DyslexiBye - Augmented Reality for Dyslexics

Charles Soe - Aung Hein Soe

December 2017
Abstract

Dyslexia is a problem for more than 40 million Americans. While many software solutions exist to help dyslexics read digital text (mainly in the form of text to speech programs), there are few, if any, solutions for text in the physical world. This continues to because much of the text we read today is still found in the physical world, e.g., on signboards, restaurant menus, paper, etc.

DyslexiBye is an iOS application that aims to fill this void. It takes an image with the camera and processes it with OCR, before leveraging Apple’s ARKit platform to project it back into the 3D virtual world, in a font/color scheme that is friendly to dyslexics. The font used in DyslexiBye is sourced from DyslexieFont.com, a studio based in the Netherlands whose founder is himself dyslexic, and created the font for people facing similar difficulties.

Augmented Reality is a new and exciting realm of development, especially in the mobile app development space, with Apple’s ARKit and Google’s ARCore leading the forefront of this push. While there have been simple demonstrations at the time of writing with overlaying hard surfaces with new textures, or projecting animated 3D sprites into virtual space, there are still few practical applications available for consumers.

This project aims to not only deliver a useful product for people with dyslexia, but also to push the boundaries of what is possible with ARKit, and in general, mobile-based, augmented reality solutions. Some of these boundaries challenged through the development of this app include the limitations of camera focus within ARKit, the poor performance of open source OCR libraries in various situations, and the lack of arbitrary plane detection provided by ARKit.
Chapter 1

Introduction

1.1 App Availability

As of December 2017, the first version of DyslexiBye is only available via TestFlight to authorized testers on iTunesConnect, Apple’s app submission platform. The source code is available via GitHub, for those who wish to download and run it on their own devices: https://github.com/CharlesW95/DyslexiBye

1.2 Components of the App

DyslexiBye is meant to be used primarily as a single screen application. The home screen (which appears on startup) holds most of the OCR/AR/application logic.

1.2.1 Home/Main screen

The main screen is where the application’s logic is executed. It holds a camera view for users to easily point their device towards a piece of text that they wish to have reformatted into dyslexic friendly font, by simply dragging their finger across the screen to draw a box around the specified text.

The same main screen also holds the Augmented Reality (AR) view, and this is where the reformatted text gets projected back into the virtually rendered 3D world, so users can interact with it.
1.2.2 Tutorial View

The tutorial view can be accessed by tapping the 'Help (?)’ button on the top right corner of the screen.

Once the view loads, the user can go through the tutorial by swiping through the view. There are 3 steps in total, laid out with screenshots to help users understand the usage of the app seamlessly.
1.2.3 Information View

The information screen can similarly be accessed by tapping the 'Info (i)' button at the top right corner of the screen. It holds information about the purpose of the app, as well as the makers of DyslexieFont, the font used within DysleXiBye.
Chapter 2

Technological Choices

In terms of the technology stack, platform and resources used, there were many choices that had to be made throughout the development of Dyslex- iBye. This section details and justifies the main choices that were made.

2.1 Platform - iOS and ARKit

The two main options for mobile platforms were iOS and Android. iOS was chosen because of the relative maturity of Apple’s ARKit, as opposed to Android’s ARCore, which had been released for less than a week at the beginning of this project.

Swift was the language of choice for development over Objective-C because of Apple’s concerted efforts in the last few years to make all iOS frameworks Swift-first.

2.2 Optical Character Recognition (OCR) Library - Tesseract

Of the easy-to-integrate and free options for iOS, Tesseract and SwiftOCR stood out as the best options. This project started off using SwiftOCR, however, performance was low in medium to low lighting conditions (SwiftOCR works better on screenshots or crops of digital documents), and as such was replaced by Tesseract in the project.
Tesseract has its own limitations (discussed below), but remains a popular option amongst most developers looking for a free, mobile OCR solution.

2.3 Font - DyslexieFont

DyslexieFont is the leading dyslexic-friendly font, developed by Christian Boer of DyslexieFont.com - it was easily available for download, and was the most fully fleshed out font of all the options that were available. Christian was contacted during the duration of this project, and he and his studio have been immensely helpful in terms of explaining how the font is meant to be used, as well as providing the color, background color and spacing options that make the font most legible to people with dyslexia.

Figure 2.1: Left: Regular fonts, Right: DyslexieFont with optimal color scheme

As can be seen above, DyslexieFont with the optimal adjusted color scheme looks significantly different from the original text. The text is made to be spaced out wider, and each character in the typeface is designed to be asymmetrical, and also to give the illusion of having weighted edges lending to a 3D effect. This makes it easier for dyslexics to distinguish between different characters in the alphabet.
Chapter 3

Challenges

Several technical and non-technical challenges were faced in the development of DyslexiBye, however, this report will mainly cover the technical challenges that were faced. Non-technical challenges involved collecting information about how people with dyslexia function, the current state of technology (especially mobile applications) available to them, etc.

