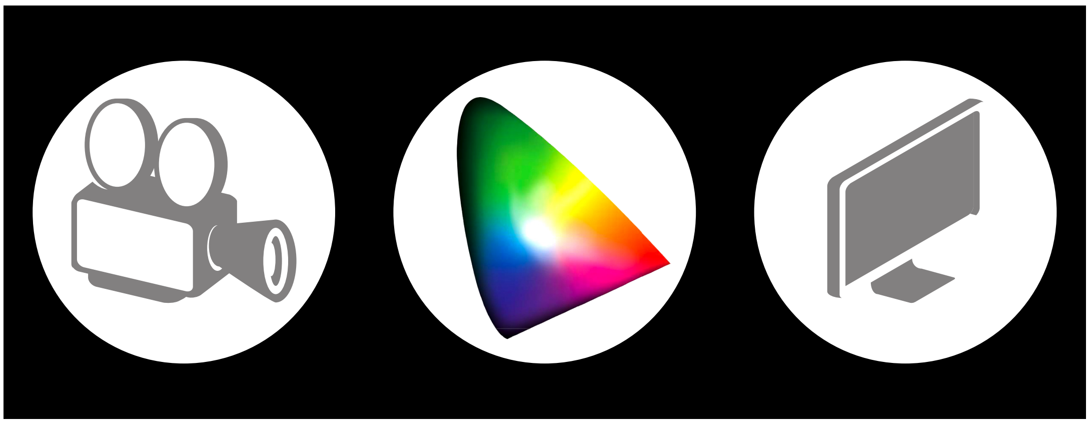
A COLOR MANAGEMENT MODULE FOR KEYFRAME

Bachelor thesis by Thomas Simon
Spring 2011
Høgskolen i Gjøvik
Hochschule der Medien, Stuttgart



Abstract

The goal of color management in movie color grading is to enable constant color reproduction throughout a film workflow. The main goal of my work was the implemantation of a color management module for the color grading application Keyframe RushesControl. Therefore, I created a connection from Keyframe's already existing functions to the color transformation functions of littleCMS in order to enable color management both based on the ICC workflow and on Nucoda lookup tables.

In this thesis, I will at first present the principles of color science and colorimetry including the basic structure of a color management workflow.

Secondly, I will present the techniques and tools I used to program the color management module with C++. Also, I will discuss functions and methods of littleCMM that my color management module mainly was based on.

Finally, I will present an experiment I made after the implementation to evaluate the performance of my CMM. This includes a discussion about the problems involved in designing a suitable experiment for our setup.

Contents

Introduction

In the fall of 2010, I met with Arne Magnus Bakke, John Christian Rosenlund and Andreas Herzog from Drylab for the first time to talk about a topic for my bachelor thesis that I was soon to begin in the beginning of 2011. I heard about the color grading software called Keyframe RushesControl they were developing and I was very eager to help working on it. For me, it was a big chance to actually program and develop a image processing application for the first time during my studies.

We discussed several possible tasks related to Keyframe RushesControl, for example the integration of ArriRAW formats or the implementation of a color management module for the software. On the one hand side, I was most interessted in color management for my future studies, and on the other hand side, color management was a very important topic for Andreas Herzog and John Christian Rosenlund as well because as film photographers they were confronted with the color reproduction problem on a daily basis. Thus, we decided to tackle on the color management module for the application.

From this point on, my task was to program and integrate a consistent color workflow in Keyframe. At first, I wanted to find about the basics and the general pupose of color management. I wanted to investigate a theoretical approach in order to gain a profound insight in color management and color science in general. That is what the part A is all about.

Secondly, I programmed the actual feature for Keyframe RushesControl with C++. I wanted to learn more about this particular programming language and about the source code that already existed. Then, the main task was to “translate” the theoretical knowledge into a user-friendly color management module for the application. That is what the part B is all about.

Finally, I wanted to find out how well my color management module performed. Therefore, I evaluated a series of color patches with the help of the basics of color management investigated for my first part of this thesis. Later on, I set up several experiments and discussed the results gained from the experiments. That is what the part C is all about.

Last but not least, I want to thank my supervising professors Jon Yngve Hardeberg og Ronald Schaul for their feedback and help during the writing and evaluation process. Also, I want to thank a Arne Magnus Bakke and Tariq Islam who were a very big help during the programming of the module. Especially without the help of Arne Magnus Bakke, the thesis would have been a lot more difficult to accomplish. Finally, I want to thank Aditya Sole who helped me with the evaluation and interpretation of the experiment results.

Simon

Thomas Simon in Gjøvik on April, 27th 2011.

PART A

What is Color Management?

The Color Reproduction Problem

Color Management is a tool to obtain constant color reproduction throughout a workflow with different media devices. The main goal of color management is to solve the so-called color reproduction problem. The color reproduction problem is the phenomenon that any given input color data is reproduced differently on different media devices. In other words, the colors of a real-life scene look different on different devices throughout an image reproduction workflow.

Two aspects of the Color Reproduction Problem

This problem is probably well known to everyone who ever used a camera and looked at this picture on a computer screen, posted it on the Internet or ordered a print copy. Imagine you take a picture; you see an impressing sunset and you take out your camera. You capture the scene.

FIGURE 2-1 Sunset in Gjøvik

Then, you look at the camera screen and you will find that the recorded picture is somewhat less expressive: The colors are less intensive and the picture might be either too dark or too light. You notice maybe an overall loss of quality compared to the original scene. You will come to a similar conclusion when you look at the picture on a computer screen later. Moreover, you will notice that the colors on the computer screen look again different both compared to the original scene and the camera display. In the case of printing the picture, you get a third, very different representation of the original scene.
At this point, we make a note that the picture of one original scene that is reproduced on three different media types result in three very different color appearances, although the original color information stays constant for all three media.

And it gets even more complex. Imagine we looking at the same picture on several different screens. We notice that the picture looks different on a laptop screen as compared to a desktop LCD screen, and even more different compared to a CRT screen. The same phenomenon can be observed for multiple printers. When we print out our picture on three different printers, we will most likely get three different color results.
Thus, we make another note that we get three very different color appearances for multiple individual devices even if these devices belong to the same group of media devices.

To sum up, the color reproduction problem can be divided into two aspects. The displayed color appearance changes, if:

1. we use **devices of different kinds** of media devices.
2. we use **different individual devices** of the same kind of media devices.

Causes for the CRD

The reasons for the color reproduction problem are numerous and I am going to explain them in detail later. The most important ones shall be named here shortly: Compared to the original scene, no capturing or displaying device is capable of reproducing the full range of colors that we can perceive with the human eye. These are the technical boundaries of our cameras, screens or printers. In terms of different results when comparing different kinds of media – like printers, scanners or cameras – we can say that all these different kinds of media devices are based on totally different principles to reproduce or record colors[1]. Moreover, the surrounding have a big influence on how color is perceived. A picture looks very different depending on whether the light in a room is turned off or on, for example.

1 Screens for example are "additive" output devices: They "produce color on a dark background through the combination of differently colored lights, known as primaries" (Sharma, 65). That means the primaries red, green and blue are mixed together to produce colors. All primaries mixed together evenly result in white. Printers in contrast are "subtractive" devices: The color for these devices "is produced through a process of removing (subtracting) unwanted spectral components from "white" light" most typically "on transparent or reflective media" (Sharma, 65f). The primary colors are most often cyan, magenta, yellow and black. All primaries mixed together evenly result in black. These fundamental structural differences in how color is achieved leads to fundamentally different ranges of color that both devices can produce when compared to each other.

Esthetic and Economical Dimensions of the CRD

Why is a different color appearance even a problem? Imagine you invested a lot of time and/or money in capturing and photoshoping a picture. You set the composition carefully, you retouched the surfaces and you made the colors to look just the way you intended them to be. Then, you post the pictures on your website and the colors look completely different. What is worse, you order a print copy online and the result does not look like what you expected it to be at all. It is frustrating for you personally but it is damaging for people who work with pictures professionally like photographers, printers, designers etc[2]. They depend on that the results of their work keep the same appearance throughout a long workflow involving various, unpredictable stations. In another example, the German telecommunication company "Deutsche Telekom" registered the magenta color of its logo as legal trademark and sued many competitors for using "their" trademarked color (c. Wikipedia: Magenta (Farbe)).

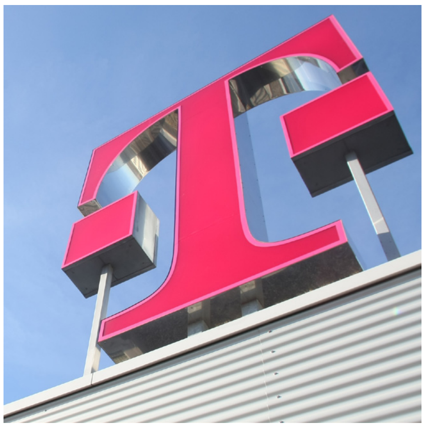


FIGURE 2-2 Telekom logo on the company's headquarter in Bonn.

The company naturally wants to have a constant representation of this magenta tone throughout the entire workflow for every picture that leaves there public relations office, a commercial ad for instance.

Keeping the right color representation is very important, both professionally and esthetically. But can we actually solve this problem? What components does the perception of color depend on? Color and our whole perception is strongly subjective. Maybe, it is just in our eyes that the colors look different. After all, the color information of our picture does stay constant throughout the workflow. Nevertheless, we **do** perceive different colors. But is it possible to measure the perceived differences empirically?

2 When I was working for a photographer in 2004, for example, I remember one time that a customer complained about "wrong" colors of a picture we made. Eventually, it turned out that this was due to wrong color management on the customer's side but we had to give him rebate on his final price anyway.

The Color Models

It is possible, to begin with! Color models are models that help us describe and categorize colors empirically. These color models are going to be the second big topic we are going to take a look at now.

What is Color?

Let us first look at what colors actually are. The International Commission of Illumination (CIE) defines color as following:

> Color is an "attribute of visual perception consisting of any combination of chromatic and achromatic content" (IEV no. 845-02-18).

That means color is an attribute of an object or a surface, that we perceive with our eyes. We describe this attribute in words by "chromatic color names such as yellow, orange, brown, red, pink, green, blue, purple, etc., or by achromatic color names such as white, grey, black, etc.," and qualify it with words like "bright, dim, light, dark, etc." (IEV no. 845-02-18).

The medium that "transports" colors is light. But the colors we perceive depend on more than that. The perceived color depends "on the spectral distribution of the color stimulus", "stimulus area" and "observer's visual system" as long with "the observer's experience" (IEV no. 845-02-18). As we can see, two more equally important components are necessary for the perception of colors so that we have a total of three components that are involved in the perception:

1. The light source
2. The object or surface of the object
3. The observer[3]

Only in combination, these three components create that what we experience as color.

To get hold of the color reproduction problem, we first need to find a way to describe color objectively. As we can guess from the definitions above, this is a somewhat complex task. On some components, especially those involving the observer, we do only have little if any influence. Others, especially those involving the visible radiation, we can influence somewhat easily with the help of technical tools.
To get a consistent color representation, we need to keep the resulting light stimulus constant and we need to assure a similar environment setting in which the observer looks at the pictures.

First of all, we need to describe the reaction of the observer to the light stimulus objectively/empirically. After all, the color sensation is triggered by light stimuli. And light is an electromagnetic wave that can be described empirically[4]. In the past, there have been multiple research studies focusing on how colors can be described, categorized and compared with each other. These studies resulted in the so-called *color models.*

3 More specifically the observer's visual system as described below.

4 In literature, light or "visible radiation" (IEV no. 731-01-04) is defined by the CIE as "any optical radiation capable of causing a visual sensation directly on a human being" (IEV no. 731-01-04). The part

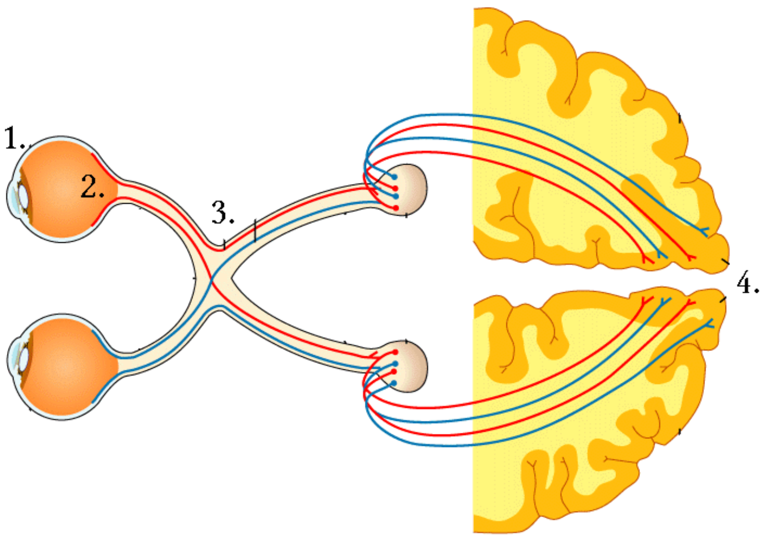


Figure 2-3 The human optical system

The Visual System

The main purpose of color models is that of describing and categorizing colors in an empirical way. Color models are based on the human visual system, that means they describe objectively what happens inside of the human observer's head. Therefore, we need to take one step back again and look at how our visual system processes and creates color impressions.

The visual system is "the part of the central nervous system [...] that enables to process visual detail" (wikipedia/visual system). It consists of four main parts (c. Heavens, 208):

1. The optical system of the eye
2. The retina of the eye
3. "The optic nerve which transmits the information to the visual cortex."
4. The virtual cortex, "a central processor" that filters and reorganizes the color information coming from the eye.

These parts can roughly be grouped into two main parts:
A "front end" that includes everything that happens within the eye, and a "back end" that includes everything that happens inside the brain or on the way to the brain. Everything related to human vision including the perception of color requires a balanced interaction between both eyes and brain – with both parts being equally important.

What happens to a beam of light that enters the human visual system can be described as following: The cornea and the lens (1.) focus the beam of light on the retina (2.) with two different types of photo receptors. Receptors are responsible for translating light into electrical signals that are then handed to the optic nerve (3.). Finally, the information is processed and interpreted

of the electromagnetic spectrum that can be perceived by the human visual system lies typically in the region between λ = 360 nm (min) and λ = 830 nm (max) (Sharma, 16).

in the brain, more precisely in the visual cortex (4) (c. Heavens, 208).
Another important thing is to understand that color is not an attribute of the light itself. The perception of color requires both electromagnetic radiation and an observer: There is no color without an observer[5]*.*

The Rods and the Cones

What happens on the retina is most important for us at this point. We distinguish between two types of photoreceptors: cones and rods.

Rods are very sensitive to light and are active even when the luminance level is very low[6]. Rods do not distinguish between different wavelengths, they can merely sense the intensity, they are responsible for our perception of dark and bright.

Cones in contrast are those that enable color vision. There are three different types, each is sensitive for a different part of the visible spectrum: S (for short wavelength sensitive cones, sensitive for what is perceived as “bluish” colors), M (medium wavelength sensitive cones, perceived as “greenish” colors) and L (long wavelength sensitive cones, perceived as “redish” colors) (c. Sharma, 20)[7].

Trichromatic Theory

The empirically evidence for the existence of the different cones at the beginning of the 20th century was an important affirmation for a theory that has been developed by Thomas Young and Herman von Helmholtz earlier in the 19th century called trichromatic theory aka Young-Helmholtz theory (c. Willumsen, 18). Helmholtz pointed out that for any given color impression, there is not just one type of photoreceptor on the retina that is being stimulated but three different types of them[8]. What is more, the trichromatic theory states that every perceivable color can be produced “by using only combinations of light from three light sources” (Sharma, 22) with the primaries being typically red, green and blue[9].

5 This fact is called the “stimulus error” and was described by John Isaac Newton as following: “The rays […] are not colored; in them there is nothing else than certain power and disposition to stir up a sensation of this or that color” (through Sharma, 18).

6 Such situations are called scotopic vision (see CIE’s definition for scotopic vision under IEV no. 845-02-10).

7 Cones are not very sensible to light and are only active when the luminance level is very high (c. Sharma, 163). This situation is called photopic vision (see also CIE’s definition for photopic vision under IEV no. 845-02-09).

8 Moreover there is always one cone that reacts strongest and the two other that react less intense (c. Willumsen, 18).

9 In the 19th century, Ewald Hering described another theory explaining the sensation of colors called the opponent color theory, aka Hering theory (c. Willumsen, 19). This theory states that the human visual system is based on two axes consisting of two opposites pair: a red-green axis and a blue-yellow axis (c. Sharma, 55). The two theories were looked upon as competing, exclusive ones, but newer researches stated that both explanations apply for the human visual system: Firstly, the color stimuli are processed trichromatic on the retina. Then, they are processed “opponently” on their way to the visual cortex (3.)(c. Sharma, 55).

Color Matching Experiments

The trichromatic theory was the starting point of several experiments organized and implemented by John Guild and W. David Wright in the late 1920s – the so-called color matching experiments (c. Green&al., 22). In a color-matching experiment, the observer looks on a sample field that is divided into two areas: The one half of the sample field is illuminated by the original color stimulus, whereas the other is illuminated by the light of three primary wavelength (being 460 nm, 530 nm and 650 nm) (c. Green&al., 22 and Sharma, 28). The observer is able to adjust the intensity of the three primary wavelengths until he/she perceives a match between the original stimulus and the combined wavelengths. In this way, Guild and Wright where able to produce curves that reflect the "amounts of primary wavelengths [...] required to match the spectral colors" (Green&al., 22). This means, they could reproduce any color within the visible spectrum using only the three primary wavelengths. Eventually, they calculated mathematical functions from these results.

The knowledge of these functions was eventually put on a theoretical foundation and adopted by the CIE in 1931 (c. Green&al., 23). The CIE called the (virtual) primaries X, Y and Z and the resulting color matching functions are since then referred to as the CIE 1931 Color Matching Functions (CMF) aka X(λ), Y(λ) and Z(λ) (c. Green&al., 23).
That was the birth of the so-called "color appearance models" (Sharma, 172) that provided an opportunity to describe any given light stimulus empirically with the help of these color matching functions.

First of all, colors could now be described objectively using the three CIE 1931 CMF. Colors could be measured empirically with the help of a colorimeter or spectrophotometer and then translated into the standardized CIE 1931 XYZ color space.
This is very important for our own solution to the color reproduction problem as well because now we can detect and describe differences between the color stimuli coming from different devices objectively. Fortunately, this is exactly what we are going to do in the evaluation part later.

Color Spaces

I know, the color matching functions sound rather abstract and rather hard to relate to. That is where a visual representation, a so-called color space, comes in handy. The CIE defines the color spaces as following:

> A color space is a "geometric representation of colors in space, usually of three dimensions" (IEV no. 845-03-25).

The XYZ color space for example represents a visualization of all visible colors as described with the CIE 1931 Color Matching Functions. Every visible color has an allocated coordination within the XYZ color space that identifies the color uniquely (see Figure 2-4).

Of course, 3-D visualizations were very difficult in the 1930s, and people thought of a way to present a 3-D graph in two-dimensional media like books and papers. Scientists decided to display only a cross section of the two-dimensional graph and they agreed on the part of the graph where all three coordinates sum up to one, the so-called unit plane (c. Sharma, 35).

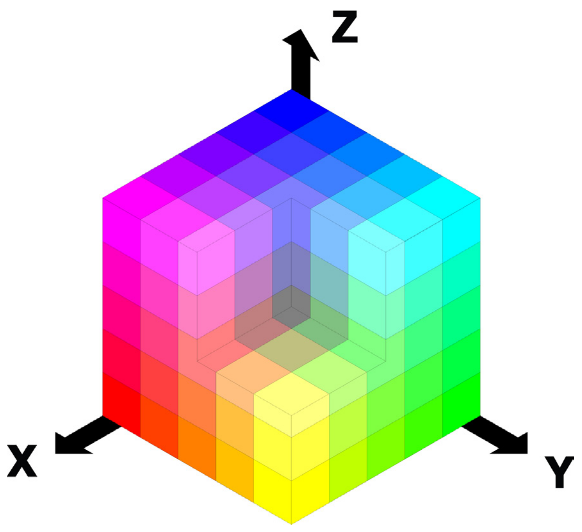


Figure 2-4 3-D representation of the XYZ color space

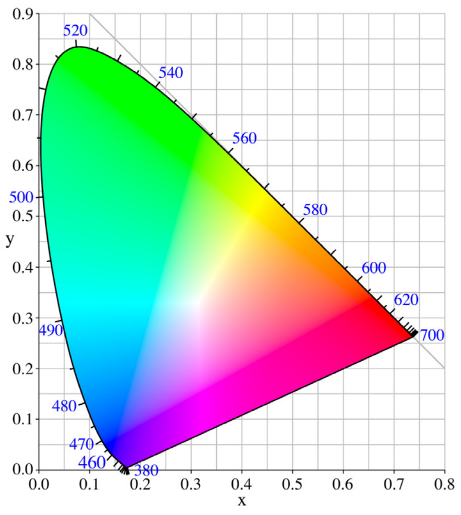


Figure 2-5 2-D representation/chromaticity diagram of the XYZ color space

The resulting graph is called the chromaticity diagram[10] (see Figure 2-5) and it shows the hue of a given color stimulus – in the same time as it discards any information about its brightness or intensity (c. Sharma, 35).

Moreover, a color space like CIE 1931 XYZ allows us to calculate the difference between to color stimuli with the help of a so-called color difference formula. The most simple difference formula is the euclidean distance between two color coordinates in the three-dimensional color space (c. Sharma, 42). *With a color difference formula, we now have the possibility to compare two colors within the model. The formula gives us an indicator how close respectively how far two colors are afar from each other.*

Uniformity of Color Spaces

However, the CIE 1931 XYZ color space and its Euclidean difference formula was unpractical for some reasons. It was criticized that the perceived color difference of color samples does not result in the same empirical color difference (c. Sharma, 41). The color space is close enough to describe colors objectively but it does not relate very much to our perceptual experience of color. In one experiment, scientists showed observers different color samples and the observers were asked, whether they could see a just noticeable difference between the samples or not. Then, the scientists would compared the CIE 1931 XYZ values for the samples that were perceived as being identical (c. Sharma, 42). The result was that the relations between the objective measured difference and the subjective perceived difference vary strongly throughout the spectrum: The CIE 1931 XYZ is not uniform (c. Sharma, 41). This means that identical empirical color differences of two color stimuli do not result in constant perceived differences.

10 aka shoe sole or horse shoe unofficially

CIE 1979 L*a*b* and its ΔE formula

This is one of several reasons why scientists developed several other color models which provided, among other things, a more adequate color difference formula: I want to name the CIE 1979 L*u*v* and the CIE 1979 L*a*b*, because they are the most common ones. They both have the one thing in common that their coordinates L, u and v respectively L, a and b are derived from the original CIE 1931 CMF in a way that each of the three axes represents one specific attribute of color (c. Sharma, 42ff and Green, 53f)[11]. Why is it important to talk about these color spaces? Because CIELAB provides a color difference formula that will be very useful later in the evaluation part of this thesis (Part C) to calculate the difference between two color patches on the main and the extra screen. The color difference formula is basically the Euclidean distance between two color coordinates within the three-dimensional CIELAB color space and it looks like this (c. Sharma, 44):

$$\Delta E_{ab} = \sqrt{(\Delta L)^2 + (\Delta a)^2 + (\Delta b)^2}$$

Gamuts

After this theoretical excursion, we will now go back to our original problem, the color reproduction problem. As I mentioned before, no display is capable of producing every existing color. Screens for example can only display a certain range of colors that is reproducible with the three primary colors. Can this range of colors be visualized and if yes, how does it look like compared to the original CIE 1931 XYZ color space?

To begin with, the answer to the first question is of course yes. This range is called color gamut and it is defined as the "range of colors achievable on a given color reproduction medium [...] under a given set of viewing conditions" (Green&al., 300). The gamut can be described as a subset of the original XYZ color space containing only those colors that can be reproduced with the help of the device's primary colors. Like the original color space, the gamut can be visualized as a color space both three-dimensionally and two-dimensionally. For the two-dimensional representation, one draws typically the boundaries of the gamut in the XYZ chromaticity diagram. As we can see on the picture below (see Figure 2-6), all reproducible colors of a typical screen lay within a triangle, whose corners a marked by the devices primary colorants.

Gamuts are very individual to every single device. We can truly say that there are as many different gamuts as there are devices. As we will see shortly, one key to solving the color reproduction problem is to "translate" between different device gamuts. To get this variety of gamuts organized, scientists and companies created so-called generic color spaces, which can be defined as gamuts that are "based on virtual output devices" (Green&al., 396). In other words, these generic color spaces represent the gamut of a virtual screen, printer or any other color reproduction device that reacts in an optimal way. The most famous generic color spaces are probably sRGB and Adobe RGB (more about them later). The specifications of such generic color spaces define a color space with a set of virtual primaries that define the colors they can produce.

