Degree thesis

Design and implementation of a mobile application for image recognition and its managing tool

Diseño e implementación de una aplicación móvil para el reconocimiento de imágenes y una herramienta para gestionar su información

Autor: Albert Camps Oller
Supervisora: Beatriz Otero
16/07/2016

Design and implementation of a mobile application, for the identification of rodents species, by using images of their skulls. And the design and implementation of its managing tool, which will be a web application where the users will be able to visualise, add and modify the information.
COPYRIGHT LICENSE

This document is distributed under a Creative Commons (CC) license:

You are free:

  To Share – to copy, distribute and transmit the work

Under the following conditions:

- Attribution – You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Noncommercial – You may not use this work for commercial purposes.
- No derivative works – You may not alter, transform, or build upon this work.

With the understanding that:

- Waiver – Any of the above conditions can be waived if you get permission from the copyright holder.
- Public Domain – Where the work or any of its elements is in the public domain under applicable law, that status is in no way affected by the license.
- Other Rights – In no way are any of the following rights affected by the license:
  - Your fair dealing or fair use rights, or other applicable copyright exceptions and limitations;
  - The author’s moral rights;
  - Rights other persons may have either in the work itself or in how the work is used, such as publicity or privacy rights.

Notice – For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.

http://creativecommons.org/licenses/by-nc-nd/3.0/deed.en
Index

Acknowledgements

Abstract

Resum

Resumen

Objectives

Content

1. Design
   1.1. Requirements and specifications
   1.2 Architecture design
   1.3 Full architecture and functionalities
   1.4 Visual design and some extra functionalities

2. Implementation
   2.1. iOS overview and MVC (mobile application)
   2.2. Angular 2, Firebase and AWS (web application)
   2.3 Image recognition and analysis
   2.4 Databases
   2.5 Let’s put it all together

3. Experimental results
   3.1 Usability of the application
   3.2 Efficiency of the application

4. Costs

5. Conclusions

References
Annex

1. Work Packages, Tasks, Milestones and PERT Diagram
2. Case of use: Mobile Application
3. Mobile Application workflow design
4. Web Application workflow design
5. How to compile the mobile application code
6. How to compile the web application code
7. Implementation of the OpenCV libraries
8. SURF vs ORB
Figure 1. Gantt’s Diagram
Figure 2. How it works
Figure 3. Full project schema
Figure 4. Color palette
Figure 5. Mobile application main screen
Figure 6. iOS Stack
Figure 7. Model View Controller (MVC)
Figure 8. Example of the species’s information that we keep on the Firebase
Figure 9. Tabla comparativa de las alternativas para la implementación de la base de datos
Figure 10. Image recognition workflow
Figure 11. Web application sidebar
Figure 12. Web application actions
Figure 13. SURF CPU Usage
Figure 14. ORB CPU Usage
Figure 15. SURF Memory Usage
Figure 16. ORB Memory Usage
Figure 17. SURF Battery Impact
Figure 18. ORB Battery impact
Figure 19. Battery impact on an Iphone 5
Figure 20. SURF and ORB time results
Figure 21. Project costs table
Figure 22. Detailed project costs table with the number of hours
Figure 23. PERT’s Diagram
Figure 24. Mobile case of use
Figure 25. Mobile application initial view page
Figure 26. Mobile application camera pages
Figure 27. Mobile application species list pages
Figure 28. Mobile application favorites’ list pages
Figure 29. Mobile application specie’s detailed information pages
Figure 30. Mobile application others pages
Figure 31. Web application home page
Figure 32. Web application login page
Figure 33. Web application register page
Figure 34. Web application dashboard page
Figure 35. Web application incompleted species page
Figure 36. Web application profile page
Figure 37. Web application edit specie page
Figure 38. SURF keypoints
Figure 39. ORB keypoints
ACKNOWLEDGEMENTS

M’agrardia agaïr a tota la gent que m’ha apollat durant la carrera: amics, professors, família i també a Beatriz Otero, ja que sense ella aquest projecte no hagués estat possible.

Y en especial m’agraïria agaïr al meu avi, que ell és i sempre sera la meva màxima inspiració.

Albert Camps Oller, 3 de Juliol de 2016.
ABSTRACT

In the past few years, the power and capacity on the mobile phones have grown exponentially. Is for this reason that thousands of new functionalities have been given to the mobile phone, some of which were unthinkable some years ago. For this reason, many developers and scientists have taken advantage of the potential of the mobile phone in order to make people’s life easier or just for sharing their knowledge to the world, so other people can learn from them. This is, as in the case of image recognition.

Some years ago, in order to do image recognition some really powerful computers were needed in order to bear the costful computational tasks that image recognition required, and only some people were experts on the materia. Nowadays, thanks to the technological advance and thanks to share the knowledge between people, the image recognition theory now is far more extended and it is easier to learn about it. Apart from that, the mobile phones have been developed so much, that now they are capable of bearing the costful computational tasks that image recognition require.

Taking advantage of these two highlights: technological advance and the needs of the people, this project is born: to create a mobile application that uses image recognition and make the application usable. The main objective of this application will be to design and develop a mobile application (for iOS) which will identify different rodent species, in special the *soricomorfos* (commonly known as shrews), using as image their morphological structure (particularly its skull). Apart from that, the project will include a database with 23 species and the creation of a web application, which will be created with the latest technology available currently on the market. The main functionality of the web application will be to give the clients a tool, from where the clients will be able to modify the database’s information more easily.

Once implemented the mobile application, we will evaluate different matching methods, and by doing it we will be able to decide which is the method that suits the best for our project, and that it optimize the time response of the mobile application, so we make sure that the application is the most usable the possible.

For last, we will be testing the application in order to check if the developed application is valid with its initial proposal. And as we will see, the mobile application’s response times for the given examples will be from the order of 1-2 seconds.
En el últims anys, ha augmentat exponencialment la capacit i potència en els telèfons mòbils. Es per a això, que se’l hi ha pogut donar milers de noves utilitats al telèfon mòbil, algunes de les quals eren impensables fa alguns anys. D’aquesta manera, molts investigadors i desarrolladors n’han tret profit d’això, amb el fi de facilitar la vida de les persones o simplement ensenyar els seus coneixements al món, per a ajudar en l’aprenentatge d’altres persones.. Aquest és el cas del processat d’imatge.

Per tal de fer processat d’imatge, uns anys enrere eren necessaris ordinadors molt potents capaços de suportar i dur a terme les operacions computacionals necessaries, i només unes poques persones eren experts sobre la materia. Avui en dia, gràcies al gran avanç tecnològic i del coneixement, el processat d’imatge està més al alcànt de la gent. A més a més, els telèfons mòbils també s’han desenvolupat tant que ja són capaços de soportar aquestes operacions computacionals necessàries per al processat d’imatge.

Aprofitant aquests dos avanços tecnològics i les necessitats de la gent, néix aquest projecte: crear una aplicació mòbil que utilitzi processat d’imatge i sigui usable. L’objectiu d’aquest projecte serà dissenyar e implementar una aplicació mòbil (per a iOS) per a identificar diferents espècies de rossegadors, en especial als soricomorfos (comúament conegudes com a musaranyes), utilitzant com a imatge la seva estructura morfològica (particularment els seus cranes). A més a més, el treball incloura una base de dades compost per 23 espècies i la creació d’una aplicació web, la qual serà creada amb la última tecnologia disponible actualment en el mercat. La funcionalitat d’aquesta pàgina web serà la de donar al client una eina des de la qual poder gestionar, més facilment, la informació de la base de dades.

Un cop implementada l’aplicació mòbil, el treball evaluarà diferents mètodes per a realitzar correctament el matching de les imatges i d’aquesta forma, poder escollir el mètode més apropiat, que a més a més mínimitzi els tems de resposta de l’aplicació mòbil, per tal de fer l’aplicació el més usable possible.

Per últim, es realitzara una cantitat suficient de tests per a comprovar la validesa de l’aplicació desenvolupada. Com veurem, els tems de resposta per a l’aplicació mòbil, per als exemples considerats oscilin en el ordre dels 1-2 segons.
En los últimos años, ha aumentado exponencialmente la capacidad y potencia en los teléfonos móviles. Es por esto, que se le ha podido dar miles de nuevas utilidades al teléfono móvil, alguna de las cuales eran impensables hace algunos años atrás. De esta forma, muchos investigadores y desarrolladores se han aprovechado de estas ventajas, con tal de facilitar la vida de las personas o simplemente enseñar sus conocimientos al mundo para ayudar en el aprendizaje de otras personas. Este es el caso del procesado de imagen.

Con tal de hacer procesado de imagen, unos años atrás eran necesarios ordenadores muy potentes capaces de soportar las operaciones computacionales necesarias para tal tarea, y solamente unas pocas personas eran expertas sobre la materia. Hoy en día, gracias al gran avance tecnológico y del conocimiento, el procesado de imagen está más al alcance de la gente. Además, los teléfonos móviles también se han desarrollado tanto que ya son capaces de soportar estas operaciones computacionales necesarias para el procesado de imagen.

Aprovechando estos dos avances tecnológicos y las necesidades de la gente, nace este proyecto: crear una aplicación móvil que use procesado de imagen y que sea usable. El objetivo de este proyecto será diseñar e implementar una aplicación móvil (para iOS) para identificar diferentes especies de roedores, en especial los soricomorfas (comúnmente conocidos como musarañas), usando como imagen su estructura morfológica (particularmente sus cráneos). Además, el trabajo incluirá una base de datos compuesta por 23 especies y la creación de una aplicación web, la cual será creada utilizando la última tecnología disponible actualmente en el mercado. La funcionalidad de esta aplicación web será la de dar al cliente una herramienta desde la cual poder gestionar, más fácilmente, la información de la base de datos.

Una vez implementada la aplicación móvil, el trabajo evaluará diferentes métodos para realizar correctamente el matching de las imagénes y de esta forma, elegirá el método más apropiado, que además minimice los tiempos de respuesta de la aplicación móvil, con tal de hacer la aplicación móvil lo más usable posible.

Por último, se realizarán una cantidad suficiente de tests para comprobar la validez de la aplicación desarrollada. Como veremos, los tiempos de respuesta de la aplicación móvil, para los ejemplos considerados oscilan en el orden de los 1-2 segundos.
OBJECTIVES

The main objective of this work is the design and implementation of a mobile application and a web application for the species’ identification, by using morphologic structures (i.e. skulls), which will work without internet connection. Furthermore, this work proposes an architecture to implement the mobile and web applications. Although for doing it, we will be using some existing external libraries, such as OpenCV. Our objective is not to improve this library but to learn to use it, and take advantage of all that we think it is the best for our project.

We will have two main different architectures (one for the mobile application and another for the web application) which will be slightly connected with a server (whenever you want to upload a new specie to the DB). In both parts we will see that they have a front-end, that is what the final user see. And a back-end, that is where all the magic will happen.

In order to accomplish this main objective we need to:

1. Design the architecture of the mobile application.
2. Design and implement an image identification system, which will be capable to distinguish the different species, by using photos. Taking advantage of all the capabilities of OpenCV.
3. Compare between all the methods and algorithms identified in point 2, and choose one or two that works the best for our project.
4. Compare between the differents local database that exists, and choose the one that works the best for our project.
5. Make a quick design of how the mobile application should look like.
6. Making a MVP of the mobile application consider the decisions taken in point 2 and 3, never forgetting to make the mobile application usable and with good design.
7. Test the two different technologies taken from point 2, compare the latency and results of each of them, and choose the best technology.
8. Add information to the local database.
9. Add information to the remote database.
10. Design an architecture to unify the web application and the mobile application.
11. Design the architecture of the web application.
12. Make a quick design of how the web application should look like.
13. Make a MVP of the web application.
14. Connect the web application with the mobile application, by adding functionalities to download the new information.

Now, in figure 1 we present a Gantt’s diagram, with the project’s planification in order to succeed our objectives. The project’s work packages (WP) and the PERT diagram can be found on the annex 1.
Figure 1. Gantt’s diagram
This document begins describing the design that we choose for our project. From identifying the requirements needed for the application (and identifying which is the technology in the market that best suited for our project) to how the application should look like. Once the general design is explained, we will focus on more detailed things, such as the structure of the application and how we thought it would be the best way to implement the DBs (the remote and the local one).

After explaining the chosen design, the document will show how the image’s recognition is done, using the library OpenCV. In this section, we will explain some of the different methods that OpenCV has, but we will focus more in two concrete methods SURF and ORB, we will explain some differences about this two and we will choose which one suits the best for our project.

Finally, we will explain some results from the application and we will try to decide whether the application will be usable for the users or not. After that we will show some conclusions and some future projects that should be done in order to improve the project.
1. Design

Before beginning with the design of the different parts of the project, we must decide which are the main objectives that we want to find and which requirements we need to accomplish. These points will have a strong impact in our project’s design.

1.1. Requirements and specifications

These are the requirements:

- This project proposes the development of a mobile application that identifies the specie of a rodent by taking a picture of its skull (this functionality has to be 100% off-line).
- The matching process must give over a 80% of success, if the specie is on the database (DB).
- Time used for the recognition of the specie must be under the 2 seconds, since the initialization of the process.
- The ability to add new species from the mobile phone.
- Make the application usable and with a good design.
- Make clean code, so other people can continue with the project.
- The user will be able to add the image for doing the comparison in two ways:
  - Using the mobile’s camera
  - Using an image in the mobile’s gallery
- The realisation of a simple web application that facilitates the users to help grow the DB, by allowing them the ability to add or modify information from the remote DB. Furthermore, the web application can offer the possibility to create new local DB.
- The remote DB, must be real-time-based DB.
- The connection between the client server will be secure, using the HTTPS protocol.
- In order to modify the information of the specie, credentials will be needed.
- The DB apart from having recognition’s information must have other information about the species, which can be modified in the web application.
- The web application must be responsive in all the devices.
- The DB must be scalable.

All these requirements, are the base which we will take as base in the next point: the architecture design.
1.2. Architecture design

1.2.1 Mobile application (Client)

As we explained in the basis of the project. The system does not require of a server in order to make the recognition operations. All these operations will be all done by the client, using a local DB.

On the other hand, as an extra the client will have the possibility to communicate with a server in order to upload new species to the remote DB, in this case it will act as a **client-server model**¹.

In order to continue with the development, first we need to talk about how the recognition algorithms work, because that will affect in the election of the local DB.

1.2.2 Web Application (Model Client-Server)

The web application will work using a **client-server model**¹. The client will communicate with a server in order to obtain all the information about the rodents, and then the client will be able to send to the server new information.

1.2.3 Image recognition

The image recognition used in this project is based on the libraries of **OpenCV**, which can be found in many different programming languages such as C, C++, Python between others. In this project we do not create/improve any new algorithm, but we take advantage of the methods already created, and try to implement the method that suits for our project. In order to choose which of the methods is the best for us, we must explain how OpenCV works.

In **OpenCV**, if you want to compare two images, and see if they are similar or not. To do this, we compare the descriptors of every image and we check if they match.

In order to get the visual descriptors of every image, OpenCV does the following steps, this process is called the learning of the image:

1. Create the method you want to use with the **parameters**, that suits you the most.
2. Detect the **keypoints** of the image’s Matrix you pass. *(Feature Detector)*

¹ The **client-server model** is simple: or every petition of the user, the system of the user opens a connection with the server, in which uploads the information. The server works with that information and it gives back a result.
3. Computing the keypoints with the image’s Matrix you will obtain the **visual descriptors**. *(Feature Extractor)*

It is **very important** to point, that the parameters must be chosen wisely, as all the process will depend on them.

Once we have the descriptors of the two images we want to compare, we are ready to match them.

The next important decision we will have to take, is the election of the **matcher (Feature Matcher)**. As every matcher works differently, and we will need a different matcher depending on the method that we are using. Then the matcher will compare the points of the two images and for every point will give as a result a number (which the lowest, the best), which will mean the proximate that those two points are (one point for every image).

Once we matched all the images with the image we want to compare, we will obtain some results. We will have to pass this results with some **filter**, in order to get only the ones with best results. The image that got the highest number of results (**best matching**) will be our final result. It must be said that this result can be or not the correct one: we will get the result that is more proximate to the main image, but that result could be wrong. Figure 2 shows this process.

![Figure 2. How it works](image)

In order to obtain the descriptors of the images, and to compare them, this operation is where the most **energy consumption** and the **identification time** will be taken. For this reason, and to accomplish the requirements, we will focus in these implementation modules of our application. This will affect in the decision of our **local DB**. It is possible that if we were to make this kind of operations on a computer, we would not have to care so much about this energy consumption. But
as in the mobile phone the resources are limited, we have to do the best that is in our hands, in order to make it usable.

Apart from that, as we want to be able to communicate with our server and to incorporate new species in our database, the size of the descriptors will be also be important, as well as the data of the image. Because if not it will affect to the usability (in order that it does not affect to the usability so much we might use some API). So we will have to keep that in mind.

1.3. Full architecture and functionalities

Let’s see a full image of all the architecture altogether (figure 3).

1.3.1 Mobile application main functionality (Client)

- Have a DB with the descriptors of the images to compare.
- Accept input images, which will be analysed.
- Detect the keypoints of the images and then compute its descriptors.
- Do the matching to get the image with the best matching.
- Show the result to the user.

1.3.2 Web application main functionality (Client)

- Visualise all the species of the DB.
- Ability to edit the information.
Send the new information to the server.
- As the DB will be real-time-based we will have to keep listening to see if we have new species.
- The ability to download the images of the remote DB, and compress them in a zip.
- The ability to download the remote DB and convert it on a local DB.

1.4. Visual design and some extra functionalities.

Apart from the main functionalities explained before, in order to make the application more usable and more attractive to the users we added some extra functionalities in the mobile application:

- The possibility to see a list with all the species of the local DB.
- The possibility to add a specie in your favorites list.
- The possibility to see a list with all your favorites species.
- A page with help information, of how to use the application.
- A page with detailed information of the selected specie.

And for the visual design we decided to use the colors seen in the following figure, which are commonly used to represent the nature:

![Color palette](image)

In order to make the design the most desirable and usable possible we decided to make all the main options from the mobile applications appeared on the main screen (figure 5), well explained and with custom logos, that helped the users to understand what each button did just by looking at it.

(*) All the screenshots and workflow (Case of Use) of the application can be found in the annex.

![Main screen](image)
2. IMPLEMENTATION

2.1. iOS overview and Mobile View Controller (mobile application)

If we were to divide it logically (from hardware to the end-user) we could find 4 levels (we have to have in mind that iOS is basically UNIX):

- **COCOA TOUCH**: is where the application will be done, where we will create buttons, animations, use the camera, use localization, MapKit, etc. All high level objects.
- **MEDIA layer**: it covers all the media stuff, OpenAL, Audio mixing, Audio recording, Video Playback, Core Audio, OpenGL ES, PDF, etc.
- **CORE SERVICES layer**: it covers the file-access, SQLite, Networking, G3, GPS, Wi-Fi, threading, etc. It has object-oriented interfaces.
- **CORE OS layer**: is the one closest to the hardware, it covers the sockets, security BSD, OSX Kernel, etc. A lot of mechanisms for doing low-level access. Most of this stuff not object-oriented.

Figure 6. IOS Stack

(*) It is called COCOA TOUCH because when they first developed for MAC, its name was COCOA, so when they created the Iphone version, it was the touchable version. But it is pretty similar between COCOA and COCOA TOUCH. This technology of COCOA is been around for around 30 years already.

Nowadays there are two main languages to write the code, one is Swift and the other one is Objective-C.

For the implementation of the application we will use the **Objective-C** language and the use of XCode, and we will be using some libraries as the one for **SQLite** and the one of **OpenCV**. And we will follow the design strategy of **MVC (Model View Controller)**. This strategy, which is schematised in figure 7, consists of:
Model. This design phase is related to: What your application is (but not how is displayed). This model should be UI independent and only could talk with the controller via notifications of updates. Model can talk with other models.

Controller: How your model is presented to the user (UI Logic). Controllers can always talk directly to both models and views and with other controllers. Controllers can have more than one model.

View: The controller’s minions. What will be displayed on the screen. The view can communicate with the controller, if you create an action or by using delegates. View do not own the data that they display.

Figure 7. Model View Controller (MVC)

2.2 Angular 2, firebase and AWS (web application)

2.2.1. Angular 2 (in beta...)

Angular 2 is an open-source web application framework mainly maintained by Google and by a community of individuals and corporations to address many of the challenges encountered in developing single-page applications.

The main points why I chose Angular 2 (which is still on beta) over other languages were:

- High performance.
- Server-side rendering.
- Take advantage of how it does the data-binding, which is required for the future websites.
- Focused for mobile phones.
- Ease of use.
2.2.2. Firebase (remote DB and server) (in alpha...)

Firebase provides a realtime database and backend as a service. The service provides application developers an API that allows application data to be synchronized across clients and stored on Firebase’s cloud.

In our application we use Firebase as the server-side and as the remote DB. As it is a very powerful and easy to use database which will help save our data somewhere in the cloud.

2.2.3. AWS (Amazon Web Services)

Amazon Web Services (AWS), is a collection of cloud computing services that make up the on-demand computing platform offered by Amazon.com. These services operate from 12 geographical regions across the world.

In our application we combine Firebase and S3 (AWS) in order to keep the images of the species kept on the database. So by combining these two, we optimise our remote DB and we give it the possibility to store hundreds of photos.

2.2.4. The reason

We decided to use this combination of technology, because I had a lot of high aspirations and I wanted to do hundreds of things in this project and of course I wanted to try Angular 2. But as I was one single developer, and the time was limited. I had to take advantage of the existing technology. If I had more time I would have chosen not to use Firebase, but to use NodeJS for the server-side and MongoDB for the remote DB.

Although I ended doing a lot of work learning how to use this kind of technology in alpha and beta.

2.3. Image recognition and analysis

In this chapter we will explain more about the image recognition, by using the library OpenCV, and how explain how it's integrated on the iOS.
2.3.1. OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision, originally developed by Intel’s research center in Nizhny Novgorod (Russia). It has many utilities such as:

- Motion understanding
- Object identification
- Structure from motion (SFM)
- Motion tracking
- Augmented reality

But for this project we will only use the detector and extractor of descriptors from the images.

Which is a process of many steps as we mentioned in the chapter 1.2. But in order to be more precise in our matching, let’s add some more steps in our process.

1. Convert the image in grayscale.
2. Obtain the image’s matrix in CV_8U\(^2\), which is necessarily for doing operations with iOS mobile phones.
3. Create the method you want to use with the parameters, that suits you the most.
4. Define the Feature Detector, which will detect the key points of the image’s Matrix you pass.
5. Define the Feature Extractor, which will obtain the visual descriptors from the image and its key points.
6. Define the Feature Matcher, which will be the one that matches the different descriptors.
7. Pass the results for a filter, in order to know which are good results. And see which has the most good results. So this way we will know that it is the best match.

For each of these steps, you can choose whether a method or another. These are the different type of elections that can be done:

Methods:

- FAST – Features from Accelerated Segment Test algorithm
- STAR – STAR algorithm
- SIFT – Scale Invariant Feature Transform algorithm
- SURF – Speeded Up Robust Features algorithm
- ORB – Oriented FAST and Rotated BRIEF algorithm

\(^2\) CV_8U is unsigned 8bit/pixel - ie a pixel can have values 0-255, this is the normal range for most image and video formats.
Matching methods:

- **BFMatcher** – *Brute force matching*
- **FLANN** – *Fast Approximate Nearest Neighbor* matching
- **BF** – *brute-force matching that uses the Hamming distance as a measure*
- Others...

From the list, not all of them are suited for doing image recognition from a mobile phone and for doing image recognition from small skulls. So the ones that would suit us the most would be the methods **SURF** and **ORB** and the matchers **BFMatcher** and **FLANN**. In the next sections we will see in more detail the pros and cons of each one and why we should decide to choose one or the other for our project.

### 2.3.2. SURF & FLANN vs ORB & BFMatcher

In this section of the document we will see the main differences between **SURF and FLANN vs ORB and BF**, as we can notice every method is connected with a matcher, that is because when you use **SURF** you should always use **FLANN**, and when you use **ORB** you should always use **BF**.

You can use different matchers, but these are the recommendations provided from the OpenCV team. So we will use them as they say and make some empiric measures in the results section.

**NOTE:** You can find more information about **SURF vs ORB** in Annex 8.

#### 2.3.2.1. BFMatcher vs FLANN

**BFMatcher** also known as *Brute Force Matcher*, is going to try all the possibilities until he finds the best match.

**FLANN**, meaning "Fast Library for Approximate Nearest Neighbors", will be much faster but will find an approximate nearest neighbors. It will find a good matching, but not necessarily the best possible one.

In other words: **FLANN** is much faster than BFMatcher but it only finds an approximate nearest neighbor, which is a good matching but not necessarily the best (you can play with the parameters of FLANN in order to increase its speed or its precision).
2.3.2.2. SURF

**SURF (Speeded Up Robust Features)** is based on the same principles and steps as SIFT; however, details in each step are different. The algorithm has three main parts: interest point detection; local neighborhood description; and matching.

1. The SURF approach uses square-shaped filters and makes use of the functions of the difference of Gaussians (DoG) where the DoG is calculated on rescaled images progressively to detect the position of the key points. And by removing the less important key points.
2. Get the orientation of the key points by using the gradient.
3. Generate the descriptors using the key points and the orientation of every key point.
4. Matching: By comparing the descriptors obtained from different images, matching pairs can be found.

The goal of a descriptor is to provide a unique and robust description of an image feature, e.g., by describing the intensity distribution of the pixels within the neighbourhood of the point of interest. Most descriptors are thus computed in a local manner, hence a description is obtained for every point of interest identified previously.

2.3.2.3. ORB

**ORB (Oriented FAST and Rotated BRIEF)** is basically a fusion of FAST keypoint detector and BRIEF descriptor with many modifications to enhance the performance. First it use FAST to find keypoints, then apply Harris corner measure to find top N points among them. **ORB is a good choice in low-power devices for panorama stitching etc.**

Also according to the documentation it says that ORB is much faster than SURF and SIFT and ORB descriptor works better than SURF. That is why we decided that it might be a good option for our project.

2.3.2.4. Comparison

So after reading the documentation these were the comparison that we could make...

So in one hand we had SURF, which was slower creating the descriptors but then it was faster doing the matching.
And then, on the other hand, we had ORB which was faster creating the descriptors but it was slower matching them.

So.. here we had a problem, because it was a big problem to choose between the two of them by only reading the documentation. That is why we decided to do some analysis in both of them (each one with its corresponding matching method).

2.3.3. Implementation

OpenCV let us implement the process in many different programming languages. In our case we are going to use an extension of Objective-C, which by adding it will let us use C++ code. So it is a good solution for our project as the libraries of OpenCV were written in C++.

In order to use the extension of C++, in our project, basically what we have to do is to change the .m endpoint for the .mm endpoint.

2.3.3.1. Installation libraries

In order to begin, we must installate the OpenCV libraries. In order to do that we have two options:

- Using the **CocoaPods**, we can install easily the library, by adding in the pods the OpenCV. (CocoaPods is a packet manager which will help install libraries and update libraries automatically)
- We can also insert the libraries manually, downloading the library from the main website. And inserting the OpenCV into our framework.

We recommend the first option, because is the easiest one. And will save us a lot of time and headaches.

**NOTE**: If you are interested in how the code implementation was done, you will find at Annex 7, some of the explications with its code.

2.4. Databases

As we mentioned on the design chapter, we will be using 2 models of databases, one local and one remote. The remote one we will be using Firebase as we mentioned in 2.2. Because as we said before it was the better solution that could be taken being a single developer. If not we would be using MongoDB and NodeJS most probably.
2.4.1. Remote database

For the remote database we will try to normalize the database as much as we can and we will end up with a relational structure like this one:

--- Users
   --- user_uid

--- Species_information
   --- specie_uid
      ---- information

--- Species_descriptors
   --- specie_uid
      ---- descriptors

--- Species_images
   --- specie_uid
      ---- images in binary

Where everyone of the branches will have many specie_uid and that uid will be shared with other branches of the DB, this way making it a relational database.

The user branch is where will be hosted all the allowed users that can be able to modify the information.
2.4.2. Local database

Let’s now decide which would be the best local database for our project then. First of all, we will need a local database, because in the specifications of the project. The offline mode was a requirement. Apart from that by looking at the project specifications, what worries us the most is being able to have a fast and usable application. That is why we will try to choose a lightweight local database for our project, which can support huge amounts of data (the descriptors) and that is easy to implement in our iOS system.

2.4.3. Local database design

2.4.3.1. Existing technologies

Let’s see the different technologies most commonly used.

● SQLite

SQLite is a relational database management system contained in a C programming library. In contrast to many other database management systems, SQLite is not a client–server database engine. Rather, it is embedded into the end program.
Core Data

Core Data is an object graph and persistence framework provided by Apple in the OS X and iOS operating systems. It allows data organised by the relational entity–attribute model to be serialized into XML, binary, or SQLite stores. The data can be manipulated using higher level objects representing entities and their relationships.

PouchDB

PouchDB is an open-source JavaScript database that it enables applications to store data locally while offline, then synchronize it with CouchDB and compatible servers when the application is back online, keeping the user’s data in sync no matter where they next login.

<table>
<thead>
<tr>
<th>Type</th>
<th>Pro</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQLite</td>
<td>1. Simple, fast and efficient. 2. Reusable in many languages. 3. Binary file (portable) 4. Ease of Use</td>
<td>1. Limited possibilities, compared to the other ones. 2. Versioning.</td>
</tr>
<tr>
<td>Core Data</td>
<td>1. It has millions of cool functionalities.</td>
<td>1. Only for iOS. 2. Hard to implement.</td>
</tr>
<tr>
<td>PouchDB</td>
<td>1. Good to implement if you use CouchDB, as you can synchronise your application. Whenever there is any change on the remote DB. 2. Good if you have a web application.</td>
<td>1. Depends on CouchDB.</td>
</tr>
</tbody>
</table>

Figura 9. Tabla comparativa de las alternativas para la implementación de la base de datos

2.4.4. DB choice

Using the information provided before and our knowledge about what the project needs, we decided to choose the SQLite DB. Although the the PouchDB implementation would have been a good option also, we would have to implement a server with a CouchDB.

- As we do not require of any special feature in our project, all we need is to read the information of the DB the fastest possible.
- With a single table of the DB is enough, for keeping all our information.
2.4.5. Let’s code!

In order to use the SQLite we will have to import the sqlite3 library:

```c
#import <Foundation/Foundation.h>
#import <sqlite3.h>
#import "Species.h"
```

Once we can access to its utilities, we will be able to open the SQLite DB and charge the list of the species. In our application we only open the DB once (at the running of the application). And then we keep the full list of the species in memory, this way we will be able to make the comparisons faster.

All the code from the initialisation of the database and how to get the list of species can be found on the annex into the class of `SpeciesDB`.

2.5. Let’s put it all together

So far we have already seen all the pieces of the puzzle. But now, let’s see how do they match.

2.5.1. Mobile application

On the initialisation of the mobile application, it opens the SQLite file and it reads all the information that is saved there (name, descriptions, family, descriptors, col and row of the descriptors, type of descriptor, ...). And we keep that information in a variable in a singleton. That way we will be able to access to that variable in any place of the code.

Once initialized, the application will begin and you will have full access to all the functionalities of the application: see a full list of the species, go to the tutorial, scan an image, upload an image to the server, etc.
The main two actions are the ones of: **take a photo** to discover the specie of the rodent or to **use an image gallery** in order to discover the specie of the rodent. Both actions, will do the main functionality that we wanted our application to do: **do image recognition.** We can observe how it would be the workflow of the application in figure 10.

Once we chose the image we want to compare, the process described before of image recognition will begin (the same as described in previous chapters):

1. Will convert the image in grayscale.
2. Make sure to convert its Matrix in type `CV_8U`.
3. Will detect its key points.
4. Will compute its descriptors.
5. Will do the matching of our recent image with all the images loaded from the SQLite.
6. Pass all the results into the filter.
7. Obtain the final result.
8. The screen will change to one with more detailed information of the specie.

Once we see the information of the specie. An alert will pop up. Asking us if we want to upload this image in order to improve the database. This option has two main functionalities:

1. It gives the possibility to add new species into the DB which we did not have in yet.
2. By uploading an image of an already existing specie in the DB, we help improve the DB so that as more images we have from the specie. The more exact will be the future predictions on that specie.

The described workflow can be seen in the following figure (Figure 10).
NOTE: We have to take care that the image that we upload have good enough quality, because if not, it will not improve the DB. Viceversa it will make it go worse with worse results.

What happens when we decide to upload the image? The image is uploaded into our remote server (Firebase), separately in three branches: one for the descriptors of the specie, another for the base64 image and another for the extra information of the specie.

2.5.1. Web application

Once the new information is added into the remote database. By using the web application, using some credentials we will be able to see and modify the information. We will see that the application is divided in three tabs: one with completed species, another for new/uncompleted species and the last one a profile (where you will be able to modify things of your account or invite more people to the application).

![Figure 11. Web application sidebar](image)

The new species added from the mobile phone, will appear on the uncomplete section. In order to complete that section you will have to fulfill some required fields of information and then the specie will become completed. The required fields are: adding photos, add a description, name, a map,...

NOTE: We must notice that all the images, if we were to keep them in binary or in base64, it would make the remote database very slow. That is why, we decided to use S3 from AWS (Amazon Web Services) in order to keep the images in a data storage place. So that in the remote server we will not be saving the base64 encoding of the data, but we will be only saving the url to the S3 server.

Once we have completed the specie. We will see that that specie will not appear on the uncomplete section. It will be automatically transferred on the completed tab. Once the specie is completed that means that is ready to be imported to our local database of the mobile application.
That is why we have 2 buttons on the top bar:

![Buttons](image)

**Figure 12. Web application actions**

Both of them are for upgrading the version of the mobile application: the first one is for **getting all the images of the AWS and compress them into a zip**, and the second one is for **converting the remote database (Firebase) into a SQLite file**. If we add both folders into our mobile application project. We will have upgraded the mobile application with all the new species, photos and descriptors and new information that we added.

So in this way the developer is the only one that can upload new versions of the application, and release them. But what we would like to have, in an ideal world, is that the user could upgrade the mobile application by himself from inside the mobile application. Without using the web application.

In order to implement it, the best way, as we commented in the past chapters, is not to use SQLite but to use another local DB for the mobile application, for example Core Data or CouchDB.

**NOTE**: On the remote server we added some minimal security into the server. But as the moment it does not look like we will be attacked soon. As we do not have any vital information. This minimal security is an implementation of a login with some credentials, and that you can only modify the information of the completed species if you are logged in. In this way, we make sure that we do not lose at least the information of the completed ones.

**NOTE**: In Annex 5 and in Annex 6, you will find the instructions that explain detailedly **how to install and compile** the application.
3. EXPERIMENTAL RESULTS

In this chapter, we will do two experiments. In the first experiment our objective will be to discover the different impact that have using the method ORB and the method SURF. In order to do that we will use a tool from Apple called Instruments (inside XCode), which will help us take some measures (CPU, memory usage, energy impact…) from the application, and it will help us understand which is the energy impact differences between ORB and SURF.

Apart from that we will also see the time that needs the process in each method.

Furthermore, the objective of the second experiment will be to discover the percentage of success when we try to do the image recognition, using the method ORB and the method SURF, although we will focus more on ORB, because as we have seen on the previous experiment, is the most probable method that is going to adapt the better in our project.

As there is no way (or at least I have not found one), to calculate the percentage of success exactly, we will learn by doing, in other words, we will give the final result, looking at the results that we have had during the experiment.

3.1 Usability of the application

3.1.1. Scenario

For this experiment we will be using an Iphone 5 (version 9.2.1).

In order to accomplish that these calculation are the most correct possible, we will use 2 different DB (one for ORB and the other for SURF). That is because, as we will see, when we compute the descriptors for each of them, every method is different and maybe in one the size of the descriptors are 500x64 and for the other is of 1200x64. So in order to do the comparison the most approximate possible and that both DB have the same size. We will have one DB for ORB and another for SURF:

- For SURF, we will use a database with 40 species, and every specie has a size of 106x64.
- For ORB, we will use a database with 23 species, and every specie will have a size from 50x64 to 500x64.

As we said before, when we are calculating SURF we will use the FLANN Matcher, and we will use the BFMatcher for ORB.
Albert Camps Oller · Universitat Politècnica de Catalunya

The image token, did not really matter as our objective in this experiment is not to calculate the percentage of success, but the energy impact. But in order to see what would happen in a real case, we used a rodent’s skull image.

The process, on which we will take the results come from are:

- Take photo.
- Read the species list.
- Convert the photo in black and white.
- Create its Matrix.
- Detect the keypoints.
- Compute the descriptors.
- Match the obtained descriptors with all the descriptors of the list.
- Get result.

3.1.2 Experimental results

If you were to say a difference between ORB and SURF, at first glance just by looking to the DB, you would notice rapidly that in ORB the descriptors are integers, and in SURF the descriptors are floats. This will be in fact, one of the main facts, that will decide which will be the best method for our project.

3.1.2.1 Differences CPU Usage

<table>
<thead>
<tr>
<th>SURF (FLANN)</th>
<th>ORB(BFMatcher)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td><strong>CPU</strong></td>
</tr>
<tr>
<td>![](Figure 13. SURF CPU Usage)</td>
<td>![](Figure 14. ORB CPU Usage)</td>
</tr>
</tbody>
</table>

Figure 13. SURF CPU Usage

Figure 14. ORB CPU Usage
In **Instruments**, when you want to see the used CPU, you can see two lines: a green and a blue. The green line represents the secure zone. If you surpass the 200 limit, your application will close. And the blue line is the maximum peak of CPU that we used. The lower the better (less conflicts). It is possible that the CPU usage varies between different devices.

As we already told at the beginning, image recognition is a complex process so in order to do the matching it will use a lot of CPU. So we should try to optimise this process the maximum possible.

In our experiment we could see that in SURF that peak went from **120%-135%**, whenever an application uses more than the 120%, is when you have to begin paying attention. Is not a bad number... But we should try that it does not grow upper than that.

And when we used ORB that peak went from **90%-100%**, which is a good number (knowing that is a complex process).

One of the reasons why ORB wins. Is because as we said before, in order to detect the keypoints and compute the descriptors. It uses **integers**, which will be much faster and easier to compute than the **floats** of SURF.

### 3.1.2.2. Differences Memory Usage

<table>
<thead>
<tr>
<th>Memory Usage</th>
<th>Memory Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SURF (FLANN)</strong></td>
<td><strong>ORB(BFMatcher)</strong></td>
</tr>
<tr>
<td><img src="image1" alt="Figure 15. SURF Memory Usage" /></td>
<td><img src="image2" alt="Figure 16. ORB Memory Usage" /></td>
</tr>
</tbody>
</table>
In **Instruments**, when you want to see the used Memory, you can see three colors in the line: a green, a yellow and a red. The green color represents the secure zone. The yellow color warns you not to surpass much more. And when it reaches to the red zone (danger zone), conflicts can appear on your application. The limits between these barriers depends on the mobile phone.

We can see once more how the integers and floats affects, meanwhile in ORB we got normally as a highest memory usage of **22-25 MB** in SURF we got a memory usage of **40-43 MB**.

### 3.1.2.3. Differences Battery Impact

In order to obtain these results, we used directly a function from the Iphone, which lets you record the battery usage that you use and then you can import that file into **Instruments**. The results are a bit strange as we will see. That is because, every mobile phone have different batteries and the battery usage depends on many things. So it is really hard to calculate a pure number of battery used. That is why in **iOS they use its units**, as we will see.

<table>
<thead>
<tr>
<th>SURF (FLANN)</th>
<th>ORB (BFMatcher)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 17. SURF Battery Impact</td>
<td>Figure 18. ORB Battery Impact</td>
</tr>
</tbody>
</table>
Albert Camps Oller · Universitat Politècnica de Catalunya

The Energy Usage instrument indicates a level from 0 to 20, indicating how much energy your application is using at any given time. These numbers are subjective. If your application’s energy usage level is occasionally high, it doesn’t necessarily mean that your app has a problem. Your app may simply require more energy for some of the tasks it performs. For example, it may use the GPS while performing complex network operations. This is valid energy use. What you should look for are spikes or areas of high energy use that are unexpected or that could be performed at more optimal times.

The energy usage level is a fraction over 20 at a particular time interval as seen on the timeline for the device. For example, 0/20 indicates the device is connected to a power source while 14/20 represents 70% of energy usage.

As we can see from the results in both of them we have a result of 17/20 (more or less, a 90% of energy usage), when the action of the matching is done. But from the table we can notice that one of the two takes longer than the other to finish. Meanwhile one of the two (ORB) return to a stable level after in 15 seconds, the other one (SURF) needs almost 25 seconds to recover a low level of battery impact.

Now let’s see in numbers how much battery would be wasted on my mobile phone, in order to find out that, we will use some Objective-C code:

```objective-c
float startBatteryLevel = [[UIDevice currentDevice] batteryLevel];
[[UIDevice currentDevice] setBatteryMonitoringEnabled:YES];
```

Then when the process is done we check again the level of the battery and we check the difference between those two, and the result is this one (on my mobile phone, in every phone might act differently):

```objective-c
float finishBatteryLevel = [[UIDevice currentDevice] batteryLevel];
```

As we can see, we begin with the 17% of the battery, and then we end up with the 17% also, so it did not have any very strong impact on our battery. But we must remark that this measure depends very much on the model of mobile phone.

After doing some more testing, in my mobile phone after doing the research 3-4 times, it went one point down, to 16%.

Taking as a result that if my battery were to be fully charged, I could use the recognition process almost 300-400 times before the battery drains out.

Although we must also notice, that the draining of the mobile phone is sometimes very notorious, and maybe it is hard to waste a 1% and then suddenly you waste 5%, without knowing why.

Figure 19. Battery impact on an iPhone 5
If we were to want exactly how many times we could do the operation before the battery drained out, what we should do is:

- Create a history file with current timestamp so that we can track when the battery runs out.
- Run an infinite loop that each time repeats the process until power runs out.
- And in every loop we should update the history file with the current timestamp information.
- And then when the power runs out extract this file from the mobile phone. And use it on the Instruments tool.

Using code would be something like this:

```swift
NSString* filePath = [NSTemporaryDirectory() stringByAppendingPathComponent:@"fileIORunHistory.txt"];

// create a new file
NSFileManager *fm = [NSFileManager defaultManager];
if (![fm createFileAtPath:filePath contents:nil attributes:nil] == NO) NSLog(@"Couldn't create a file on disk");

// Run until power runs out
while (!stopRun) {
    /* Do the process */
    /* Update history file with current timestamp so that we can track * when the battery runs out. */
}
```

### 3.1.2.4. Time results

In order to obtain these results, we added some time counters in our code, and this is were the results that we got after some captures:

**SURF**

```
Total DB time: 0.455764 s
Total match time: 0.20318 s
DB match time: 0.739544 s
To operations time (convert image to BW, detect keypoints, compute descriptors): 0.741625 s
Total execution time: 1.49077
```

**ORB**

```
Total DB time: 0.3369 s
Total match time: 0.356224 s
DB match time: 0.093124 s
To operations time (convert image to BW, detect keypoints, compute descriptors): 0.207506 s
Total execution time: 0.90063
```

Figure 20. SURF and ORB time results

To finish our results, let’s take a look at the time. As we can see, as the theory explained. In SURF, using FLANN, the **matching time was the fastest**. But, unfortunately, it **takes longer on the**
charging of the DB and the time of detecting keypoints, compute the descriptors. On the other hand ORB is slower doing the matching time (although not very much slower), and it is way faster at reading the DB and to do the recognition operations.

The time difference between ORB and SURF in this example was that ORB was 0.58 seconds faster.

### 3.1.3. Experimental conclusions

So after seeing all the results from the experiment, I think that we have a clear winner and that the best method for our project is ORB.

Not only because is faster, but because it uses less CPU, less Memory usage, and has less battery impact.

Another good point to select ORB, that we noticed during the project. Was that on the web application side. When we wanted to create the SQLite, by reading the remote DB. Using ORB descriptors (integers) it let us create the SQLite without problems.

On the other hand, when we tried to create the SQLite using SURF descriptors (floats), the web application did not resist, and many times came into conflict.

So once again, we have a clear winner in this experiment: ORB.

The next experiment will talk about the success on the image recognition.

### 3.2. Efficiency of the application

#### 3.2.1. Scenario

For this experiment we will be using an Iphone 5 (version 9.2.1).

In order to accomplish that these calculation are the most correct possible, we will use 3 different DB (two for ORB and the other for SURF):

- For SURF, we will use a database with 8 species.
- For ORB, we will use a database with 8 species and another database with 24 species.

As we said before, when we are calculating SURF we will use the FLANN Matcher, and we will use the BFMatcher for ORB.

The reason of why are we going to use two DB for ORB and just one for SURF, we will explain later. But mainly it comes from, as we are most probably use ORB for our project. We wanted to see what would happen if the DB were to be bigger. If it would have the same success or not.

The process, on which we will take the results come from are:
Take photo from a rodent’s skull.
Read the species list.
Convert the photo in black and white.
Create its Matrix.
Detect the keypoints.
Compute the descriptors.
Match the obtained descriptors with all the descriptors of the list.
And then respond to the question: Is the specie the correct one? Or not?

For our photos, we will use images with a plain background. And on the photos it will only appear the rodent’s skull. All the photos used can be found on a zip, added with this project.

There are two ways of inserting the photos:

- One way is to make a photo to the original rodent’s photo.
- The other way, is to use the original rodent’s photo.

3.2.2. Experimental results

3.2.2.1. Results

On both ways, for SURF and for ORB (on the DB of 8 species) we had similar results. If we were to use the original photo we would have a 95-100% of success. And if we were to use the photo of the original photo, then we would be getting a 85-90% of success. Although when you are taking a photo to a photo, many aspects comes in the way and should be taken in consideration in order to get an acceptable photo:

- Have good illumination.
- Do not tremble, when doing the photo.
- Have a good camera on your phone.

So after getting these good results, we decided to add some more species on the DB, to see what happened. And then is when all the problems came.

Once we made all the work and created the second ORB DB with 24 species. We tried the same to do the experiment as before: do a photo to the original photo and we had a 20-30% of success. At that moment we did not know what was going on. But then we decided do all the process with the original image and then we got again the 95-100% of success that we had before. So the problem was not with the images nor the DB. It was from the mobile camera.

After doing some testing, we came up that it did not make sense to compare photos from the mobile camera to photos made from professional cameras. So in order to make a test, we decided to create another DB with 24 species, all uploaded from photos made with the mobile phone. And then the number of success improved to 70-80%.
And in the DB, there were 2-3 species which were repeated. Thanks to that, they had a higher success ratio than the others.

### 3.2.3. Experimental conclusions

So in order to get the better results, the best that we could do is:

- Make sure that all the uploaded images have a good level of descriptors.
- Make the DB as rich as possible, the more number of repeated species that we have on the DB, the most precise resolution that we will have.
- Make all the photos the most equally possible. That way, we will also get the better results.
- Try to compare the images done, with the same camera as we uploaded the image, in order to get better results.
4. Costs

In this chapter of the document we will explain the costs of the project. In order to continue, we must notice that all the products chosen for this project were free (hosting, OpenCV libraries, test environment...).

But if we were to make this application commercial we should upgrade all these elements. For example adding a web domain, upgrade the hosting, buy iOS licenses, buy OpenCV licenses,... Although in this point it is true that the SURF method has a non-commercial license, but ORB on the other hand is completely free. And ORB is what we were to use if we were to sell our application.

Now we will show in detail the costs of the project, if we were to put it on the market:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed cost</td>
<td>1. IOS mobile phones for testing</td>
<td>1. 200€</td>
</tr>
<tr>
<td></td>
<td>2. Computers for developing</td>
<td>2. 600€</td>
</tr>
<tr>
<td></td>
<td>3. ORB licenses</td>
<td>3. Free</td>
</tr>
<tr>
<td>Periodic cost</td>
<td>1. Web hosting: Heroku Hobby plan</td>
<td>1. 5€/month</td>
</tr>
<tr>
<td></td>
<td>2. Apple developing license</td>
<td>2. 8€/month</td>
</tr>
<tr>
<td></td>
<td>3. Web domain</td>
<td>3. 1€/month</td>
</tr>
</tbody>
</table>

Figure 21. Project costs table

So summing the initial costs of the project, we would have a total of 814€ to get the materials. And then a recurring cost of 14€/month in order to continue working.

If in that price we also add the cost of of the design and development of the project. From which we used the 70% of the time on the mobile application and the other 30% on the web application. We should add to these costs:
<table>
<thead>
<tr>
<th>Description</th>
<th>Num. Credits</th>
<th>Hours/credit</th>
<th>Worked hours</th>
<th>Price/hour</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile App development</td>
<td>12 credits</td>
<td>30 hours</td>
<td>360 hours</td>
<td>60€/hour</td>
<td>21.600€</td>
</tr>
<tr>
<td>Web App development</td>
<td>6 credits</td>
<td>30 hours</td>
<td>180 hours</td>
<td>60€/hour</td>
<td>10.800€</td>
</tr>
<tr>
<td>Mobile App design</td>
<td>4.8 credits</td>
<td>30 hours</td>
<td>144 hours</td>
<td>40€/hour</td>
<td>5.760€</td>
</tr>
<tr>
<td>Web App design</td>
<td>1.2 credits</td>
<td>30 hours</td>
<td>36 hours</td>
<td>40€/hour</td>
<td>1.440€</td>
</tr>
<tr>
<td>Total</td>
<td>24 credits</td>
<td></td>
<td>720 hours</td>
<td></td>
<td>39.600€</td>
</tr>
</tbody>
</table>

Figure 22. Detailed project costs table with the number of hours

In resume, the costs of the project would be:

- Initial fixed costs 814€.
- Mensual costs 14€.
- Development cost 39.600€.

Suming a total of 40.414€ with a recurring cost of 14€/month.
5. CONCLUSIONS

As we have seen on the experimental chapter (back at chapter 3), we have succeeded in accomplishing all the requirements that we were asked to accomplish on this project. We were able to develop a mobile usable application, which it has the off-line capability to recognise the rodent’s type by taking a picture using some image recognition libraries. With a high probability of matching correctly, from the 60% to the 100% (it will depend on the number of species of the DB, the better the material in the DB has, the better results).

Apart from that, we must also mention that we surpassed by far the required time of the recognition by getting it as low as **1.2 seconds** to make a recognition process.

We also created a scalable database, which will connect the two applications, the mobile application and the web application. And from the web application (which acts a information managing tool), the user using some credentials will be able to **add/modify/remove the information**.

The developed product give us a simple solution for the need of recognising all types of rodents. Although, the application it is easily customisable, and by changing some parameters it allows us that the application also works well with other species, for example, pigs, cows, snakes, etc.

We were glad to develop this application as we think that it can really be useful for the professionals that work everyday with the rodent’s skulls. And by using this application it can make their work much easier and precise.

Although we delivered a good product, there are always improvements that could be done to the application, to make it even better, some of them are:

- Convert the application to CouchDB and PouchDB to the application, so it has a 100% closed workflow.
- Add as many new species as possible. The more species that the DB has, the more accurate will be the matching.
- Keep updated the web application. As we worked with beta frameworks, we should update them to the newest versions and packages.
- Add a strongest security to the web application.
- Develop the Android application.
- Develop the Windows phone application.
- Develop the image recognition web application.
References

# ANNEX

## Annex 1. Work Packages, Tasks, Milestones and PERT Diagram

### Work Packages (WP)

<table>
<thead>
<tr>
<th>Project: Initial specifications</th>
<th>WP ref: #1, #2, #3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: App specifications</td>
<td>Sheet 1 of 4</td>
</tr>
<tr>
<td>Short description: Make the skeleton of the application</td>
<td>Planned start date: 26/02/2016</td>
</tr>
<tr>
<td></td>
<td>Planned end date: 10/03/2016</td>
</tr>
<tr>
<td></td>
<td>Start event: Install OpenCV.</td>
</tr>
<tr>
<td></td>
<td>End event: Begin coding the mobile app</td>
</tr>
<tr>
<td>Internal task T1: Install OpenCV on IOS project.</td>
<td></td>
</tr>
<tr>
<td>Internal task T2: Design the app.</td>
<td></td>
</tr>
<tr>
<td>Internal task T3: Build the skeleton of the app.</td>
<td></td>
</tr>
<tr>
<td>Deliverables: Dates:</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Project: Develop mobile application</th>
<th>WP ref: #4, #5, #6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major constituent: SW</td>
<td>Sheet 2 of 4</td>
</tr>
<tr>
<td>Short description: Develop the mobile application, using some web functionalities.</td>
<td>Planned start date: 11/03/2016</td>
</tr>
<tr>
<td></td>
<td>Planned end date: 25/03/2016</td>
</tr>
<tr>
<td></td>
<td>Start event: Studying.</td>
</tr>
<tr>
<td></td>
<td>End event: Coding the mobile app.</td>
</tr>
<tr>
<td>Internal task T1: Study OpenCV.</td>
<td></td>
</tr>
<tr>
<td>Internal task T2: Begin creating web app.</td>
<td></td>
</tr>
<tr>
<td>Internal task T3: Make the mobile application work accomplishing the requisites.</td>
<td></td>
</tr>
<tr>
<td>Deliverables: Dates:</td>
<td></td>
</tr>
</tbody>
</table>
### Project: Finish and test mobile application

**WP ref:** #7, #8, #9

**Major constituent:** SW

**Sheet 3 of 4**

**Short description:**
Give the final touches to the mobile application.
And focus on the web site.

**Planned start date:** 26/03/2016
**Planned end date:** 15/04/2016

**Start event:** Studying.
**End event:** End coding the mobile app

**Internal task T1:** Study Angular 2.
**Internal task T2:** Study Firebase.
**Internal task T3:** Final touches mobile application and testing.

**Deliverables:**

<table>
<thead>
<tr>
<th>Dates</th>
<th></th>
</tr>
</thead>
</table>

### Project: Develop web application

**WP ref:** #10, #11

**Major constituent:** SW

**Sheet 4 of 4**

**Short description:**
Develop the web application and get everything working together.

**Start event:** 16/04/2016
**End event:** 15/06/2016

**Start event:** Studying.
**End event:** End coding the mobile app

**Internal task T1:** Develop web application.
**Internal task T2:** Create workflow between the web application and the mobile application.

**Deliverables:**

<table>
<thead>
<tr>
<th>Dates</th>
<th></th>
</tr>
</thead>
</table>

### PERT Diagram

![PERT Diagram](image)

**Figure 23. PERT’s Diagram**
## Milestones

<table>
<thead>
<tr>
<th>WP#</th>
<th>Task#</th>
<th>Short title</th>
<th>Milestone / deliverable</th>
<th>Date (week)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Install OpenCV on IOS project.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>Design the app.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>Build the skeleton of the app.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>Study OpenCV.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>Begin creating web app.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>Make the mobile application work accomplishing the requisites.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>Study Angular 2.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>Study Firebase.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>Final touches mobile application and testing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>Develop web application.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>Create workflow between the web application and the mobile application.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Annex 2. Case of use: Mobile Application

Figure 24. Mobile case of use
Annex 3. Mobile Application workflow design

Initial View

Select from gallery or make a photo
Species List

Figure 27. Mobile application species list pages

Favorites’ List

Figure 28. Mobile application favorites’ list pages
**Specie’s Details**

![Arvicola sapidus](image)

**Arvicola sapidus**

**Description**

El ratón de campo (Apodemus sylvaticus) es un micromamífero del orden de los roedores que es muy abundante en su área de distribución, comprendida entre Europa y la cuenca mediterránea. En ocasiones provoca daños en cultivos, aunque de poca importancia.

![Arvicola sapidus](image)

**Arvicola sapidus**

**Description**

El ratón de campo (Apodemus sylvaticus) es un micromamífero del orden de los roedores que es muy abundante en su área de distribución, comprendida entre Europa y la cuenca mediterránea. En ocasiones provoca daños en cultivos, aunque de poca importancia.
Others (Settings/Instruccions and app information)

Figure 30. Mobile application others pages
Annex 4. Web Application workflow design

http://shrew-upc.herokuapp.com/

Home page
Figure 31. Web application home page

Login page

Figure 32. Web application login page
Register page

Figure 33. Web application register page

Dashboard page

Figure 34. Web application dashboard page
Incompleted species page

Figure 35. Web application home page

Profile page

Figure 36. Web application profile page
Edit specie page
Figure 37. Web application edit specie page
Annex 5. How to compile the mobile application code

Requirements:

- A Mac computer.
- To have an apple license.

Tools:

- XCode

If you meet the requirements, then you are ready to continue with this guide.

**Step 1: Download the code:**

Download the code which can be found at the annex of this document.

**Step 2: Download CocoaPods**

Once downloaded, we will have to install the packages in order to make use of them. In order to do that we will install a package manager called CocoaPods ([https://cocoapods.org/](https://cocoapods.org/)), which will help us to install and to do control versioning for our project.

CocoaPods is built with Ruby and is installable with the default Ruby available on OS X. We recommend you use the default ruby.

Using the default Ruby install can require you to use `sudo` when installing gems. Further installation instructions are in the guides.

```
$ sudo gem install cocoapods --pre
```

**Step 3: Setup the package manager**

Once installed, we are ready to use it. If it is not yet installed in the project, we will need to create a document called Podfile in the root of our project and write this:
In this file we are saying which libraries we want to install and which versions we are using. For this project we will use Firebase, with a version equal or higher of 2.5 (the package manager will find out which one is the newest and install it) and we will install also OpenCV version 2.4.9 (we could not install a newer version of OpenCV, because it was bugged on IOS, at the moment that this project was created).

**Step 4: Install the libraries**

Now you can install the dependencies in your project:

```
$ pod install
```

Make sure to always open the Xcode workspace instead of the project file when building your project:

```
$ open App.xcworkspace
```

Now you can import your dependencies e.g.:

```
#import <Reachability/Reachability.h>
```
Optional step: Change the image recognition method:

If you would like to switch from method (from ORB to SURF or vice versa), it is very easy. You just need to go inside the code and in the script: CatchRat/Controller/CaptureVC/CaptureVC.mm you will have to change some variables depending on what you want:

```c
//0 para SURF
//1 para ORB
#define TYPE_MATCHING 1
```

And inside the function superAnalyzeDaescriptor...

```c
//En la función: superAnalyzeDaescriptor:(Species*)specie {...}
//Para SURF
//FlannBasedMatcher matcher;

//Para ORB
//BFMatcher matcher(NORM_L2, true);
BFMatcher matcher(NORM_HAMMING, true);
//ORB está activado por defecto
```

Step 5: Run the code!

Now we are all set up. We should be already able to run our application in any IOS device. Just connect the USB cable to your Mac.

On the XCode’s topbar select the device you want to compile and select your device and press the Run button!
Annex 6. How to compile the web application code

Requirements:
- Node
- npm - included in Node installation

Tools:
- Sublime Text
- Terminal

(*) Installation of Node

Go ahead and visit the node website and download Node. Run through the installation and you’ll have Node and npm installed!

If you meet the requirements, then you are ready to continue with this guide.

Step 1: Download the code;

Download the code which can be found at the annex of this document.

Step 2: Setup the package manager

At the beginning of this guide we installed Node and with it npm. Npm is a packet manager which will help us to install and to do control versioning for our project.

Make sure that you have the file package.json in your project and that it meets the required dependencies written down. If the document is not found in the project, we will need to create a document called package.json in the root of our project and write this:
Step 3: Webpack (up and running)!

As you might have noticed, if you have experience in web development, the project uses Webpack.

Webpack is a module bundler, this means that it takes modules with dependencies and emits static assets representing those modules. So this way, we have a very compiled version of our application and optimize the size the maximum possible.

As the project is written on Typescript (TS) and many browsers do not understand TS yet. We use webpack to transform our TS documents to javascript. So the browser can understand it.
In order to run the application we need to run these commands in the terminal:

- Npm install
- Npm start

This commands will install all the dependencies, and create a file called `bundle.js`, which is the minified version that we talk about before.

**Optional step: Deploy our app**

If we were to want to deploy our application to any hosting site. We would need to make use of Webpack and to create a static `bundle.js`, which would be like making a snapshot of the application in the moment of running the command.

- Npm run deploy
- Nodemon server.js

In order to be able to upload a version, we will need to create our local server, so that the hosting can read our code. In this application we will use **NodeJS**, because is the easiest and fastest solution, and works well with Angular 2!

Create a file called `server.js`, and write the following code:

```javascript
// get our packages
var express = require('express');
var app = express();
var bodyParser = require('body-parser'); // get body-parser
var morgan = require('morgan'); // used to see requests
var port = process.env.PORT || 8081;
var path = require('path');

// APP CONFIGURATION
//
// use body parser so we can grab information from POST requests
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// configure our app to handle CORS requests
app.use(function(req, res, next) {
  res.setHeader('Access-Control-Allow-Origin', '*');
  res.setHeader('Access-Control-Allow-Methods', 'GET, POST');
  res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,Content-Type, Authorization');
  next();
});

// log all requests to the console
app.use(morgan('dev'));

// set static files location
// used for requests that our frontend will make
app.use(express.static(__dirname + '/src'));

// start the server
app.listen(port); console.log('Magic happens on http://localhost:' + port);
```

And that is it! You have your code ready to be deployed in any host server. For example: Heroku.
Annex 7. Implementation of the OpenCV libraries

Once we have installed the libraries, we will take advantages of the following OpenCV libraries:

- 
  ```
  #import "opencv2/highgui/ios.h"
  #import <opencv2/opencv.hpp>
  #include "opencv2/nonfree/nonfree.hpp"
  ```

As we mentioned before, there are some steps that we must follow in order to get our final result. Let’s see how the code of each step looks like:

1. Convert the image in grayscale.

```
- (UIImage *)convertImageToGrayScale:(UIImage *)image
{
    // Create image rectangle with current image width/height
    CGRect imageRect = CGRectMake(0, 0, image.size.width, image.size.height);

    // Grayscale color space
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceGray();
    // Create bitmap context with current image size and grayscale colorspace
    CGContextRef context = CGBitmapContextCreate(nil, image.size.width, image.size.height, 8, 8, colorSpace, kCGImageAlphaNone);

    // Draw image into current context, with specified rectangle
    // using previously defined context (with grayscale colorspace)
    CGContextDrawImage(context, imageRect, [image CGImage]);

    // Create bitmap image info from pixel data in current context
    CGImageRef imageRef = CGBitmapContextCreateImage(context);

    // Create a new UIImage object
    UIImage *newImage = [UIImage imageWithCGImage:imageRef];

    // Release colorspace, context and bitmap information
    CGColorSpaceRelease(colorSpace);
    CGContextRelease(context);
    CGBitmapImageRelease(imageRef);

    // Return the new grayscale image
    return newImage;
}
```

2. Obtain the image’s matrix in CV_8U, which is necessarily for doing operations with IOS mobile phones.

---

CV_8U is unsigned 8bit/pixel - ie a pixel can have values 0-255, this is the normal range for most image and video formats.
3. Create the method you want to use with the parameters, that suits you the most.

For SURF:

```cpp
SurfFeatureDetector surf(2500);
surf.extended = false;
```

The parameter is the threshold for hessian keypoint detector used in SURF.

For ORB:

```cpp
OrbFeatureDetector orb(500, 1.2f, 8, 10, 0, 2, ORB::HARRIS_SCORE, 10);
```

The first parameter tells the extractor to only use the top 25 results from the detector. For a reliable estimation of an 8 DOF homography with no constraints on parameters, you should have an order of magnitude more features than parameters, i.e. 80, or just make it an even 100.

The second parameter is for scaling the images down (or the detector patch up) between octaves (or levels). Using the number 1.0f means you don’t change the scale between octaves, this makes no sense, especially since your third parameter is the number of levels which is 2 and not 1. The default is 1.2f for scale and 8 levels, for less calculations, use a scaling of 1.5f and 4 levels (again, just a suggestion, other parameters will work too).

The fourth and last parameters say that the patch size to calculate on is 10x10, that’s pretty small, but if you work on low resolution that’s fine. That is one of the key points in our project.

The score type (one before last parameter) can change runtime a bit, you can use the ORB::FAST_SCORE instead of the ORB::HARRIS_SCORE but it doesn’t matter much.
4. Define the Feature Detector, which will detect the key points of the image’s Matrix you pass.

For SURF and for ORB, is exactly the same

```cpp
vector<KeyPoint> keypoints;
orb.detect(image, keypoints);
```

```cpp
vector<KeyPoint> keypoints;
surf.detect(image, keypoints);
```

5. Define the Feature Extractor, which will obtain the visual descriptors from the image and its key points.

For SURF:

```cpp
SurfDescriptorExtractor surfDesc;
surfDesc.compute(image, keypoints, _descriptors);
```

For ORB:

```cpp
OrbDescriptorExtractor orbDesc;
orbDesc.compute(image, keypoints, _descriptors);
if(_descriptors.type() != CV_8U)
    _descriptors.convertTo(_descriptors, CV_8U);
```

We should not forget to make sure that these descriptors are from the type CV_8U, which will help our mobile phone to compute faster and to have less computational cost.

6. Define the Feature Matcher, which will be the one that matches the different descriptors.

For SURF, it could use both of them FLANN or BFMatcher with NORM_L1 or NORM_L2, but as we said before the recommended one is FLANN for SURF.

```cpp
FlannBasedMatcher matcher;
BFMatcher _matcher(NORM_L2, true);
```

For ORB, we only have the options of NORM_HAMMING or NORM_HAMMING2, NORM_HAMMING 2 is used when we decide to have the WTA_K higher than 2.

```cpp
BFMatcher matcher(NORM_HAMMING, true);
```

One of NORM_L1, NORM_L2, NORM_HAMMING, NORM_HAMMING2. L1 and L2 norms are preferable choices for SIFT and SURF descriptors, NORM_HAMMING should be used with ORB, BRISK and BRIEF, NORM_HAMMING2 should be used with ORB when WTA_K==3 or 4 (see ORB::ORB constructor description).

The true of the function is the crosscheck and is turned on, in order to get better results.

7. Pass the results for a filter, in order to know which are good results. And see which has the most good results. So this way we will know that it is the best match.
As we can see, in SURF we only get the results which are lower than 0.05 and in ORB we get the results which are lower than 40. This number is the approximate number between the points. The lower this number is the better. So we end up getting the result which have more good matches. And the one with the most of them will become the final result.

We must remember that this final result may be, or may be not, the correct result. All will depend on the quality of the images done, and the quality of the images in the DB. In an ideal world, all the photos to compare with should be done with the same camera and with the same quality. In order to get 100% of success in the photos.
Annex 8. SURF vs ORB

As we said many times before, SURF is much slower than ORB and it uses more CPU and its license is not free, then... **why some people prefer using SURF and not ORB?**

Well it really depends on what you are looking for. SURF is good at handling images with blurring and rotation, but not good at handling viewpoint change and illumination change. On the other hand, ORB is good for objects that have borders.

We can see at what we refer by looking at these images:

**SURF:**

![SURF keypoints](Figure 38. SURF keypoints)

**ORB:**

![ORB keypoints](Figure 39. ORB keypoints)

So by looking at these images, if we were to add these utilities in our project. If we were to use SURF, it would be very good for recognising the **eyes position, nose position** of the rodent for example. On the other hand, by using ORB, it would be very good for **detecting the teeth** of the animal, and other different things from the contour.

During the experiment we could see that in SURF, the descriptors size was much bigger than the one using ORB. In SURF we needed a col size of **1000-2500**, meanwhile in ORB with a col size of **500** was more than enough.

SURF, in some way, must be better when you do not have a plain background. Because it gets more descriptors from the image. And might be better at recognising the specie when the image has other things on its background.

By getting more descriptors, the more quantity of matches that you will have. And by having the more quantity of matches between two images, it means that the ending result will have better resolution and more precision.

In other words... Image that we are using ORB, in our ORB process the maximum number of descriptors that we are able to get is 500. Then if we were to make a photo with bad quality, we will most probably get **50/500 descriptors** from the image. That is a bad result. On the other hand,
if we were to take a good photo of the image we would get something like 450/500, the closer to 500 the better. Only that we have some bad images with low quality descriptors on our DB, it can affect all the other results, so we have to take care to all the images on the DB have a good quality.