGB = cairo_color_space_create_device_rgb ();
yellow = cairo_pattern_create_color (deviceRGB, components, 0.5);
```

Starting from this code, it is possible to create the sRGB yellow changing only the color space construction:

```
cairo_color_space_t *sRGB;
cairo_pattern_t *yellow;
double components[3] = { 1.0, 1.0, 0.0 };
sRGB = cairo_color_space_create_icc (sRGB_data, sRGB_len);
yellow = cairo_pattern_create_color (sRGB, components, 0.5);
```

These patterns can be used in any context and will be composited according to the color space of the target surface. Sometimes it is desirable to perform the compositing in a different color space, for example it is common to perform the graphic operations in linear RGB and to have sRGB output.

This can be accomplished as follows:

```
/* cr is assumed to be an empty cairo context */
/* create a temporary surface on which the compositing
 * operations will be performed in linear RGB */
cairo_color_space_t *linearRGB;
linearRGB = cairo_color_space_create_icc (lRGB_data, lRGB_len);
cairo_push_group_with_color_space (cr, linearRGB,
                                   CAIRO_CONTENT_COLOR_ALPHA);

...
/* drawing operations */
...

/* copy the content of the group to the target surface */
cairo_pop_group_to_source (cr);
cairo_paint (cr);
```

The last paint operation will happen in sRGB, but it will actually simply convert the group surface to the destination color space. This only works if the destination is empty at the beginning if the code. If the destination had some content, additional operations to copy it to the group surface and then to clear it before painting the group content would be needed.

4.3.1 Initial approach

The API which had been initially proposed also provided a public color type. This is not actually a new type in Cairo, because internally Cairo already defines it as a private type which stores the alpha, red, green and blue components of the color in a device-dependent RGB color space. In the proposed extention, instead, colors were represented by $(color\_space, components)$ pairs. This representation guarantees that whenever a color is being used, its color space (and thus its colorimetrical meaning) is known.

Unfortunately exposing Cairo colors through the public API caused some issues:

- The operation which adds a color stop to a gradient was defined to accept a color type, instead of sample components. This made it possible to add color stops in multiple different color spaces, which caused the color interpolation function to be ill-defined.
- Cairo internally creates some temporary colors on the stack in order to make sure that their creation cannot fail. These objects cannot respect the normal reference counting behavior, thus they need to be cloned instead of referenced, whenever they must be pinned.
- Reference counting and error management made the implementation of the new color type more complicated than the old one, without practical advantages.

For this reason, the public color type was removed and the API was modified to only accept color components and their color space, where appropriate. This approach preserves the guarantee that colors are always specified by a $(color\_space, components)$ pair, but avoids the problems introduced by the public color type.

Chapter 5

Implementation

5.1 Image surfaces

Although Cairo is a vector drawing library, its most important backend is raster-based. The *image backend* performs the drawing commands on a memory buffer which represents a raster image. This backend implements every feature available in Cairo and can therefore rasterize any sequence of drawing commands. Any other backend does not need to be able to perform every operation, because for unimplemented operations Cairo will fall back to image.

The implementation of the drawing operations in the image backend is actually split in two parts:

- shape rasterization is performed in Cairo. This includes the stroking, which transforms lines to polygons, and scan conversion of the polygons to raster images.
- pattern rasterization and compositing are performed by Pixman, the pixel-manipulation library.

As pointed out in section 4.1.5, the shape-related operations carry no color information, therefore they are basically unaffected by the addition of color spaces handling. Instead, the biggest part of the changes needed to add support for color spaces to the image backend was inside the Pixman library.

Pixman is the library which performs the computations needed to get the color of a source pattern and composite it on a destination. Even if the extension was designed to require minimal changes, both these parts need to be modified to support images with an arbitrary number of components. In addition to this, the source needs to be converted to the destination color space whenever the two spaces don't match.

To implement these changes, the functions which create patterns were extended with a color space parameter and the compositing function was augmented with the color transformation.

On top of this, the computations were rewritten to use a floating point representation instead of the original fixed point. This was needed to preserve acceptable accuracy, because image interpolation and compositing must be performed on premultiplied components, but color transformation ignores the alpha and works on straight components[1].

