Abstract

Color blindness (or color vision deficiency) is the decreased ability to see color or differences in color and affects about 8% of males and 0.5% of females. There are three generic types: Protanopia (a lack of red cones), Deuteranopia (a lack of green cones), and Tritanopia (a lack of blue cones), with deuteranopia being the most common.

Apple recently introduced ARKit with iOS 11. By affecting how someone sees and experiences the world, Apple’s ARKit enables developers to create augmented reality experiences that drastically change how a user interacts with the color of the objects around him, opening the door to new and unique transformations to the experience of colorblind people.

Colorblind Aid, an iOS application written in the Swift programming language and developed in Xcode, takes advantage of augmented reality and core image filters to create a wholly internet-free iPhone application to help colorblind people. By combining these features, Colorblind Aid provides colorblind individuals with a novel approach to their deficiency – augmented reality and core image filtering. Through its straightforward and minimal design, the application guides the user through the intertwined features, giving them seamless execution between labeling, filtering, and locally saving.

This paper describes the application specifications as well as development process and challenges of building Colorblind Aid. It first details the specifications before describing each of the main features. Although attached separately, it highlights a few sections of code and describes the pseudocode of a few of the algorithms used. The paper then describes the user interface, providing screenshots of the application. Finally, in section 5 it outlines where development deviated from the original proposal, in section 6 it describes the key takeaways, and in section 7 the challenges and limitations. Section 8 concludes and 9 provides acknowledgements.

1 Background

Color blindness (or color vision deficiency) is the decreased ability to see color or differences in color. A genetic deficiency, color blindness is incurable and can make some activities difficult as well as prohibit certain professions such as that of a pilot, train driver, or armed forces. Affecting about 8% of males and 0.5% of females, the science behind color blindness is well known. There are three generic types\footnote{excluding monochromacy, where the person can see no color at all}: Protanopia (a lack of red cones), Deuteranopia (a lack of green cones), and Tritanopia (a lack of blue cones), with deuteranopia being the most common. [2, 7]

Apple recently introduced ARKit with iOS 11. ARKit is “a new framework that allows you to easily create unparalleled augmented reality experiences for iPhone and iPad” that works by allowing applications to blend “digital objects and information with the environment...freeing them to interact with the real world in entirely new ways.” By affecting how someone sees and experiences the world, Apple’s ARKit enables developers to create augmented reality experiences that drastically change how a user interacts with the color of the objects around him, opening the door to new and unique transformations to the experience of colorblind people. [1]

Colorblind Aid, an iOS application written in the Swift programming language and developed in Xcode, takes advantage of augmented reality and core image filters to create a wholly internet-free iPhone application to help colorblind people.

2 iOS Application Specifications

Colorblind Aid consists of three main features: “color tagging,” “color filtering,” and “local tag storage” (see feature details in §3). Largely intertwined, these three features are presented in a 3-pane slidable view, which enables seamless scrolling and transition between the three panes (see UI details in §4). A brief description of each
3 FEATURES

1. “Color Tagging” – a feature that allows you to mark and label the colors of objects seen through your camera in real-time (§3.1)

2. “Color Filtering” – a feature that applies a filter to a marked color/object, altering the color to “counteract” the deficiency to appear more like what a non-color blind person would see and can be used to transform commonly confused colors into easily discriminable colors (§3.2)

3. “Local Tag Storage” – a feature that allows you to save rooms of marked and labeled colors for access in the future (§3.3)

These features work closely with the native Photos and Camera applications of the iPhone. Furthermore, the application is independent of internet connection. An important specification as it enables the application to truly help colorblind individuals without the burden of first locating a cellular or wireless connection.

3 Features

As mentioned in the above section, Colorblind Aid consists of three main features: “color tagging,” “color filtering,” and “local tag storage.” Each of these features takes advantage of a separate iOS API, with color filtering being the most complex.