Judith York of the Yale Disabilities Office and Christian Boer of Dyslexie-Font.com were both extremely helpful in providing information about the problems that dyslexic people face, as well as the current potential solutions that are either available or in development to help them.

3.1 Lack of Camera Focus in ARKit

The first challenge faced in this project revolves around ARKit’s lack of support for manual or auto-focus in it’s camera’s input stream. While in ARKit, programmatically triggering camera focus is not an option. Apple’s provides an explanation for this, stating that the detection of feature points in the camera’s view is heavily reliant on depth perception, which does not function well when the area of focus in the camera’s view is constantly being modified.

As seen in Figure 3.1, the lack of a focus option in ARKit made it difficult for the camera image to be used as input for OCR - the text was often barely legible even to the human eye.
3.1.1 Subtly swapping in a camera view for focusing and capturing the image

The solution that was employed involved instantiating a camera view (wrapped in an AVCaptureSession) and swapping it in to replace the ARKit view for the duration of the photo focus and capture, and then swapping it back out. This approach came with problems of its own.

First, iOS does not support having an ARSession running simultaneously with an AVCaptureSession. As such, the ARSession needed to be paused before any AVCaptureSession could be instantiated and run. This would mean that any prior calibration/feature detection done by ARKit would be lost, and that feature detection needed to be restarted after ARSession was resumed. Since feature detection typically takes a few seconds after ARSession begins/resumes, and requires movement of the camera, this became a problem since it was not feasible to have the user continue to hold the camera in place for an arbitrary amount of time after the photo has been taken so that feature points can be re-detected.

The final solution to this ended up involving the caching of these feature points before the ARSession gets paused for the first time, and using the 3D projection of these points to ultimately project the text back into 3D space.
The final process is as follows:

1. Cache ARSession configuration, feature points and their 3D world coordinates.
2. Pause ARSession, instantiate AVCaptureSession and camera view, and overlay it on top of paused ARSession.
3. Focus on specified area with AVCaptureSession, capture and crop (more on cropping below) photo, store photo.
4. Stop the AVCaptureSession.
5. Resume the ARSession first with cached configuration.
6. Remove the camera view from on top the already running ARSession to create a seamless transition.
7. Pass on cached 3D coordinates and cropped image to the OCR facility for further processing and projection.

3.2 Poor OCR Performance

Despite Tesseract being the best performing free out-of-the-box solution for iOS, on many occasions its performance still left a lot to be desired. It was not able to recognize text from surfaces with slight curves (bottles), or with a slight reflection (signboards, glossy posters). Furthermore, because there was often too much detail in one image (outside the bounding box of the targeted text), text extraction often an unreasonable amount of time (≈ 5 seconds).

The following steps were taken to improve OCR performance:

3.2.1 Focusing on a sub-section of the image

An active decision was made to let users select a specific area of the camera’s view to crop - since users would already be aware of where the text lies in an image, DyslexiBye would allow users to specify a bounding box (as seen in Fig 3.2) by dragging their finger across the screen. This input from the user not only allowed for a more compact, text-heavy portion of the image to be fed into the OCR engine, it also created a specific 2D co-ordinate upon which
the camera could focus on (often with high variance in depth in a single view, autofocus was landing on the wrong part of the image).

3.2.2 Image pre-processing

In order to improve OCR performance, Tesseract’s original image processing steps had to be overridden. After some research and experimentation, we decided to use a GPU-powered library to implement an adaptive threshold filter, overwriting Tesseract’s preprocessing function with this filter, before passing the image back to Tesseract.
After this step, results were much improved.

3.3 ARKit Projection

Projection of the text back into 3D space proved the most challenging portion of developing this application. Since we had to project a plane (text on background) back into 3D space, it was natural to try using ARKit’s natively implemented plane detection algorithms. However, it very quickly became apparent that this would be insufficient - ARKit currently provides support only for horizontal planes - and we required not just vertical planes, but arbitrary planes depending on where the user decided to place the text. This is because ARKit employs the Speeded-Up Robust Features (SURF) algorithm, which performs best when searching for planes in one orientation.

This became a significant problem - how should we project our plane node back into 3D space?

3.3.1 Attempt 1: Point-based projection

The first naive solution undertaken was the first remove the plane, and try projecting just the text into 3D space (since this only required specifying the center point). Using the feature points and 3D world coordinates that were cached earlier, a center point was extrapolated and used to project the text. However, performance was poor for 2 reasons:

1. With only a center point for reference, orientation became a problem along all 3 axes - ARKit did not know how to orient this 3D block of text, and often the text was assigned an arbitrary orientation (decided primarily by the position assumed by when the user first launches the app), making it uncomfortable to read most of the time.

