



UNIVERSIDAD DE MÁLAGA



Graduado en Ingeniería del Software

Traducciones en tiempo real mediante procesamiento del lenguaje natural en un entorno de realidad aumentada

Real-time translations using natural language processing in augmented reality

Realizado por  
Álvaro Lloret López

Tutorizado por  
Leonardo Franco

Departamento  
Lenguajes y Ciencias de la Computación

MÁLAGA, junio de 2022



UNIVERSIDAD  
DE MÁLAGA



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA  
GRADUADO EN INGENIERÍA DEL SOFTWARE

**Traducciones en tiempo real mediante procesamiento del  
lenguaje natural en un entorno de realidad aumentada**

**Real-time translations using natural language processing  
in augmented reality**

Realizado por  
**Álvaro Lloret López**

Tutorizado por  
**Leonardo Franco**

Departamento  
**Lenguajes y Ciencias de la Computación**

UNIVERSIDAD DE MÁLAGA  
MÁLAGA, JUNIO DE 2022

Fecha defensa: julio de 2022



# Abstract

Human language is one of the most amazing faculties of living beings, but the existence of so many different languages creates a barrier that limits the communication between people. Thanks to the work of translators and automated translation techniques, this problem can be alleviated in most cases, however, real-time automated translations are still a challenge problem to be solved.

The main purpose of this project is to build an application able to recognize what someone is saying, translate it in real-time and show the results in an augmented reality environment so that the user can see both the translation and the other person through the camera. The application aimed to be implemented in mobile phones has been developed using the Swift programming language. Once permission to use microphone and camera are granted, the user can freely activate and deactivate the microphone with the push of a button. Then, the speech recognition automatically starts and the user will see the translated text and the outside world through the camera at the same time.

Key aspects of the present project are the demonstration how different technologies can be integrated, and also how fast speech recognition and translation can work so that it can be consider a real-time application, meaning that the process happens in a milliseconds scale.

**Keywords:** Natural Language Processing, Artificial Intelligence, Augmented Reality, Language translation, Speech Recognition.



# Resumen

El lenguaje humano es una de las facultades más asombrosas de los seres vivos, pero la existencia de tantos idiomas diferentes crea una barrera que limita la comunicación entre las personas. Gracias al trabajo de los traductores y a las técnicas de traducción automática, este problema se puede paliar en la mayoría de los casos, sin embargo, las traducciones automáticas en tiempo real siguen siendo un reto por resolver.

El objetivo principal de este proyecto es construir una aplicación capaz de reconocer lo que alguien está diciendo, traducirlo en tiempo real y mostrar los resultados en un entorno de realidad aumentada para que el usuario pueda ver tanto la traducción como a la otra persona a través de la cámara. La aplicación, destinada a ser implementada para teléfonos móviles, ha sido desarrollada utilizando el lenguaje de programación Swift. Una vez se otorga el permiso para usar el micrófono y la cámara, el usuario puede activar y desactivar libremente el micrófono con solo presionar un botón. Hecho esto, el reconocimiento de voz se inicia automáticamente y el usuario verá el texto traducido y el mundo exterior a través de la cámara al mismo tiempo.

Los aspectos clave del presente proyecto son la demostración de cómo diferentes tecnologías pueden ser integradas, y también qué tan rápido pueden funcionar el reconocimiento de voz y la traducción para que pueda considerarse una aplicación en tiempo real, lo que significa que el proceso ocurre en una escala de milisegundos.

**Palabras Clave:** Procesamiento del Lenguaje Natural, Inteligencia Artificial, Realidad Aumentada, Traducción de Idiomas, Reconocimiento de Voz.



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	7
1.2	Objectives . . . . .	8
1.3	Structure of the document . . . . .	8
<b>2</b>	<b>State of the art</b>	<b>11</b>
2.1	Machine translations and artificial intelligence . . . . .	11
2.2	Speech recognition . . . . .	13
2.3	Augmented reality . . . . .	15
<b>3</b>	<b>Methodology and Software tools</b>	<b>19</b>
3.1	Methodology . . . . .	19
3.2	Software tools . . . . .	20
3.2.1	Swift and SwiftUI . . . . .	20
3.2.2	Speech and AVFoundation frameworks . . . . .	22
3.2.3	Combine framework . . . . .	23
3.2.4	Azure AI Translator . . . . .	24
<b>4</b>	<b>Implementation of the application</b>	<b>27</b>
4.1	Speech recognition module . . . . .	28
4.2	Augmented reality module . . . . .	34
4.3	Translation module . . . . .	38
4.4	Results . . . . .	41
<b>5</b>	<b>Conclusions and Futures Studies</b>	<b>45</b>
5.1	Conclusions . . . . .	45
5.2	Future Studies . . . . .	46
<b>6</b>	<b>Conclusiones y Líneas Futuras</b>	<b>49</b>
6.1	Conclusiones . . . . .	49

6.2 Líneas Futuras . . . . .	50
<b>Bibliography</b>	<b>53</b>
<b>Appendix A User Manual</b>	<b>57</b>
<b>Appendix B Installation Guide</b>	<b>59</b>

# 1

# Introduction

## 1.1 Motivation

Breaking down the language barrier between societies with different languages has been one of the main problems that humanity has tried to solve since the beginning of civilization. In the interconnected and globalized world we live in today, this problem is even more relevant now than it was a few decades ago. Thanks to advances in automated translation techniques and natural language processing models, asynchronous communication problems can be alleviated in most cases. Nevertheless, real-time translation techniques need to improve quite a bit in order to be applied in consumer applications. The main objective of this work was to discover whether a translator system using speech recognition, natural language processing and augmented reality can be feasible for the requirements of a real-time application deployed on a smartphone.

Besides the analysis of the previous main goal, the idea of this project is to advance in areas like asynchronous programming for mobile application development [Dig15], checking if complex applications with several modules can take advantage of concurrency techniques to optimize execution time. Moreover, it is also of interest to study different software tools that can be used to develop a project of this complexity trying to exploit them to see what their limits are.

Another huge motivation of this project is that it is worth trying to get closer to a world where someone could travel to any country and speak naturally to local people. To achieve that, the translation tool should allow the user to see both the translations and the other person simultaneously in real-time, and this fact will be explored using augmented reality technologies.

To summarize the overall motivation is to explore, utilize, combine and optimize applica-

tions from speech recognition, automatic translation and augmented reality to try build a real time application that can be implemented in mobile telephones.

## **1.2 Objectives**

The primary goal of this project is to overcome the challenge of integrating different libraries to build a real-time translator application that can operate in real time, i.e., all the modules combined should work in the milliseconds scale. The mobile application should be able to transcribe what someone is saying to text, translate the text and show the results along the outside world using augmented reality.

Achieving the previous goals, implies that the following four sub-goals need to be fulfilled:

1. To implement a speech recognition module capable of transcribing audio to text continuously.
2. To implement a translation module that works in real-time.
3. To implement the augmented reality module to show the external world and the translations in synchrony with the previous modules.
4. To combine all the modules and make sure that the flow of information in the application built is fast enough for a real-time application.

## **1.3 Structure of the document**

This work is organized in six main parts: Introduction (1), State of the art (2), Methodology and Software Tools (3), Implementation of the application (4), Conclusions and Future Studies (5) and "Conclusiones y Líneas Futuras" (6).

The first chapter (Introduction) essentially states the motivation and objectives of the project. The second chapter (State of the art) covers some historical background and the current state of the technologies involved in the development of the project: automatic translations and artificial intelligence, speech recognition and augmented reality. The third chapter (Methodology and Software tools) aims to present and explain the methodology that has been followed to carry out this project and also the main technologies and tools that will be used, in

addition to the analyzing and justifying why they have been chosen over different alternatives. The fourth chapter (Implementation of the application) is the core of the study, where the development phases are comprehensibly detailed, and relevant developments of the software modules are explained. Chapter five (Conclusions and Future Studies) contains a summary of what it has been accomplished by doing this project. Furthermore, there is a discussion about what could be done in the future to extend the present work . Chapter six ("Conclusiones y Líneas Futuras") is the spanish translation for chapter five.





# 2

## State of the art

This section introduces the different disciplines related to the implementation of this project. Specifically, the technologies involved are discussed, starting with machine translations and their relationship with artificial intelligence, focusing on the field known as natural language processing. Then the field of speech recognition is analyzed and discussed, in particular how this technology have improved in recent years. Finally, the current state of augmented reality and the different existing types and ways to use it nowadays is presented and analyzed.

### 2.1 Machine translations and artificial intelligence

Language translation has come a long way since the beginning of the internet era. In the past, the only two ways to communicate with people with a different language was by using a dictionary, the slow method, or by hiring an interpreter, the expensive method.

Nowadays, thanks to the advancements of artificial intelligence and natural language processing, people can almost communicate in a free and efficient way. Adding the fact that the majority of the population owns a smartphone, it can be argued that everybody carries a mobile interpreter in their pocket. The popularization of translation apps in smartphones have provided people with the tools necessary to be able to solve most asynchronous communication tasks in a fast and inexpensive way.

A clear example of this type of application can be "Google Translate", which makes use of Google Neural Machine Translation (GNMT) [Wu+16], a natural language processing model that applies example-based machine translation techniques [TP01] to improve the accuracy of the translations. Example-based machine translations, firstly introduced by Makoto Nagao [Wik21] in 1984, are characterized by using huge datasets of pairs of sentences in two different languages. In short, every sentence in one language inside the dataset is related to the translated sentence in the other language. This technique is best suited for translating sentences

that are context-dependent like those containing phrasal verbs. By using example-based machine translations, Google is providing a state-of-the-art translator to millions of users around the world.

Artificial intelligence, specifically, the intersection between Deep learning and Natural Language Processing, has been the key to unlock machine translations that are almost as good as human translations. The major contributor to the improvements in machine translation techniques is the fact that deep learning models with billions of parameters and terabytes of data can be trained in a reasonable amount of time. The best example of this is the NLP (Natural Language Processing) model trained by Open AI, GPT-3 [Bro+20], which has 175 billion parameters and is trained with a 45 terabytes text dataset.

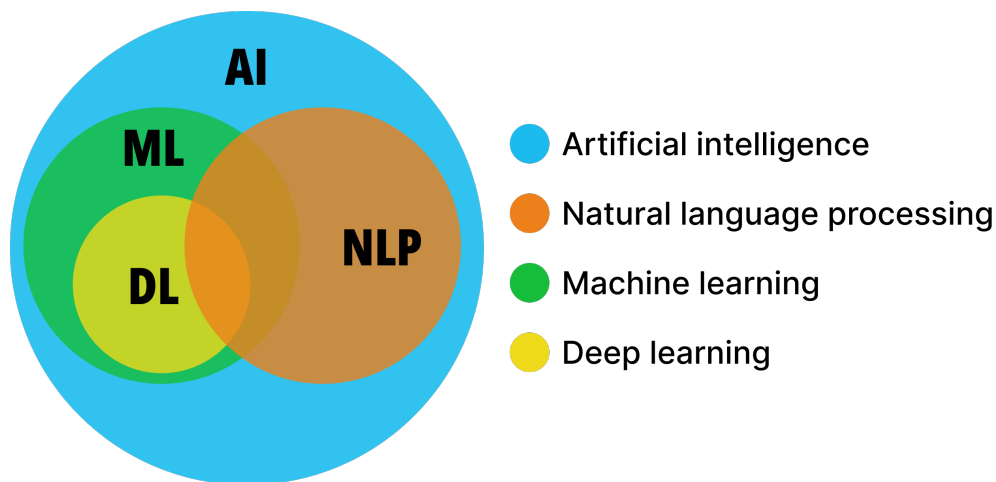


Figure 1: Relationships between AI, NLP, ML and DL. Source: Author’s own elaboration.

