Augmented Reality - Virtual Reality
Immersive Theatre Experience with VR and Speech Commands
2021-2022

Gaspari Michele 0001007249
Giambi Nico 0001007301

August 5, 2022

Introduction

This project consists in developing a Virtual Reality (VR) application to create an **immersive theatre experience**. When we deal with VR, one of the most important feature of our application is the **immersivity**. With this word we intend to provide to the user an experience in which he/she can forget about the real world and focus only on the artificial environment. To achieve this, there are several component of an application that can be considered, for example:

- realistic reproduction of the particular environment we want in our VR app, both from the point of view of the vision and sound
- creating an interesting plot/flow of execution in the app so that user attention is catched

Surely, there are more aspects of VR that we can consider when speaking about immersion, but in the particular case of our application, where we want to reproduce a theatre performance, we can focus mainly on reproducing a high-quality video of the performance (from an audiovisual point of view) and on the flow of the app execution. The app is made of two main phases:

- **Video Selection**: When the app starts, the user can see a list of available videos with their relative thumbnail from a certain source, and they can select the one they want to see;
- **Video Player**: After having selected which video to play, the user will find themselves inside a 360* Video, where they can look around and interact with the video by either a User Interface or Speech Commands.

The first phase can be customized in order to upload specific videos into the reference server. Instead, for the Video Player part, we opted for **Speech Commands**. Since watching a movie or a theatre performance is mainly a passive activity, it's important to take in consideration that most of the time the user will not be interacting with the surrounding environment. Keeping a minimal UI in the VR space, hence, is indispensable in order to maximize the immersivity of the experience.

Software

This app was developed for Google Cardboard, so it is mandatory to use Google VR SDK, a set of utilities to easily interact with Cardboard devices. This SDK, however, is only supported in older version of Unity (up to v2019.4), so we decided to use the latest available version. The app supports Android versions from 4.4, to 12 (the latest stable API level). For the Automatic Speech Recognition and Text To Speech component, we used a plugin to integrate Java code into Unity. The package we used is configured for Unity 2022, so we had to use older versions in order to work with v2019.4.

Unity Setup

To deploy the application on both Android and iOS devices, you need to install a suitable version of Unity Editor (we used the LTS version 2019.4.3f) with the modules to deploy on Android and iOS (from UnityHub):

- **Android Build Support**

  - **Android SDK & NDK tools**
  - **OpenJDK**
- **Windows Build Support (IL2CPP)**
- **iOS Build Support**
- **Mac Build Support (Mono)**

In order to build this app for Android devices, there are some mandatory configuration steps to do in the player settings (Open project with UnityHub -> File -> Build Settings -> Switch Platform to Android and then go to Player Settings):

- **Minimum API Level**: set this field to API level 19, which refers to Android 4.4 Kitkat;
- **Vulkan**: from the Graphics API list, remove Vulkan and only leave OpenGL;
- **Scripting Backend**: set this field to IL2CPP and then check ARMv7 and ARM64 fields;
- **VR support**: on the XR settings panel, check "Virtual Reality Supported" and select Google Cardboard as target device;

Lastly, in the Android Manifest we had to add a line in the activity field,

```
android:exported="true"
```

This is aimed to solve versioning problems since from Android 12, every application, hence each APK that needs to be installed in a recent phone, must contain the tag reported above.

Application Overview

Google VR SDK

The realization of this app was in great part due to the **Google VR SDK**, an open source SDK which offers a set of Unity components and scenes in order to easily interact with google VR products (Cardboard, Daydream, ecc.). The SDK includes primitives to control the VR scene:

- **Gyroscope**: Movements with the cardboard are automatically converted in VR movements by reading the device's gyroscope. This means the camera will always follow the user's movement to look around in the VR space.
- **Pointer**: At the center of the screen there is a white circle which becomes smaller when it "hits" a clickable target. This gives a visual feedback to easily understand what the user can interact with. A click on the Cardboard button is interpreted as a click over the white circle, independently of which part of the screen was really clicked.
- **Stereoscopic View**: The stereoscopic view is automatically computed and rendered on the device's screen by using this SDK. When launching the app, if the device still isn't configured for any Cardboard device, the user is asked to scan the Cardboard's QR code in order to setup the parameters to best tune the stereoscopic view. Different versions of Cardboard may have smaller/bigger lenses, focal distance and intra-lenses distance. Also, each device could have its own resolution width/height ratio, screen's pixel density and so on. All these parameters are taken into account when configuring the Cardboard in order to maximize the immersivity.

