SmartColor – Real-Time Color and Contrast Correction for Optical See-Through Head-Mounted Displays

Juan David Hincapié-Ramos, Levko Ivanchuk, Srikanth K. Sridharan and Pourang P. Irani

Abstract—Users of optical see-through head-mounted displays (OHMD) perceive color as a blend of the display color and the background. Color-blending is a major usability challenge as it leads to loss of color encodings and poor text legibility. Color correction aims at mitigating color blending by producing an alternative color which, when blended with the background, more closely approximates the color originally intended. In this paper we present an end-to-end approach to the color blending problem addressing the distortions introduced by the transparent material of the display efficiently and in real time. We also present a user evaluation of correction efficiency. Finally, we present a graphics library called SmartColor showcasing the use of color correction for different types of display content. SmartColor uses color correction to provide three management strategies: correction, contrast, and show-up-on-contrast. Correction determines the alternate color which best preserves the original color. Contrast determines the color which best supports text legibility while preserving as much of the original hue. Show-up-on-contrast makes a component visible when a related component does not have enough contrast to be legible. We describe SmartColor's architecture and illustrate the color strategies for various types of display content.

Index Terms— H.5 [Information Interfaces and Presentation]: H.5.1—Multimedia Information Systems - Artificial, Augmented, and Virtual Realities; H.5.2 - User Interfaces - Ergonomics, Evaluation / Methodology, Screen Design, Style Guides

1 Introduction

A major usability challenge for optical head-mounted displays (OHMD) is that users perceive color as a blend of the display color and the background [4]. For example, users perceive red as magenta when the background is blue, and as yellow when the background is green. Color blending has several negative effects on the display content: it reduces the readability of text [18], makes augmented reality (AR) objects look translucent and unreal [1], impairs the interpretation of color-encoded information such as status icons or system alerts, and leads users to confusion [19]. Therefore, addressing color blending is an important issue as it compromises the usability and adoption of OHMDs [10][11].

Existing solutions to color blending include occlusion support and color correction. The occlusion approach uses spatial light modulators such as liquid crystals to block light coming from the background and keep it from blending with the display color [3]. Occlusion support requires additional optical hardware, making the OHMD bulky [3][5] and compromising its transparency [14]. The color correction approach consists of determining an alternative color which, upon blending with the background, results on a color closer to the one originally desired. Color correction does not require additional optical hardware and therefore does not affect the overall transparency and lightness of the display. However, existing color correction algorithms are not usable: Weiland et al. proposed several correction algorithms which often result in black or depend on arbitrary constants [24], while Sridharan et al.'s binned profile search does not consider the effect of the display material and is not computable in real time [22].

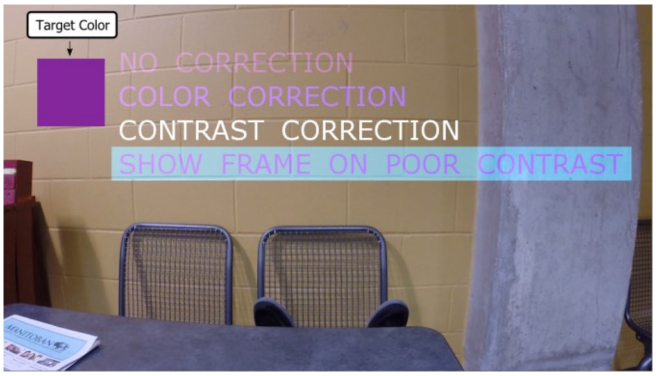

Fig. 1. SmartColor strategies for text labels (best seen in color).

In this paper we present three main contributions. First, we complete Sridharan et al.'s work and present a) a linear approach to describe the way a display material affects background colors, and b) QuickCorrection, a real-time color correction algorithm based on display color profiles. Our algorithm avoids exhaustive search by understanding the problem as a search in the three dimensions of the CIE LAB color space. We present two implementations of our algorithm for the OpenGL platform and study their accuracy and performance.

• J.D. Hincapié-Ramos is with the University of Manitoba, E2-509 EITC, Winnipeg, MB, Canada, R3T 2N2. E-mail: jdhr@cs.umanitoba.ca.
• L. Ivanchuk is with the University of Manitoba, Canada. E-mail: umivanch@cc.umanitoba.ca.
• S.K. Sridharan is with the University of Manitoba, Canada. E-mail: kirssri@cs.umanitoba.ca.
• P.P. Irani is with the University of Manitoba, E2-580 EITC, Winnipeg, MB, Canada, R3T 2N2. E-mail: irani@cs.umanitoba.ca.

Our second contribution is a user evaluation of the accuracy of QuickCorrection. Our results show that users more often identified the corrected colors as being closer to the target color.

Finally, our third contribution is SmartColor, a user-interface framework aimed at mitigating the effects of color blending on user-interface components according to three color management strategies: correction, contrast, and show-up-on-contrast (see Fig. 1). In correction mode, SmartColor executes the real-time color correction algorithm aiming at maintaining color encodings. In contrast mode, SmartColor limits the three-dimensional search to include only the colors that provide a difference of 27 units of luminance required for text readability [25] while preserving as much of the original hue as possible. Finally, the show-up-on-contrast mode makes content visible only when the contrast of related content is compromised by the background. We describe the SmartColor architecture, present different configuration options, and illustrate the color strategies for various types of content.

2 Existing Solutions to Color-Blending

Ad-hoc solutions to color blending include users placing their hand in front of the display or directing their head to darker backgrounds [18]. This section presents hardware and software solutions to manage color blending.

2.1 Hardware-based Solutions

Hardware-based solutions are normally referred to as occlusion support in the OHMD literature. Occlusion support refers to situations in which virtual objects occlude real-world objects. The counterpart, mutual occlusion, includes cases in which real-world objects can also occlude the virtual object.

To provide occlusion support, the basic approach consists of augmenting the optical element of the OHMD with the capacity to block background light at the pixel level. This effect is achieved using spatial-light modulators (SLM) [1][5][8]. For each point on the display, the graphics driver generates the colored pixels which are to be shown to the user and an equivalent opacity mask which is applied to the background light. The opacity mask effectively blocks the background colors and leaves the colored pixels on a black background. This approach uses optical elements to redirect the background and display lights through the appropriate optical pathways. This strategy requires that these systems get to be large and impractical for everyday use [1][8][9]. Gao et al. achieved smaller sizes using curved lenses [5][6].

Maimone et al. [14] presented an alternative without the need to manipulate optical pathways. Their smaller device used stacked LCD panels and a shutter in two consecutive modes. In the first mode, the shutter blocks the entire background light while the LCD shows the colored content by filtering white light from an embedded backlight source. In the second mode, the shutter lets background light through and the LCD blocks it on the pixels where it previously showed content. By quickly switching between the two modes users get the illusion of occluded virtual content. This approach is promising and its realization depends on the availability of high frequency (>480 Hz) and highly transparent LCD displays. Smithwick et al. built a closely related solution for window-size transparent displays [21].

2.2 Software-based Solutions

Software-based solutions seek to change properties of the digital content in order to minimize color blending, particularly for text readability. A captured image of the background scene is analyzed. A simple solution is to increase the intensity of the digital text (mentioned in [8]) according to the background objects in the vicinity; however the effectiveness of this solution is limited when backgrounds are bright. A system can also capture the user's background, classify it into zones in which digital text would be readable or unreadable, and relocate content to the readable regions [12]. Automatic relocation can take into account restrictions such as ordering of components [23].

Another alternative is to compensate the changes in color by means of color correction. Researchers have developed color correction techniques largely focused on projection systems for spatial augmented reality (SAR) applications [15]. For example Bimber et al. explored color correction in projection-based transparent-display showcases [1] using surface radiance measures, 3D models of the world and the possibility to change the real-world illumination. OHMDs do not have all these capabilities and therefore the projection-based approaches do not apply.

Little research exists into how color correction applies to OHMDs and transparent displays in general. Weiland et al. generated four color correction algorithms for transparent displays: trivial (as named by Weiland et al.), weighted, simple smooth and advanced smooth [24]. All these methods use a digital camera to capture the background image and map it to the pixel coordinates of the transparent display. The trivial method removes the background RGB values from the foreground image. Weighted also performs background subtraction but with user-defined weights for the background and foreground. Simple smooth introduces luminosity of the background image as a limit to the subtraction. Finally, advanced smooth extends the simple smooth by adding user defined weights for the background and foreground. The main problem of all these methods is that they operate on RGB space, which is a color space created to store and represent color but has little relation to the way colors interact or are perceived by the human visual system, as shown empirically in [22]. Therefore, subtraction of RGB components often results in undesirable black color.