The drawing pipeline places color transformation between image interpolation and compositing, therefore premultiplied components must be "unpremultiplied", then premultiplied again. Unfortunately, the premultiplied representation implies that on non-opaque colors, color components have a very limited range, which can cause big quantization errors. This is usually not a problem, because the components are directly used to composite the source image on the destination, without any intermediate operations. Instead, if the color need to be converted between two color spaces, the quantization error can be amplified.

For this reason, color computations were rewritten to provide results in floating point. This representation provides quantization levels which becomes finer toward 0, which makes it possible to have a very small quantization error even when using the premultiplied representation.

5.2 Vector surfaces

5.2.1 PDF

Every graphic object in a PDF is tagged with a color space, but Cairo has no support for color spaces, hence the current implementation of the cairo-pdf backend uses device-dependent RGB and grayscale color spaces for all PDF objects.

It is quite easy to extend the PDF backend to support 3-components ICC color spaces as they can be embedded and their reference can be used where the device-dependent RGB color space was used. This means that the profiles used in the document need to be tracked and added to the PDF stream, to avoid needlessly duplicating them.

In order to extend the PDF backend with support for color spaces with more than 3 color components, the internal representation of patterns and the functions that write them as objects in the PDF document must be extended accordingly.

[1] *Premultiplied* colors are represented by the n-uple $(\alpha, \alpha c_1, ..., \alpha c_n)$, or in the common case of RGB $(\alpha, \alpha r, \alpha g, \alpha b)$, opposed to the *straight* representation $(\alpha, c_1, ..., c_n)$ (or (α, r, g, b)). The compositing formulas for premultiplied colors require no division and are generally simpler, therefore compositing is typically implemented on premultiplied components. Image interpolation introduces fringes with the color of the image border (usually black) if it is performed on straight components.

For color spaces or drawing operations which are not supported by the PDF specification, a fallback image is generated and the raster object is used instead of vector operations, thus in these cases the PDF backend relies on the image backend to perform the drawing operation.

5.2.2 PostScript

PostScript is strictly related to PDF and the implementation if the two backends in Cairo reflects this. In fact, the cairo-ps backend declares some function to make the differences between the two formats as small as possible.

Just like PDF, PostScript graphic objects have a color space entry which indicates the meaning of their components. The cairo-ps backend only uses objects in device-dependent RGB, thus internal objects have 3 components. To support device-dependent color spaces with a different number of components (the native device-dependent grayscale and CMYK color spaces), the data structures that represent colors in the PostScript backend must be generalized.

Moreover, to support device-independent color spaces, the input ICC profiles used to specify the color space for source patterns must be converted to *Color Specification Arrays*. Although PostScript can express any ICC profile (with 1, 3 or 4 components) as a CSA, their representation is not the same. A similar problem happens for output ICC profiles set on the surfaces: they need to be converted to PostScript *Color Rendering Dictionaries*.

Converting ICC profiles to PostScript-specific color space representations is a common problem, that color managed applications need to handle if they want to support PostScript output. For this reason most color management systems provide facilities to perform the conversion between PostScript color spaces and ICC color profiles.

It should also be noted that PostScript only provides a limited subset of Cairo operations, in particular it only supports opaque patterns and cannot change the compositing operator, hence whenever these are needed, an image fallback is generated.

5.3 Quartz surfaces

Quartz is the native MacOS X drawing API. Just like Cairo, it provides PDF-like drawing commands, but Quartz always requires the colors to be specified in some color space.

The cairo-quartz backend implements the Cairo API on top of Quartz. The current Cairo implementation has no color space information, therefore the Quartz backend ignores the color management features provided by MacOS X. This is accomplished by always using the device-dependent RGB color space.

With the new color space extension, it is possible to directly use Quartz color management features by transforming Cairo color space objects into Quartz color spaces and using them instead of the device-dependent ones.

This works quite well except when the backend has to fallback because of a missing operation. In this case the color transformation is performed by a different color management system, which can produce slightly different results in most cases and very different results for device-dependent color spaces.

This problem has been reduced significantly by improving the drawing operations to avoiding fallback paths as much as possible.

5.4 OpenGL surfaces

Recent Cairo releases and development snapshots contain an experimental OpenGL-based hardware-accelerated backend, which performs compositing and source pattern interpolation using fragment shaders.

