Color Characterization and Calibration of an External Display

Andrew Crocker, Austin Martin, Jon Sandness
Department of Math, Statistics, and Computer Science
St. Olaf College
1500 St. Olaf Avenue, Northfield, MN 55057
crocker@stolaf.edu, martinaj@stolaf.edu, sandnesj@stolaf.edu

Abstract

A new real-time color correction technique is presented. Our method attempts to characterize inconsistencies in color representation of a monitor, and corrects them by modifying the input to account for color distortion. Correction is achieved by use of a per-pixel shader that alters the value of every pixel drawn to the screen in real-time. Our method relies on a precise mapping from a camera image to the display screen, created using a method analogous to a binary search. Our method produces a generalized per-pixel color correction function for the display under specific lighting conditions. Each pixel is changed to reflect a known value captured by the camera in the camera-display system. Careful measurements and interpolation gives us a complete function from desired output to necessary input. These results provide an important proof of concept that will require further investigation and refinement. Mapping, calibration, and color correction techniques are presented.

Introduction / Problem Statement

Color displays are everywhere: TVs, monitors, cell-phones, etc. Although they are widely available, their usefulness depends in part on the accuracy of their image reproduction. If the user wants an image of their calendar and instead receives an image of a giraffe, it is of little use. Similarly, if the user wants an image of a red fire truck and that truck is displayed, but its color is yellow - that image is less useful. By characterizing the color inaccuracies between a source image and the display, we can modify the input image in order to produce a more accurate output image. For instance: If we can tell that the screen is displaying more red than it is supposed to, we can modify the input image to have less red, thereby correcting the output. Our method attempts to describe the RGB differences between input/output images on a per-pixel basis, allowing finely-grained control over the color scheme displayed to a screen. It is to be used to for real-time color correction of a walkthrough 3D model displayed on a television.

Background / Previous Work

To the extent of our knowledge, this is a novel approach to an old problem. There have been many attempts to improve the process of color calibration, but these methods all make one global correction to account for inaccuracies within the display itself. The difference with our method is that adjustments are made to each pixel on an individual basis in order to account for inaccuracies external to the display, such as reflections. Existing methods also tend to directly modify the monitor settings to adjust color, rather than modify the image being displayed. While the previous work in this field solves a somewhat different problem, some of the techniques we used in our solution have been done before. The binary striping technique used to create our mapping comes from the work of Olaf Hall-Holt and Szymon Rusinkiewicz in a paper entitled *Stripe Boundary Codes for Real-Time Structured-Light Range Scanning of Moving Objects* [2]. Additionally, we made use of a freely distributed software package called gPhoto2 to control our camera remotely during the calibration process [1].

1 Calibration

In order to describe the color difference between an image and its TV representation, we need to be able to compare the input image with the light that is physically produced by the display. More precisely, we need a way of quantifying the actual color output of each pixel on the display so we can make a per-pixel comparison between the input image and the output image to determine whether local adjustments need to be made. Accurately quantifying each pixel's output depends on our ability to locate an individual pixel from the display in an image taken by a digital camera. This is done by creating a precise mapping from the source image region to a picture that contains this source image. (See Fig. 1)

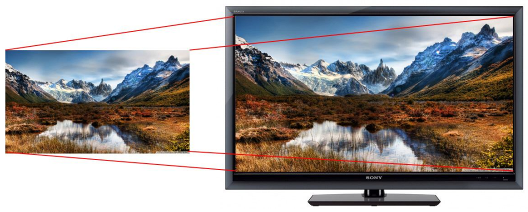
Figure 1: A depiction of the mapping concept, where an image is taken of the object on the right, and the active screen region is identified.

We call this mapping process *Calibration*. Calibration involves taking pictures of the display and extracting data from those pictures. The first step is to isolate the screen region. We use a camera in a fixed location with all exposure/brightness settings also fixed. First, we display a white screen and take a photo, followed by a black screen and another photo. Calculating the difference between the two photos tells us which pixels have changed significantly from one image to the next. The area of the image that changes represents the TV screen. Note: It is imperative that the camera remain in the same position throughout the calibration and color correction process.

After the boundary region of the TV has been identified, we can begin to map individual pixels.

1.1 Recursive Striping & Pixel Mapping

The detailed calibration process involves taking pictures of alternating black and white stripes displayed onto the television. The width of each vertical stripe and height of each horizontal stripe is a power of two. A picture is taken of each image as it is displayed to the screen.

We begin by splitting the screen approximately in half with a black stripe 2^10 pixels wide on the left side and white on the right, then a picture is taken. The screen is subdivided again so that the stripes are 2^9 pixels wide, and the process repeats, each time halving the width of the stripes. It ends when the width of the stripe is a single pixel. The reason for this process becomes more clear by following one pixel, as shown in Figure 2.

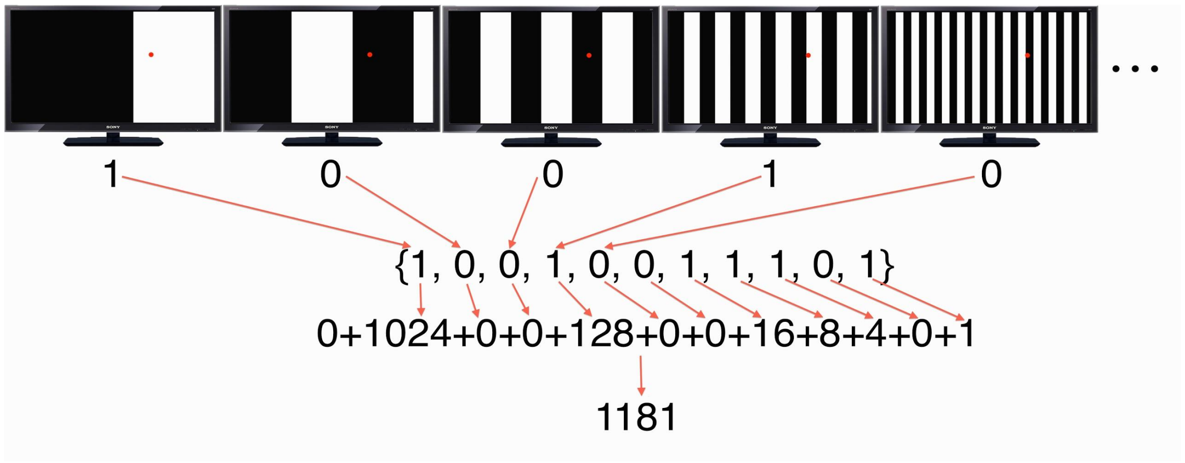

Figure 2: An example of the calibration process, where successively smaller stripes are placed on the screen and photographed. The location of each pixel is found by resolving the binary address {1,0,0,1,0,0,1,1,1,0,1} into a decimal location.

The initial location of our pixel is set to 0. In the first image, we see that our pixel lies in a white region, so we add the width of the stripe (1024) to the location that we are calculating. In images 2 and 3, the pixel is in a black region so we don't add anything to our calculation. Image 4 shows our pixel in white again, so we add the stripe width (128) to our cumulative sum. Every time the pixel in question falls within a white region, we simply add the stripe width to our calculation. When we have processed all of the images with vertical striping, then we have finished calculating the horizontal location of the pixel. Then, the process is repeated with horizontal stripes to calculate the vertical location. The resulting ordered pair (horizontal location, vertical location) is the mapped location from camera to screen.

By finding this mapping for each pixel in the camera image, we end up with a *1* : *Many* map of [*1920x1080 Source Image Pixels*] : [*5184x3456 Camera Pixels*]. So each display pixel maps to 0-N pixels in the camera image. Therefore, if we choose a random pixel from our display, say (1181, 734), we hope to find between 4 and 9 camera pixels mapping to this exact location.

1.2 Map Refinement

Once we have a complete map, it is important to remove any erroneous data points so that the map is accurate. Because our method of resolving screen location relies on the black/white color of a given pixel as interpreted by a camera, it is error prone. Imagine the binary addresses 1001 and 0001 from the calibration process. There is only one different black/white value between these two, but they are ScreenWidth/2 distance from each other. This means that getting a single black/white value wrong can change the mapped location to the opposite end of the screen. With data that is this fragile, it's important to remove any inaccurately mapped points. In an attempt to resolve the poorly mapped areas we assume that a clump of nearby pixels is correct, and that any pixels greater than one standard deviation from the average of the distances between all the pixels in that clump should be removed. (See Fig. 3) Removing these outliers leaves us with a very accurate map, explained further in the Results section.

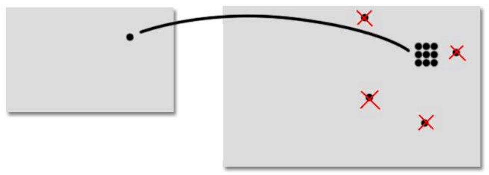

Figure 3: An illustration of the removal of extraneous pixels from the mapping.

2 Color Correction

Our ultimate goal is to have our monitor replicate the exact light conditions captured by the original camera. In order to present the correct color to the screen, we must understand what the screen does to the colors it displays. Distortions in color displays can come from a few different sources including hardware limitations, built in image processing software, and environmental factors such as the presence of ambient illuminants. To compensate for the latter two when considering a single image, the following iterative correction method was developed.

2.1 Iterative Approximation for Single Images

Our correction process attempts to use mapped camera feedback compared with a reference image which we call a *gold standard* (GS) in order to approximate the correct color values to output to the display. Our method relies on a starting *modified image* (MI) in which all values are set to RGB (0,0,0). A picture is taken of the display, and using the mapping created during calibration, we reconstruct the image in the same dimensions as the monitor. We call this a mapped image (CI_{Mapped}), and we compare it to our GS in order to obtain a difference map between the two. We scale our difference map by a damping coefficient of .30 to avoid overshooting our end goal. This difference map is like a filter which we then apply to our MI_i to obtain MI_{i+1}. This iterative process stops when our filters can no longer adjust the image. Figure 4 gives a visual explanation of the process.

Results of our tests proved quite satisfactory with average per-pixel color difference of 1.4 on a pixel range of 0-255. In certain regions, the color difference was as high as 60 suggesting that the display hardware was not capable of displaying the exact values we sent it. These results suggested that additional work would need to be done to generalize our correction process for all images.