3 Understanding Color Blending

In our work, we understand color blending as the addition of the display and background colors after being subjected to two distortions – render and material – introduced by the display (see Fig. 2). The render distortion accounts for the particular way a display shows a given color (Dc – display color). The material distortion accounts for how the transparent display alters a background color (Bc) as it passes through the transparent material. Equation 1 defines the blended color as the addition (in the CIE XYZ color space) of the colors resulting from the two distortions.

$$Blend(Dc, Bc) = f_{render}(Dc) + f_{material}(Bc) \quad (1)$$

Equation 2 presents the general definition of color correction for a display color Dc on background Bc. The correction color Xc, is that color which blend with Bc is at the minimum distance to the desired display color Dc.

$$Correct(Dc, Bc) = \\ Xc \mid min(distance(Blend(Xc, Bc), Dc)) \quad (2)$$

The particular way in which the render and material distortions are accounted for defines the particular approach to the correction algorithm, that is, the *min* function of equation 2. For example, Sridharan et al. showed that the render distortion can be accurately addressed by measuring the way a particular display shows a representative set of the display colors [22]. This subset of colors is called a binned display profile and it was shown to accurately predict color blending using the CIE XYZ and LAB color spaces for color addition and perceptual difference respectively. However, Sridharan et al. bypassed the material distortion by using colorimetric measurements of the background through the see-through display.

In order to evaluate the *min* function, Sridharan et al. proposed a best-fit approach where all colors in the display profile are evaluated. A best-fit algorithm does not scale to real-time systems in which the profile resolution needs to be much higher (Sridharan et al. used only 8390 discrete bins, which makes for poor color resolution) and millions of corrections are required every second. Nonetheless, this algorithm provides a ground truth for what the best alternative color is. We refer to this correction approach as the BestFit method in the rest of this paper.

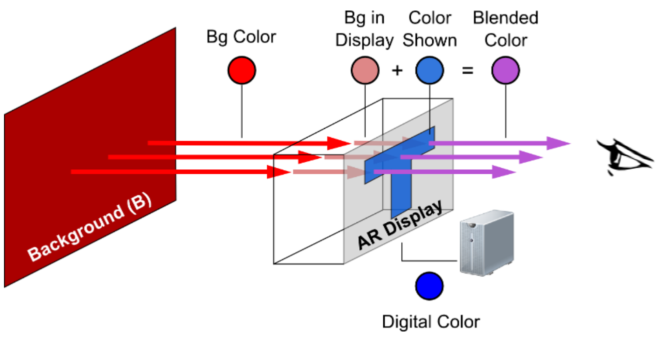

Fig. 2. Color blending as a function of the render and material distortions. Image taken from [22]. (best seen in color)

4 Scope and Methods

The goals of this paper are three. First, we seek to present an end-to-end color correction solution where both display distortions are accounted for, and the *min* function can be executed in real-time. Second, we go beyond an instrumental analysis of colors and perform a user evaluation of the perception of corrected colors. And finally, through our SmartColor framework, we aim to illustrate how such an end-to-end color correction solution can be leveraged to create different types of content.

4.1 Experimental Environments

For our first goal, we developed and tested our algorithm using a software-simulated OHMD environment which simulates color blending by means of color addition in the CIE XYZ color space. In the simulations, we assumed a white color of D65 (i.e. a color temperature of 6500K). The accuracy of the color blending estimations provided by this environment has been measured [22] to be below 1 just noticeable difference (JND = 2.3 points in LAB [13]). This level of precision allows us to generalize our results to actual OHMDs, provided a measured display profile is used. As background light we used video capture at a D65 white point.

For our second goal, the user evaluation, we used an EPSON Moverio BT-200 display for which we created a display profile (see below) using a Konica Minolta CS-200 luminance and colorimeter following the procedure described in [22]. To generate background colors we used a Dell U2312HM VGA LCD display calibrated at the standard D65 white point.

4.2 Display Binned-Profile Resolution

Similarly to Sridharan et al. we use display binned-profile to address the render distortion [22]. As a display profile we binned the LAB color space from values taken from sRGB. Previous work binned the CIA LAB color space at 5x5x5 to warranty all colors within the bin to be within 1 JND [7][22]. This approach results in two consecutive bins having a perceptual distance of ~2 JNDs. A smaller distance between consecutive bins results in smoother transitions between closely related corrections. A distance of ~1 JND between all neighboring bins can be achieved by binning at 1.32x1.32x1.321. However, the memory needed to store such high resolution profile is significant (approx. 132MB). We generated a binned profile at 2.3x2.3x2.3 to warranty a difference of ~1 JND between two consecutive bins on the LAB axes (approx. 21MB).

5 Handling the Material Distortion

The material distortion represents the degree in which the optical element of the OHMD modifies the light coming from the real world as it passes through it. When the material is very clear, objects are perceived as if there were nothing between them and the eye. Conversely, shaded materials alter background colors before they reach the eye – normally decreasing their brightness. Understanding the material distortion is necessary to correctly predict the blending of background and display colors.

Using a Dell U2312HM VGA LCD display calibrated at the standard D65 white point we generated random colors and, using the colorimeter, we measured their LAB components directly and through the display of the Moverio

[1] The distance between a bin and its furthest neighbor <= 2.3.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2015.2450745, IEEE Transactions on Visualization and Computer Graphics

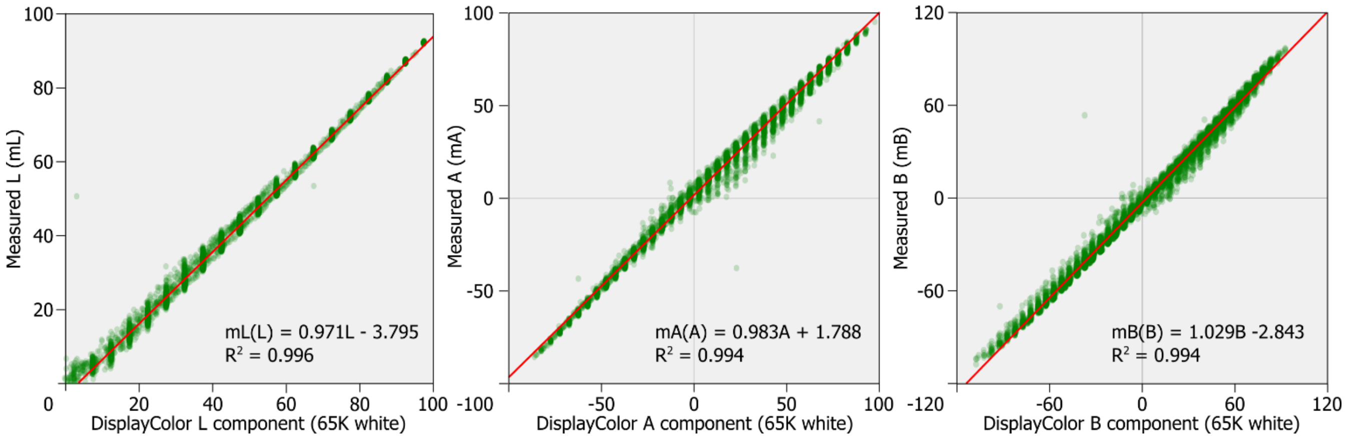

Fig. 3. The material distortion for the Moverio BT-200 can be defined as liner functions for the L, A, and B components (best seen in color).

BT200 wearable display. We sampled ~8000 random colors uniformly from the sRGB gamut.

Fig. 3 shows the resulting relation between each component before and after passing through the display material. Linear regression analysis between each component and its measured values revealed significant correlations in all cases ($p < 0.001$) with R^2 greater than = 0.994. Results also show that the standard error of the estimate is 1.35, 2.89 and 3.32 for L, A, and B respectively. These results show that, for the Moverio BT-200 display, each of the LAB components are affected linearly in a “very strong to perfect association”. Moreover, the component values can be used in a system of linear equations to predict 98.40% of the variability of the observed color (calculated as $R^2_L \times R^2_A \times R^2_B = 0.996 \times 0.996 \times 0.994 = 0.9840$). The remaining variability (expressed in the standard error of each component) could be explained by differences in how different wavelengths interact with the material, and any residual color in it.

Note that linear equations is only one possible approaches to quantify the material distortion. We work with linear equations given their high predictive power (R^2 values) and their low computational demand. Further research should determine alternative approaches which could account for the differences observed in the lower values of L and the center areas of A and B (see Fig. 3).