GL fragment shaders are small programs which perform a parallel computation on the GPU. Although the implementation of the generic ICC color conversion as a fragment shader is possible in theory, it is very unpractical, because it requires additional logic to implement n-components vectors and n-dimensional color lookup tables. In fact, the implementation of generic 3-components color spaces is still quite complicated and inefficient, even if OpenGL supports 3-dimensional lookup tables in the form of 3D textures.

The straightforward implementation of 3-components color transformation uses 1D textures to represent the curves from the ICC profile and a 3D texture for the LUT. This requires 10 texture accesses for the input color space to PCS transformations and 10 more texture accesses for the PCS to output color space one. A slightly more efficient implementation would merge the two conversions, but in general this would only save 3 texture lookups.

A much bigger improvement can be obtained by noticing that a very big family of ICC profiles is actually much simpler and only specifies three primaries and three tone reproduction curves. When connecting two such profiles, the color conversion process is defined as in figure 5.1. Moreover the two matrix multiplication operations can be combined in a single matrix matrix multiplication, further simplifying the fragment shader. These assumptions make it possible to implement a GL shader which performs the color transformation using only 6 texture lookups and one vector-matrix multiplication.

Although this color transformation is much more constrained than the generic transformation, it is also very common, because most display profiles and profiles for standard color spaces have exactly this structure. In particular, the sRGB color space and the standard YC_bC_r color spaces match this

$$\begin{tikzcd}
& f_r \arrow[dr] & & & & f_{r'}^{-1} \arrow[dr] & \\
\begin{bmatrix}R\\G\\B\end{bmatrix} \arrow[ur] \arrow[r] \arrow[dr] & f_g \arrow[r] & \boxed{M_{3\times 3}} \arrow[r, shift left=2] \arrow[r] \arrow[r, shift right=2] & \begin{bmatrix}X\\Y\\Z\end{bmatrix} \arrow[r, shift left=2] \arrow[r] \arrow[r, shift right=2] & \boxed{M'_{3\times 3}} \arrow[ur] \arrow[r] \arrow[dr] & f_{g'}^{-1} \arrow[r] & \begin{bmatrix}R'\\G'\\B'\end{bmatrix} \\
& f_b \arrow[ur] & & & & f_{b'}^{-1} \arrow[ur] &
\end{tikzcd}$$

Figure 5.1: The common $RGB \rightarrow XYZ \rightarrow RGB$ conversion

structure. This makes is possible to use these color spaces in GL without having to perform a costly fall back, which would require copying the image data between the video card and main memory.

5.5 Testing

The correctness of the Cairo implementation is checked using a test suite which performs some interesting sequences of graphic operations whose expected result is known. The actual result is then compared to a reference image and the outcome of the test is a failure if noticeable differences are found. The tests are run on every backend to ensure that the result only depends on the sequence of Cairo operations and not on the output backend. This is an effective way to check that all the backends have the desired behavior.

Tests are used to check if the library behaves as expected, which is especially important when adding new functions, and to keep track of bugs, making sure that there are no regressions. Because of this, the test suite is usually extended when new API functions are added and when a bug is found.

The public functions added by the proposed extension are tested by creating patterns of every possible type with multiple different interpolation color spaces. In addition to these tests, the ICC profile support test from `http://color.org/version4html.xalter` was implemented using Cairo graphic operations to draw the images from the original test.

Chapter 6

Results

6.1 Regressions

The code changes in Pixman introduce some regressions in the test suite because of different rounding between the integer the floating point compositing functions. The difference is usually not visible, but in some cases the test suite still reports these as failures. An example of such a regression can be seen in figure 6.1.

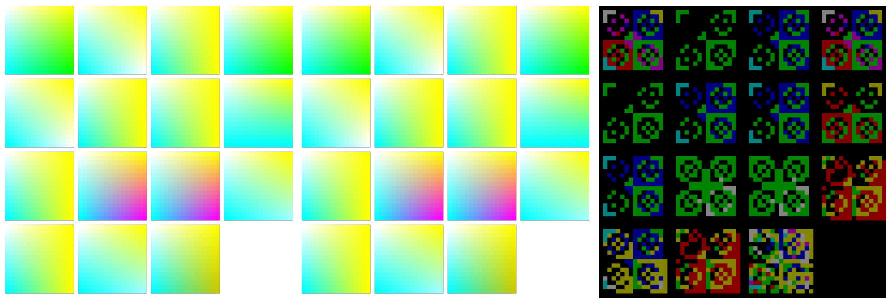

Figure 6.1: The regression in the extended-blend-mode test, caused by floating point computations. From left to right: reference image, result, difference (amplified).

This kind of regressions can be safely ignored, because the new results are likely more accurate than the old ones, in which case the differences actually indicate incorrect rounding in the integer implementation of the operation.

Another group of test failures is caused by bugs in the external tools used by the test suite to convert its output to raster images that can be compared. This is unfortunately frequent when new features are added or the output of the vector backends changes in nontrivial ways. An example of this kind of failure can be seen in figure 6.2.

Besides from these regressions, the extension is backward-compatible,

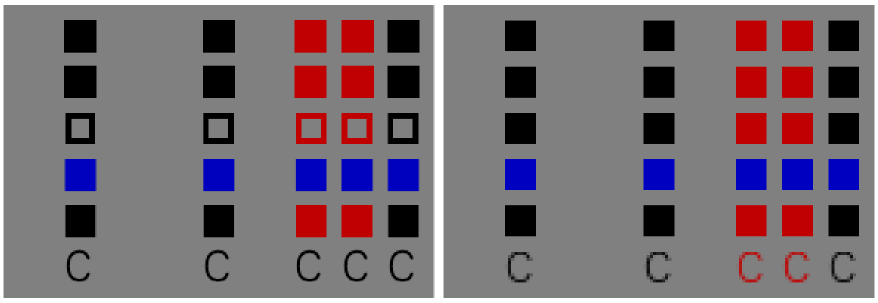

Figure 6.2: The regression in the clear-source test, caused by a bug in the external PDF rasterization tool. On the left, the PDF output; on the right, the PNG image produced by the rasterizer.

therefore, as expected, it causes no changes in the output of the existing tests.

6.2 New tests

The new tests check that the extended Cairo can handle color conversions as expected. Figure 6.3 shows the result of the test based on the ICC profile test. The left image shows that the 4 parts of the image are drawn ignoring their color profile, thus using the color components as if they were expressed in the device-dependent RGB color space. The right image shows the correct result obtained by applying the color profiles of those images.

