Dan Burkhardt

Professor Kaiser

Topics in SW Engineering

6 May 2017

Final Report: VRPlus

Deliverables & Source Code

There are two places where the source for this project is located, both on my public Git profile.

iOS Framework: https://github.com/DanBurkhardt/VRPlus

Node.JS API for video downloading: https://github.com/DanBurkhardt/VRPlusAPI

The VR Playlist Feed Source (in app content):

https://www.youtube.com/playlist?list=PL762an06h83jwiiCscMR5MvP5PnTlDxqD

Running The Project

Please view the readme on the iOS project in order to run the project on an iOS simulator or your iOS device. Please see the readme at: https://github.com/DanBurkhardt/VRPlus

Goal of VRPlus

From my initial **proposal**, I have envisioned a framework for iOS that would make it easy for any developer to implement VR in an iOS application by handling all of the UI and networking associated with displaying a feed of 360 degree videos. I also wanted to ensure that the minimum number of lines would be required to instantiate and configure the VRView, which would be the view controller that contains the list view UI. Underpinning this project is the vast amount of VR content available publicly on YouTube. When multiple videos are assembled into a playlist of similar content, it makes it easy to consume VR content in a mobile

application. My project aimed to leverage the amount of content on YouTube in order to feed a playlist of video content into the application.

iOS Framework Development

I started off working with the YouTube API in order to determine the exact endpoints that I would need to use to get data for a playlist, including video IDs and metadata of the videos on the playlist. Once I figured out how to reliably access a playlist, I began writing classes in Swift to enable the retrieval process to happen with very little configuration from the end user. After that process was tested and complete, I turned my attention to the UI. I followed my original design mockup as a guide and created a basic single-view application that displays a table of content based on the YouTube playlist that is retrieved using my YouTube API class.

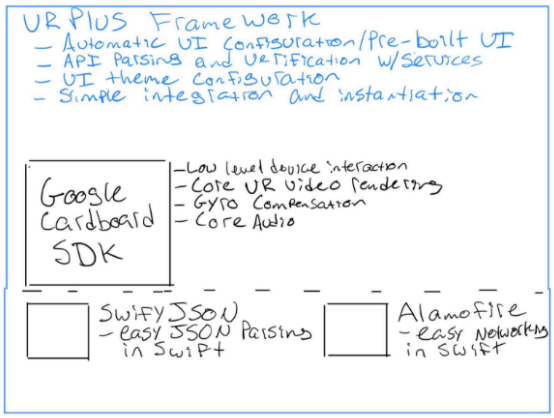

Figure 1: The framework model with all bundled frameworks included.

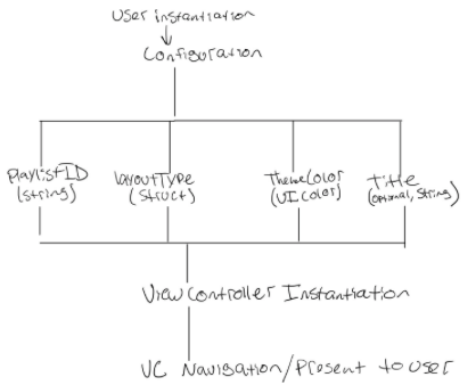

Figure 2: The proposed minimum number of configuration options

When a user makes a selection by tapping the item in the list that they wish to view, the process of retrieving a video from YouTube becomes a little bit complicated. I discovered that the Google Cardboard SDK does not accept a video content link from YouTube directly because it only works if you pass it a direct URL to a video file. This was a pretty massive technical issue for me, as I discovered this after I had already developed the UI and I had never imagined that it would be a conflict given that YouTube and Cardboard are both Google Video projects. Nevertheless, I was determined to make my initial vision work, so I developed a very basic API that would allow my application to gain direct URL access to a video file by putting my own server application in between the framework and the video on YouTube. (I will explain this workaround in the next section).