6 Real-Time Binned-Profile Correction

In order to propose a new correction algorithm, we studied how a display profile behaves in LAB coordinates during color blending. Fig. 4A shows the sRGB color space binned at 5x5x5 on the LAB color space (L - vertical axis, A - slanted red, B – slanted blue). Each sphere in the graph represents one bin of the display profile. Fig. 4B shows all bins in the profile change when blended with a greenish background. The bin color changes according to the background, and also its location in the LAB color space (see the trace lines). The bin representing “black” shifts to the location of the background color (black + green = green). A purplish background moves all bins toward purple (see Fig. 4C). And a light gray background moves all the bins upwards toward white (see Fig. 4D). This representation is consistent with Gabbard et al.’s two-dimensional plots of color blending [4], and reveals three properties of a blended profile:

1. All bins changed according to the background and move together.
2. Background brightness is passed on to the whole profile, so that no blend is darker than the background.
3. Despite the considerable changes in shape and size, the profile bins preserve their spatial relations.

The implication of these properties is that for any given point in LAB, the distances from the bins to the point are ordered. This pattern is also followed after the bins blend with a background color. For the sake of simplicity, we explain the following steps in a one-dimensional space. Given colors C1, C2, and C3 where:

$$C1 < C2 < C3$$

From Fig. 4 we deduce that for any background Bc

$$Blend(C1, Bc) < Blend(C2, Bc) < Blend(C3, Bc)$$

Therefore, for any display color Dc, if

$$distance(Blend(C2, Bc), Dc) < distance(Blend(C3, Bc), Dc)$$

Then

$$distance(Blend(C1, Bc), Dc) < distance(Blend(C3, Bc), Dc)$$

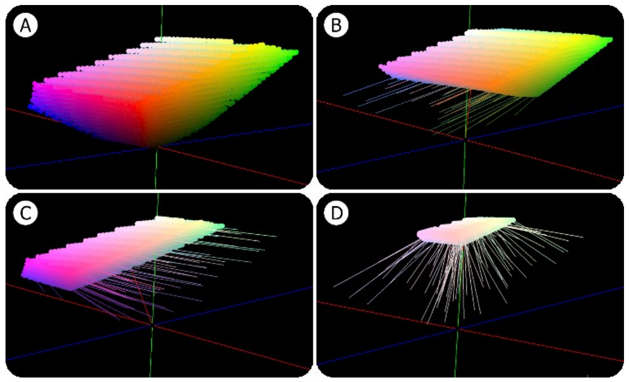

Fig. 4. Gamut blending in CIELAB. A) Binned sRGB on a black background. B) Blending with green (R: 59, G: 119, B: 31). C) Blending with purple (R: 105, G: 54, B: 169). D) Blending with light gray (R: 193, G: 198, B: 198). Tracing lines are drawn every 100th bin.

Our algorithm, called QuickCorrection, leverages the ordered natured of the blended profile to calculate distance to the desired display color on a subset of the display profile bins. Our algorithm is inspired by traditional binary search for ordered vectors. When correcting a display color Dc on a background Bc proceed as:

1. The display profile is represented as a 3D matrix of size L_{max}, A_{max} and B_{max}. To start, the algorithm locates itself on the center bin (L_0, A_0, B_0). The current location is called origin.
2. Calculate the distance from the blended origin to Dc.
3. A variable step exists for each axis, initialized as a quarter of the axis size, e.g. stepL = $L_{max}/4$.
4. While at least one step variable is greater than zero, do:
   - 4.1. For each axis find the bins locate at origin+step and origin–step, to form a set of 6 samples, 2 in each axis. Each axis configures a sampling plane.
   - 4.2. Calculate the distance from the blended samples to Dc, and estimate how much smaller each distance is compared distance calculated for the origin.
   - 4.3. If none of the samples is closer than the origin then divide all steps by 2. Go to step 4.
   - 4.4. If at least one sample is closer than origin then move the origin along a vector created by the relative weights of the closer samples. Go to step 4.
5. Return to origin.

6.1 Experimental Analysis

In this section we analyze the correction capacity of the BestFit [22] and QuickCorrection algorithms. We implemented the algorithms as fragment shaders for OpenGL. See the next section for details.

As test data we used random RGB pixels (alpha=255) for pairs of foreground and background 384x216 images (82,944 pixels). We collected data for 4 such image pairs for a total of 384x216x4 = 331,716 corrections. For each pixel pair we computed the normal blend (no correction) and the solutions with the two algorithms. We measured the distance in LAB from the intended foreground to the normal blend and to the blends of the two solutions. Colors are blended by using color addition in the CIE XYZ color space, and distances are calculated using the CIE LAB color space. In all cases we used a D65 white point. This color blending prediction has a measured accuracy within 1 JND [22].

Fig. 5 shows the distribution of the blend distances. Both correction algorithms present a similar correction pattern with the mean distance moving from 48.29 (no correction) to 31.53 in BestFit and 32.74 in QuickCorrection. These reductions mean a general improvement of 34.7% and 32.2% respectively. In other words, BestFit colors are in general 34.7% closer to the desired color. Similarly, QuickCorrection colors are 32.2% closer to the desired color. Statistical analysis with the Wilcoxon test for nonparametric data revealed both methods are significantly different than no correction (BestFit Z=-222.014 p=0.0, QuickCorrection Z=-221.593 p=0.0) and from each other (Z=-120.425 p=0.0).

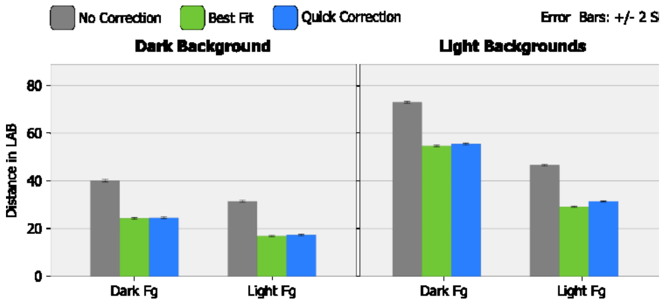

Fig. 7. Accuracy by levels of brightness/luminosity.

Fig. 7 shows the original blend (gray) and the blends corrected with the BestFit (green) and QuickCorrection (blue) algorithms. The chart bars are divided according to dark (L <=50) and light (L > 50) foreground and backgrounds. The foreground colors most affected by color blending are those with low luminosity (dark), particularly in light backgrounds (such as daylight conditions). This is due to the additive nature of color blending by which the L value of the background will "move up" the entire display profile. Fig. 7 also shows the correction accuracy of QuickCorrection to behave similarly to BestFit across the different luminosity levels. An important corollary of our analysis is that, taking into account that the BestFit algorithm tries all possible display colors, software-only color correction cannot achieve correction rates higher than 34.7%. In other words, the effects of the background color –particularly L–cannot be completely removed with software-based color correction alone.

A closer comparison between BestFit and QuickCorrection showed that in 2.91% of the pairs for which a perfect solution was possible (distance between the desired foreground and the BestFit solution blended with the background equals to zero) QuickCorrection achieved it in 2.33% of the cases. Moreover, QuickCorrection arrived to the exact same solution as BestFit in 53.25% of the cases, and in 61.40% the two solutions were perceptually equivalent (distance in LAB <= 2.3). Fig. 6-left shows the distribution of the remaining samples which did not result on the same color. The graph shows that 50% of the differences (80.7% of the total) are within 6.1 LAB units or less than 3 JNDs. Also, 75% of the differences (90.35% of the total)

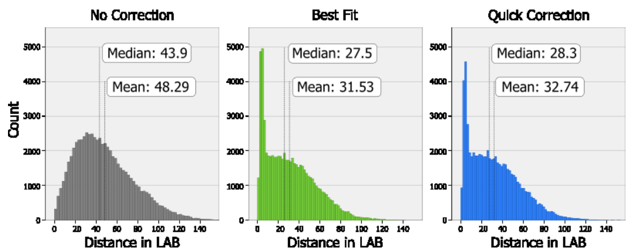

Fig. 5. Accuracy of the correction algorithms.

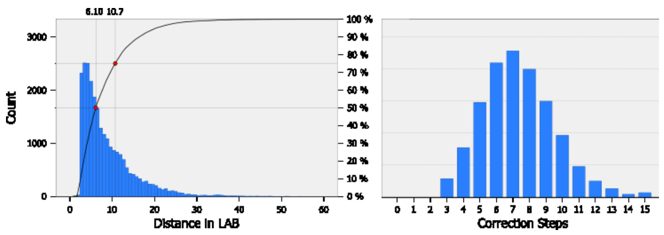

Fig. 6. Left: Differential between BestFit and QuickCorrection. Right: Number of steps performed by QuickCorrection.

are within 10.7 LAB units or less than 5 JNDs. Interestingly, in 0.56% of the corrections QuickCorrection arrived to a solution worse than the original blend.

Finally, we measured the number of steps QuickCorrection required to arrive at a solution (steps 4.3 and 4.4 of the algorithm). Fig. 6-right shows the histogram of the required steps. Visual inspection showed the mean number of steps to be between 7 and 8, and that most solutions were found within 15 steps.

6.2 Graphics Pipeline Implementation

Implementing color correction with modern graphic pipelines (such as DirectX or OpenGL) can be done at different stages of the rendering process. The rendering process receives drawing coordinates (vertices) and primitives (line, triangle, square, etc.) and converts them into pixels on the display. User code can intervene in the process and modify the attributes of vertices or fragments such as their location and color. A detailed explanation of the graphics pipeline is beyond the scope of this paper and we refer the reader to [16][17]. However, we would like to highlight the different computational demands of the vertex and fragment programs. Understanding this demands is important because ideally we would like a color correction algorithm to be integrated into the graphics processing while using the fewest possible computational resources.

The basic building blocks for computer graphics are primitives. A primitive is an order to the graphics card to draw basic shapes from a set of vertices. For example, to draw a triangle the system requires three vertices grouped into a TRIANGLE primitive. Complex shapes are an accumulation of simpler primitives: e.g., a rectangle is four vertices grouped in two triangles. The rasterization stage converts vertices and primitives into fragments, which can be understood as pixel candidates. The location and color of each fragment is calculated by interpolating the values on the vertices. Following this process, modern graphic systems generate large numbers of pixels from small amounts of data (vertices, primitives). The rest of this section explores the implementation of the QuickCorrection algorithm as a fragment and a vertex program. Fig. 8 presents the background (top-left) and foreground (top-right) used in the subsequent examples. Please note that Fig. 8-top-right is made by the 24 ColorChecker colors and each square is built with two OpenGL triangle primitives. Fig. 8-bottom shows the blend without correction.

6.2.1 Fragment Shader Implementation

In order to implement color correction on OHMDs we need to capture the background colors which blend with the display. With a camera on the OHMD we can capture the background image and estimate the background color for each individual pixel on the display. With this background image as input we first explore QuickCorrection as a fragment program.

We implemented QuickCorrection on OpenGL Shading Language (GLSL) 1.40 for compatibility with computers with low-end graphics cards. The algorithms used a 2.3x2.3x2.3 display binned profile created from sRGB, and passed it to the shader as a rectangular texture. The aligned background image was also passed down to the shader as a texture. Based on the analysis of steps discussed before, we limited the number of steps to 15.

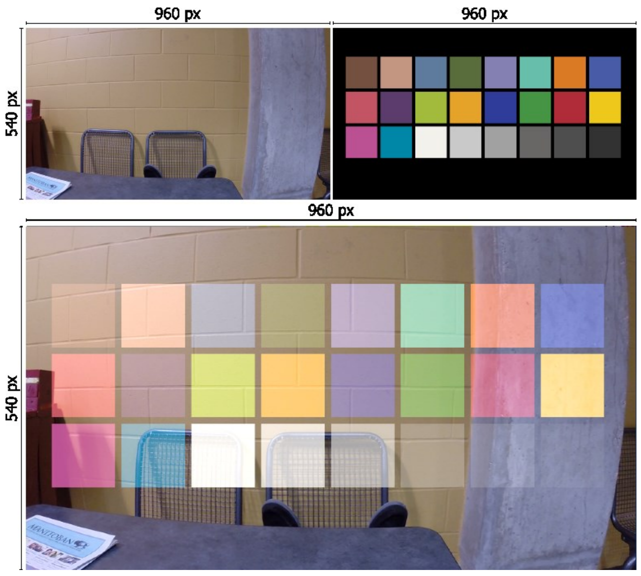

Fig. 8. Top: Background and foreground images. Bottom: Merged images is the blend to be corrected. (best seen in color)

Fig. 9 shows our pair of sample images corrected with the QuickCorrection fragment shader. The first impression is that the colors look, in general, more saturated than the blended image and closer to the desired foregrounds. In the case of the gray squares in the middle correction can be achieved almost perfectly to the point of occluding the chair behind. Another result of the QuickCorrection algorithm is that rectangles in colors similar to the background (top-left and lower-right) almost disappear. This near disappearance is due to the corrected result being a low luminosity color. The circular close-ups show how pixel-level correction might results in significantly different colors for consecutive display pixels, with the resulting display pixels forming a "rough" image.

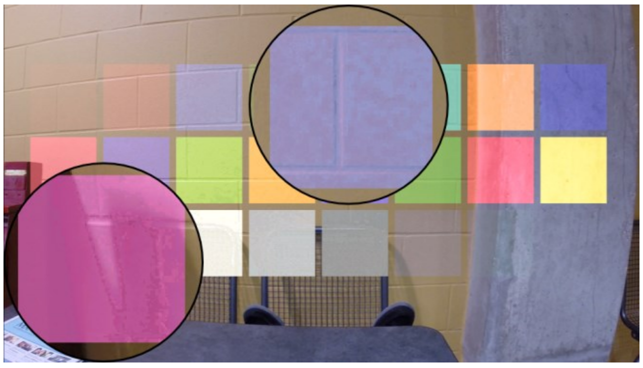
Fig. 9. QuickCorrection of the ColorChecker sample foreground. The highlights show lack of smoothness. (best seen in color)

We further analyzed the runtime performance of QuickCorrection. We measured the runtime performance for a rectangle of semi-random pixels covering different percentages of the display area (10% to 100%; at 100% the image size is 960x540 pixels, which is a common image size for commercially available OHMDs). The background pixels were also randomly generated. Both background and foreground pixels were generated only once in order to remove the generation cost from our runtime analysis. At 30% of the display area, the foreground image is 288x162 = 46,656 pixels. Hence, for each frame the QuickCorrection fragment shader is executed 46,656 times, with its associated access to memory to navigate the display profile. Similarly, at 70% the foreground image is 672x378 = 254,016 pixels or corrections per frame. We ran the QuickCorrection shader on a mobile Intel HD-3000 GPU @ 350 MHz (2 cores, 4 threads, OpenGL 3.1) and on a high-end Nvidia K4000 @ 810 MHz (768 cores, OpenGL 4.3). For each percentage and platform we ran the shader for 10 seconds and calculated the average frames-per-second (FPS).

Fig. 10 shows the runtime performance results. As expected the high-end platform was not significantly affected by the correction algorithm. Only at 90% of the display area (420K corrections per frame) a slow decline was registered, a trend that continued when correcting 100% of the display area down to 45.6 FPS. On the other hand, the performance of the low-end (mobile) platform decreased exponentially as a function of the display area corrected. The platform can only preserve real-time correction rates at 17% of the display area. The exponential decrease restricts the applicability of QuickCorrection as a fragment shader to computationally capable GPUs. Note that no other CPU or GPU code executed while collecting data, meaning that the FPS would be lower than measured when using the application-specific GPU and CPU code.

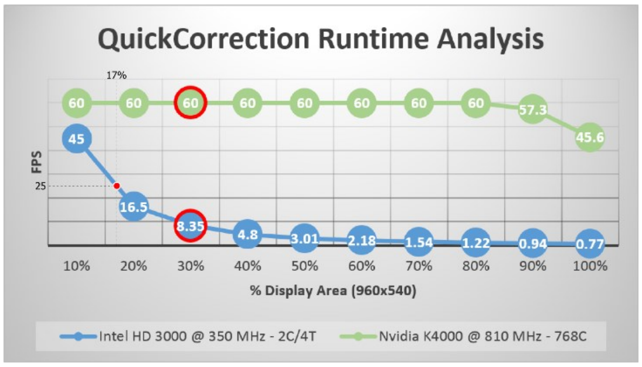

Fig. 10. QuickCorrection performance as a fragment shader.

6.2.2 Vertex Shader Implementation

Our second implementation moves the QuickCorrection algorithm to the vertex shader. Operating on the smaller number of vertices rather than fragments reduces the computational demand on the GPU. Moreover, by leveraging the interpolation capabilities of the graphics pipeline, we expect the consecutive fragments to vary less, resulting in smoother surfaces within an object. Two further considerations motivate the transition to the vertex shader. The first motivation is the observation that real-world objects often have uniformly colored surfaces. This consistency in color allows our vision system to make out contours which we use to recognize such objects. The second motivation relates to the parallax inconsistency between the user's eye and the camera capturing the background. The parallax problem means background captures need to be transformed to the perspective of the eye before mapping the display and background pixels. However, transformation techniques cannot account for the differences in depth between the objects in the scene. This limitation renders it impossible to achieve perfect matching between foreground and background pixels. Nonetheless, if the distance between the eye and the camera is short enough, and given our first motivation (uniform surfaces) the background color for a particular vertex can be estimated as an average from a number of pixels. The particular shape of the zone to average depends on the physical configuration of hardware elements and is beyond the scope of the current paper. For the purpose of validating the correction algorithm, we assumed an arbitrary average rectangle of 20 pixel sides. We arrived to the 20px mark iteratively.