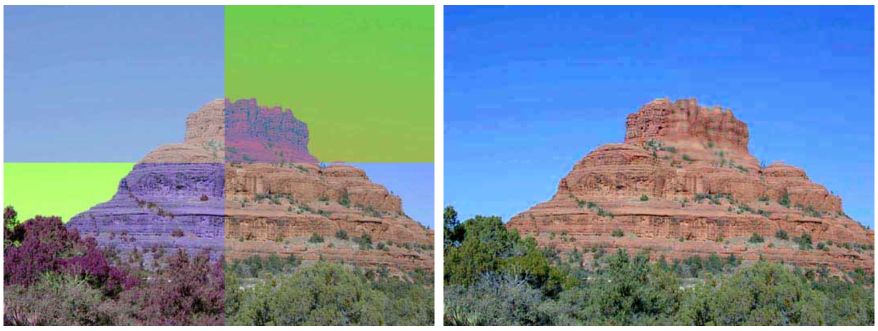

Figure 6.3: An example image showing images expressed in different color spaces. On the left, the output without color management; on the right, the output with color management.

The new tests also show that different backends handle device-dependent colors in different ways. A test which makes this clearly visible is provided in figure 6.4. The images in this figure contain 8 squares, each painted using primaries from a different color space. Each square is divided in 4 parts, each filled with a different primary color. The squares show (from top to bottom, from left to right):

- a device-dependent color space with no components, thus with no primaries
- the grayscale device-dependent color space (white)
- the RGB device-dependent color space (red, green, blue)
- the CMYK device-dependent color space (cyan, magenta, yellow and black)
- e-sRGB, an extension of the sRGB color space
- YCC, a luma-chroma color space (whose components are luminosity, blue chromaticity and red chromaticity)
- GBR, an sRGB-like color space whose primaries have been swapped (green, blue, red)
- Adobe RGB, the standard Adobe RGB color space (red, green, blue)

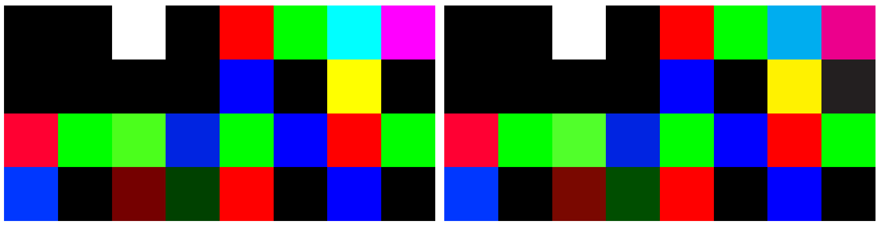

Figure 6.4: An example image showing colors from multiple color spaces. On the left, the output of the image backend; on the right, the output of the PDF backend.

It is very easy to spot the differences between the different "red" colors in different color spaces. Another visible difference can be noticed by comparing the CMYK primaries in the image backend and in the PDF backend.

6.3 Conclusions

The proposed extension makes it possible to add color management to the Cairo library without breaking existing applications which assume that the library operates in the device-dependent RGB color space.

This extension provides support for color spaces described using ICC profiles, which makes it possible to have a much more accurate control on the color handling. As special cases, it allows compositing to happen in linear

RGB or in sRGB, to perform gamma correction and to use device-dependent CMYK color spaces. The extension also provides the ability to perform compositing in YC_bC_r color spaces, which can result in a performance boost when working with video streams.

This extension has been designed to extend the whole drawing pipeline with color management in a way which is efficient for raster images and matches the behavior of common vector formats, thus avoiding expensive fallbacks and simplifying its implementation. In addition to this, it provides a generic color management workflow, which lets the application control the color transformations and operations performed in each step. This generality is important, because it allows the implementation of different drawing models on top of Cairo.

Even if the design of this extension is quite flexible and the implementation shows that it works, there are some issues, which should be solved to make this extension merge-ready.

The main issue that would currently prevent a merge of this extension is that Cairo and Pixman have changed significantly during the development of this extension. Cairo added support for a new pattern type and Pixman modified its internal structure. For the extension to be useful, the changes will need to be extended to also provide the new pattern type in arbitrary color spaces and to conform to the new Pixman compositing structure.

Another issue is performance, because most functions in Pixman have been modified to work on floating point data. Although this is reasonable on modern processors which can perform floating point operations, Pixman is also used on some platforms where floating point computations are performed in software, therefore if these architectures are to be supported by Pixman, the new code would need to be profiled on them. If the floating point code results in a major slowdown, a fixed-point implementation would probably be needed to provide acceptable performance on these architectures.

6.4 Further developments

The extension required extensive changes to the drawing pipeline, hence there are some enhancements that could should be implemented to take full advantage of the new features:

- As stated in section 4.2.3, it would be appropriate to also extend backend-specific API to support the creation of documents in any color space.

- The extension implements a floating point compositing pipeline, but does not export this functionality through an image format. Exposing floating point raster image buffers through the public API would make it possible to perform drawing operations and color transformations with greater accuracy. This would be especially important for operations that amplify the quantization error.

- In order to improve the performance when compositing video frames, support for YC_bC_r color spaces is not sufficient. Video frames are usually stored as planar downsampled images, but these image formats are currently unsupported in Cairo. In this case, avoiding the color conversion might not be sufficient to improve the performance, unless the conversion between different memory layouts is avoided as well.
- It would be possible to extend the color space support to include the special color spaces defined in the PDF specification.

New public API functions