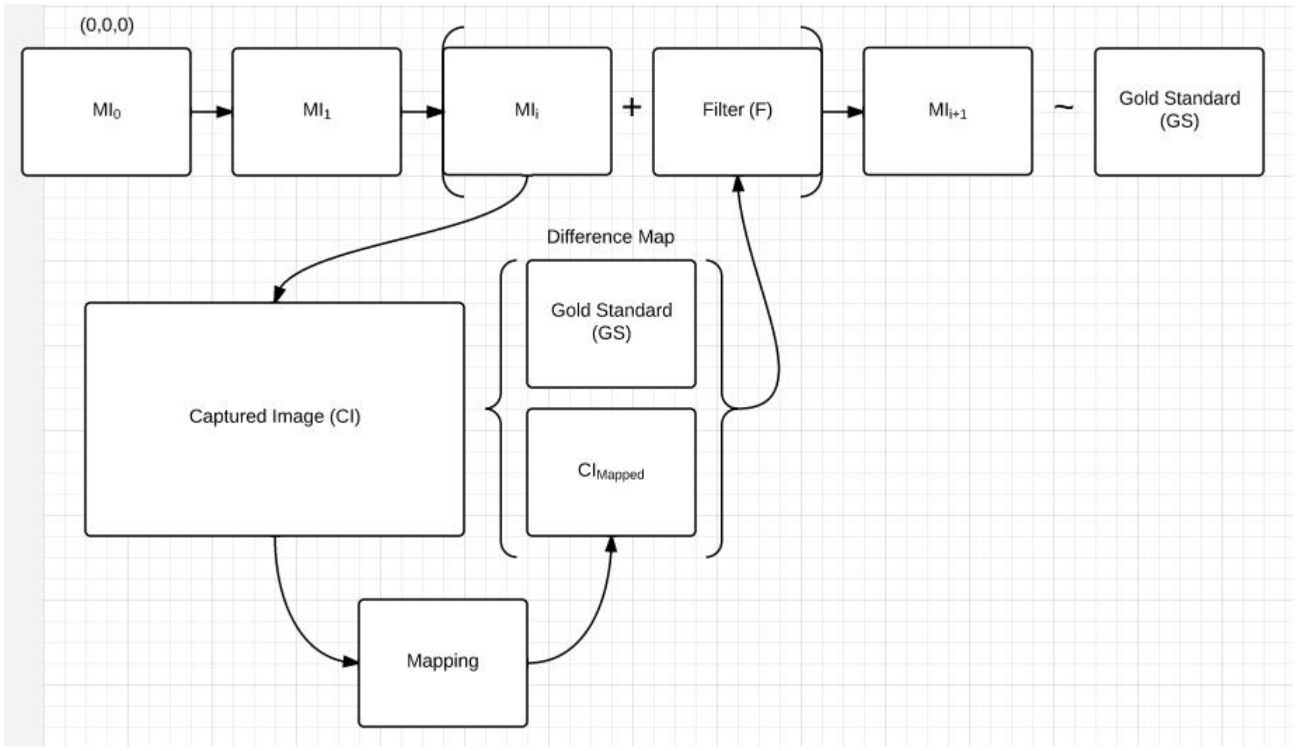

Figure 4: An expanded view of the iterative correction process shows each step involved. An image is captured with a camera, and using the mapping from earlier it is reconstructed in the display's dimensions. It is compared with the *Gold Standard* and added to the starting image.

2.2 Color Characterization

To characterize how our screen modifies color and generalize our iterative process and move toward a real time correction method, we observe the behavior of sample color values displayed on-screen. Specifically, we sample designated color swatches along the spectrum of each isolated RGB channel. To generate 256 values for each of the Red, Blue, and Green channels, we display an image that looks similar to what's shown in Figure 5 for each of the R, G, and B color channels, but further subdivided into 20 evenly spaced squares rather than 9:

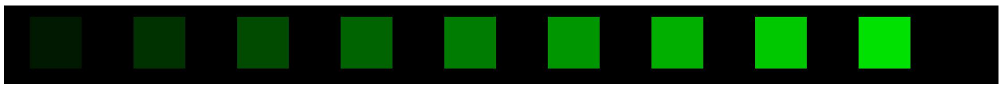
Figure 5: A simplified version of the discrete color points sampled in performing Color Characterization.

Each row of colors represents intensities from 0-255 of its respective channel and each color square represents 19/256 or 7% of its channel. This method of color sampling gives us data points for specific values on each channel's spectrum, but we need approximations for *every* value between 0-255. To recover these missing values, we linearly interpolate between each known discrete value. What we end up with is three separate color channels with intensities from 0-255.

2.3 Color Cube Mapping

Color Characterization shows how each RGB channel behaves on its own, but this only helps us if our image is monochromatic. Approximating the output color for RGB values like (160,60,200) requires that we again interpolate between our three individual channels to create a full-spectrum of color that we call the color cube. For example, to estimate the output RGB that the monitor will produce for the input (160,60,200), we add together the output RGB values for R=160, G=60, and B=200. We end up with an expected output of (234,140,237) for the input (160,60,200), as shown in Table 1.