According to plan, I was able to develop the application incorporating multiple third party frameworks as planned, the core of which is the Google Cardboard SDK. This SDK handles all of the underlying video rendering and processing to actually display and interact with a VR video. Along with that, I incorporated a popular networking framework called Alamofire. This framework allows for very easy remote data access and REST API communication. To handle the

JSON response, I used SwiftyJSON, which is another popular framework for working with JSON in Swift.

VRPlus API

As I mentioned earlier, I had to develop a special API as a workaround in order to deal with limitations in the Google Cardboard SDK. As it turned out, unfortunately, the Cardboard SDK does not accept video content streams from YouTube, which was one of the underlying premises of the project which was based on the YouTube playlist API. The architecture for that process is shown in figure 3.

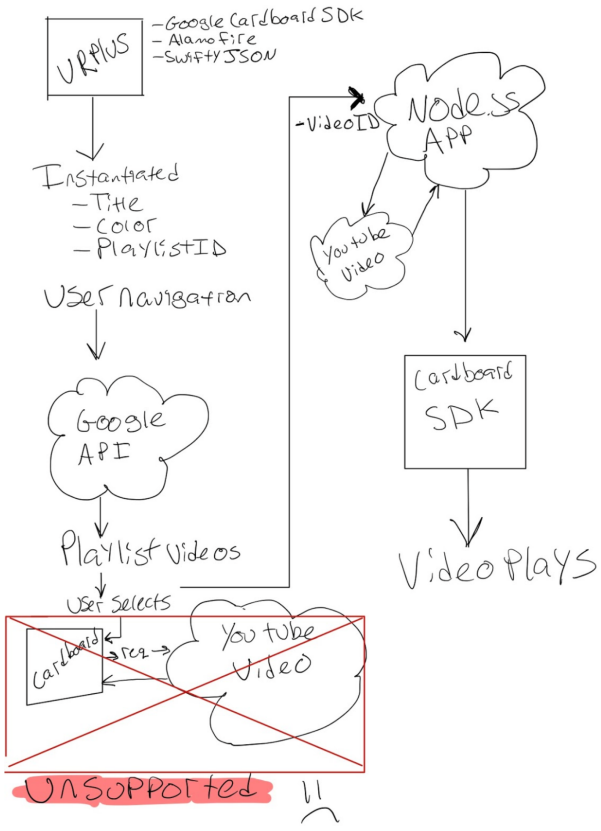

Figure 3: Original plan boxed in red, actual implementation on the right.

This API is relatively simple. There are two unprotected routes. The first is "downloadVideo" and the second is "getVideo", and each route accepts a videoID parameter.

The download video route initiates the video ripping process by retrieving the video from the YouTube server. The “getvideo” route is meant to function as a callback, so that only after the “downloadVideo” endpoint has returned, is the “getVideo” function called, to ensure that the download has always completed before attempting to stream the video file. Here is a sample of the API usage:

1) User selects a video from the list. The video ID is passed to the server.
2) Server is called with the URL including the video ID as an argument: http://vrplus.mybluemix.net/downloadVideo?videoID=[videoid]
3) Server returns after download is complete
4) Application then calls the “getVideo” endpoint which returns the actual video file: http://vrplus.mybluemix.net/getVideo?videoID=[videoid]

This solution functions well to enable the project to work as envisioned and serves to make the process of consuming video relatively invisible to the end user and the developer. However, while this solution works well technically, it is actually against YouTube’s end user agreement and terms of service, so this is not a solution that can scale or be used in any public commercial sense.

Findings / Outcome

While I have achieved my intended outcome of developing a framework that makes it easy to incorporate VR into any iOS app, and while it is one that requires only a very minimal amount of configuration, what I learned about the incompatibility between the Google Cardboard SDK and YouTube will prevent me from making this framework public or from making this solution releasable without significant changes, or without developing my own VR

video playlist feed platform. This framework will have to remain a proof of concept for the time being until Google Cardboard SDK supports a direct link from YouTube (which I think will happen in the future at some point) in which case I will be well-positioned to make my original idea. Until then, it will work as when using or testing on any user's iOS device (iPhone only).