```
/**
 * cairo_color_space_create_device_gray:
 *
 * Creates a new #cairo_color_space_t corresponding to a
 * device-dependent grayscale color space.
 *
 * Return value: a reference to the static device gray color space.
 * It doesn't require unreferencing, but it is safe to call
 * cairo_color_space_destroy() on it.
 *
 * This function will always return a valid pointer.
 **/
cairo_color_space_t *
cairo_color_space_create_device_gray (void);

/**
 * cairo_color_space_create_device_rgb:
 *
 * Creates a new #cairo_color_space_t corresponding to a
 * device-dependent RGB color space.
 *
 * Return value: a reference to the static device RGB color space.
 * It doesn't require unreferencing, but it is safe to call
 * cairo_color_space_destroy() on it.
 *
 * This function will always return a valid pointer.
 **/
cairo_color_space_t *
cairo_color_space_create_device_rgb (void);

/**
 * cairo_color_space_create_device_cmyk:
 *
 * Creates a new #cairo_color_space_t corresponding to a
 * device-dependent CMYK color space.
 *
 * Return value: a reference to the static device CMYK color space.
 * It doesn't require unreferencing, but it is safe to call
 * cairo_color_space_destroy() on it.
 *
 * This function will always return a valid pointer.
 **/
cairo_color_space_t *
cairo_color_space_create_device_cmyk (void);