11 For the CIELAB color space, for example, the L* axis represents the luminance level, whereas a* represents the red-green axis respectively b* represents the yellow-blue axis (c. Sharma, 45).

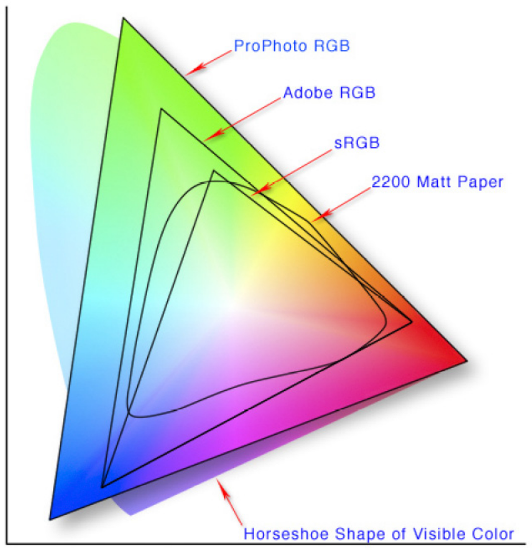


Figure 2-6 The chromaticity gamuts of sRGB, Adobe RGB and a CMYK color space

To sum up, we put on the record that with the CIE 1931 XYZ respectively the CIE 1976 L*a*b* color spaces, we have two color appearance models that are very helpful to describe the color reproduction problem empirically. On the one hand side they allow us to describe color stimuli objectively with the help of the color matching functions. And on the other hand side, they give us a tool to measure and evaluate the difference between two given color stimuli. We will now take a step forward to tackling our color reproduction problem and – with the help of the tools we learned about in this chapter – we will describe a solution to the problem.

The Definition of Color Management

Color management can be defined as tool to “provide predictable and consistent color results” (Green&al., 247). As we discussed before, there are two aspects of the color reproduction problem. For both aspects, color management has to provide a solution.

Firstly, there is the problem of translating color information between different types of media devices. Imagine we import a picture from a camera into the computer for displaying it on the screen, or we print a picture after having it edited in Photoshop. The first thing we have to be aware of is this: Most of the time, when we transfer color data between two different kind of media, the original data gets altered in some way. The camera contains a CCD chip that transforms photons into electrons, the screen uses three primaries red, green and blue to combine colors additively and the printer mixes four inks cyan, magenta, yellow and black subtractively. Three different devices, that means three different ways of producing color. With what we have learned in the previous paragraphs, we can further say that all three devices are based on three different color spaces. That implies that they can only produce a certain range of colors, a certain gamut. Let us assume the camera is typically based on an Adobe RGB color space, whereas the screen uses a sRGB based color space and the printer a CMYK based color space.

Imagine now, we go from capturing to displaying on screen to printing. We can not just keep the data as it is, because we would necessarily be confronted with values from one color space that could not be interpreted in another color space. What we have to do is this: We have to translate the color information between the color spaces. And what is more, we want to translate it in a way that we have as few loss of quality as possible. Additionally, we want to have the translated picture to look as similar to the original picture as possible.

Thus, we take a note that the first task of color management is to translate between color spaces that are fundamentally different from each other.

Secondly, imagine we use the same kind of color reproduction device but two different individual devices. We look at one and the same picture on two different screens. We will see different color appearances even if the original data stays constant. For example when we take a constant set of input color data and we measure the trichromatic values in CIELAB or CIEXYZ that are emitted from two different screens, we will get results that are different both from each other and from the original data. But in an optimal workflow, we want the actual reproduced data to be the same as the input data. On the other hand, two screens may produce colors in the same way using the same kind of primary illuminants. Nevertheless the performance can turn out slightly different based on technical differences. The CIELAB or CIEXYZ values are not only different from the original data, but they are different from each other as well[12].

We make another note, that the second task of color management is to control the color reproduction performance in a way that the reproduced values are identical to the input data.

12 This could be due to the quality or consistency of led illuminants being used. LCDs for example are known for changing their performance over time. We want both screens, however, to produce the same color stimulus that stays constant over time.

In other words, imagine we measure the difference between two color stimuli of the same input color information being displayed on two different screens. The goal of color management in display technique would be to keep the color difference for identical color input information on different media devices as small as possible. The goal is to get "near-identical image representations" (Green&al., 248) on different capturing respectively output devices.

Gamut Mapping

How to tackle transformation between different color spaces and gamuts will be our next topic. Let us go back to the three gamuts for the camera, the screen and the printer we discussed above. We saw that there are many colors that can be recorded with the camera, which cannot be seen on a screen or on a print. In the same way, there are plenty of colors that cannot be seen on the screen, but which can be printed out on a paper. This situation is called "color mismatch" (Sharma, 648). Let us take a look at the gamut pictures of the color spaces combined in one single coordinate system. As we said before a gamut can be represented as a volume in a bigger color space e.g. CIEXYZ. Furthermore, a gamut has a surface that is called "gamut boundary" (c. Sharma, 649). Application like ColorSync make it possible to visualize gamuts of ICC profiles by mapping these gamut boundaries in a three-dimensional coordinate system. Moreover, ColorSync makes it possible to compare the gamut boundaries of two color spaces like for example two RGB color spaces (see Figure 2-7).

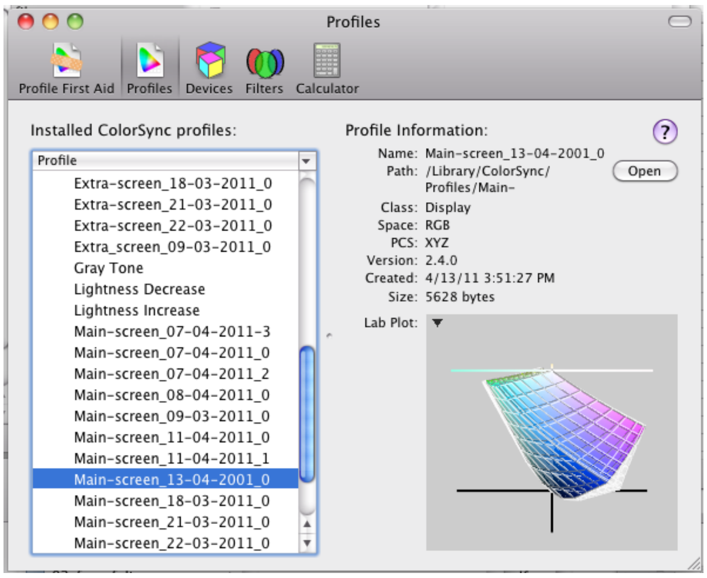
In the picture, we notice that some parts of both gamuts overlap and others do not. The parts that do not overlap (the white parts) are the very color mismatches. If we sent an unaltered value from the camera that is a color mismatch for a printer to the printing device, we would, at its best, get some confusing results. The printer would map it to a random color. The printer does not know how to interpret the color information, because it lies outside its range of producible colors, outside its gamut. What we have to do is to "alter the original colors to ones that [the] given medium is capable of reproducing" (ibid.). This can be done easily by applying a mathematical function that maps every value of the original color space onto values of the target color space. This procedure is referred to as "gamut mapping" (ibid.).

Gamut algorithms

The functions that are used to implement gamut mapping are called "gamut mapping algorithms" (Sharma, 669). And there are different approaches for designing a mapping algorithm.

Gamut Clipping and Gamut Compression

The basic one is that of gamut clipping (c. Sharma, 670), which means that the algorithm affects only such colors that lie outside of the target color space. These values from the original gamut are most commonly mapped to the nearest point on the gamut boundary. Values that lie within a subset of both gamuts stay unaltered. Problems with a such algorithm are mostly the loss of details, change in colors and a reduction of details (c. Sharma, 673f).



Installed ColorSync profiles:

Profile
Extra-screen_18-03-2011_0
Extra-screen_21-03-2011_0
Extra-screen_22-03-2011_0
Extra_screen_09-03-2011_0
Gray Tone
Lightness Decrease
Lightness Increase
Main-screen_07-04-2011-3
Main-screen_07-04-2011_0
Main-screen_07-04-2011_2
Main-screen_08-04-2011_0
Main-screen_09-03-2011_0
Main-screen_11-04-2011_0
Main-screen_11-04-2011_1
Main-screen_13-04-2001_0
Main-screen_18-03-2011_0
Main-screen_21-03-2011_0
Main-screen_22-03-2011_0

Profile Information:

Name:	Main-screen_13-04-2001_0
Path:	/Library/ColorSync/ Profiles/Main-
Class:	Display
Space:	RGB
PCS:	XYZ
Version:	2.4.0
Created:	4/13/11 3:51:27 PM
Size:	5628 bytes

Figure 2-7 Comparison of two RGB color spaces in ColorSync

The second basic mapping algorithm is that of gamut compression (c. Sharma, 677). In opposite to the clipping algorithm, the compression algorithm is applied on all values of the original gamut – not just the out-of-bound values. This means that the distances between multiple points of the original gamut stay proportionally the same in the target color space. The advantages of a compression algorithms are that they preserve the original's variation with the cost of a reduction in chroma (c. Sharma, 680).

Both algorithms have their pros and cons, depending on which part of a color space they are applied on. That is why there have been attempts of combining the advantages of both types.

This third type of mapping algorithms is called composite gamut mapping and is based on combining elements of gamut clipping with elements of gamut compression (c. Sharma, 681ff).

Whichever of these three types of mapping algorithm should be used depends on the kind of input data and the user's intentions. One could for example ask if the input data is a presentation chart that contains only iconographic pictures or if it is a photographic picture with many different colors and gradients. Also, one could ask about the accuracy of reproduction that is needed. Should the data be displayed on the Internet without need of accuracy but with the necessity of "looking good" at all means? Or is an accurate reproduction of an original needed for example a canvas painting of the 19th century? Of course, the user is not required to choose a specific mapping algorithm by himself, but it is very common that a color management workflow provides more user-friendly choices that result in different mapping algorithms.

Rendering Intents

A very common color management workflow is that of the ICC, which will be described in the next paragraph. The ICC defines four so-called rendering intents that use different mapping algorithms (c. ICC "Introduction to the ICC profile format", all following quotations taken from this side as well):

- *Perceptual intent*: Main goal is to "preserve detail throughout the tonal range" at the cost of exact contrast.
- *Saturation intent*: Main goal is to "preserve the vividness of pure colors" at the cost of exact hue. Useful for charts or diagrams.
- *Media-relative colorimetric intent*: "Rescales the in-gamut [...] such that the white point of the actual medium is mapped to the white point of the reference medium".
- *ICC-absolute colorimetric intent*: "In-gamut colors are unchanged". Useful for spot colors or proofing.

The last two colorimetric intents focus on preserving in-gamut values and would thus typically apply clipping algorithms respectively composites with focus on clipping. Whereas the first two would typically hold on to compression algorithms respectively composites with focus on compression.

To sum up the previous paragraph, we looked at how color values can be translated from one gamut to another via gamut mapping. We learned about the different gamut mapping algorithm types that can be used and we looked at the rendering intents that help us choosing the algorithm that best suits our needs. Our next step would be to investigate in how this algorithms would typically be implemented technically.

Intermediate Color Space

But first, we have to consider one more thing: We have to choose an adequate method for encoding our data, that means we have to choose a color appearance model that represents our color values. We have to choose an intermediate color space (c. Sharma, 254&652). An intermediate color space is basically a device-independent color space, in which all color transformations take place. Why do we need that? The colors being reproduced by a device are based on a device-dependent color space. The colors reproduced by a typical LCD screen for instance are based on a 8bit RGB color space. That means that every color can be described in terms of a triple of values between 0 and 255. The color green for example would be defined as (0 / 255 / 0), blue as (0 / 0 / 255) and so on. However, according to the color reproduction problem discussed above, a given RGB triple does not result in the same color stimulus when reproduced on different screens. The device-dependent RGB values may be the same, but the device-independent CIELAB values – they have to be determined by a measuring instrument – are different. That is why we need to describe device-dependent values in a device-independent color space first. That is why we need an intermediate color space.

The right choice of intermediate color space depends on different aspects (as discussed at Sharma, 254&652). Typically the intermediate color space is a superset of the gamuts being involved. (It has to be capable of describing all device-dependent values after all.) Intermediate color spaces one could use include for example CIELAB, CIEXYZ or CIECAM97[13]. The optimal choice depends strongly on the rendering intents, because all color spaces have their pros and cons for different goals. If the goal is to preserve the most accurate colorimetric representation, one would typically stick to either CIELAB or CIEXYZ. If the goal is to preserve the most pleas-

13 A color space that takes into account the viewing conditions along with the colorimetric values. Check Sharma, 176ff for further information.

ant representation under changing viewing conditions, one would preferably stick to CIECAM97 (c. Sharma, 653). Also one has to take in account, in which color space a designated gamut mapping algorithm works best.

Lookup tables

When it comes to implementing gamut mapping algorithms, there are basically two tools that proved to be best-practice: lookup tables and interpolation.

The purpose of a lookup table is to “precompute the transform for all possible digital inputs and store the corresponding outputs” (c. Sharma, 695). Typically a LUT for a screen RGB color space is visualized as three-dimensional lattice (c. Sharma, 696f and see Figure 2-4). Every node represents a value in the original space and carries the values of the corresponding point in the target space. In praxis, however, a lookup table is not implemented as actual three-dimensional lattice. Most often, a lookup table is just a text file containing a list of values. Every line represents a node in the source color space and the values represent the values in the target color space.

A single lookup table works very well for small gamuts. However a purely lookup table based transformation would lead to performance problems when dealing with bigger gamuts. The size of a such LUT depends strongly on the numbers of primaries and the depth of details. Let us look at the sizes of some typical LUTs that include every single value (ibid.):

1. 8 Bit RGB (for example standard screen): $3 \cdot 2^{3 \cdot 8} = 3 \cdot 2^{24} = 3 \cdot 2^{4+20} =$ 48 MB
2. 8 Bit CMYK (for example standard printer): $4 \cdot 2^{4 \cdot 8} = 4 \cdot 2^{32} = 4 \cdot 2^{2+30} =$ 16 GB

The size of one single LUT like this is already vast for itself and now keep in mind that this LUT has to be applied to every single pixel within the picture to be transformed. This is something only few CPUs and old graphic cards can handle.

Interpolation

The solution to this problem is fairly easy. One uses a LUT which defines a “lattice of nodes that partition the input color space into a set of smaller subvolumes” (Sharma, 695). In other words one defines a LUT using a smaller sampling that is only a subset of the original gamut. Afterwards, one uses “multidimensional interpolation for [any] input point that do not coincide with the LUT nodes”. Whereas interpolation is defined as “a method of constructing new data points within the range of a discrete set of known data points” (wikipedia/interpolation). In other words, when translating from the original to the target color space, this is what is going to happen: Given a specific input value, the processor tries at first to find a matching node on the LUT, and if that does not exist, it interpolates the missing value from the existing values on the lookup table[14].

14 The exact interpolation geometries and formulas are discussed at Sharma, 11.2.2 and 11.2.3. But they are not important for our investigation right now.

Given this background information, we can describe lookup table combined with multidimensional interpolation as following (c. Sharma, 695f):

1. Firstly, we find the subvolume of the lattice that the input value is element of.
2. Then, we retrieve the nodes that define the subvolume.
3. Finally, we interpolate the output value with the help of the corresponding values in the output color space of the retrieved nodes.

Using a combination of lookup tables and interpolation is from a computational standpoint usually less expensive than purely lookup table based gamut mapping. Most graphic cards handle interpolation rather fast. The speed can additionally be increased by caching certain node values (c. Sharma, 710).

To sum up, we have seen that color management requires gamut mapping between different color spaces. Which mapping algorithm to use depends on the user's intents. Usually these mapping transformations are implemented through a combination of lookup tables and interpolation.

As end-users, we usually do not have to worry about mapping algorithms. Many color management workflow exist that implement various different algorithms and that let us choose different options depending on our actual intent. The most common used color management workflow is that of the ICC, which I will discuss in the next paragraph.

The ICC workflow

We will now take a closer look on how to implement color management, that means how to translate between multiple color spaces.

Closed-loop vs. device-independent

To begin with, we can say that there are basically two ways of organizing color management: A closed-loop workflow and a device-independent workflow (c. Sharma, 283). A workflow consists usually of multiple devices. There are various input devices like cameras or scanners and output devices like screens and printers. Whenever we transfer data from one device to another, we have to apply color management.

A typical closed-loop workflow would translate directly from one device's color space to the other device's color space. Or in other words a "specific [...] device is optimized for rendering images to a[nother] [...] device" (Sharma, 283). The problem with this workflow is the vast number of transformations that needs to be implemented. Imagine you work in a media department that owns three photo cameras and you have four employees with each his/her screen. You would need exactly $3 \cdot 4 = 12$ color transformations for a complete closed-loop color management (c. Sharma, 254). Now imagine a very big media department with multiple cameras and multiple employees, you would need to apply a total of number of cameras multiplied with the number of employees to get a valid color management. And it becomes even more complicated if we take into account printers and other devices as well. This is highly ineffective; such a workflow should not be used for large scales. A closed-loop can work though for a small, relatively restricted workflow with a manageable numbers of media devices.

For a complex setting, the device-independent workflow is much more suited. In such a workflow, every device is independently translated into a intermediary device-independent color space (c. Sharma, 283). This way, every device has to be translated just once, which means that the total number of transformations equals only the sum of numbers of devices involved. A media department with 12 cameras and 35 screens results thus in 47 transformations assuming that we apply a device-independent workflow (in contrast to 420 transformations with a closed-loop workflow). The intermediary device-independent color space is usually identical with the intermediary color space that is needed for the gamut mapping transformations we talked about before. One would typical use a standard colorimetric color space like CIEXYZ or CIELAB.

The benefit of a device-independent color management workflow is not only that of fewer transformations. It is also easily portable between different platforms and it can be extended by an almost unlimited number of additional devices. In fact the number of involved devices is not important because it is only the transformation into the device-independent color space (DVI, after Sharma, 104) that is relevant. In praxis such a workflow would work look like this: A picture taken from one camera is translated into the DVI.

ICC components

A very common implementation of this workflow is that of the International Color Consortium (ICC) that is used in most operating systems like Mac OS and Windows. The ICC color management architecture consists of four main components (c. Green&al., 249):

1. *The color management framework*
   It is "responsible for the most important color management functions" and it "provides an interface to the various color management methods" (ICC "Color Management: Current Practice"). On Macintosh computers this framework is called ColorSync and it provides a bunch of methods and functions to access profiles etc. We will see more of ColorSync in Part B.

2. *The color management module (CMM)*
   It is responsible for connecting color spaces by computing any transformations or interpolation between the two color spaces that is needed (c. Green&al., 249). It creates a link between the two profiles (s. profiles) of the original and the target color spaces. Most operating systems and some applications[15] have a default CMM that is capable of applying basic color transformations. The ICC architecture knows on the other hand no limitation of CMM extensions. That means any third party CMM can enhance a default CMM. The main task of my bachelor thesis will be to program such a CMM extension for the Drylab Keyframe application.
   Another important thing related to the CMM is the question of the right Profile Connection Space (PCS), in which all transformations take place. More about the PCS just below.

3. *The application*
   Any application can use one of the default or 3rd party CMMs to handle color management. As mentioned above, we will look at the Drylab Keyframe application that implements such an ICC architecture under Part B.

4. *The ICC profiles*
   The profiles define the device models involved in the workflow "by providing the relationship between the device coordinates and those of the reference color space [the PCS]." They are so to speak the dictionary from the device's color space into the intermediate color space.

The Profile Connection Space (PCS)

As we saw above, the whole ICC color management workflow is based on the idea of computing all transformations through a common device-independent color space. This color space is called Profile Connection Space (PCS). As it name suggests, its main purpose is to provide "an unambiguous connection between the input and output profiles […]. It is the virtual destination for input transforms and the virtual source for output transforms" (ICC "Introduction to the ICC profile format"). What does that mean? Imagine we import a picture from a camera in order to display it on a screen. With a well adjusted ICC architecture, this would work like this: First the

15 Especially image processing applications like Photoshop or Avid.

image data is translated from the camera's color space into the PCS, and eventually the color information is translated from the PCS into the screen's color space. Usually the PCS is one of the CIE-based reference color spaces like CIELAB or CIEXYZ (c. Green&al., 249).

With the PCS, we only need one transformation for each device to respectively from the PCS. And this transformation stays somewhat constant over a reasonable period of time (s. calibration and characterization below). Now, we just need a way to attach this information about transformation to the device so that it could easily be read and used by any CMM. This attachment exists and it is called an ICC profile.

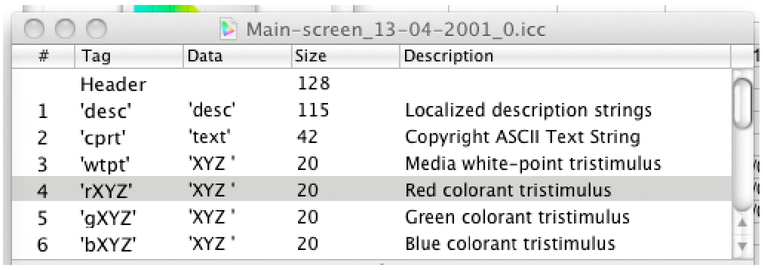
The ICC profiles

The ICC profiles are data files that provide "the information necessary to convert color data between native device colour spaces and device independent color spaces" (ICC.1:2004-10, 9). The ICC profile describes the device and its color characteristics. These are the most important information that can be included in an ICC Profile (c. Green&al., 253):

- The transformation data that is needed to convert color information from the device's color space to the PCS in form of one or multidimensional lookup tables.
- Information about how to convert from the PCS back to the device's color space if needed. This is not required for all types of devices. Input devices do not need that kind of information for example because they merely collect color information that is transferred into the PCS. Output devices however necessarily need that kind of transformation in order to display a given input data correctly.
- Metadata that describe the device. This metadata includes name and type of the device, brand, company, its color characteristics like white point and coordinates of primaries etc.

The ICC file format is based on a tag-based structure (c. Sharma, 253 and see Figure 2-8), that means it provides a set of predefined tags, between which the data is stored. More than that, the ICC made it possible to include a variable amount of tags that can be defined by 3rd parties as required. There is for example a predefined tag called technologyTag that contains information about the device's technology such as film scanner, LCD or ink jet printer (c. ICC.1:2004-10, 49). Another tag called copyrightTag contains information about the profile copyright information (c. ICC.1:2004-10, 105) and so on. The tags that are most interesting for us – translating into and from the PCS – are called redTRCTag, blueTRCTag, greenTRCTag and some others.

What kind of data is stored and how we can use this data for proper color management? This question is closely related to how we get the transformation from device-dependent color space to DVI in the first place and how to make sure that this relation stays constant over time.


Main-screen_13-04-2001_0.icc

#	Tag	Data	Size	Description
	Header		128	
1	'desc'	'desc'	115	Localized description strings
2	'cprt'	'text'	42	Copyright ASCII Text String
3	'wtpt'	'XYZ '	20	Media white-point tristimulus
4	'rXYZ'	'XYZ '	20	Red colorant tristimulus
5	'gXYZ'	'XYZ '	20	Green colorant tristimulus
6	'bXYZ'	'XYZ '	20	Blue colorant tristimulus

Figure 2-8 Tags of an ICC-profile

Device calibration

The color performance of one and the same device can vary drastically over time. This is because the technical elements get used up etc. It is therefore most important that we put our device in a known state before we begin our actual color management.

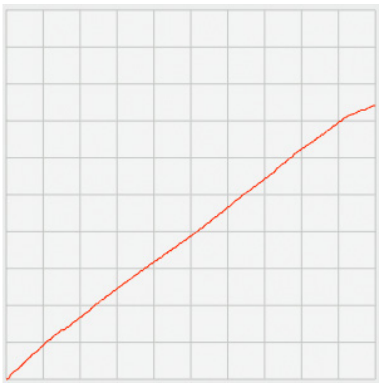


Figure 2-9 Red channel TRC

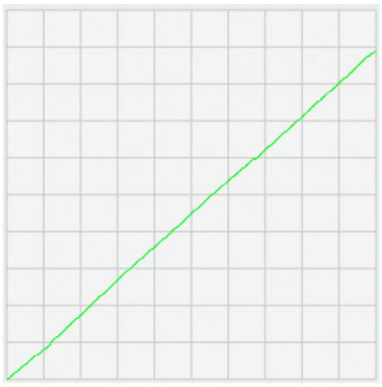


Figure 2-10 Green ch. TRC

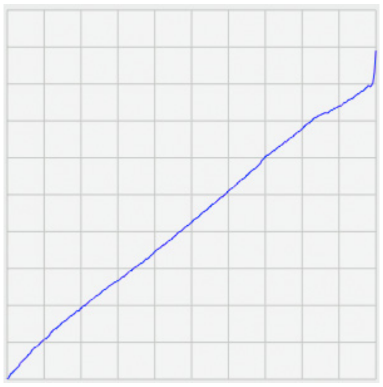


Figure 2-11 Blue ch. TRC

This process is called calibration, which is "the process of maintaining the device with a fixed known characteristic color response". As we discussed above, any given input data may result in a wrong color representation.