Asynchronous translations are not considered a problem anymore, as several online translators can get the job done nowadays quite efficiently. Current natural language processing models are capable of translating text into any language with a performance almost similar to human accuracy. The problem arises in synchronous or real-time translations because the speed and memory size of these enormous deep learning models are not feasible for the requirements of real-time applications.

A key aspect to solve this problem is finding a light and fast model able to translate text with an accuracy similar to those huge deep learning models like GPT-3. State of the art examples that try to achieve this can significantly reduced parameter size and compress the model size up to four times without compromising too much on performance [TZT19].

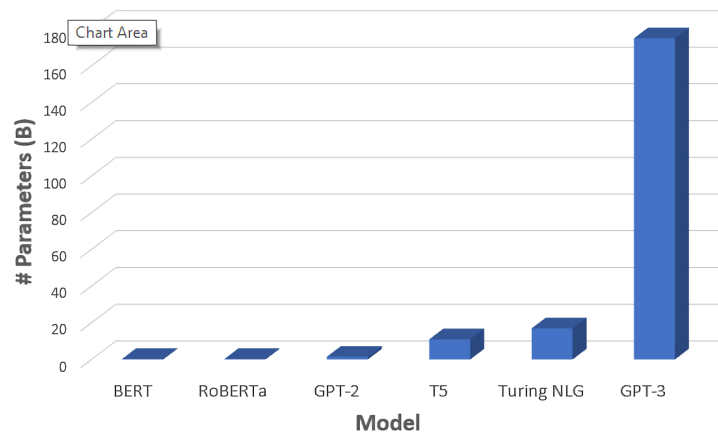


Figure 2: Bar chart comparing the number of parameters of different machine learning models. Source: [Bha20]

The common problem of these lightweight natural language processing models is that their goal is not translating text, but doing simpler tasks like natural language inference, paraphrase detection and dialogue response selection. Translating usually involves several of these usual natural language tasks, so small models tend to be designed to perform these tasks meanwhile larger models tend to be designed to being able to translate text.

Real-time translations usually involves lots of trade-offs. If larger NLP models are being used for the translation, the speed is not usually good enough to being able to carry a conversation in a synchronous way. However, if smaller NLP models are being used for the translation, the accuracy of the translations might be worse, so even if they are fast, there might be interpretation errors in the conversation. The most common approach for real-time translators is defining a speed requirement at the start of the development phase and after setting that up, building the largest NLP model possible that satisfies the speed requirement.

## 2.2 Speech recognition

In recent years, speech recognition technologies have improved to the point of becoming one of the main input methods along with clicking and typing. A simple definition of speech recognition could be the process of converting human sound signals into words by a machine, which could be used as instructions.

Specifically, speech recognition is a sub-field found in the intersection of computer science

and linguistics. The main goal of this sub-field is automating the transformation of regular spoken language into text using the power of computers, that is why sometimes it is usually called speech-to-text as well.

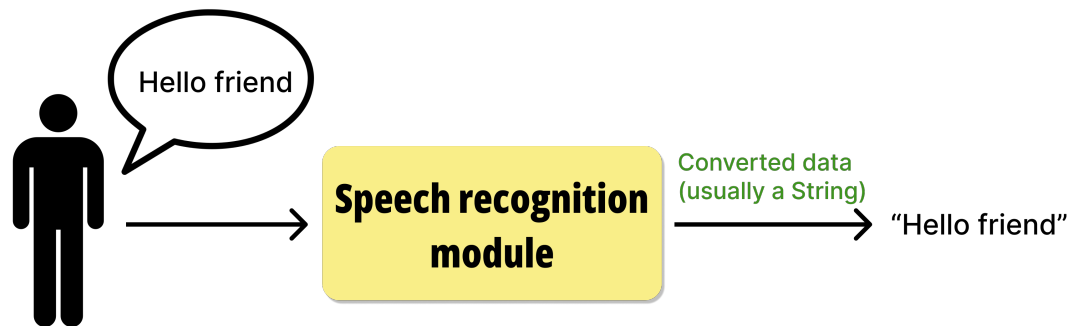


Figure 3: Visual explanation of speech recognition. Source: Author's own elaboration.

Speech recognition technology works by, firstly, digitizing the voice of a speaker person, then breaking up the recorded speech into discrete segments formed by different tones which are visualized in the form of spectrograms. These spectrograms are analyzed and transcribed individually using a natural language processing module that links the current spectrogram to the words with the highest probability in the vocabulary of a specific language. In most cases, there is another natural language module that works as a contextual layer, this final layer tries to make sure that all the transcriptions makes sense and that they do not have grammatical errors.

Nowadays, thanks to the improvements mentioned in the previous section, natural language processing models have achieved almost human accuracy, therefore, speech recognition has improved tremendously in the last ten years. A clear example of this can be seen with Google. Since 2012, Google's word error rate has fallen by more than 30% thanks to the addition of neural networks [Nov17] into their systems.

The use of artificial intelligence, specifically, natural language processing models have made speech recognition more accurate, however, speech recognition libraries also need to be fast to provide a good user experience. In order to accelerate speech recognition, two main approaches exist, the first one is simply doing the speech recognition processing locally [GP21], trying to avoid making requests to external servers. The problem of this approach is that the accuracy tends to be lower because, unlike speech recognition in external servers,

there is not a contextual layer using natural language processing to try to correct the results of the raw transcription made locally. The second approach is subdividing the speech recognition in asynchronous tasks, for example, one task is continuously recording the live audio and sending it to another task that is constantly transcribing a piece of audio to text, after that, an additional task receives the raw transcription and tries to correct errors. This way, the different parts of the speech recognition system can be executed concurrently, improving the speed of the module significantly.

The accuracy and speed improvements of speech recognition has led to the incorporation of this technology in many commercial products, examples of this are smart home hubs, speakers, text editors, smartphones, etc. The estimated number of voice assistant users for 2022 in the United States is 135.6 million people [Sta20], AI personal assistants like Siri or Alexa, have become mainstream thanks to the mentioned enhancements in speech recognition techniques and natural language processing models.

### 2.3 Augmented reality

Augmented reality, usually abbreviated as AR, is a sub-field of computer science that aims to enhance the physical world by presenting information in a natural way in the environment of the user. In most cases, this is usually achieved by showing relevant digital visual elements or by using spatial audio technology.



Figure 4: A clear example of an AR application. Source: [Vod21]

Augmented reality applications that need a deep understanding of the surroundings of the user often use SLAM (Simultaneous localization and mapping) [Rei+10]. By using SLAM algorithms the application can construct a map of an unknown environment and simultaneously keep track of the position of the user. Once the application has a virtual environment version of the physical surroundings, it can start showing relevant contextual information near the user's position. Maintaining coherence with virtual objects is the main reason that SLAM is used, this means that moving in the physical world should not affect the position of the object, nevertheless, the size should change based on where you are now. For example, if you place a virtual statue in top of a table, the statue should remain in the same position even if you move across the room, but the size should change accordingly.

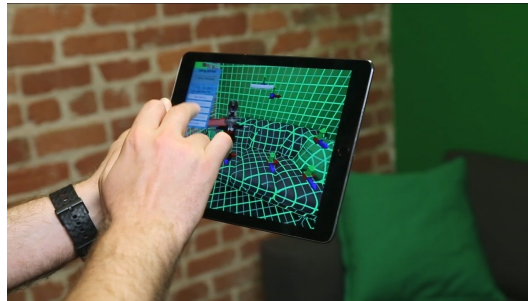


Figure 5: A room being scanned by an SLAM algorithm so that digital objects can adapt to the environment. Source: [Occ16]

On the other hand, many augmented reality programs do not need to keep track of the environment or the user. The essence of AR is showing useful information while simultaneously allowing the users to see their physical surroundings. Some examples of this could be presenting a news feed or showing the current weather, even though these are useful applications, they do not need to create a map of the environment, the reason for this is that the data could be constantly presented in superposition to the view of the physical world that the user has. In these type of applications, the users have the ability to show or hide the information that is currently in their field of view.

There are two main ways to use augmented reality applications nowadays, using a smartphone or using early AR glasses. Up until now, AR glasses have been focused on industry use cases, however, there are a few early consumer's AR glasses in the market. The problem with this early AR glasses is that the technical specifications are not excellent because they need to

put the power of a computer into the size of normal looking glasses, so the applications tend to be simple and they do not show the full potential of AR.



Figure 6: Early AR glasses example, Nreal Air. Source: [Nre22]

Alternatively, smartphones are a mainstream technology that have already proven their usefulness to the world. Smartphones do not squeeze the potential of AR to the maximum, however, it is a good gateway to show what augmented reality is capable of. For instance, Pokemon Go is the most popular AR application. The technical specifications of smartphones excels the technical specifications of AR glasses, specially when you compare CPU speed and battery time. AR applications on smartphones are more complex and can easily be integrated with software libraries and hardware sensors that smartphones already use. Furthermore, more than 83% of the population owns a smartphone [Ban22], so this is why it is a good idea to develop AR applications on this platform meanwhile AR glasses technology is improving.

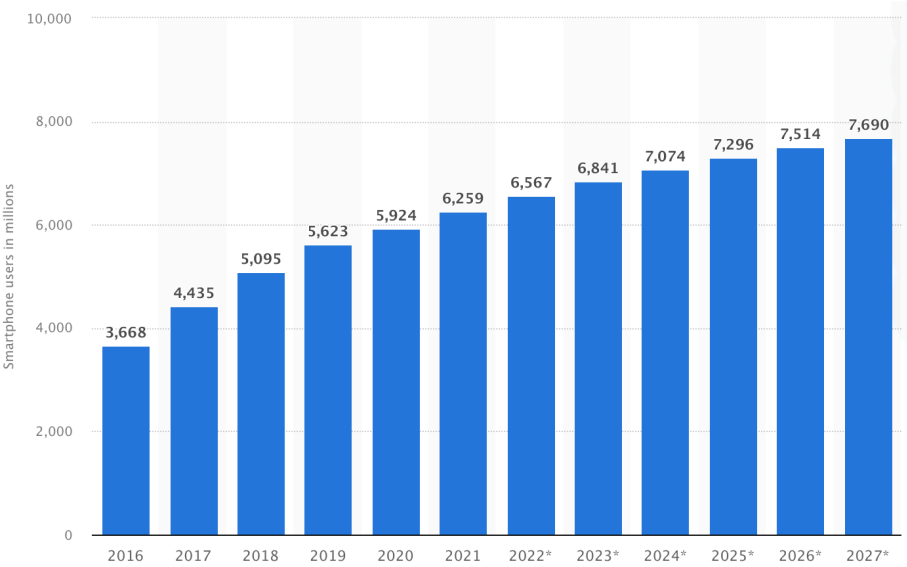


Figure 7: Growth of smartphone users 2016-2027. Source: [Sta22]





# 3

## Methodology and Software tools

### 3.1 Methodology

In this section, the methodology used to carry out this project will be discussed. The philosophy of agile methodologies [Abr+02] was followed to complete this work, given the amount of possible changes this application could have due to its innovative nature. A set of general features was defined at the start, but with the evolution of the project, this set of features has been evolving as well. In order to get done the different sections, these sections were divided into smaller tasks, that way the tasks were specific and well-defined, moreover the progress was incremental and steady.