3.1 Color Tagging

This feature allows the user to mark and label the colors of objects seen through the camera in real-time. In essence, color tagging must convert the color at the center of the image from the inherit RGB values into a name, such as “blue” or “red”. This feature builds on Apple’s ARKit, Chirag Mehta’s Name that Color algorithm, and Github user @jathu’s UIImageColor to draw augmented reality nodes with appropriately named color hues [1, 4, 5].

Using Apple’s ARKit provided a straightforward method to create augmented reality nodes. The difficulty in implementing this feature was in converting RGB values to color names.

3.1.1 Initial Approach

My initial approach to solving this problem was to convert the RGB values of the pixel at the center of the camera to XY color values, which can be mapped onto a parabolic 2D graph of colors. Then, a switch statement could be used to determine the hue of the color. For example, (0.3, 0.5) might map to red.

Following code provided by Philips Hues, the steps to convert RGB to XY are as follows [6]:

1. Get the RGB values from your color object and convert them to be between 0 and 1
2. Apply a gamma correction to the RGB values, which makes the color more vivid and more the like the color displayed on the screen of your device
3. Convert the RGB values to XYZ using the Wide RGB D65 conversion formula
4. Calculate the xy values from the XYZ values
5. If the found xy value is not within the color gamut of the light, calculate the closest point on the color gamut triangle and use that as xy value
6. Use the Y value of XYZ as brightness

Given the CIE Chromaticity Diagram, I mapped the xy values to the appropriate colors.

Unfortunately, this process provided, dare I say, awful results. Due to imprecision in both the mapping process as well as the conversion process, the compounded errors resulted in incorrect labeling.

3.1.2 Final Approach

Given this initial failure, I pivoted to a completely different approach. Building off code provided by Chirag Mehta’s Name that Color algorithm, I compiled a list of about sixteen hundred common RGB and HSL color values, names, and hues (e.g. red, blue, green, orange, grey, brown, violet, white, black, yellow). For example, one such entry is colorName: "Baby Blue", hueName: "Blue", red: 111, green: 255, blue: 255, hue: 127, sat: 255, lum: 183.
Now, given the RGB values for a color, the HSL could be calculated and a straightforward minimum distance algorithm run through the 1600 entries to find the closest would return an approximate color and hue. Of course, this approach relies upon the compiled list of colors to be more or less extensive, which appears to be the case in application.

That being said, there were occasional blips in accuracy, presumably caused by the deviation and unreliability of any single pixel. Therefore, rather than use the pixel at the center of the canvas, @jathu’s UIImageColor algorithm was applied to a small square around the center which extracted the most prominent color. The conversion from color to name was then applied to this color.

These results were substantially better than the initial approach and held up during testing when under good lighting.

3.2 Color Filtering

This feature allows the user to apply a filter to a selected region, altering the color to counteract the deficiency of color blindness to appear more like what a non-colorblind person would see and can be used to transform commonly confused colors into easily discriminable colors. In essence, color filtering applies a transformation to the RGB values of the pixels of an image.

Using Apple’s Core Image API provided a clear method through which to manipulate the RGB values of a pixel. Building upon the project of Onur Fidaner, Poliang Lin, and Nevran Ozgufen who did an analysis of color blindness, in simple terms the process is as follows [3]:

1. Simulate color blindness using a commonly established transformation matrix

2. Isolate invisible colors to color vision deficiency (i.e. the difference between the original and the simulated pixel)

3. Shift colors towards visible spectrum by applying an error modification matrix (not all values should be affected equally)

4. Add back the compensated error values to the pixel’s original values

5. Range bound the RGB values

6. Return the transformed values

Two versions of this algorithm were developed, the difference being that in the second version the RGB values are first converted to LMS before color blindness is simulated. Simulation matrices for twelve types of color blindness are used and can be found in the appendix (§10). The three ending in LMS are for the LMS version of the algorithm.

