Universitat de les Illes Balears

escola politècnica superior

Treball Final de Carrera

ENGINYERIA INFORMÀTICA

Ponster, tu app de realidad aumentada

Álvaro Medina Ballester

Tutor
Ramón Mas Sansó

Escola Politècnica Superior
Universitat de les Illes Balears
Palma, September 14, 2014

Contents

CHAPTER 1

ABSTRACT

CHAPTER 2

THANKS

Many thanks to everybody.

CHAPTER 3

INTRO

Augmented reality has become a very popular topic in the last five years. With the introduction of mobile smartphones, developers started to have the chance to develop applications using the powerful CPUs and GPUs of those devices.

CHAPTER 4

State of the art

The techinique 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 smartphones (citation needed), 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 past years.

Most of the augmented reality apps follow this behaviour:

- 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 them.

4.1 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 smartphone camera to the user location. Ponster searches a particular image inside the camera input in order to draw the poster image above. In computer vision, this 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 smartphone camera, taking account of scale, rotation and perspective differences, becomes an object tracking technique.

4.2 Template matching

Template matching[1] consists of finding areas of an image that are similar to a provided template image. We have to provide 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)[2]. Template matching is also called area-based approach.

OpenCV provides a method to perform template matching with several methods, such as `SQDIFF` or `CCORR`. 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 25/30fps-like detection. However, match template does not take account of scale, rotation and perspective invariance by itself. There are several approaches to bring invariation to match template. For instance, image pyramids are used to make match template scale and rotation invariant[3], but it is not part of the OpenCV match template function, although it provides some methods to implement image pyramids[4].

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 rotation and scale invariance and perspective warp are required features.

4.3 Feature detection

A feature-based approach can be presented as a three step method. First of all, we have to detect keypoints[5] (also called interest points) in the image. Usually, interest points are corners, blobs or T-junctions. 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 neighbourhoods 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 vectors computed from interest point neighbourhoods. 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 mantain robustness and reducing computation time. Robustness it's also very important, but less robust algorithms are also been developed in favour of reducing computation time. One good example is FAST[6] keypoint detector, which is not rotation invariant. Next, we are going to describe the feature-detection algorithms tested on Ponster: SURF, FREAK and ORB.

4.3.1 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[7]. It's 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 neighbourhood, 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) is not good enough to deliver a decent user experience, taking between 0.7 and 1 second to compute each image. Also, SURF is a patented algorithm and it's not intended to use it in commercial applications.

4.3.2 Features from Accelerated Segment Test - FAST

FAST is a keypoint detector based on corner detection. It was introduced by Edward Rosten and Tom Drummond[8] and it's primary purpose was to bring 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 contiguos 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.

4.3.3 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, 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 intensity centroid. This technique assumes that a corner's intensity is offset from it's center, and this vector may be used to impute an orientation[6]. 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. Only a 15 fps processing have been achieved, with slightly worst detection than with SURF.

4.4 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.

4.4.1 Brute-Force Matcher

Brute-Force Matcher, as it's name states, will compare each of the descriptors found in the images performing a linear search. Althought 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 quickly. In Ponster, Hamming distance has been tested for ORB and L2 normalization with SURF.

4.4.2 Fast Library for Approximate Nearest Neighbors - FLANN

Instead of performing a linear search for matching descriptors, we can use a nearest neighbour matching technique. The nearest neighbour 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[9] 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. Several KD-tree algorithm variants have been introduced, such as approximate nearest neighbour, in order to reduce the computational cost of KD-tree searches with large datasets.

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 mantain a 95% precision.

Locality-Sensitive Hashing

Locality-Sensitive Hashing is a matching algorithm to solve the nearest neighbour 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 neighbours of each point querying the other points in it's bucket[10].

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 explaination of LSH can be found in this[PDF] paper.

4.5 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 performance on mobile devices. Another technique has been proven to be successfull 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[11]. 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 the Neumann article[11], optical flux is presented to estimate the camera movement. In our mobile environment, we can also use the gyroscope. 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.

CHAPTER 5

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[12] 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.

5.1 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 other third party libraries.

5.1.1 UIKit

UIKit[13] 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 videogames do not make extensive use of the framework due to the specific user interfaces and game engines that most of them use.