/**
 * cairo_color_space_create_icc:
 * @profile: an ICC color profile
 * @profile_len: the size of @profile
 *
 * Creates a new #cairo_color_space_t corresponding to an ICC profile.
 *
 * Return value: the newly created #cairo_color_space_t if successful,
 * or an error color space in case of no memory.  The caller owns the
 * returned object and should call cairo_color_space_destroy() when
 * finished with it.
 *
 * This function will always return a valid pointer, but if an error
 * occurred the color space status will be set to an error.  To
 * inspect the status of a color space use cairo_pattern_status().
 **/
cairo_color_space_t *
cairo_color_space_create_icc (const void    *profile,
                              unsigned long  profile_len);

/**
 * cairo_color_space_reference:
 * @color_space: a #cairo_color_space_t
 *
 * Increases the reference count on @color_space by one. This prevents
 * @color_space from being destroyed until a matching call to
 * cairo_color_space_destroy() is made.
 *
 * The number of references to a #cairo_color_space_t can be get using
 * cairo_color_space_get_reference_count().
 *
 * Return value: the referenced #cairo_color_space_t.
 **/
cairo_color_space_t *
cairo_color_space_reference (cairo_color_space_t *color_space);

/**
 * cairo_color_space_destroy:
 * @color_space: a #cairo_color_space_t
 *
 * Decreases the reference count on @color_space by one. If the result
 * is zero, then @color_space and all associated resources are freed.
 * See cairo_color_space_reference().
 **/
void
cairo_color_space_destroy (cairo_color_space_t *color_space);

/**
 * cairo_color_space_get_reference_count:
 * @color_space: a #cairo_color_space_t
 *
 * Returns the current reference count of @color_space.
 *
 * Return value: the current reference count of @color_space.  If
 * @color_space is a nil object or a static color space, 0 will be
 * returned.
 **/
unsigned int
cairo_color_space_get_reference_count (cairo_color_space_t *color_space);

/**
 * cairo_color_space_status:
 * @color_space: a #cairo_color_space_t
 *
 * Checks whether an error has previously occurred for this color
 * space.
 *
 * Return value: %CAIRO_STATUS_SUCCESS, %CAIRO_STATUS_NO_MEMORY.
 **/
cairo_status_t
cairo_color_space_status (cairo_color_space_t *color_space);

/**
 * cairo_color_space_get_number_of_components:
 * @color_space: a #cairo_color_space_t
 *
 * Returns the number of components of @color_space.
 *
 * Return value: the number of components of @color_space.  If
 * @color_space is a nil object, 0 will be returned.
 **/
unsigned int
cairo_color_space_get_number_of_components (
    cairo_color_space_t *color_space);

/**
 * cairo_pattern_create_color:
 *
 * @color_space: the color space in which the @components define the
 *               desired solid color
 * @components: the components of the color in the given color space
 * @alpha: the alpha component of the color
 *
 * Creates a new #cairo_pattern_t corresponding to a translucent color.
 *
 * Return value: the newly created #cairo_pattern_t if successful, or
 * an error pattern in case of no memory.  The caller owns the
 * returned object and should call cairo_pattern_destroy() when
 * finished with it.
 *
 * This function will always return a valid pointer, but if an error
 * occurred the pattern status will be set to an error.  To inspect
 * the status of a pattern use cairo_pattern_status().
 **/
cairo_pattern_t *
cairo_pattern_create_color (cairo_color_space_t *color_space,
                            double              *components,
                            double               alpha);

/**
 * cairo_pattern_create_linear_with_color_space:
 * @color_space: the color space in which the color stops will be
 *               specified and interpolated
 * @x0: x coordinate of the start point
 * @y0: y coordinate of the start point
 * @x1: x coordinate of the end point
 * @y1: y coordinate of the end point
 *
 * Create a new linear gradient #cairo_pattern_t along the line
 * defined by (x0, y0) and (x1, y1). Before using the gradient
 * pattern, a number of color stops should be defined using
 * cairo_pattern_add_color_stop_color().
 *
 * Note: The coordinates here are in pattern space. For a new pattern,
 * pattern space is identical to user space, but the relationship
 * between the spaces can be changed with cairo_pattern_set_matrix().
 *
 * Return value: the newly created #cairo_pattern_t if successful, or
 * an error pattern in case of no memory.  The caller owns the
 * returned object and should call cairo_pattern_destroy() when
 * finished with it.
 *
 * This function will always return a valid pointer, but if an error
 * occurred the pattern status will be set to an error.  To inspect
 * the status of a pattern use cairo_pattern_status().
 **/
cairo_pattern_t *
cairo_pattern_create_linear_with_color_space (
    cairo_color_space_t  *color_space,
    double                x0,
    double                y0,
    double                x1,
    double                y1);