2. Because there was only one point being used to anchor the text node, there was very little stability and consistency - the text node would often change its position in the virtual 3D space, not being able to identify a large enough number of points off which to define and root its position. This was rather disconcerting to watch, and it was decided that this approach would not work.
3.3.2 Attempt 2: Precise corner detection

Given the previous failed approach, it was decided that a single was insufficient for projection - we needed to obtain a plane on which to project, and since ARKit was not able to provide us with arbitrarily oriented planes, we needed to calculate and create these planes on our own.

The first approach to computing planes was simple enough - given the cached feature points on the exact four corners of the bounding box specified (drawn) by the user, we would take three of these points, compute 2 plane vectors and obtain the normal vector through the cross product of these vectors - we would then have a plane on which to project.

However, the main limitation with this was that feature points were actually being collected in a much more sparse fashion than was originally anticipated. In fact, more than 90% of the time, there were no feature points on the four corners of the bounding box, leaving us with insufficient coordinates from which to compute a plane.

3.3.3 Attempt 3: Plane approximation with 3D rotation

In order to improve on the above approach, 2 things became clear:

1. The user should not start drawing a bounding box until there are sufficient feature points detected by ARKit.

2. We needed some way of approximating coordinates using detected feature points that were close to the corners of the bounding box, without shifting too far away from the original corners.

To the first point, a calibration light was added in the top corner of the screen. The light would start off as red, and an asynchronous worker would be dispatched every 0.5 seconds to check for and count the number of detected feature points. The light turns green when there is a reasonable number of feature points detected. Users are informed during the tutorial that DyslexiBye achieves its best performance when the bounding box is drawn when the calibration light is green - this has helped to ensure there are enough feature points to work with.

On the second point, a simple search-and-approximate algorithm was developed to approximate the 4 corners. This algorithm can be expressed as
follows:

1. For each of the four corners of the user-drawn bounding box, perform an outwardly spiraling search for the nearest feature point, computing the Euclidean distance traveled in order to reach this point. If four feature points are not detected within a certain number of steps, return failure.

2. For each of the four feature points detected, retrieve the associated 3D world coordinate, and based on the Euclidean distance traveled from the original corner of the bounding box, offset this world coordinate to obtain an approximate 3D coordinate that represents where the corner of the plane should be.

3. Return the 4 3D world coordinates.

This approach worked significantly better in terms of obtaining the four 3D corners coordinates of the plane we wanted to project. The only problem now was the orientation of the plane - the default normal vector for all planes is determined to be the negative direction of the z-axis (determined as the direction facing the camera when the app starts up), and so some of the planes we were projecting using the points obtained were oriented wrongly. To solve this, we calculated the new rotation axis (cross product of our computed normal and the default normal, and rotated the plane around this axis by the inverse cosine of the normalized projection of our computed normal onto the default normal.

This finally resulted in a high-performing 3D projection.

Figure 3.4: Plane projection with the final, high-performing plane approximation method.
Chapter 4

Limitations

4.1 Tesseract Performance Limitations

Even with the adaptive threshold filter, Tesseract seems to function best when the input is an image with black text on a white background - when it meets other color combinations, it struggles, sometimes returning nonsensical results. This is something that could be improved on within this app with better image preprocessing.

Tesseract itself also faces limitations in terms of the variety of fonts supported. While Tesseract supports (has been trained on) most common fonts, there is still a substantial number of font styles in the physical world (e.g. many engravings on Yale buildings) that are fashioned in very uncommon fonts, and Tesseract does not naturally handle this well.

4.2 ARKit Text Projection Limitations

ARKit itself is very intensive on the GPU, and rendering 3D text takes a heavy toll on even the iPhone X’s processor. As such, when the OCR outputs a string with more than around 200 characters, ARKit is not able to project the entire string of 200 subnodes into 3D space, often resulting in the app crashing because of memory constraints.

While smaller optimizations could be made on our end, this is a problem Apple recognizes with ARKit, and will probably be best fixed by a rendering upgrade with ARKit itself.
Chapter 5

Future Developments

Future potential developments for this application include the following:

1. Improving image preprocessing to be color-sensitive (mentioned above).

2. Allowing users to calibrate their own color schemes and font settings - although the default font already works well for most people with dyslexia, there is potential to tailor this app to each individual with slightly different preferences.

3. Enabling users to save recognized text on landmarks - so that other users don’t have to re-initiate the OCR and projection process, and instead enjoy text already projected by other users.

To follow up on this project, we will continue working with Christian Boer and his team at DyslexieFont.com to bring this app onto the App Store so that it can reach users with dyslexia.
Chapter 6

Conclusion

Augmented Reality is very much a nascent and experimental technology. Developing DyslexiBye was a journey fraught with many more challenges than initially anticipated, most of them arising from extending ARKit’s capabilities and working around its limitations.

However, the fact that a functional prototype was created by working around these challenges shows that ARKit (and, AR itself), is a very exciting field, and that more work deserves to be done to enable even more powerful applications in this domain.