**UNIVERSITY OF INFORMATION TECHNOLOGY**

**\_\_Computer Science\_**

**~~~~~~~~~**o0o**~~~~~~~~~**

**Computer Vision Report**

**AUGMENTED REALITY WITH LOCAL FEATURE**

***Instructor* : Nguyen Vinh Tiep**

***Members* : Dang Nguyen Anh Tuan -- 18521591**

**Dang Huu Toan -- 18521503**

**Nguyen Van -- 18521632**

***Lớp* : CS231.L12.KHCL**

**HO CHI MINH­ City--2020**

CONTENT

[I. INTRODUCTION 3](#_Toc61776459)

[II. ANALYZING PROBLEMS 5](#_Toc61776460)

[**1.** **Reason for implementation** 5](#_Toc61776461)

[**2.** **Data description** 5](#_Toc61776462)

[III. IDENTIFY THE TARGET SURFACE 7](#_Toc61776463)

[**1.** **Features Detection** 7](#_Toc61776464)

[1.1. Feature extraction 7](#_Toc61776465)

[1.2. Feature description ORB 8](#_Toc61776466)

[**2.** **Feature Matching** 8](#_Toc61776467)

[IV. HOMOGRAPHY ESTIMATION USING RANSAC 11](#_Toc61776468)

[V. PROJECT 2D IMAGE 19](#_Toc61776469)

[**1.** **Augmented Reality with Image** 19](#_Toc61776470)

[**2.** **Creating the Mask** 19](#_Toc61776471)

[**3.** **Projecting 2D image** 20](#_Toc61776472)

[VI. PROJECT 3D MODEL 22](#_Toc61776473)

[**1.** **Pose estimation from a plane** 22](#_Toc61776474)

[**2.** **Model projection** 28](#_Toc61776475)

[VII. EVALUATE PROJECT 31](#_Toc61776476)

[VIII. SOURCE 32](#_Toc61776477)

# INTRODUCTION

We’ve all come across Augmented Reality at some point or the other in our lives. Be it while playing a game of Pokemon Go, amusing ourselves by clicking selfies trying out those wild Snapchat filters, decorating our homes through the IKEA app, or even while trying out different varieties of makeup with the L’Oreal app.

That’s augmented reality. To be present in the real world, yet to be able to interact with something that you can see and manipulate which isn’t really there.

So, what exactly is Augmented reality? In layman terms, Augmented Reality is a technology that enhances the real world by affixing layers of digital elements onto it. These elements include computer-generated graphics, sound or video effects, haptic feedback, or sensory projects. The intention behind adding this digital information is to provide an engaging and dynamic customer experience that is enabled with the input received from varied hardware like smart glass, smart lenses, and smartphones.

Augmented Reality (AR) is often mistaken with Virtual Reality (VR). The main difference between the two is that while Virtual Reality replaces the entire real environment with an artificial one, Augmented Reality is applied in a direct view of an existing real environment and adds elements like sounds, videos, or graphics onto it.

Augmented reality has highly advanced and developed over recent years. And it now established itself as an impressive tool for the industry as well as the general public. Such as in Military, Medical, Navigation, Tourism, maintenance and gaming

Therefore, we would like to introduce the methods to make that kind of thing. It may not good as the advanced technology nowaday, but we will try our best so that nothing goes wrong in the explanation process, and will make sure this is still the right way to do

Before that, we would like to thank instructor Nguyen Vinh Tiep, for allowing us to study the subject, and guide us through the learning process and solving difficult problems. Thank you.

# ANALYZING PROBLEMS

1. **Reason for implementation**

Our team chose Augmented Reality, partly because during the learning process, we were interested in this topic and wanted to learn more about it.

The rest is, most likely because AR is also a pretty popular topic in technology world nowaday, like Gokemon Go game once dominated the prominent mobile game market, or in fantasy movies with advanced technologies. So that why we want to have more knowledge in Augmented Reality to create more interesting things in life.

1. **Data description**

For AR work, we need something to render them in real-time which position and orientation matches the position and orientation of some predefined flat surface. So that why, we will use Webcam, which consist the flat surface of reference image to communicate between electronic device and real world

|  |  |  |
| --- | --- | --- |
| Target Image | Replace Image (2D) | 3D model |
| Firstly, we need a “Target image” for “Webcam” to recognize flat surface, so it can render the image2D or Model 3D into real world. | Secondly, we need “Replace image” (2D) to replace the “Target image” surface in real world via “Webcam” | Lastly, we need a “3D model”, for it to render on the “Target image” surface in real world via “Webcam” |
| https://scontent-sin6-2.xx.fbcdn.net/v/t1.15752-9/123140903_298984644433456_4672461618367074562_n.jpg?_nc_cat=108&ccb=2&_nc_sid=ae9488&_nc_ohc=wSwhIqtO7TAAX8FzTVA&_nc_ht=scontent-sin6-2.xx&oh=fa34545cb1ee0137096680fed028fb4a&oe=5FF9F742  “card” | https://scontent-sin6-2.xx.fbcdn.net/v/t1.15752-9/130531633_655797225095895_7496073300581232860_n.jpg?_nc_cat=103&ccb=2&_nc_sid=ae9488&_nc_ohc=jiPnuVJ5nt0AX_gy-Wk&_nc_ht=scontent-sin6-2.xx&oh=3dedb6f5c2f3501d83146ceab9c159a6&oe=5FF903F8  “Blue box” | https://scontent-sin6-1.xx.fbcdn.net/v/t1.15752-9/129980290_786496338573043_2082408292508889662_n.png?_nc_cat=104&ccb=2&_nc_sid=ae9488&_nc_ohc=9Jf-nfKH_ukAX-Gx5uP&_nc_ht=scontent-sin6-1.xx&oh=05637c2404e318f803ab32c09c5caaa0&oe=5FFA89A2  “Wolf” |

1. **Approaching problem**

The goal of this "Augmented reality with local feature" project is to project Replace Image (3D) and 3D model onto the flat surface of Target Image through Webcam, in python language.

For convenience, we would like to present the project into the following sections:

* The first part is identify the flat surface of Target Image, to do that, we have to:
  + Detect features of the Target Image and Webcam images using ORB
  + Find match features between the Target Image and Webcam images
* Part 2 is to find Homography using RANSAC.
* Part 3 project 2D Replace image onto the Target Image surface via webcam using bitwise function
* Part 4 estimating plane surface to project 3D model onto the Target Image surface via webcam
* The last one is the program code, the data file and the results for easier visualization

Due to limited references, limited qualifications and poor practical experience, the project has many shortcomings. We hope to receive comments and opinions from everyone to help us improve.

# IDENTIFY THE TARGET SURFACE

1. **Features Detection**

### Feature extraction

This step consists in first looking in both the reference surface in the webcam scene and target images for features that stand out and describe part the object to be recognized. This features can be later used to find the reference object in the target image.

We will assume we have found the object when a certain number of positive feature matches are found between the target and reference images. For this to work it is important to have a reference image where the only thing seen is the object (or surface, in this case) to be found.  We don’t want to detect features that are not part of the surface. And, although we will deal with this later, we will use the dimensions of the reference image when estimating the pose of the surface in a scene.

For a region or point of an image to be labeled as feature it should fulfill two important properties: first of all, it should present some uniqueness at least locally. Good examples of this could be corners or edges. Secondly, since we don’t know beforehand which will be, for example, the orientation, scale or brightness conditions of this same object in the image where we want to recognize it a feature should, ideally, be invariant to transformations; i.e, invariant against scale, rotation or brightness changes. As a rule of thumb, the more invariant the better.



Figure 2: features detected from the surface of reference in webcam scene

Figure 1: features detected from the target Image

### Feature description ORB

Once features have been found we should find a suitable representation of the information they provide. This will allow us to look for them in other images and also to obtain a measure of how similar two detected features are when being compared. This is were descriptors roll in.

A descriptor provides a representation of the information given by a feature and its surroundings. Once the descriptors have been computed the object to be recognized can then be abstracted to a [feature vector](https://en.wikipedia.org/wiki/Feature_vector),  which is a vector that contains the descriptors of the keypoints found in the image with the reference object.

There are many algorithms that extract image features and compute its descriptors and, for example , [SIFT](http://aishack.in/tutorials/sift-scale-invariant-feature-transform-introduction/), [SURF](http://www.vision.ee.ethz.ch/~surf/eccv06.pdf), or [Harris](http://aishack.in/tutorials/harris-corner-detector/). The one we will be using was developed at the OpenCV Lab and it is called [ORB](http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_orb/py_orb.html) (Oriented FAST and Rotated BRIEF). The shape and values of the descriptor depend on the algorithm used and, in our case,  the descriptors obtained will be binary strings.

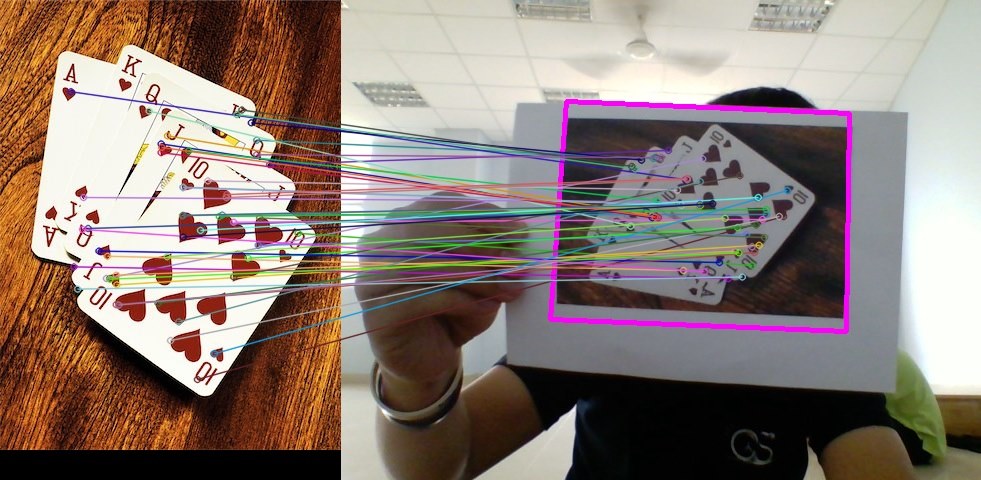
1. **Feature Matching**

Once we have found the features of both the object and the scene were the object is to be found and computed its descriptors it is time to look for matches between them. The simplest way of doing this is to take the descriptor of each feature in the first set, compute the distance to all the descriptors in the second set and return the closest one as the best match ( it is important to choose a way of measuring distances suitable with the descriptors being used. Since our descriptors will be binary strings we will use [Hamming distance](https://en.wikipedia.org/wiki/Hamming_distance)). This is a brute force approach, and more sophisticated methods exist.

For example, and this is what we will be also using, we could check that the match found as explained before is also the best match when computing matches the other way around, from features in the second set to features in the first set. This means that both features match each other. Once the matching has finished in both directions we will take as valid matches only the ones that fulfilled the previous condition. Figure 3 presents the best 20 matches found using this method.

Another option to reduce the number of false positives would be to check if the distance to the second to best match is below a certain threshold.  If it is, then the match is considered valid.

Figure 3: Closest 20 brute force matches found between the target surface and the scene



Finally, after matches have been found, we should define some criteria to decide if the object has been found or not. For this we defined a threshold on the minimum number of matches that should be found. If the number of matches is above the threshold, then we assume the object has been found. Otherwise we consider that there isn’t enough evidence to say that the recognition was successful.

MIN\_MATCHES = 20

cap = cv2.imread('scene.jpg', 0)

model = cv2.imread('model.jpg', 0)

*# ORB keypoint detector*

orb = cv2.ORB\_create()

*# create brute force matcher object*

bf = cv2.BFMatcher(cv2.NORM\_HAMMING, crossCheck=True)

*# Compute model keypoints and its descriptors*

kp\_model, des\_model = orb.detectAndCompute(model, None)

*# Compute scene keypoints and its descriptors*

kp\_frame, des\_frame = orb.detectAndCompute(cap, None)

*# Match frame descriptors with model descriptors*

matches = bf.match(des\_model, des\_frame)

*# Sort them in the order of their distance*

matches = sorted(matches, key=**lambda** x: x.distance)

**if** len(matches) > MIN\_MATCHES:

*# draw first 15 matches.*

cap = cv2.drawMatches(model, kp\_model, cap, kp\_frame,

matches[:MIN\_MATCHES], 0, flags=2)

*# show result*

cv2.imshow('frame', cap)

cv2.waitKey(0)

**else**:

**print** "Not enough matches have been found - %d/%d" % (len(matches), MIN\_MATCHES)

On a final note and before stepping into the next step of the process we must point out that, since we want a real time application, it would have been better to implement a tracking technique and not just plain recognition. This is due to the fact that object recognition will be performed in each frame independently without taking into account previous frames that could add valuable information about the location of the reference object.

Another thing to take into account is that, the easier to found the reference surface the more robust detection will be. In this particular sense, the reference surface we are using might not be the best option, but it helps to understand the process.

# HOMOGRAPHY ESTIMATION USING RANSAC

Once we have identified the reference surface in the current frame and have a set of valid matches we can proceed to estimate the homography between both images. As explained before, we want to find the transformation that maps points from the surface plane to the image plane (see Figure 4). This transformation will have to be updated each new frame we process.

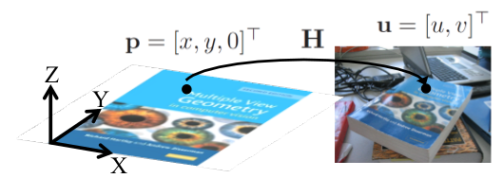


Figure 4: Homography between a plane and an image.

How can we find such a transformation? Since we have already found a set of matches between both images we can certainly find directly by any of the existing methods (We will be using RANSAC) an [homogeneous transformation](https://en.wikipedia.org/wiki/Transformation_matrix) that performs the mapping, but let’s get some insight into what we are doing here (see Figure 5). will explain the reasoning behind the transformation we are going to estimate.

What we have is an object (a plane in this case) with known coordinates in the, let’s say, World coordinate system and we take a picture of it with a camera located at a certain position and orientation with respect to the World coordinate system. We will assume the camera works following the [pinhole model](https://en.wikipedia.org/wiki/Pinhole_camera_model), which roughly means that the rays passing through a 3D point **p** and the corresponding 2D point **u** intersect at **c**, the camera center. A good resource if you are interested in knowing more about the pinhole model can be found [here](http://alumni.media.mit.edu/~maov/classes/comp_photo_vision08f/lect/09_image_formation.pdf).

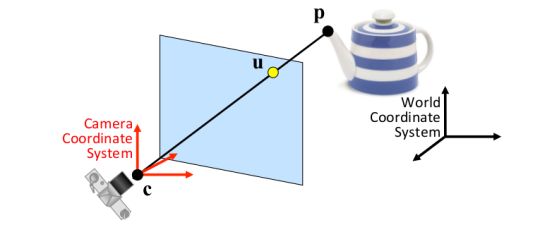


Figure 5: Image formation assuming a camera pinhole model.

Although not entirely true, the pinhole model assumption eases our calculations and works well enough for our purposes. The **u, v** coordinates (coordinates in the image plane) of a point **p** expressed in the Camera coordinate system if we assume a pinhole camera can be computed as (the derivation of the equation is left as an exercise to the reader):

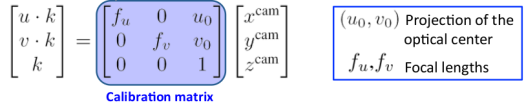


Figure 6: Image formation assuming a pinhole camera model.

Where the focal length is the distance from the pinhole to the image plane, the projection of the optical center is the position of the optical center in the image plane and **k** is a scaling factor. The previous equation then tells us how the image is formed. However, as stated before, we know the coordinates of the point **p**in the World coordinate system and not in the Camera coordinate system, so we have to add another transformation that maps points from the World coordinate system to the Camera coordinate system. The transformation that tells us the coordinates in the image plane of a point **p**in the World coordinate system is then:

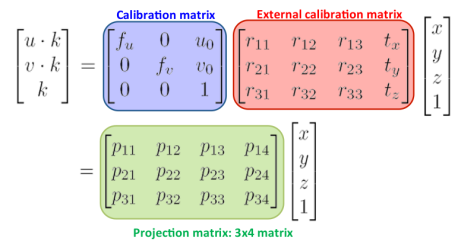


Figure 7: Computation of the projection matrix.

Luckily, since the points in the reference surface plane do always have its **z**coordinate equal to 0 (see Figure 4) we can simplify the transformation that we found above. It can be easily seen that the product of the **z** coordinate and the third column of the projection matrix will always be 0 so we can drop this column and the **z** coordinate from the previous equation. By renaming the calibration matrix as **A**and taking into account that the external calibration matrix is an homogeneous transformation:

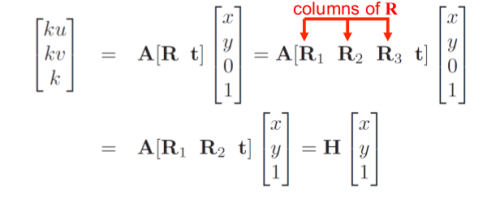


Figure 8: Simplification of the projection matrix.

From Figure 8 we can conclude that the homography between the reference surface and the image plane, which is the matrix we will estimate from the previous matches we found is:

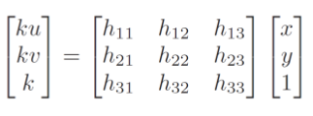


Figure 9: Homography between the reference surface plane and the target image plane.

There are several methods that allow us to estimate the values of the homography matrix, and you maight be familiar with some of them. The one we will be using is **RAN**dom **SA**mple **C**onsensus ([RANSAC](https://en.wikipedia.org/wiki/Random_sample_consensus)).

RANSAC is an iterative algorithm used for model fitting in the presence of a large number of outliers, and Figure 11 illustrates the main outline of the process. Since we cannot guarantee that all the matches we have found are actually valid matches we have to consider that there might be some false matches (which will be our outliers) and, hence, we have to use an estimation method that is robust against outliers. Figure 11 illustrates the problems we could have when estimating the homography if we considered that there were no outliers.

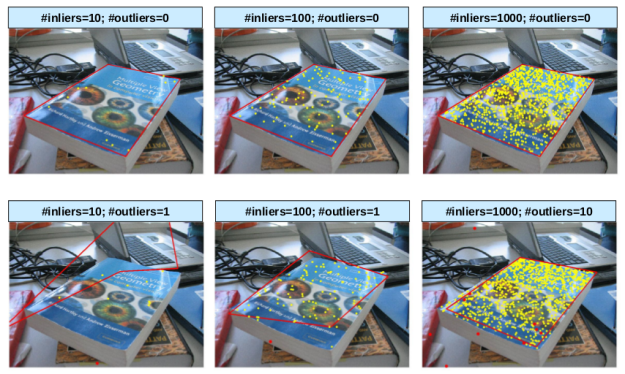


Figure 10: Homography estimation in the presence of outliers.

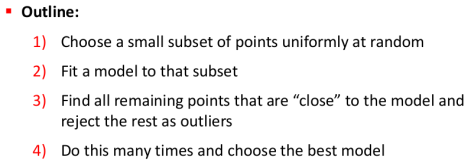


Figure 11: RANSAC algorithm outline

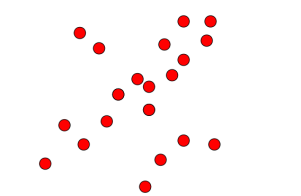
As a demonstration of how RANSAC works and to make things clearer, assume we had the following set of points for which we wanted to fit a line using RANSAC:

Figure 12: RANSAC algorithm outline

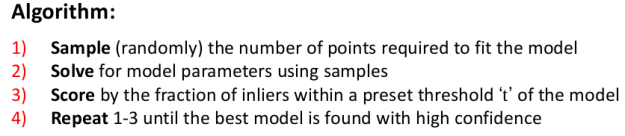
From the general outline presented in Figure 11 we can derive the specific process to fit a line using RANSAC (Figure 13).

Figure 13: RANSAC algorithm to fit a line to a set of points.

A possible outcome of running the algorithm presented above can be seen in Figure 14. Note that the first 3 steps of the algorithm are only shown for the first iteration (indicated by the bottom right number), and from that on only the scoring step is shown.

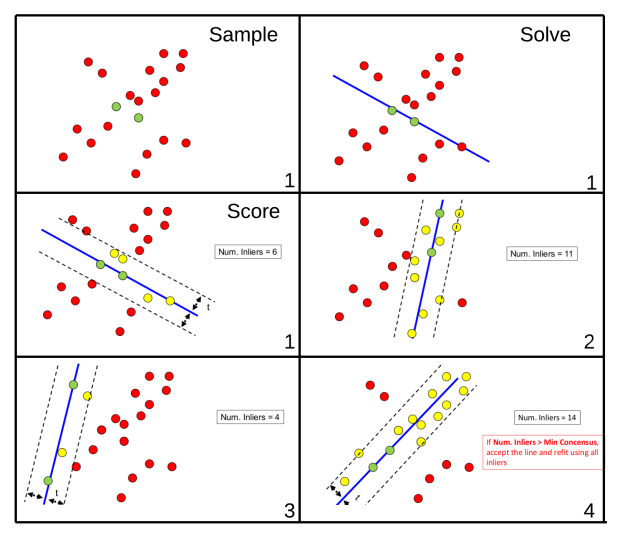


Figure 14: Using RANSAC to fit a line to a set of points.

Now back to our use case, homography estimation. For homography estimation the algorithm is presented in Figure 15. Since it is mainly math, we won’t go into details on why 4 matches are needed or on how to estimate **H**. However, if you want to know why and how it’s done, [this](http://www.uio.no/studier/emner/matnat/its/UNIK4690/v16/forelesninger/lecture_4_3-estimating-homographies-from-feature-correspondences.pdf) is a good explanation of it.

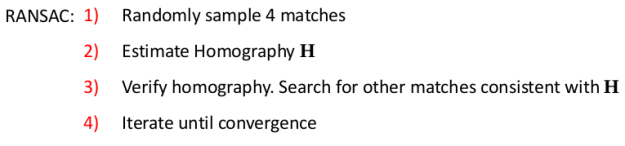


Figure 15: RANSAC for homography estimation

Before seeing how OpenCV can handle this for us we should discuss one final aspect of the algorithm, which is what does it mean that a match is consistent with **H**. What this mainly means is that if after estimating an homography we project into the target image the matches that were not used to estimate it then the projected points from the reference surface should be close to its matches in the target image. How close they should be to be considered consistent is up to you.

In OpenCV we can estimating the homography with RANSAC is as easy as:

*# assuming matches stores the matches found and*

*# returned by bf.match(des\_model, des\_frame)*

*# differenciate between source points and destination points*

src\_pts = np.float32([kp\_model[m.queryIdx].pt **for** m **in** matches]).reshape(-1, 1, 2)

dst\_pts = np.float32([kp\_frame[m.trainIdx].pt **for** m **in** matches]).reshape(-1, 1, 2)

*# compute Homography*

M, mask = cv2.findHomography(src\_pts, dst\_pts, cv2.RANSAC, 5.0)

Where 5.0 is the threshold distance to determine if a match is consistent with the estimated homography. If after estimating the homography we project the four corners of the reference surface on the target image and connect them with a line we should expect the resulting lines to enclose the reference surface in the target image. We can do this by:

*# Draw a rectangle that marks the found model in the frame*

h, w = model.shape

pts = np.float32([[0, 0], [0, h - 1], [w - 1, h - 1], [w - 1, 0]]).reshape(-1, 1, 2)

*# project corners into frame*

dst = cv2.perspectiveTransform(pts, M)

*# connect them with lines*

img2 = cv2.polylines(img\_rgb, [np.int32(dst)], True, 255, 3, cv2.LINE\_AA)

cv2.imshow('frame', cap)

cv2.waitKey(0)

Which results in:



Figure 16: Projected corners of the reference surface with the estimated homography. (Bounding Box)

# PROJECT 2D IMAGE

1. **Augmented Reality with Image**

Since we already have the bounding box we will use this to adjust our image so that it is in the correct spot and ready to be augmented. We will use the warpPerspective function for this.



Figure 18: Warp image

Figure 17: Bounding box

1. **Creating the Mask**

Inorder to add our images we can simply use the addWeighted function. But this function blends the images rather than overlaying one on top of the other. So the final result of this is no very realistic.

The other method is to remove the area in the webcam image where we want to overlay our target image and then add them together. This can be done using masking. So we already have one image ready which is the warped image that we created in the previous part. Now we will create the other image that we will add it with.

So first we are creating a mask based on the location of the target found. Now we can use the inverse method to find its negative. If we add the mask inverse and the webcam image we would get the anew image where all the webcam image information is shown except where the image is suppose to be augmented. So the black area can be thought of an empty space where we can add our image.

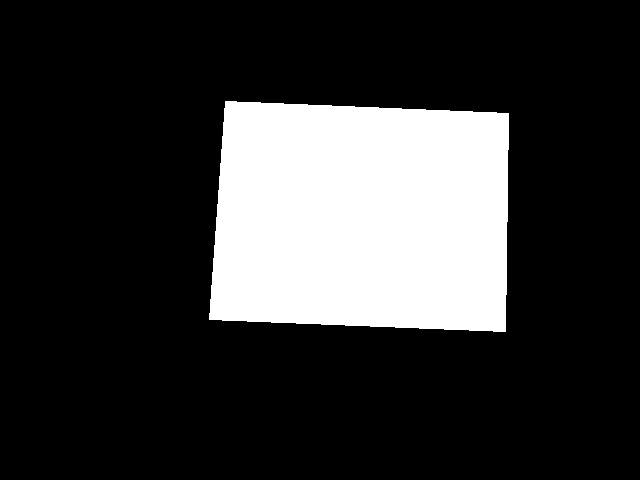


Figure 20: Inverse mask (bitwise not)

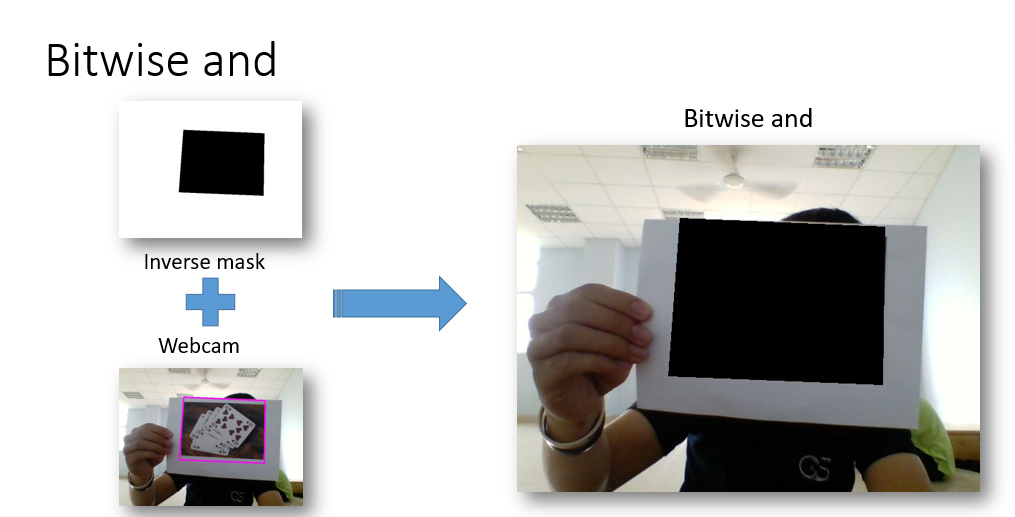
Figure 19: mask

1. **Projecting 2D image**

So once we have the imgAug which is our new masked image and the imgWarp, we can simply add them up using the bit wise or operator

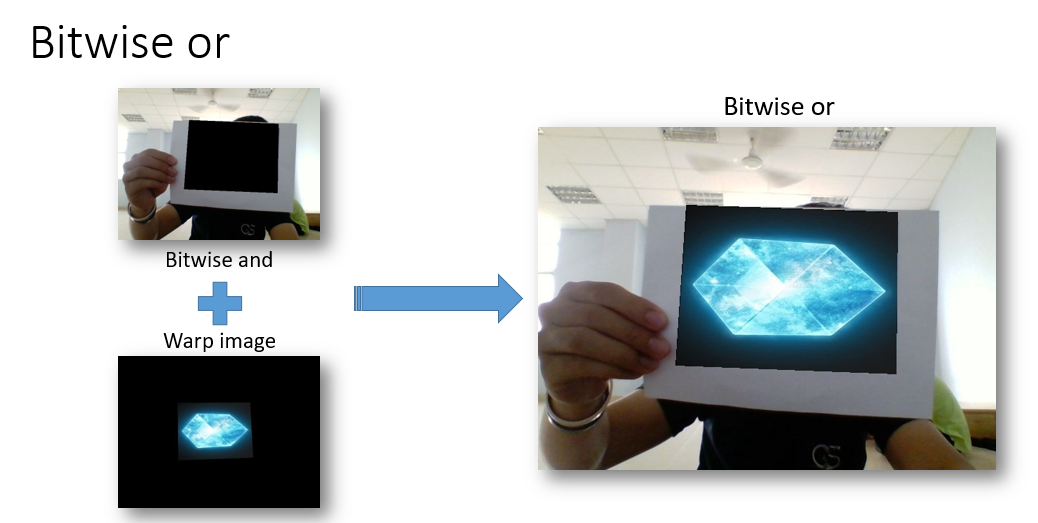
We will simply use Bitwise and function to add the inverse mask to the webcam scene:

* 1(mask--white back ground) \* ~1(Webcam’s inside boundary) = 1
* 0(mask—black boundary) \* ~1(Webcam’s outside boundary) = 0



Then after that, we will use bitwise or function to mix the Bitwise and scene and warp image together

* 0(the black of the warp Image) + ~1(bitwise and - background) = 1
* ~1(image of the warp Image) + ~1(bitwise and - blackmask ) = 1



# PROJECT 3D MODEL

## **Pose estimation from a plane**

What we should achieve to project our 3D models in the frame is, as we have already said, to extend our homography matrix. We have to be able to project not only points contained in the reference surface plane (z-coordinate is 0), which is what we can do now, but any point in the reference space (with a z-coordinate different than 0).

If we go back to the first paragraphs of the section Homography Estimation, we reached the conclusion that the 3×3 homography matrix was the product of the camera calibration matrix (**A**) by the external calibration matrix – which is an homogeneous transformation-. We dropped the third column (**R3**) of the homogeneous transformation because the z-coordinate of all the points we wanted to map was 0 (since all of them were contained in the reference surface plane). Figure 21 shows again the last steps of how we obtained the final expression of the homography matrix.

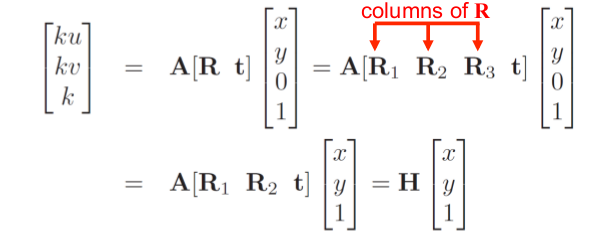


Figure 21: Derivation of the homography matrix

This meant that  we were left with the equation presented in Figure 22.

Selection_013

Figure 22: Components of the homography matrix.

However, we now want to project points whose z-coordinate is different than 0, which is what will allow us to project 3D models. To do so we should find a way to compute, from what we know, the value of **R3**. Why? Because once **z** is different than 0 we can no longer drop R3 from the transformation (see Figure 21) since it is needed to map the **z**-value of the point we want to project. The problem of extending our transformation from 2D to 3D will be solved then when we find a way to obtain the value of **R3** (see Figure 21 again). But, how can we get the value of **R3**?

We have already estimated the homography (**H**) , so its value is known. Furthermore, either by looking up the camera parameters or with a bit of common sense, we can easily know or make an educated guess of the values of the camera calibration matrix (**A).**Remember that the camera calibration matrix was:

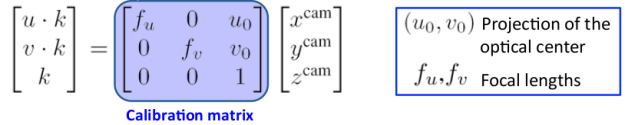


Figure 23: Camera calibration matrix.

Here you can find a nice article (part 3 of a recommended [series](http://ksimek.github.io/2012/08/13/introduction/)) that talks in detail about the camera calibration matrix and each of its values, and you can even [play with them](http://ksimek.github.io/perspective_camera_toy.html). Since all we am building is a prototype application we just made an educated guess of the values of this matrix. When it comes to the projection of the optical center, we just set **u0** and **v0**to be half the resolution of the frames we am capturing with OpenCV (**u0**=320 and**v0**=240). As for the focal length, this [article](https://www.learnopencv.com/approximate-focal-length-for-webcams-and-cell-phone-cameras/) provides some useful information on how to estimate the focal length of a webcam or a cellphone camera. we set **fu** and **fv**to the same value, and found that a focal length of 800 worked quite well for me. You may have to adjust these parameters to your actual set up or even go on [calibrating your camera](http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html).

Now that we have estimates of both the homography matrix **H** and our camera calibration matrix **A,**we can easily recover **R1**, **R2** and **t**by multiplying the inverse of **A**by **H:**

Selection_014

Figure 24: Recovering the external calibration matrix values from the estimated homography and the camera calibration matrix.

Were the values of **G1**, **G2** and **G3** can be regarded as:

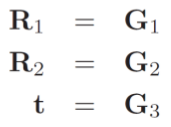


Figure 25: Extracting the values of the external calibration matrix.

Now, since the external calibration matrix [**R1 R2 R3 t**] is an homogeneous transformation that maps points amongst two different reference frames we can be sure that [**R1 R2 R3**] have to be orthonormal. Hence, theoretically we can compute **R3** as:

Selection_016

Figure 26: Computation of R3.

Unluckily, getting the value of **R3**is not as simple as that. Since we obtained **G1**,**G2** and **G3**from estimations of **A** and **H**there is no guarantee that **[R1=G1 R2=G2 R3=G1xG2]** will be orthonormal. The problem is then to get a pair of vectors that are close to**G1** and **G2** (since**G1** and **G2** are estimates of the real values of **R1** and **R2**) and that are orthonormal. Once this pair of vectors has been found (**R1′** and **R2′**) then it will be true that **R3 = R1’xR2′**, so finding the value of **R3** will be trivial. There are many ways in which we can find such a basis, an we will explain on of them. Its main benefit, from my point of view, is that it does not directly include any angle-related computation and that, once you get the hang of it, it is quite straightforward.

Finally and before diving into the explanation, the fact that the vectors we are looking for have to be close to **G1** and **G2**and not just any orthonormal basis in the same plane as **G1** and **G2**is important in understanding why some of the next steps are required so make sure you understand it before moving on. we will try my best in explaining the process by which we will get this new basis but if don’t succeed in doing so do not hesitate to tellusand we will try to rephrase the explanation and make it clearer. It will be useful to have at hand Figure 24 since it provides visual information that can help in understanding the process. **Note that what we am calling G1 and G2 are called R1 and R2 respectively in Figure 27**. Let’s go for it!

We start with the reasonable assumption that, since **G1** and **G2** are estimates of the real **R1** and **R2** (which are orthonormal), the angle between **G1** and **G2** will be approximately 90 degrees (in the ideal case it will be exactly 90 degrees). Furthermore, the modulus of each of this vectors will be close to 1. From **G1** and **G2** we can easily compute an orthogonal basis -this meaning that the angle between the basis vectors will **exactly** be 90 degrees- that will be rotated approximately 45 degrees clockwise with respect to the basis formed by **G1** and **G2.**This basis is the one formed by **c=G1+G2** and  **d = c x p = (G1+G2) x (G1 x G2) i**n Figure 24. If the vectors that form this new basis (**c**,**d**) are made unit vectors and rotated 45 degrees counterclockwise (note that once the vectors have been transformed into unit vectors – **v / ||v||** – rotating the basis is as easy as **d’ = c / ||c|| + d / ||d||**and  **c’ = c / ||c|| – d / ||d||**), guess what? We will have an orthogonal basis which is pretty close to our original basis (**G1**, **G2)**. If we normalize this rotated basis we will finally get the pair of vectors we were looking for. You can see this whole process on Figure 27.

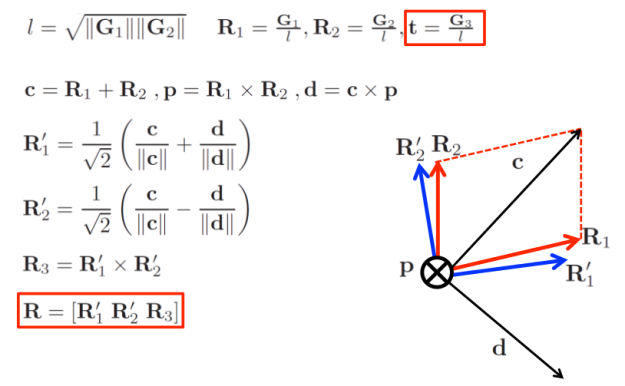


Figure 27: Normalization of [R1 R2 R3] to guarantee that they are orthonormal.

Once this basis (**R1′, R2′**) has been obtained it is trivial to get the value of **R3** as the cross product of **R1′** and **R2′**.  This was tough, but we are all set now to finally obtain the matrix that will allow us to project 3D points into the image. This matrix will be the product of the camera calibration matrix **A**by **[R1′ R2′ R3 t]** (where **t** has been updated as shown in Figure 24). So, finally:

**3D projection matrix = A · [R1′ R2′ R3 t]**

Note that this 3D projection matrix will have to be computed for each new frame. With numpy we can, in a few lines of code, define a function that computes it for us:

**def** projection\_matrix(camera\_parameters, homography):

*"""*

*From the camera calibration matrix and the estimated homography*

*compute the 3D projection matrix*

*"""*

*# Compute rotation along the x and y axis as well as the translation*

homography = homography \* (-1)

rot\_and\_transl = np.dot(np.linalg.inv(camera\_parameters), homography)

col\_1 = rot\_and\_transl[:, 0]

col\_2 = rot\_and\_transl[:, 1]

col\_3 = rot\_and\_transl[:, 2]

*# normalise vectors*

l = math.sqrt(np.linalg.norm(col\_1, 2) \* np.linalg.norm(col\_2, 2))

rot\_1 = col\_1 / l

rot\_2 = col\_2 / l

translation = col\_3 / l

*# compute the orthonormal basis*

c = rot\_1 + rot\_2

p = np.cross(rot\_1, rot\_2)

d = np.cross(c, p)

rot\_1 = np.dot(c / np.linalg.norm(c, 2) + d / np.linalg.norm(d, 2), 1 / math.sqrt(2))

rot\_2 = np.dot(c / np.linalg.norm(c, 2) - d / np.linalg.norm(d, 2), 1 / math.sqrt(2))

rot\_3 = np.cross(rot\_1, rot\_2)

*# finally, compute the 3D projection matrix from the model to the current frame*

projection = np.stack((rot\_1, rot\_2, rot\_3, translation)).T

**return** np.dot(camera\_parameters, projection)

Note that the sign of the homography matrix is changed in the first line of the function. we will let you think why this is required.

As a summary, let us shortly recap our thought process to estimate the 3D matrix projection.

1. Derive the mathematical model of the projection (image formation). Conclude that, at this point, everything is an unknown.
2. Heuristically estimate the homography via keypoint matching and RANSAC. -> H is no longer unknown.
3. Estimate the camera calibration matrix. -> A is no longer unknown.
4. From the estimations of the homography and the camera calibration matrix along with the mathematical model derived in 1, compute the values of G1, G2 and t.
5. Find an orthonormal basis in the plane (R1′, R2′) that is similar to (G1,G2), compute R3 from it and update the value of t.

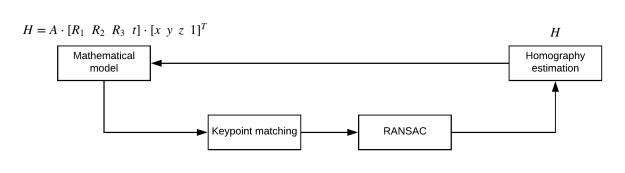


Figure 28: Thought process to recover the 3D projection matrix.

1. **Model projection**

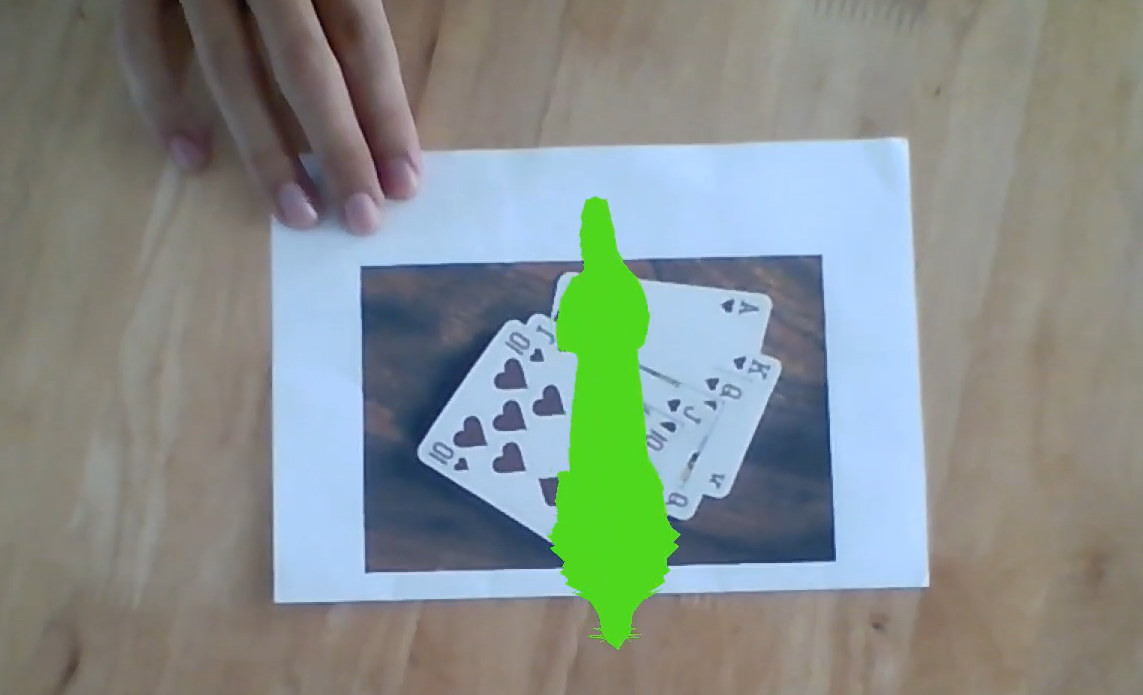


Figure 29: Wolf projection

We already have all the required tools needed to project our 3D models. The only thing we have to do now is get some 3D figures and project them

We are currently only using simple models in Wavefront .obj format. Why OBJ format? Because we found them easy to process and render directly with bare Python without having to make use of other libraries such as OpenGL. The problem with complex models is that the amount of processing they require is way more than what my computer can handle. Since we want my application to be real-time, this limits the complexity of the models we am able to render.

We downloaded 3D models format from internet . Quoting Wikipedia, *a .obj file is a geometry definition file format*. If you download it and open the .obj file with your favorite text editor you will get an idea on how the model’s geometry is stored.

Once the model is loaded we just have to implement a function that reads this data and projects it on top of the video frame with the projection matrix we obtained in the previous section. To do so we take every point used to define the model and multiply it by the projection matrix. One this has been done, we only have to fill with color the faces of the model. The following function can be used to do so:

**def** render(img, obj, projection, model, color=False):

vertices = obj.vertices

scale\_matrix = np.eye(3) \* 3

h, w = model.shape

**for** face **in** obj.faces:

face\_vertices = face[0]

points = np.array([vertices[vertex - 1] **for** vertex **in** face\_vertices])

points = np.dot(points, scale\_matrix)

*# render model in the middle of the reference surface. To do so,*

*# model points must be displaced*

points = np.array([[p[0] + w / 2, p[1] + h / 2, p[2]] **for** p **in** points])

dst = cv2.perspectiveTransform(points.reshape(-1, 1, 3), projection)

imgpts = np.int32(dst)

**if** color **is** False:

cv2.fillConvexPoly(img, imgpts, (137, 27, 211))

**else**:

color = hex\_to\_rgb(face[-1])

color = color[::-1] *# reverse*

cv2.fillConvexPoly(img, imgpts, color)

**return** img

There are three things to be highlighted from the previous function:

1. The scale factor: Since we don’t know the actual size of the model with respect to the rest of the frame, we may have to scale it (manually for now) so that it haves the desired size. The scale matrix allows us to resize the model.
2. We like the model to be rendered on the middle of the reference surface frame. However, the reference frame of the models is located at the center of the model. This means that if we project directly the points of the OBJ model in the video frame our model will be rendered on one corner of the reference surface. To locate the model in the middle of the reference surface we have to, before projecting the points on the video frame, displace the x and y coordinates of all the model points by half the width and height of the reference surface.
3. There is an optional color parameter than can be set to True. This is because some models also have color information that can be used to color the different faces of the model. we didn’t test it enough and setting it to True might result in some problems. It is not 100% guaranteed this feature will work.

# EVALUATE PROJECT

Those are all the fundamental ways to make Augment reality using local features, with it everyone can freely use and improve it

Because we don’t have enough knowledge to use OpenGL in our project, so The 3d model don’t have texture

Beside that, since we use ORB descriptor, it’ll very vibrational, and we still haven’t figured it out how to fix that problem

Still, if we have time, we will improve this project in the future.

# SOURCE

Example Result:

Source code:

Document:

1. <https://bitesofcode.wordpress.com/2017/09/12/augmented-reality-with-python-and-opencv-part-1/>
2. <https://bitesofcode.wordpress.com/2018/09/16/augmented-reality-with-python-and-opencv-part-2/>
3. <https://www.murtazahassan.com/augmented-reality-opencv-python/>