A Novel Approach to Color Conversion on Embedded Devices

Alexandros Dermenakis
Abstract

A Novel Approach to Color Conversion on Embedded Devices

Alexandros Dermenakis

Color profiles have become an integral part of output devices. They are used to maintain color consistency among different input and output devices such as screens and printers. Embedded systems have omitted support for color profiles since they have limited resources and are not able to handle such operations in real-time. The goal of this thesis report, is to provide a solution for such systems by using a graphics accelerator to perform color profile conversion real-time. Benchmarks on different mobile devices are presented discussing the performance differences and image quality improvements by using color profiles, along with drawbacks of each system. The aim of this report is to provide sufficient information on how color profile conversion takes place, how to optimize the high cost of conversion, as well as a smart color profile conversion mechanism that can be integrated in the graphics stack of Qt.
Contents

List of Figures 3

1 Introduction 4
  1.1 Problem ........................................... 4
  1.2 Coverage & Overview ............................. 4

2 Background Information 6
  2.1 Color Profiles .................................... 6
  2.2 Mathematical Background ........................ 8
  2.3 Similar Work ..................................... 8

3 Analysis & Experimentation 10
  3.1 Hardware Used ................................... 10
     3.1.1 Nokia C0 .................................. 11
     3.1.2 Nokia N900 ................................. 11
     3.1.3 Nokia N9 .................................. 11
  3.2 Conversion on the Central Processing Unit (CPU) .. 11
  3.3 Conversion on the Graphics Processing Unit (GPU) ... 12
  3.4 Color Conversion ................................ 12
     3.4.1 Convert from image to linear color space ... 13
     3.4.2 Convert from image to a Gamma Color Space .. 14
     3.4.3 Convert from image to screen color space ... 18

4 Discussion & Conclusion 22
  4.1 Comparison of Techniques .......................... 22
     4.1.1 Comparison of Benchmarks .................. 23
     4.1.2 Future Work .................................. 23

5 Appendix 26
  5.1 Appendix A : LittleCMS Plugin Code ............... 26

Glossary 30

References 32
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cone sensitivities of our optical system</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>Chromatic diagram of the CIE 1931 color space</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Gamma curve and exponential curve converting to linear</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>RGB to XYZ Transformation</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Conversion to RGB procedure</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>Color Conversion CPU Benchmarks on different phones</td>
<td>12</td>
</tr>
<tr>
<td>7</td>
<td>Color Conversion on CPU vs GPU</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>FBO read benchmark</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>Conversion dataflow with Gamma2</td>
<td>15</td>
</tr>
<tr>
<td>10</td>
<td>Conversion dataflow with sRGB</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td>Gamma2/sRGB to Any vs Any to Any Benchmark</td>
<td>23</td>
</tr>
</tbody>
</table>
1 Introduction

Color imaging expects color quality, along with portability. With the term portability we refer to maintaining the colors of an image across devices. Color profiles were designed to preserve color properties of an image in order to have the same output result on all output devices. Presently there is no solution for embedded devices to perform color conversion. The purpose of this research project is to provide a real-time solution for performing color conversion on embedded systems when rendering on their displays. In addition to the initial conversion mechanism we research, we are also experimenting and optimizing the solution since embedded systems need to be efficient in terms of power consumption and we need to take into account their computational limitations.

1.1 Problem

Color Profiles are meant to maintain the chromatic characteristics of an image when sent to output devices. Each screen, printer or other type of output device represents colors in a different way. In order to be able to maintain color consistency when transferring color information among different devices we can use Color Profiles to maintain it. To solve that problem we can prepare an image on a specific screen as an example and then transport the image in the color space of the screen where the image was constructed. Upon reception of the image, the receiver will have to open the image and convert from the source color profile to his screen’s color profile. By performing that color transformation, the output result the receiver will see is the same (considering no compression/precision losses) with the sender.

As presented later in section 2.3 where we present external libraries implementing color profiles, there are solutions that allow color transformations of images on embedded devices. These libraries lack optimizations; hence they can not be used for transforming 30 Frames Per Second (FPS) which is the minimum required frame rate for being real-time. This is the main problem we are discussing and experimenting in the upcoming sections.

1.2 Coverage & Overview

Color Profiles are a very broad aspect of imaging, in this report we are focusing on conversion using OpenGL ES 2.x and OpenGL ES 3.x. We have skipped implementation for Microsoft based embedded systems since they require a different implementation based on DirectX.

The research aspects covered in this report are related to color blending with a low loss in quality and optimization of the color conversion for real time usage on embedded devices. We are mainly targeting embedded devices since desktop systems already implement several color profile mechanisms using GPUs and CPUs; hence this would be a different topic mainly focused on optimization and performance.