Given the relatively large size of the interface components, such as the squares of our color checker sample (see Fig. 8-top-right), background changes between vertices are unaccounted for. While this scenario is desirable when the changes are relatively small (as explained before), when changes are large, colors should ideally be corrected again. To address this concern we tessellate the larger triangle primitives into smaller ones, and correct the color of the resulting vertices. This approach increases the number of vertices, but operates in still smaller numbers of vertices than fragments. Fig. 11 shows color correction at two tessellation levels. For uniform surfaces like the yellow

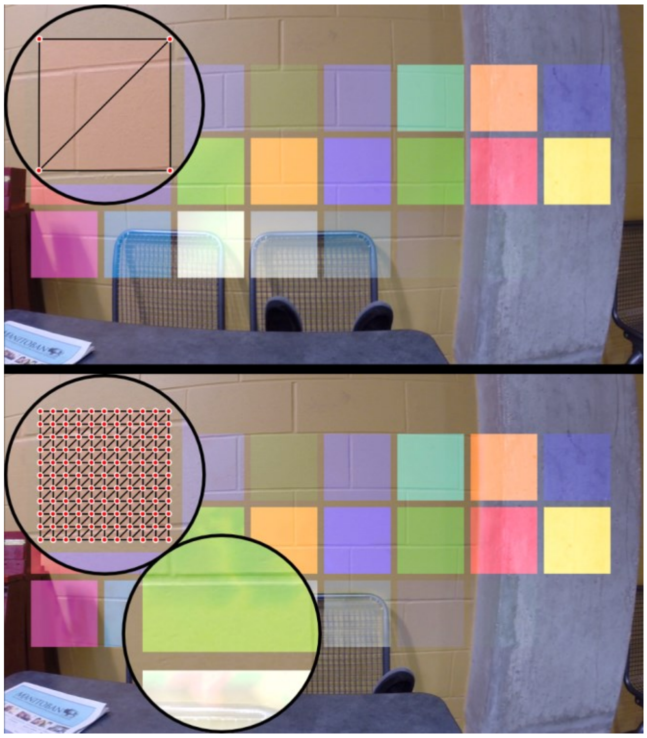

Fig. 11. Quick correction on the vertex shader. Top: A rectangle is made by two large triangles leading to flickering corners. Botton: A rectangle is tessellated into smaller into smaller triangles; the resulting image with internal noise.

wall, both results are equivalent to correction on the fragment shader. In uneven backgrounds, the finely tessellated rectangles adapt their corrected colors to the objects in the scene (see lower part of Fig. 11-bottom).

Correcting at the level of the vertex shader also presents problems such as uneven correction and flickering. The green highlight in Fig. 11-bottom shows that for the mostly uniform yellow wall, the internal vertices of the rectangle change their colors considerably giving the impression of unevenly corrected figures. In order to maximize consistency throughout the corrected figure and minimize the unevenness of the rectangle, we implemented a simple voting option which, after correcting at the vertex level, determines the bin which appeared most commonly as the correction result and propagates it to all the vertices. We implemented voting using the transform feedback mechanism of OpenGL. Fig. 12 shows all the correction alternatives, where E) shows an unevenly corrected rectangle and F) shows the results after a simple majority vote. The second problem is the observable flicker between frames due to subsequent frames finding different correction results when the background change is small. A smoothing mechanism can be used to reduce the flicker where the color to be applied in the current frame is a function of a few recent corrections (e.g. average of the last 20 corrections).

To study the performance of QuickCorrection as a vertex shader we used random noise as background and a 480x270 pixels rectangle. This size is equivalent to 50% of the size in the previous experiment. The rectangle was tessellated from 50 to 10 pixels-per-correction (PPC) in increases of 5. Each vertex of the resulting mesh was assigned a random color. Fig. 13 shows performance of QuickCorrection as a vertex shader, with and without voting. Without voting, the high-end computer did not have any noticeable performance impact (60 FPS), while the mobile computer consistently decreased performance. Compared to the fragment shader results (3.1 FPS at

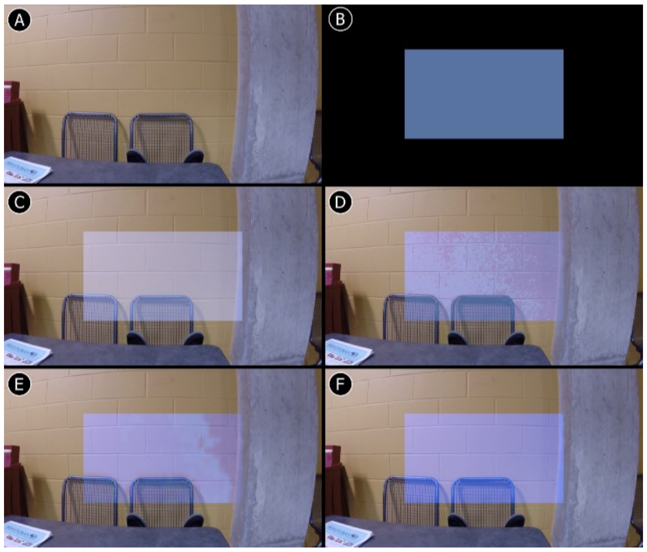

Fig. 12. A) Background image. B) Display image. C) Images blended without correction. D) QuickCorrection at the fragment level. E) QuickCorrection at the vertex level with tesselation. F) QuickCorrection@vertex with simple-majority voting.

50%), the mobile computer has a noticeably higher performance up to 15 PPC where it runs at 12 FPS. Interestingly, at 10 PPC the performance is similar to the fragment shader. One reason for such low performance is the added overhead of calculating an average background of 20px^2 per vertex. A more interesting observation is how quickly the voting alternative degrades in the high-end machine. More research is needed to determine its cause.

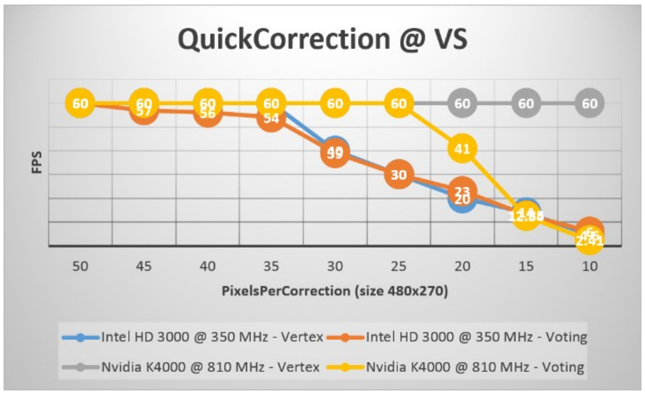

Fig. 13. QuickCorrection performance as a vertex shader.

7 User-based Analysis

Our second goal is to evaluate QuickCorrection from a user perspective; that is, whether users perceive the corrected colors as closer to a desired color when compared to their non-corrected version. To do this we conducted a controlled experiment.

Apparatus: The QuickCorrection algorithm and experimental application were run on a Windows 7 PC connected to a 23 inches Dell U2312HM VGA LCD display calibrated at the standard D65 white point. Participants wore an EPSON BT-200 head-mounted display running Android 4.2. Participants accessed the experimental software through a full screen browser running in the headset. All color corrections ran on the desktop computer taking as parameters the display profile of the BT-200 and the material distortion functions presented in section 3. Users sat in front of a monitor wearing a Moverio BT-200 head-mounted display as shown in Fig. 14.

Subjects: 12 participants (3 female) volunteered, ages 22-42 (mean 28). None of the participants exhibited signs of color-blindness as per the Ishihara color deficiency test. Six participants used prescription glasses to correct their vision. No participant had experience with OHMDs.

Task: The monitor displayed a solid background on most of the display, and a target color on the top 150px (about 4 cm). Users observed the target color through the OHMD[2]. At the same time, the OHMD displayed two rectangles side by side. One rectangle showed the color without correction, the other displayed the rectangle in the corrected color. Participants had to select the rectangle that, when seen on top of the background shown by the display, most closely resembled to color at the top of the display. The location of

[2] Target color was affected by material distortion.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TVCG.2015.2450745, IEEE Transactions on Visualization and Computer Graphics

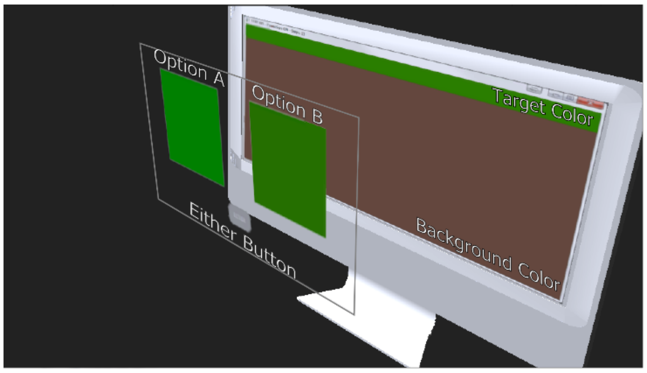

Fig. 14. Experimental set-up for the user studies. The user wearing the head mounted display can select between two rectagles the one that comes closest to the target color at the top of the computer display. Users can also select the "EITHER ONE" option shown in gray between the two rectagles.

the corrected color (left or right) changed randomly. Participants also had the possibility to select a third option - "EITHER" - when the two colors were equally close or far from the target color. For each trial we recorded the participant selection: the corrected color, the uncorrected color, or the "EITHER" option. No foreground color was repeated for a given background color, and the background colors were shown randomly.

Experimental Design: The dependent variable was the user selection (corrected, uncorrected, or EITHER). The independent variables were the background and foreground colors. Participants were presented with 10 random foreground colors for each of the 24 Color Checker colors as backgrounds. A researcher tested the participants for color blindness using the Ishihara color-blindness test, trained them with the system and demonstrated the experimental task. With a total of 24x10 = 240 trials per user, we collected a total of 2880 selections. Each selection lasted around 2 seconds, for a total time of 15 minutes (8 minutes in experiment + training).

Results: Participants selected the corrected rectangle 1075 times (37.3%), the non-corrected 753 times (26.1%) and either one 1053 times (36.6%) as shown in Fig. 15. A Friedman significance test showed a significant difference between all options ($\chi^2(2) = 24.876$, $p < 0.001$). Post-hoc pair-wise comparisons between conditions using a Wilcoxon Signed Ranks test revealed significant differences between *non-corrected* and *either* (Z=-3.437, p<0.001) and *non-orrected* and *corrected* (Z=-6.234, p<0.001), but no significant difference between *corrected* and *either* (Z=-0.575, p=0.565). These results mean that, in most cases, participants perceived corrected colors closer to the desired color than the non-corrected ones. However, in a large number of cases participants reported no difference between the two: they were both either equally good or equally bad representations of the desired color. Our data does not offer insight as to why participants selected the "EITHER" option with such. A hypothesis to explore in future research is that, when differences between the blends and target colors are small (< 1 JND), participants cannot differentiate blends; and when such differences are large and the correction is far from the target color participants

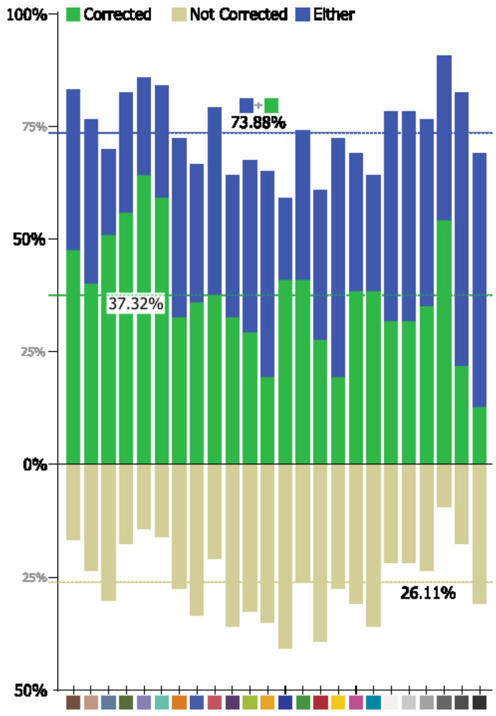

Fig. 15. Results of the user study. Users preferred the corrected version of the color more often than the not corrected version.

failed to perceive the correction as moving as closer to the target color.

We also examined how the perception of color correction changes for colors of different brightness levels. Fig. 16 shows that participants identified the corrected color as closer to the desired color more often when the color to correct is a bright one (L > 50) for any background. This result suggests that a color correction implementation could decide to leave dark foreground colors uncorrected (given that users will not be able to appreciate the difference) thus saving processing time.

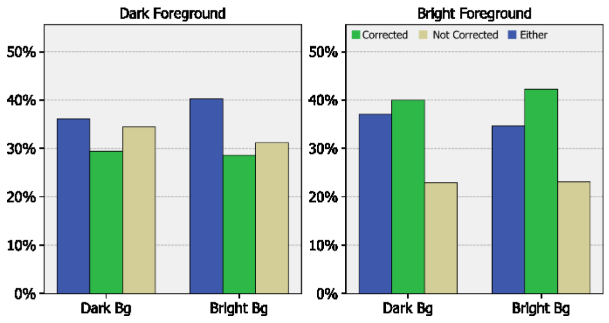

Fig. 16. Participants' preferences according to brightness of colors.

We also analyzed the users' perceived correctness as a function of the initial error between the uncorrected color and the desired color. Fig. 17 shows that when such distance is smaller than roughly 50 points in LAB, users more often identified the corrected color as closer to the desired one. However, when the initial error is large (> 50 points in LAB) users more often preferred the non-corrected color.

This is a counter-intuitive finding because, at first, it appears that color blends with the highest error would benefit the most from color correction. Further research is needed to determine the cause of this result. Nonetheless, the result seems to suggest that a color correction implementation could decide not to correct colors for which the calculated initial error is large. The exact threshold value should be determined upon more extensive experimentation and is beyond the scope of the current paper.

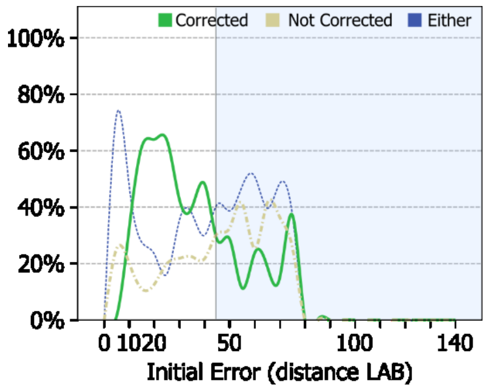

Fig. 17. Participants' preferences according to the initial error between the desired color and the blend. Images shows data only for bright foreground colors.

8 SmartColor: A Middleware for User-Interface Components in Optical Head-Mounted Displays

In this section we present SmartColor, a graphics middleware which leverages QuickCorrection to facilitate the creation of user interface components for OHMDs. We present the first SmartColor implementation, which focuses on two low-level *UI* components:

- 2D shapes: rectangle, ellipse, triangle, and line. These components are used as icons, and drawing elements.
- Text labels: Provide colored textual content.

Although a limited set, these components are the foundation of all two-dimensional user-interfaces such as WIMP. They can be used to build wearable interfaces similar to Google Glass, and 2D augmented-reality applications such the showcased ORA S display.

SmartColor relies on our QuickCorrection implementation as a vertex shader. All user interface components are defined as a set of vertices. The 2D shapes are defined via OpenGL meshes in which a series of TRIANGLE primitives configure the desired shape. Ellipse components have a circle resolution of 360 circumference points. Text labels are implemented as a textured font. Texture fonts map the coordinates of a letter in a texture image to the coordinates for the letter on the display (scaled and translated). This mapping means there are only four vertices per character. Given that the number of vertices directly influences the quality of the correction, text labels rely on a background mesh over which corrections are made. This mesh is equivalent to the one on the 2D rectangle component, and is not made visible in the user interface.

Interface developers can also define the pixels-per-correction (PPC) and correction steps parameters. The mesh behind each component is tessellated based on the provided PPC – default: 20px. We limit the PPC to minimum 15px, to warranty high performance even on lower-end processing computers as shown in Fig. 13. The correction steps parameter defines the ma number of refinements per correction. Based on the performance analysis shown in Fig. 6, we limit the number of steps to a max of 15.

8.1 Color Manipulation Strategies

Based on the QuickCorrection algorithm, SmartColor supports the following color manipulation strategies: correction, contrast and show-up-on-contrast. Correction simply applies the correction algorithm to the component's mesh in the way presented in the previous section. Fig. 11 and Fig. 12-EF present correction examples for 2D rectangles.

The contrast strategy aims at providing text content with the required luminance difference of 27 units against the background [25]. This strategy is implemented by limiting the QuickCorrection search to the bins of the display which, upon blending, have a minimum difference of 27 units of luminance with the background. Fig. 18 shows how this volume changes size according to the background. The bins painted white do not provide the required difference in luminance. In Fig. 18A the profile is blended with a black background. In this case the bins that produce the required contrast are a large subset of the profile. However, increases in the background's luminosity decreases the number of suitable bins. For contrast operations QuickCorrection uses the top-most bin (white) as its initial position in order to warranty white as the solution to the backgrounds with the highest luminosity. We limit the QuickCorrection search algorithm to the valid bins by checking that the bin sampled in the vertical axis has the required contrast. Using QuickCorrection to correct for contrast not only results in a color that maximizes readability for a given background (something which could be achieved by modifying the L component of the display color [8]), but also on a hue which is as close as possible to the originally desired hue.

Fig. 19 shows examples of correction and contrast for text labels at different foreground colors. The rectangle on the left shows the color as we want it to appear through the transparent display (added for reference purposes). The top label is neither corrected nor contrast adjusted and it shows how the text would normally look. The second label

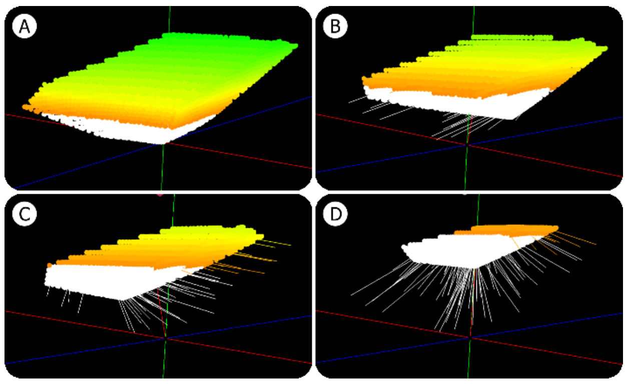

Fig. 18. Contrast ratio of the sRGB gamut on different backgrounds. White areas do not have enough contrast. (best seen in color)

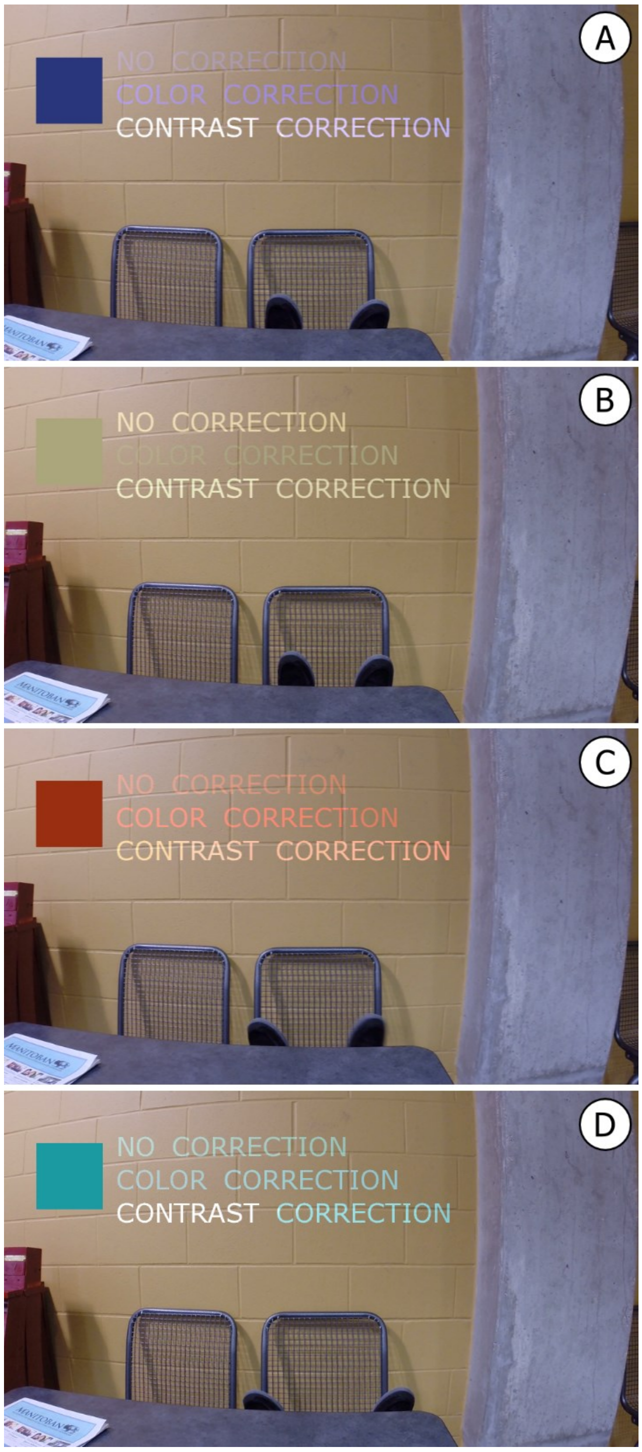

Fig. 19. Comparison of correction and contrast for text labels. (best seen in color).

is color corrected, and it shows how correction actually brings the color closer to the desired color on the neighboring square. Note that for cases A and B the color corrected label is almost indistinguishable from the background. The bottom label shows the contrast corrected textual content which is both visible and maintains some of the desired hue in all cases.

The final color manipulation strategy is show-up-on-contrast, or better understood as "shown when a related component presents a poor contrast ratio with the background".

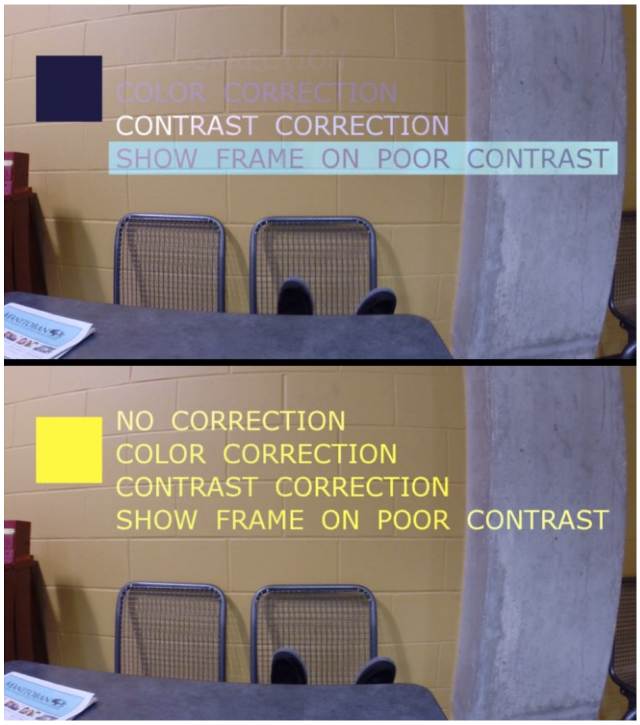

Fig. 20. Show on poor contrast. (best seen in color)

Fig. 20-top shows the no correction, correction, and contrast strategies, together with the shown-upon-contrast strategy. The light blue bounding box of the last label is shown because the dark blue color of the text does not have the necessary contrast difference from the background. Fig. 20-bottom shows that when the required contrast difference exists the bounding box is not shown. This strategy is an alternative to the contrast strategy and aims at maintaining the text color for situations where color encoding of the text is important (e.g. alert or status messages). Note that for all manipulation strategies, interface developers can determine whether to use a simple correction at the vertex level, or to execute a further voting (see Fig. 11F) and smoothing steps.

8.2 Implementation

We implemented SmartColor on top of OpenFrameworks (OF) 0.8 for C++. Our target platform was OpenGL 3.1 with GLSL 1.40. The display profile is passed as a 2D texture. By relying on OF, SmartColor is multi-platform for desktop and laptop computers. Support for OpenGL ES and GLSL ES is left for future work.

9 Conclusions, Limitations and Future Work

In this paper we advance the state of the art in color correction by presenting a way to address the material distortion and a real-time color correction algorithm based on display profiles. We showed that the material distortion can be addressed as linear functions on the LAB components. QuickCorrection navigates display profiles using sampling, a radically new approach. This approach is not obvious from a traditional computer graphics perspective (i.e. colors in RGB), and is valid only once we consider a perceptual representation of color (i.e. using the CIE LAB color space). Our key contribution is to demonstrate the real-time feasibility of color correction for OHMDs. We explored fragment and vertex shader implementations in terms of performance, raising important implications for user interface design: fragment-level correction is both costly and impractical (parallax problems); correction should happen at the vertex level, implying the use of shader-based texturing.