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,

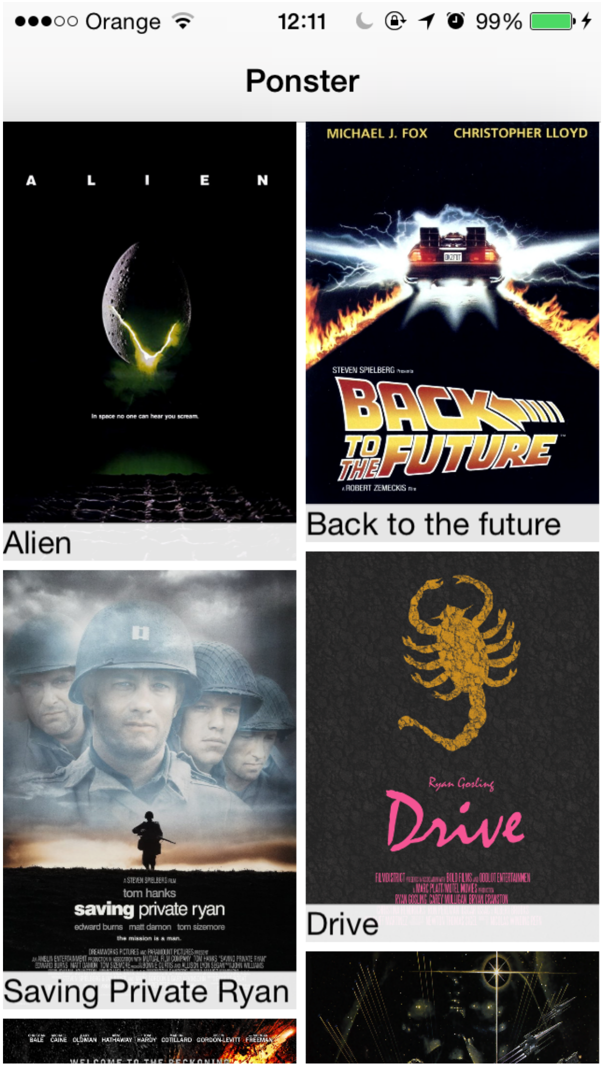

Figure 5.1: Main interface of Ponster demonstrating the use of navigation controllers and collection layouts.

making easy to identify to which framework belongs each class. For instance, all the UIKit classes have the `UI` class prefix. `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 5.1 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`.

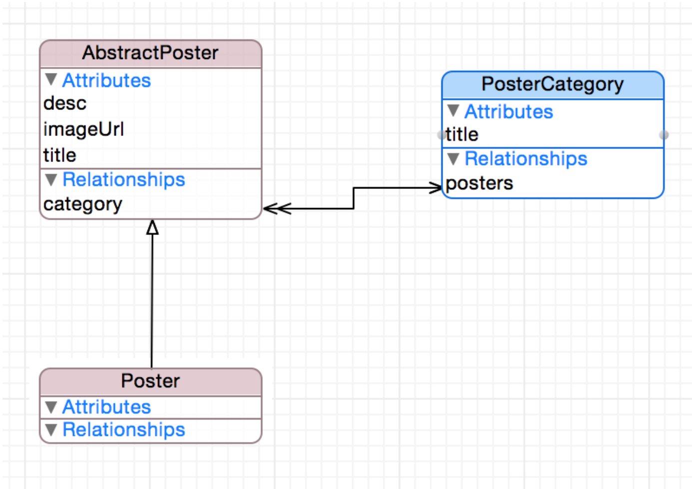

Figure 5.2: Basic data model of Ponster. The CoreData model editor enables us to design the model and then generate all the object classes. The double arrow represents a to-many relationship and the white arrow represents object inheritance.

5.1.2 CoreData, persistence layer for iOS applications

The main purpose of the research presented in the chapter 4 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 persistency.

In order to mantain 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 is to save the data in the preferences `.plist` file and then read all the values. Also, we could use `NSKeyedArchiver` to save data from our object model into the application sandbox. But, the best way to maintain data and to query it from 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 5.2) 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 or 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 in the following way:

```
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 **??**, our CoreData architecture is presented.

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 responsable of applying the changes to the SQLite backend. 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 **??**, 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 class in our model, with any predicate we want, thus performing an automatic refresh of our data every time some change in our model has been done in the monitorized classes. The fetched results controller is the class that comunicates our view controllers with the data model.

5.1.3 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.

5.1.4 Third-party libraries

Several third-party libraries are used in Ponster.

CocoaPods dependency manager

5.2 Computer vision

5.2.1 OpenCV

5.2.2 FastCV

5.2.3 Vuforia

CHAPTER 6

DEVELOPMENT

6.1 Application Architecture

6.2 Features

CHAPTER 7

CONCLUSIONS

Bibliography

[1] O. Docs, "Template matching," http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html. 4.2

[2] 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. 4.2

[3] 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. 4.2

[4] O. Docs, "Image pyramids," http://docs.opencv.org/doc/tutorials/imgproc/pyramids/pyramids.html. 4.2

[5] L. W. Kheng, "Feature detection and matching," CS4243 Computer Vision and Pattern Recognition, http://www.comp.nus.edu.sg/~cs4243/lecture/feature.pdf. 4.3

[6] 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. 4.3, 4.3.3

[7] 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 4.3.1

[8] 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 4.3.2

[9] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISSAPP (1)'09*, 2009, pp. 331–340. 4.4.2

[10] 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 4.4.2

[11] U. Neumann and S. You, "Natural feature tracking for augmented reality," *IEEE Transactions on Multimedia*, vol. 1, no. 1, pp. 53–64, 1999. 4.5

[12] Qualcomm, "Vuforia," https://developer.vuforia.com. 5

[13] Apple, "Uikit," https://developer.apple.com/library/ios/documentation/uikit/reference/uikit_framework/_index.html. 5.1.1