Card Game with Augmented Reality

Mariana Oliveira Ramos - up201806869
https://sigarra.up.pt/feup/pt/
Daniel Garcia Lima Sarmento da Silva - up201806524
https://sigarra.up.pt/feup/pt/
Nuno Guilherme Amaral Santos - up201405774
https://sigarra.up.pt/feup/pt/
Professor Jorge Alves da Silva
https://sigarra.up.pt/feup/pt/
Professor Daniel Filipe Martins Tavares Mendes
https://sigarra.up.pt/feup/pt/

Faculdade de Engenharia
Universidade do Porto
Porto, PT. October, 2022.

Abstract

Almost everyone knows a few card games. The simplicity and flexibility of 52 pieces of numbered card stock are unrivaled. However, we made it even better by adding a layer of augmented reality to these "analog" card games. An application made to assist the famous Portuguese card game "Sueca" was created. It not only counts the points of the game so the user doesn't have to but also displays the winner at the end in a very visually appealing way.

1 Introduction

This document aims at detailing the implementation of an application to assist the playing of a simple card game with augmented reality techniques. In our case, the game being assisted is *Sueca*. *Sueca* is a very popular card game in Portugal played by four people (two teams of two players) where there are 120 points up for grabs and the pair that scores more than 60 points wins.

The program developed, as specified during this report, will assist the game by showing the number of points of each team in each round. By the end of the match, a trophy with the faces of the winning team will show up at the center of the table indicating which team won.

For simplicity purposes and to facilitate the demonstration of the game, the team who reaches 10 points will automatically win and the trophy will be displayed.

In order for the assistive application to work optimally, a camera should be placed pointing directly at the cards, on the table, and the conditions of the environment light should be favorable.

This implementation uses OpenCV in Python and integrates **image processing techniques**, **object recognition**, **camera calibration** and **visual information rendering**. All of the details can be found in the chapters below.

To know more regarding the structure and compilation of this project, there is a README.md file delivered together with the code.

2 Camera Calibration

One of the first steps in performing this project was calibrating the camera.

Some pinhole cameras introduce significant distortion to images. Two major kinds of distortion are radial distortion and tangential distortion and in order to get rid of these distortions we should find the distortion coefficients.

In addition to this, we need some other information, like the intrinsic and extrinsic parameters of the camera. The intrinsic parameters of the camera, like focal length and optical centers, can be used to create a camera matrix, which can then be used to remove distortion. On the other hand, extrinsic parameters correspond to the rotation and translation vectors which can be used to translate the coordinates of a 3D point in the world to a coordinate system.

To find all these parameters, we used the **Zhengyou Zhang camera calibration method** [5], in which we used ten sample images of a well-defined pattern, in our case, a chessboard. These images can be found on the file *chessboard_calibration* delivered together with the code.

For this method, we needed the set of 3D real-world points and the corresponding 2D coordinates of these points in the image. In order to find pattern in the chessboard we used the function *cv.findChessboardCorners()*. Then, we increased the corner's accuracy using *cv.cornerSubPix()*. Having our object points and image points, we proceeded to the calibration. For that, we used the function, *cv.calibrateCamera()* which returns the camera matrix with the intrinsic parameters, distortion coefficients, rotation, translation vectors, etc.

All the code implemented in this process is in the *camera_calibration.py* file and was based on the OpenCV Camera Calibration Tutorial [3].

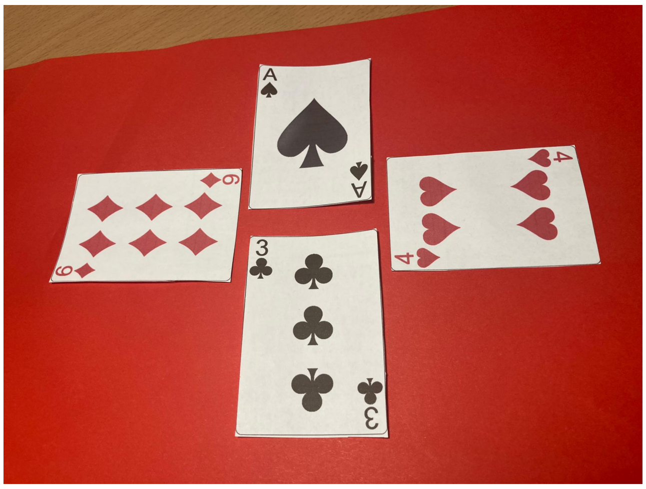
3 Global Solution

This section describes in detail the global implementation of our project.

We can break our solution down into parts. The heart of the problem is to get the points that each team collects on each round, based on the cards they place on the table.

To get the value of each card, first, we have to look at the cards using a camera and **identify that they are cards**. Then, based on the orientation of each card, we need to **find to which team it belongs to**. Afterward, we process each card and **recognize it** using our card database. Finally, once we calculate the points and which team won, we **create the trophy using image augmentation** techniques.

3.1 Card Segmentation



As shown in the image above, we have a red dark background for a tabletop, and the cards are laid out in a cross.

They are, however, not quite vertically aligned, and the picture itself is at a perspective. Thus, the first step was to perform **image segmentation** to pick out each card by itself and then **image registration** to line up the picture of each card with a flat, rectangular representation of a card.

We first read the picture and preprocess it (greyscale, blur, and threshold) it to get rid of artifacts and focus on the important features [4]. For that we built the function *binarize_image(image)*.

After the image is *"binarized"* we can find all contours in the image. These can be the edges of the cards themselves or the contours of the figures and letters in the cards. Thus, we looked for the four contours that span the most area, meaning the four cards, which we can then cycle through one by one.

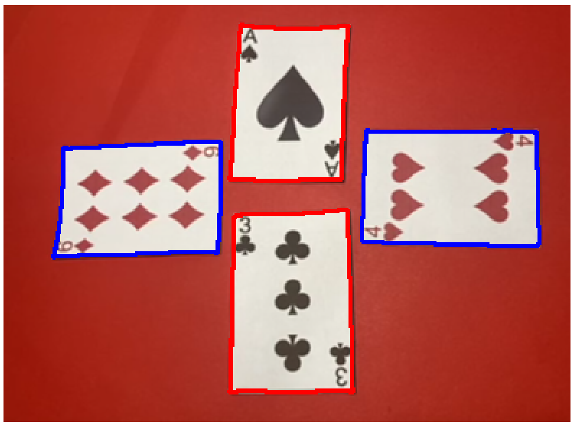
Now, we registered each card into a rectangular representation. We did this by approximating a polynomial from the contour and finding the minimum rotation-free bounding box. We used the functions *cv.findCoutours* and *cv.approxPolyDP()*. Note that, in case there are more objects in the scene, only countors with four corners are considered, ensuring they are indeed cards.

3.2 Teams classification

In order to find out to which team each card belonged, we simply decided to create a rule: the cards facing the horizontal direction are "team 1" and the cards in the vertical direction are "team 2".

This was checked on the function *find_team()*. We used the four corners of a card to find the top left point: min(x+y) and the top right point: max(x-y). Having these two points we calculated the bottom points and checked the direction of the card by measuring its sides.

The image below shows the teams being classified. The cards contoured in blue are team 1 and the ones contoured in red are team 2.



3.3 Card Recognition

After having a registered representation of each card and the teams defined, the next step is to recognize which card it is and its value. For this, we use a training deck provided by the professor, which contains the fifty-two images corresponding to each card.

Knowing what each card looks like, the next step is to match it up with the incoming candidate card. Having both images lined up/registered, and both have been through the same preprocessing, we found a lot of robust algorithms for this problem. One example of these algorithms was SIFT and SURF which can be used for this problem: we could recognize each digit/letter using character recognition and characteristic points, or even recognize the suit symbols on each card and count them to get the digit.

However, the simplest way we found here, was to calculate how different is the picture of the incoming card from each of the training cards.

However this candidate card needs to be placed in a frontal view so it can be actually compared with the cards stored in the data base. To do this we recurred to the *homography* concept [2].

Homography is a geometric transformation based on a matrix that relates a collineation from one projective space to another. We mapped the coordinates from the camera's frame image being processed with each one of the original cards stored on all possible rotations. In our implementation, we used the function *cv.getPerspectiveTransform()* to get the homography matrix and *cv.warpPerspective()* to obtain the actually frontal view perspective of the card.

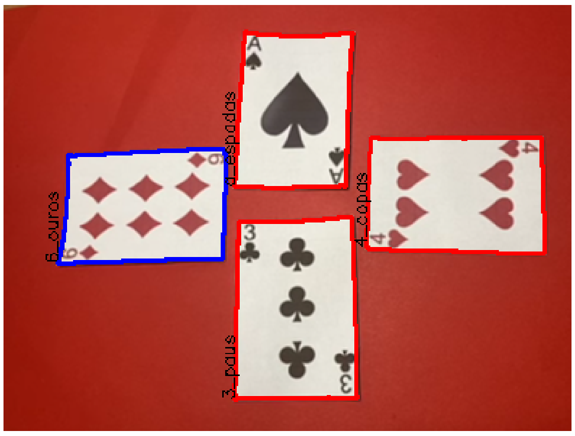
We also found several ways of calculating how different were the images:

- Template matching (*cv.matchTemplate()*)
- Feature matchig (*cv.BFMatcher()*)
- Doing it by pixel (*cv.bitwise_xor()*)
- Using Structural Similarity Index (*ssim()*)
- Using Histograms (*cv.calcHist()*)

However, the most efficient and effective one was using the **absolute difference between images**. For that we used the function *cv2.absdiff()*

Thus recognizing each card becomes a simple process of comparing each incoming card against each card against the deck, and going with the one with the minimum difference. As noted before, there are many more complex and robust approaches (including ones that involve the use of classifiers). However, since this is a fun project and I only had a few hours to write code, we're going to go with the version that takes the least amount of time!

The results of the card recognition process can be found in the image below, where the names of each card appear next to the corresponding one.

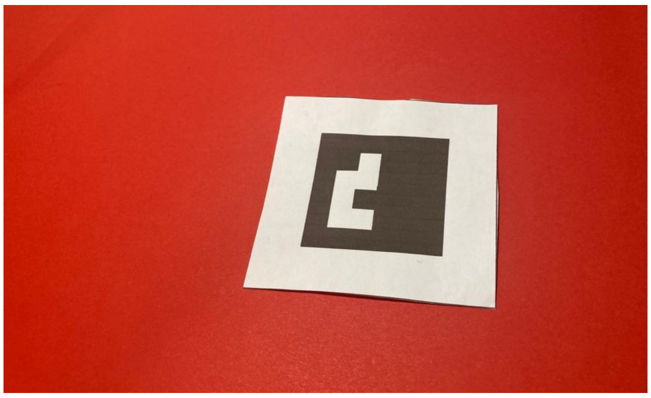


3.4 Image Augmentation

Once we knew which team won the round the final step was to build the trophy.

Firstly, we had to recognize our pattern image on the table where we wanted to draw the trophy. This process was the same as described in section *4.3 Card Recognition* and can be found on the code in the function *find_marker()*.

The image below shows the pattern image used.



In section *3. Camera Calibration*, we briefly explain how we did our camera calibration and found the **camera matrix and distortion coefficients**.

So once we had the camera calibration parameters and the coordinates of our pattern image, we just needed to find how the camera was placed in space to see our pattern image. Once we know how the object lies in space, we can draw some 2D diagrams in it to simulate the 3D effect. And that is what we did.

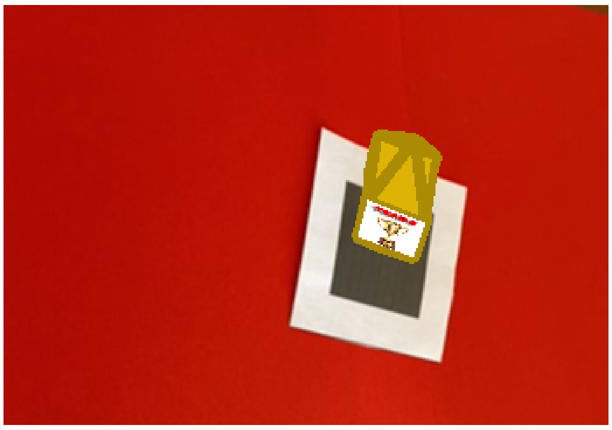
To calculate the **rotation and translation matrices**, we used the function, *cv.solvePnPRansac()*. This gave us the matrix of extrinsic parameters.

The RANSAC [1] parameter forces the function to use the Random Sample Consensus algorithm to compute the homography. This algorithm allows a robust fitting in the presence of a considerable number of outliers, which in real-world scenarios is very common. It iteratively calculates better-fitted homographies classifying the matched points as inliers or outliers on the distance between their coordinates and the coordinates in the original image after applying the intermediate homography.

Once we had both transformation matrices (the one with the intrincic parameters and the one with the extrinsic parameters obtained with the homography), we used them to project our cube points to the image plane, with the function *cv.projectPoints*. Essentially, we found the points on the image plane corresponding to each of our cube points in 3D space.

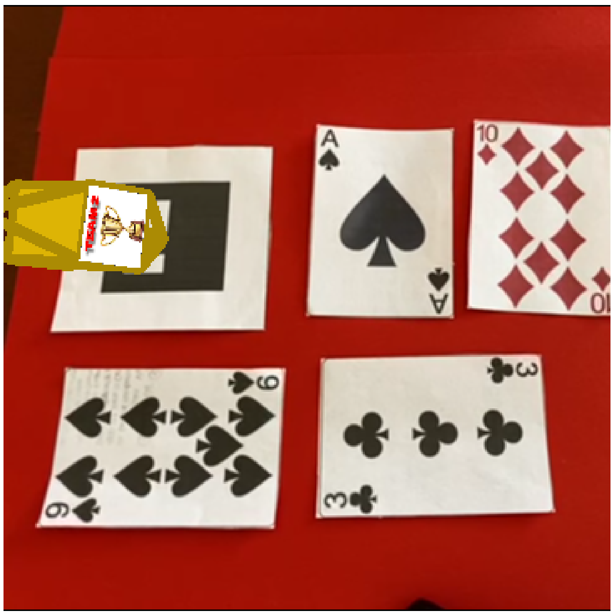
Once we got them, we just draw lines from the first corner of our pattern image to each of these points.

The results of the trophy can be seen in the figure below.



After the cube was drawn in perspective, we had to add an image (of the winning team) on top, also in perspective to the camera. Knowing the vertices of the cube and the corresponding points in the image plane, we used *cv.warpPerspective* which applies a perspective transformation to an image.

The final stage was to merge both images together, the image with the cards, marker and the drawn cube, and the warped image of the winning team. This was done in the function *_merge_images(image1, image2)*. The end result can be seen below.

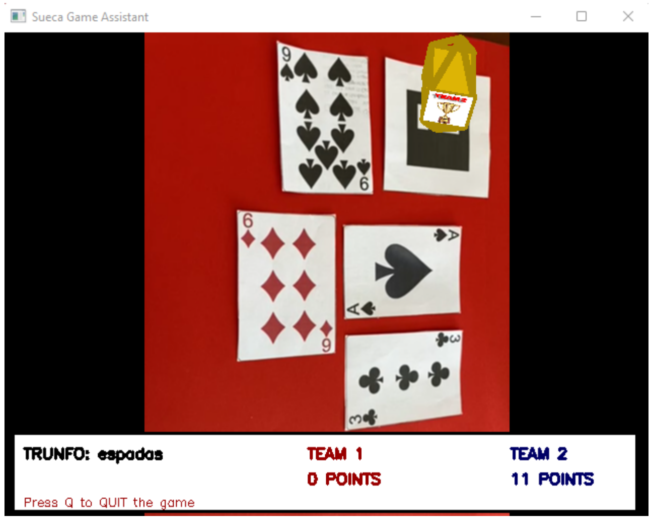


4 User Interface

The user is informed about the status of the game through an overlay at the bottom of the screen, which reports on:

- the trump card;
- the points for each team;
- the state of the game, i.e. if it can begin, what is the card to be assisted, or if the round is finished;
- the way to quit the game, by pressing 'Q'.

In terms of colour, team 1 is associated with the colour red and team 2 is corresponded by the colour blue. This is manifested in the card highlights when they are recognized, in the team points in the overlay and in the trophy image at the end of the game.



Sueca Game Assistant

TRUNFO: espadas

TEAM 1
0 POINTS

TEAM 2
11 POINTS

Press Q to QUIT the game

5 Limitations

Although we achieved great results in our implementation, there are several limitations to the used methods, due to lighting and camera position changes, and marker occlusion.

These problems can affect the card detector, which sometimes can't find sufficient marker features to match the camera frame to the cards_image. This also affects the way we detect the marker on the table, because of the miscalculation of the homography in some frames.

6 Conclusions

In this report, we described how to build an application to assist the playing of a simple card game with augmented reality techniques, using OpenCV. This implementation integrates image processing techniques, object recognition, camera calibration and visual information rendering.

We identified the cards with the help of methods such as *homography*, the *ransac* algorithm and absolute difference between images. And finally, we learned to exploit OpenCV graphics and augmented reality features to create 3D effects in our frame.

With OpenCV, building optical character recognition is simple and accurate. Although our results were good, they would be better with a better lighted environment and using lesser reflective surfaces, that combined with a higher quality of the camera could lead to a better outcome, but for this project these perfect conditions could not be reached.

References

[1] Satya Mallick. Random sample consensus, 2016. `https://en.wikipedia.org/wiki/Random_sample_consensus`.

[2] OpenCV. Basic concepts of the homography explained with code. `https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html`.

[3] OpenCV. Camera calibration, 2022. `https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html`.

[4] OpenCV. Image thresholding, 2022. `https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html`.

[5] Zhengyou Zhang. A flexible new technique for camera calibration. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr98-71.pdf`.