Using a custom Core Image Filter written in kernel code, this transformation is applied to an image to alter and counteract color blindness.

In limited testing, the results of this transformation were incredible (although hard to show to someone who is not colorblind)! As is displayed in the three images below, applying this algorithm allows colorblind individuals to see previously invisible numerals in images that are used to test for color blindness, by altering the image through the above transformation algorithm.

Furthermore, these simulation matrices can instead be used to simulate color blindness, rather than correct for it. By applying the transformation without the subsequent adjustment and error modification, the pixels are adjusted to mimic what is seen by someone with color blindness. This simulation has many uses in application development where the developer would be wise to ensure elements of the application are clearly visible to both colorblind and non-colorblind users.
3.2 Color Filtering

An example from a color blindness test - the numeral is not visible to deuteranopes

The same image with the LMS deuteranopia algorithm applied

The same image with the deuteranopia algorithm applied. The 8 is now visible!

The same image with the deuteranopia simulation applied
3.3 Local Storage

This feature allows the user to save rooms of marked and labeled colors for access in the future. By combining the augmented reality nodes with Apple’s NSCoding protocol and local KeyedArchiver, rooms labeled in the AR view (§4.2) can be locally saved on a user’s device.

Keeping with the goal of being internet-independent, using the local KeyedArchiver enables local storage without the necessity of the cloud. Furthermore, it opens the possibility for future development of sharing saved rooms between users of the app by sharing these already saved objects over a connection.

4 User Interface

The user interface of Colorblind Aid strives to be as simple and seamless as possible. Consisting of three side-by-side panes, the user can slide between the three much like as is possible through Snapchat. The panes, from left to right, are Saved Rooms, AR Labeling, and Filtering (Figure 1). The swipe animation, about halfway between each pane, is displayed below in Figure 2.
4.1 Saved Rooms View

The leftmost pane is the Saved Rooms view. This view consists of a collection view of locally stored rooms of augmented reality nodes (§3.3). Arranged two across, the user may tap on a view to load those nodes into the AR Labeling view. The associated image for each room is a screenshot of the camera taken upon saving the room.

The user may also long-press (defined as a press lasting longer than 1 second) a particular room to be prompted by an action sheet where they may load (the same as tapping) or delete a room.

Along the bottom toolbar of this view is a button labeled “AR” which will transition to the AR Labeling view (§4.2). This same transition is also possible by swiping to the left.

4.2 AR Labeling View

The middle pane is the AR Labeling view. This view contains the app’s augmented reality capabilities (§3.1). After the user has enabled access to their camera, they will be displayed with a view consisting of what is visible through their camera, ontop of which the augmented reality nodes will be placed.

By tapping on the screen, the user will create a node at the center of the screen, whose location is designated by a light grey crosshair. This node, as visible in the middle image of Figure 1, is made up of a cyan sphere and orange bubble text. The sphere marks the location and the text is the color at that location. As this node is placed via ARKit’s API, it will reside in the 3D space of the room, not the 2D space of the screen. Furthermore, the text of the node will only appear parallel along the Y axis, allowing it to rotate, and thus be readable, as the user moves about the
4.3 Filtering View

The final pane is the rightmost, the Filtering view. This view is the interface through which the user performs the color filtering feature (§3.2). Initially blank, the user selects an image using the crosshair button along the bottom toolbar. Tapping the button prompts an action sheet where the user can either select a photo from their photo library or take a new one using the camera. As mentioned above, they may also capture a screenshot from the AR Labeling view using the capture button.

Once an image is selected, as long as the bottom right button says “Done” the user may filter the photo. Tapping that button enables and disables filtering.

While in filter mode, the user may draw rectangles over the image, creating a region over which they can filter the image. By tapping on a filter rectangle, the user is presented an action sheet allowing them to move, filter, or delete the rectangle. Selecting move prompts the user to tap on the image to change the center of the rectangle. Selecting filter filters the region of the image using the current filter. Delete removes the rectangle altogether.

The default current filter is “normal” or no filter. To change the current filter, the user taps the color filter icon on the right of the top toolbar. Upon tapping this icon the user is presented with a table of the twelve available filters (as described in §3.2). Now, choosing to filter a rectangle will apply this filter.

The user can also simulate color blindness, rather than correct for it, by long-pressing the color filter icon and selecting “Simulate” (versus the default of “Daltonize”).

Finally, the user can save the filtered photo by tapping the save icon on the left of the top toolbar. Tapping this icon will blend the filtered rectangles into the original image and save it to the users photo library.

Additionally, along the left of the bottom toolbar of this view is a button labeled “AR” which will transition to the AR Labeling view (§4.2). This same transition is also possible by swiping to the right.

5 Deviations from Original Outline

In this section I briefly share where the final project deviated from the original proposal and the subsequent reasoning.

5.1 User Accounts

I had originally planned to create user accounts with knowledge of the user’s color blindness. In the final product, these accounts were scrapped due to one the necessary associated internet connection and two the ineffectualness and inflexibleness provided by knowledge of the user’s color blindness.

While an early iteration of the application included Google OAuth authentication in order to establish user accounts, this feature was removed due to the reliance it created on the internet. I believe that any assistance application should be net neutral and have no innate need to connect to the internet. Furthermore, use of local storage was just as effective a mechanism as cloud storage making authentication even less necessary.
Secondly, not only may some users either feel uncomfortable disclosing their color vision deficiency to the app developer but others may not themselves be colorblind. Instead, the application allows any user to select any of the twelve algorithms to counteract or simulate color blindness, rather than preempt the needs of the user through a personalized user account. Additionally, again the use of local storage makes a personal account unnecessary as the rooms and photos are stored locally on the user’s device.

5.2 Augmented Reality Filtering

Another substantial deviation was the decision to only enable filtering through images, reserving augmented reality solely for labeling.

This decision was the result of discussion with early users and other developers. Not only would augmented reality filtering fail to be effectual as it would be nearly impossible to track an object as it moved, but also the feature did not seem relevant to many early testers. When presented with an option, most would rather filter on images, as is currently provided, rather than in augmented reality. They felt labeling was a great feature for AR and filtering less so.

6 Key Takeaways

Much was learned about application development and color science throughout the semester. To the first point, the necessity of being flexible and able to quickly adjust and pivot original designs proved paramount throughout development. By iteratively updating the design and features of the application when speaking with both the advisor group as well as other testers (namely my brother), the application became both easier to use as well as more helpful – after all, the purpose of the app is to assist those with color blindness.

Furthermore, an important realization while developing was that many of the icons and actions used from the beginning of development felt like second nature to me, however to an outside they were not always intuitive. A prime example of this is the capture button on the AR view. Many users initially thought that this button would place a node, not send them to filter a screenshot of the camera. My goal with the documentation of features document is to clear up these sort of confusions.

Another takeaway (and challenge) that very quickly made itself apparent was the non-generic nature of color blindness. While there are, broadly, three main types of color blindness, each individual sees color slightly different. This not only made it difficult to create generic filters, but also created a cacophony of replies when asking individuals about filtering. For some it worked perfectly, while for the occasional other it barely made a difference. These results made me not only appreciate the beauty of color but the science behind it. They also were incredible frustrating as I simply could not see the same thing that they were seeing!

These results also doubled my efforts to base the filters in science and transformation. The knowledge that there was an algorithm behind the feature promoted its ability to work.

7 Challenges and Limitations

Many of the challenges experienced throughout develop were those expected in the original proposal document. As stated, often the most difficult challenges are those you cannot expect. That was the case in this project. Two large challenges dealt with the iOS image framework and user interface display. The fact that the application can run on many separate devices proved to cause certain issues when placing the augmented reality nodes and creating rectangle filter views because the centering was not always consistent (especially on the new iPhone X which has much different dimensions).

Another challenge was the requirement to write the Core Image Filter in kernel code. Writing the algorithm took knowledge of a programming language with little documentation (a subsidiary of sorts to Swift). Furthermore, as mentioned above, it was difficult to test and tweak the parameters of the algorithm as different individuals reacted differently to the changes.

A final challenge was the variance of color labeling under different lighting. The same color, under different lighting, could be labeled as a whole slew of separate colors. Although I was not able to completely counteract this effect, consid-
erable progress was made by using the primary color of the region around the center of the screen, rather than a singular center pixel. This process (with UIImageColor) reduced this variance and made the label more reliable, although good, bright lighting is still heavily recommended when using the color tagging feature.

8 Conclusion

Colorblind Aid takes advantage of Apple’s ARKit and Core Image Filter as well as advances in color science to create an application that helps colorblind individuals. By providing three features: Color Tagging, Color Filtering, and Local Tag Storage, the application enables an individual to tag colors with augmented reality nodes and gives them the option to save them for later loading, as well as filter various images to either counteract or simulate color blindness.

As a colorblind individual, the application let me see numerals in images that were previously invisible and helped me distinguish between blues and purples in particular.

A fun foray into Xcode and the Swift programming language, this CPSC 490 Final Project led me through a whole manner of programming techniques and documentations as well as behind the scenes processes of application development.

9 Acknowledgements

I would like to thank Professor Holly Rushmeier for advising me throughout the semester, providing me both with insight into certain graphic and color design philosophies as well as advice on application features and development tracts. You proved to be an extremely valuable asset, thank you! I would also like to thank my advising group who gave great feedback and everyone else who tested the application throughout its development. Finally, I would like to thank my parents.
Appendix

Deliverables

1. iOS Color Blindness Aid Application
2. Swift source code
3. Brief documentation of application’s features and screenshots of application in action
4. Final Project Report

10 Simulation Matrices

normal: red: \((1, 0, 0)\), green: \((0, 1, 0)\), blue: \((0, 0, 1)\)
protanopia: red: \((0.567, 0.433, 0)\), green: \((0.558, 0.442, 0)\), blue: \((0, 0.242, 0.758)\)
protanomalred: \((0.817, 0.183, 0)\), green: \((0.333, 0.667, 0)\), blue: \((0, 0.125, 0.875)\)
deuteranopia: red: \((0.625, 0.375, 0)\), green: \((0.7, 0.3, 0)\), blue: \((0, 0.3, 0.7)\)
deuteranomalred: \((0.8, 0.2, 0)\), green: \((0.258, 0.742, 0)\), blue: \((0, 0.142, 0.858)\)
tritanopia: red: \((0.95, 0.05, 0)\), green: \((0, 0.433, 0.567)\), blue: \((0, 0.475, 0.525)\)
tritanomal: red: \((0.967, 0.033, 0)\), green: \((0, 0.733, 0.267)\), blue: \((0, 0.183, 0.817)\)
achromatopsia: red: \((0.3, 0.59, 0.11)\), green: \((0.3, 0.59, 0.11)\), blue: \((0.3, 0.59, 0.11)\)
achromatomal: red: \((0.6, 0.3, 0.06)\), green: \((0.16, 0.78, 0.06)\), blue: \((0.16, 0.3, 0.5)\)
protanopiaLMS: red: \((0.0, 2.02, -2.53)\), green: \((0.0, 1.0, 0.0)\), blue: \((0.0, 0.0, 1.0)\)
deuteranopiaLMS: red: \((1.0, 0.0, 0.0)\), green: \((0.494, 0.0, 1.248)\), blue: \((0.0, 0.0, 1.0)\)
tritanopiaLMS: red: \((1.0, 0.0, 0.0)\), green: \((0.0, 1.0, 0.0)\), blue: \((-0.396, 0.801, 0.0)\)
References