In terms of actually developing the framework and the time invested, it took me about 3 hours of planning and experimentation with APIs, 2.5 hours of planning and research on the approach I wanted to take, and about 6 hours to develop and test the project it to the state that it is in now. In total this represents approximately 11.5 hours of total time invested in the project. I believe that this represents the typical amount of time that an advanced/experienced iOS developer would invest in reaching the same state of development. However, when this framework is applied to a brand new project with no other features incorporated, the total implementation time for anyone familiar with using mobile frameworks would be under 30 minutes for basic customization and configuration. It took me about another 2-3 hours of development and research to develop and deploy the custom API, but I would omit that from the general calculations of the time cost to reach an outcome on this project because it would not be necessary under circumstances where YouTube and Cardboard are compatible and was only developed as a workaround.

These findings about the amount of time and effort involved to reach this basic feature set regarding VR incorporation, and the difference between writing it from scratch vs. incorporating it as a framework justifies my core assertion that an easy-to-implement framework for incorporating VR content into iOS applications is indeed both a worthwhile pursuit and a viable project for general commercial release. While the underlying issue of

compatibility between Google's services is still a significant development issue, it is reasonable to believe that this is an issue that will eventually be resolved, in which case this framework model would stand to potentially save tens, or eventually hundreds, of man hours of feature development in order to incorporate VR content consumption into iOS applications.

What I Learned

The biggest takeaway lesson from this project is that even though a company has two products that seem complimentary in nature, it is not safe to assume that they will work together. I was surprised, and frankly, very worried about the viability of my project after hitting the roadblock of incompatibility between YouTube and the Google Cardboard SDK. While I was able to overcome this issue by developing my own solution, this solution doesn't scale and essentially violates YouTube terms of service and is therefore not releasable. If I were to approach this project again, I would try to focus on developing my own basic VR playlist hosting solution for use within the framework as a software solution. I will be following up with the SDK development team as well in order to open ticket on the Github to suggest this feature for future releases. I also learned about the challenges of developing a framework for iOS that would allow me to incorporate and bundle UI and UI specific files (I.E. Storyboards in iOS).

Things I Did Not Get To

I wanted to do a bunch of things on the error-checking and stability side of things, but I got so side-tracked by the challenge of having to figure out a way around the issue of not being able to feed a video directly from YouTube and refactoring my code to handle it that I was not able to get to most of it. Here is a short list of items I still would like to complete:

- Building a separate array of content after the initial YouTube playlist returns and then populating it with only verified 360 degree video content to ensure the stability and reliability of the Cardboard SDK
- Enabling custom menu bar color and typeface configuration
- Deploying the framework to Cocoapods

Third Party Software Used

During the creation and testing of this application, I have incorporated three third party frameworks. Alamofire and SwiftyJSON handle networking and JSON parsing & unwrapping respectively. Google Cardboard SDK is the underlying VR framework that handles the actual rendering and display of VR content.

Frameworks:

- Google VR: https://github.com/googlevr/gvr-ios-sdk
- Alamofire (easy networking in swift): https://github.com/Alamofire/Alamofire
- SwiftyJSON (easy JSON handling/parsing): https://github.com/SwiftyJSON/SwiftyJSON

On the API side, I leveraged IBM's Bluemix platform as a backend for my application. I used a few third party frameworks to aid in the retrieval of video files and network communications. Express and CFENV are packages that I used for network communications and interaction with the Bluemix environment respectively. I leveraged a package for downloading YouTube videos called "youtube-dl" that basically handles all of the underlying tasks associated with ripping a YouTube video from the YouTube servers.

Packages:

- https://www.npmjs.com/package/youtube-dl

- https://www.npmjs.com/package/express
- https://www.npmjs.com/package/cfenv

Sources/Reference

https://developers.google.com/vr/ios/

https://github.com/googlevr/gvr-ios-sdk/

Nava. Elad Nava, 19 Oct. 2016. Web. 9 Mar. 2017. <https://eladnava.com/publish-a-universal-binary-ios-framework-in-swift-using-cocoapods/>.