Moreover, the VR SDK offers some demos, including one for Video Players. It shows different kind of interactions with a video in the VR space. The three examples shows:

- **Plane local video**: A rectangular panel is place in the VR space and a locally available video is played over it.
- **Plane remote video**: Same as before but the video is gotten from an URL
- **360 Video**: Here, the user is immersed in a 360° video all around him/her. This could be what we needed but there were some problems with it. This demo used a normal Video and distorted it in a way it looked like it was recorded in a 360° camera, and only then they would apply it to the insides of a sphere to render it. This meant that using 360° video from the start couldn't work because there would have been an extra distortion step which would have rendered the video in a corrupted way.

Given this problem, we decided to re-implement the scene and the interactions basically from scratch, while taking inspiration from the demo and exploiting the aforementioned VR SDK components to easily deploy the app.

Video Catalog

When the application is launched, the first thing the user sees is a black space all around them and a grid of videos with the relative titles right in front of them. The black space is implemented with a black sphere centered at the camera. In this version, the grid is a 3x3, meaning only a max of nine videos can be shown, but we already set its container to be a **ScrollView** in order to easily expand its dimension in the future by dynamically create **Game Objects** containing the videos preview. The shown videos are crawled with a regex starting from an URL pointing to a remote directory (by changing the URL, the app is automatically redirected to another endpoint, hence it's easy to configure it to a different server if needed). [1] When this directory is loaded, all the **href** tags are unwrapped and if the links points to an **.mp4** file, it is stored in an Array-like structure in order to easily retrieve it later. After obtaining all the available videos in the directory, each URL is dynamically assigned to a **Video Player** component which shows a preview of the video in the VR space. Here some controls are made in order to manage scenarios like:

- # **Retrieved videos** > # **available grid cells**: only the first nine videos are shown.
- # **Retrieved videos** < # **available grid cells**: shows the retrieved videos and set as inactive the remaining grid cells.

All the background components (canvas, panels, scroll view, ecc..) are specified to not be **Raycast Targets**. This means that the white circle will only expand when hovering over a selectable video. To choose a video, all the user has to do is point it with the white circle and press the Cardboard button. This action stores the selected video URL and passes it to **MyVideoControlManager** class, which will be later explained. Then, the **Catalog** component is rendered inactive and the actual 360° Video player scene is activated in its place, hence starting the immersive experience.

[1] At the moment the server endpoint is set to localhost and doesn't work on mobile. To set an active server as the endpoint, change the *uri* variable's value to the server path in **CatalogManager.cs**

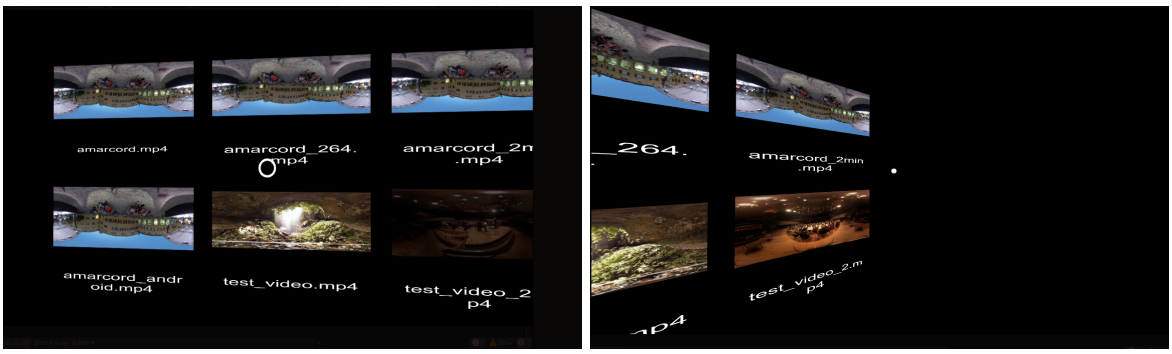

Figure 1: Catalog example with only 6 videos

VR Video Player

This **scene** is the main component of the app, where all the immersive experience takes place. The camera, as before, is placed at the center of a big sphere with inverted normals. This means the renderer can display its texture (in our case the video player) on its inside instead of its outside. By instantiating a **Video Player** component on the sphere, with the previously stored URL, the frames of the chosen videos are unwrapped on the sphere's internal surface, hence giving the impression to be inside the real-life scene.
On the bottom-frontal part a legend showing the available vocal commands is placed. The legend is structured as three panels with **Text** components, structured in a **depliant** style in order to display all the text facing the camera (the user's point of view).

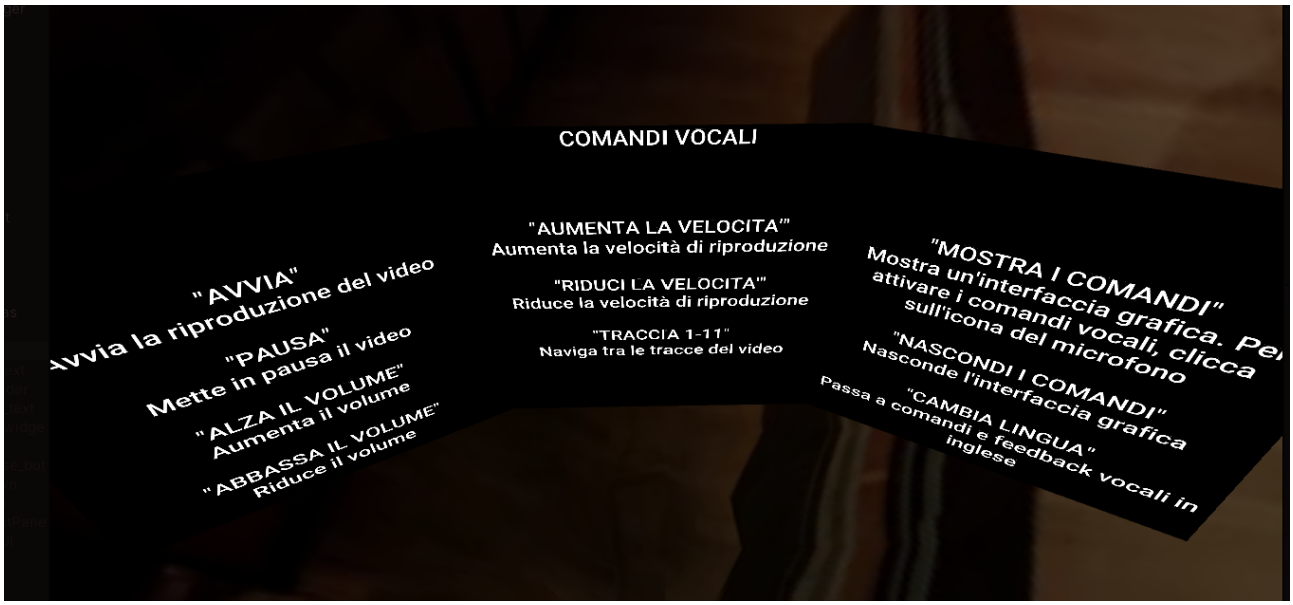

Figure 2: Italian Legend

In the actual version of the app, the language is set to **Italian**, and the legend shows the Italian version of the commands. As we will see later, the user can easily switch to **English** speech commands.
When we get to this scene, the video playback is automatically started in order to not have a frozen view. Now the user can start interacting with the video by clicking on a point in the scene and keeping pressed while saying one of the available **Speech Commands**. In any case (both available and unavailable command), the user will get a **vocal feedback** stating what action was done, in case of an available command, or a suggestion to utter another command in case of a failed match. All these commands interact with either the video or the UI. As mentioned before, we tried to recycle as much code as we could from the Google VR SDK demo, but unfortunately they used their own classes to interact with the video. Moreover, they had a very limited set of available interactions with their own video player components. This is why we decided to stick with Unity's video player, since it gave us much more degrees of interaction. This also means we had to rewrite from scratch the callbacks used to manipulate the video and manually add more primitives to slow down / speed up the playback, skip to predefined tracks and so on. In **MyVideoControlManager.cs** we defined all the new

functions to manipulate the video playback. To do this, we instantiated a pointer to the video player component, and each of it's modifiable property is managed by a separate function which is called either by using a vocal command or by clicking its corresponding button in the UI. As already said, by saying "**show/hide interface**" we can enable a small UI which contains:

- A **Play/Pause** button;
- A **Fast Forward** button (right)
- A **Slow Down** button (left)
- A **Microphone** button (on top) to trigger the speech commands listener.
- A **Slider** (below) with a movable dot to go back and forth through the video. It also shows the actual time of the video (left) and the video duration (right)
- A **Volume** icon (far left) which, if clicked, displays a vertical slider to control the volume.

These buttons and sliders are meant to manipulate each of the available video player properties. This was the starting point of the app, where we linked each button to its relative callback. All these buttons resides in a single **Canvas**, in order to hide them all at once in case the user wants to only interact via speech commands.

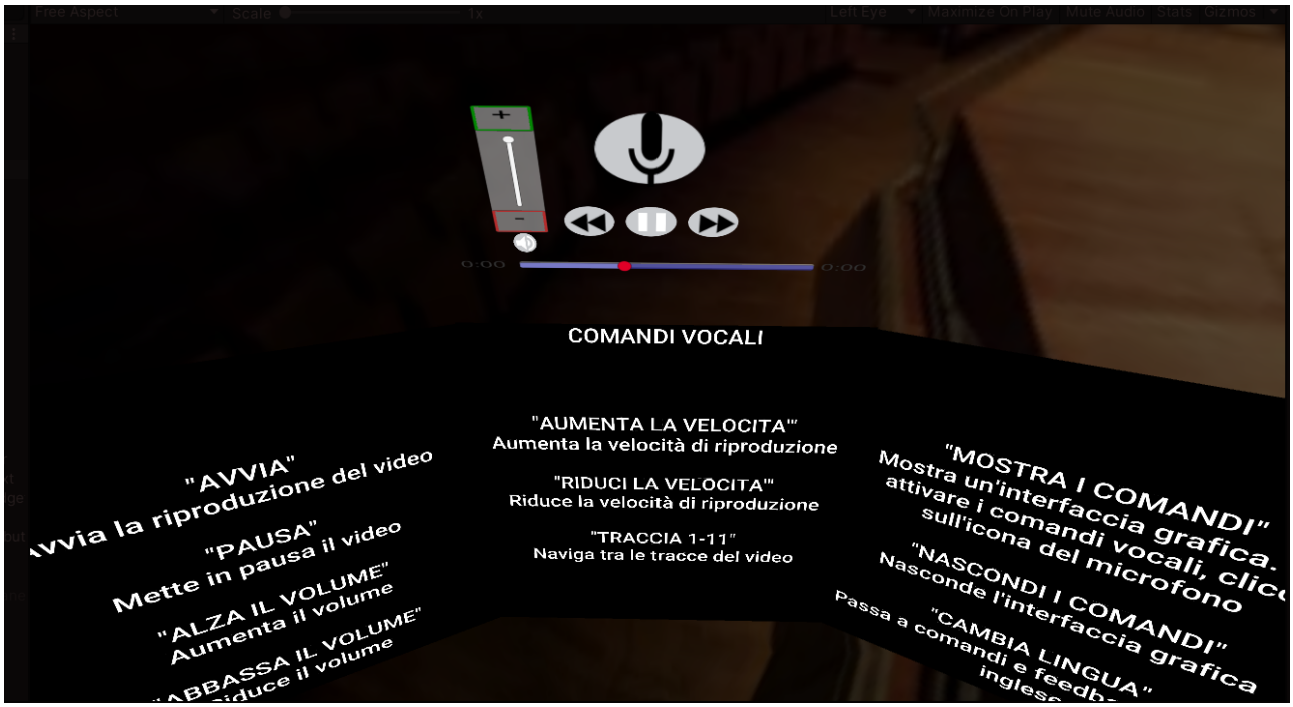

Figure 3: User Interface

Speech Commands for Unity

As pointed out in the introduction, our purpose is to provide a more immersive VR experience of a theatre performance. In order to avoid breaking user interaction with the artificial environment, to select the desired command, we thought of **hiding the user interface in favour of speech commands**. In this way, once the user has memorized the available commands, can continue to interact with the app without looking for the UI, that would distract the user and definitely break the performance immersion. To develop such feature of our app, we need:

1. a way to trigger vocal interaction
2. a tool that can effectively perform speech recognition and synthesis

In order to start recording the microphone stream and thus recognize the vocal command, we thought of a simple hold and click scenario. In particular, to trigger the speech recognizer, the user must click and hold on the panel in the VR scene. While holding, it's possible to speak and once the thumb is lifted the recognizer

performs the uttered command (usually a control light switches on/off in the phone when we speak at the mic). Eventually, the user receives an adequate feedback thanks to the speech synthesizer.

The user can speak in italian (there's the specific vocal command to switch to english) in order to execute one of the following commands that are self-explanatory and easy to understand:

English Command	**Italian Command**	**Effect**
Play	Avvia	Play the video
Stop	Pausa	Pause the video
Speed up	Aumenta la velocità	Speed up the video
Slow down	Riduci la velocità	Slow down the video
Volume up	Alza il volume	Turn the volum up (one unit)
Volume down	Abbassa il volume	Turn the volume down (one unit)
Track 1 .. 11	Traccia 1 .. 11	Skip to a predefined video part
Show interface	Mostra i comandi	Show an clickable user interface. To enable vocal commands, click on the mic icon
Hide interface	Nascondi i comandi	Hide the aforementioned UI
Switch language	Cambia lingua	Switch ARS and TTS language between English and Italian

Table 1: Vocal Commands List

The execution of these commands, along with the handling of automatic speech recognition (ASR) and text to speech (TTS), is in **VoiceController.cs** script. In this script, in the classic *Start* function, an instance of both the ASR an TTS is allocated in memory, following a singleton pattern. In this way subsequent call to the two components will not cause an expensive memory consumption. After the setup, in the VR scene there are two important game objects, *SpeechRecognizer* and *TextToSpeech*[2], that are associated with **VoiceController.cs**. In particular, when we click the panel of the scene, the *OnStartRecording* callback from the script is executed. This callback is linked, in the *Start* function, as method of the singleton instance of the ASR. *OnStartSpeaking* is instead the callback associated to the TTS instance, that is also registered during the *Start* function execution. Last interesting thing about *Start* is that we have to distinguish between the two possible platforms: **Android + Java** or **iOS + Swift**. We can adapt our code to the two different operating systems using pre-processor directives: for example, Swift ASR/TTS automatically asks the user for microphone permissions, while this is not always true for Android, so if we are developing for Android, we need to use a pre-processor directive to check for permission.

The code of the ASR/TTS instances is reported in the *TextSpeech* namespace, located in the *SpeechToText* folder of the Unity project, in the *SpeechToText.cs* and *TextToSpeech.cs* scripts. This part of code is very important, because it is full of the just mentioned preprocessor directives in order to send/receive messages to the android/ios ASR/TTS.

Vocal Feedback

Whenever the user inputs a speech command, the most natural way to give them a feedback is by letting the speech synthesis emit a vocal feedback. Each speech command has its relative hardcoded **vocal feedback**, so each command has a personalized response instead of a general purpose message like "command executed". This is useful in order to confirm the action requested by the user is exactly the one they asked for. For example, if the user gives the **"Play"** command while the video is already playing, the message **"currently playing"** is uttered, while no real action is taken. If, instead, an invalid or misinterpreted message goes through the rule based match, the user will hear **"No valid command found, please try again"**. Of course there are both the Italian and the English version of the vocal feedback, and when the user switch between the two languages, a new instance of ASR and TTS classes is set up, in order to avoid bad pronunciation made by, let's say, an English TTS trying to pronounce the sentence **"Velocità di riproduzione aumentata"**.

Plugins

Both the Android and the iOS plugins, are used in the same way in the Unity code as just discussed, making the specific platform distinction by means of pre-processor directives, hence requiring just version of the code. Instead, is the actual implementation of the ASR/TTS service that must be obviously implemented two times, one in Java for Android and one in Swift for iOS. Both of them are exported in the Assets' Plugins folder as an executable that Unity can run. When the app is installed for the first time into the device, **internet**

[2]these two names of the game objects cannot be changed, be ware

connection is required to download the ASR and TTS, but next executions of the app won't need connection.

We take the plugins from https://github.com/j1mmyto9/speech-and-text-unity-ios-android. As we can see from the github repo, only the Android Studio Project is provided to create the executable jar for Android, while the XCode project for Swift is not necessary because to use the speech recognizer and synthesizer in iOS you just need to correctly specify the configuration file of XCode.

Android Plugin

As we can see in the *MainActivity.java* file, al the callbacks needed to handle the voice are reported. These are the methods that are linked to the singleton instances created in C#. A call to a C# callback will result in the execution of a method reported in *MainActivity.java*. The *Bridge.java* file has exactly these purpose, i.e. to provide a proxy for the *MainActivity.java*'s functions as **static methods** since ASR/TTS are singleton instance in memory. Last important to notice is the import of **com.unity3d.player** that allows Android to interact with the Unity Activity. Normally, in an Android application you would have your main activity where you put the UI, but in our case the main activity is a bit particular since it's run by Unity.

iOS plugin

The concepts for iOS plugin are the same as for Android. What's important for the iOS plugin is to include the **AVFoundation.framework** and the **Speech.framework** in the configuration of of the XCode build, before deploying the app on an iPhone. As said, iOS already prompts the user to ask for permission to use the microphone and when we build and deploy with XCode we can specify this behaviour from the build configuration rather than call a check function as for Android.

More on iOS

Because of a lack of the required tools to build and develop on the Apple environment, we didn't succeed in testing the app on a iPhone. In particular, thanks to a friend (that borrowed us an iPhone 5) and a colleague (that borrowed us a Mac) we tried to make the VR Cardboard and the Speech Commands work. Since the tools weren't ours, we barely had an afternoon to try to correctly set the Unity Player Settings and XCode configurations to make the app run on an iOS, but we didn't make it in time. In particular, the code in Unity build for both Android and iOS without errors, the problem is just a matter of trying different configurations of the Player Settings in Unity and XCode configuration. The is an enormous amount of possible combinations of Unity and XCode configurations, so googling around to understand what were the problems became infeasible in just a couple of hours.
Because the VR Cardboard were not working properly, we neither tested the ASR/TTS on iOS.

Future works and other ideas

We have thought of some improvements/add-ons to be integrated in the application in order to maximize the experience and the degree of freedom when controlling the video player. As already said in the **Catalog** section, the current version of the app is limited to a 3x3 grid of displayable video previews to choose from. Making the grid larger or higher could cause the user not to be able to see at all the furthest previews or read those titles. To solve this problem, at least partially, we wrapped the grid in a **Scroll View** in order to scroll across one or both direction to expand the amount of displayable previews. This is actually limited by the fact that the **RawImage** components on which the preview is shown on are fixed. With some work, one could dynamically create the RawImage components, attach them their texture and consequentially their own video player. Another idea we looked into is the use of a **Curved UI** Canvas on which displaying the video, so the user can look around themselves alternatively to scrolling through the videos, which could result in a clunkier interaction. To do this, although, one would need to replace the whole catalog structure and use the curved components, available to separately download, and obtain 360° Catalog.
Another interesting thing we thought about is the management of a **multi track audio/video**. If we work with multi-track videos, we could hypothetically interact with each single audio track, hence obtaining a real-time speech guided equalizer. In a concert-like scenario, one could mix the volume of each instrument to their own liking. We implemented a naive generalized version to perform this kind of thing, where we always assume the video is multi-track, but since we couldn't access to a real-case multi-track video, we contemporaneously increase/decrease the volume of each audio track. By managing this routine separately for each audio track, one could easily obtain the aforementioned equalizer. Of course a series of extra vocal command and rules should be defined to get the most out of this feature.

Eventually, a more immersive experience can be provided by **completely deleting the interaction with the screen**. In order to do so, we thought of exploiting Unity Microphone to constantly recording environment sounds. When the detected sound intensity exceed a threshold, we can call the ASR module to detect the command. We didn't go deeper in this direction since we encountered problem in the handling of the microphone since both Unity and Android/iOS could access it. Moreover, when the user starts to speak, the ASR won't make in time to record the full sentence uttered by the user. This ends up in the ASR transcribing a sentence without the initial part of the waveform.

Conclusions

Our aim was to develop an application to grant an immersive 360° video experience. The degree of immersivity is of course subjective. One may prefer interacting with a video by a UI because it's the most straightforward way people use to consume their 2D media (e.g. Youtube, Netflix, ecc ..), while others may prefer being completely surrounded by the scene and not visualize strange floating buttons which could avert the user from reality. In the second case, there would be no possibility to interact with the video, so we tried to also implement a Speech Command driven invisible interface with vocal feedback. Fortunately, we found a way to integrate mobile's own ASR and TTS classes into a Unity application, giving us the possibility to let the user choose between the two interaction modes. Moreover, the initial catalog section is needed in order to easily deploy new videos to the server folder and be able to expand the use of the app through time.

Of course, the quality of the experience is more variable by accessing multiple videos on a server instead of tuning one video in order to have the correct direction, inclination, quality and so on. If, anyway, certain checks and pre-processing steps are performed on the videos before uploading them to the server, the experience quality should stay the same. This app is supported by Android devices from version 4.4 on. Old devices, most likely have low resolution displays, which may cause a low degree of immersivity. While retro-compatibility is important to let most people use the app, we need to take into consideration the hardware and software specs in order to at least advise the user if their device can grant them a good experience.