		R	G	B
R	160	160	30	20
G	60	4	60	17
B	200	70	50	200
	Sum:	234	140	237

Table 1: A table showing how the estimated output is calculated for a given input. Each row gives the color produced in each channel for a given input.

By doing this additive operation for every value of Red, Blue, and Green between 0-255, we can associate every color's input with a given output. We then create an inverse color cube that allows us to look up the calculated input for a desired output. So, if we want the monitor to display the color (234,140,237), we can look up that point in our inverse color cube to see that it is produced by input (160,60,200). There are inherent errors in this process though. One issue is that multiple input RGB values can map to a single output value. It's also possible for the additive operation to produce a color beyond the (255,255,255) spectrum. Because of this, there are holes in the inverse color cube where a desired output has no known input. Figure 6 below shows two cross sections of our color cubes.

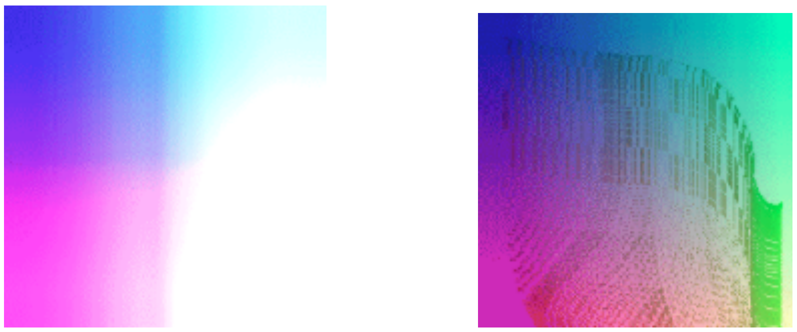

Figure 6: A cross section of the color cube created from our interpolation is shown on the left, and on the right is a cross section of the inverse color cube, where the irregular looking values are points that have data associated with them.

Our preliminary work gave us an inverse cube only filled to 13.9 percent capacity. Having color cube holes is undesirable because it leaves us guessing any time we want a color that doesn't appear in our cube. Our current strategy for filling the unknown values gives us a rough approximation. If the pixel (25,25,25) does not have a mapping, we find the nearest neighbor with a value and copy its value. In this way we completely fill the color cube while retaining approximately accurate results.

2.4 Filter Creation

Every step so far has been completed with the goal of creating a filter that can make a displayed image as close to the original as possible. Because the color cube is a mapping from any RGB value in the real life color space to any RGB value in the TV color space, we can invert this mapping to color-correct a display. This is more easily understood with an example: If a pixel's true color is (25,25,25) and we want to display that color to the screen, we can check our inverse color cube and find (25,25,25) then see what input value produced that color. This allows the color cube to act as a filter between input values and their appropriate on-screen representation. (See Fig. 7)

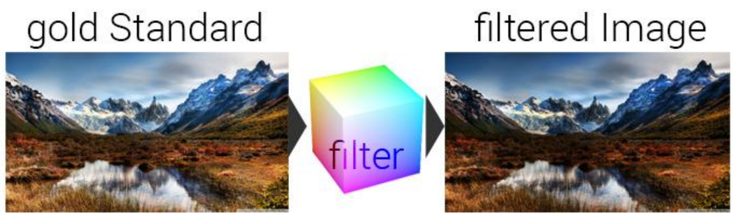

Figure 7: A depiction of an image before and after the color correction process.

An OpenGL shader is capable of acting as the filter in order to provide real-time color correction to any screen. To achieve this, we let the common OpenGL render loop run to completion, stopping just prior to the glSwapBuffers command. At this point the screen buffer holds all pixel values that would normally be displayed on-screen. By iterating across these pixel values and performing the inverse lookup operation on our color cube, we can create a 1:1 texture map of all corrected pixel values. The OpenGL render loop is then re-run from its beginning, but this time our fragment shader is applied. The texture map is passed to the fragment shader and applied to each fragment that runs through it, color correcting the image's pixels.

3 Error

There are several known sources of error within our method. The first of these is an imperfect mapping which may be caused by a number of factors such as camera movement during the calibration process. A more likely source of error is pixel noise within the camera itself. If the camera records a certain pixel as being black when it should have been white, we could end up with an erroneous pixel in our mapping.

Another issue with our mapping was caused by the TV that we were using. When displaying single-pixel-wide stripes, certain regions of the screen were unable to properly resolve the image, possibly due to unshielded electrical components within the TV.

A second source of error is introduced when we perform the linear interpolation between our known data points. We would expect intermediate points to follow some normal curve, but that may not be the case. Therefore, it is conceivable that our additive method of interpolation gives inaccurate values. This error could be mitigated by capturing additional images of combined color channels to increase the number of known data points.

Color bleeding is one of the most difficult sources of error that we faced because it makes it difficult to determine what color is actually being displayed by a given pixel. For example, if a black pixel is displayed directly adjacent to a white pixel, the black pixel will appear more blue than black. The same issue happens with other colors as well. At this point, we are unsure of any viable method for addressing color bleeding.

One last issue with our method involves reflections on the screen surface. Our iterative refinement process accurately accounts for reflections, but unfortunately the process is limited by the brightness of the reflection. If the reflection is too strong, it becomes impossible to modify the input to account for the reflection.

4 Results

We achieved significant results in several components of our work. The first notable result is our calibration process which created a mapping from a camera image to the monitor. On average, poorly mapped pixel values represented 22% of the total pixels in the original mapping. The mapped pixel clumps measure 4-10 pixels in size, with 99% of them lying between 4-10, and 75% within 4-6 pixels.

After we generated a mapping, we used an iterative process to re-create a single image as displayed to the screen. A comparison of our resulting image to the desired image showed an average pixel difference of 1.4 on a scale of 0-255. In certain regions of our reconstructed image, the color was off by up to 60 points.

The next part of our process is creating a color cube and its inverse. Our initial attempts at creating an inverse color cube gave us a cube that was 13.9 percent complete. We hope that additional work on this component of the process will produce better end results.

Lastly, we created an OpenGL shader to modify all inputs based on the results from our inverse color cube. The shader works as desired, but without accurate information in our color cube, the results are not visually appealing.

5 Real World Applications / Future Work

The applications of a real-time environment sensitive color calibration software are numerous. The aforementioned mapping and calibration process serves as a proof of concept that this sort of fine image adjustment can happen not only at a per-pixel level, but with a degree of detail that compensates for two of the three possible causes of color distortion, namely additional illuminants and software image processing unique to the display. This bodes well for future work into the field of real time color calibration. Our method's ability to cheaply characterize screen reflection in a camera-display system suggests that more work in this area could improve an algorithm for adjusting to ambient screen reflection in real time. Furthermore, our method, if improved upon, has the ability to do real-time screen color correction based on any combination of RGB channel intensities within the OpenGL pipeline via a simple calibration process which could lead to more intricate algorithms in color enhancement. More work will need to be done to refine and speed up our current process.

6 Conclusion

A new method of real time color calibration is presented based on iterative camera feedback. Our method relies on a precise screen mapping and interpolated color cube to define a function by which we can pass any image and receive a filtered image back. The filtered image is adjusted based on the color characterization of the monitor as documented in the color cube. By looking up the values within the color cube, a constant time operation, our method has the potential to be used in real-time scenarios where per-pixel color correction is desired.

References

[1] *gPhoto2 Digital Camera Software.* Vers. 2.5.4. Computer Software. gPhoto2, 2014. Multi-Platform, 6.7MB, Free Software.

[2] Hall-Holt, O.; Rusinkiewicz, S., "Stripe boundary codes for real-time structured-light range scanning of moving objects," *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on* , vol.2, no., pp.359,366 vol.2, 2001