Now, the main phases of the development of this project will be presented. Although, the following enumeration is a numbered list, this does not mean that their execution was done sequentially, in fact, some of these phases were executed simultaneously to make sure that there were not compatibility issues. Despite of this, it was a good idea to set up an order for the different parts to establish a path to follow.

1. Researching and documenting about the state of the art of the different technologies used in this project.
2. Developing the speech recognition module.
3. Developing the augmented reality module.
4. Developing the real-time translation module.
5. Merging the different software modules and making sure they work properly.

## 6. Writing the memory.

### 3.2 Software tools

Regarding the technologies used to develop this project, a description for each of them will be provided along with the reasons why they have been chosen over different alternatives. All of these technologies have been carefully chosen to fit the real time requirement needed to successfully carry out this project.

#### 3.2.1 Swift and SwiftUI

First of all, the operating system where this application will be deployed is iOS. There are several reasons for this, the first one is that the A15, Apple's CPU for iPhone 13, is the fastest and most powerful mobile CPU in the market, beating the Snapdragon 888, the current Android's flagship CPU. In real-time applications, the flow of information must be as fast as possible to provide the best user experience, by using the most powerful smartphone on the market we make sure that the speed of the instructions will not be a problem.

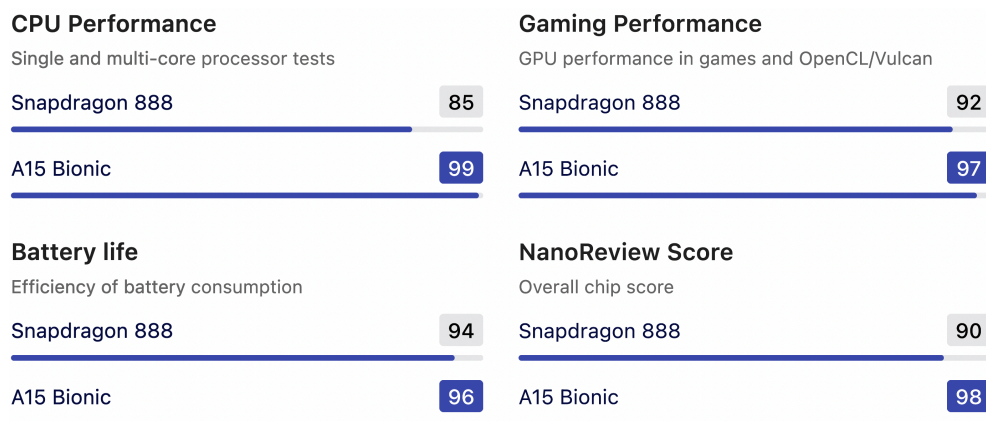


Figure 8: Benchmark comparison between A15 and Snapdragon 888. Source: [Nan21].

Besides the powerful hardware, iOS also provides many software libraries that will be very useful to carry out this project successfully, these technologies will be explained individually in the following paragraphs, but one key thing about them is how well they work together. The interoperability between software modules and also hardware, is the reason why their applications are able to squeeze the power of their hardware products to the maximum.

The last reason is that Apple is planning to launch an augmented reality (AR) headset in the upcoming years. They already confirmed that this upcoming product will use a modified version of iOS, so by using Apple software libraries, this project could be smoothly ported to this AR headset in the future.

Now, we will present Swift [App14a], which is an open-source programming language created by Apple [App14e]. This modern programming language was released to the public in 2014, the main goal of this release was to make it the primary language developers use to build applications in Apple's ecosystem. The main features of this programming language are the following:

- Object-oriented programming [Str91].
- Automatic Reference Counting [App14b] for high-performance memory management.
- A dynamically typed language with the possibility to become strongly typed if desired.
- Expressive, simple and easy to learn syntax.
- Support for lambda expression/anonymous functions.
- Traditional data types with the addition of Optionals and Generics.
- Uses the LLVM [LA04] as its compiler framework.
- Complete support for Apple's software tools and main programming language to develop iOS, macOS, watchOS and tvOS applications.

As it can be seen from the list of features, Swift combines a powerful type inference and a simple but expressive syntax so that laborious ideas can easily be written in a clear and short way. If you add the support of Apple's software libraries and its high-performance, Swift is the trouble-free choice to develop rich real-time applications on Apple's devices.

Concerning SwiftUI, as the name suggests, it is a framework to develop user interfaces built on top of Swift. This user interface toolkit allows developers to design apps in a declarative way for any Apple platform. SwiftUI works as a cross-platform layer for the design of your applications, this means that the same user interface will work across iOS, macOS, tvOS and watchOS.

One of the toughest aspects of user interface development is synchronizing the state of the application across all the views, this means that when the state is changed, the changes must be reflected accordingly in all the user interfaces. In other frameworks like UIKit or AppKit, the developer needs to pass this information across views manually, however, this does not happen with SwiftUI. In SwiftUI, the view is not the result of a sequence of events. A view is a function of state, this means that when the state of the application changes, the views will change as well, eliminating all the problems and bugs that this might have caused.

The powerful thing about Swift and SwiftUI is that by learning one language and one user interface framework you can deploy your code in smartphones, laptops, smartwatches and TVs without the hustle to make many changes for each platform.

### 3.2.2 Speech and AVFoundation frameworks

In order to manage real-time audio and video, two libraries have been used. For audio, the Speech framework [App16a] was selected because it is able to perform speech recognition on live audio, although it can also work on prerecorded audio if needed. On the other hand, the AVFoundation framework [App08a] was the chosen one to manage video, this library is capable of working with time-based audiovisual media formats like QuickTime movies, MPEG-4 files, HLS streams and more, however, in this application it will be used to get access to the camera view.

Both of these Apple libraries are able to work simultaneously, there are not interoperability issues, so applications can do real-time speech recognition through the microphone and make use of the camera at the same time without compromising the application performance.

It is relatively simple to get transcriptions of live audio using the Speech library, however, it requires a lot of work to also get the transcriptions in real-time. By default, this library sends the live audio to Apple servers by making an HTTP request, once the information has been processed and the transcription is ready, the information needs to go back to the smartphone. In this case, this process is too slow for a real-time application because there is a notable difference between the time the user speaks and the time the transcription is shown in the screen. To solve this, the Speech framework gives the possibility to do the processing of the information on-device, which is ideal for privacy-focus applications or real-time applications like this project.

The main use case of the AVFoundation in this project is providing the augmented reality layer. This will be achieved by allowing the application to access the view of the camera, which will be crucial to show the translations and the outside world concurrently.

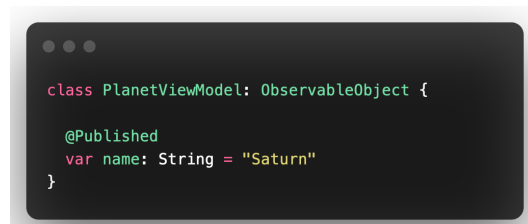
### 3.2.3 Combine framework

The Combine framework provides a Swift API that can easily be imported in any project to process data over time. Using Combine, developers can customize handling of asynchronous events by combining event-processing operators. In simple terms, Combine have two key concepts, *publishers* and *subscribers*. Publishers are in charge of exposing values that can change over time, meanwhile subscribers are the ones to receive those values from the publishers.

The primary use case is using it to know when an important variable of the state of your application has changed. Many times developers want to trigger an specific action when certain variable changes, by using this framework this behaviour could easily be implemented without worrying about when the asynchronous event is going to change the variable.

A useful and common example of this is wanting to trigger a view update whenever a field of a class changes. To achieve this, a developer would only need to do two things using the Combine framework, creating a class that implements the ObservableObject protocol [App19b] and then marking a property of the class with the @Published property wrapper [App19a]. A protocol in Swift is very similar to an interface in Java, but in addition protocols can also specify properties that must be implemented.

In the figure below, a simple example using ObservableObject and @Published is presented. In the code snippet a class called *PlanetViewModel* implements the ObservableObject protocol, so now the property flagged with @Published, in this case *name*, will automatically trigger an update when its value changes for all the user interfaces that were using it.



```
class PlanetViewModel: ObservableObject {  
  
    @Published  
    var name: String = "Saturn"  
}
```

Figure 9: A code snippet of ObservableObject and @Published. Source: Author's own elaboration.

### 3.2.4 Azure AI Translator

Concerning the technology used to perform translations, the Azure AI Translator was chosen between a large number of different candidates. To be able to make a decision, first, we need to have a deep understanding about machine translation and what the application needs.

The main requirement of this application is being fast enough to provide a good user experience for a real-time application. To achieve this, the speech recognition, the translation and showing the result to the user should happen in less than one second. Besides being fast, the translator API (Application Programming Interface) must be accessible through a REST API (Representational state transfer) [IBM20], that way Swift will be able to send and receive information from the service by following a standard through the HTTP protocol, without needing a custom library made just for Swift. This is also important for the reusability and interoperability of the module, because by using a REST API we make sure that this module could be used by other programming languages by making the same HTTP requests.

In the figure below, there is a comparison between different translation APIs showing the latency of the request. In this case, that is the time between sending the text to translate and receiving the translation result.

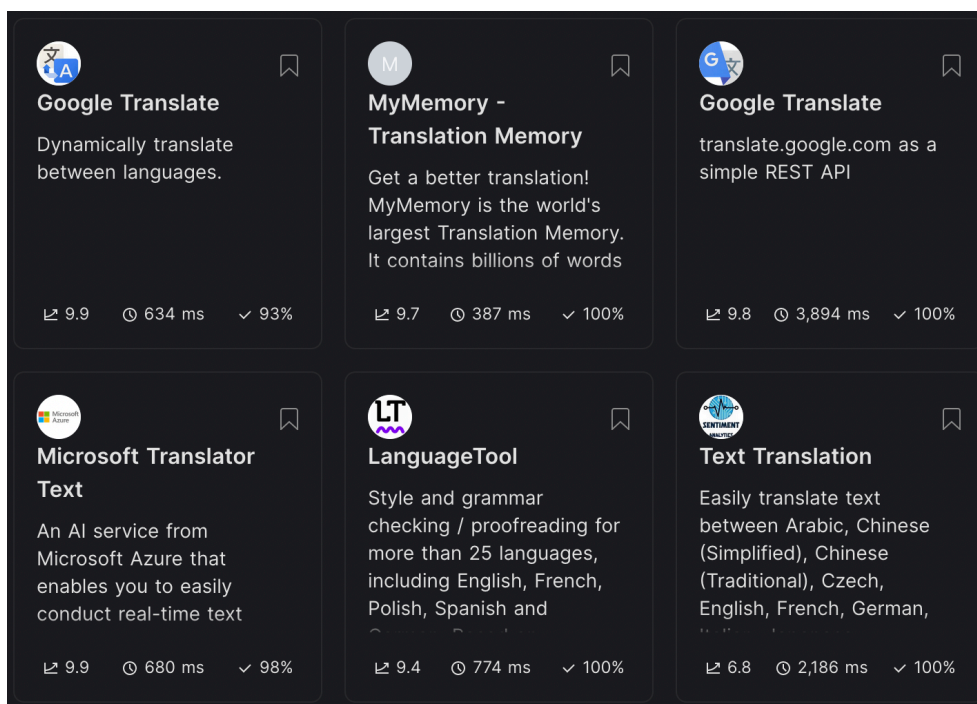


Figure 10: Comparison of different translation APIs. Source: [Rap22].