Nakamura describes this problem as following: "Many imaging devices have nonlinear characteristics when it comes to the process of capturing the light of a scene and transforming it into electronic signals. Many displays [...] have nonlinear characteristics" (Nakamura, 233). This phenomenon is often visualized with the help of so-called tone reproduction curves (TRC).

TRCs can be plotted for each channel individually or all three combined as gray scale TRC. For a individual channel, the input values from 0 to 255 are plotted against the measured luminance in order to obtain the channel's TRC (c. Sharma, 284). A typical result can be seen above.

As we can see, the input values are not plotted linear, but in a irregular curve. This is what Nakamura meant by saying "nonlinear characteristics". Another way is that of gray-balancing, where "equal amounts of device color signals (e.g., R = G = B or C = M = Y) correspond to device-independent measurements that are neutral or gray (e.g. a* = b* = 0 in CIELAB coordinates)" (Sharma, 284).

Device calibration means to use these TRCs "to linearize the device" (Sharma, 717), in other words to linearize the output performance. Given the nonlinear TRCs, the CMM can easily compute the inverse functions for each channel, which "neutralize" the nonlinear TRCs. After the inverse TRC functions are applied, the result is a linear output performance. And a linear output performance means that our device is put in a known state. We can now begin to characterize the data with respect to the PCS.

Device characterization

Device characterization is defined as deriving "the relationship between device-dependent and device-independent color representations for a calibrated device" (Sharma, 285). In other words it is the process when every value in the device-depended color space is mapped to a value in the PCS. There are two types of characterizations, forward and inverse characterizations: "The forward characterization transform defines the response of the device to a known input, thus describing the color characteristics of the device. The inverse characterization transform compensates for these characteristics and determines the input to the device that is required to obtain a desired response." (Sharma, 286)

The characterization is slightly different for input and output devices, for output devices like screens or printers, this looks as following:

- Forward characterization
  A given device-dependent value is displayed on the screen and measured in order to obtain the device-independent coordinates (c. Sharma, 286).
- Inverse characterization
  Defines to each device-independent coordinate the required device-dependent input that is need to gain this very device-independent output.

Technically, characterization is done by measuring empirically the signals of a set of color samples that are shown on the screen. The color samples typically include a limited range of shades of the primary and secondary colors plus a set of memory colors and the device-dependent values of these color samples is known. Then, the output values in the device-independent color space are measured with the help of a colorimeter, a spectrophotometer etc. Finally, the output data is related to the input data in form of a lookup table. The inverse of that LUT can eventually be used as foundation for a gamut mapping algorithm between PCS and device-dependent color space.

To sum up, we learned now how to work with a color management workflow that is based on a device-independent color space using the example of the ICC workflow. We then learned the different components of the ICC workflow, especially the profile connection space (PCS) and the ICC profiles.

Now, we have everything we need to know about color management in general to program our very own color management module for Keyframe. What is more, we need to find out how the CMM in the Mac OS works and what functions we need to add to gain an optimal color management for our pictures.

The Macintosh Color Management Module

The Default Mac OS CMM

The Macintosh Operating System (Mac OS) uses a color management architecture that is called ColorSync. Apple says: “ColorSync [...] provides an interface to system-wide color management settings” (Apple “ColorSync Manager Reference”). In other words, ColorSync provides an interface that allows programmers to program a color management module (CMM) for a given application. As we will see in part C, Drylab’s Keyframe is an application that runs exclusively on Macintosh computers and that is why I used the ColorSync framework to program Keyframe’s very own CMM. Before jumping into planing Keyframe’s CMM however, we will first look at the default CMM that is used by the Mac OS. This way, we can avoid redundancies. We do not have to start our own CMM from scratch but we just need to implement the functions that are not included in the default CMM yet.

In the Mac OS, we have the possibility to upload a screen profile – one we got after calibrating and characterization the screen for example as described later under part C. Then the Mac OS calibrates the screens independently with the help of the “vcgt”-tag.

It does not implement, however, an inverse characterization function, that means it does not properly map from device-dependent color space to device-independent color space. It displays device-dependent input data arbitrarily. In other words, we have no way of knowing how any given input data is displayed in terms of a device-independent color space.

Let us follow a picture through the Mac OS color management process. We assume our picture is encoded in sRGB color space that means every color information is processed as a triple of values from 0 to 255 for each of the channels red, green and blue. As mentioned above, the sRGB color space is a color space that is based on a optimal virtual output device[16]. The sRGB specification contains information about the CIEXYZ coordinates of the virtual primaries and it defines how to translate from sRGB device-dependent color space to device-independent color space CIEXYZ (c. IEC 61966-2-1:1999). With this specification engineers created a sRGB profile that can be attached to any image like the one we are using for this example. Furthermore, the sRGB profile enables the CMM to compute a transformation that translates any given sRGB value into a fixed CIEXYZ value by characterization. The default Mac OS CMM does not do that because it does not enable characterization.

What does that mean for our picture? The image data contains color information that is based on RGB triples. These triples can easily interpreted by the engine that displays the picture on the screen because it uses a color space that is based on sRGB. However, the CIELAB values it finally displays on the screen, after the calibration function is applied, are not identical with those that are defined by the ICC sRGB specification[17].

16 The color space was originally developed for CRT monitors.

17 That is because the coordinates of the primaries are slightly different for the actual screen than if compared with the optimal virtual device’s primaries.

What can we do to improve that? – We need to enable characterization.
That means we need to find a way to compute a transformation from sRGB to the screen's color space so that the screen displays the right CIELAB values according to the sRGB profile. This transformation would transform the original sRGB triples in screen triples that produce the desired color response. My main task for this assignment will be to enable this characterization for Drylab's Keyframe by computing lookup tables for the pictures, which are based on the ICC profiles of both the computer screen and the original picture.

To sum up, we saw what tools and engines are provided by the Macintosh Operating System. We found out that the default Color Management Module implements calibration only. That means we need to enable characterization for application. How this can be done, we will see in part B.

PART B

Implementing a Color Management Module

After having explored the theoretical background of color managing, we will now move on to the practical implementation of a Color Management Module (CMM) for Drylab's Keyframe RushesControl application.

Who is Who? And What Is What?

Introducing the Actors

Who is Drylab?

Drylab is a small software company located in Oslo, Norway. It is developing application components that are aimed at supporting the workflow of film technicians. Two of the main founders of Drylab are John Christian Rosenlund, director of photography, and Andreas Herzog, digital image technicia and they have been working in the Norwegian film industry for many years and have been contributing to some of the most popular films in Norwegian movies during the last years[1]. They both know the working routines on a film set on a daily bases and they have been thinking about how to simplify them.

Working as a camera assistant for example involves quite a lot of paper work. Paper work that has to do with the collection of information for each scene. More specifically: The recording of a movie is divided into multiple scenes, which are divided into sets, which are divided into takes. All sets involve different camera positions, different focal length, different f-stop values, exposure times, background information about the lighting settings etc. That is the information that needs to be collected because it can be very helpful in cases that a shot has to be redone after several weeks or it is helpful by supporting the color grader at the end of the movie making workflow. The information has to be collected and organized so that everyone can practically access and use them afterwards. So far this was done with pen and paper because not many companies were providing an application to handle this problem. That is the main reason why Rosenlund and Herzog decided to make an application to organize this information. Eventually, they got help from Arne Magnus Bakke who is a Ph.D. student for informatic engineering at the College University Gjøvik and Tariq Islam, and they began working on an application that was later known as Keyframe RushesControl.

1 Rosenlund for example was the director of photography for "Den brysomme mannen", a movie that won three Amanda awards in 2006, which is the highest ranked award in the Norwegian film industry.

What is Keyframe RushesControl?

Keyframe RushesControl is part of the Keyframe family, which includes workflow tools that support the collection and organization of information related to the movie recording from the recording on set, color grading to post production and editing. A film workflow usually works like this: The director of photography captures the pictures, the pictures are sent to the lab, where the film gets digitalized. Then, a color grader does his/her job, the post production team creates additional content and the editor finally puts the different scenes together to a complete movie.

Within this workflow, the Keyframe concept focuses mostly on the needs of the director of photography (DoP) and his/her assistants. Keyframe supports the communication between the DoP and the people working on the picture. For example, the DoP and his/her assistants need to record technical details on set about the camera settings. Within the Keyframe concept, this can be managed with the help of CameraReport, an iPhone app that provides a GUI for data input that is needed, and which sends the collected data to a server for backup. Furthermore, the data on the server can be accessed by anyone involved in the film project, both immediately under production or later when needed for other tasks.

Another big task of the DoP for example is that of color grading. The DoP does not usually do color grading by him/herself, but he/she communicates his/her ideas to the color grader who sets the colors and mood of the final picture. One way to hand down the ideas is verbally or in written form, although the disadvantage is very obvious: Words can only vaguely describe, what later should be seen on the screen. The Keyframe concept offers help in the form of Keyframe RushesControl, a desktop application that combines the pictures that have been recorded with the data that has been collected on the set. But what is more, Keyframe provides a grading editor that allows the DoP to edit a picture after his/her ideas. These pictures can later be exported as prototypes to the color grader. It is not very common that the DoP applies final color correction, though, because the software that is needed for the final grading is usually very expensive and requires a perfectly calibrated environment – something, which is usually not available on the set. Furthermore, a common grading software does not provide a sufficient interface to organize the previously collected data.

To sum up, we can see of the basic tasks of Keyframe components on the picture on the next page (see Figure 3-12). Keyframe RushesControl is an easy-to-use application that is used on the film set to visualize ideas and to easily communicate with the color grader.

What does already exist?

The Keyframe RushesControl (RC) application is already in its final state of development, that means the basic and most important features do already exist. Arne Magnus Bakke and Tariq Islam have already programed many features that enable color correction, downloading information etc. But one important feature is missing: a comprehensive color management architecture. Usually, a DoP uses two screens: One main screen for the application with the GUI, buttons etc. to pregrade the picture and another extra screen to open another preview window so that he/she can manipulate colors and look at the picture in full size simultaneously.

As I discussed in part A, the Macintosh OS provides color management only to the degree that it calibrates the monitors. The Mac OS does not however enable characterization. Thus, we need to program a Color Management Module (CMM) for our Keyframe application that enables

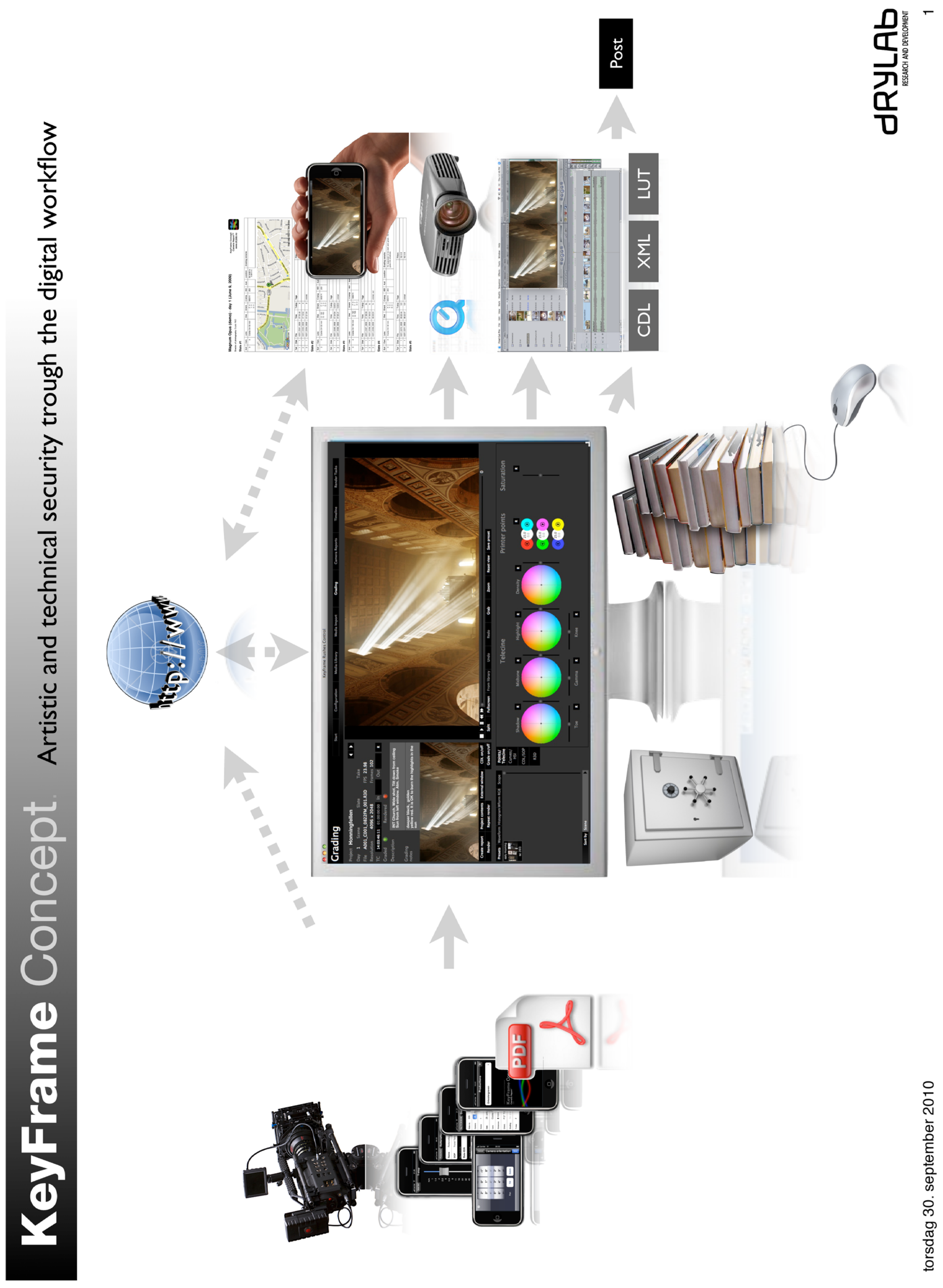

KeyFrame Concept.
Artistic and technical security trough the digital workflow
http://www
CDL
XML
LUT
Post
dRYLAb
RESEARCH AND DEVELOPMENT
1
torsdag 30. september 2010

Figure 3-12 Basic screenflow of the Keyframe concept

this very feature. Ideally, the characterization should be based on the ICC workflow, which requires that the CMM gets access to the ICC profiles of the two screens and that it computes the gamut transformations separately for both screens. Additionally, we want to enable another feature that allows uploading Nucoda lookup tables, a LUT format that is widely used amongst film technicians. Moreover, we need a user-friendly user interface that provides a clear overview of the color management settings. But before all that, we want to look at the technology that we will be using to program this feature.

Introducing the Technology

The Keyframe application is programmed in C++, one of the most widely spread programming languages in the world. Some of the best known operating systems are programmed in C++ respectively its predecessor C: Windows, Unix and the Macintosh OS. I will not discuss the details of the C++ language but I will focus on two of the most important libraries that I used to program my CMM: FLTK, the GUI library, and littleCMS, the ICC transformation library.

What is FLTK?

FLTK is a “C++ GUI toolkit” (Fltk “Introduction to FLTK”) running on several operating systems including Linux, Windows and Mac OS. It is a library that provides the most frequently used GUI elements like buttons, labels or text areas and offers a framework to easily access and integrate their functionality into our own application.

Why do we need such a library? Imagine we did not have a library that provides a consistent GUI framework. We would have to program every graphical element that is shown on the screen from scratch by hand. This would not be very effective as we wish to focus mainly on the functionality of our data organizing and pregrading features. On the other hand, the FLTK library offers a whole bunch of classes and functions that can easily be altered to present the desired graphical element on the screen. Those functions include for example the event handlers. When we click on a button, an event is triggered and we can allocate multiple behaviors like for example opening a menu or showing a preview. Frameworks like FLTK handle the back-end work of events like that. Thus, we do not have to worry about how to trigger an event but we can focus on what we want to happen after the triggering. And thus focusing on the functionality of our program.

What is littleCMS?

LittleCMS is a “small-footprint color management engine” (LittleCMS “About Little CMS”). It is a library based on the ideas by the ICC workflow and it provides classes and methods that help to compute color transformations between different ICC profiles.

LittleCMS can open, interpret and create ICC profiles. It can compute color transformations between two color spaces and it can apply these transformations to a given picture.

The littleCMS library comes in handy, because it provides a set of best-practice functions for easy and fast standard color transformations. It spares us the time of programming basic color transformations by hand.

Introducing the Code

Code inventory

As mentioned before, the fundamental classes and functionality have already been programmed for Keyframe. It is not time or place to present all classes in detail but I want to discuss one specific class, GlImage, because it is basically the heart of our CMM. This is because it represents both an ordinary image and the lookup table that enables all transformations.

The lookup table class GlImage

The GlImage class, which was programed by Arne Magnus Bakke, has basically two functions within the Keyframe code – two functions that might sound contradictory in the beginning.

On the one hand side, the class can be used as wrapper class to display two-dimensional pictures on the screen. In this role, it provides functions like draw(...) that specifies where and how big the picture is supposed to be displayed on the screen or the thumbnail attribute, which is "a smaller (thumbnail) size copy of the image in RGB format" (c. source code).

On the other hand side, GlImage can serve as three-dimensional lookup table that is used for color transformations – together with some interpolation algorithms.

How is it possible that one and the same class can serve as both RGB image and lookup table? Well, when you look at it, a LUT is nothing but a three dimensional picture by definition. The coordinates of the LUT nodes represent the RGB values in the original picture and every single of these nodes carries the values in the target color space as an RGB triple. In contrast, a two-dimensional image is like a two dimensional map where each pixel or node has its individual RGB value attached. GlImage provides the possibility of multi-dimensional RGB images, that means both two-dimensional and three-dimensional. Ergo it can be used for both two-dimensional pictures on the screen and three-dimensional lookup table.

OpenGL

The GlImage class is based on another library called OpenGL (that is where it name comes from).

OpenGL is known to be one of the "most widely used and supported 2-D and 3-D graphics application programming interface (API)" (OpenGL "OpenGL overview"). It is a library, programmed in C, whose main purpose is to effectively handle image processing by providing basic classes and functions that implement standard rendering and shading algorithms. OpenGL is providing the functions that are directly computing and producing the image information that is to be displayed on the screen. For our CMM GlImage class, OpenGL comes in handy because it contains one-and-multi-dimensional texture classes that can be used for color transformations. Textures are basically lookup tables that are applied on a given set of input data. As described under part A, a CMM that follows the ICC workflow extracts information from the profile and converts it into three independent one-dimensional lookup tables, each of them will be applied on each color channel (c. Green&al., 257). In the GlImage class, it is the one-dimensional OpenGL textures that act the part as these one-dimensional LUTs. They hand the transformation data to the shader, which finally applies it on the image that is to be color managed.

In the previous paragraphs, we got to know the company and parties involved in this project. Furthermore, we got a short overview of the application and its functionality. Secondly, we had a short introduction in the technology being used: We learned about the GUI library (FLTK) and the color engine library (littleCMS). And finally, we had a short introduction in the existing code especially the GlImage class GlImage.

Goals and Tasks

The main goal of my thesis was to program a color management module that can be used as framework for all different kinds of color transformations within the Keyframe application. More specifically, I wanted to enable color management that is based on the ICC workflow and I wanted to enable the possibility to upload Nucoda lookup tables.

Therefore, I needed to program classes that use the littleCMS library and integrate them into the existing source code. littleCMS provides some solid transformations from one color space to another so that I did not have to program this by hand. I had to make sure, however, that I could access these functions from Keyframe. Furthermore, I had to make a graphical user interface (GUI) where the new feature was visualized. Also, the user was given the possibility to select the new color management options with this new GUI.

In the next chapter, I want to present my color management screenflow and its GUI representation on screen.

The Graphical User Interface

Before we think of a graphical user interface (GUI) for our color management feature, we will take a look on what already exists. On the start screen of the Keyframe application, we can create new projects or load and open existing one. Once we have opened a project, we can choose between different tasks: We can upload media files – both moving pictures and photographs –, we can grade them, we can organize our metadata and we can select our overall project and application settings. Apart from general settings like the projects name and its most important actors, we have a tab that is called lab. Here we see fields that contain information about the ratio and the ICC profiles used for the pictures of the project. But these field do not have any functionality yet. This tab is where we later will integrate our color management workflow.

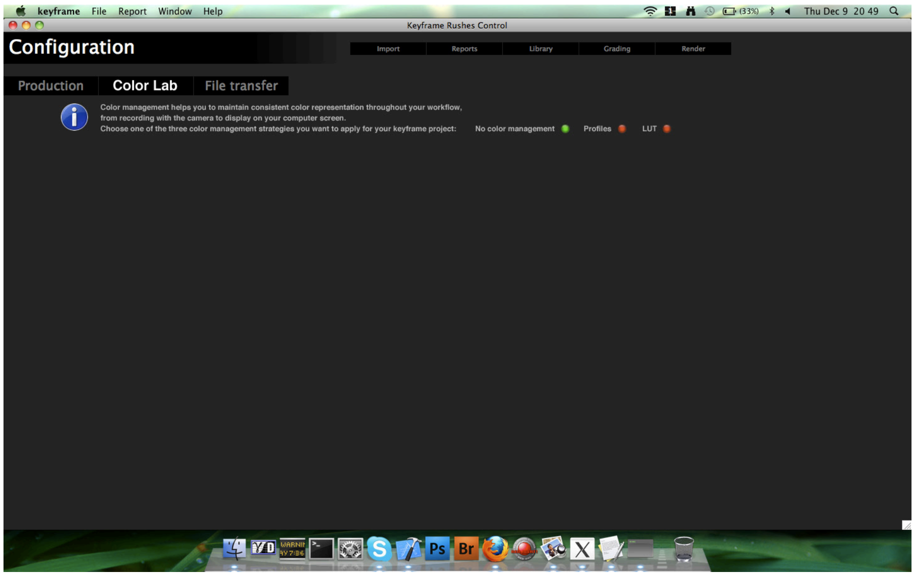
The GUI Draft

To begin with, we will now think of a draft for the color management workflow and the GUI of the feature in Keyframe RushesControl.

By visualizing the problem, it is easier to communicate with others about the project.
As I described before, there have been various actors involved in managing this project, and most of them were working in completely different places. Arne Magnus Bakke, Tariq Islam and I were seated in Gjøvik, Andreas Herzog and John Christian Rosenlund in Oslo. The distance between these two places required a big effort of preparation before meeting and discussing the elements that were desired and need for the color management feature. Especially the graphical elements of our workflow GUI are hard to describe verbally, in the same way as our final color management feature can only be described insufficiently with words because it is a feature that is supposed to be interactive. In other words by drafting a general GUI of the final feature so early, I could more easily communicate to others at one glance what I was planing to do. Considering this, I came up with the GUI drafts that I made in Photoshop and that can been seen on the next page.

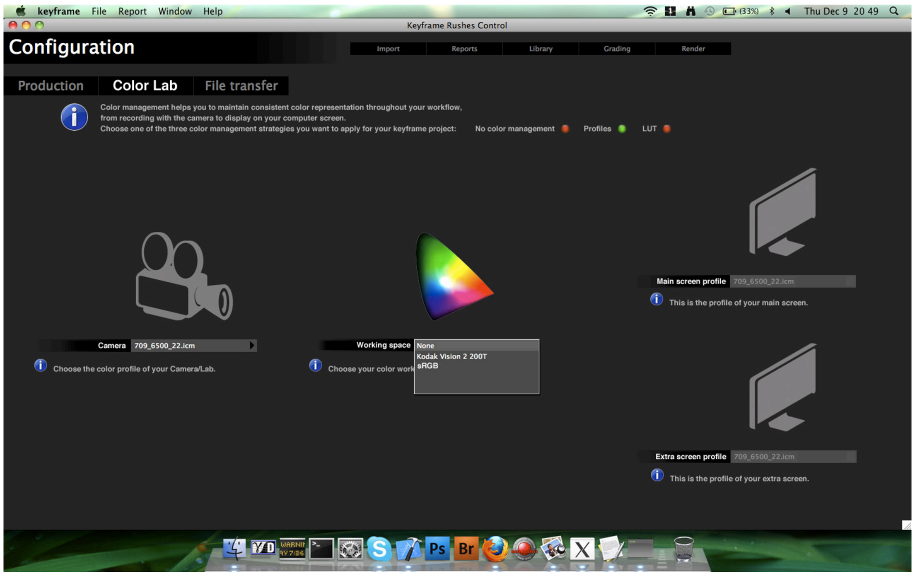
Color management is a very abstract thing and it is somewhat difficult to make it comprehensible to the user. As I pointed out in part A, people are often not aware of it, even if the majority of them have already been confronted with it. That is why I decided to separate the necessary options visually and as clearly as possible. What is more, I added help texts for every important element explaining what they can be used for.