/**
 * cairo_pattern_create_radial_with_color_space:
 * @color_space: the color space in which the color stops will be
 *               specified and interpolated
 * @cx0: x coordinate for the center of the start circle
 * @cy0: y coordinate for the center of the start circle
 * @radius0: radius of the start circle
 * @cx1: x coordinate for the center of the end circle
 * @cy1: y coordinate for the center of the end circle
 * @radius1: radius of the end circle
 *
 * Creates a new radial gradient #cairo_pattern_t between the two
 * circles defined by (cx0, cy0, radius0) and (cx1, cy1, radius1).
 * Before using the gradient pattern, a number of color stops should
 * be defined using cairo_pattern_add_color_stop_color().
 *
 * Note: The coordinates here are in pattern space. For a new pattern,
 * pattern space is identical to user space, but the relationship
 * between the spaces can be changed with cairo_pattern_set_matrix().
 *
 * Return value: the newly created #cairo_pattern_t if successful, or
 * an error pattern in case of no memory.  The caller owns the
 * returned object and should call cairo_pattern_destroy() when
 * finished with it.
 *
 * This function will always return a valid pointer, but if an error
 * occurred the pattern status will be set to an error.  To inspect
 * the status of a pattern use cairo_pattern_status().
 **/
cairo_pattern_t *
cairo_pattern_create_radial_with_color_space (
    cairo_color_space_t *color_space,
    double cx0, double cy0, double radius0,
    double cx1, double cy1, double radius1);

/**
 * cairo_pattern_get_color_space
 * @pattern: a #cairo_pattern_t
 *
 * Returns the color space of @pattern.
 *
 * Return value: a #cairo_color_space_t. The caller owns the returned
 * object and should call cairo_color_space_destroy() when finished
 * with it.
 **/
```

```
cairo_color_space_t *
cairo_pattern_get_color_space (cairo_pattern_t *pattern);

/**
 * cairo_pattern_add_color_stop_rgba:
 * @pattern: a #cairo_pattern_t
 * @offset: an offset in the range [0.0 .. 1.0]
 * @components: the color components of the stop to be added
 * @alpha: the alpha component of the stop to be added
 *
 * Adds a translucent color stop to a gradient pattern. The offset
 * specifies the location along the gradient's control vector. For
 * example, a linear gradient's control vector is from (x0,y0) to
 * (x1,y1) while a radial gradient's control vector is from any point
 * on the start circle to the corresponding point on the end circle.
 *
 * The color is specified in the same way as in
 * cairo_pattern_create_color().
 *
 * If two (or more) stops are specified with identical offset values,
 * they will be sorted according to the order in which the stops are
 * added, (stops added earlier will compare less than stops added
 * later). This can be useful for reliably making sharp color
 * transitions instead of the typical blend.
 *
 * Note: If the pattern is not a gradient pattern, (eg. a linear or
 * radial pattern), then the pattern will be put into an error status
 * with a status of %CAIRO_STATUS_PATTERN_TYPE_MISMATCH.
 **/
void
cairo_pattern_add_color_stop_color (cairo_pattern_t *pattern,
                                    double           offset,
                                    const double    *components,
                                    double           alpha);

/**
 * cairo_surface_create_similar_with_color_space:
 * @other: an existing surface used to select the backend of the new
 *         surface
 * @color_space: the color space for the new surface
 * @content: the content for the new surface
 * @width: width of the new surface, (in device-space units)
 * @height: height of the new surface (in device-space units)
 *
 * Create a new surface that is as compatible as possible with an
 * existing surface. For example the new surface will have the same
 * fallback resolution and font options as @other. Generally, the new
 * surface will also use the same backend as @other, unless that is
 * not possible for some reason. The type of the returned surface may
 * be examined with cairo_surface_get_type().
 *
 * Initially the surface contents are all 0 (transparent if contents
 * have transparency).
 *
 * Return value: a pointer to the newly allocated surface. The caller
 * owns the surface and should call cairo_surface_destroy() when done
 * with it.
 *
 * This function always returns a valid pointer, but it will return a
 * pointer to a "nil" surface if @other is already in an error state
 * or any other error occurs.
 **/
```