First of all, the APIs with more than 700 milliseconds of latency were discarded. After that, there were three APIs remaining that satisfied this requirement, they were the Google Cloud translate service, Microsoft Azure AI translator and MyMemory translation service. MyMemory is poorly documented and their usage limit is very constrained, less than 5000 characters per day, so this option was discarded too. In the case of the Google Cloud and Azure solutions, both of them were very well documented and they offered a REST API to get access to the service, however, the Azure free tier of the translator service (2,000,000 characters per month) was more generous than Google Cloud (500,000 characters per month), and in case the application needs to scale up, the superior tiers were also more affordable with Azure.

To sum up the reasons, the Azure AI translator [[Mic22](#)] was chosen due to having low latency, providing a well documented REST API and being the cheapest and most scalable solution.





# 4

## Implementation of the application

In this section, the different modules implemented to build this application will be thoroughly explained. This project consist of three layers: speech recognition, augmented reality and translation. The code will also be detailed along the ways these different modules have been integrated with each other. To follow up, a summary of the implementation is presented.

In the following figure, the first step of the implementation was creating the base SwiftUI project. The implementation continued by developing a basic speech recognition module using an Apple template. This initial module worked in simulators but after testing it on a physical device the performance was slow. The reason was that the template was constantly sending information to Apple servers. After deeply understanding how the Speech framework worked, a new module was created from scratch using only the needed functions. As it was expected, the speed of the speech recognition module improved significantly.

Simultaneously, the augmented reality module was being developed. The first thing to do was coding all the necessary camera classes and structs that will be explained in their corresponding subsection. After that, it was time to make sure that the camera view was properly presented in the screen. Next, we tried to place text over the view of the camera. After achieving this, the speech recognition and augmented reality modules were the first to be integrated, this was done by placing the transcriptions from the speech recognition module into the camera view of the augmented reality module.

The translation module implementation was started with a long phase of research. We found out two software libraries that could work for real-time applications, Google translator API and Azure AI translator. Azure AI translator was chosen for being the cheapest and most scalable solution. Finally, all the modules were integrated to show the translated transcriptions

over the camera view and we checked they worked fast enough for a real-time application.

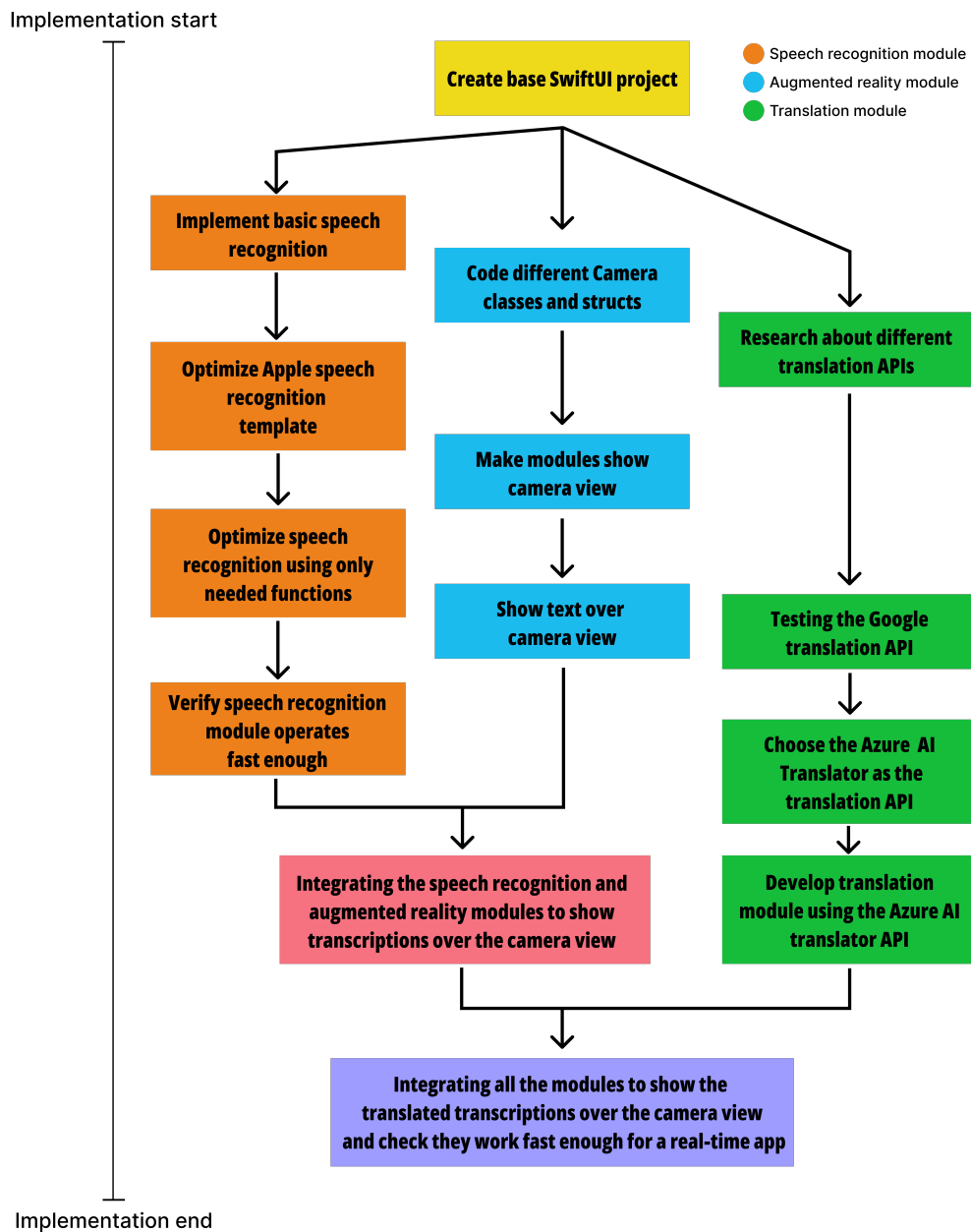


Figure 11: General scheme of the implementation of the application developed in this project. Source: Author’s own elaboration.

#### 4.1 Speech recognition module

Speech recognition is the core of the application because, in order to translate what the users are saying, we first need to transcribe their speech. This process must be done accurately but, specially for this project, it also needs to work as fast as possible, in the milliseconds scale. In

the following paragraphs, it will be detailed how this has been achieved. The main steps to develop the speech recognition module are presented below:

1. Asking permission to use the microphone and do speech recognition.
2. Checking all necessary preconditions.
3. Creating and starting the speech recognition task.
4. Finishing the speech recognition task when requested.

Before starting to use the Speech framework, first, we need to add privacy keys to the Info.plist file of the project. This must be included because if it is not present, the application will stop working when attempting to request authorization or trying to use the APIs of the Speech framework.

To achieve this, the project needs to be opened with the Xcode IDE (Integrated Development Environment), after that, we need to add two different keys to the Info.plist file, these are "Privacy - Speech Recognition Usage Description" and "Privacy - Microphone Usage Description". Furthermore, the privacy keys need a text description of why the application is going to make use of them, this is important to build trust with the users and make them feel safe using the application.

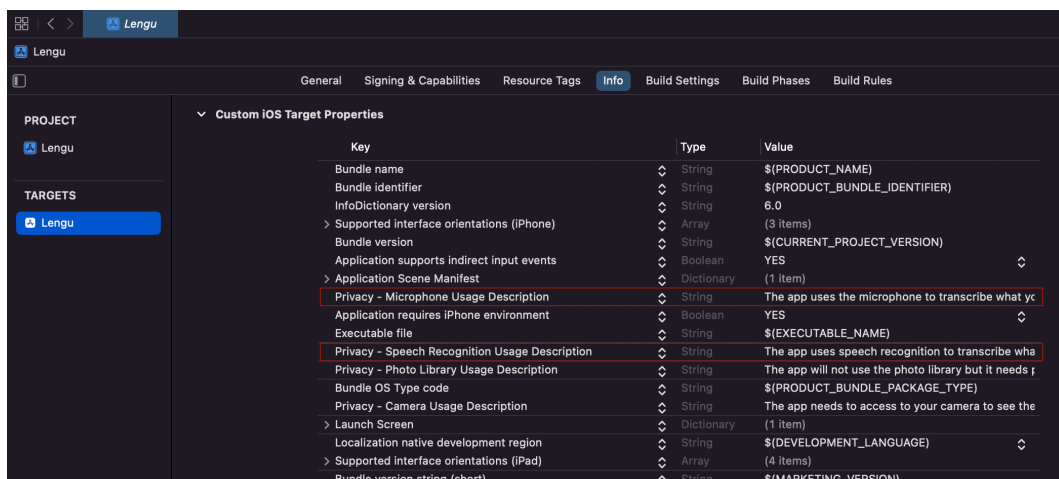


Figure 12: Privacy keys added to Info.plist related to speech recognition. Source: Author's own elaboration.

After successfully adding the privacy keys, the next step is requesting permission to the user. This must be done the first time the user opens the application, so when the main view of the application is first displayed, the permission request should appear as well. This process has been encapsulated in a function as it can be seen from the following code snippet.

```
func requestPermission() {
    SFSpeechRecognizer.requestAuthorization { (authState) in
        OperationQueue.main.addOperation {
            if authState == .authorized {
                permissionStatus = .authorized
            } else if authState == .denied {
                permissionStatus = .denied
            } else if authState == .notDetermined {
                permissionStatus = .notDetermined
            } else if authState == .restricted {
                permissionStatus = .restricted
            }
        }
    }
}
```

Figure 13: The implementation of the `requestPermission` function. Source: Author’s own elaboration.

The variable `permissionStatus` is an enumeration from the Speech framework [App16d] that must be authorized in order to make use of the framework. The permission is actually requested using `SFSpeechRecognizer.requestAuthorization` [App16b], putting the decision of the user into the `authState` variable, after that the only thing left is checking its value and giving `permissionStatus` the corresponding value. The reason why there is an *if-else* structure is because we cannot simply assign the value of `authState` to `permissionStatus` because `authState` may have corrupted values that may crash the application, we only want the values we specified. Finally, the `requestPermission()` function will simply be invoked by using `onAppear{requestPermission()}` [App19c] on the main view of the application.

Now that the application has permission to access the microphone and do speech recognition, it is time to explain how this is done. Following a similar approach to when requesting user permission, the process of starting and performing speech recognition has been encapsulated in a function called `startSpeechRecognition`.

At the start of this method, we first need to prepare and start the audio engine. This audio engine is capable of performing complex real-time audio processing. It is commonly used to simplify audio operations and input/output tasks, however, in this case it will be used to

capture the audio of a real-time conversation. In order to do this, we must access the input node of the audio engine and setting the buffer size and the format, which in this case, it will be the recording format. After doing that, we must append the buffer to the *request* variable which is an instance of the *SFSpeechAudioBufferRecognitionRequest* class. This class is in charge of making a request to recognize speech from captured audio content, such as audio from the microphone. Finally, *audioEngine.prepare()* is executed to preallocate the required resources to start the audio engine and *audioEngine.start()* to start up the engine.

```
func startSpeechRecognition(){
    let node = audioEngine.inputNode
    let recordingFormat = node.outputFormat(forBus: 0)

    node.installTap(onBus: 0, bufferSize: 1024, format: recordingFormat) { (buffer, _) in
        request.append(buffer)
    }

    audioEngine.prepare()
    do {
        try audioEngine.start()
    } catch {
        errorMessage = "Error starting the audio engine."
    }
}
```

Figure 14: The beginning of the *startSpeechRecognition* function. Source: Author's own elaboration.

