Universitat de les Illes Balears

escola politècnica superior

Treball Final de Carrera

ENGINYERIA INFORMÀTICA

Augmented Reality on Mobile Devices

Álvaro Medina Ballester

Tutor
Ramón Mas Sansó

Escola Politècnica Superior
Universitat de les Illes Balears

Palma, September 29, 2014

THANKS

I would like to thank my wife for her continuous support and my parents for letting me have the chance to study in the University.

Many thanks to Ramon Mas Sansó for helping me with this work and thanks to the great teachers I had the chance to have in the University.

Also thanks to Pau Rul·lan Ferragut and Biel Moyà Alcover for helping me to write and develop this work.

Contents

LIST OF FIGURES

CHAPTER 1

Abstract

Augmented reality has become a very popular topic in the last years. With the introduction of mobile smart phones and their increasing capability of powerful CPUs and GPUs, developers started to have the chance to build AR applications on these devices. Between all the computer vision technologies available, OpenCV and Vuforia —specifically built for mobile devices— are among the most used. In this work a comparison of both technologies is made with the purpose of building a mobile application to enable users to see virtual posters on their walls.

CHAPTER 2

INTRO

Bringing augmented reality to mobile devices can be done in several ways. Getting data from the device's camera and process its output to track a selected object is one of the ways for querying the environment to introduce virtual elements in it. To accomplish this, many computer vision algorithms exist and they are called object recognition techniques.

Using an object recognition algorithm between two images can help us to determine whether the object we are searching for is present in the image or not. This technique can be seen in the figure 2.1. But, what happens if we want to know if the object is present on a dynamic scene such as a live video feed? We can apply the same algorithms for each of the video frame that comes from the camera. Also, we can combine these object detection algorithms with the data provided by the sensors of mobile smart phones, such as accelerometers or gyroscopes. This data can help the algorithms to compute the user movement and improve the tracking.

In order to determine which is the best technique to detect features in a video feed, we present an study of three different approximations using OpenCV algorithms and another one using Vuforia SDK[1] by Qualcomm®. A comparison of performance and robustness between this four different approaches is made to decide which is the best to bring augmented reality to a mobile environment. This comparison runs on a real device in order to determine how this techniques behave on a limited performance device such as a mobile smart phone.

[1]Software Development Kit.

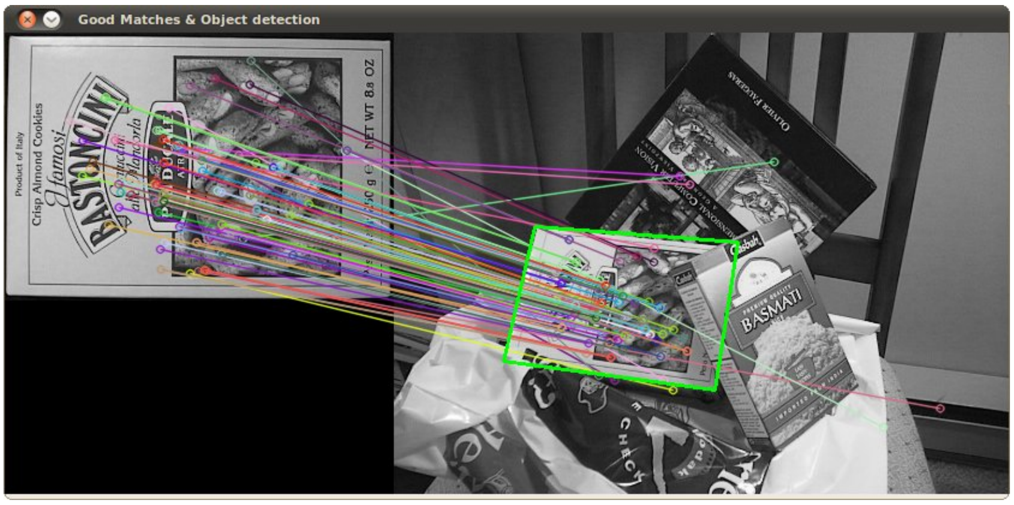

Figure 2.1: Object detection using OpenCV's SURF. Taken from [1].

The goal of the study is to build an augmented reality application that enables users to try posters on their walls using augmented reality technology. The name given to this application is Ponster.

Structure of the work

State of the art

In this chapter we are going to explain all the techniques used in Ponster to detect and track the object, what's the theory behind those algorithms and why we have chosen one above another. All the OpenCV approximations are detailed: template matching and the three feature tracking algorithms tested. Also, the theory behind Vuforia SDK is explained.

Technologies

All the technologies used to develop the Ponster app are explained in this chapter. Both iOS-only SDKs and computer vision technologies are detailed, from the user interface SDK to the persistence layer. Also, an extended overview of how Vuforia works is presented.

Development

In this chapter the architecture of the Ponster application is presented, along with the model view controller hierarchy and the persistence layer. Also, a performance comparison of all the augmented reality algorithms used is shown. Finally, we present a detailed explanation of the Ponster features.

Commercial applications and future work

We list different augmented reality applications built for mobile smart phones that have been an inspiration to build Ponster, and we give details about how the application can be improved with future work.

Conclusions

The final part of this work is presented as a summary of all the important things learned when bringing augmented reality —using object tracking— to a power constrained device as the mobile smart phones are.

CHAPTER 3

STATE OF THE ART

The technique of mixing real world elements with virtual elements displayed on the screen of a device is what we call augmented reality. In the field of augmented reality, a lot of things have happened during the last years. The progress in the fields of computer vision and image processing have led to several new techniques of detection and tracking. This, combined with the increasing availability of powerful mobile devices, has enabled developers to build a plethora of high quality AR-based applications. Nowadays, modern mobile devices use integrated cameras, motion sensors and proximity sensors to make these AR-based experiences.

Augmented reality in mobile devices is slightly different from what can be seen in desktop environments. Although every year we have more powerful mobile smart-phones, processing and drawing into the device's screen is still an expensive operation in terms of computational cost. This is one of the reasons why cost-efficient computer vision algorithms and techniques have emerged in the recent years.

Most of the augmented reality apps follow this behavior:

- Get the input from the camera or a video.
- Search for an object of interest.
- Introduce our object into the scene, considering the camera or the input position.

In this chapter we are going to describe which are the different techniques that enables us to do this kind of processing, but before that we need to explain some key concepts.

We call *source image* or *source object* to the image used as the pattern that the algorithm is searching for in the scene. Depending on the source image, the algorithms tested will perform better or worst. We define *robustness* as the quality of a tracking algorithm when the scene changes and the object to track is not equal as the source image. This difference between the image to track and the image presented in the camera can be due to *rotation*, *scale* or *perspective transformation*. Some algorithms described below are unaffected to some of this changes. For instance, being *rotation invariant* means that the algorithm is *robust* enough to still recognize the object in the scene despite of its rotation difference between the original object.

Before starting to introduce this technologies, it is interesting to explain the frame rate reference of 60 FPS and why is the main goal of many graphics visual applications. Douglas Trumbull introduced the 60 frames per second rate as the most convenient for human eye while developing Showscan, his cinematic process. The main reason for achieving this performance is that it enables a better emotional connection with the viewer because the frame rate is higher than the regular 24 FPS used in 35mm film. Trumbull ran laboratory tests with electroencephalograms and electrocardiograms when projecting 24, 36, 48, 60, 66, and 72 FPS films to viewers [2]. The result was that the stimulus when viewing 60 frames per second films was higher than others. Although the goal is to achieve 60 frames per second, the human eye perceives sloppiness at frame rates lower than 20 FPS, so getting a higher value than that is considered enough for most applications.

The goal of this work is to find a combination of techniques that enable us to develop an application with object tracking invariant to rotation, scale and perspective warp. The different algorithms tested were selected with this target in mind.

3.1 Object recognition

In order to provide an augmented reality experience, we have to know first which is the real world element that we are going to use as a reference to mix the real world input with our virtual elements. This reference can be from an image from the smart-phone camera to the user location. Our application searches a particular image inside the camera input in order to draw the poster image above. Many other augmented reality mobile apps do not necessarily use image processing techniques to mix real world elements with virtual ones. For instance, there are applications that helps the user to identify constellations with the mobile phone. These applications only use GPS location and digital compass orientation to enable the user to see which constellations are they pointing their device to.

The technique of searching an image and follow it along it's movement is usually referred as object tracking. In computer vision there are a lot of object recognition techniques. In the development of Ponster, two of these techniques have been tested: template matching and feature-based detection. Detecting the image in a continuous input from the smart-phone camera, taking account of scale, rotation and perspective differences, becomes an object tracking technique.

3.2 Template matching

Template matching[3] consists of finding areas of an image that are similar to a provided template image. We have to set as the input of the algorithm a template image —the image that we want to look for— and compare it with the source image —the image in which we want to search[4]—. Template matching is also called area-based approach.

OpenCV provides a way to perform template matching with several methods, such as square difference matching[1] or correlation-matching methods[2]. With the latest, `CCORR`, we use a correlation formula to check if the template is inside the image. Instead of applying a yes/no approximation, we can bring a positive match with a certain threshold.

Performing a template matching operation using OpenCV on mobile devices is fast enough to deliver a smooth 25/30 FPS[3]-like detection. However, match template does not take account of scale, rotation and perspective invariance by itself. There are several approaches to bring invariance to match template. For instance, image pyramids are used to make match template scale and rotation invariant[5], but it is not part of the OpenCV match template function, although it provides some methods to implement image pyramids[6]. A visual representation of an image pyramid can be seen in figure 3.2.

[1]`SQDIFF`.
[2]`CCORR`.
[3]Frames per second.

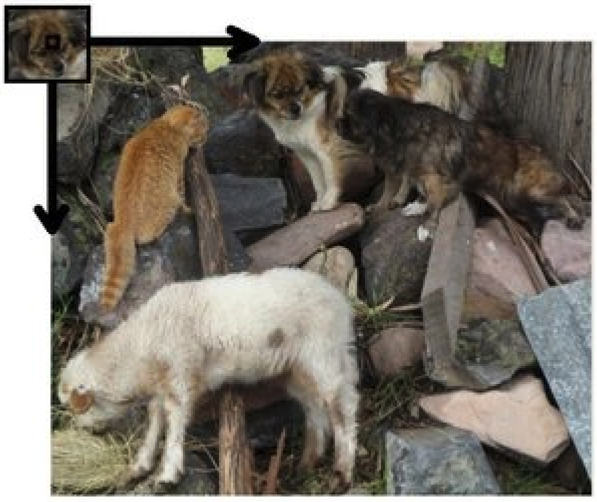

Figure 3.1: How the template matching algorithm works in OpenCV. Taken from `http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html`

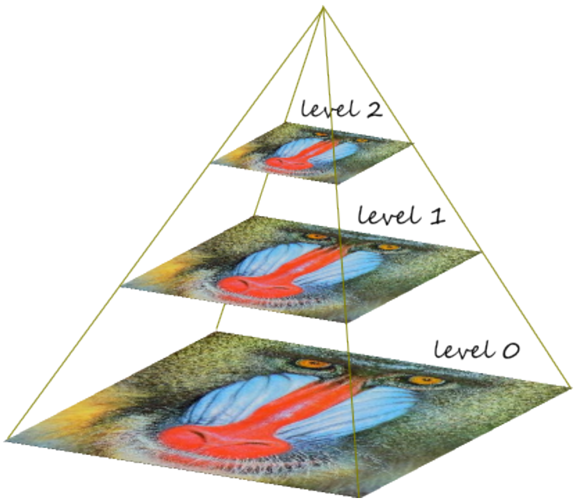

Figure 3.2: An example of image pyramid technique. Taken from [7].

In the testing of template matching in Ponster, a basic image pyramid algorithm has been developed. Our simple image pyramid algorithm consists of building several sizes of the pattern image provided to the app. We generate four smaller versions of the image and then we try to match them in each frame of the camera loop. If any of the images is detected, we take account of the image scale to draw the poster above the detected pattern. In our tests this worked really well, but more scale images were needed to bring a better scale invariance template matching algorithm.

Match template has been tested during the development of Ponster. Also, a basic image pyramid system has been developed for scale-invariance, but match template has been discarded in favor of feature detection algorithms because they bring rotation invariance and perspective warp, and this are required features to deliver a good augmented reality experience.

3.3 Feature detection

A feature-based approach can be presented as a three step method. First of all, we have to detect keypoints[8] —also called interest points— in the image. Usually, interest points are corners, blobs[4] or junctions —see figure 3.3—. A good keypoint is a *repeatable* keypoint; if we can find the same keypoint under different conditions such as light difference or rotation, it's considered as a quality keypoint. The second step is to compute *descriptors* or feature vectors. These descriptors are represented as neighborhoods of interest points. Assuming that feature detection makes sense when we have two images to compare, these two steps have to be performed on both images. Once we've done that, we have a group of descriptors for each image, and we have to compare them in order to *match* features. If the features of the source image are present in the input image, we can assume that the object has been detected. The matching is based on the distance between the feature vectors.

Usually, source images with enough keypoints are easier to detect than more uniform images. This is why it's better to select a good source image with many features and good contrast. As we've said before, in order to deliver a good augmented reality experience, we need to make our detection algorithm scale, rotation and perspective invariant. Feature detection techniques can be scaled invariant by extracting features that are invariant to scale, such as feature

[4]Regions and points that have different brightness, color or contrast compared with their neighbor region.