Let us take a look on the color management tab now, which I renamed to "Color Lab". On the top, we see a help text explaining what the tab can be used for color management and describing what color management is. Furthermore, we see a group of radio buttons that let us choose one of the three color management workflow options. With John Christian Rosenlund and Arne Magnus Bakke, we decided to provide these three – or better two and a half choices – for our color management settings: no color management, color management based on profiles and color management based on Nucoda LUTs.



Color management helps you to maintain consistent color representation throughout your workflow,
from recording with the camera to display on your computer screen.
Choose one of the three color management strategies you want to apply for your keyframe project: No color management 🟢 Profiles 🔴 LUT 🔴

FIGURE 3-13 Draft for “No Colormanagement” activated



Color management helps you to maintain consistent color representation throughout your workflow,
from recording with the camera to display on your computer screen.
Choose one of the three color management strategies you want to apply for your keyframe project: No color management 🔴 Profiles 🟢 LUT 🔴

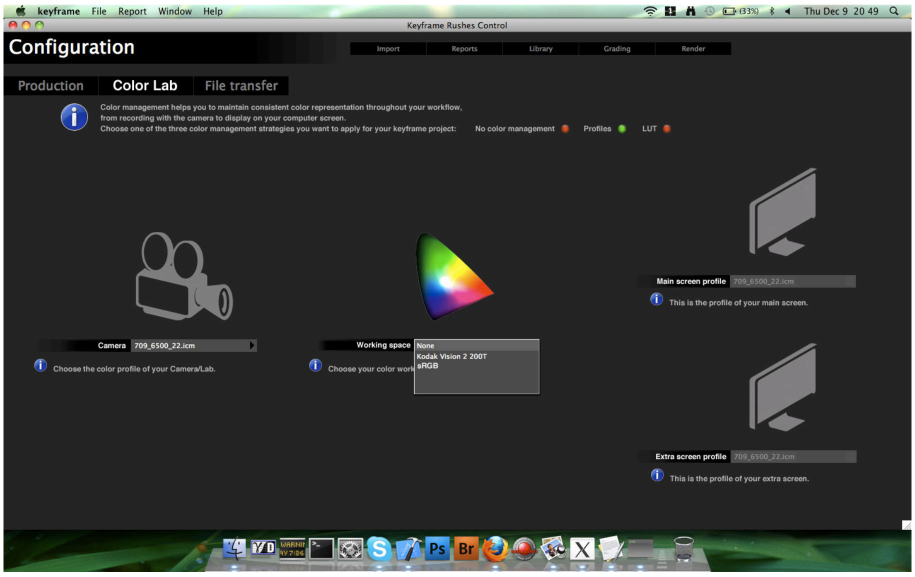
Camera 709_6500_22.icm
Choose the color profile of your Camera/Lab.

Working space None / Kodak Vision 2 200T / sRGB
Choose your color work

Main screen profile 709_6500_22.icm
This is the profile of your main screen.

Extra screen profile 709_6500_22.icm
This is the profile of your extra screen.

FIGURE 3-14 Draft for “Profiles” activated



Color management helps you to maintain consistent color representation throughout your workflow,
from recording with the camera to display on your computer screen.
Choose one of the three color management strategies you want to apply for your keyframe project: No color management 🔴 Profiles 🟢 LUT 🔴

Camera 709_6500_22.icm
Choose the color profile of your Camera/Lab.

Working space None / Kodak Vision 2 200T / sRGB
Choose your color work

Main screen profile 709_6500_22.icm
This is the profile of your main screen.

Extra screen profile 709_6500_22.icm
This is the profile of your extra screen.

FIGURE 3-15 Draft for “LUT” activated

No Color Management workflow

Firstly, “No Color Management”[2] is set as default. No additional characterization transformations are applied to the pictures in the preview windows. With this option, color management is completely in the hands of the Mac OS as described in part A before.

People who work on a computer that does not allow calibration or does not support profiles might use this option. Maybe, the user thinks that things have to be fast or they just do not care about color management. When this choice is selected, no gamut transformations at all are applied to our picture. Thus, the rest of the color lab panel is left blank.

Camera Type and Working Space

The next two options do have two fields in common that do not change for either one. The first option is that of the camera type being used, and the second is that of the selected color working space being used.

On the one hand side, the camera choice is just a dummy field yet. We see a drop box on the left side of the panel that allows us to select one of three camera types supported by Keyframe RushesControl (Red Canon, 5D and Arri). Choosing a camera does not have any functionality yet but additional features may be added later. A future feature could be for example the option of calibrating and characterizing a film camera. Something that is not very common in the film industry just yet. Also, one could thing of an option to upload camera profiles.

On the other hand side, the choice of working space is really important. The working space is the color space that describes the encoding for color information of the picture or movie. It is the source that the CMM transforms into the color space of the screens from. We decided to provide the six most widely used color space standards in the film industry. These standards are used literally all across the globe:

- **Rec. 709 (ITU-R Recommendation BT.709)**
  Standard for high-definition (HD) television (c. BT.709-5 (04/02)).
- **Rec. 601 (ITU-R Recommendation BT.601)**
  Standard for standard-definition (SD) television (c. ITU BT.601-6 (01/07)).
- **sRGB**
  Standard RGB color space for monitors, printers and the Internet. Based on Rec. 709. (c. W3C “A Standard Default Color Space for the Internet - sRGB” and ICC “IEC 61966-2-1:1999”).
- **Adobe RGB**
  Standard developed by Adobe that should be able to display most colors achievable by a printer with a device that uses RGB primary colors like a computer screen.

2 The option that I consider as the “half option”.

- **PAL (ITU-R BT.470-6)**
  One of the three most common standards used in broadcast television systems. Main areas are Europe and almost any other continent except North America (c. wikipedia/PAL).
- **NTSC (ITU-R BT.470-7)**
  One of the three most common standards used in broadcast television systems. Main areas are North America, Japan and parts of South America (c. wikipedia/NTSC).
- **SECAM**
  One of the three most common standards used in broadcast television systems. Main areas are France, Africa and Northern Asia including Russia (c. wikipedia/SECAM).

After I had defined the selectable color spaces, I downloaded the ICC profiles from the respective websites and stored them in a common folder within the application's file system so that they could be accessed by the CMM. Furthermore, I made a drop box In the middle of the color lab tab where you can choose one of the described color spaces.
Now that we have defined the elements that are common for the profiles and LUTs workflow, we will look at the elements that are very individual for each option.

Profiles Workflow

As second option, we want to enable a color management workflow that is based on the ICC profiles of the screens. This will probably be the most common choice for all users.
This option is based on the ICC workflow as described under part A: After calibrating and characterizing a monitor, the calibration software usually creates an ICC profile that is selected under "System Preferences > Displays > Color" in Mac OS. These ICC profiles are the foundation for our ICC based workflow option. More precisely, I added two read-only fields on the right side of the panel that show the chosen ICC profiles for both main and extra screen. I assumed that the preview of the grading panel will always be opened on the main screen whereas the external window will be opened on the extra screen. The fields are read-only because the choice of profiles should be handled centrally by the operating system under "System Preferences" as mentioned above.

LUTs Workflow

As third and last option, we want to enable a color management workflow that allows the user to upload self-generated lookup tables from the working space to each color space of the two screens. This is useful for those film technicians who use stand-alone programs to compute such LUTs. For the beginning, we decided to enable transformation only for Nucoda files because Drylab has had several cooperations with this company in the past already. As for the profile option, it is possible to choose the working space. However, the CMM gets all the information that is needed for color transformations from the uploaded LUTs alone. We decided to keep the color space field however so that the user can always keep in mind, which color space he/she is using
To upload the files, we added two file input fields that open a file selector dialog to select the desired LUT.

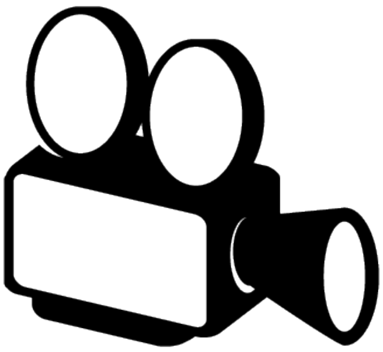


FIGURE 3-16 Camera icon

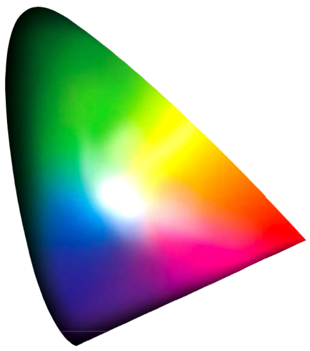


FIGURE 3-17
Workingspace icon

FIGURE 3-18
Screen icon

To sum up, we discussed the various elements for the different workflow options on the “Color Lab” tab. As we can see, we have three different areas on the lower part of the color tab. The first on the left is related to the camera, the second in the middle to the color space and the third on the right to the screens. To separate these areas more easily visually I created three icons that represent each of the options named before.
We will now look at their respective functionality.

GUI functionality

How does the user use the color tab?
The user opens the “Color Lab” tag and can choose one of the three options; by default “No color management” will be chosen.

1. **“No color management”**
   The user will only see the introductory help text and the three radio buttons with the color management choices.
2. **“Profiles”**
   The user can now choose a camera, the working space of the picture or movie and he/she sees the profiles of the main and extra screen being used. When choosing a new color space the help text changes and displays some background information about the chosen profile.
3. **“LUTs”**
   The user can now choose a camera, the working space being used and he/she can select and upload two Nucoda LUTs for the main and extra screen being used.

Now that we have the draft, we just need to program the elements and add the functionality. We start typically by programing a dummy GUI for the application.

Dummy GUI

In the beginning of programing, I implemented a dummy GUI in the actual Keyframe application. That means, I added all elements, widgets, icons and labels on the “Color Lab” tab as described above, but without any functionality enabled yet. That means at this point of the development, there was no reaction at all when clicking on a drop box or selecting a different radio button.

I did that, because of several reasons. Firstly, as I have not worked with the program before, I figured this would be a great opportunity to get used to the source code and the various classes being involved. Moreover, it helped me to get a general feeling about the libraries FLTK, littleCMS etc. Secondly, by visualizing the final GUI on screen, I could see if the draft turned out to be as user-friendly and usable as I intended it to be.

When the dummy GUI was done, I began to add some actual functionality as described in the next paragraph.

The Code

Colormanagement.cpp/Colormanagement.h

The heart of the Color Management Engine for Drylab's Keyframe application is the C++ class ColorManagement that is defined in the two files colormanagement.cpp and colormanagement.h. The ColorManagement class contains the attributes and functions that are needed to enable color management as described above.

Every project is associated to one single ColorManagement instance, that means that a ColorManagement instance gets initialized every time that we load a project. This makes sense because usually the color management settings are intended to be constant within a given project but can change from one project to another. Imagine for example that Project One uses color management based on LUTs whereas Project Two uses color management based on profiles and a third project uses color management based on profiles as well but using a different color space.

After initializing the ColorManagement instance, the compiler calls the function initializeColorManagementWorkflowAttribute (Project *project) to initialize the current workflow settings that are stored for the project. Also it sets the right values for the workflow radio button. The values of these buttons are stored for every project as boolean: One of the three values has to be true whereas the other two have to be false. The workflow option that is connected to the radio button holding the true value represents the currently selected workflow option. This information is then stored in another project attribute called ATTRIBUTE_PROJECT_COLORMANAGEMENTWORKFLOW in the forme of a string depending on the selected workflow as determined before. I decided to create this forth attribute because I wanted to store the information about the workflow in one variable rather than three independent check boxes. Usually, I would have taken radio buttons that combine two or more check boxes in a way that one can easily ask the value of theses boxes combined. Unfortunately, there was no pre-programmed radio button class that I could use. And thus, I decided on the procedure as described above

doColorManagement(Project *project)

After that, the function doColorManagement(Project *project) is called that serves as a transit function to call one of the following functions depending on the workflow setting that is selected:

1. doNoColorManagement(Project *project)
2. doProfilesColorManagement(Project *project)
3. doLUTsColorManagement(Project *project)

When either the profiles or the LUTs option is selected, the application will determine the current screen profiles and working space setting by calling initializeScreenProfilesAttributes() and initializeWorkingSpaceAttribute(project). doColorManagement(Project *project) is called again, whenever entering or leaving the color tab, or whenever different choices are selected on the “Color Lab” tab.

Now, we will take a look on what these functions actually do in detail. We will begin with some helping classes.

initializeColorManagementWorkflowAttribute(Project *project)

This function checks if a color management workflow has been selected, and if not this function sets it on default: no color management at all.

The information of the workflow setting is called in a project attribute called ATTRIBUTE_PROJECT_COLORMANAGEMENTWORKFLOW in form of a string: “lutcm”. “profilecm” or “nocm”.

```
const char * currColorManagementWorkflow;

currColorManagementWorkflow = project-
>getAttributeValue(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
FLOW);
```

When calling the function, the compiler firstly checks, if the attribute holds a value and then it sets the radio button accordingly. That means, it sets the radio button that is connected to the selected option to true and the other two to false. If no value is stored for the COLORMANAGEMENTWORKFLOW attribute, it checks if a value has been saved for one of the radio buttons. If yes, it sets the COLORMANAGEMENTWORKFLOW attribute and the radio buttons accordingly.

```
if (strcmp(“lutcm”, currColorManagementWorkflow) == 0) {

        printf(“Your color management settings are current-
ly based on LUTs.\n”);


        project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM,
true);

        project->setAttribute(ATTRIBUTE_PROJECT_RADIOPRO-
FILESCM, false);

        project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM,
false);

        project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGE-
MENTWORKFLOW, “lutcm”);


 } else if (strcmp(“profilecm”, currColorManagementWorkflow)
== 0) { …
```

If this does not work out either, the compiler sets the COLORMANAGEMENTWORKFLOW and the radio buttons to a default no colormangement workflow.

Also, check the diagram on the next page (see Figure 3-19).

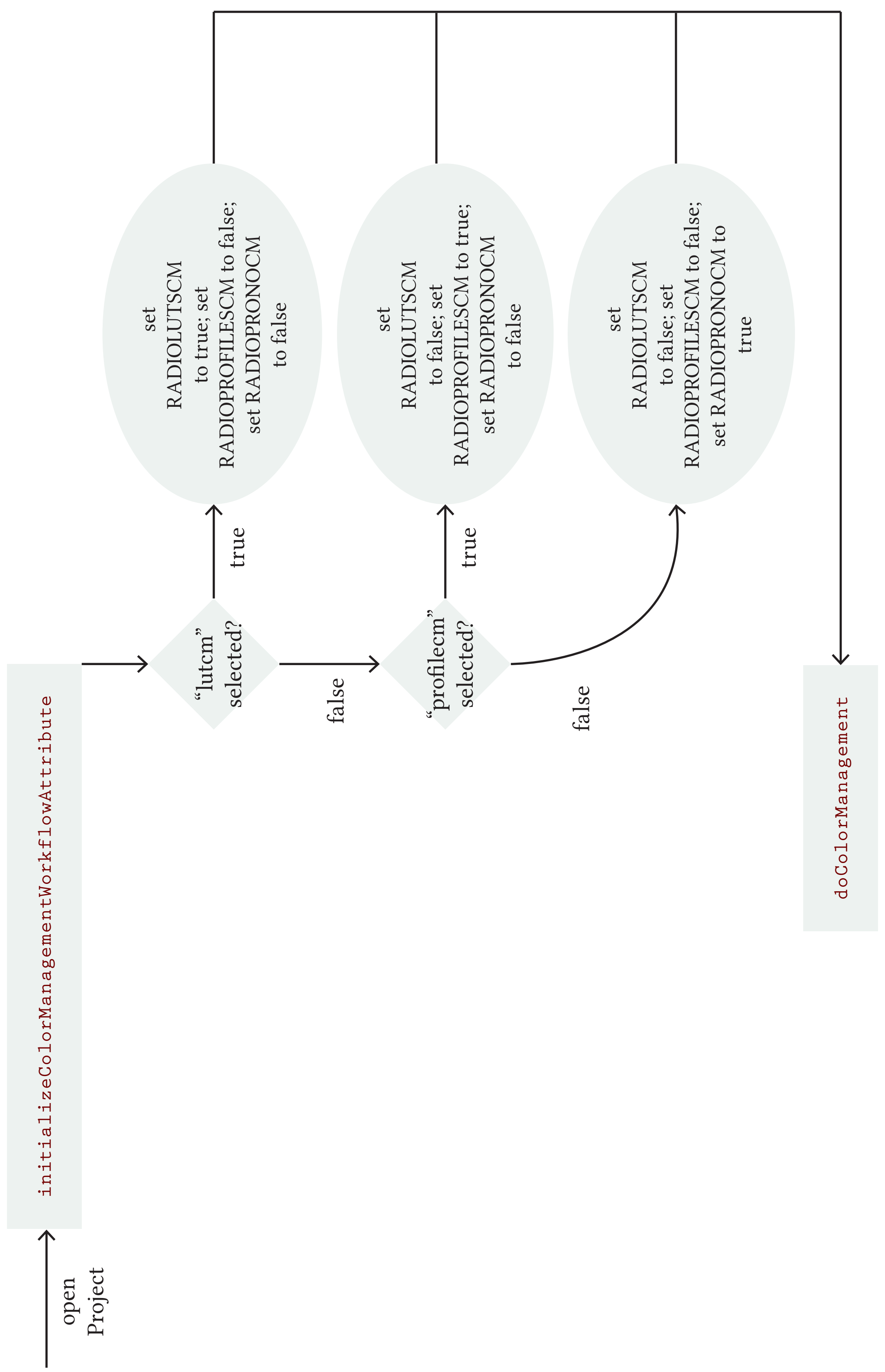

open Project
initializeColorManagementWorkflowAttribute
"lutcm" selected?
true
set RADIOLUTSCM to true; set RADIOPROFILESCM to false; set RADIOPRONOCM to false
false
"profilecm" selected?
true
set RADIOLUTSCM to false; set RADIOPROFILESCM to true; set RADIOPRONOCM to false
false
set RADIOLUTSCM to false; set RADIOPROFILESCM to false; set RADIOPRONOCM to true
doColorManagement

Figure 3-19 Functionality initializeColorManagementWorkflowAttribute

resetGlCanvas(GlCanvas *canvas)

GlCanvas is a Keyframe specific class that represents the canvas on which GlImages are displayed. The GlCanvas holds two GlImage references: The first which is the actual image that is displayed on the screen and a second, which is a lookup reference called "lut" that can be used for color transformations on the actual image. This function resets the "lut"-variable of a GlCanvas to a neutral LUT, which means that no color transformation is applied.

What is important about this function is how the GlCanvas lut is accessed. We get a pointer on the actual LUT of the GlCanvas instance by calling canvas->getLut3D().

```
GlImage *lut = canvas->getLut3D();
```

Now there are two possibilities:

1. The GlCanvas already has a lut. That means we have to deactivate it, reallocate it and initialize it again.

```
        lut->deActivate();
        lut->allocate(32, 32, 3, 32);
        lut->initializeLut();
        lut->gain(1.0);
```

2. The GlCanvas has no lut yet and thus, the lut reference will be NULL. That means we have to create a new GlImage LUT and allocate it to the canvas.

```
        lut = new GlImage(32, 32, 3, COLORIMETRIC_RGB, 32);
        lut->initializeLut();
        lut->gain(1.0);
        canvas->setLut3D(lut);
```

These two possibilities will be important for all three functions. We should always call getLut3D to check if a LUT already exists because of performance reasons.

doNoColorManagement(Project *project)

This function is called in the case that "No color management" selected and its purpose is to reset the LUTs for both main screen and extra screen. In Keyframe RushesControl, there are two preview windows: The first one is part of the grading window and this is the one that is supposed to be always on the main screen. We can access it by calling gradingWindow-

>getPrimaryCanvas(). The second one opens in an extra window when clicking on “External window” in the grading window and it is supposed to be always opened on the extra screen[3]. We can access it by calling gradingWindow->getTertiaryCanvas().

```
GlCanvas *mainScreenCanvas = application->window.
getGradingWindow()->getPrimaryCanvas();
GlCanvas *extraScreenCanvas = application->window.
getGradingWindow()->getTertiaryCanvas();


if (mainScreenCanvas){
 this->resetGlCanvas(mainScreenCanvas);
}
if (extraScreenCanvas){
 this->resetGlCanvas(extraScreenCanvas);
}
```

After we have accessed the GlCanvas for both main and extra screen, we reset both canvas’ luts by calling resetGlCanvas(mainScreenCanvas)/resetGlCanvas(extraScreenCanvas).

initializeScreenProfilesAttributes()

This function checks how many screens are currently connected to the system, also it stores the location of the profiles of both screens for the whole application in the application attributes MAINSCREENPROFILEURL and EXTRASCREENPROFILEURL. This is the first time that we actually use the ColorSync architecture of Mac OS. ColorSync provides a variety of functions to access the screens and to get information about their profiles.

At first the function tries to get a list with all active displays. At this point the number of screens is limited to two. The function CGGetACtiveDisplayList is called, the IDs of the active displays is stored in an array variable called displayID.

```
cgErr = CGGetActiveDisplayList(2, displayID, &numDisplays);
```

After we have successfully initialized the list, the function gets hold on the first element in the display array, which is the ID of the main screen. With the help of ColorSync’s CMGetProfileByAVID we can access the ICC profile of the main screen.

```
OSStatus cmErr = CMGetProfileByAVID(mainScreenID,
&mainProfileRef);
```

3 This definition implies that our CMM only succeeds with reasonable results as long as the main application window is on the main screen and the external window ison the extra screen.

Theoretically, we could now use the ICC profile for our CMM, practically there is another problem. To access the actual screen profile, we need the ColorSync library to compute the gamut transformations we want to use the littleCMS library. Both libraries, however, use different data models to represent profiles and they can hardly be translated from one to each other. This problem could be solved with the following trick: ICC profiles are existing files that are stored somewhere on the file system. The profile classes of both libraries, ColorSync and littleCMS, are based on these very text files, which means both libraries provide functions to create a profile instance based on the file's path. The trick was to get hold on the path to the profile of the ColorSync instance, and open this path later using the littleCMS framework.

```
CMError mErr = CMGetProfileDescriptions(mainProfileRef,
aName, &aCount, NULL, NULL, NULL, NULL);
```

I used the functions CMGetProfileDescriptions and NCMGetProfileLocation to access the name and URL of the profiles and stored it in an application attribute: MAINSCREENPROFILENAME and MAINSCREENPROFILEURL.

```
 application->configuration.setAttribute(ATTRIBUTE_
CONFIGURATION_MAINSCREENPROFILENAME, aName);

application->configuration.setAttribute(ATTRIBUTE_
CONFIGURATION_MAINSCREENPROFILEURL,
mainProfileLocation.u.pathLoc.path);
```

I decided to define this information for the whole application in opposite to for every project because the screen profiles are supposed to stay the same whatever project is opened, and because they can only be changed outside of the application anyway (c. above). After having initialized the main screen attributes, it is now time to look for an extra screen.

The function checks the size of the displayID array. When it's greater or equal two, we know that there are more than one screen connected to the display. The function gets hold on the second element in the display array, which is the ID of the extra screen.

```
if (numDisplays>1) { …

 CGDirectDisplayID extraScreenID = (CMDisplayIDType)
displayID[1];
```

Again: at this point, color management is only enabled for up to two screens. The function accesses the extra screen profile as described for the main screen and stores the profile's information in two application attributes EXTRASCREENPROFILENAME and EXTRASCREENPROFILEURL. What is more I decided to store another application attribute called EXTRASCREENEXISTS that holds a boolean, depending on whether a second screen actually exists or not. This last named attribute is needed for the GUI, because it enables changing the help text for the extra screen element.

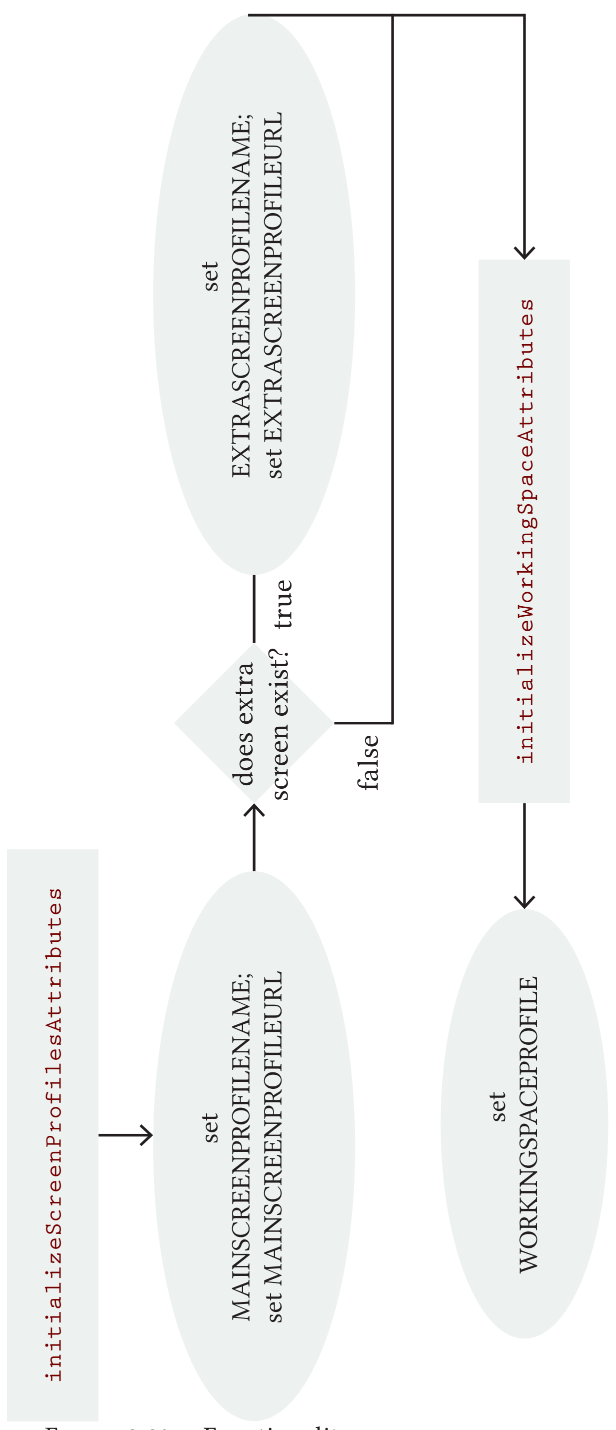

initializeScreenProfilesAttributes
set MAINSCREENPROFILENAME; set MAINSCREENPROFILEURL
does extra screen exist?
true
false
set EXTRASCREENPROFILENAME; set EXTRASCREENPROFILEURL
initializeWorkingSpaceAttributes
set WORKINGSPACEPROFILE

Figure 3-20 Functionality of initializeProfilesAttributes and initializeWorkingspaceAttributes

initializeWorkingSpaceAttribute(Project *project)

This function sets the working space for the project and stores the URL to the profile in a project attribute. Two information related to the working space of a project are stored for each project: WORKINGSPACEPROFILE and WORKINGSPACEPROFILEURL. Both attributes are defined for each project individually because they change from one to another project but stay the same within one specific project. The first attribute holds the name of the working space being used and the second attribute holds the URL to the actual profile file on the file system. Again: storing the URL to the profile makes it possible to open the profile with littleCMS.

```
ColorSpaceMetaData *tempMetaData = workingSpaceProfilePath
Dict[currWorkingSpace];

Path pathWorkingSpaceProfile = tempMetaData-
>getPathToICCProfile();

project->setAttribute(ATTRIBUTE_PROJECT_
WORKINGSPACEPROFILEURL, (const char*)
pathWorkingSpaceProfile.getBuffer());
```

Please check the diagram on the previous page for the previous two functions (see Figure 3-20).

doProfilesColorManagement(Project *project)

This function computes the color transformation and creates a GlImage lut for the main and extra canvas depending on the profiles of the selected working space and the two screens.

To begin with, we access the main and extra screen canvas as described for doNoColorManagement above.

Secondly, we get access to the profile of the current working space. We retrieve the value stored in the WORKINGSPACEPROFILEURL attribute and, if it exists, open the profile by calling the littleCMS function cmsOpenProfileFromFile.

Thirdly, if the profile can be opened, we move on to creating the GlImage LUT of the main screen. We access the value stored in the MAINSCREENPROFILEURL and open it as littleCMS profile instance. Then, we use both the working space and the main screen profile to compute a gamut transformation with the help of littleCMS' cmsCreateTransform function. cmsCreateTransform requires multiple parameters:

- The input color space of the working space profile.
- The output color space of the main screen profile.
- The reproduction intent, I chose a relative colorimetric intent because the transformation is supposed to become as accurate as possible. Still, it should be computed relatively to the medium's white point.

```
cmsHTRANSFORM tempLUT = cmsCreateTransform(
workingSpaceProfile, TYPE_RGB_8, mainScreenProfile,
TYPE_RGB_8, INTENT_RELATIVE_COLORIMETRIC, cmsFLAGS_
GAMUTCHECK);
```

Note: The cmsCreateTransform function does not apply any transformations just yet but it creates a transform object that can later be used to apply the transformation on the actual image.

What is more, we need to convert the littleCMS transformation into a GlImage so that it can be used for our preview screens. Therefore, we access the current lut of the main canvas in the same way as described under doNoColorManagement and on the other hand we create a temporary GlImage LUT with the same size as the current canvas lut. Then, we reset both lookup tables. Why is that? As described before, lookup tables like our GlImage lut are three-dimensional images that can have two functionalities. They can be used as source for color transformations but they can also be used as targets for color transformations. Every node on the lookup table lattice holds an RGB value. These values are neutral now because we reset them before. In other words, a node with the coordination (50/50/50) holds an RGB value of (50/50/50) as well. By applying a color transformation on this neutral LUT, we get a GlImage lookup table that we can use later for color management. More precisely: By applying the littleCMS transformation on a neutral GlImage LUT, we practically "clone" the transformations from the littleCMS instance onto the GlImage instance. In other words, we convert the transformation from one format (littleCMS) to another (GlImage). Remember: We need to do that because we decided to use GlImages as color lut for our previews. There is just one question left: why do we need both the lut connected to the canvas and a temporary lut? That has to do with the structure of the littleCMS function cmsDoTransform that actually applies the color transformation. This function requires the following parameters:

- The littleCms transformation instance.
- An input image, which is represented by the neutral temporary input lookup table.
- An output image, which is represented by our final lookup table that is associated to the canvas. The result of the transformation is stored in this image.

```
cmsDoTransform(tempLUT, tempInput->getBufferU8(),
mainScreenLUT->getBufferU8(), 32*32*32);
```

After the transformation is applied, the lut connected to the main canvas contains the new gamut mapping information. We close both profiles and the transformation because of performance reasons. Now that we have the color LUT for the main screen, it is only the extra screen that is missing.

And once more, the function checks if an extra screen is connected by retrieving the EXTRASCREENPROFILEURL attribute. If the extra screen exists, the function creates a GlImage LUT as described for the main screen.

Finally, whenever the function cannot open a profile, because no working space is registered or the profile of a screen does not exist respectively the extra screen does not exist. The function

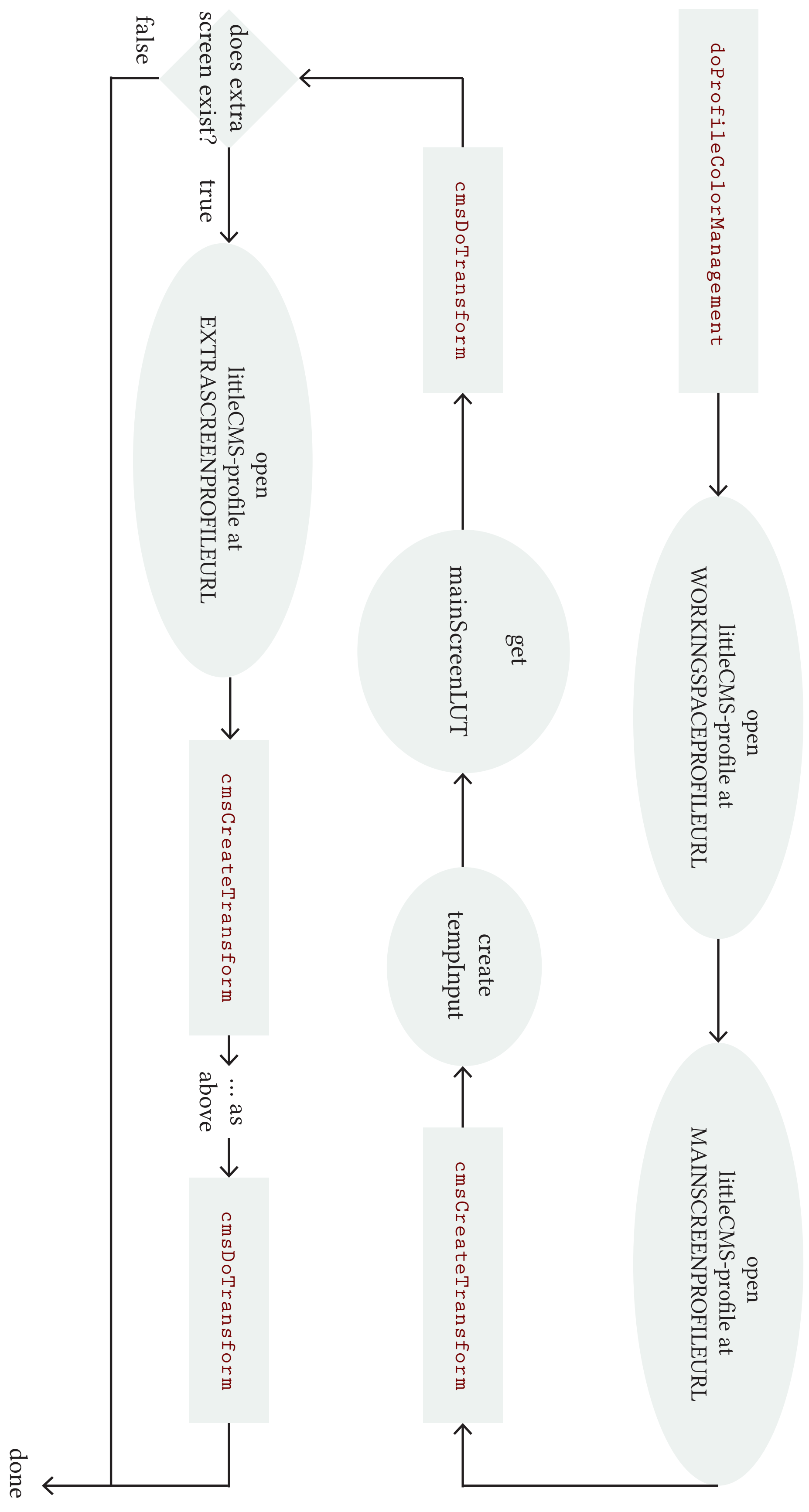


Figure 3-21 Functionality of doProfilesColorManagement

deactivates color management by calling resetGlImage(...) for the respective canvas.

Please, see the diagram on the previous page as well (see Figure 3-21).

vector<RGBPixel> getRGBValuesFromNucodaFile(const char *path)

This function retrieves the RGB values from a Nucoda file and returns them as a vector of RGBPixels. The function is called with a string parameter that holds the path to a Nucoda file. A typical Nucoda file can be seen here.

```
# Lut3D.exe <1D-size> <3D-size> <invert-green> <1d-3d-test>
# see Lut3D.cpp for details on optional arguments
NUCODA_3D_CUBE 2
TITLE “Identity Lut, 3D Only, Version 2 format”
LUT_3D_SIZE 8

0.000000  0.000000  0.000000
0.142857  0.000000  0.000000
0.285714  0.000000  0.000000
…
```

As you can see, the structure is always the same: In the beginning there are some comments, signatures and metadata. At the end, there are lines that hold the RGB values of the nodes. Keep in mind that every one of these lines represents one node on the lattice of the source lookup table. What we have to do is to retrieve these values and save them in a vector so that they can be converted into a GlImage later.

If the path to the file is not empty and the file can actually be opened, the function goes through the following steps:

```
      const char * tempString = “NUCODA_3D_CUBE 2”;
      isNucodaFile = (int)line.find(tempString) != -1;
```

Firstly, it finds out whether the uploaded file is a NUCODA LUT or not. Therefore, it parses the file for a signature that looks like this: “NUCODA_3D_CUBE 2”. This signature proofs that the file is a valid Nucoda file. I actually sourced this functionality out to another function called isNucodaFile(const char *path).

If the signature exists, the function finds out the depth of the NUCODA LUT. This, it parses the function for the following line “LUT_3D_SIZE XXX” with XXX being an integer that represents the depth of the file. The depth represents the steps that are used for the quantization for each channel. A depth of 8 for example means that on each axis there are seven (eight minus one) subdivisions from zero to the maximal RGB value resulting in 512 nodes (eight times eight times eight). This functionality has already been outsourced to a function called

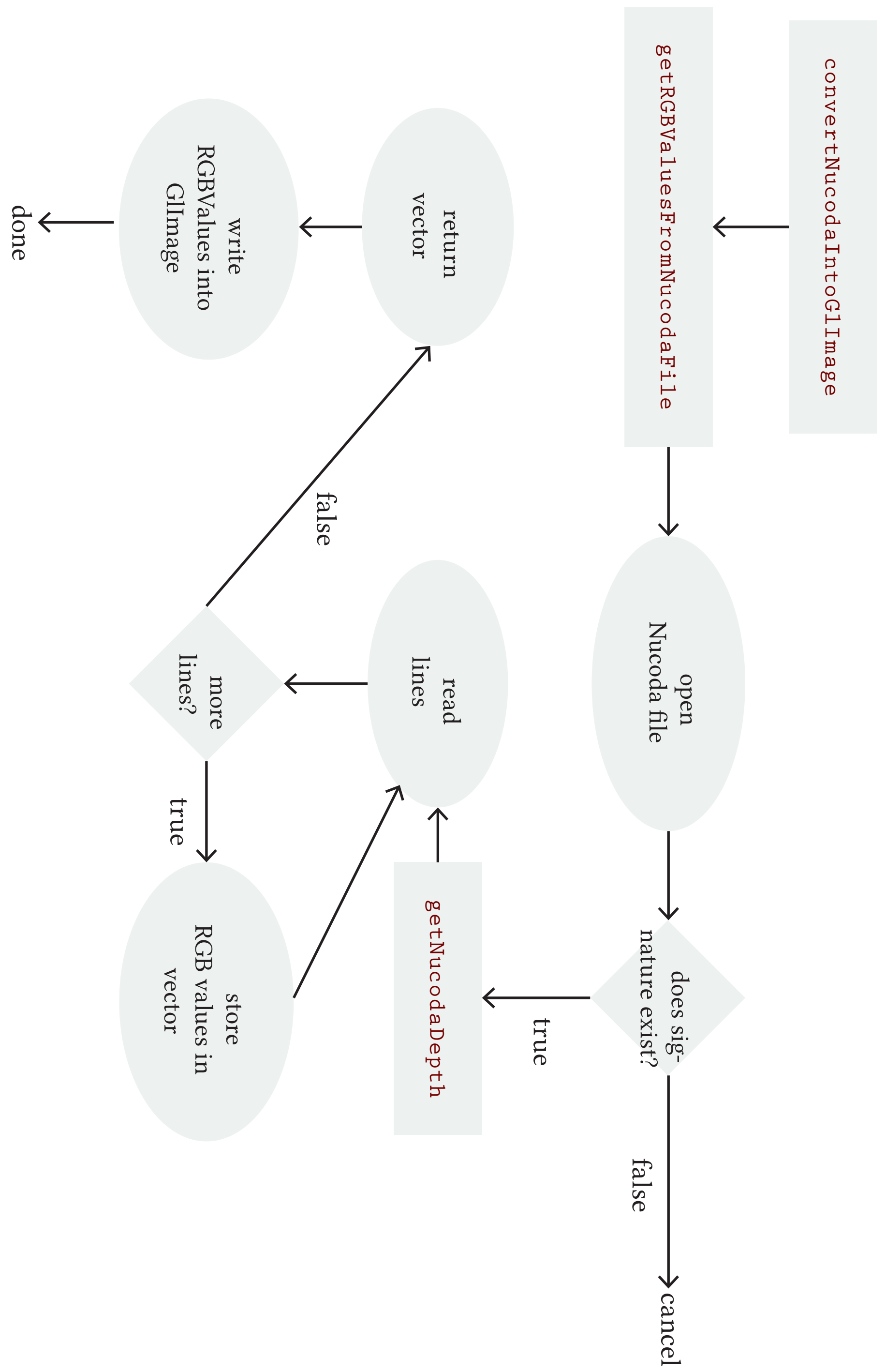


Figure 3-22 Functionaity of convertNucodaIntoGlImage and getRGBValuesFromNucodaFile

getNucodaDepth(const char *path).

```
isRGBValue = (“%d\n”, sscanf(line.c_str(), “%f %f %f\n”,
&p.r, &p.g, &p.b)) == 3;
```

Then, the function will parse each line, one-by-one compared to a regular expression string. The regular expression holds three placeholders for floats divided by white space. That is exactly the form of a line that holds the RGB data of the in the NUCODA file: three floats one for each channel. When the line matches the regular expression, the function creates an RGBPixel for this line and saves the float values in the RGBPixel's respective r, g and b co-ordinates. After that, the function adds the RGBPixel to the end of the vector using push_back(...).

```
if (isRGBValue){ rgbValues.push_back(p); }
```

Finally, the function returns the vector that stores the <size to the three> items.
The RGBPixel values in the vector can eventually be used to initialize our GlImage LUT.

Please, see the diagram on the previous page as well (see Figure 3-22).

convertNucodaIntoGlImage(const char *path, GlImage *lut, int depth)

This function converts an input Nucoda file as retrieved by path into an GlImage instance. The GlImage information ist stored in the lut variable. Also, the function requires the depth of the Nucoda file as input data.

A big part of the work has already been handled by the previous function called getRGBValuesFromNucodaFile(...).

To begin with, the function calls getRGBValuesFromNucodaFile() and stores the returned RGBPixel vector in a local variable.

Then, it compares the actual size of this vector to the assumed size, which is equal to depth to the power of three (s. previous paragraph when I talked about the depth variable). If the vector holds the right – that means the expected – amount of values, the function proceeds; otherwise it returns.

```
if((int)rgbValues.size()==depth*depth*depth)
```

After that, it loops through the values of the GlImage instance and copies the desired RGB values from the RGBPixel vector into the GlImage file. The values in the Nucoda file are stored in the following way: The rows from first row to the <number of depth>th row represent the RGB values where red and green are set to zero but blue increases by one. The next <number of depth> rows represent the RGB values where red is still set to zero but green is increased by one and blue increases by one. For the next <number of depth> rows green is again increased by one and blue increases by one. This is repeated until green reaches a value of <number of depth>-1. From this point on, the red value is increased by one, green is set back to zero and blue gain increases by one. After <number of depth> rows, red is still set on 1, green is increased by one and blue increases by one. This is continued, until all three values are set to <number of depth>-1 and the loop terminates.

Inside the loop, the RGB values from the current RGBPixel are stored inside the current GlImage co-ordinate. And when the loop moves on the current RGBPixel is dropped for the next RGBPixel in the vector.

```
int x=0;
for(int b=0;b<depth; b++){
 for (int g=0;g<depth;g++){
  for (int r=0;r<depth;r++) {
   lut->setSample(lut->getSampleIndex(r, g, 1, b),
rgbValues[x].g);
   lut->setSample(lut->getSampleIndex(r, g, 1, b),
rgbValues[x].g);
   lut->setSample(lut->getSampleIndex(r, g, 2, b),
rgbValues[x].b);
   x++; } } }
```

Finally, the function returns a boolean that is either true or false. True if everything goes alright. Thus, the lut variable holds the information previously only stored inside the Nucoda file. False if there was a problem with the conversion of the values. In this case, the lut can be reset afterwards.

doLUTsColorManagement(Project *project)

This function converts a chosen Nucoda file into a GlImage that is eventually used for color management.

To begin with, the function gets access to the main canvas and the Nucoda file that is selected for color management.

Secondly, it checks, whether the selected file is a Nucoda file or not by parsing through the file for the Nucoda signature as described for the convertNucodaIntoGlImage(...) function.

If yes, it will then retrieve the depth size of the Nucoda lookup table, because this data has to be handed down to the convertNucodaIntoGlImage(...) function in order to check, if the file has enough values.

```
extraSuccess = convertNucodaIntoGlImage(
extraLUTNucodaPath, extraScreenLut, extraDepth);
```

Furthermore, the program gets hold of the lut that is registered for the main canvas and converts the information in the Nucoda file into the main canvas' lut via convertNucodaIntoGlImage(...).

If anything of the above throws any errors, the lut for the main screen will be reset via resetGlCanvas(...).

What is more, the previous steps are repeated for the extra screen canvas.

This feature threw a lot of errors at the beginning. The main problem was that the tertiary canvas was not initialized before clicking on the "External window" button. However, the function used to apply color management for this canvas whenever doColorManagement(...) was called – regardless whether the canvas existed or not. Thus, the program crashed when the canvas was not actually initialized. To avoid this problem I added a simple if-clause to check if the external screen was opened at all, and if the tertiary canvas existed.

Screen Flow Video

This was an overview of the most important classes involved. Of course there were many more, especially those concerning the GUI that I will not discuss here. To sum things up, you can now take a look on the screenflow of the final feature that I programed for the last weeks:

http://tinyurl.com/6hkn4w6

To choose the new color management settings, go to "Configuration>Color Lab". There, you will find three radio button options: "No Color Management", "Profiles" and "LUTs".
You select "No Color Management", nothing is going to happen.

You select "Profiles", will you see more options. You will for example have to choose the color working space you are currently working with. What's more, the application gets the profiles automatically. If you want to choose other profiles, you have to go to "System Preferences > Displays > Color".

You select "LUTs", you will get more options as well. Now, you have to upload Nucoda luts.

Finally, if you only have got one screen. The color tab is only showing one screen. In the following video, I disconnected the extra screen while the program was still open:

http://tinyurl.com/44gjfmq

Discussion

During the first week, I experimented with the different libraries and got used to the source code and its structure. I drew the icons that were needed and programed the GUI so that every element could be seen on the screen. In the beginning, it was a bit tricky to get used to the GUI framework but in the end I managed the functions pretty well. In the beginning, I found it very tough that many elements I knew from other programming languages did not exist or did not work properly with the fltk framework. For example, there was no user-friendly radio button class. Furthermore, I did not like the way attributes were stored for the project respectively the whole application. Being most experienced with web applications, I was used to store complex objects by their primary key in a database. For this project, I could only store the basic types like strings, booleans and integers. Thus, the complicated solution for the radio menu with three plus one attributes being stored for a project. It would have been nicer to store one actual profile instance rather than just the URL to the file for example.

From the second week on, I began to work with the doXXXColorManagement(...) functions and continued adding functionality to the dummy GUI. This working continued throughout the third week as well. The hardest task during this step was to communicate between the ColorSync library and the littleCMS library. Both libraries use basically the same files but they access them by different library-specific class representations. This fact made it necessary to think about how to convert from one framework to the other. Eventually, I came up with the solutions as described above.

During the fourth and last week, I finally focused on GUI details and minor bugs within the code. In this phase, it was all about timing. The functions were working but the compiler was throwing errors. This was due to the fact, that the functions called instances that have not been initialized yet. At this point, I had to include several if-clauses checking if extra screen exists for example and initialize variables that have not been initialized before. Moreover, I had to adjust the incidents when the doColorManagement(...) function was actually called. At first, for example, the lut did not change when a new working space was selected. Thus, I had to add a call for doColorManagement(...) in the callback function for the drop box widget.

Finally, there might be some possibilities to clean up the code. Especially, the help texts for example should not be stored in the ColorManagement class but centrally for the whole application.

To sum up, we have now looked on the technical details of the color management module and the most important classes and functions in the source code. We will now proceed in testing if and how accurate the module works in our last part C about evaluation.

PART C

Evaluation of the Color Management Module

In the previous part, we looked at the technical implementation of our color management module (CMM) for Drylab's Keyframe application. Now, there are two interesting questions left:

Can we visualize the improvement that was enabled by our CMM? And if yes, how can we do that?

In part A, we stated that the main purpose of color management is to solve the color reproduction problem. The color reproduction problem is the problem that any given input data is displayed differently on different devices. An optimal color management neutralizes these differences completely; an average color management reduces these differences. Luckily for us, these differences can be measured using colorimetry and the color difference formula described in part A. And here is what we are going to do: We will measure the CIELAB output values of some color samples that are displayed in Keyframe and that are shown on two screens that are connected to the system – both with and without color management feature activated. Then, we will calculate the color difference between those batches.

Experiment design and evaluation

For the experiment, I used a Mac with MAC OS X 10.6.4.

I had three screens: Two DELL LCD screens and one Phillips CRT. I kept one of the LCD screens fixed as main screen, and swapped between the remaining CRT and LCD screens for the various series of experimentation to investigate the different behavior of CRT and LCD screens.

Furthermore, I used an Gretag-MacBeth eye-one display 2 colorimeter for the calibration and measuring jobs. The profiles were created with the help of the eye-one match software and the measurements were taken with the MeasureTool.

Throughout the evaluation process, I used various different color patches to compare the output on main and extra screen. In total, I was testing on seven days (E1-E7) and I made subtle changes for each experiment. The basic procedure, however, was the same for all experiments. I want to explain this basic procedure using the setup of the final, seventh experiment on April, 13th 2011 (E7). In the discussion afterwards, I will point out some of the variations I made during the other experiments, and I will analyze possible sources of error.

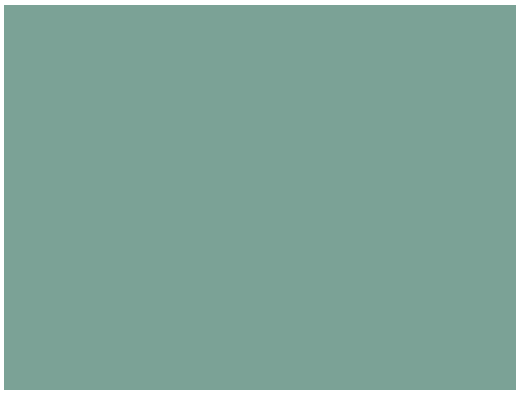


FIGURE 4-23 #1: sRGB (123/162/150)

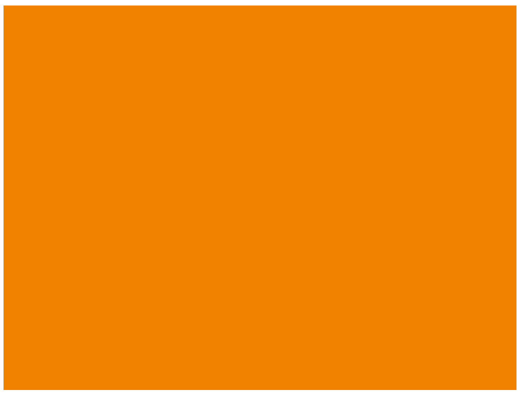


FIGURE 4-24 #2: sRGB (240/130/0)

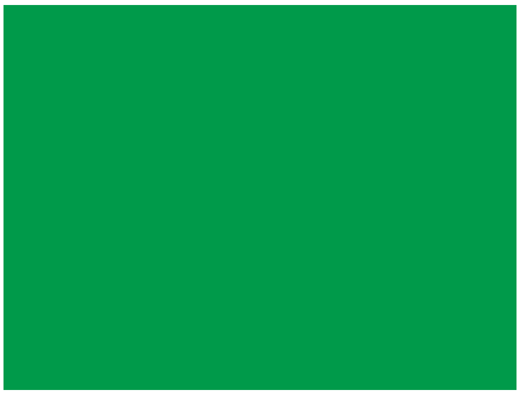


FIGURE 4-26 #3: sRGB (0/154/74)

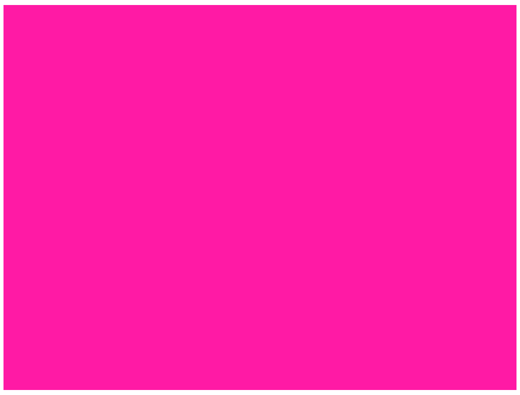


FIGURE 4-25 #4: sRGB (255/26/165)

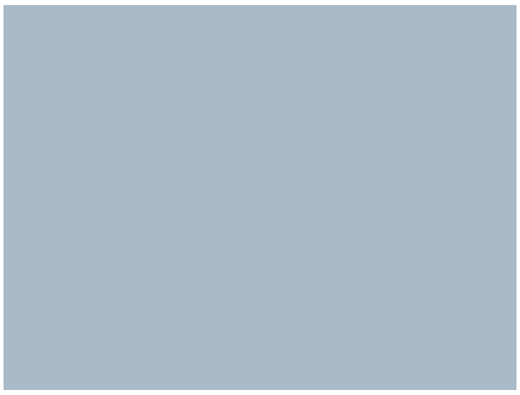


FIGURE 4-27 #5: sRGB (168/187/198)

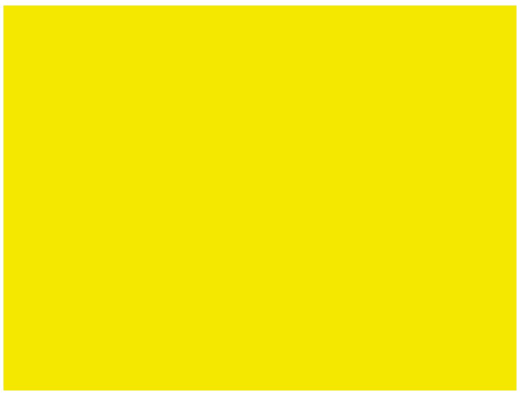


FIGURE 4-28 #6: sRGB (244/232/0)

FIGURE 4-30 #7: sRGB (255/255/255)

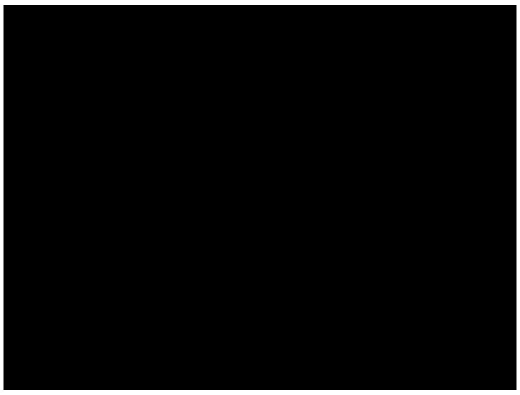


FIGURE 4-29 #8: sRGB (0/0/0)

1. Design of the color patches

I made eight random color patches in Adobe Photoshop (see Figure 4-23 – Figure 4-29 on the next page). The working color space of Photoshop was set to sRGB and every color patch was assigned with the sRGB profile. I used random color patches because I wanted to test if the CMM would give reasonable results for any color within the color space. For my first experiments, I used 32 color patches whereof three times ten were gradients from black to each of the primaries red, green and blue plus black and white. That was because the primaries are the most important colors responsible for the creation of any other color, and black and white because I wanted to see the performance when the screen reaches its min respectively max performance. I dropped these gradient patches, however, because these patches are most likely to be used by the profiling software for the creation of the profiles. This means that they would most likely result in good results because they are directly translated from source to target color space using the profile's LUT. In contrast to that, I wanted to check if the CMM's transformation works as good for values that had to be interpolated. That is way I switched to the randomly chosen color patches.

What is more, I chose color patches that lie within the color space i.e. I tried not to use any patches that have one of the three RGB channels to the max. The screens do have different primaries coordinates. As described under part A, a screen is only capable of reproducing colors that lie within the triangle formed by the primaries in the CIEXYZ chromaticity diagram. Colors that have a maximum value of either one of the channels' primaries often lie directly on the surface of the screens' gamuts. Then, if the primaries are just slightly different, it is the surface of the screens' gamuts that differ the most when one screen's gamut is compared to another screen's gamut. Thus, a color that lies on the surface of the gamut is most likely to not be mapped correctly from one color space to another. In other words, the color difference of a color that lies on the surface of a gamut is naturally expected to be above average. Thus, I decided later not to use surface colors because such colors give by design a big color difference.

Secondly, I used only eight color patches later because the measuring of the patches was rather time-consuming: I had to open every color patch manually in Keyframe, measure the colorimetric value by hand with MeasureTool and write the results manually in a Google spreadsheet. All of this had to be done twice for each screen, before and after color management. For the 32 color patches in the beginning, an experiment took about 90 minutes. By contrast to that, I was unsure about my experimental design and most of all about the results I got for the first measurements. That is way I wanted to do more series of measurements within the time being given, which was impossible when one measurement series took 90 minutes. Thus, I concentrated on the few but more accurate experimental design with eight patches[1].

Finally, I included the white color patch because I needed the XYZ values of this reference later to convert XYZ values into LAB values.

1 As it turned out later, I had problems with the XYZ/LAB values I got from the Measure Tool in the beginning. This will investigated later under discussion.

2. Monitor calibration

I calibrated each screen independently with the help of the Eye-one Match software. As white point, I chose D65, which correlates to a color temperature of 6500K. The gamma was set to 2.2, which is the default for Mac OS. Finally, I set the luminance as following:

- When comparing the LCD to the CRT screen, I set the luminance level to 80 candela/m^2. Most CRT screen are not capable of producing an equally high luminance level as LCD screens. Thus, to be completely certain, I set the luminance level to 80 because I wanted to be sure that both screens could reproduce the same luminance level.
- When comparing the both LCD screens, I set the luminance level to 120. On the one hand for it was recommended by the software. On the other hand because I was sure both were capable of reproducing the same luminance level. After all they were of the same type and brand.

3. Profiling

At the same time of the calibration, the Eye-one Match software creates a profile. This profile is handed to the operative system which sets the profile as currently chosen profile. Of course, the profile can be changed manually by choosing a different profile under “System Preferences > Displays > Color”. It is important to mention that the profiles can be chosen and changed for both screens independently. When choosing the “Settings” menu a box opens on each screen and a profile can be chosen for either one. Mac OS X uses either profile independently for the calibration of each screen. In other words, to compute the calibration function for the main screen for instance, it extracts the information stored in the “vcgt”-tag of the profile chosen in the dialog of the main screen. Respectively to compute the calibration function for the extra screen Mac OS X extract the information stored in the “vcgt”-tag of the profile chosen in the dialog of the extra screen.

After this, I wanted to take a look at the profiles that I just had created. ColorSync Utility is a software that allows access to the data of an ICC profile. As we can see in the pictures, we can look at the data stored in the different tags as for example the white point of the profile (wtpt-tag), the XYZ coordinates of the primaries (rXYZ-, gXYZ, bXYZ-tag in the pictures on the next page: Figure 4-35, Figure 4-36), the calibration curve (vcgt-tag), the tone reproduction curves of each channel (rTRC-, gTRC-, bTRC-tag) etc.

The ColorSync Utility provides furthermore a possibility to present a 3-D presentation of the screen’s gamut as stored in the profile. With this feature, we have the opportunity to compare two gamuts with each other. In the pictures on the next page (see Figure 4-31, Figure 4-32), we can see the gamut of the main LCD screen as a white net cube and the gamut of the extra LCD screen as inner cube that is solid and colored. As we can see here, the two gamuts are very similar due two the sameness of type and brand.

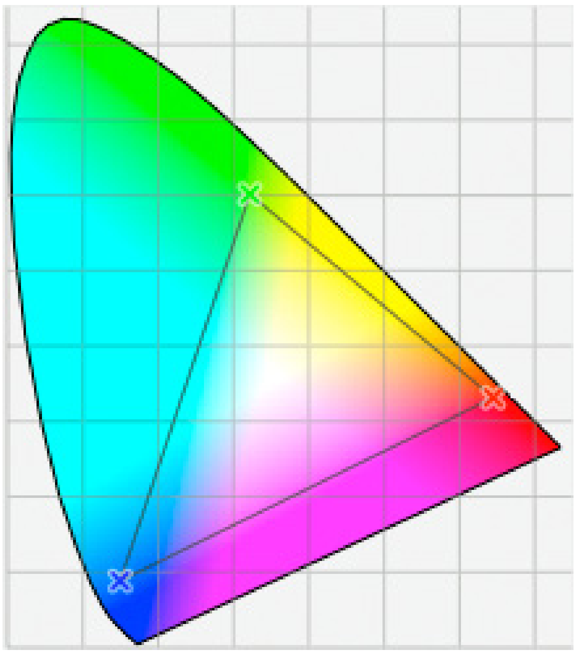


Figure 4-35 Primary coordinates of the main LCD screen

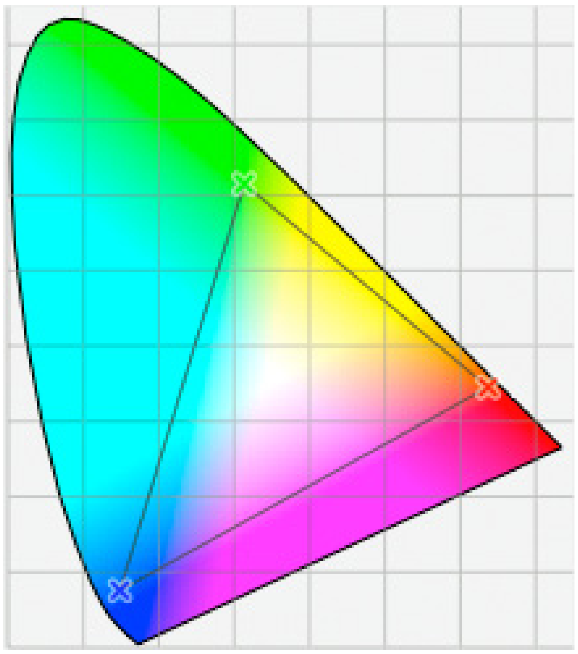


Figure 4-36 Primary coordinates of the extra CRT screen

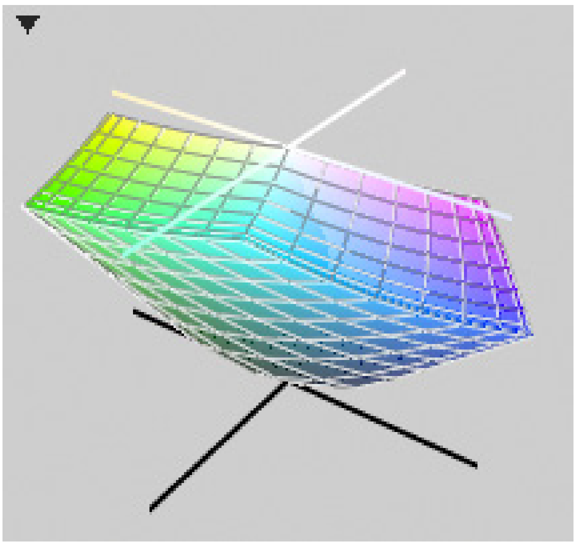


Figure 4-31 Gamut of the main LCD screen compared to the extra LCD screen gamut (1)

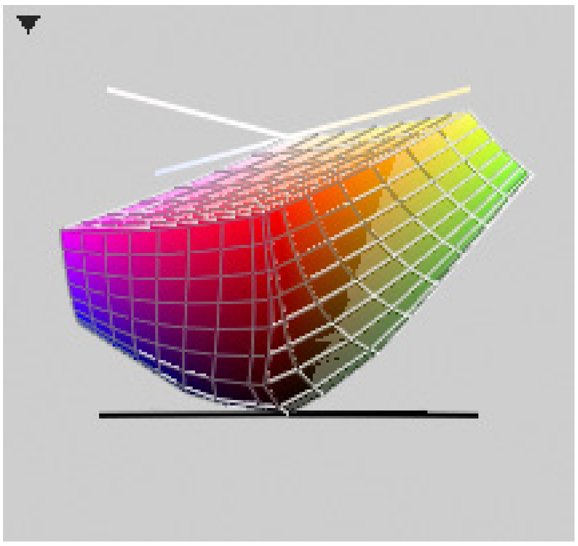


Figure 4-32 Gamut of the main LCD screen compared to the extra LCD screen gamut (2)

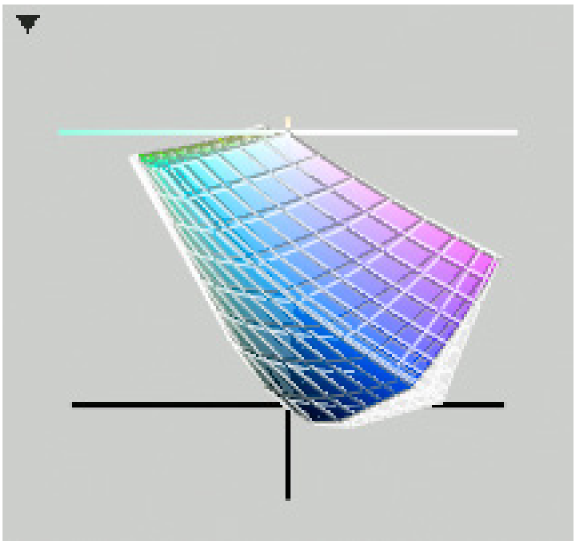


Figure 4-33 Gamut of the main LCD screen compared to the extra CRT screen gamut (1)

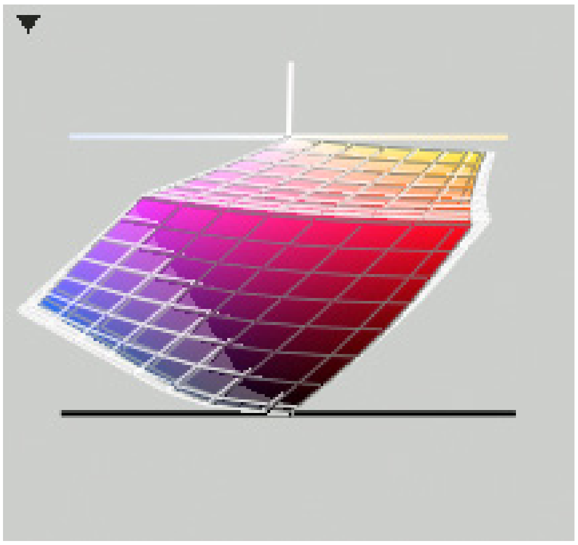


Figure 4-34 Gamut of the main LCD screen compared to the extra CRT screen gamut (2)

In contrast, the gamut of the LCD screen as solid and colorful cube compared to the gamut of the CRT as white net cube in the pictures below (see Figure 4-33, Figure 4-34). We can see that for example the CRT's gamut is a bit bigger than the LCD's gamut (except for colors in the red areas).

4. Measuring

I measured the tristimulus values of the color patches on both screens before and after color management in Keyframe.
At first, I created a new Keyframe project called "Evaluation", in which I imported the 8 sample targets.
Then, I chose "No Color Management" under settings in Keyframe. Moreover, I opened the MeasureTool software, chose "Spot Measuring" in the menu on the left and set the values to XYZ.
Furthermore, I put the colorimeter in the middle of the main screen. One by one, I opened each picture of my color patches in the grading window so that I could see the color patch in the preview window of the grading window on the main screen. I measured the XYZ value of the displayed color patch and copied these values into a Google spreadsheet. Note: I always kept the colorimeter on the same spot of the screen. Actually I never moved it until I moved on to measuring the patches on the extra screen.
After that I was finished with the measurement of all 8 color patches, I chose the "Profiles" option under settings in Keyframe.
Eventually, I repeated the whole procedure for all 8 color patches again – but this time the color management feature was turned on.

Secondly, I placed the colorimeter in the middle of the extra screen. I put the color management settings back to "No Color Management".
One by one I opened each picture of my color patches in the grading window and I clicked on "External window" to open the picture in an extra window.
Then, tracked this window to the spot where I placed the colorimeter on the extra screen. I measured the XYZ value with the MeasureTool software and copied the values into my Google spreadsheet.
The same procedure was repeated for all 8 color patches again with the color management settings set to "Profiles".

To sum up, we have now the XYZ values of eight color patches for both screens, before and after color management which resulted in four times eight, 32 XYZ triples.

5. Evaluation

I calculated the ΔE for both screens before and after color management, compared the screen both with each other and with the original input data.
To calculate the ΔE, it is necessary to convert the XYZ data into LAB first. For this purpose, I used the conversion function with the help of a Excel document (c. Lindbloom's v: XYZ to Lab) that can be seen below:

$$L = 116 f_y - 16 \text{ , where } \quad f_x = \begin{cases} \sqrt[3]{x_r} & x_r > \varepsilon \\ \dfrac{\kappa x_r + 16}{116} & x_r \le \varepsilon \end{cases} \quad \text{, with } \quad x_r = \frac{X}{X_r}$$

$$a = 500\left(f_x - f_y\right)$$

$$b = 200\left(f_y - f_z\right)$$

$$f_y = \begin{cases} \sqrt[3]{y_r} & y_r > \varepsilon \\ \dfrac{\kappa y_r + 16}{116} & y_r \le \varepsilon \end{cases} \qquad y_r = \frac{Y}{Y_r}$$

$$z_r = \frac{Z}{Z_r}$$

$$\varepsilon = \begin{cases} 0.008856 \\ 216/24389 \end{cases}$$

$$f_z = \begin{cases} \sqrt[3]{z_r} & z_r > \varepsilon \\ \dfrac{\kappa z_r + 16}{116} & z_r \le \varepsilon \end{cases} \qquad \kappa = \begin{cases} 903.3 \\ 24389/27 \end{cases}$$

Figure 4-37 Formula to convert from CIEXYZ to CIELAB

As we can see in the function. It is necessary to declare a reference white with the XYZ coordinates X_r, Y_r and Z_r to normalize the values. For this reference white, I used the XYZ values of the white color patch that I measured for every data set[2]. Moreover, I had to convert the original input RGB values into LAB values. This time, I relied on the values that Adobe Photoshop provided me.

2 This reference white was a source of big confusion in the beginning as I will point out later in the discussion.

With the LAB values at hand we can easily calculate the ΔE with the formula as described under part A. Now, it was time to compare the LAB values. I compared the values in two ways:

- At first, I compared the values of each screen independently with the original input LAB values.

E7	Target L	Target a	Target b	before $\longrightarrow$	Main L	Main a	Main b	ΔE main screen	after $\longrightarrow$	Main L	Main a	Main b	ΔE main screen
#1	63	-16	2		63,8	-19,4	4,72	4,388724188		62,8	-16,2	1,98	0,333616546
#2	66	39	73		66,3	37,5	70	3,363970868		65,3	36,7	69,1	4,572067366
#3	57	84	-14		57,8	85	-9,87	4,32919161		56,9	85,8	-10,4	4,008840231
#4	56	-50	32		54,5	-54,7	32,2	4,896917398		54,5	-53,4	30,1	4,189582318
#5	75	-5	-8		75,3	-7,68	-4	4,824147593		74,6	-5,55	-5,31	2,780683369
#6	91	-10	88		90	-12,2	88,1	2,429176815		90,9	-15,1	87,9	5,10331265
#7	100	0	0		100	0	0	0		100	0	0	0
#8	0	0	0		1,1	0,18	-1,43	1,813091283		1,11	0,18	-0,6	1,274558747

FIGURE 4-38 Target and sample Lab values measured on the main LCD screen before and after color management in Keyframe

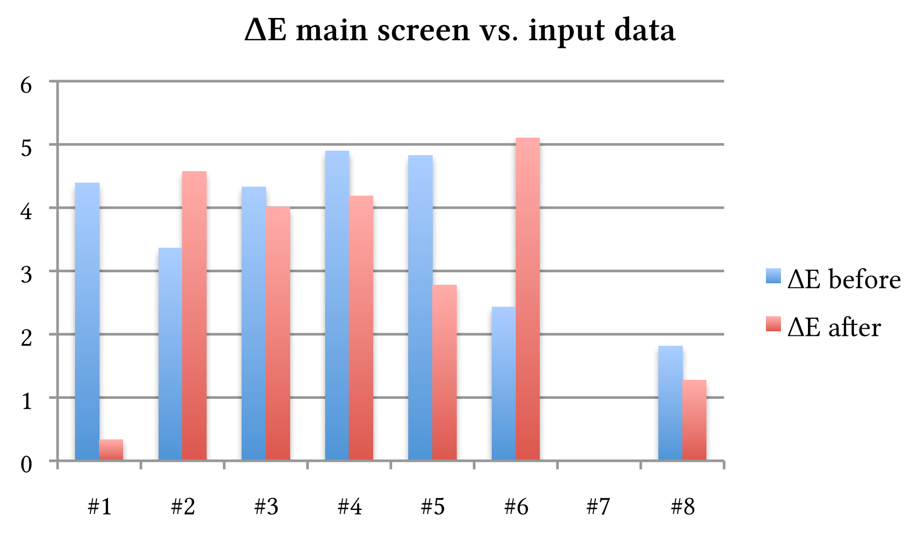


FIGURE 4-39 Comparison of ΔE between target and sample Lab values measured on the main LCD screen before and after color management in Keyframe

E7	Target L	Target a	Target b	before ⟶	Extra L	Extra a	Extra b	ΔE extra screen	after ⟶	Extra L	Extra a	Extra b	ΔE extra screen
#1	63	-16	2		61,4	-15,9	6,36	4,651924333		63,2	-19,7	7,33	6,513478333
#2	66	39	73		63,4	48	66,6	11,33328725		62,7	40,5	63,7	9,9603263
#3	57	84	-14		57,7	83,3	-9,42	4,679893161		57,2	78	-10,7	6,886174555
#4	56	-50	32		53,8	-50,4	37,9	6,332803487		55,6	-50,7	37,7	5,748556341
#5	75	-5	-8		72,6	-0,85	-6,12	5,140165367		73,8	-4,8	-5,95	2,393929824
#6	91	-10	88		88,8	-6,2	87,6	4,394189345		89,5	-12,2	83,7	5,080442894
#7	100	0	0		100	0	0	0		100	0	0	0
#8	0	0	0		3,15	1,23	0,25	3,390855349		3,29	0,69	0,24	3,370133529

Figure 4-41 Target and sample Lab values measured on the extra CRT screen before and after color management in Keyframe

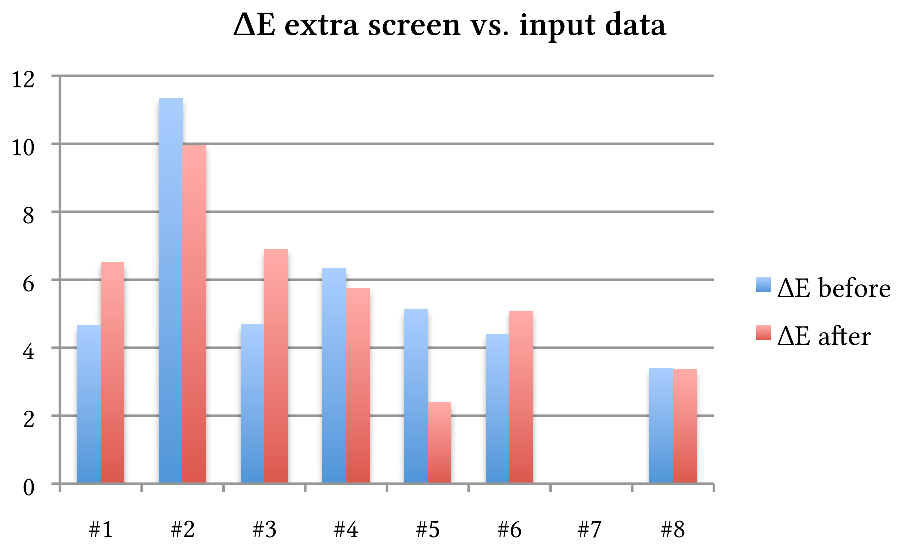


Figure 4-40 Comparison of ΔE between target and sample Lab values measured on the extra CRT screen before and after color management in Keyframe

- Secondly, I calculated the difference between the values on each screen directly regardless of the actual original data.

E7	Target L	Target a	Target b	before →	Main L	Main a	Main b	Extra L	Extra a	Extra b	ΔE before main vs. Extra
#1	63	-16	2		63,8	-19,4	4,72	61,4	-15,9	6,36	4,506761587
#2	66	39	73		66,3	37,5	70	63,4	48	66,6	11,41140219
#3	57	84	-14		57,8	85	-9,87	57,7	83,3	-9,42	1,781937148
#4	56	-50	32		54,5	-54,7	32,2	53,8	-50,4	37,9	7,185193108
#5	75	-5	-8		75,3	-7,68	-4	72,6	-0,85	-6,12	7,637126423
#6	91	-10	88		90	-12,2	88,1	88,8	-6,2	87,6	6,1689059
#7	100	0	0		100	0	0	100	0	0	0
#8	0	0	0		1,1	0,18	-1,43	3,15	1,23	0,25	2,85085952

FIGURE 4-42 Lab values on main LCD and extra CRT screen before management in Keyframe

E7	Target L	Target a	Target b	after →	Main L	Main a	Main b	Extra L	Extra a	Extra b	ΔE after main vs. Extra
#1	63	-16	2		62,8	-16,2	1,98	63,2	-19,7	7,33	6,417889061
#2	66	39	73		65,3	36,7	69,1	62,7	40,5	63,7	7,12659105
#3	57	84	-14		56,9	85,8	-10,4	57,2	78	-10,7	7,840446416
#4	56	-50	32		54,5	-53,4	30,1	55,6	-50,7	37,7	8,161452077
#5	75	-5	-8		74,6	-5,55	-5,31	73,8	-4,8	-5,95	1,257179383
#6	91	-10	88		90,9	-15,1	87,9	89,5	-12,2	83,7	5,263088447
#7	100	0	0		100	0	0	100	0	0	0
#8	0	0	0		1,11	0,18	-0,6	3,29	0,69	0,24	2,391254901

FIGURE 4-43 Lab values on main LCD and extra CRT screen after management in Keyframe

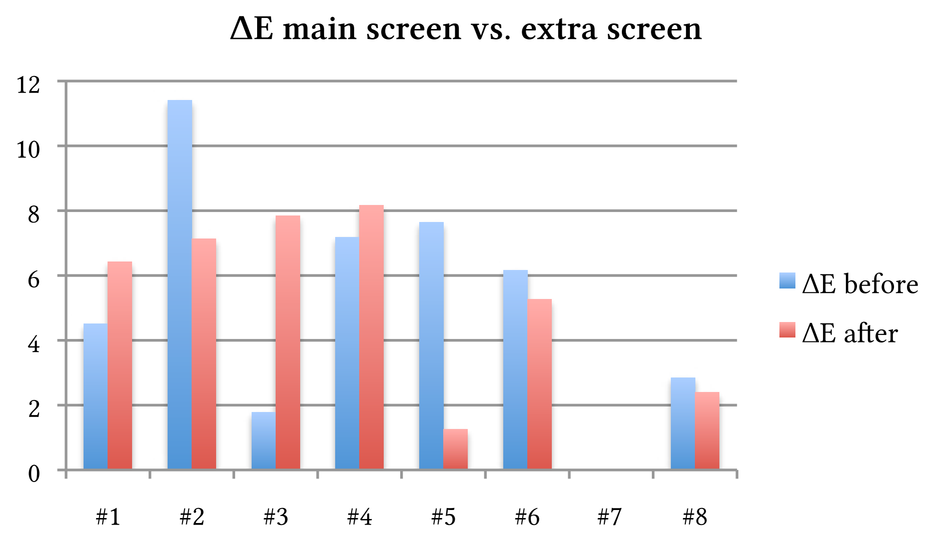


FIGURE 4-44 Comparison of ΔE between main LCD and extra CRT screen before and after color management in Keyframe

Discussion

Interpretation of the experimental design

As we can see, the direct comparison with and without color management shows very different results. For some patches, the color reproduction problem seems to have decreased (Patch #4 and #5) whereas for other it seems to have increased (Patch #1 and #6). Thus, I made a simple t-test analysis for my results as described by the pharmaceutical institute of the University of Oslo (c. Farmasøytisk institutt, universitetet i Oslo). My null hypothesis was as following:

H0 = There is no difference for our average ΔE before and after color management that results from the experiment designed as described above.

For the calculations, I used GraphPad's t-test online calculator. I set up a paired test and typed in the ΔE between the main screen data and original data before and after color management. Then, I repeated this procedure for the ΔE between the extra screen data and input data, and finally for the ΔE between the main screen and the extra screen data, both before and after color management. The website gave the following p-values for the investigated tests:

	ΔE main screen vs. input data	ΔE extra screen vs. input data	ΔE main screen vs. extra screen
P value	0,5268	0,9933	0,7811

Figure 4-45 P values for the test as described above

As we can see, the p-values are very much higher than the statistical significance level of 0.05. This means that our results are not statistically significant. In other words, the differences between the results occurred more likely due to chance than due to our color management engine (c. Farmasøytisk institutt, universitetet i Oslo).

Does this mean that our CMM does not work at all? No, not necessarily. I just means that with the recent experiment design, we cannot make a profound decision on whether the CMM works or not. The results of the t-test point out that the results before and after color management are basically the same. In other words, the test showed that our experiment did not prove that our results were better after color management than before. Simultaneously however, the test showed that the experiment did not prove that our results were much worse after color management neither. If the CMM would work faultily – if the CMM would apply double color transformation, one by the Keyframe CMM and one by the Mac OS X CMM, for example –, the results would be much worse after color management than before. At least, we can deduce that our CMM is not completely wrong.

The question, whether our CMM is an improvement or not, cannot be answered with our current experiment design. The setup was two rough to measure the subtle changes in ΔE before and after color management. In the following discussion, I point out various factors that are responsible for the uncertain experiment results.

At first, there is the uncertainty of the measuring instrument. I tested the repeatability of the instrument on the CRT screen during experiment 6 (E6) and on the LCD screen during experiment 5 (E5).

For the measurement on the CRT screen, I measured the dark green patch (#4) ten times in a row. Then, I computed the CIELAB values by taking the white patch (#8) as white reference. Finally, I calculated the ΔE between these measurements and the (fixed) input LAB values as can be seen below:

E7	Target L	Target a	Target b	before ⟶	Extra L	Extra a	Extra b	ΔE extra screen
#4-1	56	-50	32		53,3	-47,4	36,6	5,975985274
#4-2	56	-50	32		53,3	-49,1	36,6	5,444749765
#4-3	56	-50	32		53,4	-48,7	36,8	5,632166546
#4-4	56	-50	32		53,4	-48,7	37,3	6,005713946
#4-5	56	-50	32		53,4	-47,9	36,8	5,877388876
#4-6	56	-50	32		53,7	-49,8	37,3	5,768960045
#4-7	56	-50	32		53,5	-48,4	37,1	5,834123756
#4-8	56	-50	32		53,4	-48,7	36,8	5,632166546
#4-9	56	-50	32		53,4	-47,9	37,3	6,236264908
#4-10	56	-50	32		53,4	-48,7	37,3	6,005713946

FIGURE 4-46 Repeatability of the extra CRT screen

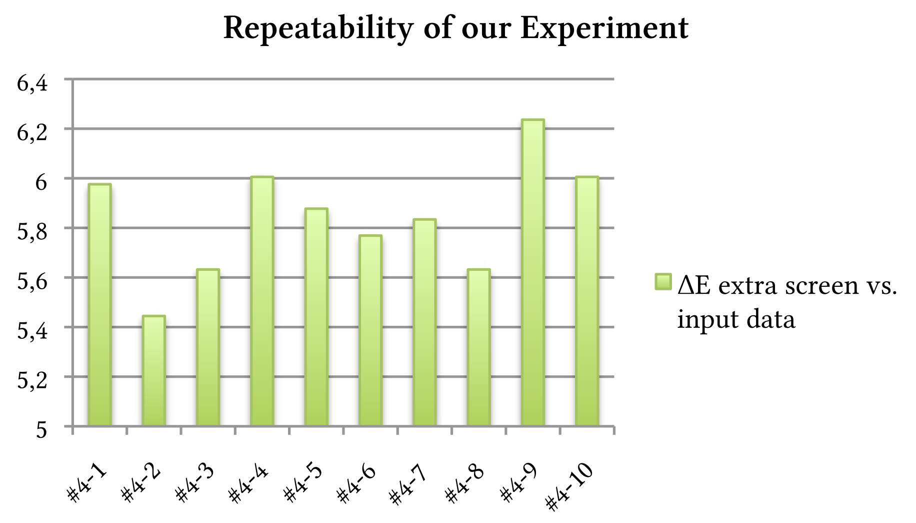


FIGURE 4-47 ΔE comparison for #4 patch on the extra CRT screen

The measurements vary heavily. For the CRT monitor in E7, for example, we get a variation of almost 0.8 ΔE between the lowest value (ΔE = 5.44) and the highest value (ΔE = 6.24)[3]. Now remember that for some before/after comparison of our color patches, the difference was below 1.0 ΔE. Thus the repeatability of our measurement with the colorimeter on our screens is too imprecise to make a well-grounded conclusion.

On the one hand, this could be improved by using a more stable measuring instrument. For example, one could use a spectroradiometer instead of the colorimeter. On the other hand, one could use two screens that are more stable than the ones I used for my experiments. The repeat-

3 For a random measurement of repeatability of the yellow patch (#6) – not documented – I got even bigger variations of the ΔE.

ability is too inacurate for our purpose.

Secondly, the non-uniformity of either one of the screens is a vast source of errors. For the previous measurements, the color patches were measured at one and the same patch. The results of the measurements depend strongly on which geometrically coordinate of the surface of the screen one places the colorimeter on. When looking at the white patch on the LCD screen, for instance, one can see a subtle gradient from a light bluish white on top to a more neutral white at the bottom. During the sixth experiment (E6) I tested the uniformity of the LCD screen by measuring six color patches with the colorimeter placed on the top border of the screen, and then I placed the colorimeter again on the bottom border of the screen:

E6	Target L	Target a	Target b	before →	above Main L	above Main a	above Main b	below Main L	below Main a	below Main b	ΔE above vs. below main screen
#1	63	-16	2		58,93	-18,09	3,38	61,42	-16,26	3,72	3,108793978
#2	66	39	73		60,97	35,15	64,65	63,62	36,8	67,17	4,011907277
#3	57	84	-14		53,45	78,34	-10,02	55,63	82,49	-10,77	4,74735716
#4	56	-50	32		49,27	-51,4	30,38	52,05	-52,4	31,4	3,125507959
#5	75	-5	-8		69,57	-6,51	-5,86	72,67	-4,84	-5,26	3,571960246
#6	91	-10	88		83,86	-13,91	81,92	87,05	-11,49	85,23	5,195055341

Figure 4-48 Lab values measured on the main LCD screen at the top and on the bottom border of the screen

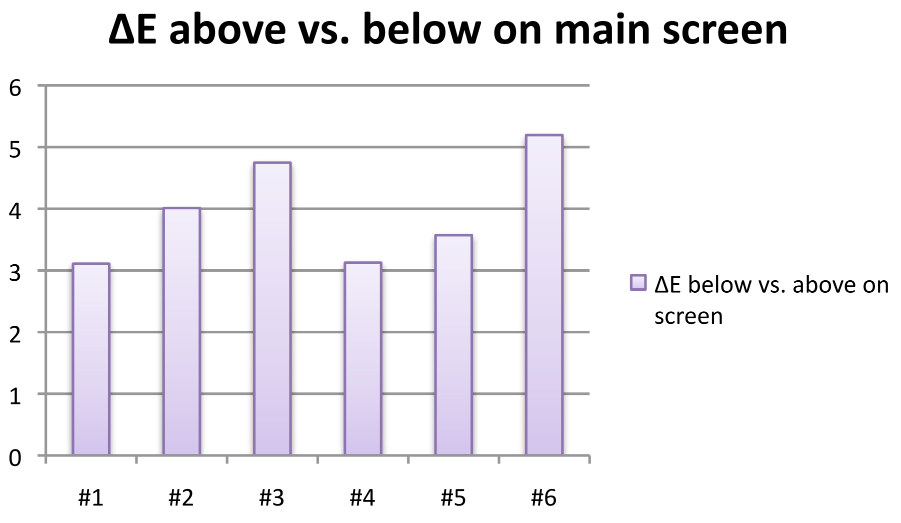


Figure 4-49 ΔE between top and bottom border of the main LCD screen

Please keep in mind that we are looking at measurements on one and the same screen: The differences are quite significant. Some of them are even bigger than some of the measurements we made for two different screens. I tried to reduce this error as I put the colorimeter in the center of the screen both for the characterization and for the measuring. However, even now the differences can vary a lot when the colorimeter is just moved by some centimeters.

In other words, the results vary depending on the geometrical position on the screen's surface. The calibration and characterization that was done for one specific spot on the screen, might be slightly different for any other spot on the screen. During my experimentation, I changed the spot between calibration/characterization and measuring. Thus, the color transformation that are accurate for the spot of the calibration/characterization might be unfitting for the measuring spot. In other words, by moving the colorimeter between calibration/characterization, we have another source of errors.

We could reduce this error related to non-uniformity of the screens as we immediately measure the color patches right after the calibration/characterization without moving the measuring instrument at all. This is somewhat unpractical as I made several measurements on one day and thus moved the colorimeter. In that case, we would have calibrate before every measurement, which would be even more time consuming. Furthermore, we could try to get some screens that are more uniform than the ones used for our experimentation. This is very hard, and nevertheless the improvement would probably not be very high because screens tend to be very non-uniform to a certain degree.

Thirdly, there is the uncertainty of CIELAB conversion that is used for the colorimeter. During my first observations (E1, E2, E3), I got LAB values that had an L value which was over 100. Of course, this can be true but it is quiet unpractical for our purposes. LAB values are computed from the XYZ values relatively to the white point. In our formula, this white point is represented by its XYZ values Xn, Yn and Zn that are obtained by measuring a white patch. The reason for is that of gaining CIELAB values that are normalized so that white results in a CIELAB triple of (100/0/0). The fact that we get L values higher than 100 suggests that MeasuringTool computes the LAB values relatively to a white point whose luminance lies under the screens' actual maximum capacity. Considering furthermore the fact that MeasureTool never asked for a measurement of a reference white patch or any other calibration before the actual measurement, I assume that MeasureTool uses a standard default white point for its calculation like for example D50 or D65.

Nevertheless, assuming that the software used the same white point for all calculations and thus representing a somewhat "absolute" measurement, we could still use the LAB values to compare the difference of a stimulus on one screen to that on the other because. We cannot, however, use these values to compare a measured stimulus to its original input data: As we saw in the previous chapter, our color transformations are based on the relative transformation intent. That means that all transformations occur relative to a device's white point. Thus, in order to compare the performance of the CMM, we have to compare the CIELAB values that are relative to the device's/profile's white point. Unfortunately, the MeasureTool software never asked for our specific white point and thus did not calibrate the transformation.

What is more, I measured the XYZ values of the color patches because I wanted to calculate the LAB values myself during my later experimentations (E4, E5). However, I forgot to include a white patch as reference in the beginning so that I did not have a reference white to compute the values. I figured I could maybe use the XYZ values that are stored in the luminance ("lumi") tag of the profile but the results were not satisfactory either (c. E5). I assume this has to do with the non-uniformity of the screens as described above: Two different positions are taken for both the profiling and measurement; each position with its slightly different, individual white point and maximal luminance. Thus, the luminance stored in the profile's "lumi"-tag might be accurate for the profiling spot but it might not suit the measuring spot at all. Thereby, taking the luminance of the profile might corrupt the results as well.

Fourthly, there is the uncertainty of the ΔE computing. I used two methods for obtaining the ΔE: In the beginning (E1, E2, E3) I relied solely on the ΔE values I got from the MeasureTool but in the end I calculated the ΔE myself. For the first three experiments, I still used a different experimentation setup. I opened a color patch, measured the LAB value on the main screen, opened the external window, measured the LAB value on the extra screen and wrote down both values and the ΔE. Then, I opened the next color patch, measured on the main screen etc. Thus, I changed the position of the colorimeter for every patch which means of course an additional source of error. Furthermore, when I compared the ΔE values from MeasureTool with the values I calculated by hand (i.e. with the help of excel), I noticed that the two values did not resemble each other at all. I assume that MeasureTool is not using the original CIE1976 ΔE formula I have been using but a modified version of it[4].

Fifthly and finally, the number of color patches is not sufficient enough to make a thorough analysis. In order to get a representation of the performance of the CMM, one would definitely use more than eight patches. A more accurate view on the performance could be obtained by evaluating around 40 patches, for example. On the other hand side, 40 patches are very time consuming to measure because everything has to be done by hand (as described above). If one could automatize the presentation and measuring of the patches on screen, one would get a broader and more precise view on the performance. In fact, the Keyframe Software provides already an interface to communicate with the eye-one measuring tool. Thus, one could program a feature that displays random color patches on both screens, measures the CIEXYZ values with the help of the eye-one tool, computes the differences between both screens and presents the results afterwards. Indeed, this could very well be feature or topic of future theses.

4 I later typed in some random values of mine in Bruce Lindbloom's online color difference calculator at http://www.brucelindbloom.com/ and I noticed that my ΔE calculations equaled the CIE1976 value on Lindbloom's page whereas MeasureTools ΔE equaled the CIE1994 value on Lindbloom's page.

Conclusion

What can we improve in the future?

For my bachelor thesis, I implemented a basic color management framework that processes ICC profiles and computes color transformations. Furthermore, the color management feature can be used to upload Nucoda LUTs. Our goal was to prove that the implemented CMM would help solving the color reproduction problem, which means that it would reduce the color difference between main and extra screen when the color management feature is turned on.

Unfortunately, this could not be confirmed with my experimentation. The current experimentation setup could not satisfactorily evaluate the performance of the color management module. At this point, we cannot prove that it works but at least we can assume that it does not fail neither. To get some more accurate results, one would have to improve the experimentation design as described above in the discussion. One would have to use a more accurate measuring instrument like spectrophotometer and some more stable monitors. For a thorough experiment, however, this would be very time consuming and could therefore not be completed for my particular bachelor thesis just yet. A such more precise experimentation could help us to optimize the module, however. So I suggest, somebody could do it in the future.

Moreover, I gained some very valuable experience in setting up an experiment and evaluating its results. As I have never done an experimentation like this before, I made some rather stupid beginner's mistake like not measuring the white patch. So I am sure, I will be much more professional for future experimentations.

Also, I made some very valuable experiences in the field of image programming. I worked for the first time on a challenging project with C++, which helped me to improve my skills in this programming language. What is more, I had my first practical introduction in the field of digital image processing and color engineering and I learned very much about color transformations and ICC profiles.

Finally, this framework could be augmented to enabling proofing, as I would suggest for future projects. In other words, the functions and methods that are already programed for the CMM can be reused for uploading other media's profiles and for emulating their special look in the preview window. One could for example upload the profile of an old Super-8 film and copy its look to a digital movie.

To sum up, in my bachelor thesis I cleared the way for color transformations in Drylab's Keyframe application. This framework could be augmented in the ways as described above. Which would be interesting challenges for future theses.

Figures

References

Adobe. Adobe RGB. http://www.adobe.com/digitalimag/adobergb.html. Checked on March, 14th at 1.51 p.m.

Apple. ColorSync Manager Reference. http://developer.apple.com/library/mac/#documentation/GraphicsImaging/Reference/ColorSync_Manager/Reference/reference.html. Checked on March, 11th 2011 at 10.40 a.m.

Fltk. Introduction to FLTK. http://fltk.org/doc-1.3/intro.html. Checked on March, 14th 2011 at 5.18 p.m.

GraphPad Software, Inc. http://www.graphpad.com/quickcalcs/ttest1.cfm

Green, Phil and Lindsay MacDonald (2002). Colour Engineering. West Sussex: John Wiley & Sons, Ltd.

Heavens, Oliver S. & Robert W. Ditchburn (1991). Insight into optics. New York: John Wiley & Sons, Inc.

IEV. Check International Electrotechnical Commission (IEC). Electropedia.

International Color Consortium (ICC). Introduction to the ICC profile format. http://www.color.org/iccprofile.xalter. Checked on March, 3rd 2011 at 7.11 p.m.

International Color Consortium (ICC). Color Management: Current Practice and The Adoption of a New Standard. http://www.color.org/wpaper1.xalter. Checked on April, 19th at 18.32 pm.

International Color Consortium (ICC). Specification ICC.1:2004-10 (Profile version 4.2.0.0). http://www.color.org/ICC1v42_2006-05.pdf. Checked on April, 19th at 6.31 pm.

International Color Consortium (ICC). Specification of sRGB. IEC 61966-2-1:1999. http://www.color.org/sRGB.pdf. Checked on March, 11th at 12:31 pm.

International Color Consortium (ICC). sRGB (IEC 61966-2-1:1999). http://www.color.org/chardata/rgb/srgb.xalter. Checked on March, 15th 2011 at 1.43 p.m.

International Electrotechnical Commission (IEC). Electropedia. http://www.electropedia.org/ All references are indicated by IEV no. #XXX. Just type in the reference number into the website and you will get the full reference.

International Telecommunication Union (ITU). Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios (BT.601-6 (01/07)). http://www.itu.int/rec/R-REC-BT.601/en. Checked on March, 15th 2011.

International Telecommunication Union (ITU). Parameter values for the HDTV standards for production and international programme exchange (BT.709-5 (04/02)). http://www.itu.int/rec/R-REC-BT.709/en. Checked on March, 15th 2011.

Lindbloom, Bruce. http://www.brucelindbloom.com/

LittleCMS. About Little CMS. http://www.littlecms.com/. Checked on March, 14th 2011 at 5.19 p.m.

Nakamura, Junichi. Image Sensors and Signal Processing for Digital Still Cameras.

Den Norske Filmfestivalen Haugesund. Amandavinnerne 2006. http://www.filmweb.no/filmfestivalen2006/incoming/article111498.ece. Checked on March, 15th 2011.

OpenGL. OpenGL Overview. http://www.opengl.org/about/overview/#1. Checked on March, 14th 2011 at 7.46 p.m.

Farmasøytisk institutt, universitetet i Oslo. "T-test og statistisk signifikans: en smakebit". http://www.uio.no/studier/emner/matnat/farmasi/FRM1210/v05/undervisningsmateriale/T_test.doc Downloaded on April, 18th 2011 at 12:46.

Rosenlund, John Christian's homepage: www.jcr.no.

Sharma, Gaurav (2003). Digital Color Imaging Handbook. Boca Raton: CRC Press LLC.

W3C. A Standard Default Color Space for the Internet - sRGB. http://www.w3.org/Graphics/Color/sRGB. Checked on March, 15th 2011 at 1.45 p.m.

Wikipedia. NTSC. http://en.wikipedia.org/wiki/Ntsc. Checked on March, 15th 2001 at 1.59 p.m.

Wikipedia. PAL. http://en.wikipedia.org/wiki/ITU-R_BT.470-6. Checked on March, 15th 2011 at 1.58 p.m.

Wikipedia. Secam. http://en.wikipedia.org/wiki/SECAM. Checked on March, 15th 2011 at 2.00 p.m.

Wikipedia. Magenta (Farbe). Link: http://de.wikipedia.org/wiki/Magenta_%28Farbe%29. Checked on April, 19th 2011 at 1.45 pm.

Wikipedia. Visual system. Link: http://en.wikipedia.org/wiki/Visual_system, Checked on April, 19th 2011 at 2.30 pm.

Willumsen, Urban. 1991. Fargelære. Oslo: Ad Notam forlag AS.

Appendix: Source Code

```
/*
 *  ColorManagement.cpp
 *  keyframe
 *
 *  Created by Joschua on 1/14/11.
 *  Copyright 2011 Visitech AS. All rights reserved.
 *
 */

#include <map>
#include <iostream>
#include <fstream>
#include <vector>



#include “ColorManagement.h”
#include “../Keyframe.h”
#include “../gui/GradingWindow.h”
#include “../gui/ProjectWindow.h”

ColorManagement::ColorManagement() {
}


void ColorManagement::initializeColorManagementWorkflowAttribute(Project
      *project){


      /* This function checks if a color management workflow has been se-
      lected, and if not this function sets it on default: no color manage-
      ment at all */
      const char * currColorManagementWorkflow;
```

```
currColorManagementWorkflow = project->getAttributeValue(ATTRIBUTE_
PROJECT_COLORMANAGEMENTWORKFLOW);


this->initializeScreenProfilesAttributes();
this->initializeWorkingSpaceAttribute(project);


if (strcmp(project->getAttributeValue(ATTRIBUTE_PROJECT_COLORMANAGE-
MENTWORKFLOW), “”)) {
 if (strcmp(“lutcm”, currColorManagementWorkflow) == 0) {
        printf(“Your color management settings are currently based on
LUTs.\n”);


        project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, true);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM,
false);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, false);
        project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
FLOW, “lutcm”);


 } else if (strcmp(“profilecm”, currColorManagementWorkflow) == 0) {
        printf(“Your color management settings are currently based on
ICC profiles.\n”);


        project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, false);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM, true);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, false);
        project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
FLOW, “profilecm”);
 } else {
        printf(“You have decided to currently not use any color man-
agement at all.\n”);


        project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, false);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM,
false);
        project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, true);
        project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
FLOW, “nocm”);
 }


} else {
```

```
         if (strcmp(project->getAttributeValue(ATTRIBUTE_PROJECT_RADIOLUTSCM),
        “”)) {
                printf(“Your color management settings are currently based on
        LUTs.\n”);


                project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, true);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM,
        false);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, false);
                project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
        FLOW, “lutcm”);


         } else if (strcmp(project->getAttributeValue(ATTRIBUTE_PROJECT_RADIO-
        PROFILESCM), “”)) {
                printf(“Your color management settings are currently based on
        ICC profiles.\n”);


                project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, false);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM, true);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, false);
                project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
        FLOW, “profilecm”);
         } else {
                printf(“You have decided to currently not use any color man-
        agement at all.\n”);


                project->setAttribute(ATTRIBUTE_PROJECT_RADIOLUTSCM, false);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIOPROFILESCM,
        false);
                project->setAttribute(ATTRIBUTE_PROJECT_RADIONOCM, true);
                project->setAttribute(ATTRIBUTE_PROJECT_COLORMANAGEMENTWORK-
        FLOW, “nocm”);
         }


        }
}


//GlImage computeLUTFromProfiles();


void ColorManagement::initializeScreenProfilesAttributes(){
```

```
/* This function checks how many screens are currently connected to
the system
 * and stores the location of the profiles of both screens for the
whole application.
 */
printf("Beginning to initialize screen profiles\' attributes.\n");

/* Get a list with all active displays. Limited to two at this point*/

CGDisplayCount numDisplays;
CGDisplayCount allocatedDisplays = 0;
CGDirectDisplayID *displayID = NULL;

OSStatus cgErr = CGGetActiveDisplayList(0, NULL, &numDisplays);
if (cgErr!=CGDisplayNoErr){
 printf("Error finding number of displays for system1.\n");
 return;
}
allocatedDisplays = numDisplays;
displayID = new CGDirectDisplayID[numDisplays];
cgErr = CGGetActiveDisplayList(2, displayID, &numDisplays);
if (cgErr!=CGDisplayNoErr) {
 printf("Error finding the displays for the system2.\n");
}

/* Getting the profile of the main screen of the system */
CMProfileRef mainProfileRef;
CMProfileLocation mainProfileLocation;

CGDirectDisplayID mainScreenID = (CMDisplayIDType)displayID[0];
OSStatus cmErr = CMGetProfileByAVID(mainScreenID, &mainProfileRef);

if (cmErr!=noErr) {
 printf("ColorSync profile of the main screen could not be openend.");
 return;
}

UInt32 aCount = 255;
```

```
char aName[256];
CMError mErr = CMGetProfileDescriptions(mainProfileRef, aName,
&aCount, NULL, NULL, NULL, NULL);
if (mErr==noErr) {
 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_MAIN-
SCREENPROFILENAME, aName);
}

UInt32 locSize = cmCurrentProfileLocationSize;
OSStatus cmPathError = NCMGetProfileLocation(mainProfileRef, &main-
ProfileLocation, &locSize);

if (cmPathError!=noErr) {
 printf("ColorSync location of the profile for the main screen could
not be obtained.");
 return;
}

application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_MAIN-
SCREENPROFILEURL, mainProfileLocation.u.pathLoc.path);

if (numDisplays>1) {
 CMProfileRef extraProfileRef;
 CMProfileLocation extraProfileLocation;

 CGDirectDisplayID extraScreenID = (CMDisplayIDType)displayID[1];
 OSStatus cmErr = CMGetProfileByAVID(extraScreenID, &extraProfileRef
);

 if (cmErr!=noErr) {
        printf("ColorSync profile of the extra screen could not be
openend.\n");
        return;
 }

 UInt32 bCount = 255;
 char bName[256];
 mErr = CMGetProfileDescriptions(extraProfileRef, bName, &bCount,
NULL, NULL, NULL, NULL);
 if (mErr==noErr) {
        application->configuration.setAttribute(ATTRIBUTE_CONFIGURA-
TION_EXTRASCREENPROFILENAME, bName);
 }


 locSize = cmCurrentProfileLocationSize;
 cmPathError = NCMGetProfileLocation(extraProfileRef, &extraProfile-
Location, &locSize);


 if (cmPathError!=noErr) {
        printf("ColorSync location of the profile for the extra screen
could not be obtained.\n");
        return;
 }


 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_EX-
TRASCREENPROFILEURL, extraProfileLocation.u.pathLoc.path);
 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_EX-
TRASCREENEXISTS, 1);
} else {
 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_EX-
TRASCREENPROFILENAME, "");
 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_EX-
TRASCREENPROFILEURL, "");
 application->configuration.setAttribute(ATTRIBUTE_CONFIGURATION_EX-
TRASCREENEXISTS, 0);
}
}


void ColorManagement::doColorManagement(Project *project){


      const char * currColorManagementWorkflow;
      currColorManagementWorkflow = project->getAttributeValue(ATTRIBUTE_
PROJECT_COLORMANAGEMENTWORKFLOW);


      if (strcmp("profilecm", currColorManagementWorkflow) == 0) {
       this->initializeScreenProfilesAttributes();
       this->initializeWorkingSpaceAttribute(project);


       this->doProfileColorManagement(project);
```

```
	 printf(“Your color management settings are currently based on ICC
	profiles.\n”);
	} else if (strcmp(“lutcm”, currColorManagementWorkflow) == 0) {
	 this->initializeScreenProfilesAttributes();
	 this->initializeWorkingSpaceAttribute(project);

	 this->doLUTColorManagement(project);

	 printf(“Your color management settings are currently based on
	LUTs.\n”);
	} else if (strcmp(“nocm”, currColorManagementWorkflow) == 0) {
	 this->doNoColorManagement(project);

	 printf(“You have decided to currently not use any color management
	at all.\n”);
	} else {
	 printf(“Your color managenment workflow settings could currently not
	be detected.\n”);
	}

	application->window.getGradingWindow()->updateTransformed();

	return;

}

void ColorManagement::doNoColorManagement(Project *project){
	/* Reset the luts for both main screen and extra screen */

	GlCanvas *mainScreenCanvas = application->window.getGradingWindow()-
	>getPrimaryCanvas();
	GlCanvas *extraScreenCanvas = application->window.getGradingWindow()-
	>getTertiaryCanvas();

	if (mainScreenCanvas){
	 this->resetGlCanvas(mainScreenCanvas);
	}
	if (extraScreenCanvas){
	 this->resetGlCanvas(extraScreenCanvas);
```

```
        }
}


vector<RGBPixel> getRGBValuesFromNucodaFile(const char * path){
        /* Retrieves the RGB values from a Nucoda file and returns them as a
        vector of RGBPixels */


        bool isNucodaFile = false;
        bool isRGBValue = false;
        int depth = 0;
        vector<RGBPixel> rgbValues;


        isNucodaFile = ColorManagement::isNucodaFile(path);


        if(isNucodaFile){


         ifstream f(path);



         string line = "";


         while (getline(f, line)) {
                 RGBPixel p;


                 isRGBValue = ("%d\n", sscanf(line.c_str(), "%f %f %f\n", &p.r,
        &p.g, &p.b)) == 3;


                 if (isRGBValue){
                         rgbValues.push_back(p);
                 }
         }
        }
}


        return rgbValues;
}


bool ColorManagement::isNucodaFile(const char *path){
      bool isNucodaFile = false;

      if (strcmp(path, “”) != 0) {
       ifstream f(path);

       if ( ! f.is_open() ) {
              printf(“Failed to open Nucoda LUT File.\n”);
       }
       else {
              string line = “”;

              while (getline(f, line)) {

                     if (!isNucodaFile){
                            const char * tempString = “NUCODA_3D_CUBE 2”;

                            isNucodaFile = (int)line.find(tempString) != -1;
                     } else {
                            return true;
                     }
              }
       }

      } else {
       printf(“You have not chosen a Nucoda LUT directory yet.\n”);
      }

      return isNucodaFile;
}

int ColorManagement::getNucodaDepth(const char *path){

      int depth = 0;

      if (strcmp(path, “”) != 0) {
       ifstream f(path);
```

```
        if ( ! f.is_open() ) {
              printf("Failed to open Nucoda LUT File.\n");
        }
        else {
              string line = "";

              while (getline(f, line)) {

                    if (depth==0) {
                          sscanf(line.c_str(), "LUT_3D_SIZE %d\n", &depth);
                    } else {
                          return depth;
                    }
              }
        }
        return depth;

       } else {
        printf("You have not chosen a Nucoda LUT directory yet.\n");
        return depth;
       }
}

bool ColorManagement::convertNucodaIntoGlImage(const char *path, GlImage
      *lut, int depth){

      bool success = false;

      vector<RGBPixel> rgbValues;

      if (strcmp(path, "") != 0) {

       rgbValues = getRGBValuesFromNucodaFile(path);

       printf("size of the final vector %d\n", (int)rgbValues.size() );
       if((int)rgbValues.size()==depth*depth*depth){
```

```
                int x=0;
                for(int b=0;b<depth; b++){
                        for (int g=0;g<depth;g++){
                                for (int r=0;r<depth;r++) {
                                        lut->setSample(lut->getSampleIndex(r, g, 1,
        b), rgbValues[x].g);
                                        lut->setSample(lut->getSampleIndex(r, g, 1,
        b), rgbValues[x].g);
                                        lut->setSample(lut->getSampleIndex(r, g, 2,
        b), rgbValues[x].b);
                                        x++;
                                }
                        }
                }

                success = true;

                printf("Converting NUCODA to GlImage succeeded.\n");
         } else {
                printf("Something is wrong with your Nucoda file: Not enough
        values for all sample points.\n");
         }

        } else {
         printf("You have not chosen a Nucoda LUT directory yet.\n");
        }

        return success;
}

void ColorManagement::doLUTColorManagement(Project *project){

        bool mainSuccess = false;
        bool extraSuccess = false;

        /* Set LUT for main screen */
        GlCanvas *mainScreenCanvas = application->window.getGradingWindow()-
        >getPrimaryCanvas();
        //GlImage *mainScreenLut = NULL;
```

```
const char * mainLUTNucodaPath = project->getAttributeValue(ATTRIBUTE_
PROJECT_IMPORTMAINSCREENLUTDIRECTORY);
int mainDepth;
if(isNucodaFile(mainLUTNucodaPath)){
 mainDepth = getNucodaDepth(mainLUTNucodaPath);
 if(mainDepth!=0){
        mainScreenLUT = mainScreenCanvas->getLut3D();
        if (mainScreenLUT) {
               mainScreenLUT->deActivate();
               mainScreenLUT->allocate(mainDepth, mainDepth, 3, main-
Depth);
               mainScreenLUT->initializeLut();
        } else {
               mainScreenLUT = new GlImage(mainDepth, mainDepth, 3,
COLORIMETRIC_RGB, mainDepth);
               mainScreenLUT->initializeLut();
               mainScreenCanvas->setLut3D(mainScreenLUT);
        }
        printf("Converting main screen Nucoda LUT.\n");
        mainSuccess = convertNucodaIntoGlImage(mainLUTNucodaPath,
mainScreenLUT, mainDepth);
 }


}


if (!mainSuccess) {
 this->resetGlCanvas(mainScreenCanvas);
}



/* Set LUT for extra Screen */
GlCanvas *extraScreenCanvas = application->window.getGradingWindow()-
>getTertiaryCanvas();
GlImage *extraScreenLut = NULL;


if (extraScreenCanvas) {
 const char * extraLUTNucodaPath = project-
>getAttributeValue(ATTRIBUTE_PROJECT_IMPORTEXTRASCREENLUTDIRECTORY);
```

```
        int extraDepth;

        if(isNucodaFile(extraLUTNucodaPath)){
               extraDepth = getNucodaDepth(extraLUTNucodaPath);
               if(extraDepth!=0){

                      extraScreenLut = extraScreenCanvas->getLut3D();
                      if (extraScreenLut) {
                             extraScreenLut->deActivate();
                             extraScreenLut->allocate(extraDepth, extraDepth,
       3, extraDepth);
                             extraScreenLut->initializeLut();
                      } else {
                             extraScreenLut = new GlImage(extraDepth, ex-
       traDepth, 3, COLORIMETRIC_RGB, extraDepth);
                             extraScreenLUT->initializeLut();
                             extraScreenCanvas->setLut3D(extraScreenLut);
                      }
                      printf("Converting extra screen Nucoda LUT.\n");
                      extraSuccess = convertNucodaIntoGlImage(extraLUTNucodaP
       ath, extraScreenLut, extraDepth);
               }
        }
        if (!extraSuccess) {
               this->resetGlCanvas(extraScreenCanvas);
        }
       }
}

void ColorManagement::initializeWorkingSpaceAttribute(Project *project){
       /* Set the working space for the project and stores the URL to the
       profile in an attribute. */

       const char * currWorkingSpace;
       currWorkingSpace = project->getAttributeValue(ATTRIBUTE_PROJECT_WORK-
       INGSPACEPROFILE);

       map<string, ColorSpaceMetaData *> workingSpaceProfilePathDict;
```

```
      /* #HDEG: Initialize this collection for the whole application */
      workingSpaceProfilePathDict[“sRGB”] = new ColorSpaceMetaData(true, ap-
      plication->directoryHelper.getProfilesDirectory() / “sRGB Color Space
      Profile.ICM” );
      workingSpaceProfilePathDict[“DCI”] = new ColorSpaceMetaData(false, ap-
      plication->directoryHelper.getProfilesDirectory() / “sRGB Color Space
      Profile.ICM” );
      workingSpaceProfilePathDict[“None”] = NULL; //new
      ColorSpaceMetaData(false, application->directoryHelper.getProfilesDi-
      rectory() / “sRGB Color Space Profile.ICM” );
      workingSpaceProfilePathDict[“Adobe RGB (1998)”] = new
      ColorSpaceMetaData(true, application->directoryHelper.getProfilesDi-
      rectory() / “AdobeRGB1998.icc” );
      workingSpaceProfilePathDict[“Rec. 709”] = new ColorSpaceMetaData(true,
      application->directoryHelper.getProfilesDirectory() / “VideoHD.icc” );
      workingSpaceProfilePathDict[“Rec. 601 (PAL/SECAM)”] = new
      ColorSpaceMetaData(true, application->directoryHelper.getProfilesDi-
      rectory() / “VideoPAL.icc” );
      workingSpaceProfilePathDict[“Rec. 601 (NTSC)”] = new
      ColorSpaceMetaData(true, application->directoryHelper.getProfilesDi-
      rectory() / “VideoNTSC.icc” );


      zzzz} else {
       printf(“No working space chosen.\n”);
       project->setAttribute(ATTRIBUTE_PROJECT_WORKINGSPACEPROFILEURL, “”);
      }
}


void ColorManagement::resetGlCanvas(GlCanvas *canvas){
      /* Reset the lut connected to a GlCanvas to a neutral LUT. */


      GlImage *lut = canvas->getLut3D();
      if(lut){
       lut->deActivate();
       lut->allocate(32, 32, 3, 32);
       lut->initializeLut();
       lut->gain(1.0);
      }else {
       lut = new GlImage(32, 32, 3, COLORIMETRIC_RGB, 32);
       lut->initializeLut();
       lut->gain(1.0);
       canvas->setLut3D(lut);
      }
}


void ColorManagement::doProfileColorManagement(Project *project){
      /* Compute the color transformation and create GlImage lut depending
      on the profiles of the selected working space and the two screens. */



      printf("working_space: %s\n", project->getAttributeValue(ATTRIBUTE_
      PROJECT_WORKINGSPACEPROFILE));


      GlCanvas *mainScreenCanvas = application->window.getGradingWindow()-
      >getPrimaryCanvas();
      GlCanvas *extraScreenCanvas = application->window.getGradingWindow()-
      >getTertiaryCanvas();


      /* Getting the profile of the current working space */
      const char * workingSpaceProfileURL = project-
      >getAttributeValue(ATTRIBUTE_PROJECT_WORKINGSPACEPROFILEURL);


      if (strcmp(workingSpaceProfileURL, "") != 0){
       workingSpaceProfile = cmsOpenProfileFromFile(workingSpaceProfileURL,
      "s");
      } else {
       printf("Working space profile could not be found.\n");
       if (mainScreenCanvas) {
              this->resetGlCanvas(mainScreenCanvas);
       }
       if (extraScreenCanvas) {
              this->resetGlCanvas(extraScreenCanvas);
       }


       return;
      }


      if (!workingSpaceProfile){
       printf("Working space profile could not be opened.\n");
       if (mainScreenCanvas) {
              this->resetGlCanvas(mainScreenCanvas);
```

```
 }
 if (extraScreenCanvas) {
        this->resetGlCanvas(extraScreenCanvas);
 }

 return;
} else {

 /* getMainScreenProfileLUT*/
 const char *mainProfileLocationPath = application->configuration.
getAttributeValue(ATTRIBUTE_CONFIGURATION_MAINSCREENPROFILEURL);
 printf("mainscreenprofilepath: %s\n", mainProfileLocationPath);
 mainScreenProfile = cmsOpenProfileFromFile(mainProfileLocationPath,
"s");

 if (mainScreenProfile == NULL) {
        printf("LittleCMS profile of the main screen could not be
opened.\n");
        if (mainScreenCanvas) {
               this->resetGlCanvas(mainScreenCanvas);
        }
 } else {

        cmsHTRANSFORM tempLUT = cmsCreateTransform(workingSpaceProfi
le, TYPE_RGB_8, mainScreenProfile, TYPE_RGB_8, INTENT_RELATIVE_COL-
ORIMETRIC, cmsFLAGS_GAMUTCHECK);

        GlImage *tempInput = new GlImage(32, 32, 3, COLORIMETRIC_RGB,
32);

        mainScreenLUT = mainScreenCanvas->getLut3D();
        if (mainScreenLUT) {
               mainScreenLUT->deActivate();
               mainScreenLUT->allocate(32, 32, 3, 32);
               mainScreenLUT->initializeLut();
        } else {
               mainScreenLUT = new GlImage(32, 32, 3, COLORIMETRIC_RGB,
32);
               mainScreenCanvas->setLut3D(mainScreenLUT);
        }
        tempInput->initializeLut();
        cmsDoTransform(tempLUT, tempInput->getBufferU8(), mainScreen-
LUT->getBufferU8(), 32*32*32);

        cmsCloseProfile(mainScreenProfile);
        cmsDeleteTransform(tempLUT);
 }

 /* getExtraScreenLUT */
 if (extraScreenCanvas) {

        const char *extraProfileLocationPath = application-
>configuration.getAttributeValue(ATTRIBUTE_CONFIGURATION_EXTRASCREEN-
PROFILEURL);
        //Path extraPath = application->directoryHelper.getProfilesDi-
rectory() / “Extra-screen_11-04-2011_1.icc”;
        //const char *extraProfileLocationPath = (const char *)extra-
Path.getBuffer();
        printf(“extrascreenprofilepath: %s\n”, extraProfileLocation-
Path);
        extraScreenProfile = cmsOpenProfileFromFile(extraProfileLocat
ionPath, “s”);

        if (extraScreenProfile == NULL) {
               printf(“LittleCMS profile of the extra screen could not
be opened.”);
               if (extraScreenCanvas) {
                      this->resetGlCanvas(extraScreenCanvas);

               }
               return;
        } else {
               printf(“extraScreenCanvas found.\n”);

               cmsHTRANSFORM tempLUT = cmsCreateTransform(workingSpa
ceProfile, TYPE_RGB_8, extraScreenProfile, TYPE_RGB_8, INTENT_RELA-
TIVE_COLORIMETRIC, cmsFLAGS_GAMUTCHECK);

               GlImage *tempInput = new GlImage(32, 32, 3, COLORIMET-
RIC_RGB, 32);
```

```
                mainScreenLUT = extraScreenCanvas->getLut3D();
                if (mainScreenLUT) {
                        mainScreenLUT->deActivate();
                        mainScreenLUT->allocate(32, 32, 3, 32);
                        mainScreenLUT->initializeLut();
                } else {
                        mainScreenLUT = new GlImage(32, 32, 3, COLORIMET-
RIC_RGB, 32);
                        extraScreenCanvas->setLut3D(mainScreenLUT);
                }

                tempInput->initializeLut();
                cmsDoTransform(tempLUT, tempInput->getBufferU8(), main-
ScreenLUT->getBufferU8(), 32*32*32);

                cmsCloseProfile(extraScreenProfile);
                cmsDeleteTransform(tempLUT);
        }
 }

 return;

}
}
```