The middle part of the *startSpeechRecognition* function consists in making all the necessary checks to make sure there are no problems when trying to start the speech recognition task.

To understand the *guard-let* conditional statement, it is important to know a little bit more about Swift. The *guard* statement is very similar to the *if* statement, however, the code block inside an *if* statement will be executed when a certain condition is met, meanwhile the code block inside a *guard* statement is only executed if the condition is false, furthermore, they must have a return statement, so they are designed to exit the current function or loop when the condition fails. It can be understood as an *if-else* statement that only has the code block of the *else* statement. The *guard* statement is usually used to verify that some conditions are correct up from and only doing something when this does not happen. The *let* statement by itself is used to define constants, however, when it is used inside a conditional statement (*guard* or *if*), it allows to define a variable that can also unwrap an optional value. If the optional value ends up being *nil* (equivalence of null in Swift) the condition will be considered false, but if the optional value is a normal datatype (Int, String, etc) the condition will be considered true.

Now that the *guard-let* statement has been clarified, the meaning of the first verification

```
guard let mySpeechRecognizer = SFSpeechRecognizer() else {
    errorMessage = "Recognition is not allowed on your locale."
    return
}

if !mySpeechRecognizer.isAvailable {
    errorMessage = "Recognition is not available right now. Please try again after some time."
}
```

Figure 15: The middle part of the `startSpeechRecognition` function. Source: Author's own elaboration.

can be explained. The first check means that creating an object calling `SFSpeechRecognizer()` should create an instance of the `SFSpeechRecognizer` class [App16c], but if the value is `nil`, this means that the speech recognition is not available for the user locale and the flow of execution should exit the current function. The second verification is tested using a normal `if` statement, when the variable `mySpeechRecognizer` is not available this means that speech recognition task cannot be created at the moment, this could be caused by other applications making use of the microphone or the current language not being supported.

Once this has been done, it is time to finally create the speech recognition task. This is done by assigning the value of `mySpeechRecognizer.recognitionTask(with: request)` to the variable `task` of the `SFSpeechRecognitionTask` class. This class is used to monitor the speech recognition progress, specifically to determine the state of a speech recognition task, cancelling an ongoing task or signaling the end of the task, which will be useful later on. The `response` variable will contain the transcription but first there is a verification using the `guard-let` statement to see if `response` is `nil` or not, in case it is `nil`, we set the corresponding error message. If it turns out that `response` is not `nil`, the flow of execution continues by defining a constant called `message`, the value of this constant will be a string of the best transcription made by the speech recognition module. In the end, we translate the value of `message` using the `makeTranslationRequest` function which will be explained in the "Translation module" section later on. The value of the `transcription` variable will be displayed in an augmented reality environment, this variable will contain the result of performing the translation to the message captured using speech recognition.

As it has been mentioned in the previous paragraph, the `SFSpeechRecognitionTask` class is used to monitor the speech recognition progress, specifically to determine the state of a speech recognition task, cancelling an ongoing task or signaling the end of the task. By using the `task`

```

task = mySpeechRecognizer.recognitionTask(with: request) { (response, error) in
    guard let response = response else {
        if error != nil {
            errorMessage = "For this functionality to work, you need to provide permission in
your settings."
        } else {
            errorMessage = "Problem in giving the response."
        }
        return
    }
    let message = response.bestTranscription.formattedString
    transcription = makeTranslationRequest(text: message)
}

```

Figure 16: The final part of the `startSpeechRecognition` function. Source: Author's own elaboration.

variable, which is an instance of `SFSpeechRecognitionTask` created in the `startSpeechRecognition` function, it is possible to give the user the possibility to stop the speech recognition when desired.

In the following code snippet, the `cancelSpeechRecognition` function is shown, its main goal is cancelling or ending all the processes started in the `startSpeechRecognition` function. To be precise, the speech recognition task, the audio engine and the instance of the `SFSpeechAudioBufferRecognitionRequest` class are halted. That way, when the user decides to start the speech recognition again, the `startSpeechRecognition` function will be executed again and these processes will have a fresh start.

```

func cancelSpeechRecognition() {
    task?.finish()
    task?.cancel()
    task = nil

    request.endAudio()
    audioEngine.stop()

    if audioEngine.inputNode.numberOfInputs > 0 {
        audioEngine.inputNode.removeTap(onBus: 0)
    }
}

```

Figure 17: The implementation of the `cancelSpeechRecognition` function. Source: Author's own elaboration.

## 4.2 Augmented reality module

The augmented reality layer of this application allows the user to see the translations in real time and the other person talking simultaneously. The approach that was followed to achieve this was, firstly, getting access to the views of the frontal and back cameras, secondly, overlay the real time translations over the views of the cameras.

The AVFoundation library was used to achieve the first task. This library, previously explained in "Methodology and Software tools", is capable of working with time-based audiovisual media formats like QuickTime movies, MPEG-4 files, HLS streams and more, however, in the context of this application it will be used to get access to the camera view.

Similarly to the speech recognition module, the augmented reality module also needs to add some privacy keys to the Info.plist file of the project. In this case, the added keys are called "Privacy - Camera Usage Description" and "Privacy - Photo Library Usage Description". These keys also need a description to explain the user why they are needed. The three main pieces of code that has been developed to achieve the goal of this layer are the following, they are ordered from the highest to the lowest level of abstraction:

1. CameraPreview struct.
2. CameraModel class.
3. CameraService class.

The *CameraPreview* struct is in charge of providing the user interface view that will show the outside world through the cameras of the mobile phone. Inside of this struct, there is a class defined called *VideoPreviewView*, this class inherits from the *UIView* class, so an object of this class could be used as the user interface of the application, at the same time, it also uses the *AVCaptureVideoPreviewLayer* class from AVFoundation, this class provides a preview of the content the camera captures. After that, there are two functions that needed to be implemented to make *CameraPreview* conform to the *UIViewRepresentable* protocol, these are *makeUIView* and *updateUIView*, although the second one will not be needed in this project, the first one, *makeUIView*, is responsible for returning the user interface view of what the cameras are capturing. This is achieved by returning an instance of the *VideoPreviewView*



class mentioned before. SwiftUI will later catch this view and show it as the main view of the application.

```
struct CameraPreview: UIViewRepresentable {
    let session: AVCaptureSession

    class VideoPreviewView: UIView {
        override class var layerClass: AnyClass {
            AVCaptureVideoPreviewLayer.self
        }

        var videoPreviewLayer: AVCaptureVideoPreviewLayer {
            return layer as! AVCaptureVideoPreviewLayer
        }
    }

    func makeUIView(context: Context) -> VideoPreviewView {
        let view = VideoPreviewView()
        view.backgroundColor = .black
        view.videoPreviewLayer.cornerRadius = 0
        view.videoPreviewLayer.session = session
        view.videoPreviewLayer.connection?.videoOrientation = .portrait

        return view
    }

    func updateUIView(_ uiView: VideoPreviewView, context: Context) {
    }
}
```

Figure 18: The implementation of the CameraPreview struct. Source: Author’s own elaboration.

The *CameraModel* class is high-level implementation for *CameraService* that provides some useful utility functions. These functions are *configure*, which will request permission to access the camera, *flipCamera* and *zoom*. These functions are simply invoking functions from the *CameraService* class, which will be explained afterwards. It is important to mention that the *configure* function is executed the first time the user opens the application.

The *CameraService* provides many low level functions useful to camera applications. In the context of this application, it will mainly do two things, checking camera permissions and configuring the cameras that are going to be used. The *checkForPermissions* function defined in *CameraService* is very similar to the *requestPermission* function explained in the “Speech recognition module” section. In the case of configuring the cameras, this is done in the *configureSession* function. As it can be seen from the code snippet below, this is done inside a *do-catch* statement in case the cameras are not accessible. First of all, we create an object of the *AVCaptureDevice* class called *defaultVideoDevice*, this object will be the virtual representation of the camera used. Next, the *if-let* statement tries to create a constant called *backCameraDevice* with the value of the virtual representation of the back camera, if this is not nil, the value of

```

final class CameraModel: ObservableObject {
    private let service = CameraService()
    @Published var showAlertError = false
    var alertError: AlertError!
    var session: AVCaptureSession
    private var subscriptions = Set<AnyCancellable>()

    init() {
        self.session = service.session

        service.$shouldShowAlertView.sink { [weak self] (val) in
            self?.alertError = self?.service.alertError
            self?.showAlertError = val
        }
        .store(in: &self.subscriptions)
    }

    func configure() {
        service.checkForPermissions()
        service.configure()
    }

    func flipCamera() {
        service.changeCamera()
    }

    func zoom(with factor: CGFloat) {
        service.set(zoom: factor)
    }
}

```

Figure 19: The implementation of the CameraModel class. Source: Author's own elaboration.

*backCameraDevice* will be assigned to *defaultVideoDevice*, otherwise, the same is done for the front camera. After that, we check if *defaultVideoDevice* is nil or a virtual representation of any of the cameras, in case it is nil, the setup of the cameras must be considered failed. The flow of execution continues by creating a *AVCaptureDeviceInput* object out of the *videoDevice* recently created, according to Apple [App08b], an object of the *AVCaptureDeviceInput* class "provides media input from a capture device to a capture session". At last, the *videoDeviceInput* is added to the current session, so now the cameras are configured and ready to show what they are capturing.

```
do {
    var defaultVideoDevice: AVCaptureDevice?

    if let backCameraDevice = AVCaptureDevice.default(.builtInWideAngleCamera, for: .video,
        position: .back) {
        defaultVideoDevice = backCameraDevice
    } else if let frontCameraDevice = AVCaptureDevice.default(.builtInWideAngleCamera, for:
        .video, position: .front) {
        defaultVideoDevice = frontCameraDevice
    }

    guard let videoDevice = defaultVideoDevice else {
        print("Default video device is unavailable.")
        setupResult = .configurationFailed
        session.commitConfiguration()
        return
    }

    let videoDeviceInput = try AVCaptureDeviceInput(device: videoDevice)

    if session.canAddInput(videoDeviceInput) {
        session.addInput(videoDeviceInput)
        self.videoDeviceInput = videoDeviceInput
    } else {
        print("Couldn't add video device input to the session.")
        setupResult = .configurationFailed
        session.commitConfiguration()
        return
    }
} catch {
    print("Couldn't create video device input: \(error)")
    setupResult = .configurationFailed
    session.commitConfiguration()
    return
}
```

Figure 20: Configuring the cameras to be used in the application. Source: Author's own elaboration.