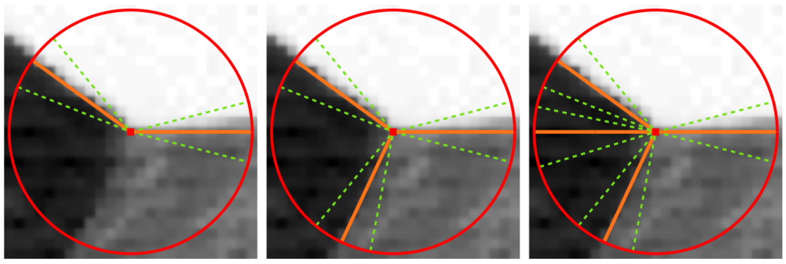
Figure 3.3: Different types of junctions, taken from [9].

vectors computed from interest point neighborhoods. For rotation invariance, algorithms can estimate the orientation of the keypoint.

There are plenty of feature-detection based algorithms, many of them based on Scale-Invariant Feature Transform, or SIFT. Cost efficiency is one of the most important features of these algorithms, as every new technique introduced tries to maintain robustness while reducing computation time. Robustness it's also very important, but less robust algorithms are also been developed in favor of more efficient techniques. One good example is FAST[10] keypoint detector, which is not rotation invariant but it's faster to compute.

Next, we are going to describe the feature-detection algorithms tested during the development of Ponster: SURF, FREAK and ORB.

3.3.1 Speeded-Up Robust Features - SURF

Speeded-Up Robust Features —also know as SURF— is a group of detector and descriptor introduced by Herbert Bay et al. SURF is faster and more robust than other alternatives like SIFT[11]. Its descriptors are rotation and scale invariant. Perspective transformations are also considered, but in lower order.

The keypoint detection in SURF uses a Hessian-matrix approximation. This use of integral images reduces computational cost in comparison with another interest point detection techniques such as Harris corner detection. Scale invariance is achieved by calculating integral image pyramids, but instead of reducing the image size, integral images allow SURF to upscale and build the pyramids more efficiently.

The SURF descriptor calculation is slightly based on SIFT. SURF descriptor describes the distribution of the intensity content within the interest point neighborhood, which is similar to the gradient information used by SIFT. It is done in two steps, fixing a reproducible orientation based on information from a circular region around the keypoint, and then building a square region aligned to the calculated orientation. Once we have the descriptors, the last step is to perform the matching. Descriptors are compared only if they have the same type of contrast, allowing to perform a faster matching. In Ponster, two different matching algorithms have been tested, Brute-force Matcher and FLANN-based matching.

SURF is as robust as other alternatives such as SIFT, but it's faster to compute due to the use of integral images. It is rotational and scale-invariant, which is better than the template matching technique described before, but it's performance running on the device (iPhone 5, iOS 7.1.2 and iOS 8) is not good enough to deliver a decent user experience, taking between 0.7 and 1 second to compute each image, as can be seen in the performance tests in section 5.2. Also, SURF is a patented algorithm and it's not allowed its use in commercial applications.

3.3.2 Fast Retina Keypoint - FREAK

FREAK stands for Fast Retina Keypoint[12]. As we have stated before, the trend is to make descriptors faster to compute while remaining robust and scale, rotation invariant. FREAK uses a keypoint descriptor designed like the human retina with this goal in mind.

Fast Retina Keypoint proposes to use a retinal sampling grid that takes account of the higher density points near the center of the sample points. This is exactly how the human retina works, and the FREAK sampling pattern imitates this natural behavior 3.4. With FREAK, every keypoint gets this retina sampling pattern to compute its descriptors. Then, the algorithm compares pairs of image intensities obtained by this pattern.

One of the main reasons of choosing FREAK to test as a descriptor combined with SURF is that it has more robustness in term of rotation and scale invariance than the SURF descriptor extractor. The comparison of SURF with different descriptor extractor[13] shown in the figures 3.5 and 3.6 shows that the robustness of the detection in both parameters is increased when using FREAK.

In the section 5.2 we present a comparison between the three approaches chosen for the object detection. We can clearly see that using FREAK as the descriptor extractor can boost the performance compared with SURF descriptors, but it also delivers less matches.

3.3.3 Features from Accelerated Segment Test - FAST

FAST is a keypoint detector based on corner detection. It was introduced by Edward Rosten and Tom Drummond[14] and its primary purpose was to bring

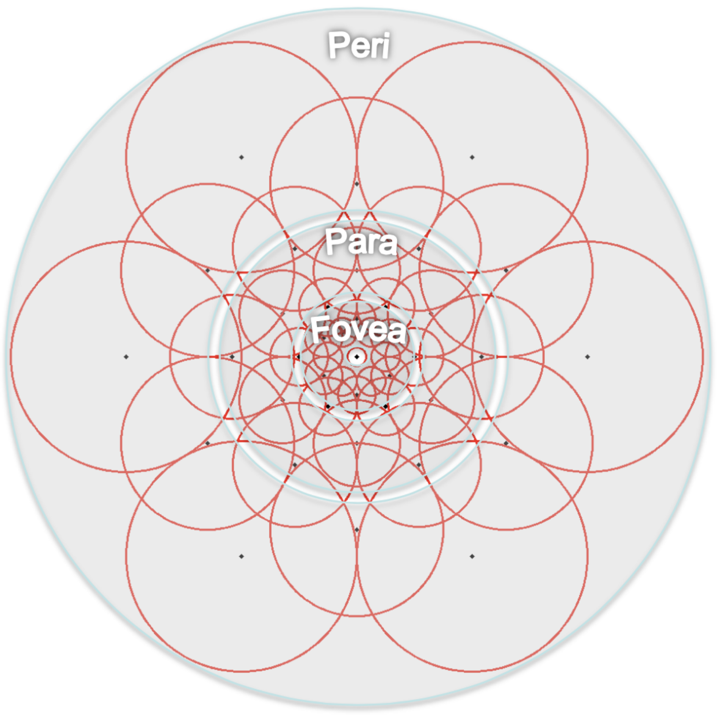

Figure 3.4: FREAK sampling pattern with it's equivalent in the human retina.

a real time interest point detector. FAST considers a circle of sixteen pixels around each corner candidate, and detects a candidate as a corner if there are n contiguous pixels in that circle with brighter intensity than the candidate pixel.

This technique is faster than others for corner detection, but FAST is not scale and rotation invariant. Also, it does not perform very well under high noise images. Many other techniques uses FAST as a starting point of a detector, bringing scale and rotation invariance and a corresponding extractor. One example of this is ORB, tested in the development of Ponster.

3.3.4 Oriented FAST and Rotated BRIEF - ORB

As we have said before, real time performance has been a very popular topic in object detection during the last years. The main characteristic of ORB is to perform as good as SIFT in terms of robustness, but doing it twice as fast. ORB uses a variant of FAST as the interest point detector, and BRIEF as the descriptor extractor.

FAST does not have rotation invariance. This is why ORB uses oFAST (FAST keypoint orientation), which is a variant of FAST that computes orientation by

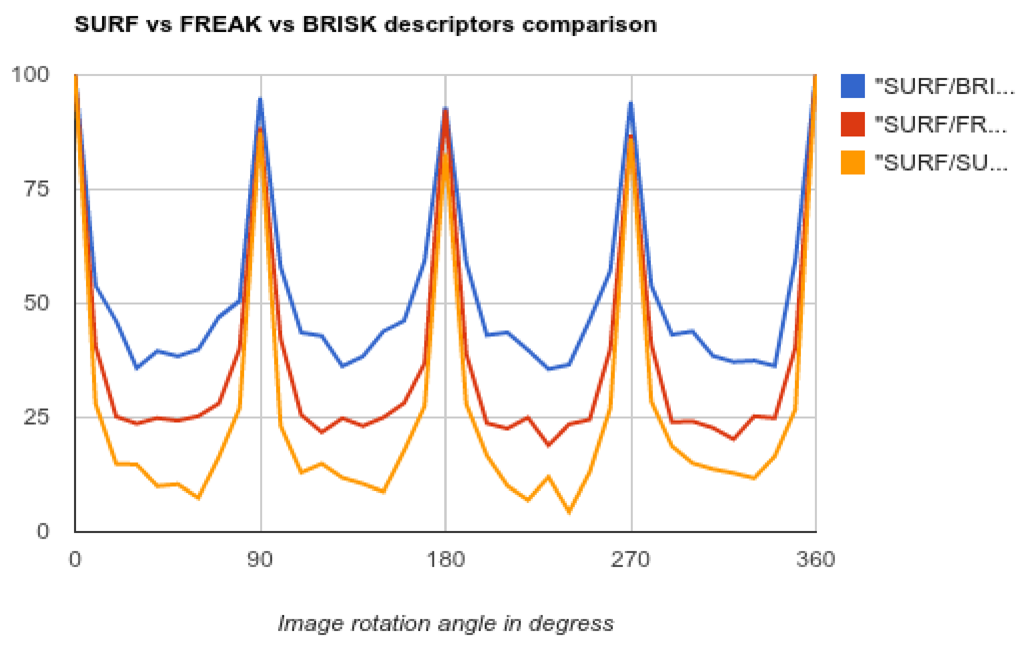

Figure 3.5: Rotation invariance comparison between different descriptor extractors combined with SURF. Source [13].

intensity centroid. This technique assumes that a corner's intensity is offset from its center, and this vector may be used to impute an orientation[10]. To bring scale invariance, ORB employs a scale pyramid of the image and computes FAST on each level of the pyramid.

ORB also uses a variant of BRIEF called rBRIEF, or Rotation-aware BRIEF. The BRIEF descriptor, unlike SURF, is a binary descriptor. rBRIEF is based on steered BRIEF, which uses the keypoint orientation; in addition to steered BRIEF, a learning method for choosing good binary features is applied, resulting into rBRIEF.

In Ponster, ORB has been tested with better results than the other previous techniques, but again with poor performance in the device[5]. Only a 15 FPS processing has been achieved, with slightly the same detection quality as SURF, as it can bee seen in the chapter 5.2. It is interesting to note that, although each of the algorithms tested are scale invariant, they tend to lose the tracked image when the scale of the object represented in the camera feed is 50% smaller than the original. This is also happening with SURF-SURF and SURF-FREAK

[5]We can consider poor performance a FPS under 20.

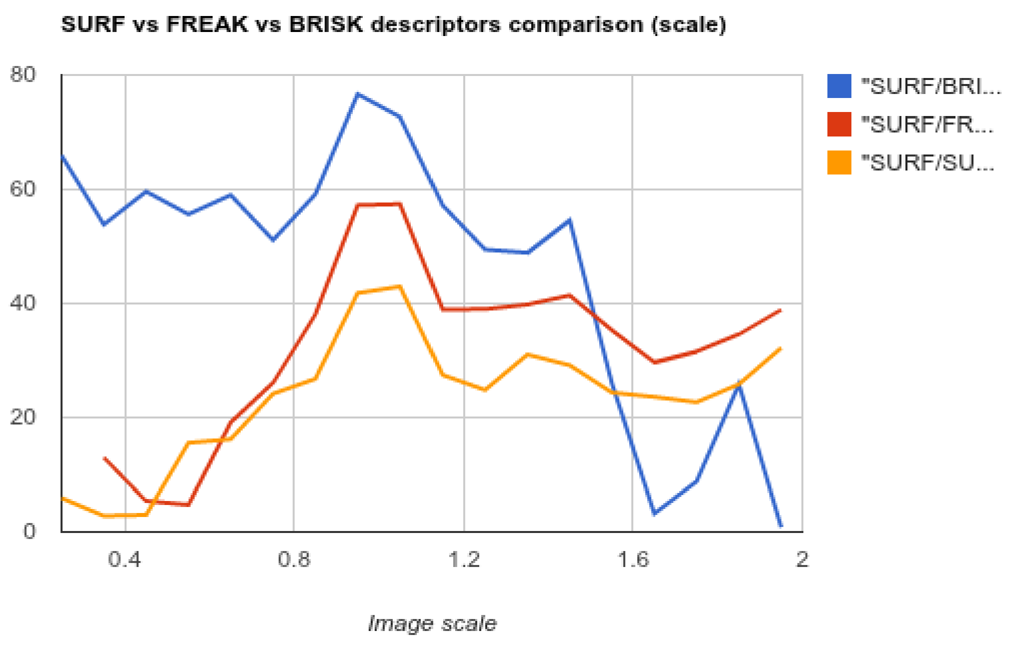

Figure 3.6: Scale invariance comparison between different descriptor extractors combined with SURF. Source [13].

combinations 3.6.

3.4 Matching

Once we have calculated the interest points and computed the descriptors in the two images that we want to compare, we have to perform a match between this two sources. Depending on the descriptor extraction method, one or another matcher must be used. ORB uses binary descriptors, but SURF does not, so the matching is performed in a different way.

We will discuss two of this methods, both used in the development of Ponster, Brute-force matcher and Fast Library for Approximate Nearest Neighbors.

3.4.1 Brute-Force Matcher

Brute-Force Matcher, as it's name states, will compare each of the descriptors found in the images, thus performing a linear search. Although it may seem that this approach is not very efficient, BF-matcher performs really well on binary descriptors like ORB.

The comparison is done by a distance function. There are many functions in BF-matcher:

- `NORM_L1` better with SURF/SIFT.
- `NORM_L2` better with SURF/SIFT.
- `NORM_HAMMING` better with ORB.
- `NORM_HAMMING2` better with ORB.

Hamming distance can be computed with bit manipulation operations, which are very quick. In Ponster, Hamming distance has been tested for ORB and L2 normalization with SURF.

3.4.2 Fast Library for Approximate Nearest Neighbors - FLANN

Instead of performing a linear search for matching descriptors, we can use a nearest neighbor matching technique. The nearest neighbor search tries to find, given a set of points P in a vector space X, all the points that are close to a given point q. FLANN[15] is a library that enables us to perform this kind of searches with several algorithms. Two have been tested in Ponster, randomized KD-tree search for SURF and Locality-Sensitive Hashing for ORB.

Randomized KD-tree

Basic KD-tree search performs well for small datasets, but quickly degrades its performance when the dimensionality increases. In order to reduce the computational cost of KD-tree searches with large datasets, several KD-tree algorithm variants have been introduced, such as approximate nearest neighbor.

This algorithm creates multiple randomized KD-trees, built by choosing the split dimension randomly from the first 5 dimensions on which data has the greatest variance. Then, a priority queue is created while searching all the trees, so the search can be ordered by increasing distance to each bin boundary.

Using this approximation techniques can boost performance by reducing the precision of the matching, although the loss is usually small enough to maintain a 95% precision.

Locality-Sensitive Hashing

Locality-Sensitive Hashing is a matching algorithm to solve the nearest neighbor search in high datasets. LSH is used with binary descriptors like the ones computed with ORB. The main idea of LSH is to hash the points with functions that ensure that close points will be more likely to key collision, thus allowing to get the nearest neighbors of each point querying the other points in it's bucket[16].

The LSH parameters defines the hash functions *amplification*. This means that the hash functions must be *amplified* enough to ensure hash collision; otherwise, the algorithm would be useless. The effect of this parameters and a more in-depth explanation of LSH can be found in this [16] paper.

3.5 Natural feature tracking

Although the technologies explained in previous sections are robust and reliable, and have been successfully tested, they have not been used in the last version of Ponster due to it's poor performance on mobile devices. Another technique has been proven to be successful to deliver both high performance in mobile devices and robust object tracking.

Natural feature tracking consists of computing the motion of a feature in the scene[17]. We call nature feature to any point or region candidate to be detected and selected as a reference. Usually this follows the next steps:

- Natural feature detection and selection.
- Motion estimation based on detected features.
- Evaluation feedback for stabilized detection and tracking.

The feature detection consists in selecting points and regions in the image with specific characteristics that are easy and robust to track. The motion estimation can be executed by several ways. For example, in Neumann's paper [17], optical flux is presented to estimate the camera movement. In our mobile environment, we can also use the built-in gyroscope of the iPhone 5. The evaluation feedback provides a way to reject detected features that do not specifically match with the plane that should be representing, thus allowing the algorithm to avoid false positives and do not corrupt the tracking output.

The technology used in Ponster to perform the augmented reality is called Vuforia and uses Natural feature tracking to track the object in the scene. In the section 4.2.2 we describe how Vuforia works, although it is a propietary solution and thus we cannot have detailed information about how internally does the tracking.

CHAPTER 4

TECHNOLOGIES

Ponster is an augmented reality app developed for the iOS platform. In order to make possible all the features that enable us to try the poster images in the camera scene, two main technologies have been used. First of all, the iOS SDK provided by Apple® is needed to develop any iOS application. For the augmented reality, OpenCV has been the first SDK tested, but finally the Vuforia SDK[18] developed by Qualcomm has been the one used in Ponster. We will start this discussing first the iOS SDK, and then the augmented reality technologies used.

4.1 iOS

The Apple iOS SDK has been used to develop the native, augmented reality application. The development started using the 7.1 version and the final release has been made with the latest SDK version, iOS 8. Several frameworks have been used to enable us to show the posters with a waterfall layout, fetch the data of the posters from the local database and to get input from the camera of the device.

The most important frameworks used are described in the next sections, and include UIKit, CoreData and third party libraries.

4.1.1 UIKit

UIKit[19] is Apple's framework to build iOS interfaces both in iPad and iPhone devices. All of the UIKit's classes inherit from a common interface object called `NSObject`. This framework provides classes to manage gestures, fonts, navigation bars, tab bars, text inputs, images, tables, buttons and many more elements. Almost every iOS application uses UIKit in one way or another. Only some video games do not make extensive use of the framework due to the specific user interfaces and game engines that most of them are based on.

Most of the Apple's frameworks naming convention date back from the NeXTStep era. They're written in Objective-C and they use the two-letter class prefix in all of them, making easy to identify to which framework belongs each class. For instance, all the UIKit classes have the `UI` class prefix, so `UITextField` or `UIView` both belong to UIKit. In other hand, `NSObject`, which belongs to Foundation framework, has the NeXTStep `NS` prefix.

In Ponster several UIKit features have been used. For the main user interface, a navigation-based UI is provided by `UINavigationController`. The main screen of Ponster shows a `UICollectionViewLayout` interface with a custom waterfall layout. This layout enables us to fit images with different sizes preserving their aspect ratio. Each image represents a subclass of `UICollectionViewCell` with custom elements, such as a `UIImageView` for the poster and a `UILabel` for it's title. In the image 4.1 we can see an example of the navigation-based interface of Ponster and the collection view layout.

Other UIKit features used in Ponster include `UIButtons`, `UISwitch` and `UIGestureRecognizers`.

4.1.2 CoreData, persistence layer for iOS applications

The main purpose of the research presented in the chapter 3 was to bring augmented reality to display posters in any surface the user wants to. However, Ponster app has been developed with the addition of more features in mind. One of this features is data persistence.

In order to maintain a list of all the posters included in the app —they are bundled inside the app in this version—, we have several methods to include and display this information. The simplest option would have been to save the data in the preferences `.plist` file and then read all the values. Also, we could have used `NSKeyedArchiver` to save data from our object model into the application sandbox[1]. But the best way to maintain data and to query it from

[1]The application sandbox is the folder that contains the application itself and all its data.

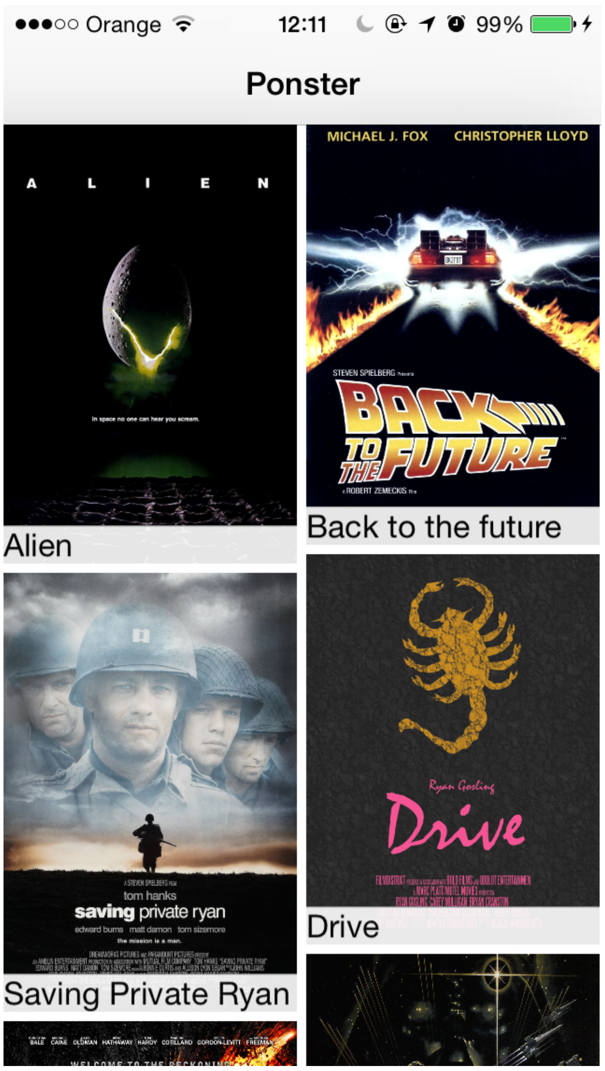

Figure 4.1: Main interface of Ponster demonstrating the use of navigation controllers and collection layouts.

our application is to use a real SQL database. CoreData integrates the SQLite database with a class-based model approach that enables us to save and retrieve information easily.

With the CoreData framework, we design an object model (figure 4.2) and then we generate the model classes. Each class is a subclass of `NSManagedObject` that has all the attributes that we've added in our model. Inside the application, we can query our model by class, and then retrieve the objects as an array. Each model object is like any other object, and we can access to it's properties using the common dot notation. For example, we can access to the `imageUrl` property of a `Poster` entity in the following way:

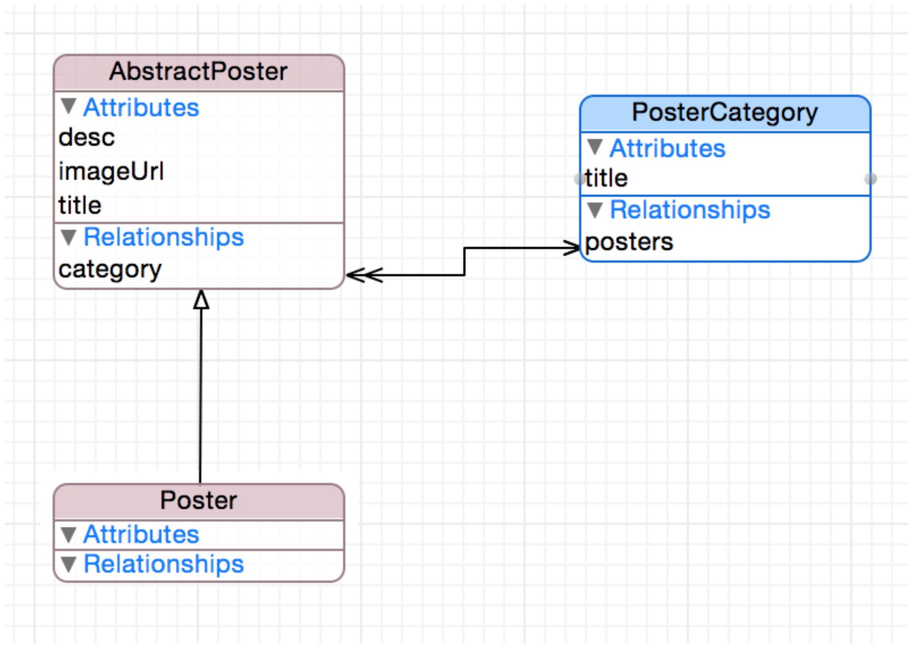

Figure 4.2: Basic data model of Ponster. The CoreData model editor enables us to design the model and then generate all the object classes. The double arrow represents a to-many relationship and the white arrow represents object inheritance. This model is explained in chapter 5.1.2.

```
Poster *poster = (Poster *)item;
UIImage *posterImage = [UIImage imageNamed:poster.imageUrl];
```

CoreData's stack enables us to perform queries and to save information in the SQLite store. Due to the fact that iOS applications use a maximum-priority thread for the UI calculations and several other threads with lower priority to perform other tasks, it's fundamental to understand how CoreData performs it's operations and how can we use it efficiently. In the section 5.1, our CoreData architecture is presented with this goal in mind.

Three objects are fundamental to CoreData, the **context**, the **coordinator** and the **store**. The `NSManagedObjectContext` is used to perform any save or query operation to the CoreData stack. The `NSPersistentStoreCoordinator` is the object that communicates between the contexts and the stores. Finally, the `NSPersistentStore` is the responsible of applying the changes to the SQLite back-end. The store and the coordinator are initialized once you start your app. All the data that we've saved is present in the SQLite file, inside our application sandbox. Apps can use more than one context, as we discuss in 5.1, and every operation against the database must be performed in a context. For example, when we want to perform a request for a selected *entity*, we have to specify the context:

```
NSFetchRequest *request = [[NSFetchRequest alloc] init];
NSEntityDescription *entityDescription = [NSEntityDescription
entityForName:entityName inManagedObjectContext:context];
[request setEntity:entityDescription];
// Build and perform the query
[request setPredicate:...];
```

Finally, CoreData works very well with another important class, `NSFetchedResultsController`. This class allows us to watch for changes in any entity in our model, with any predicate we want, and then perform an automatic refresh of our data every time some change in our model has been done in the observed classes. Thus, the fetched results controller is the class that communicates our view controllers with the data model.

4.1.3 AVFoundation

AVFoundation is used in Ponster to get the images from the camera of the device. A `AVCaptureVideoDataOutput` can be configured to get the frames of the camera. This output can be configured to deliver an specific amount of frames per second, the pixel format of the output or the orientation of the camera. Each of the frames delivered by the `AVCaptureVideoDataOutputSampleBufferDelegate` is sent to the augmented reality algorithm to perform the matching.

4.1.4 Third-party libraries

Several third-party libraries are used in Ponster. One of them is the OpenCV SDK and Vuforia SDK, which we're going to discuss in the next section. The other two interesting libraries are **MagicalRecord** and **PDKTCollectionViewWaterfallLayout**, along with the package dependency manager **CocoaPods**.

MagicalRecord

MagicalRecord[20] is a library written by Saul Mora that makes easy to use CoreData. It acts as a wrapper for most of the CoreData's actions and makes possible to perform any action with less lines of code. Also, MagicalRecord helps us to create new contexts, which is very important in our application architecture.

PDKTCollectionViewWaterfallLayout

PDKTCollectionViewWaterfallLayout is a `UICollectionViewLayout` that enables us to organize the cells of the collection view in a waterfall-style layout. With this we can display different cell sizes by preserving their aspect ratio and maintaining the same width.

CocoaPods dependency manager

This two third-party libraries, along with OpenCV, have been installed and added to Ponster by using CocoaPods. CocoaPods is the dependency manager for Objective-C projects[21]. It is an open-source project sponsored by several companies that it's becoming very popular among iOS and Mac developers.

With CocoaPods it is easy for developers to generate a *Podspec* for their projects and share them with the community. For instance, an iOS OpenCV compilation is maintained by several developers and they have recently added a Podspec —a small file with the description of the version of the project, the repository of the source code and their dependencies— to make easier to integrate OpenCV in iOS projects. With CocoaPods we just have to create a `Podfile` with all the dependencies we want to install, and then by just entering `pod install` in the project directory, all the dependencies are installed and linked as a subproject.

This dependency manager allows developers to easily add or remove dependencies to their projects, and also makes easy to share their libraries with others. CocoaPods is distributed as a Ruby gem and it can be installed easily in any Mac computer.

4.2 Computer vision

Two computer vision libraries have been used, OpenCV in the first stages of development, and later Vuforia, which is made by Qualcomm.

4.2.1 OpenCV

OpenCV is an open source computer vision library[22]. The library has more than 2,500 computer vision and machine learning algorithms. It has been released as a BSD-licensed source, so most of the code it contains can be used for commercial and non-commercial applications.

The OpenCV library can be used in Windows, GNU/Linux and OS X platforms. The latest OpenCV library available for iOS platforms is the 2.4.9 build[23].

Some of the algorithms used in Ponster –like SURF or SIFT– are patented, so they are available in the `non-free` headers. These algorithms cannot be used in commercial applications, but an implementation is included in the library with educational purposes.

There are a lot of applications that use OpenCV. C++ and Python programming languages are supported, among others. The C++ version is the one that has been tested in Ponster, using C++ and Objective-C++ files. Although OpenCV has a very good performance in most platforms, the Vuforia alternative performed much faster and with a constant 30 FPS frame rate.

4.2.2 Vuforia

Vuforia is an augmented reality SDK developed by the American chip company Qualcomm. The Vuforia API supports C++, Objective-C, Java and .NET —with the Unity game engine—. Currently, Vuforia is at it's 3.0 version.

The SDK consists of several core components[24]:

Camera
Delivers the image frame with a custom data structure, independent from the platform.

Image Converter
This downscales the image for a better performance and converts the image format to a suitable OpenGL ES format.

Tracker
The tracker performs the computer vision algorithms and stores the result in a shared object used by the background renderer. The tracker has many different detection algorithms that can be used depending on the camera image.

Video Background Renderer
This is the responsible of rendering the video image into the screen. It's optimized for the different platforms that Vuforia works with.

Application Code
For each processed frame, we have to perform three main steps: query the state object for newly detected targets or updated states, update the application with the new information received and render the graphics. This scheme is almost identical to the first custom scheme used when testing OpenCV feature detection methods.

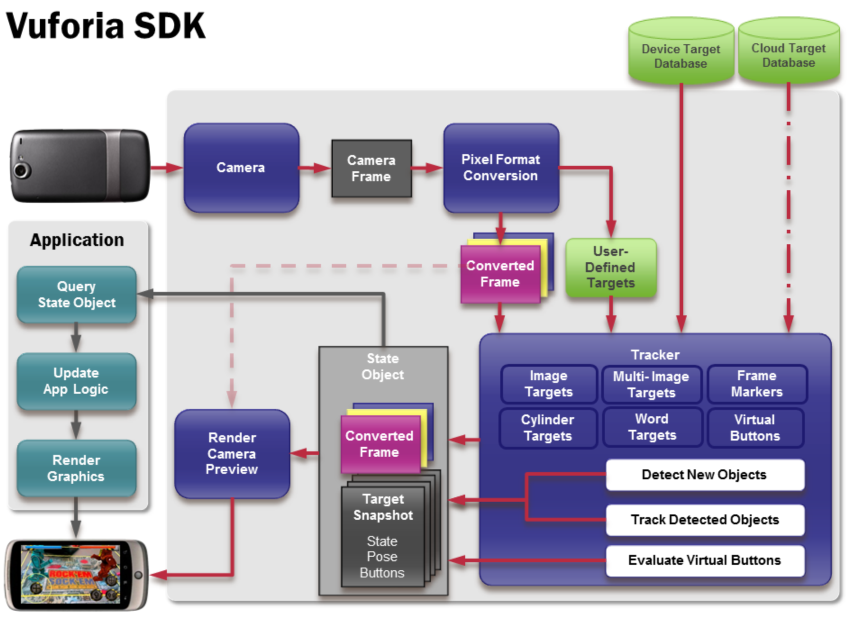

Figure 4.3: Vuforia SDK architecture.

User-defined Targets

In Ponster, users can define a tracking object by taking a picture of the scene, thus enabling to use any image as a placeholder of the poster. This works by processing the captured frame and caching its result for the augmented reality session. Vuforia enables developers to use pre-defined targets also.

One of the main advantages of Vuforia SDK is that it is optimized to run in ARM-based devices. In the tests performed during the development of Ponster, Vuforia has shown a clear advantage in terms of frames per second processed in comparison with OpenCV. While SURF performed roughly at 1 or 2 FPS, Vuforia has a constant FPS rate of 30 to 29 frames.

Extended tracking

Although we don't know exactly how Vuforia tracking algorithms internally work, some parts of the API architecture are explained in the Vuforia developer website. In the chapter 3 we have explained how the tracking algorithm uses natural feature detection. It is worth to mention how the extended tracking works and what it represents when the features detected are not good enough because it does not appear entirely on the screen.

The extended tracking enables augmented reality applications to maintain the object detection even if the whole pattern is not visible. With iOS, Vuforia can bring this using CoreMotion framework. It is not defined how exactly Vuforia makes extended tracking possible under iOS applications, but it is sure that they use the accelerometer and the gyroscope to keep the object visible in the scene while the user moves the smart-phone camera changing the scene.

FastCV

Vuforia is based on FastCV, which is another library developed by Qualcomm that brings computer vision algorithms optimized for mobile architectures. FastCV has the main computer vision features[25]:

- Gesture recognition algorithms.
- Face detection, tracking and pattern recognition.
- Augmented reality.

When running on ARM devices, FastCV is CPU-optimized. Due to the fact that Qualcomm builds the Snapdragon ARM microchip, FastCV is finely tuned for that architecture. Internally, Vuforia uses all of the FastCV algorithms, but it also takes advantage of other inputs such as the gyroscope or the accelerometer.

CHAPTER 5

Development

5.1 Application Architecture

All the iOS applications follow the model view controller architecture. This architecture separates the data model —inside the *model*—, the presentation of the data —the *view*— and the interaction and logic between them —the *controller*—.

First of all we're going to discuss how Ponster applies the MVC architecture; then we will introduce the selected persistence layer with CoreData and finally how the augmented reality fits into the app.

5.1.1 Model View Controller

Each of the main components of Ponster is represented by a subclass of UIKit's controller, the `UIViewController`. When developing complex applications is frequent to have a base view controller with shared functionality. Then, the rest of the view controllers inherit from it. In Ponster, the main view controller from where all of our controllers inherit from are the ones presented by UIKit, without any other feature added.

We can separate the view controllers in our app with the following list:

- Main screen view controller.
  - Collection view controller.

- Poster view controller
- Augmented reality view controller.

There is one view controller that is built from two view controllers, the main screen. It is common in iOS to represent tables or collection views using the UIKit view controller that is ready for those tasks, `UITableViewController` or `UICollectionViewController`. Usually this is done because both view controllers have built-in methods like the refresh control that are easier and more correct to use when sub-classing from those UIKit view controllers. In order to customize the rest of the view controller and to keep responsibility separated —one view manages the collection, the other the whole screen— we use view controller containment to embed the collection VC inside the main screen view controller. This allows us to keep the single responsibility principle and to have lighter view controllers.

Apart of the view controllers, all the data model is separated into `NSManagedObject` subclasses. Each of that subclass represent an entity in our model. Business logic is usually implemented as a category of each of the model subclass. Categories in Objective-C are capable of adding methods to any object without having to change its implementations. This logic is usually added as a category in order to not interfere with the `NSManagedObject` subclass, because every time a change is made to a model, we have to generate the subclass again and this would erase all the logic that we could have built inside.

The only custom view subclass we have in Ponster is the cell subclass to represent each poster in the main screen view controller. The `UICollectionViewCell` subclass called `PNSPosterCollectionViewCell` encapsulates all the views that are needed to display the poster image and it's title. We just need to provide the `Poster` entity and this subclass manages to draw the entire view.

5.1.2 Persistence layer architecture

In order to deliver a good user experience, we have to understand the iOS architecture. iOS has a high-priority thread called *Main Thread* where all the UIKit operations must be executed. Thanks to this, the responsiveness of every UI has the top execution priority, but it also has downsides. If our code is blocking the Main Thread, it will also block the user interface, thus delivering a poor user experience. To solve this potential issue, we have to send all the possible operations to another threads.

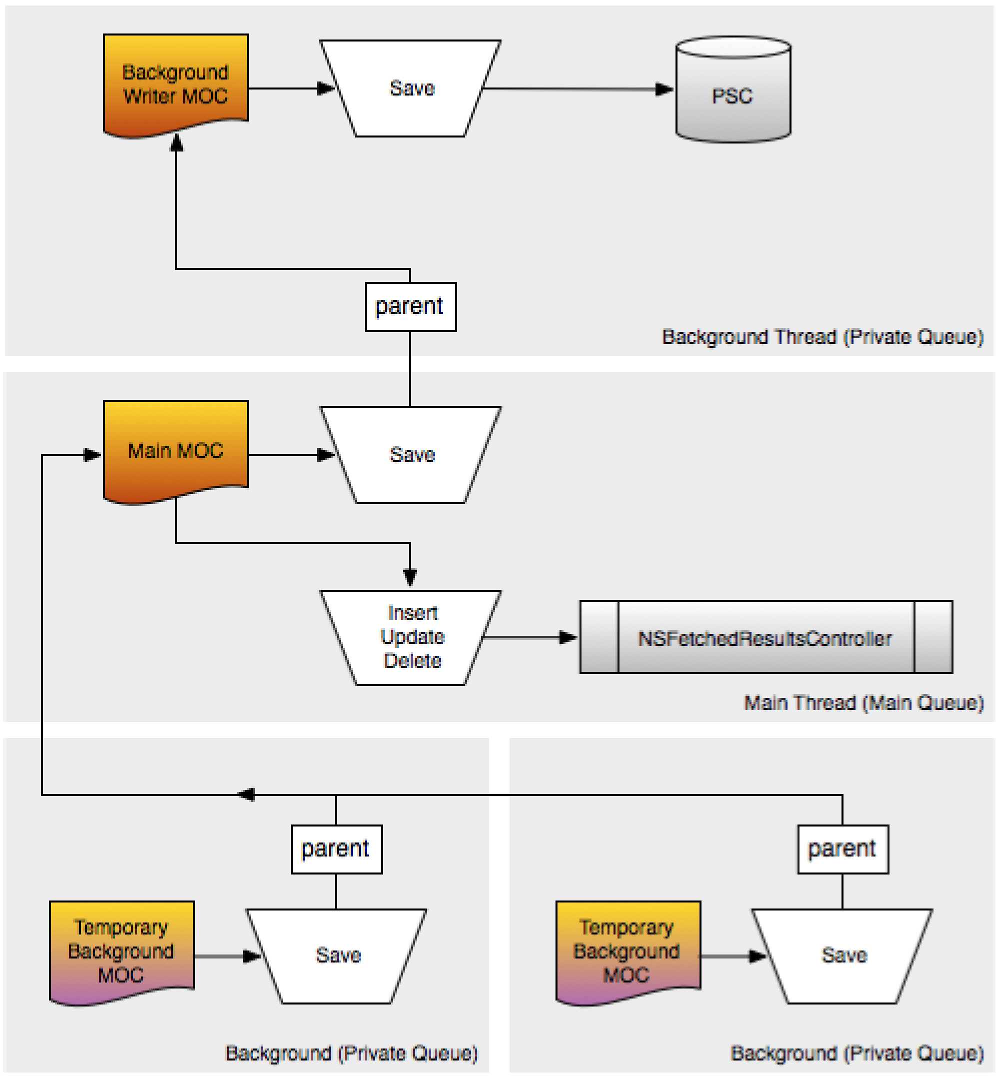

Figure 5.1: CoreData context architecture. Taken from [26]

When using a CoreData model, it is a good idea to create different `NSManaged-ObjectContexts` in order to have contexts using background threads and one context tied to the UI code to provide the views with the database objects. If we send all the saves to the background queue, the risk of blocking the Main Thread is reduced. Our proposed CoreData architecture [26] has one *background* context attached to the persistent store, another one attached to the Main Thread to use it when providing data to our views and several background contexts to perform saves.

The context attached to the persistent store saves all the data on disk, but on a background thread, so the UI is never blocked. Then, all the save contexts that we can create use a background thread and trigger the Main Thread, UI-context when they have saved any info. This architecture is often known as *child-context*, because there is a parent-child relationship between all the contexts 5.1.

Model design

In our model we have three entities. One is the poster category, to separate the different kind of posters that we can choose: movie posters, art, and so on. Then, we have two entities for the posters. One is an *abstract* entity that the main Poster entity inherits from.

This is a very common way to build models in CoreData. Entity inheritance enables developers to perform changes easily in the data model. Every time we have to ship a new version of the application, if the model has changed we need to perform a migration. It is quite common to change the model when new features are added, but migrations are difficult to implement if there are deep changes. One way to make migrations easier is to use entity inheritance. If somehow in the near future we need to add different posters, we can inherit from the `AbstractPoster` entity without touching the `Poster` one.

With this model design we can take advantage of *Lightweight migrations* in CoreData. Lightweight migrations are automatic migrations performed by the framework. Being able to maintain our model versioning using this kind of migrations is another reason of why we are using entity inheritance. Otherwise, we would have to perform manual migrations, which are slower to compute and error prone.

5.2 Augmented reality performance

During the development of Ponster, both OpenCV and Vuforia have been tested. Due to the low performance of OpenCV on the mobile device, Vuforia has been the chosen library to bring the augmented reality to our app. In the performance chart 5.2 we can find the average time to analyze each frame with the different technologies tested.

All of these tests have been executed with the iPhone 5 mobile device using iOS 8. The selected pattern to test the matching is the famous image of Lena Söderberg, which is widely used when testing computer vision algorithms. The test consists on running the algorithm against the Lena image while moving and rotating the camera for a minute.

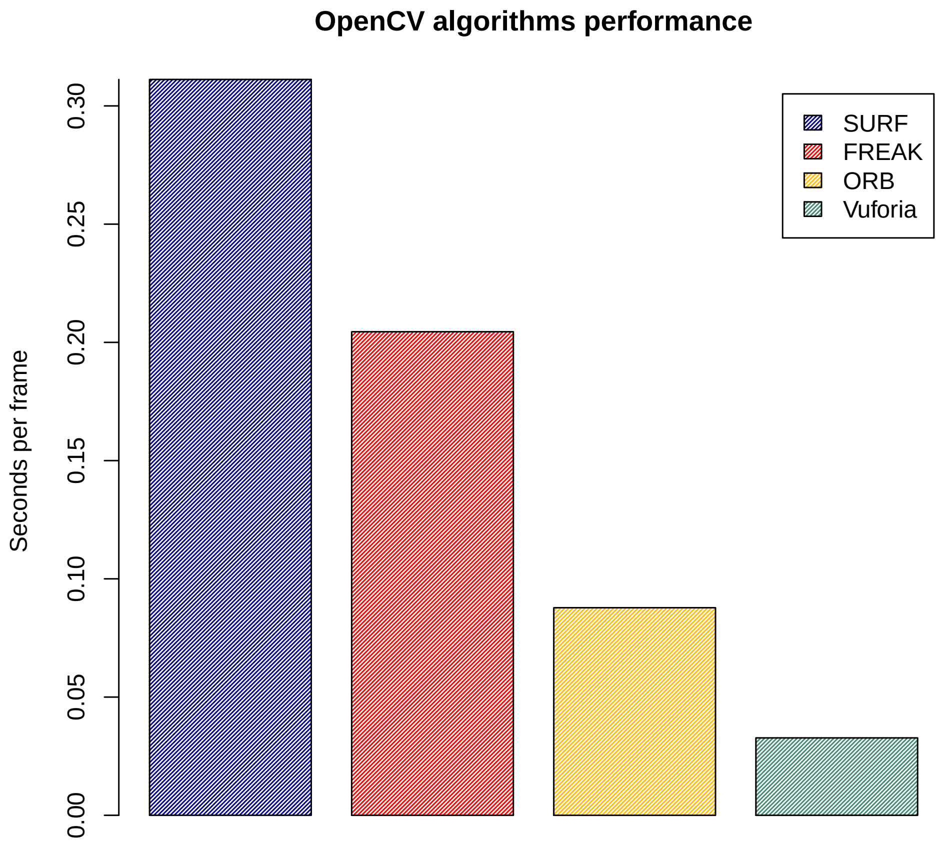

Figure 5.2: Average time spent processing each frame with the different algorithms. The best technique evaluated, Vuforia, is 10 times faster than SURF, which has the poorest performance of all the tracking algorithms tested.

As we can see, SURF is, by far, the most expensive technique tested on the device. This is the expected behavior, as it's stated in the cited articles in section 3. FREAK uses SURF to compute the keypoints, so the performance boost happens on the descriptor calculation only. Between all the OpenCV algorithms tested, ORB is the one that performs better while providing almost the same robustness as the other two algorithms tested. However, the only technique that led to a real-time performance has been the Vuforia SDK. The average frames per second of the processing with Vuforia is between 29 and 30 FPS, while maintaining the best robustness and tracking of all the above algorithms tested. This is due to the natural feature approximation of Vuforia 3.5.

The reason behind testing only SURF, FREAK and ORB under OpenCV is

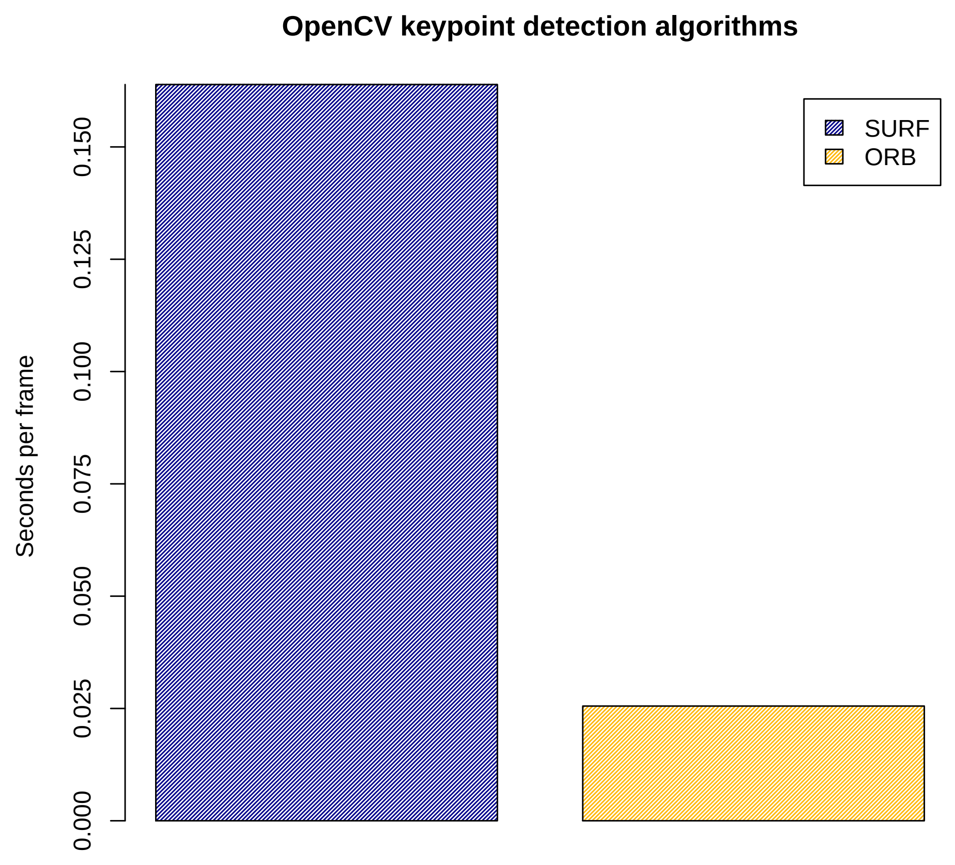

Figure 5.3: Mean time spent for detecting each frame keypoints with SURF and ORB.

that they represent different approaches to the same problem of object tracking. Analyzing each part of the feature detection process can bring some light to know which step is the most expensive in terms of computational cost. In the figure 5.3 we can see the average time spent in each frame when computing the keypoints with SURF and ORB detectors. FREAK has been tested with SURF keypoint detection, so we can clearly see that the descriptor extraction by FREAK is much more efficient than the SURF extraction.

The number of keypoints detected between SURF and ORB is also quite different. In the figure 5.4 we can see the average number of keypoint detected for each frame. It is interesting to see that while ORB detects around 60% of the keypoints detected by SURF, it performs almost as good as SURF in terms of robustness when detecting the selected pattern. However, while executing

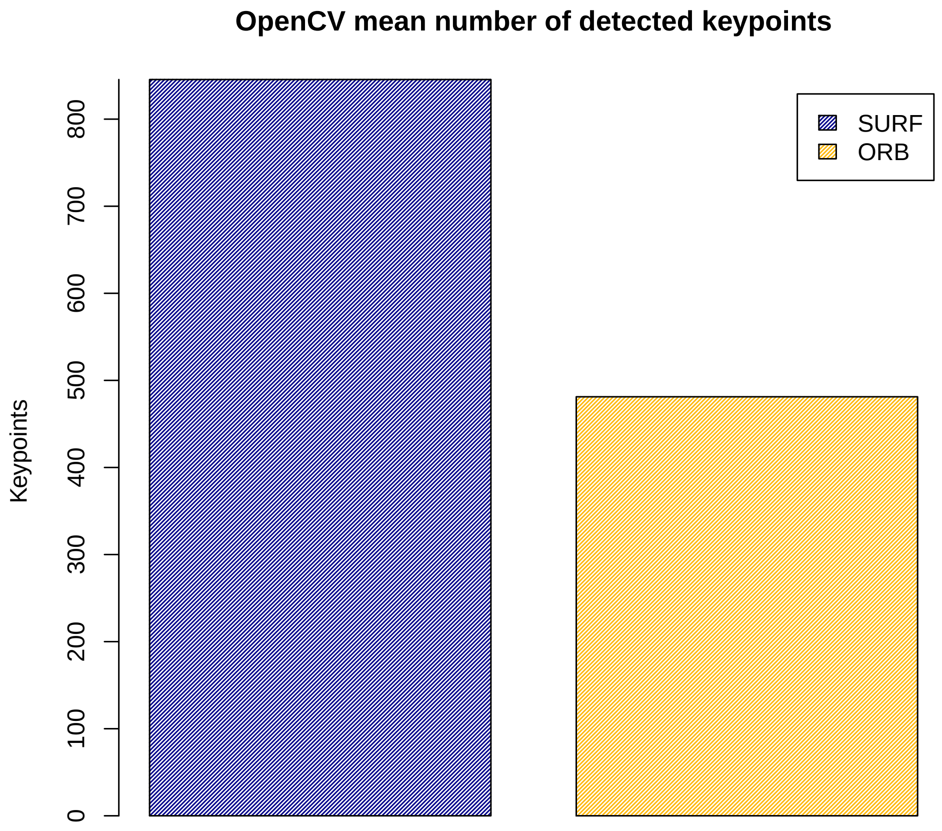

Figure 5.4: Average number of keypoints detected using SURF and ORB keypoint detection algorithms.

both algorithms in the app, the detection of the pattern is less prone to error with blurred or translated image using the SURF keypoint and descriptor combination.

In the figure 5.5 we can see the difference of performance between the descriptor extractors. The difference of performance between SURF keypoint detector and descriptor extractor and SURF keypoint detector and FREAK descriptor extractor can be explained because FREAK takes much less time to compute the descriptors than SURF. This is an expected behavior, as it can be seen in Table 1 of the FREAK paper [12], where the computation time for a descriptor using SIFT, SURF, BRISK and FREAK is compared. In that comparison, FREAK is 77 times faster than SURF to compute descriptors.

Finally, when we have extracted the descriptors of the image from the cam-

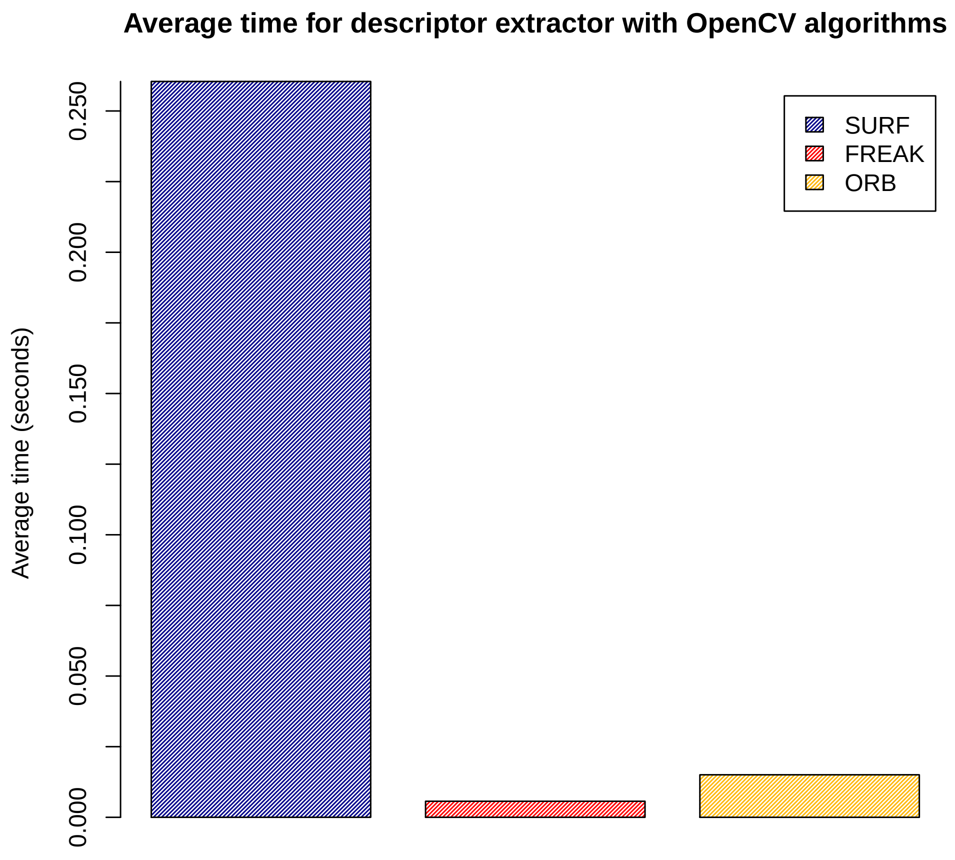

Figure 5.5: Average time to extract descriptors with SURF, FREAK and ORB.

era, we have to match them with the descriptors obtained from the pattern image. Once the match is performed, we have to calculate the homography to locate the object in the image. Then, we can get the corners of the pattern and calculate the perspective transform in order to draw the poster above the detected image.

In the figure 5.6 we can see that the SURF and Flann-based matcher combination is the one that produces more matches, but it's also the more expensive in terms of computational cost. SURF keypoint detector with FREAK descriptors perform faster than the SURF approximation, but it also has lower precision when detecting the pattern. Finally, ORB and Flann with Locality-Sensitive Hashing parameters for the matching is the one that outperforms the previous techniques and delivers an average number of matches similar to SURF.

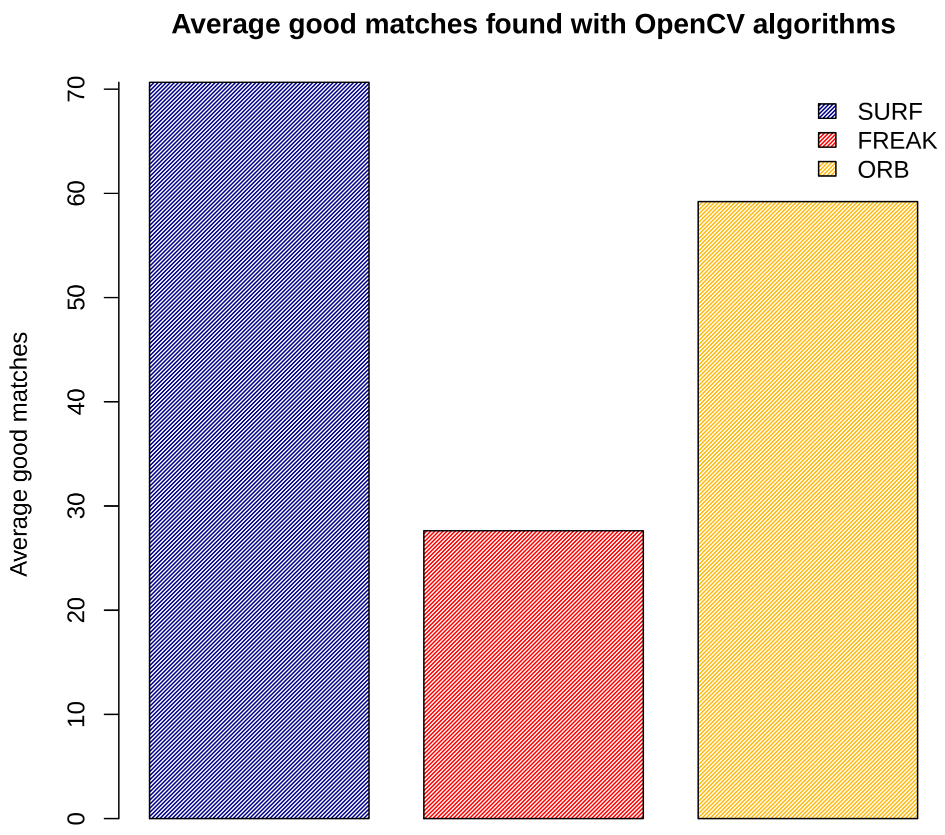

Figure 5.6: Average number of good matches detected using the different algorithm and matcher combinations.

Performance comparison between algorithms					
Algorithms	Avg. Frame (s)	Avg. Keypoint (s)	Avg. Descriptor (s)	Avg. number keypoints	Avg. good matches
SURF	0.3111817	0.1638936	0.260446452	845.4911	70.65957
FREAK	0.2044958	—	0.005685782	—	27.62121
ORB	0.08778909	0.02553993	0.015048248	481.1667	59.21101
Vuforia	0.03273424	—	—	—	—

Table 5.1: Data used to build the performance figures.

As we have stated before, Vuforia is a propietary SDK based on natural feature detection, and we do not have access to how it detects the pattern. This is the reason why it is only shown in the figure 5.2, because we can estimate the time spent processing for each frame delivered by the camera.

5.3 Features

The Ponster app has three main features: list posters, try how they look wherever the user wants and capture an screen-shot of the poster in the scene.

List posters

The main screen of Ponster shows a complete list of the images available to test. In the figure 4.1 we can see how this screen is. Each of the images represented come from a `Poster` entity from the database. This posters are queried immediately using a special class called `NSFetchedResultsController`.

This screen is ready for reading data from an API, because the way it has been implemented enables to parse a JSON into a `Poster` entity and automatically update the collection view of all the posters.

The user can scroll the screen to see all the posters and when tapping on one, a new view is presented with more information about the poster and the ability to test the augmented reality using both the OpenCV approximation and the Vuforia SDK, as it can be seen in figure 5.7.

Augmented reality

The main feature of Ponster is to show the posters where the user wants to try them, using augmented reality techniques. Tapping on the **Try Me** button opens the Vuforia window with the camera output. Once there, we present two buttons, one for the Screen-shot 5.3 feature and another one for the selection of the pattern to track.

When we tap the camera button, we capture an image from the video output and use it as our pattern to track. Then, every time the Vuforia SDK detects the pattern, the selected poster is presented. The extended tracking is also enabled, so the user can still see the poster even if the pattern is not anymore present in the video output.

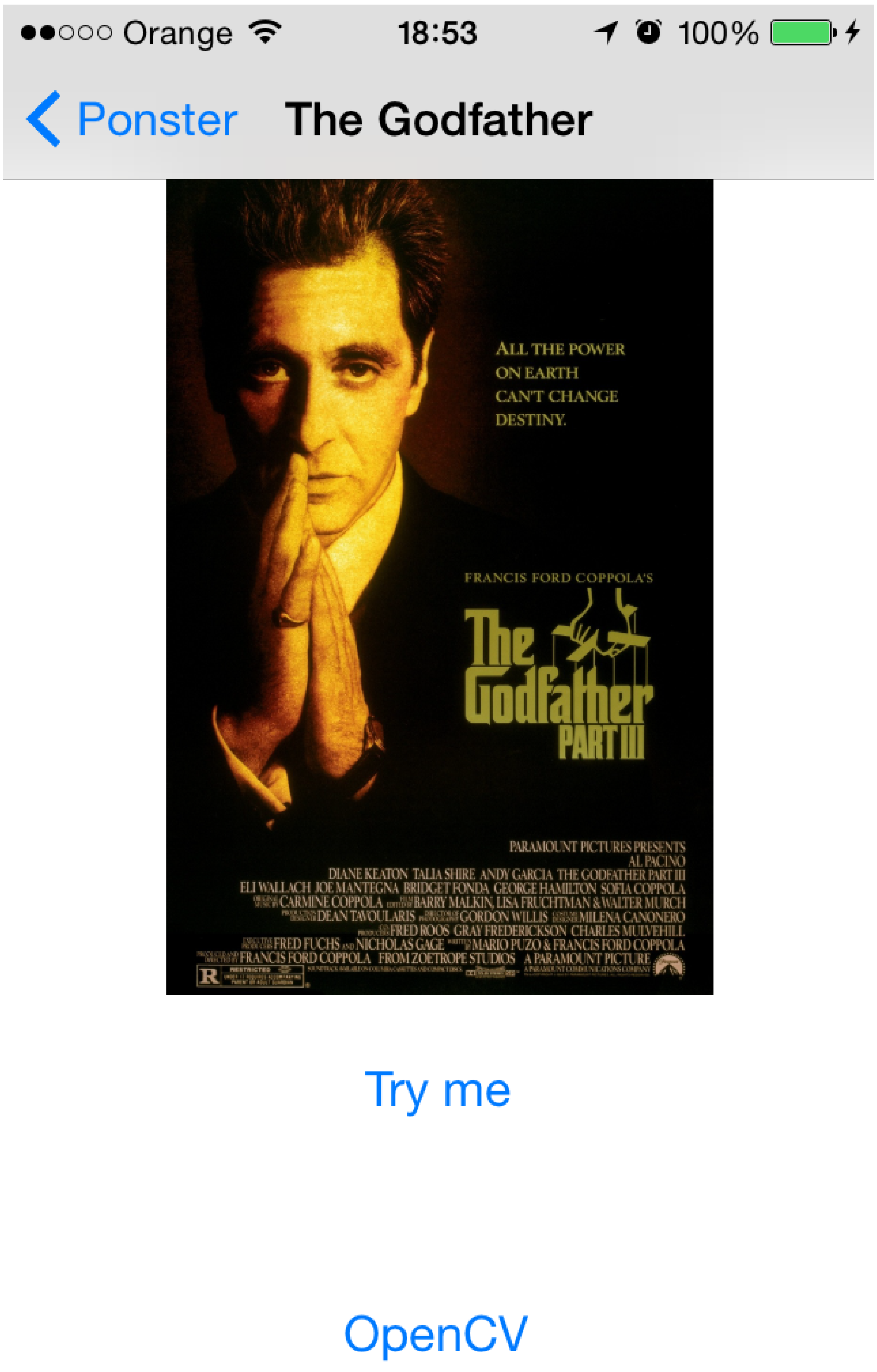

Figure 5.7: The poster view of Ponster app.

For testing purposes, we also bring the opportunity to test the augmented reality using OpenCV algorithms. This feature has been used to run the speed and robustness tests presented in the performance section 5.2.

Screen-shot

The screen-shot feature enables the user to take a picture of what's seen on the video output. This is very useful when it is combined with the augmented reality tracking, and allows the user to keep a still picture in the camera roll with the poster located in the scene. With this, the tested posters in the scene can be shared.

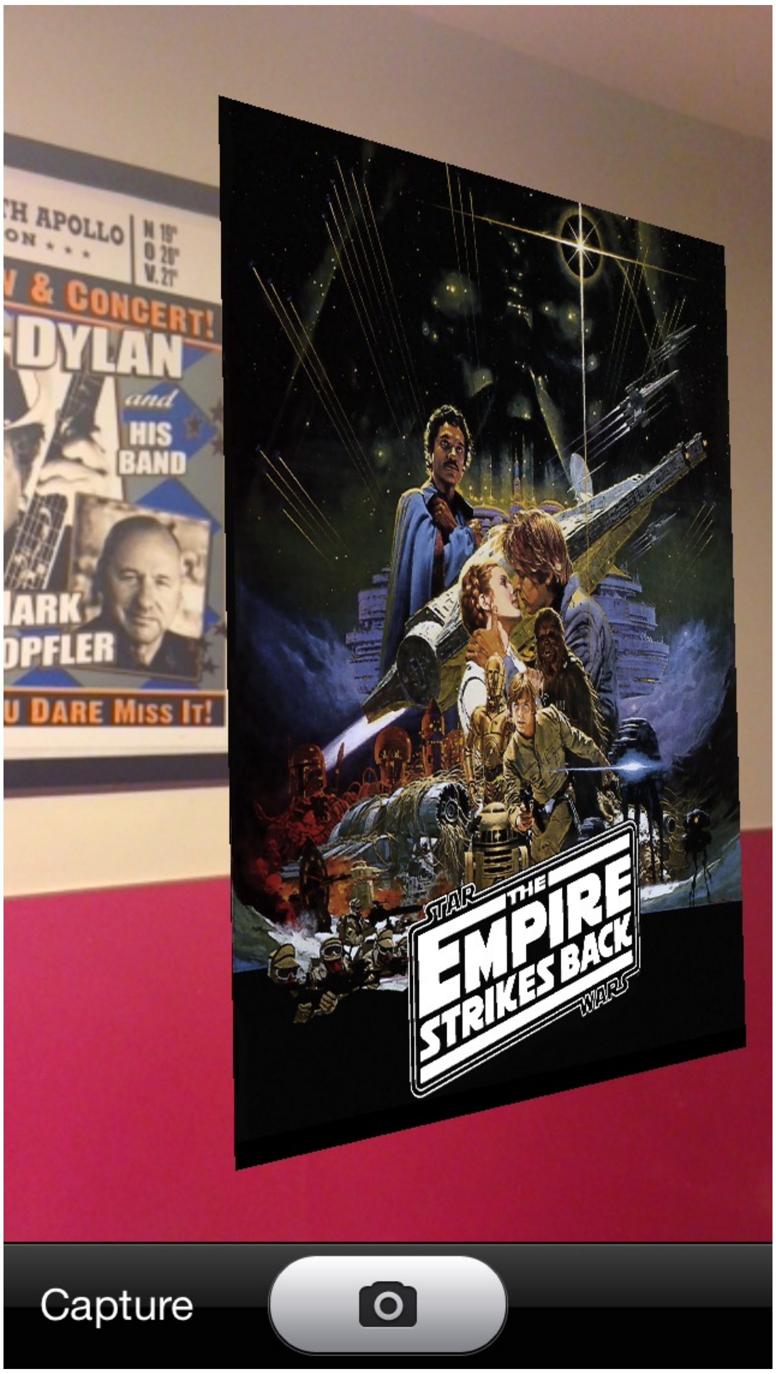

Figure 5.8: Augmented reality using a frame as our marker to introduce the poster in the scene.

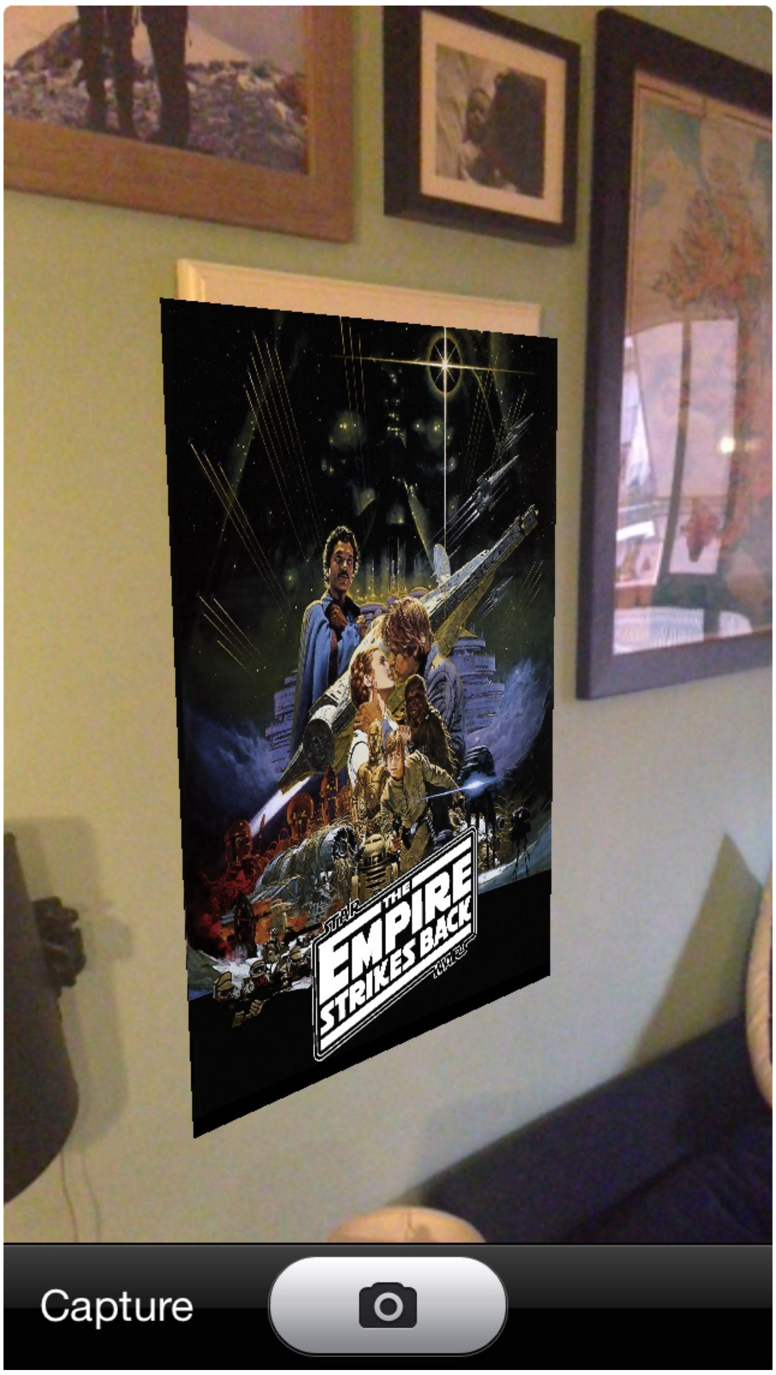

Figure 5.9: Another example of augmented reality using another pattern to track. In this case, defining a better located pattern results on having more realistic results. Note that the pattern selected is the white frame at the size of the camera viewport, so thanks to Vuforia's robustness we are still able to track the object despite the scale and perspective changes.

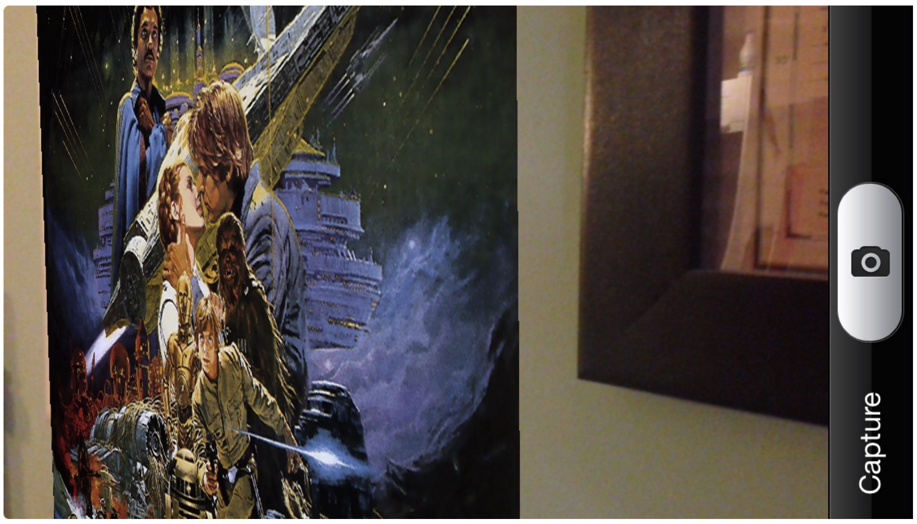

Figure 5.10: Thanks to the use of the gyroscope and accelerometer provided by CoreMotion framework, Vuforia manages to track correctly the pattern even if it is not entirely present in the scene.

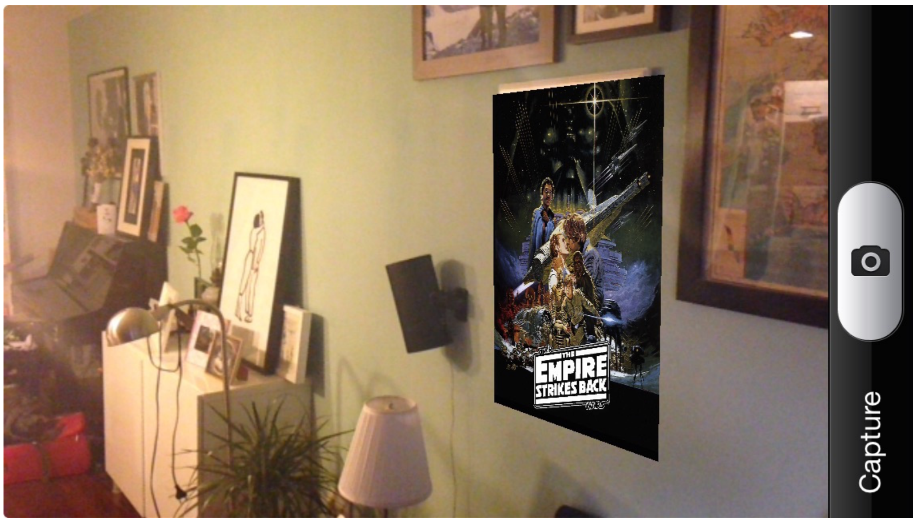

Figure 5.11: Despite the small size of the marker defined, we are still capable of detecting it and presenting the object correctly in the scene.

CHAPTER 6

TRACKING

In this chapter we are going to review our basic object tracking algorithm with OpenCV, and we will also explain where the Vuforia SDK gets the detected object for each frame and draws the poster.

6.1 OpenCV algorithm

The OpenCV based tracking algorithm consists on two parts, the video source and the tracking algorithm. With the video source we can get the frames from the camera with the selected quality, and with the tracking algorithm we use that frames to find the chosen pattern.

6.1.1 Video feed

To receive the images from the camera, we have created an Objective-C++ class called `PNSImageCapture`. In this class, we use `AVCaptureVideo` to get the video frames, and then we send each frame as a parameter to the `PNSImageCapture-Delegate`. Our class just uses `AVCaptureVideo` API to instantiate the camera object, so the main functionality comes from the delegate:

`PNSImageCaptureDelegate.h`

```
#import <AVFoundation/AVFoundation.h>
#import <VideoFrame.h>
```

```

#pragma mark - PNSImageCaptureProtocol

@protocol PNSImageCaptureDelegate <NSObject>

@required
- (void)frameReady:(VideoFrame)frame;

@end

#pragma mark - PNSImageCapture interface

@interface PNSImageCapture : NSObject
<AVCaptureVideoDataOutputSampleBufferDelegate>

@property (strong, nonatomic) AVCaptureSession *captureSession;
@property (weak, nonatomic) id<PNSImageCaptureDelegate> delegate;

- (BOOL)startWithDevicePosition:(AVCaptureDevicePosition)devicePosition;

@end
```

The required delegate `frameReady:` is used to deliver the `VideoFrame` to whatever class that needs to get the video image. This class must implement `AVCaptureVideoDataOutputSampleBufferDelegate` in order to get the data from the camera. The delegate method implementation of this class is presented next.

`AVCaptureVideoDataOutputSampleBufferDelegate` method

```
- (void)captureOutput:(AVCaptureOutput *)captureOutput
didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer
       fromConnection:(AVCaptureConnection *)connection
{
    CVImageBufferRef imageBuffer =
    CMSampleBufferGetImageBuffer(sampleBuffer);

    CVPixelBufferLockBaseAddress(imageBuffer,
    kCVPixelBufferLock_ReadOnly);

    // Dispatch VideoFrame to VideoSource delegate
    if ([self.delegate respondsToSelector:@selector(frameReady:)]) {
        // Construct VideoFrame struct
        uint8_t *baseAddress =
        (uint8_t*)CVPixelBufferGetBaseAddress(imageBuffer);
        size_t width = CVPixelBufferGetWidth(imageBuffer);
        size_t height = CVPixelBufferGetHeight(imageBuffer);
        size_t stride = CVPixelBufferGetBytesPerRow(imageBuffer);
        VideoFrame frame = {width, height, stride, baseAddress};
        [self.delegate frameReady:frame];
    }

    CVPixelBufferUnlockBaseAddress(imageBuffer, 0);
}
```

6.1.2 Tracking algorithm

Our tracking algorithm is presented in the `PatternDetector.cpp` file and `PatternDetector.h` header file. In this class, the first thing we do in the constructor is pass a pattern image to track and the poster image to draw on it.

Here is a shortened version of the `PatternDetector` class constructor.

```
PatternDetector::PatternDetector(const Mat& patternImage, const Mat&
posterImage)
{
    m_scaleFactor = kDefaultScaleFactor;

    // Save the pattern image
    m_patternImage = patternImage;

    // Create a resized grayscale version of the pattern image
    resize(posterImage, m_resizedPosterImage,
    Size(posterImage.cols/m_scaleFactor, posterImage.rows/m_scaleFactor));

    Mat patternImageGray;
    switch (patternImage.channels())
    {
        case 4: /* 3 color channels + 1 alpha */
```

```
            cvtColor(m_patternImage, m_patternImageGray, CV_RGBA2GRAY);
            break;
        case 3: /* 3 color channels */
            cvtColor(m_patternImage, m_patternImageGray, CV_RGB2GRAY);
            break;
        case 1: /* 1 color channel, grayscale */
            m_patternImageGray = m_patternImage;
            break;
    }

    // SURF
    m_detector = SurfFeatureDetector(4);
    m_detector.detect(m_patternImageGray, m_posterKeypoints);

    // SURF descriptors
    m_extractor = SurfDescriptorExtractor();
    m_extractor.compute(m_patternImageGray, m_posterKeypoints,
    m_posterDescriptors);
    if (m_posterDescriptors.empty()) {
        printf(``WARNING empty descriptors \n'');
    }

    // FlannBasedMatcher matcher
    m_matcher = FlannBasedMatcher();
}
```

Then, every time the video frame delegate send us a new frame from the camera, the tracking algorithm is executed. Here we have a basic example of how this algorithm works.

```
Mat PatternDetector::fastDetection(VideoFrame frame)
{
    // Build the grayscale query image from the camera data
    Mat queryImageGray, queryImageGrayResized, outputImage;
    Mat queryImage = Mat(frame.width, frame.height, CV_8UC4,
    frame.data, frame.bytesPerRow);
    cvtColor(queryImage, queryImageGray, CV_RGBA2GRAY);

    // Apply filter to reduce noise
    GaussianBlur(queryImageGrayResized, queryImageGrayResized,
    Size(7, 7), 2, 2 );
    cvtColor(queryImage, outputImage, CV_BGR2BGRA);

    // Detect scene keypoints
    vector<KeyPoint> sceneKeypoints;
    m_detector.detect(queryImageGrayResized, sceneKeypoints);
    if (sceneKeypoints.size() <= 1) {
        return outputImage;
    }

    // Calculate scene descriptors
    Mat sceneDescriptors;
    m_extractor.compute(queryImageGrayResized, sceneKeypoints,
    sceneDescriptors);
    if (sceneDescriptors.empty()) {
        return outputImage;
    }

    // Match descriptors
    t = (double)getTickCount();
    vector<DMatch> matches;
    m_matcher.match(m_posterDescriptors, sceneDescriptors, matches);

    // Compute good matches
    for (int i = 0; i < m_posterDescriptors.rows; i++) {
        double dist = matches[i].distance;
        if (dist < min_dist) min_dist = dist;
        if (dist > max_dist) max_dist = dist;
    }

    // Draw only ``good'' matches
    vector<DMatch> good_matches;
    for (int i = 0; i < m_posterDescriptors.rows; i++) {
        if (matches[i].distance < 3*min_dist) {
            good_matches.push_back(matches[i]);
        }
    }
    if (good_matches.size() < 4) {
        return outputImage;
    }

    // Localize the object
    vector<Point2f> obj;
    vector<Point2f> scene;

    for (int i = 0; i < good_matches.size(); i++) {
        // Get the keypoints from the good matches
        obj.push_back(m_posterKeypoints[good_matches[i].queryIdx].pt);
        scene.push_back(sceneKeypoints[good_matches[i].trainIdx].pt);
    }

    Mat H = findHomography(obj, scene, CV_RANSAC);

    // Get the corners from the image
    vector<Point2f> obj_corners(4);
    obj_corners[0] = cvPoint(0, 0);
    obj_corners[1] = cvPoint(m_patternImageGray.cols, 0);
    obj_corners[2] = cvPoint(m_patternImageGray.cols,
    m_patternImageGray.rows);
    obj_corners[3] = cvPoint(0, m_patternImageGray.rows);
    vector<Point2f> scene_corners(4);

    perspectiveTransform(obj_corners, scene_corners, H);

    // Draw lines between the corners
    Scalar green = Scalar(0, 255, 0);
    line(outputImage, scene_corners[0], scene_corners[1], green, 4);
    line(outputImage, scene_corners[1], scene_corners[2], green, 4);
    line(outputImage, scene_corners[2], scene_corners[3], green, 4);
    line(outputImage, scene_corners[3], scene_corners[0], green, 4);

    circle(outputImage, scene_corners[0], 15, Scalar(255, 0, 0));
    circle(outputImage, scene_corners[1], 15, Scalar(0, 255, 0));
    circle(outputImage, scene_corners[2], 15, Scalar(0, 0, 255));
    circle(outputImage, scene_corners[3], 15, Scalar(255, 255, 0));

    return outputImage;
}
```

In this algorithm we perform exactly the same keypoint and descriptor calculation, and then we use this data with the descriptors obtained from the pattern image in the constructor to perform the match. In this example, a Flann-based matcher using a randomized KD-tree is used to perform the matching between the descriptors. It is worth to mention the first lines of the algorithm, where we get the `VideoFrame` and make transformations to its image, are very important to the final performance of the algorithm. Converting the image to grayscale and using a Gaussian blur filter to reduce it's noise helps us to compute faster the keypoints and to have more robust results.

Once we have computed the matches, we use homography to make a perspective transformation to get the four corners of the object in the scene. In this code example, we use the corners to draw a rectangle where the pattern is located and four circles to identify each of the pattern corners.

It is easy to calculate the performance results showed in section 5.2 with this algorithm. We only need to get the system time before and after performing each of the steps to see where the algorithm takes more computation time.

6.2 Vuforia SDK

With Vuforia, as we have said before, we do not have an exact representation of the tracking algorithm. Everything is done through the QCAR namespace, which stands for Qualcomm Augmented Reality. Our background video view implements `UIGLViewProtocol`, which is an iOS-only protocol that allows Vuforia to render the current frame. The view responsible of rendering the frame must implement the method `renderFrameQCAR`.

Inside our implementation of `renderFrameQCAR` in the view that draws the video output, we query the QCAR object to see if there are tracking results:

```
QCAR::State state = QCAR::Renderer::getInstance().begin();
for (int i = 0; i < state.getNumTrackableResults(); ++i) {
    // Get the trackable
    const QCAR::TrackableResult* result = state.getTrackableResult(i);
    QCAR::Matrix44F modelViewMatrix =
        QCAR::Tool::convertPose2GLMatrix(result->getPose());
    // Drawing code
    // ...
}
```

With the data inside the modelViewMatrix variable, we have all that is necessary to know where to draw the poster object. All the drawing is done with OpenGL ES, so first we load the textures and then we draw them:

```
// Get the projection of the tracked object
glUniformMatrix4fv(mvpMatrixHandle, 1, GL_FALSE,
    (const GLfloat*)&modelViewProjection.data[0]);
glUniform1i(texSampler2DHandle, 0);
// Draw the poster into the scene
glDrawElements(GL_TRIANGLES, NUM_QUAD_INDEX,
    GL_UNSIGNED_SHORT, (const GLvoid*)quadIndices);
```

Our object is just a rectangle defined in a C++ header, but with Vuforia we can set any object we want by just creating another texture. For instance, in the SDK examples a teapot is used to demonstrate its features.

CHAPTER 7

COMMERCIAL APPLICATIONS AND FUTURE WORK

Building Ponster has been an opportunity to research how the computer vision field is evolving and how other companies and institutions are using augmented reality to build products and new techniques.

In this chapter we are going to cite several applications that have served as an inspiration to research about augmented reality and to build Ponster app. Also, we are going to describe the future work that can be applied to enhance the app functionality.

7.1 Commercial applications

Speaking about mobile environments only, there are great commercial applications that use augmented reality available to the major platforms, iOS and Android. In this section we are going to review a few applications that have served as an inspiration to Ponster.

SkyGuide

SkyGuide is an augmented reality app that enables the user to identify and get information about the constellations that are present in the sky. This app uses the user location and the gyroscope to get both where the user is located and how he is moving the device. With this information, the application presents a view that resembles to the night sky with all the information about the visible constellations, planets and stars.

IKEA

The IKEA application also uses Vuforia SDK to bring quality augmented reality experience to the mobile phone. Using the IKEA catalog as the object to track, users can throw the catalog to the floor and test how any furniture would look. The size of the furniture is realistic because they know the exact sizes of the catalog, so they can easily determine at what size the objects must be rendered.

Sony TV size

This application enables users to test if a Sony TV is going to fit well in their living room. They use a custom pattern that the user has to put where they want to test the virtual television. Again, this application manages to render a real size TV by using a know-size pattern.

LEGO Connect

LEGO Connect also uses Vuforia to virtually enhance their product catalog. With a LEGO product catalog, the user can identify any of the products displayed and draw a virtual 3D model of any of their toys. These models can also be animated.

7.2 Future work

There are several improvements that can be done to Ponster to enhance the functionality and user experience of the app. After reviewing the applications mentioned in the previous section 7.1, it is clear that augmented reality-based applications can improve customer experience when browsing products.

In this section we are going to review some of the different improvements that can be implemented in Ponster app.

7.2.1 API integration

Ponster has been developed to be ready to integrate to an external API. From the model to the design of the controllers or the tracking algorithm, everything is independent in terms of where the poster data comes from.

One of the most important improvements that could be made to Ponster is to integrate it with an REST API to get the poster data. Actually the poster meta-data is attached when the application starts, and the poster images are embedded inside the application sandbox. Integrating an API would be easy with the actual state of the application. We could send a GET request to the API to download the poster meta-data when the application starts. This meta-data should have the poster information and the URL of the images, so when the poster gets in the screen we download its image.

This API should have three main features: provide a registration process to the user, enable the user to browse different types of posters or wall-related products and providing a payment process to purchase posters.

7.2.2 Purchasing posters

Ponster has been built with the goal in mind of allowing users to browse, try and buy posters directly from the application. The API integration can bring the ability to get the products and the purchase process.

With the latest iOS API, a new way to make payments has been introduced to the recent iPhone devices. This new feature is called Apple Pay[27] and uses the TouchID fingerprint sensor and the credit card associated with the Apple ID account to process the payments.

7.2.3 Custom images

The current version of Ponster supports png images as the poster images only. One improvement that could be made to the application is to enable users to choose a poster image from its camera roll.

In order to implement this feature, there should be changes to the navigation and menus of the app and to the texture load of the augmented reality algorithm. This could be integrated with an option in the API to enable the user to print their custom image instead of buying an existing one.

7.2.4 Poster image sizes

When trying a poster, it is very important to represent it with the real size of it. This version of Ponster do not draws the posters in real size because it is dependent of the size of the marker chosen by the user, so if a small marker is chosen, a small version of the poster will be rendered.

One fundamental feature to improve Ponster could be enabling the user to print a custom marker —like the catalog of IKEA or the pattern used in the Sony app— with fixed size in order to render the posters with their real size. This could be used to improve the buying experience.

Also, we could use this to deliver several sizes to each poster. With this, the user could try the size that fits better to their needs and also try it knowing that the size of the poster that is being drawn with is actually the real size.

CHAPTER 8

CONCLUSIONS

Augmented Reality has become a very popular topic in the last years. In the field of computer vision, many object detection algorithms have been introduced recently. The motivation behind this new techniques is to perform faster than its predecessors while maintaining robustness in the detection. Due to the increasing power of the mobile smart phones, bringing image processing and computer vision algorithms to the mobile world is now possible.

However, providing augmented reality experiences only with feature detection algorithms is not the way to go on mobile devices. Nowadays, all of the modern mobile smart phones have integrated GPUs and different sensors such as gyroscopes, GPS, accelerometers and others that can be used in combination with feature detection to bring desktop-class augmented reality experiences to the mobile world. This is where Qualcomm's Vuforia SDK is on, combining the optimization of FastCV algorithms for ARM microprocessors with all the sensors that the mobile smart phone have.

In this work, two different approaches have been introduced —OpenCV and Vuforia SDK— in order to build an app that enables the user to try posters in their walls. While OpenCV has been proven to be useful to detect features in an image, the natural feature detection approach used by Vuforia has much more better results in terms of performance and detection quality.

The availability of OpenCV, Vuforia SDK and FastCV in mobile devices enables developers to build a plethora of computer vision based applications that can be powerful enough to process images in real time, apply filters or create augmented reality experiences. Understanding how these technologies work is key to develop any augmented reality application on a mobile device. As we have seen, this augmented reality applications can be used for marketing purposes, education and better shopping experiences.

BIBLIOGRAPHY

[1] OpenCV, “Features2d + homography to find a known object,” http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html. (document), 2.1

[2] D. Trumbull, “Showscan patent,” http://douglastrumbull.com/patents. 3

[3] O. Docs, “Template matching,” http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. 3.2

[4] T. Mahalakshmi, R. Muthaiah, and P. Swaminathan, “Review article: An overview of template matching technique in image processing,” *Research Journal of Applied Sciences, Engineering and Technology*, 2012, http://maxwellsci.com/print/rjaset/v4-5469-5473.pdf. 3.2

[5] J. Montoya-Zegarra, N. Leite, and R. da S.Torres, “Rotation-invariant and scale-invariant steerable pyramid decomposition for texture image retrieval,” in *Computer Graphics and Image Processing, 2007. SIBGRAPI 2007. XX Brazilian Symposium on*, Oct 2007, pp. 121–128. 3.2

[6] O. Docs, “Image pyramids,” http://docs.opencv.org/doc/tutorials/imgproc/pyramids/pyramids.html. 3.2

[7] C. V. Labs, “Image pyramids,” http://compvisionlab.wordpress.com/2013/04/. (document), 3.2

[8] L. W. Kheng, “Feature detection and matching,” CS4243 Computer Vision and Pattern Recognition, http://www.comp.nus.edu.sg/~cs4243/lecture/feature.pdf. 3.3

[9] G.-S. Xia, J. Delon, and Y. Gousseau, “Accurate junction detection and characterization in natural images,” *International Journal of Computer Vision*, vol. 106, no. 1, pp. 31–56, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11263-013-0640-1 (document), 3.3

[10] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "Orb: An efficient alternative to sift or surf," in *Computer Vision (ICCV), 2011 IEEE International Conference on*, Nov 2011, pp. 2564–2571. 3.3, 3.3.4

[11] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "Speeded-up robust features (surf)," *Comput. Vis. Image Underst.*, vol. 110, no. 3, pp. 346–359, Jun. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cviu.2007.09.014 3.3.1

[12] R. Ortiz, "Freak: Fast retina keypoint," in *Proceedings of the 2012 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, ser. CVPR '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 510–517. [Online]. Available: http://dl.acm.org/citation.cfm?id=2354409.2354903 3.3.2, 5.2

[13] E. Khvedchenia, "A battle of three descriptors: Surf, freak and brisk," http://computer-vision-talks.com/articles/2012-08-18-a-battle-of-three-descriptors-surf-freak-and-brisk/. (document), 3.3.2, 3.5, 3.6

[14] E. Rosten and T. Drummond, "Machine learning for high-speed corner detection," in *Proceedings of the 9th European Conference on Computer Vision - Volume Part I*, ser. ECCV'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 430–443. [Online]. Available: http://dx.doi.org/10.1007/11744023_34 3.3.3

[15] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISSAPP (1)'09*, 2009, pp. 331–340. 3.4.2

[16] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, no. 1, pp. 117–122, Jan. 2008. [Online]. Available: http://doi.acm.org/10.1145/1327452.1327494 3.4.2

[17] U. Neumann and S. You, "Natural feature tracking for augmented reality," *IEEE Transactions on Multimedia*, vol. 1, no. 1, pp. 53–64, 1999. 3.5

[18] Qualcomm, "Vuforia," https://developer.vuforia.com. 4

[19] Apple, "Uikit," https://developer.apple.com/library/ios/documentation/uikit/reference/uikit_framework/_index.html. 4.1.1

[20] S. Mora, “Magicalrecord,” https://github.com/magicalpanda/MagicalRecord. 4.1.4

[21] CocoaPods, “Cocoapods,” http://cocoapods.org. 4.1.4

[22] OpenCV, “Opencv,” http://opencv.org/about.html. 4.2.1

[23] CocoaPods, “Opencv podspec,” http://github.com/CocoaPods/Specs/blob/master/Specs/OpenCV/2.4.9/OpenCV.podspec.json. 4.2.1

[24] Qualcomm, “Vuforia sdk architecture,” https://developer.vuforia.com/resources/dev-guide/vuforia-ar-architecture. 4.2.2

[25] ——, “Fastcv,” https://developer.qualcomm.com/mobile-development/add-advanced-features/computer-vision-fastcv. 4.2.2

[26] O. Drobnik, “Multi-context coredata,” http://www.cocoanetics.com/2012/07/multi-context-coredata/. (document), 5.1, 5.1.2

[27] Apple, “Getting started with apple pay,” https://developer.apple.com/apple-pay/Getting-Started-with-Apple-Pay.pdf. 7.2.2