This report describes the results of different color tests and benchmarks presenting the importance of color profiles in digital images, as well as benchmarks providing comparison among different solutions. The focus is on portability of a color correction mechanism for embedded systems through GPUs that has
support for sRGB color space in hardware. We have used different techniques for performing color correction and balancing between performance and image quality. The aim of this report is to provide a concrete background for developing a cross-platform system for color correction on embedded systems that could be also ported in the graphics stack of Qt running on embedded devices. We have omitted support for output devices such as printers, since embedded devices deal mainly with real-time rendering on displays and screens. Devices other than screens are bind to using the CPU for the color conversion. Additionally, devices such as printers do not have any real-time rendering requirements and frame rates.
2 Background Information

This section focuses on the basics of Color Profiles and provides sufficient background so that the reader can follow the preceding sections. It gives an overview of the human optical system and the perception of colors. Additionally, it gives a brief history background on color profiles, as well as information on gamma correction and the mathematical background for performing color conversion.

2.1 Color Profiles

The Color Profile research started it the 1920s by William David Wright\textsuperscript{[22]} and John Guild\textsuperscript{[21]} who contributed in the creation of the CIE 1931\textsuperscript{[9]} color space. This color space is considered the milestone of Color Profiles. The CIE 1931 color space considers the human eye’s perception of colors and is based in the cone sensitivities of the human optical system. The diagram below shows the different cone sensitivities of our optical system which indicates that we are a lot more sensitive to blue color than green and red.

As stated above the human eye has high sensitivity to blue; hence the CIE 1931 color space boosts more the red and green part of the spectrum since it is easy for our eye to perceive different tonal values of blue. The diagram below shows the chromatic diagram of the CIE 1931 color space which shows the distribution of color and blendings.

Apart from the coloring information held in the CIE 1931 color space, there is a gamma curve which is used for converting from a linear color space to a Logarithmic representation of the colors since the perception of our visual system behaves logarithmically\textsuperscript{[11]}. The diagram below shows the logarithmic curve (the curve of a screen or other output device) and the exponential curve (the curve we use when we are applying gamma correction), which are combined in order to achieve a linear distribution of colors.

In 1993 eight big companies that were having issues with color consistency when transferring image data among devices founded the International Color Consortium (ICC). ICC is a foundation that designs a color profile standard and distributes it openly. The ICC standard has been accepted by the International Organization for Standardization (ISO) and is implemented on almost...
every input (i.e. photographic camera) and output (i.e. screen, printer) device on the market. Microsoft in collaboration with Hewlett-Packard (HP) constructed a new color profile in 1996 that was using a different gamma curve and had different distribution of colors. In M. Stokes et al. discuss how the Cathode Ray Tube (CRT) monitors work and the importance of the gamma curve for properly visualizing colors. CRT monitors have different gamma values; hence there is a need to convert the colors from their initial color space to the color space of the monitor.
2.2 Mathematical Background

Color transformations can be represented by matrix operations. The input of the matrix system are the color values and the transformation matrix that converts from the source to the destination Color Profile (RGB is the color input for the system of matrices below). The resulted colors are represented by the XYZ in the output of the matrix operations.

\[
\begin{bmatrix}
X \\
Y \\
Z
\end{bmatrix} = \begin{bmatrix}
R \\
G \\
B
\end{bmatrix}
\]

Figure 4: RGB to XYZ Transformation[6].

where:

\[
\begin{bmatrix}
\end{bmatrix}
\]

\[
\begin{bmatrix}
X_r - x_r / y_r \\
y_r = 1 \\
Z_r = (1-x_r - y_r) / y_r \\
X_r - x_r / y_r \\
y_r = 1 \\
Z_r = (1-x_r - y_r) / y_r \\
X_r - x_r / y_r \\
y_r = 1 \\
Z_r = (1-x_r - y_r) / y_r
\end{bmatrix}
\]

\[
\begin{bmatrix}
\end{bmatrix}
\]

Figure 5: Conversion to RGB procedure[6].

For the most commonly used color spaces there are constants which are meant to replace the components of the matrix above. These constants can be found at http://www.brucelindbloom.com/index.html?Eqn_RGB_XYZ_Matrix.html These constants could also incorporate the gamma value of source and destination profile, reducing the higher cost of computations at run time for the gamma correction.

2.3 Similar Work

There exist several color profile implementations on various Operating Systems (OSs). The most complete OS implementation of color profiles is the one from Mac OS X. For more information and sample code of using the OS’s functions for color profiles use this link: https://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/DrawColor/Tasks/UsingColorSpaces.html Windows also provides an Application Programming Interface (API) for color profiles through the OS but its implementation is not as complete as that of Mac OS X. On Linux there are several side projects that are implementing color
profiles and can be added as modules to the Window Manager of the system. Through these modules the user has high level access to apply the color profile of his screen for every buffer that is rendered on the screen. The most famous project is called “Oyranos” and is designed to integrate to the Gnome and KDE window manager under Linux (for more information about the “Oyranos” project refer to its website at: [http://www.oyranos.org/](http://www.oyranos.org/)). This functionality is limited but sufficient for many users and programmers. In addition to the aforementioned Color Profile support from OSs, there is the LittleCMS library which provides a more concrete and low level implementation for applying color profiles. This library loads directly binaries of ICC Color Profile and can generate transfer functions from one color profile to another, as well as perform the transformation of a graphics buffer to a color profile. The LittleCMS library can be easily compiled for embedded systems since its implementation is in C and does not use any OS specific calls, neither CPU optimizations.
3 Analysis & Experimentation

The experiments section provides information on different approaches used while experiment on embedded devices. It gives feedback on the analytical steps taken to achieve Color Profile conversion on different devices.