Finally, after setting up all the code necessary for the augmented reality layer (*CameraPreview* struct and the *CameraModel* and *CameraService* classes), the last thing left was showing what the camera is capturing as a view of the application. Fortunately, the *CameraPreview* struct implements the *UIViewRepresentable* protocol, so it can directly be shown in SwiftUI as an user interface. To achieve this, it is as simple as calling *CameraPreview(session: model.session)* inside the body of a SwiftUI view. In the code snippet below, *CameraPreview(session: model.session)* has been executed along with some modifiers. Modifiers are extra pieces of code to enrich an user interface, in this case, the *gesture* modifier has been used to

make zoom when the drag gesture is done. The second modifier, *onAppear*, allows to execute a function when the view is first shown, in this case *model.configure()* is executed to ask for camera permissions. Another modifier used is *alert*, which will warn the user when an error happens. The most important modifier is *overlay*, this modifier allows to show things over the current view, that way, the translated transcriptions will be shown to the user at the same time he sees the outside world through the camera.

```

CameraPreview(session: model.session)
    .gesture(
        DragGesture().onChanged({ (val) in
            // Only accept vertical drag
            if abs(val.translation.height) > abs(val.translation.width) {
                // Get the percentage of vertical screen space covered by drag
                let percentage: CGFloat = -(val.translation.height /
                    reader.size.height)
                // Calculate new zoom factor
                let calc = currentZoomFactor + percentage
                // Limit zoom factor to a maximum of 5x and a minimum of 1x
                let zoomFactor: CGFloat = min(max(calc, 1), 5)
                // Store the newly calculated zoom factor
                currentZoomFactor = zoomFactor
                // Sets the zoom factor to the capture device session
                model.zoom(with: zoomFactor)
            }
        })
    )
    .onAppear {
        model.configure()
    }
    .alert(isPresented: $model.showError, content: {
        Alert(title: Text(model.alertError.title), message:
            Text(model.alertError.message), dismissButton:
                .default(Text(model.alertError.primaryButtonTitle), action: {
                    model.alertError.primaryAction?()
                })
    })
    )
    .overlay(
        //Al poner las transcripciones traducidas en el overlay
        // se pondrá por encima de la vista de la cámara.
        Text(transcription)
            .padding()
    )
)

```

Figure 21: The user interface that shows what the camera is capturing and the translated transcriptions simultaneously. Source: Author’s own elaboration.

### 4.3 Translation module

The last module of this application is the real time translation module. As it was explained in the “Methodology and Software tools” section, the Azure AI translator [Mic22] was chosen among a handful of candidates due to having low latency, providing a well documented REST API and being the cheapest and most scalable solution.

At the start of the *TranslationService.swift* file, there are two structs used to decode the response JSON that we will receive from Azure AI Translator. In Swift, the decodification of JSONs needs to be done with the *JSONDecoder* class [App22]. *JSONDecoder* will automatically

be able to decode JSONs that follow the structure of a struct that implements the *Codable* protocol. The structs implemented in this case are shown below.

```
struct TranslationBody: Codable {
    let text: String
    let to: String
}

struct Translation: Codable{
    let translations: [TranslationBody]
}
```

Figure 22: Structs to decode the response JSON and get the translation data. Source: Author's own elaboration.

Below there is a screenshot of the response JSON that Azure AI translator sends back. The *TranslationBody* struct will be used to decode the inner object with the "text" and "to" keys. On the other hand, the *Translation* struct will be used to decode the external object with the "translations" key.

```
1  [
2  {
3      "translations": [
4          {
5              "text": "Que tengas un buen día mi amigo",
6              "to": "es"
7          }
8      ]
9  }
10 ]
```

Figure 23: Response JSON that Azure AI translator sends back. Source: Author's own elaboration.

The *makeTranslationRequest* function is in charge of translating the transcriptions. This method takes english text as input and it outputs the spanish translation for that text. First of all, there is an initialization of a Semaphore [ZP08], which will be used when making the request to control the flow of execution of the asynchronous event. After that, the variable *translation\_body* is created, this variable contains a dictionary with a "text" key and the input of the function as the corresponding value of this key. Next, this *translation\_body* needs to be converted into an object of the *Data* class so that it can later be used as the HTTP body of the request, the transformation is done using the *JSONSerialization.data* function taking the *translation\_body* as input. The flow of execution continues by defining the URL where the requests will be made to, following this up, the actual request is created. The *URLRequest*

constructor takes the recently created URL as a parameter, after that it was important to set the type of REST request, which in this case is a POST request. The only two things left of the initial part of the *makeTranslationRequest* function are setting up all the needed HTTP headers (Content type, API key, API region, client id) and assigning the JSON data to the HTTP body of the request. The JSON data that was added to the request contains the text that will be translated by the service.

```
func makeTranslationRequest(text: String) -> String {
    let sem = DispatchSemaphore.init(value: 0)

    let translation_body = [
        "text": text
    ]

    let jsonData = try? JSONSerialization.data(withJSONObject: translation_body)

    let url = URL(string:
        "https://api-eur.cognitive.microsofttranslator.com/translate?api-version=3.0&from=en&to=es")!

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")
    request.setValue("f090c0af71254f68bca17e00bb1f3be0", forHTTPHeaderField:
        "Ocp-Apim-Subscription-Key")
    request.setValue("westeurope", forHTTPHeaderField: "Ocp-Apim-Subscription-Region")
    request.setValue("944161d6-455c-4dc0-9c30-aba05cdf1c67", forHTTPHeaderField: "X-ClientTraceId")

    request.httpBody = jsonData
}
```

Figure 24: Initial part of the *makeTranslationRequest* function. Source: Author's own elaboration.

The final part of the *makeTranslationRequest* function starts by defining *translationResult*, a String variable that will be the output of this method. Right after this, the task that invokes the translation request is created. This task makes use of the *dataTask(with: request)* function from the *URLSession.shared* object. This object from the *URLSession* class provides a shared singleton [Sel22] session that can be used to fetch something from a URL in a simple way. The shared session achieves this by using the *dataTask(with: request)* function, that takes the recently created *request* associated with the URL as input. Next, the block of code inside the task can be seen, this code will be executed asynchronously by the task. Asynchronous events can be tricky because the function may return the *translationResult* without the request being completed, so *translationResult* may not have the intended value. To solve this concurrency problem, a Semaphore [ZP08] has been used to control the flow of execution. Below the definition of the *task*, this asynchronous event is started using *task.resume()*, right below this function, the semaphore variable waits until the same semaphore variable sends a signal inside the body of the *task*, this assures that the *translationResult* will be returned with the intended

value. Coming back to the block of code inside the *task*, the result of the request will be in the *data* variable. After making sure that there are no errors and the data is not corrupt, the decodification of this *data* variable is done using the *JSONDecoder* mechanism explained at the start of this section. After decoding the response JSON, it can be accessed as a common dictionary, so we access the text of the translation and assign it to the *translationResult* variable that will be the output of this *makeTranslationRequest* function. The value of *translationResult* will later be shown to the user through the augmented reality environment that was previously explained.

```
var translationResult: String = ""

//Make the request
let task = URLSession.shared.dataTask(with: request) { data, response, error in

    defer { sem.signal() }

    guard let data = data, error == nil else {
        print(error?.localizedDescription ?? "No data")
        return
    }

    do {
        let responseJSON = try JSONDecoder().decode([Translation].self, from: data)
        translationResult = "\(responseJSON[0].translations[0].text)"

    }catch {
        print(error)
    }
}

task.resume()

sem.wait()

return translationResult
}
```

Figure 25: Final part of the *makeTranslationRequest* function. Source: Author's own elaboration.

## 4.4 Results

Regarding the results, some measurements needed to be taken first. The primary method to measure the different modules was by creating a function called *timeElapsedInSecondsWhenRunningCode*. This method makes use of the *CFAbsoluteTimeGetCurrent* function [App06] which returns the current absolute time. The *timeElapsedInSecondsWhenRunningCode* function will capture the absolute time at the start, then the function to be measured will be executed and the time elapsed will be the subtraction of the current absolute time and the time

captured previously. This is returned as a Double to have sub-second accuracy as best as the system stores it.

```
func timeElapsedInSecondsWhenRunningCode() -> Double {
    let startTime = CFAbsoluteTimeGetCurrent()
    functionToBeMeasured()
    let timeElapsed = CFAbsoluteTimeGetCurrent() - startTime
    return Double(timeElapsed)
}
```

Figure 26: Implementation of the function used to make measurements. Source: Author's own elaboration.

To appropriately take measurements, the modules needed to be isolated, that way only the execution time of a module would be measured, in order to achieve that, the *functionToBeMeasured* function on the following code snippet will be substituted by the isolated function of a module. In case of the translation module, the time between receiving the transcription text, making the request and returning the translated text will be measured. For the speech recognition module, we will measure the time between the speech recognition receives the audio input and the moment it transcribes the live audio to text. Finally, for the augmented reality module, the time between receiving the translated text and showing it in the camera view will be measured. The results of these measurements are the following:

- Execution time range for Translation module: [300ms, 500ms].
- Execution time range for Speech recognition module: [100ms, 250ms].
- Execution time range for Augmented reality module: [50ms, 75ms].

These measurements were the results of testing a set of strings that varies from single words to whole compound sentences that go up to 250 characters per sentence. As it can be seen from the results, the addition of the measurements would produce a [450ms, 825ms] range, being 825ms the worst case scenario, which beats our main goal of producing a real-time translator app that works in less than 1 second.

The results are good, but the asynchronous execution of the different modules improves the speed of the application even more. The first time a sentence is translated, it will produce the results seeing above, but after that, the asynchronous execution of the modules will reduce the waiting time that the user experiences. For example, meanwhile the first sentence is being



translated, the user has already said the second sentence, so the speech recognition module can transcribe the second sentence to text meanwhile the first sentence is translated by the translation module. Moreover, this does not end here because there can be more than one task of the same module being executed concurrently, for example, there can be more than one translation task being executed at the same time, so the time between showing the translations will get even shorter.

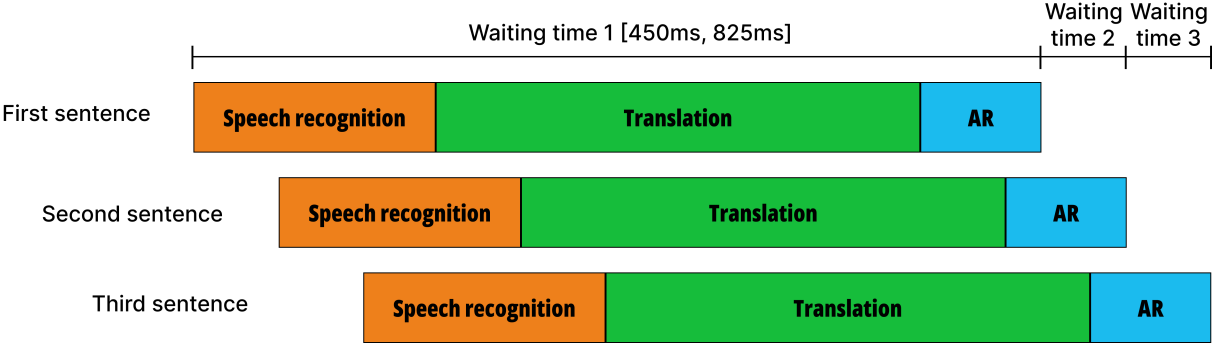


Figure 27: Diagram showing the concurrent execution of the modules. Source: Author’s own elaboration.

In this case, the main contribution is the use of asynchronous tasks to accelerate the speed of the complete workflow of the application. Specifically, the core component of this improvement has been taking advantage of the *Task* struct [App14d] to exploit the concurrency [App14c] capabilities of the Swift programming language [App14a]. This approach can similarly be applied to other programming languages and technologies to speed up the execution time when asynchronous tasks are a possibility. In conclusion, the challenge of being able to create a real-time translator, that is able to integrate all the modules with an execution time of less than one second has been overcome successfully.