```
cairo_surface_t *
cairo_surface_create_similar_with_color_space (cairo_surface_t      *other,
                                               cairo_color_space_t *color_space,
                                               cairo_content_t  content,
                                               int              width,
                                               int              height);

/**
 * cairo_push_group_with_color_space:
 * @cr: a cairo context
 * @color_space: the color space of the group to be created
 * @content: a #cairo_content_t indicating the type of group that
 *           will be created
 *
 * Temporarily redirects drawing to an intermediate surface known as a
 * group. The redirection lasts until the group is completed by a call
 * to cairo_pop_group() or cairo_pop_group_to_source(). These calls
 * provide the result of any drawing to the group as a pattern,
 * (either as an explicit object, or set as the source pattern).
 *
 * The group will have a content type of @content and a color space of
 * @color_space.
 */
void
cairo_push_group_with_color_space (cairo_t              *cr,
                                   cairo_color_space_t *color_space,
                                   cairo_content_t      content);

/**
 * cairo_set_rendering_intent:
 * @cr: a #cairo_t
 * @intent: the rendering intent to be used for color transformations
 *
 * Sets the rendering intent to be used for all drawing
 * operations. See #cairo_rendering_intent_t for details on the
 * semantics of each available rendering intent.
 **/
void
cairo_set_rendering_intent (cairo_t *cr, cairo_rendering_intent_t intent);

/**
 * cairo_get_rendering_intent:
 * @cr: a #cairo_t
 *
 * Returns the rendering intent of a cairo context.
 *
 * Return value: the current rendering intent of @cr.
 **/
cairo_rendering_intent_t
cairo_get_rendering_intent (cairo_t *cr);
```

Bibliography

[1] Eric Brasseur. Gamma error in picture scaling. http://www.4p8.com/eric.brasseur/gamma.html.

[2] European Broadcasting Union Technical Centre. Standard for chromaticity tolerances for studio monitors. EBU Tech. 3213-E, August 1975.

[3] International Color Consortium. *Image technology colour management – Architecture, profile format, and data structure.* ICC, May 2006.

[4] World Wide Web Consortium. SVG color 1.2, part 2: Language. WD-SVGColor12-20091001, http://www.w3.org/TR/2009/WD-SVGColor12-20091001/, October 2009.

[5] Commission Internationale d'Eclairage. *Colorimetry.* CIE, Central Bureau of the CIE, Vienna, second edition, 1986.

[6] David Duce. Portable network graphics (png) specification. http://www.w3.org/TR/PNG, November 2003.

[7] Jon Ferraiolo, Jun Fujisawa, and Dean Jackson. Scalable vector graphics (SVG) 1.1 specification. World Wide Web Consortium, Recommendation REC-SVG11-20030114, http://www.w3.org/TR/2003/REC-SVG11-20030114, Jabuary 2003.

[8] International Organization for Standardization. Document management – portable document format – part 1: Pdf 1.7. ISO 32000-1:2008, July 2008.

[9] Gernot Hoffmann. Ps-tutor function graphs and other applications for postscript. http://www.fho-emden.de/~hoffmann/pstutor22112002.pdf.

[10] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison-Wesley Publishing Company, Reading, Massachusetts, USA, 3rd edition, February 1999.

[11] Microsoft. Windows Color System. http://www.microsoft.com/color.

[12] Hubert Nguyen, editor. *GPU Gems 3*. Addison-Wesley, 2008.

[13] Society of Motion Picture and Television Engineers. Composite analog video signal – ntsc for studio applications. SMPTE 170M-1999, 1999.

[14] Keith Packard. The X Rendering Extension. http://cgit.freedesktop.org/xorg/proto/renderproto/plain/renderproto.txt.

[15] Thomas Porter and Tom Duff. Compositing digital images. In Hank Christiansen, editor, *Computer Graphics (SIGGRAPH '84 Proceedings)*, volume 18, pages 253–259, July 1984.

[16] Michael Stokes, Matthew Anderson, Srinivasan Chandrasekar, and Ricardo Motta. A standard default color space for the internet — sRGB. http://www.w3.org/Graphics/Color/sRGB, November 1996.

[17] International Telecommunication Union. Parameter values for the hdtv standards for production and international programme exchange. ITU-R BT.709-5, 2002.