Fiducial Markers

Bruno Maia, João Costa, *Faculty of Engineering of the University of Porto*

Fiducial markers are usually divided in two distinct application: either to store information (such as QRCodes) or for perspective calculations (such as the AR-Toolkit markers). In this work we present an hybrid marker, that allows for both information storage and Augmented Reality applications.

Introduction

In a practical approach the two most common technologies used by Augmented reality (AR) for pinpoint where to deploy digital content in real-time video frames are georeferentiation and Computer Vision (CV) techniques. Although applied in some marketing and entertainment mobile applications, the use of CV techniques in real-world situations is difficult due to several constraints like scenario mutation over time, high visual noise or lack of unchanged anchor points.

Despite these difficulties, AR supported on CV is growing with the development of new frameworks and powerful new algorithms. This evolution, combined with sophisticated mobile devices, leads us to believe It's time to take Computer Vision and Augmented Reality to different industries , using its potential to solve a wider array of problems.

Goals

This projects goals can be summarized by the:

- Detecting of a singular fiducial marker.
- Tracking of a singular fiducial marker.
- Augmenting the fiducial marker with an image.
- Displaying the identifier encoded in the fiducial marker.

There are a few restrictions to be taken into consideration such as:

- The detection/tracking must be done with a mobile smartphone which presents reduced processing power.
- The fiducial marker size must be small.
- The fiducial marker should be recognized from at least 1 meter away.
- The detection/tracking must be implemented by using OpenCV4Android.
- The algorithm should perform in real time, i.e. framerate must be sustainable and should be optimized.

Solution

The solution is divided into 2 parts:

(i) The Encoder part, which handles the conversion of an unique id (in our implementation, simply a number between 0 and $2^{15} - 1$) into a unique fiducial marker with redundancy.

(ii) The Decoder part, which is an android application which scans and detects fiducial markers and augments them with an image while displaying the stored unique identifier.

Fiducial Marker

The fiducial marker opted for is a matrix-like representation where its tracking points are achieved by the means of four colored cells in the corners of the matrix (3 green and 1 red) as seen in figure 1.

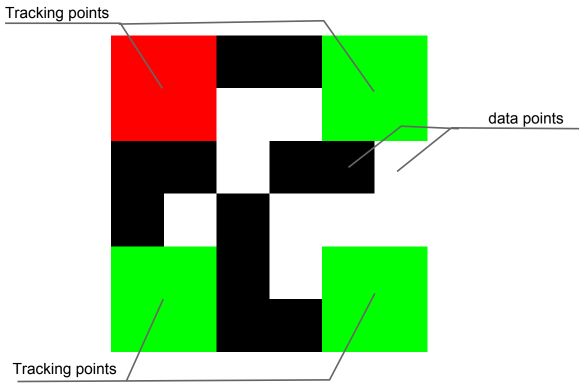

Figure 1: *Fiducial marker's types of cells*

The selected colors for the tracking points provide for optimal contrast and are bigger than data points to facilitate tracking.

It's composed by an 6x6 cell matrix which gives us 20 cells when we take the tracking cells. From these 20 cells, which we can equate to one bit each, weve spliitted 15 bits for the identificator ($2^{15} = 32768$ possibilities) and 5 bits for redundancy as seen in figure 2.

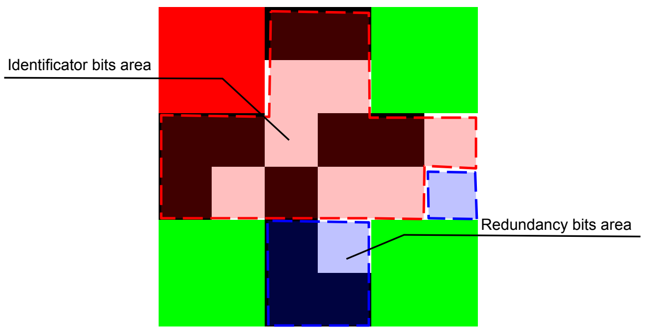

Figure 2: *Fiducial marker's areas of data*

The representation is read from left to right, and from top to bottom and is set in a way that the less significant bit is the first one as seen in figure 3.

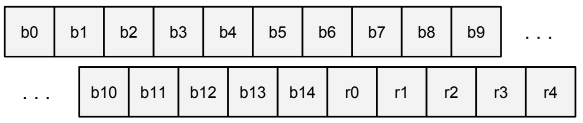

Figure 3: *Data representation format*

The 5 bits used for redundancy allow us to code our message using a Hamming(31,26) code, and therefore we are able to either correct 1-bit errors or detect 2-bit errors.

Marker Encoder

The encoder program was made in C++ and its usage is as such:

```
decoder <identifier> [output_file_name]
```

We pass it an unique number which will be stored within the fiducial mark and we can also specify the output filename, otherwise it will default to "default.bmp". The successful output artifact of the program is an bmp containing the fiducial mark encoded with the identifier.

Marker Decoder

The decoder was written in Java with OpenCV. It works in two phases: A detection phase and a tracking phase.

The detection phase is the most intensive, and therefore the decoder starts by downsampling the image by a factor of 4. It then converts it to HSV and detects the red and green regions via a simple inRange filter and a closing operation.

After that, the red and green blobs are extracted using a flood fill algorithm, and their centroids are roughly estimated based on their top-left corner and their area[1].

Next, we find the points that represent a marker according to algorithm 1. Figure 4 shows a visual interpretation of this method.

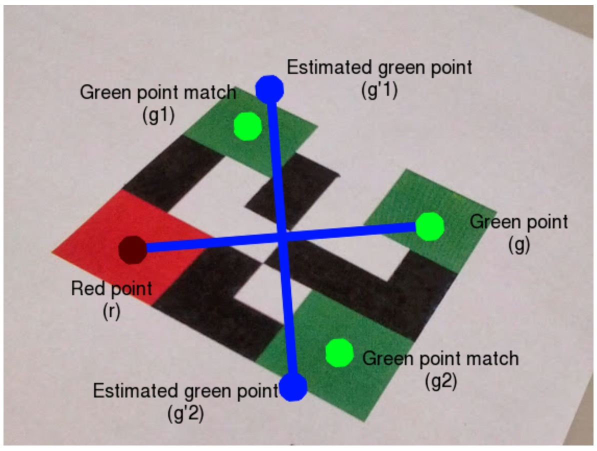

Figure 4: *Visualization of the detection algorithm*

We continue with the homography between the marker and the image is calculated (taking into

[1]Using a different blob detection algorithm, it might be possible to extract the correct centroids and have a faster running time, although there's no fast blob detector in OpenCV

```
Data: A set of red keypoints $R$ and a set of green keypoints $G$
Result: The coordinates of the center of the colored corners of a marker
forall the $r \in R$ do
    forall the $g \in G$ do
        Trace a line $l = \overline{rg}$ with center $c$;
        Trace a line $n \perp l$ that goes through c;
        Create two points $g'1$ and $g'2$ in $n$ with $\|c, g'1\| = \|c, g'2\| = \|r, c\|$ and $g'1 \neq g'2$;
        Pick two points, $g1, g2 \in G$ which are closest to $g'1$ and $g'2$ respectively;
        if $\|g1, g'1\|$ and $\|g2, g'2\|$ are small enough then
            return $[r, g1, g, g2]$;
        end
    end
end
```

Algorithm 1: Detection algorithm

account that the detected points are the corner centers and not the marker's corners). Once the detection is successful, the detector goes into the tracking phase.

The tracking phase takes into account the previously detected points to update the marker position. To do this, it fetches 4 small regions from the image, one for each of the previously detected points (the size of these regions depend on the size of the detected marker).

It's assumed that each region will only contain one blob, and therefore all the values that pass the inRange test are used to calculate the region centroid (this time correctly calculated).

This new centroids are interpolated with the previously detected coordinates using a constant weight, so that there's a smooth transition between frames. The homography is recalculated using this new points.

After each homography calculation, the inverse of the homography is used to read the marker's content.

Tests

A series of tests were made throughout and after the completion of the project. Among these, a set of approaches were found that weren't considered as performant as the final, proposed one.

The tests were performed with a Galaxy Nexus (Dual-core 1Ghz, 1Gb RAM) which is less powerful than the probable hardware that the program will run as specified in the problem description (Quad-core and 1Gb RAM).

Less performant approaches

Here, we'll explore some of the approaches we took and ultimately left and its reasons:

3 colored corners Our first marker design used 3 colors: one red corner on the top-left, one green corner on the bottom-right and two blue corners. Unfortunately, low quality printers and some lighting conditions make blue points hard to detect, therefore this idea was discarded.

Smaller tracker cells Initially, the tracking cells were only 1 cell wide and the matrix size was only 5x5 which gave us a 25 which gave us a 25-4 = 21 bits for the identifier and the redundancy. We're having problems tracking the marker at longer distances and final solution provided better results with only 1 bit lost for the identifier.

NDK performance At first, we developed our algorithm in C++ using the laptop camera and later ported it for android. This resulted in poor performance. Migrating the code for the Java API which performed better and simplified the code and building process.

Parallel multithreaded processing When trying to optimize the framerate and the algorithm, we've explored the possibility of parallel multithreading. This, theoretically, would enable us to optimize the camera framerate by sending each camera frame caught to a thread pool of the size of the number of processors on the device as seen in figure 5 mitigating the framerate slowdown by a delay in the output image.

We've extended a thread to have all the required memory allocations for our openCV matrices, so they wouldn't have to be re-allocated at each iteration and try to boost our application. Ultimately, while testing this showed a poorer performance.

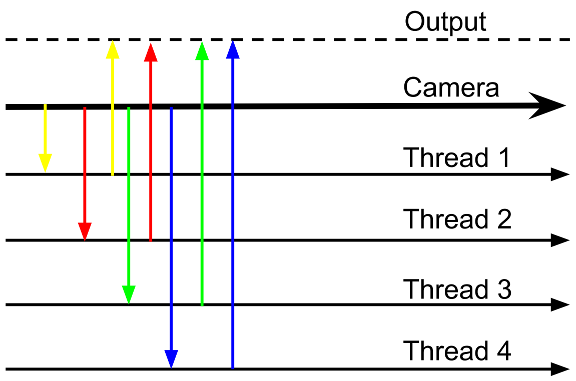

Figure 5: *Diagram of camera augmenting interpolation*

Detection results

Here, we show the augmenting of the image caught by the smartphone camera with the PT logo and the number encoded in the fiducial marker.

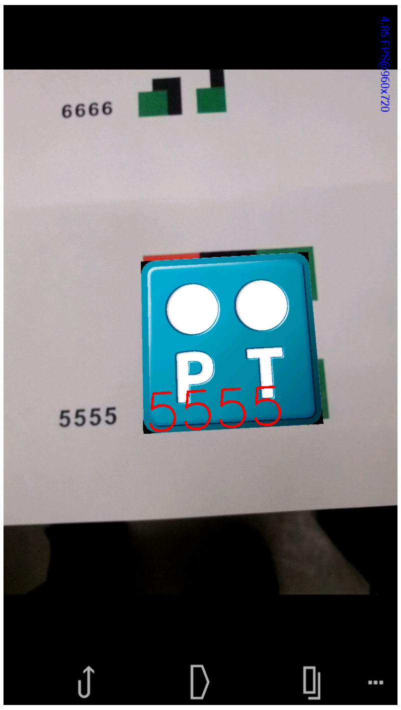

Figure 6: *Smartphone screenshot of image detection and it's augmentation*

Conclusion

We think this implementation clears all goals and requirements of the project. Even though we optimized the program, we wished we could had improved even further the framerate. Perhaps our multithreaded approach would be a good point to further explore and with the correct tweaking may had yielded better results.

Any question related to any of the work may and should be forwarded to the contact email.