# 5

## Conclusions and Futures Studies

### 5.1 Conclusions

The main goal of this project was to overcome the challenge of integrating different libraries to build a real-time translator application, essentially implying that all the operating modules combined should produce results in less than a second. In this section, the results and contributions of this project will be shown along the key aspects that permitted to accomplish the proposed goal. Furthermore, I will personally talk about all the takeaways and knowledge I have gained by successfully carrying out this project.

On a personal level, the challenge of building a real-time translator that shows the translations using augmented reality is something I have been wanting to do for several years. Due to the complexity of the task, I decided to postpone it until the final year of my software engineering degree, so when I finally started the project I was happy and motivated to do my best and learn everything I can in the process. This project has helped me to go deeper into advanced topics like natural language processing, asynchronous programming or real-time app development, and also to consider the requirements of a mobile application in a serious way. The main two takeaways of this project has been, firstly, all the experts I have met in areas like augmented reality, natural language processing and speech recognition, and secondly, all the knowledge I have gained in those areas that I can now apply in my future professional life.

Regarding the results, I analyze the overall key contributions made and the ones specific for each of the involved technologies. On a general level it was quite relevant to analyze the different available libraries and frameworks in which this project can be based, choosing the proper ones, bearing in mind that they should be integrated for a common goal. Despite choosing the appropriate tools, making them work in real-time using concurrency techniques

is a key aspect of the work of this project, and much effort was put to optimize this.

In case of the speech recognition module, the main contribution has been making the processing of the speech on-device instead of sending the audio to external servers, which was the default configuration at the start. This is the main reason why the speech recognition module is fast enough for the real-time requirements established at the start of this work.

For the augmented reality module, the core aspect has been finding out how to get access to the cameras and show the actual view of the cameras following the SwiftUI standards. Furthermore, finding out how to use view modifiers to overlay text to custom camera views has been a great effort.

Regarding the translation module, researching and selecting the translation API that fits the best for the real-time requirements has been key for the successful implementation of this module. Creating asynchronous translation tasks that can be executed concurrently with the other modules has been the main contribution. This has been extremely important to reduce execution time and achieve the real-time requirements of this application.

As an overall conclusion, I am very happy and proud of this project as I have not only learned several useful and novel software libraries and technologies, but also achieve my long term goal of build a real-time translator application.

## 5.2 Future Studies

Regarding future studies, there are a few things that could be worked on. The current project is able to translate from English to Spanish, and thus adding more languages would be a relevant improvement. In fact, the translation module could already do this with very few changes. Nevertheless, problems arise when trying to do speech recognition with languages other than English, because the accuracy of the transcriptions will not be at the same level, as libraries are not so optimized. In other words, in order to add more languages to translate, improving the accuracy of speech recognition for other languages would be necessary.

Another improvement for the foreseeable future could be adding a NLP machine learning model that works as a contextual layer [Dev22] for the speech recognition module. This NLP model would be able to distinguish if the current word transcribed from the speech recognition module makes sense in the context of the whole sentence. For example, when having two words with similar pronunciations, *dish* and *fish*, in the context of a phrase like "I went to the

beach and caught a <word>”, this model would be able to pick *fish* as the word that fits the best in that situation, improving even more the accuracy of the speech recognition module.

Something that could be useful to some users is automatically clearing up the text of the translation every few sentences. The reason why this has not been done is because some people may need more time to read the translated sentences, so a button was provided to clear up the text when needed. Another feature that could be implemented, specially for business use cases, is recording a conversation along with the translations so that the user can access them later. In case this was added, there should be a prior agreement between the different people having the conversation and the application should explicitly say that the current conversation is being recorded.

One of the many reasons why this project was developed using Apple technologies was to smoothly port this application to the upcoming Apple augmented reality headset. As it was mentioned in the ”Methodology and Software tools” section, SwiftUI is a cross-platform technology that works for all Apple products. Furthermore, Apple is the company that cares the most about the interoperability of their product ecosystem, so when the AR headset launches, this application would be in a good position to be the first AR translator available in the platform.



# 6

## Conclusiones y Líneas Futuras

### 6.1 Conclusiones

El objetivo principal de este proyecto era superar el desafío de integrar diferentes bibliotecas para crear una aplicación de traducción en tiempo real, lo que esencialmente implica que todos los módulos combinados deberían producir resultados en menos de un segundo. En esta sección se mostrarán los resultados y aportes de este proyecto junto con los aspectos clave que permitieron lograr el objetivo propuesto. Además, hablaré personalmente sobre todos los aprendizajes y conocimientos que he adquirido al llevar a cabo con éxito este proyecto.

A nivel personal, el reto de construir un traductor en tiempo real que muestre las traducciones usando realidad aumentada es algo que quería hacer desde hace varios años. Debido a la complejidad de esta tarea, decidí posponerlo hasta el último año de la carrera de Ingeniería del Software, así que cuando finalmente comencé el proyecto estaba feliz y motivado para dar lo mejor de mí y aprender todo lo que pudiera en el proceso. Este proyecto me ha ayudado a profundizar en temas avanzados como el procesamiento del lenguaje natural, la programación asíncrona o el desarrollo de aplicaciones en tiempo real, y también a considerar seriamente los requisitos de una aplicación móvil. Las dos cosas principales que me ha aportado este proyecto han sido, en primer lugar, todos los expertos que he conocido en áreas como la realidad aumentada, el procesamiento del lenguaje natural y el reconocimiento de voz, y en segundo lugar, todo el conocimiento que he adquirido en esas áreas que ahora puedo aplicar en mi vida profesional.

En el caso del módulo de reconocimiento de voz, la principal contribución ha sido realizar el procesamiento de la voz en el propio dispositivo en lugar de enviar el audio a servidores

externos, que era la configuración predeterminada al principio. Esta es la razón principal por la que el módulo de reconocimiento de voz es lo suficientemente rápido para los requisitos de tiempo real establecidos al inicio de este trabajo.

Para el módulo de realidad aumentada, el aspecto central ha sido descubrir cómo acceder a las cámaras y mostrar la vista real de las cámaras siguiendo los estándares de SwiftUI. Además, descubrir cómo usar modificadores para superponer texto en vistas personalizadas de cámaras ha sido un gran esfuerzo.

Con respecto al módulo de traducción, investigar y seleccionar la API de traducción que mejor se adapte a los requisitos en tiempo real ha sido clave para la implementación exitosa de este módulo. La principal contribución ha sido crear tareas de traducción asincrónicas que puedan ejecutarse simultáneamente con los otros módulos. Esto ha sido extremadamente importante para reducir el tiempo de ejecución y lograr los requisitos de tiempo real de esta aplicación.

Como conclusión general, estoy muy contento y orgulloso de este proyecto, ya que no solo he aprendido sobre varias tecnologías y librerías de software útiles y novedosas, sino que también he logrado mi objetivo a largo plazo de crear una aplicación de traducción en tiempo real.

## **6.2 Líneas Futuras**

En cuanto a las líneas futuras, hay varias cosas en las que se podría trabajar. El proyecto actualmente puede traducir del inglés al español, y por lo tanto agregar más idiomas sería una mejora relevante. De hecho, el módulo de traducción ya podría hacer esto con muy pocos cambios. Sin embargo, los problemas surgen al intentar hacer reconocimiento de voz con idiomas distintos al inglés, porque la precisión de las transcripciones no estarán al mismo nivel, ya que las bibliotecas no están tan optimizadas. En otras palabras, para agregar más idiomas para traducir, sería necesario mejorar la precisión del reconocimiento de voz para otros idiomas.

Otra mejora que podría hacerse en el futuro sería añadir un modelo de machine learning de procesamiento del lenguaje natural que funcione como una capa contextual [Dev22] del módulo de reconocimiento de voz. Este modelo de procesamiento del lenguaje natural sería capaz de distinguir si la palabra actual transcrita por el módulo de reconocimiento de voz tiene



sentido en el contexto de la oración completa. Por ejemplo, al tener dos palabras con pronunciaci3nes similares, *mercado* y *pescado*, en el contexto de una oraci3n como "Fui a la playa y captur3 un <palabra>", este modelo ser3a capaz de elegir *pescado* como la palabra que encaja mejor en esa situaci3n, mejorando incluso m3s la precisi3n del m3dulo de reconocimiento de voz.

Algo que podr3a serle 3til a algunos usuarios es limpiar autom3ticamente el texto de la traducci3n cada pocas oraciones. La raz3n por la que esto no se ha hecho es porque algunas personas pueden necesitar m3s tiempo para leer las transcripciones traducidas, por lo que se proporcion3 un bot3n para limpiar el texto cuando sea necesario. Otra caracter3stica que podr3a implementarse, especialmente para empresas, es grabar la conversaci3n junto a las traducciones para que el usuario pueda acceder a ellas m3s tarde. En caso de que se agregue esto, debe haber un acuerdo previo entre las diferentes personas que tienen la conversaci3n y la aplicaci3n debe decir expl3citamente que la conversaci3n actual est3 siendo grabada.

Una de las muchas razones por las que este proyecto se desarroll3 utilizando tecnolog3as de Apple fue para trasladar facilmente esta aplicaci3n a las pr3ximas gafas de realidad aumentada de Apple. Como se mencion3 en la secci3n "Metodolog3a y herramientas de software", SwiftUI es una tecnolog3a multiplataforma que funciona para todos los productos de Apple. Adem3s, Apple es la empresa que m3s se preocupa por la interoperabilidad de su ecosistema de productos, por lo que cuando se lancen los gafas de realidad aumentada, esta aplicaci3n estar3a en una buena posici3n para ser el primer traductor de realidad aumentada disponible en la plataforma.



# Bibliography

- [Str91] Bjarne Stroustrup. *What is “Object-Oriented Programming”?* 1991. URL: <https://www.stroustrup.com/whatis.pdf>.
- [TP01] Davide Turcato and Fred Popowich. *What is Example-Based Machine Translation?* 2001. URL: <https://aclanthology.org/2001.mtsummit-ebmt.7.pdf>.
- [Abr+02] Pekka Abrahamsson et al. *Agile Software Development Methods: Review and Analysis*. 2002. URL: <https://arxiv.org/ftp/arxiv/papers/1709/1709.08439.pdf>.
- [LA04] Chris Lattner and Vikram Adve. *LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation*. 2004. URL: <https://llvm.org/pubs/2004-01-30-CGO-LLVM.pdf>.
- [App06] Apple. *Official documentation of CFAbsoluteTimeGetCurrent function*. 2006. URL: <https://developer.apple.com/documentation/corefoundation/1543542-cfabsolutetimegetcurrent>.
- [App08a] Apple. *Official documentation for the AVFoundation framework*. 2008. URL: <https://developer.apple.com/av-foundation/>.
- [App08b] Apple. *Official documentation of AVCaptureDeviceInput class*. 2008. URL: <https://developer.apple.com/documentation/avfoundation/avcapturedeviceinput>.
- [ZP08] Julie Zelenski and Nick Parlante. *Thread and Semaphore Examples*. 2008. URL: <https://see.stanford.edu/materials/icsppcs107/23-Concurrency-Examples.pdf>.
- [Rei+10] Gerhard Reitmayr et al. “Simultaneous Localization and Mapping for Augmented Reality”. In: *International Symposium on Ubiquitous Virtual Reality 0* (July 2010), pp. 5–8. DOI: [10.1109/ISUVR.2010.12](https://doi.org/10.1109/ISUVR.2010.12).
- [App14a] Apple. *About Swift*. 2014. URL: <https://docs.swift.org/swift-book/>.
- [App14b] Apple. *Automatic Reference Counting*. 2014. URL: <https://docs.swift.org/swift-book/LanguageGuide/AutomaticReferenceCounting.html>.