Moreover, we evaluated the results of QuickCorrection using actual human participants instead of colorimetric measurements. Our results showed that participants more often judged corrected colors as closer the desired colors, over the non-corrected alternative.

Based on QuickCorrection we proposed SmartColor, a framework for basic *UI* components. SmartColor allows the correction of color and contrast of *UI* components according to the background, and to display content conditional to the contrast level of another. While these contributions stand far from complete occlusion support, they offer new options for the increased legibility and preservation of color encodings in OHMDs.

SmartColor's two major missing components are the support for textures and 3D content. For images we cannot apply operations at the vertex level and therefore need to apply recoloring strategies. 3D objects such as planes, spheres, cylinders, cones and arbitrary 3D meshes, are needed for creating immersive augmented reality applications. Our current efforts focus on supporting this type of content, although we note that any such implementation is limited in its capacity to reproduce proper lighting (dark zones in an object) due to the transparency factor. A direction that follows from our current work, particularly the approach we used for text labels, is to use a background mesh to determine the base color for a 3D object (material color).

Our future work also includes the deployment of SmartColor on other OHMDs hardware. For this purpose we need to measure their display profiles and material distortions, and calibrate the camera parameters for accurate color capture and binocular parallax artifacts. Other directions include background dependent mesh tessellation and the use of the more detailed CIEDE2000 formula for color differences in the LAB color space [20].

ACKNOWLEDGMENT

We would like to acknowledge the Natural Sciences and Engineering Research Council of Canada's strategic Project grant for funding this work, and Konika Minolta, Inc., for granting us access to their color measurement equipment.

REFERENCES

[1] Bimber, O., Grundhöfer, A., Wetzstein, G. and Knödel. S. 2003. Consistent Illumination within Optical See-Through Augmented Environments. In Proc. ISMAR '03. IEEE Computer Society.

[2] Bimber, O. and Raskar, R. 2005. Spatial Augmented Reality: Merging Real and Virtual Worlds. A. K. Peters Ltd.,Natick, USA.

[3] Cakmakci, O., Ha, Y. and Rolland, J. 2004. A compact optical seethrough head-worn display with occlusion support. In Proc. ISMAR '04. IEEE/ACM.

[4] Gabbard, J., Swan, J., Zedlitz, J. and Winchester, W.W. 2010. More than meets the eye: An engineering study to empirically examine the blending of real and virtual color spaces. In Proc. VR '10. IEEE.

[5] Gao, C., Lin, Y. and Hua, H. 2012. Occlusion capable optical seethrough head-mounted display using freeform optics. In Proc. ISMAR '12. IEEE.

[6] Gao, C., Lin, Y. and Hua, H. 2013. Optical see-through head-mounted display with occlusion capability. In Proc. SPIE 8735, Head/HelmetMounted Displays XVIII: Design and Applications.

[7] Heer, J. and Stone, M. 2012. Color naming models for color selection, image editing and palette design. In Proc. CHI '12. ACM.

[8] Kiyokawa, K., Kurata, Y. and Ohno, H. 2001. An optical see-through display for mutual occlusion with a real-time stereovision system. Computers and Graphics 25, 5, 765 – 779.

[9] Kiyokawa, K., Ohno, H. and Kurata, Y. 2002. Occlusive optical seethrough displays in a collaborative setup. In ACM SIGGRAPH 2002 conference abstracts and applications, ACM.

[10] Kerr, S.J., Rice, M.D., Teo, Y., Wan, M., Cheong, Y.L., NG, J., NgThamrin, L., Thura-Myo, T. and Wren, D. 2011. Wearable mobile augmented reality: evaluating outdoor user experience. In Proc. VRCAI '11. ACM.

[11] Kruijff, E., Swan, J. and Feiner, S. 2010. Perceptual issues in augmented reality revisited. In Proc. ISMAR '10. IEEE/ACM.

[12] Leykin, A. and Tuceryan, M. 2004. Automatic determination of text readability over textured backgrounds for augmented reality systems. In Proc. ISMAR '04. ACM/IEEE. 224–230.

[13] Mahy, M., Eycken, L.V. and Oosterlinck, A. 1994. Evaluation of uniform color spaces developed after the adoption of CIELAB and CIELUV. Color Research and Application 19, 2, 105–121.

[14] Maimone, A. and Fuchs, H. 2013. Computational augmented reality eyeglasses. In Proc. ISMAR '2013. IEEE.

[15] Menk, C. and Koch, R. 2011. Interactive visualization technique for truthful color reproduction in spatial augmented reality applications. In Proc. ISMAR '11. IEEE. 157–164.

[16] Movania, M. M. 2013. OpenGL Development Cookbook. Packt Publishing Ltd.

[17] OpenGL –The Industry. 2014. Offical website for the OpenGL standard. [Website] http://www. khronos.org/opengl/

[18] Pingel, T.J. and Clarke, K.C. 2005. Assessing the usability of a wearable computer system for outdoor pedestrian navigation. Autocarto ACSM.

[19] Sekuler, A. and Palmer, S. 1992. Perception of partly occluded objects: A microgenetic analysis. Journal of Exp. Psychology 121, 1.

[20] Sharma, G., Wu, W. and Dalal, E. N. 2005. The CIEDE2000 color-difference formula: Implementation notes, supplementary test data, and mathematical observations. Color Research & Application, 30(1).

[21] Smithwick, Q.Y.J., Reetz, D. and Smoot, L. 2014. LCD masks for spatial augmented reality. In Proc. SPIE 9011, Stereoscopic Displays and Applications XXV, 90110O.

[22] Sridharan, S.K., Hincapié-Ramos, J.D., Flatla, D.R. and Irani, P. 2013. Color correction for optical see-through displays using display color profiles. In Proc. VRST '13. ACM.

[23] Tanaka, K., Kishino, Y., Miyamae, M., Terada, T. and Nisho, S. 2008. An information layout method for an optical see-through head mounted display focusing on the viewability. In Proc. ISMAR '08. IEEE/ACM. 139–142.

[24] Weiland, C., Braun, A.K. and Heiden, W. 2009. Colorimetric and-Photometric Compensation for Optical See-Through Displays. InProc. UAHCI '09. Springer-Verlag.

[25] Zuffi, S., Brambilla, C., Beretta, G. and Scala, P. 2007. Human Computer Interaction: Legibility and Contrast. In Proc. ICIAP '07. IEEE Computer Society.

Juan David Hincapié-Ramos is a post-doctoral researcher in Human-Computer Interaction (HCI) at the University of Manitoba (Canada). He completed his PhD in HCI and Ubiquitous Computing in 2011 at the IT University of Copenhagen (Denmark), and his bachelor in Systems Engineering in 2007 at EAFIT University (Colombia). He contributes to the IEEE augmented reality symposium (ISMAR) and the ACM conferences on human-computer interaction (CHI), user-interface software and technologies (UIST), interactive surfaces and tabletops (ITS), design of interactive systems (DIS) and ubiquitous computing (Ubicomp). He received a best paper award at ITS'11 for his work on tabletops, a student research competition award at UIST'11 for his work on prototyping tangibles, and honorary mentions at DIS'14 for his work on transparent mobile devices and at CHI'11 for his work on infrastructure awareness systems. His current research interests include augmented reality, head-mounted displays, interaction design, and prototyping and user studies.

Levko Ivanchuk is an undergraduate student at the University of Manitoba and a visiting student at the University of British Columbia. He currently works as a Research Intern at Honda Research Institute, Japan (HRI JP) under the supervision of Professor Kazuhiro Nakadai. His current work focuses on augmenting voice recognition with real-time human tracking. Previous to Honda, he worked as a Research Assistant in Human-Computer Interaction (HCI) at the University of Manitoba (Canada).

Srikanth K. Sridharan is currently a software developer with Varian Medical Systems. He completed his MSc in Human-Computer Interaction (HCI) from the University of Manitoba (Canada) in 2014. He completed his BSc in Computer Science and Engineering from Anna University Chennai (India) in 2009. He contributes to ACM symposium on Virtual Reality Software and Technology (VRST) and the IEEE augmented reality symposium (ISMAR). He received a student research competition award at UIST'12 for his work on prototyping an interactive hardware. His current career and research interests includes software prototyping, software architecture, interaction design, augmented reality and novel digital displays.

Pourang P. Irani received the BSc and the PhD degree from the University of New Brunswick, Canada. He is a Professor in human-computer interaction at the University of Manitoba and Canada Research Chair in Ubiquitous Analytics. His work focuses on the design, implementation and evaluation of novel software and hardware interfaces for advanced interactions and visualizations.