The first step was to analyze the information of the hardware limitations related to the devices presently on the market. The second step was to search for similar work that would help understanding how color conversion takes place, as well as why the existing implementations are not used on embedded systems.

In the section 2.3 we mentioned the LittleCMS library which provides a full implementation for color profile conversion for ICC Color Profiles. The first experiment described in section 3.2 uses the LittleCMS library and presents the timing graphs after performing the color conversion on three different devices. After the conversion on the CPU we tried performing a conversion using the GPU. In section 3.2 where we are discussing the conversion on the GPU we run a simple color profile conversion and time it.

Apart from the experimentation on hardware we also compare the quality of visual results. Most images that do not require an alpha channel are stored in Red Green Blue (RGB) format using 8 bits per color (meaning a total of 24 bit for storing each pixel). For the images that require an alpha channel RGB is used with 8 bit per color, resulting in a total of 32 bit per pixel [5]. The different types of byte alignment in the memory is described in [5], showing the paddings that need to be used in order to extract specific colors from images.

By using 8 bit per pixel we are bound to use 256 different values for each color, adding a limitation on the amount of information that can be stored for each pixel [10]. By limiting the amount of data that can be stored for a pixel we can observe a loss of quality since not all colors can be represented; hence the 16 bit format was introduced. With the 16 bit format the amount of color data that can be stored increases exponentially providing better image quality. As [2] suggests, when images require several phases of color modifications and saving passes the 16 bit format shall be used.

The Color Profile conversion can be seen as some way of image editing since we are manipulating the data of the image, as well as often when images are loaded several blending operations take place for merging two or more images. All these operations could result in high loss of data. In our tests we are presenting only 8 bit conversions since it is the most widely used format. In this report since we can not change on the GPU the bits to be used for each color we will not cover this aspect. Improving the color quality by using more precision could be further researched on future hardware that will have the ability to use more bits per color.

3.1 Hardware Used

For the benchmarks we used three different phones provided by Nokia Norge AS. It has to be noted that publicly there is lack of information on the Nokia C0 since it was not publicly released. These three devices are from three different generations which allows understanding the expectations of upcoming hardware.
3.1.1 Nokia C0

- **Internal Release December 2010**
- **CPU** 680 MHz ARM 11
- **GPU** Broadcom BCM2727
- **RAM** 128 MB
- **OS** Symbian 3

3.1.2 Nokia N900

- **Released in November 2009**
- **CPU** TI OMAP3430, Single core, 600 MHz, ARM Cortex-A8
- **GPU** PowerVR SGX530
- **RAM** 256 MB
- **OS** Linux (Maemo 5)

3.1.3 Nokia N9

- **Released in June 2011**
- **CPU** TI OMAP3630, Single core, 1 GHz, ARM Cortex-A8
- **GPU** PowerVR SGX530
- **RAM** 1 GB
- **OS** Linux MeeGo OS, v1.2 Harmattan

3.2 Conversion on the CPU

We have used this approach to benchmark the execution time for performing the color profile conversion through different CPUs. Since CPUs are design to serve different purposes they have a flexible implementation which also allows them to perform graphics operations. The drawback of their generic usage is a high performance overhead when working with graphics since the hardware is not specialized for such operations.

The benchmarks below show execution of color conversions on three different types of embedded systems, providing information on how long it takes to simply convert a pixel buffer from one color profile to another.

The results presented above show that the embedded devices presently on the market are not capable of performing color conversion on the fly for every frame. In order to successfully render real time we need to have a maximum processing of 33ms per frame to achieve a framerate of 30 frames per second. The aforementioned time includes all the operations that take place in the graphics thread for each frame.
3.3 Conversion on the GPU

In this approach we are using a graphics hardware accelerator to perform the color profile conversion. We are using the graphics unit of an embedded system for performing the color conversions. GPUs have been designed to work with floating point arithmetic. Additionally, the memory that a GPU has access to is the place where the buffers sent to an output device are held. Moreover, GPUs by default allow performing multithreaded conversion since most of the time fragments are independent of each other. This way higher performance is achieved.

The benchmark below shows the execution times of the same test discussed in the previous section. The input used is the same as that of the previous section, as well as the hardware that has been used. The difference is that we are using a GPU instead of a CPU for the color conversion.

We tried grabbing the converted information from the graphics memory of the embedded device by using an Frame Buffer Object (FBO) read command. The FBO read command transfer the graphics buffer from the graphics memory to RAM allowing the user to manipulate the buffer using the CPU before re-sending it to the CPU for being drawn. The purpose of this test was to see the cost of reading the graphics memory in order to allow manipulating the buffer through existing code running on the CPU.

The benchmark below shows the FBO read times for the three different devices we are using in our tests. We can clearly see that performing every frame an FBO read, would not be possible since we will not be real-time.

3.4 Color Conversion

This section provides an analytical explanation of the different paths we researched for performing the color correction and blending. The sections 3.2 and 3.3 analyzed the performance of the hardware component that performs the
3.4.1 Convert from image to linear color space

In this approach we are reading an input image which is converted at load time into a linear color space. We are choosing linear since all OpenGL operations such as blending and texture filtering assume linear values \[14\]. This way we provide to OpenGL linear values as it expects and this mechanism allows proper blending and linear interpolation of color values.

As soon as all the blending and filtering operations have taken place, we need to covert from the linear color space to the color profile of the screen while drawing conversion. The discussion of the results are compared in section \[4\].
or right before drawing.
The result of storing an image in a linear color space results in a high quality loss. As described in section 2, the human vision system uses a logarithmic scale; hence we would need a more than 8 bit precision. This solution could be considered with respect to quality if at least 16 bits could be used.

3.4.2 Convert from image to a Gamma Color Space

This scenario describes the case that we convert the input image from its color space to a linear color space and then depending on the hardware we convert from linear color space to sRGB or Gamma 2. The initial conversion takes place on the CPU since we know it is a static cost and does not affect the real-time performance. For this case using the GPU could be considered since there is hardware that can accelerate that conversion. This is out of the scope of this research paper; hence we will only mention it.

OpenGL ES 2.x

OpenGL ES 2.x has no sRGB support in the definition of its standard. There are devices that implement a special extension named "GL_EXT_sRGB." For hardware supporting this extension, please refer to section 3.4.2. This section provides a solution for the devices implementing the OpenGL ES 2.x standard and lack the special sRGB extension. The alternative solution described below, is to convert the input images to a gamma 2 color space. The hardware accelerated blending can not be used since it assumes linear textures and framebuffers. To solve that problem we are using shaders for all the blending operations. The gamma 2 color space was chosen due to its low conversion cost. For converting from a linear color space to a gamma 2 color space we need to compute the square root of the value of each pixel. In order to convert from gamma 2 to linear, we need to square the value of each pixel. As explained in [12] to achieve a higher gammut compression we could have used gamma 2.2 or 2.4 with the drawback of lower performance. By using the aforementioned special cases, the system will use a generic power function which requires more instructions than working directly with powers of 2.

The diagram below illustrates the steps required in order to perform proper color blending. We keep the pixel color information stored with a gamma 2 color profile in memory and whenever we perform any blending operations on the shader we convert to linear and then back to gamma 2 as described in [12].

The sample code below is taken from [12] and converted into OpenGL ES shader code. The code shows the steps that need to be taken in order to and from gamma 2/2.2 to linear and vice versa. As stated above these operations do not require much computational power hence when they are added in the blending shader.

Manually Converting Color Values to a Linear Space

```cpp
vec3 diffuseCol = pow( texture(diffTex, texCoord).rgb, 2.2 );
// Or (cheaper, but assuming gamma of 2.0 rather than 2.2)
vec3 diffuseCol = texture( diffTex, texCoord );
fuseCol = diffuseCol * diffuseCol;
```

Last-Stage-Output Gamma Correction

14
vec3 finalCol = do_all_lighting_and_shading();
float pixelAlpha = compute_pixel_alpha();
return vec4(pow(finalCol, 1.0 / 2.2), pixelAlpha);
// Or (cheaper, but assuming gamma of 2.0 rather than 2.2)
return vec4( sqrt( finalCol ), pixelAlpha );

The code above shows how to perform the blending and the conversion using RGB based color spaces. Very often chroma sub sampling color spaces are used for videos, compressed images and for custom pixel storage formats. Popular formats of chroma sub sampling are the YCoCg and YCbCr which are used in JPEG and several video codecs since it allows compression and in-
terpolation between frames at low quality loss. These formats are often used since our vision system is more sensitive to luminance changes and chroma changes come secondary. An example an implementation using YCoCg as an intermediate format can be found at http:// codedeposit.blogspot.be/2010/01/pre-linearized-wide-gamut-dxt5-ycocg.html. In that case a custom YCoCg implentation is used for the textures processed in the shader. In order to blend these textures a different mechanism used by OpenGL shall be used.

OpenGL ES 3.x

In OpenGL 3.x there is support for textures in the sRGB color space or having an sRGB frame buffer. All hardware that implements OpenGL ES 3.x has a hardware accelerated Look Up Table (LUT) that allows performing conversion between a linear color space and a gamma based sRGB color space. This hardware implementation can be found also in some devices implementing the OpenGL ES 2.x standard as an extension with the identifier “GL_EXT_sRGB”.

As stated in the EXT_sRGB standard and OpenGL ES 3.x, “OpenGL assumes framebuffer color components are stored in a linear color space. In particular, framebuffer blending is a linear operation.”

Apart from the framebuffer information, the standard also states that ”Conventional texture formats assume a linear color space. So for a conventional internal texture format such as GL_RGB8, the 256 discrete values for each 8 bit color component map linearly and uniformly to the [0,1] range.”

OpenGL ES 3.x allows users to create a frame buffer that is interpreted as sRGB. The creation of the sRGB frame buffer is OS related. After creating an sRGB frame buffer we need to inform OpenGL ES that we are using an sRGB frame buffer so that it interprets properly the textures and writes back to the frame buffer as sRGB.

The sample code provided below shows how to create an sRGB texture and we inform OpenGL that we are using an sRGB frame buffer right before drawing. In addition, we are creating a texture with sRGB data and when constructing the texture we tell OpenGL that the data is in the sRGB color space.

```
#include <gl3.h>
...

GLuint texId;
glGenTextures(1,&texId);

// Bind the texture ID
glBindTexture(texId);

// Create a 2D texture and state it’s in sRGB color space
// internalFormat could be GL_SRGB8_ALPHA8 in case we have alpha
```
The diagram below illustrates graphically the steps that are executed by the hardware to perform the sRGB and linear conversions.

![Diagram of conversion dataflow with sRGB](image)

**Figure 10**: Conversion dataflow with sRGB.

Even for the latest hardware supporting OpenGL ES 3.x it can be convenient to use a different color space. When chroma sub sampling is used, blending shall take place in the shader since the hardware only supports the sRGB color space.

An applicable case of that mechanism is the Bayer Pattern (very common in CCD sensors) where the image format is not sRGB. This is out of the scope of this research paper; hence we will only mention it.

There are several popular formats that are using chroma sub sampling such as JPEG and many video codecs. The chroma sub sampling can be used also for cases where the developer wants to use a custom codec based on YCbCr or YCoCg. For these cases in order to speed up the conversion and/or decompression it is better to use a shader to perform the conversion as described in...
As described earlier in this section the linear blending would still be problematic. This is the reason for which the implementation discussed uses also gamma correction.

### 3.4.3 Convert from image to screen color space

This section combines the results of section 3.4.2 on OpenGL ES 3.x with the screen’s color profile. The last fragment shader executed when drawing the buffer on the display takes care of performing the conversion to the color profile of the output screen. In this section we will analyze this approach and provide different solutions. Here we are researching a method that dynamically adapts to the requirements of the hardware and using a shader generator. Using the shader generator we can reduce the execution time, as well as provide the flexibility to handle differently special cases depending on both screen and hardware. The step of hardware detection and shader generation has a static cost and can be automatically adjusted the first time an application is executed on some device. Since this is a static one time cost, it provides a high gain in terms of performance, quality and flexibility during execution.

Since providing a full implementation for the shader generator is out of the scope of this research paper, we are presenting the most common cases. These cases are trying to cover as many color profiles and not being tuned for specific screens. Using this path we can achieve a high performance and large coverage of displays.

The shaders provided are based on OpenGL ES 2.x and will not execute if mixed with OpenGL ES 3.x.

**Note:**

No previews showing the quality can be presented since the results are device dependent and can not be printed on paper.

For the cases that a transformation matrix is required in order to perform color correction we will use littleCMS for generating the matrix. This matrix is afterwards passed to the shader as a uniform variable and apply the correction at render time.

Section 5.1 of the Appending contains the sample code that shows how a transformation matrix and the gammut LUT can be generated through littleCMS. We have omitted the sections of the code that perform the checking providing only the code that generates a conversion from RGB to RGB.

For the implementation we present we had to modify the littleCMS plugin in order to grab the conversion information on the different steps of the system. By default littleCMS goes through a list several optimization plugins. While iterating through the list if a plugin succeeds to optimize the pipeline the optimization step is assumed to be finished. We introduced a new littleCMS plugin which was acting as a dummy optimization tool that was sniffing information from the pipeline without affecting the data. The problem faced was that out plugin was the first to be executed and we were lacking the matrix and gamma optimizations. The modifications that took place where related to getting the dummy plugin to execute after the optimization stages built into littleCMS.
Color adjustment lacking Gamma Correction

This approach uses a 3x3 matrix for performing linear adjustments to the output image. These adjustments are related to brightness, contrast and chroma. The shader code for that operation receives as an input a transformation matrix to perform the color change. In that case we are performing a color correction ignoring the gamma either because it is not necessary to take it into account or we have limited hardware resources. For the case that no gamma correction is required the result will match the color profile of the screen and be displayed as expected. For the case that gamma correction is required but omitted, the result will be improved but will not match perfectly the display’s color profile.

```glsl
precision mediump float;
uniform mat3 transform;
uniform vec3 offset;
uniform sampler2D screenTexture;

varying mediump vec2 v_texCoord;

void main()
{
    vec4 color;
    color = texture2D(screenTexture, v_texCoord);
    // Remove gamma curve with an approximation
    // to improve quality
    color.rgb *= color.rgb;
    // Add back the gamma curve
    color.rgb = sqrt(transform * color.rgb + offset);
    gl_FragColor = color;
}
```

Gamma Correction

Gamma correction is most of the time more important than a linear transformation. In many cases gamma correction is the only transformation required. Since most screens are based on sRGB color profiles, the only part that differs among them is their gamma value. This case covers the situation where only gamma correction is required and the case were performing just a gamma correction can improve the rendered outcome.

```glsl
precision mediump float;
uniform sampler2D screenTexture;
uniform vec4 scaleOffset;
uniform sampler2D lutOut;

varying mediump vec2 v_texCoord;
```
Color adjustment with Gamma Correction

In this case we are combining the two cases presented above. We perform both a gamma correction and a linear transformation to improve the quality. This case applies to hardware capable of performing both color correction linearly and gamma correction. This case is not always the optimal as stated at the beginning of the section since very often of the two transformations could be omitted.

```cpp
precision mediump float;
uniform mat3 transform;
uniform vec3 offset;
uniform sampler2D screenTexture;
uniform vec4 scaleOffset;
uniform sampler2D lutOut;

varying mediump vec2 v_texCoord;

void main()
{
  vec4 color;
  color = texture2D(screenTexture, v_texCoord);
  color.rgb *= color.rgb;
  color.r = texture2D(lutOut, vec2(color.r * 
    scaleOffset.z + scaleOffset.w, 0)).r;
  color.g = texture2D(lutOut, vec2(color.g * 
    scaleOffset.z + scaleOffset.w, 0)).g;
  color.b = texture2D(lutOut, vec2(color.b * 
    scaleOffset.z + scaleOffset.w, 0)).b;
  color.rgb = transform * color.rgb + offset;
  gl_FragColor = color;
}
```
Non Linear Color Adjustment using a 3D LUT

This section provides a solution to non-linear color transformations. For this case we are constructing a LUT and passing it to the shader in order to perform the color conversion. This 3D LUT is used for mapping the colors of the source color profile to those of the destination color profile. The code below is written in OpenGL ES 3 shading language. The code could be easily converted to OpenGL ES 2 shading language but its compilation will only be possible when the extension GL_OES_texture_3D is available which allows usage of 3D textures in OpenGL ES 3 [13]. This case is not very common since most screens are following the sRGB color space.

The shader below is written in OpenGL ES 3 and will not compile on hardware based on OpenGL ES 2.

```cpp
uniform sampler3D lut;
uniform sampler2D screenTexture;
uniform vec3 scale;
uniform vec3 offset;

// Computation outside the shader
// scale = vec3(1.0f) - (vec3(1.0f) / 
// vec3(textureWidth, textureHeight, textureDepth))
// offset = vec3(0.5f) / 
// vec3(textureWidth, textureHeight, textureDepth)

void main (void)
{
    // Grab the color value
    vec4 color = texture(screenTexture, gl_TexCoord[0].xy);
    // half-texel scale and offset
    color.rgb = color.rgb * scale + offset;
    // Here we are mapping the current color of the fragment to 
    // the color of the lookup table
    color.rgb = texture(lut, color.rgb).rgb;

    gl_FragColor = color;
}
```

Three different ways for performing color correction shown above are compared in detail in section [4]. Here we have presented different solutions that improve color quality with various performance results.
4 Discussion & Conclusion

In the previous sections we presented various scenarios and the results of our experiments. This subsection compares all these techniques and performance results giving a more concrete answer to the problem presented by this paper. Adding full color profile support on embedded devices is not a simple task since the OSs of embedded devices not not provide any direct implementations. It is an intense operation which needs to be applied on every frame. In order to achieve that goal it need to be properly optimized and the intense operation taking place for every frame shall take place on the GPU.

4.1 Comparison of Techniques

After researching different aspects of color profiles we noticed that color profiles are not always uniform and we need to have a dynamic way of identifying the requirements of each system. Depending on the result we need to apply different ways for performing the color conversion. In the upcoming section 4.1.2 we are suggesting the implementation of a library that detects the conversion required and the hardware adapts the appropriate implementation dynamically. Most screens base their color profile on sRGB whereas other output devices such as printers use custom color spaces and very often are not even RGB based. The aspect on these output devices which do not have real-time rendering requirements is simply mentioned but not researched in this paper.

The experiments conducted have shown that using solely a CPU real-time conversion can not be achieved on embedded devices. The optimal performance was measured when using the GPU. Depending on the hardware support we can choose either the solution provided in section 3.4.2 or the solution presented in section 3.4.3.

At the time that this research is conducted, the market of embedded and mobile devices is conquered by ARM processors based on Mali GPUs that implement OpenGL ES 2.x and are running on *NIX based OSs such as Android and Linux. We explored the capabilities of these devices and pushed to the limits the technologies supported by their GPUs. It has been shown that these devices are capable of performing the conversion and can fulfill the requirements of most embedded system devices related to color profiles.

The implementation using OpenGL ES 2.x has several drawbacks related to the quality of output. When there is sRGB support, the hardware performs the texture wrapping and interpolation in the texture as soon as the sRGB texture is converted to linear when fetching. For the case of OpenGL ES 2.x where we are using the gamma 2, the texture is assumed to be linear and the interpolation and wrapping takes place linearly. OpenGL ES 2.x does not provide the flexibility to change the interpolation and wrapping of textures which causes a quality loss when blending and converting to the screen color profile.

Looking at the current demand of the market, as well as the quality, performance and portability of this implementation we can consider it the optimal solution since it covers most aspects. We were to implement an OpenGL ES 3.x solution which was ran on desktop systems since the OpenGL ES 3.x embedded devices are not yet available on the market. According to the specifications and the added support for the sRGB color space we assume that the hardware acceleration will provide a performance boost. This is an assumption and in order to
be verified the code we provided in the previous section shall be benchmarked on compared with the output of OpenGL ES 2.x devices.

4.1.1 Comparison of Benchmarks

For the benchmarks reflecting our implementation of color profiles on embedded devices we tried it on the Nokia N9 which was the most powerful hardware available to us. Our implementation has two different approaches, in the first approach we are performing a conversion from an AdobeRGB color profile to tan sRGB. The second approach converts the input image into the Gamma2 color space which is later converted to sRGB while rendering.

![Diagram showing performance comparison](figure11.png)

Figure 11: Gamma2/sRGB to Any vs Any to Any Benchmark.

From the diagram above we can see a performance boost of 30% which is related to the usage of Gamma2. When using Gamma2 we have a performance boost due to the fact that special instructions can be used since we are using multiples of two. Looking at the two bars it can be observed that we still can not reach a real-time rendering system on this hardware (best performance 0.037 seconds per frame which results in 28FPS).

Comparing the results the conversion on the CPU and GPU we have proven that the conversion using a GPU is 10x faster for the worst case of the color conversion. The CPU is still very important in our pipeline since it performs the initial conversion and helps preparing the attributes passed to the GPU.

4.1.2 Future Work

Future work refers to usage of this research paper to provide a solid implementation of color profiles for embedded devices and desktop systems. In addition, it provides information so that further research could be conducted in order to combine the outcome of the research paper with other fields.
The previous sections provide a concrete research background for performing color correction and blending. The implementations described above based on OpenGL ES 2.x and 3.x could be combined. This combination shall be included in a graphic engine that is capable of performing hardware detection and proper color blending. In that case the cross-platform library shall be capable of choosing the appropriate implementation according to the extensions supported on the hardware. In addition to the extensions supported, the library shall be capable of detecting the type of conversion required after getting the destination color profile (color profile of the screen). This operation shall take place once and can be executed the first time an application launches. This way all the required shaders could be generated and recycled every time the application runs. The fact that embedded devices are bind to their hardware makes it easier to have the hardware detection taking place once and be stored in the preferences of an application. It is almost impossible to change the display of an embedded device with a piece of hardware that has different layout and/or different communication profiles.

The sample code we provide in this paper is targeting only RGB color spaces due to lack of time and market interest. The solution of the cross platform library presented in this section requires the implementation of a system that chooses the appropriate shader according to the conversion. Taking this idea one step further we can create a shader generator system that generates a custom shader depending on the needs of the conversion. By providing such a system we add flexibility and support for other color spaces such as CMYK and XYZ. Looking at the implementation of the previous section, there will be some changes required in the "shader_inspector()" function related to table sizes and the parts that checks the conversion.

Taking this implementation one step further, support for DirectX could be added in order to allow windows embedded devices. For OpenGL devices the OpenGL ES implementation could recycled since most of the hardware that implements OpenGL 3 and above supports the extensions GL_ARB_ES2_compatibility and GL_ARB_ES3_compatibility. This extension allows executing natively OpenGL ES code [17][18]. In order to add more features related to desktop systems a new implementation based on OpenGL could be implemented but that would require more human resources and research.

Finalizing Color Profile Conversion and correct blending in [Qt]

The current state of color profile conversion on [Qt] is very limited. My contribution on the project was to simply allow converting using the [CPU] through the LittleCMS library among different color profiles. Considering the previous section where the development of a shader based color conversion tool is suggested, [Qt] will be able to integrate this part in its graphics stack and successfully accomplish color conversion on every frame.

[Qt] is using OpenGL for rendering all the Graphical User Interface (GUI) related components; hence it is easy to integrate such a conversion mechanism to take place within [Qt]'s rendering pipeline. [Qt] 5 is built with the ability to dynamically generate shaders depending on the hardware. This gives the ability to easily inject the color correction shader code generated through the library de-
scribed in section 4.1.2. Injecting new shader code reduces the amount of render operations required but has the drawback of increasing the shader size. Apart from the final color conversion part, all input graphics shall be in the same color profile and shall be compatible with linear color blending. To achieve that we shall use the mechanisms described in section 3.4.2.
5 Appendix

5.1 Appendix A: LittleCMS Plugin Code

```c
// This function grabs the the lcms pipeline the values
// we are interested in

int shader_inspector(cmsPipeline** Lut,
                     cmsUInt32Number Intent,
                     cmsUInt32Number* InputFormat,
                     cmsUInt32Number* OutputFormat,
                     cmsUInt32Number* dwFlags)
{
    cmsStage* stage;
    int i, j, index;

    shader_flags = 0;
    ...

    stage = cmsPipelineGetPtrToFirstStage(*Lut);

    while (stage != NULL)
    {
        if (cmsStageType(stage) == cmsSigIdentityElemType)
        {
            stage = cmsStageNext(stage);
        }
        else
        {
            break;
        }
    }
    ...

    if (cmsStageType(stage) == cmsSigCurveSetElemType)
    {
        _cmsStageToneCurvesData* stageToneCurvesData =
        (_cmsStageToneCurvesData*)cmsStageData(stage);
        ...
        for (i = 0; i < 3; ++i)
        {
            for (j = 0; j < 256; ++j)
            {
```
int temp = stageToneCurvesData->nCurves - 1;
index = MIN(i, temp);
shader_lut_in[j][i] =
cmsEvalToneCurveFloat(
  stageToneCurvesData->TheCurves[index],
  j / 255.0f) * 255.0f + 0.5f;
}
}
shader_flags |= 1;
stage = cmsStageNext(stage);
while (stage != NULL)
{
  if (cmsStageType(stage) == cmsSigIdentityElemType)
  {
    stage = cmsStageNext(stage);
  }
  else
  {
    break;
  }
}
if (stage != NULL)
{
  ...
  if (cmsStageType(stage) == cmsSigMatrixElemType)
  {
    _cmsStageMatrixData* stageMatrixData =
      (_cmsStageMatrixData*)cmsStageData(stage);
    ...
    for (i = 0; i < 9; ++i)
    {
      shader_matrix[i] = stageMatrixData->Double[i];
    }
    if (stageMatrixData->Offset != NULL)
    {
      for (i = 0; i < 3; ++i)
      {
        shader_offset[i] =
          stageMatrixData->Offset[i];
      }
    }
  }
}
else
{
    for (i = 0; i < 3; ++i)
    {
        shader_offset[i] = 0.0f;
    }
}

shader_flags |= 2;

stage = cmsStageNext(stage);

while (stage != NULL)
{
    if (cmsStageType(stage) == cmsSigIdentityElemType)
    {
        stage = cmsStageNext(stage);
    }
    else
    {
        break;
    }
}

if (stage != NULL)
{
    ...

if (cmsStageType(stage) == cmsSigCurveSetElemType)
{
    _cmsStageToneCurvesData* stageToneCurvesData =
        (_cmsStageToneCurvesData*)cmsStageData(stage);

    for (i = 0; i < 3; ++i)
    {
        for (j = 0; j < 256; ++j)
        {
            int temp = stageToneCurvesData->nCurves - 1;
            index = MIN(i, temp);
            shader_lut_out[j][i] = cmsEvalToneCurveFloat(
                stageToneCurvesData->TheCurves[index],
                j / 255.0f) * 255.0f + 0.5f;
        }
    }
}

shader_flags |= 4;
stage = cmsStageNext(stage);

while (stage != NULL)
{
    if (cmsStageType(stage) == cmsSigIdentityElemType)
    {
        stage = cmsStageNext(stage);
    }
    else
    {
        break;
    }
}

return FALSE;
Glossary

Symbols

| Symbols | A | B | C | D | G | H | L | Q |

Symbols

ICC Color Profile is the International Color Consortium Color Profile standard. 8, 9, 29

A

alpha channel is the color channel referring to the transparency of a pixel. 9

B

blending is the mixing of different pixel colors. 29

C

Cg (C for Graphics) is a shading language designed by Nvidia that can be easily compiled for DirectX and OpenGL. 29

D

DirectX is a Windows specific graphics library for 2D and 3D rendering. 27

G

GLSL (Open Graphics Library Shading Language) is the OpenGL Shading Language. 29

H

HLSL (High Level Shading Language) is the DirectX Shading Language. 29

L

linear blending is the blending that assumes that the alpha channel is linearly indexed. 29

linear color space is a color space where the color intensities are increasing linearly for all color channels (ie. Red, Green, Blue and Alpha). 6, 13, 15

LittleCMS is a powerful Open Source library that allows performing color profile conversion using the CPU. 8, 9, 27, 29

O

OpenGL (Open Graphics Library) is a library that allows cross-platform development of for 2D and 3D rendering. 13, 15, 27, 29, 30
OpenGL ES [OpenGL] standard for embedded systems. 3 23 25 26 29

Q

Qt is an OS framework that allows cross-platform development for both desktop and embedded systems. 1 4 27 29

S

shader a program that runs on the GPU and performs graphics related operations when rendering on a buffer. 17 27 29 30

shader generator is a tool that allows combining different shader programs, in order to execute them in one drawing cycle. 17 27 29
References