- [App14c] Apple. *Concurrency in Swift*. 2014. URL: <https://docs.swift.org/swift-book/LanguageGuide/Concurrency.html>.
- [App14d] Apple. *Official documentation of Task struct*. 2014. URL: <https://developer.apple.com/documentation/swift/task>.
- [App14e] Apple. *The Swift programming language Github's repository*. 2014. URL: <https://github.com/apple/swift>.
- [Dig15] Danny Dig. *Refactoring for Asynchronous Execution on Mobile*. 2015. URL: <http://dig.cs.illinois.edu/papers/refactoringAsync.pdf>.
- [App16a] Apple. *Official documentation for the Speech framework*. 2016. URL: <https://developer.apple.com/documentation/speech>.
- [App16b] Apple. *Official documentation of requestAuthorization method*. 2016. URL: <https://developer.apple.com/documentation/speech/sfspeechrecognizer/1649892-requestauthorization>.
- [App16c] Apple. *Official documentation of SFSpeechRecognizer class*. 2016. URL: <https://developer.apple.com/documentation/speech/sfspeechrecognizer>.
- [App16d] Apple. *Official documentation of SFSpeechRecognizerAuthorizationStatus enumeration*. 2016. URL: <https://developer.apple.com/documentation/speech/sfspeechrecognizerauthorizationstatus>.
- [Occ16] Occipital. *Occipital Brings AR to Life In Any Room*. 2016. URL: [https://www.youtube.com/watch?v=vpc0\\_K715wg](https://www.youtube.com/watch?v=vpc0_K715wg).
- [Wu+16] Yonghui Wu et al. *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*. 2016. URL: <https://arxiv.org/pdf/1609.08144.pdf>.
- [Nov17] Jordan Novet. *Google has slashed its speech recognition word error rate by more than 30% since 2012*. 2017. URL: <https://venturebeat.com/2017/01/11/google-has-slashed-its-speech-recognition-word-error-by-more-than-30-since-2012/>.
- [App19a] Apple. *Official documentation for the @Published property wrapper*. 2019. URL: <https://developer.apple.com/documentation/combine/published>.

- [App19b] Apple. *Official documentation for the ObservableObject protocol*. 2019. URL: <https://developer.apple.com/documentation/combine/observableobject>.
- [App19c] Apple. *Official documentation of onAppear instance method*. 2019. URL: [https://developer.apple.com/documentation/SwiftUI/View/onAppear\(perform:\)](https://developer.apple.com/documentation/SwiftUI/View/onAppear(perform:)).
- [TZT19] Yi Tay, Aston Zhang, and Luu Anh Tuan. *Lightweight and Efficient Neural Natural Language Processing with Quaternion Networks*. 2019. URL: <https://arxiv.org/pdf/1906.04393.pdf>.
- [Bha20] Manick Bhan. *What the commoditization of search engine technology with GPT-3 means for Google and SEO*. 2020. URL: <https://www.searchenginewatch.com/2020/08/21/what-the-commoditization-of-search-engine-technology-with-gpt-3-means-for-google-and-seo/>.
- [Bro+20] Tom B. Brown et al. *Language Models are Few-Shot Learners*. [GPT-3]. 2020. URL: <https://arxiv.org/pdf/2005.14165v4.pdf>.
- [IBM20] IBM. *What is a REST API?* 2020. URL: <https://www.youtube.com/watch?v=lsMQRaeKNDk>.
- [Sta20] Statista. *Number of voice assistant users in the United States from 2017 to 2022*. 2020. URL: <https://www.statista.com/statistics/1029573/us-voice-assistant-users/>.
- [GP21] Santosh Gondi and Vineel Pratap. "Performance Evaluation of Offline Speech Recognition on Edge Devices". In: *Electronics* 10 (Nov. 2021), p. 2697. DOI: [10.3390/electronics10212697](https://doi.org/10.3390/electronics10212697).
- [Nan21] NanoReview. *Snapdragon 888 vs A15 Bionic*. 2021. URL: <https://nanoreview.net/en/soc-compare/qualcomm-snapdragon-875-vs-apple-a15-bionic>.
- [Vod21] Vodafone. *Descubre Nreal Light con Vodafone 5G Reality AR*. 2021. URL: <https://www.youtube.com/watch?v=g3huPp6EW8g&t=55s>.
- [Wik21] Wikipedia. *Makoto Nagao biography*. 2021. URL: [https://en.wikipedia.org/wiki/Makoto\\_Nagao](https://en.wikipedia.org/wiki/Makoto_Nagao).
- [App22] Apple. *Official documentation of JSONDecoder class*. 2022. URL: <https://developer.apple.com/documentation/foundation/jsondecoder>.

- [Ban22] Bankmycell. *How Many Smartphones Are In The World?* 2022. URL: <https://www.bankmycell.com/blog/how-many-phones-are-in-the-world#part-1>.
- [Dev22] Jacob Devlin. *Contextual Word Representations with BERT and Other Pre-trained Language Models*. 2022. URL: [https://web.stanford.edu/class/cs224n/slides/Jacob\\_Devlin\\_BERT.pdf](https://web.stanford.edu/class/cs224n/slides/Jacob_Devlin_BERT.pdf).
- [Git22] Github. *Repository with the source code of this TFG*. 2022. URL: <https://github.com/lloretalvaro/lengu>.
- [Mic22] Microsoft. *Azure AI translator reference page*. 2022. URL: <https://azure.microsoft.com/en-us/services/cognitive-services/translator/>.
- [Nre22] Nreal. *Official Nreal Air website*. 2022. URL: <https://www.nreal.ai/air/>.
- [Rap22] RapidAPI. *Google Translate APIs and Alternatives*. 2022. URL: <https://rapidapi.com/collection/google-translate-api-alternatives>.
- [Sel22] Toni Sellarès. *The Singleton Pattern*. 2022. URL: <https://ima.udg.edu/~sellares/EINF-ES1/SingletonToni.pdf>.
- [Sta22] Statista. *Number of smartphone subscriptions worldwide from 2016 to 2027*. 2022. URL: <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>.

# Appendix A

## User Manual

In this appendix, we are going to show how to use the visual interface of the application.

First of all, the user interface of the application has one view formed by three components. These components will be named up component, middle component and bottom component. The main component is the middle one, where the user will be able to see the outside world through the camera view and the translations simultaneously. The translations will be in a text label placed at the center of the camera view, if the length of the translations increases, the text will adjust automatically while always maintaining a certain padding.



Figure 28: User interface of the application. Source: Author's own elaboration.

The up component contains two buttons that activate the main functionality of the application. If the user wants to activate the speech recognition to start translating he should push

the "Press to start" button. Once this is done, the app will automatically catch everything the person is saying, translate the transcribed text and finally show the translations in the middle component. If the user wants to stop the speech recognition from capturing what he is saying, he should push the same button that he pushed to start it, but now the button text should be "Press to finish". After doing that, all speech recognition tasks will be canceled and the application will stop making use of the microphone. The last translation text will stay in the middle component so that the user has time to read it. If the user decides to clear the text, he can push the "Clear text" button, for a better experience this button should only be pushed when the microphone is not activated, that is when the button below has the "Press to start" text.

Finally, the bottom component has a label containing the available translation, in this case English to Spanish. Next to the label, there is a button with the icon of a camera that will change the camera that is being used, this camera can be either the frontal or back camera, the switch of cameras will be seen in the middle component.



# Appendix B

# Installation Guide

In this appendix, we are going to show how to install the application to be able to execute it in a physical iPhone. The requirements are the following: having a computer with macOS Monterey, an iPhone with iOS 15.0 or higher and Xcode version 13.0.0 or higher.

The first step is downloading the provided source code. The name given to the project is "Lengu" so the "Lengu.zip" file will have all the source code of this application. Alternatively, the source code of this project is publicly available on GitHub, so it can also be downloaded from there in the following reference [Git22]. After decompressing the "Lengu.zip" file, there will be another folder called "Lengu" and a file called "Lengu.xcodeproj". By simply clicking this "Lengu.xcodeproj" file, Xcode should open the project and configure everything needed.

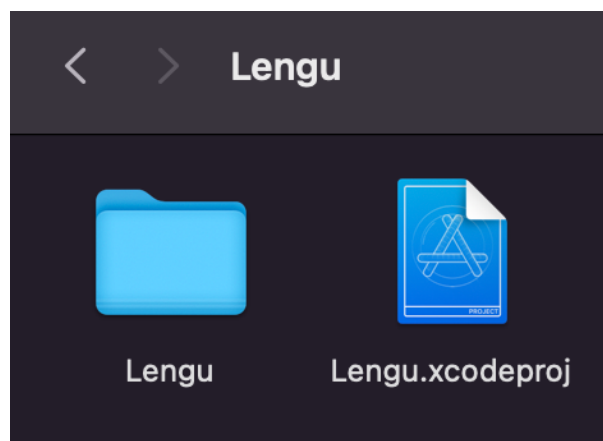


Figure 29: Content inside Lengu.zip. Source: Author's own elaboration.

Once Xcode opens this project, make sure that the requirements written at the start of this appendix are met (macOS, iPhone and Xcode). Next, the physical iPhone should be connected to the computer using a Lightning-to-USB-C cable, Xcode should automatically detect the connection and it will get the iPhone ready for development. After that, the iPhone should be selected as the target of the execution. In order to run the application, it is as simple as clicking the play button found in the top left corner of Xcode. In the following figure, the play

button and the iPhone selected as development target are inside green rectangles to facilitate their localization.

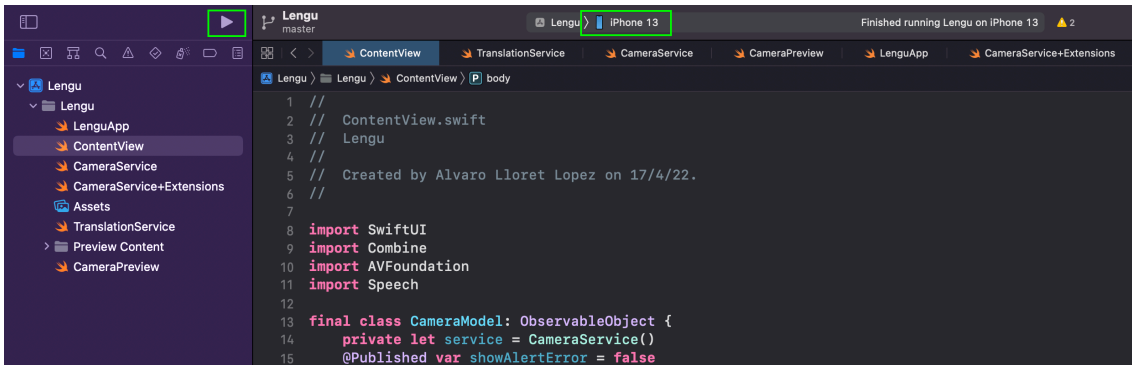


Figure 30: View of the project inside Xcode. Source: Author's own elaboration.



UNIVERSIDAD  
DE MÁLAGA

| [uma.es](http://uma.es)

E.T.S de Ingeniería Informática  
Bulevar Louis Pasteur, 35  
Campus de Teatinos  
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA