# Augmented Reality (AR)

Is a real time enhancement of object which reside in real world with computer-generated perpetual information which can be text, graphic, audio.

1) Script: import necessary libraries

```
# import the necessary packages
import numpy as np
import argparse
import imutils
import sys
import cv2
```

2) Script:using the command python opencv_ar_image.py --image examples/input_01.jpg --source sources/squirrel.jpg to run the project,the code below picks the arguments.--image the path to the image containing pantone which we want to enhance,while source is the path to image to replace the pantone in the input image

```
# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required=True,
    help="path to input image containing ArUCo tag")
ap.add_argument("-s", "--source", required=True,
    help="path to input source image that will be put on input")
args = vars(ap.parse_args())
```

3) Script:Here we read the input image from path specified,then using imutils.resize to resize the width of the input image.The resized images spatial dimensions is saved in imgH and imgW.Also load the source image from disk.

```
# load the input image from disk, resize it, and grab its spatial
# dimensions
print("[INFO] loading input image and source image...")
image = cv2.imread(args["image"])
image = imutils.resize(image, width=600)
(imgH, imgW) = image.shape[:2]
# load the source image from disk
source = cv2.imread(args["source"])
```

4) Script:The goal here is to create aruco detector ,detect the markers in the image ,and to do that we need aruco dictionary and set aruco detector parameter to be used

```
# load the ArUCo dictionary, grab the ArUCo parameters, and detect
# the markers
print("[INFO] detecting markers...")
arucoDict = cv2.aruco.Dictionary_get(cv2.aruco.DICT_ARUCO_ORIGINAL)
arucoParams = cv2.aruco.DetectorParameters_create()
(corners, ids, rejected) = cv2.aruco.detectMarkers(image, arucoDict,
    parameters=arucoParams)
```

5) script:Check if there exist the four corner is not exit the program using sys.exit(0)

```
if len(corners) != 4:
    print("[INFO] could not find 4 corners...exiting")
    sys.exit(0)
```

6) Script: Getting to this following code before means the program found the 4 corners.We get the ids of the Aruco markers and initialize reference points

```
# otherwise, we've found the four ArUco markers, so we can continue
# by flattening the ArUco IDs list and initializing our list of
# reference points
print("[INFO] constructing augmented reality visualization...")
ids = ids.flatten()
refPts = []
```

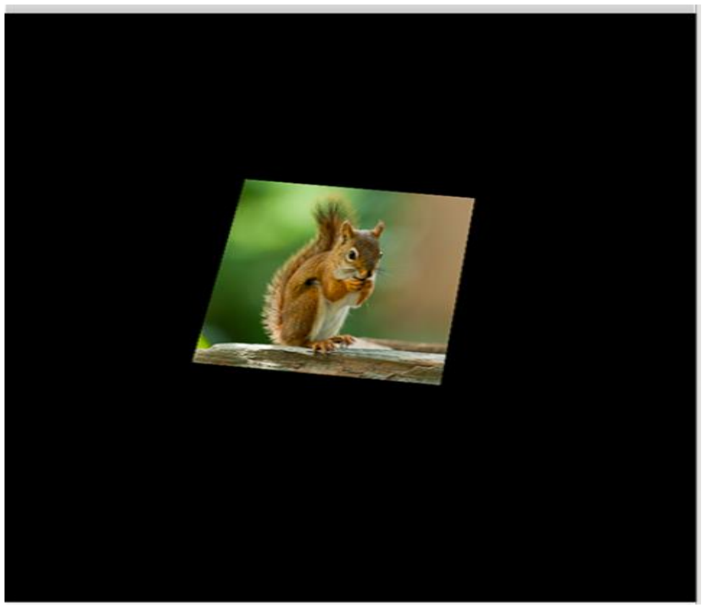
7) Script:Looping through the ids to get the indices in the order,top-left,top-right,bottom-right,bottom-left so as get corners to populate the refPts

```
for i in (923, 1001, 241, 1007):
    # grab the index of the corner with the current ID and append
the
    # corner (x, y)-coordinates to our list of reference points
    j = np.squeeze(np.where(ids == i))
    print(j)
    corner = np.squeeze(corners[j])
    refPts.append(corner)
```

8) Script: The next goal is to do perspective warp and to do so we need compute homography matrix which needs both source and destination matrix. The destination matrix(dstMat) is obtained from our reference point while the source matrix is obtained from the source

image.With cv2.findHomography(srcMat, dstMat) we find the homography matrix there perform perspective warp using cv2.warpPerspective()

```
# unpack our ArUco reference points and use the reference points to
# define the *destination* transform matrix, making sure the points
# are specified in top-left, top-right, bottom-right, and bottom-
left
# order
(refPtTL, refPtTR, refPtBR, refPtBL) = refPts
dstMat = [refPtTL[0], refPtTR[1], refPtBR[2], refPtBL[3]]
dstMat = np.array(dstMat)

# grab the spatial dimensions of the source image and define the
# transform matrix for the *source* image in top-left, top-right,
# bottom-right, and bottom-left order
(srcH, srcW) = source.shape[:2]
srcMat = np.array([[0, 0], [srcW, 0], [srcW, srcH], [0, srcH]])

# compute the homography matrix and then warp the source image to
the
# destination based on the homography
(H, _) = cv2.findHomography(srcMat, dstMat)
warped = cv2.warpPerspective(source, H, (imgW, imgH))
```



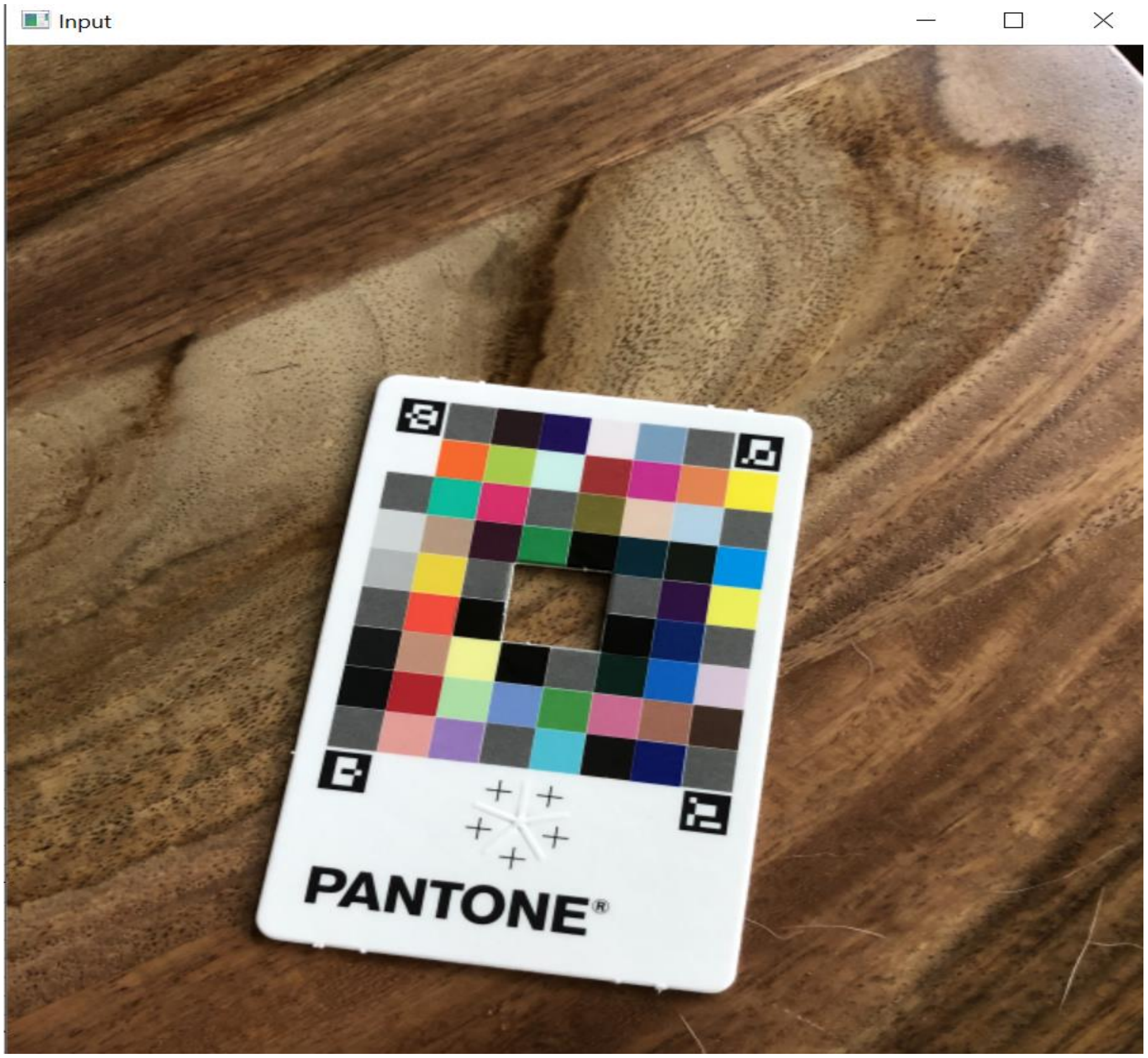
9) Script:Next go is to overlay the warp image on the destination image.First we create a mask,then depending on if we want black lines around the warped image we apply dilation operation.the mask created so far has single channel,so we create a 3 channel mask by stacking depth-wise.The next step is to multiply the masked and wrapedImage also the input image and the masked together and add the two result

```
# construct a mask for the source image now that the perspective
warp
# has taken place (we'll need this mask to copy the source image
into
# the destination)
mask = np.zeros((imgH, imgW), dtype="uint8")
cv2.fillConvexPoly(mask, dstMat.astype("int32"), (255, 255, 255),
    cv2.LINE_AA)

# this step is optional, but to give the source image a black border
# surrounding it when applied to the source image, you can apply a
# dilation operation
rect = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
mask = cv2.dilate(mask, rect, iterations=2)
# create a three channel version of the mask by stacking it depth-
wise,
# such that we can copy the warped source image into the input image
maskScaled = mask.copy() / 255.0
maskScaled = np.dstack([maskScaled] * 3)

# copy the warped source image into the input image by (1)
multiplying
# the warped image and masked together, (2) multiplying the original
# input image with the mask (giving more weight to the input where
# there *ARE NOT* masked pixels), and (3) adding the resulting
# multiplications together
warpedMultiplied = cv2.multiply(warped.astype("float"), maskScaled)
imageMultiplied = cv2.multiply(image.astype(float), 1.0 -
maskScaled)
output = cv2.add(warpedMultiplied, imageMultiplied)
output = output.astype("uint8")
```

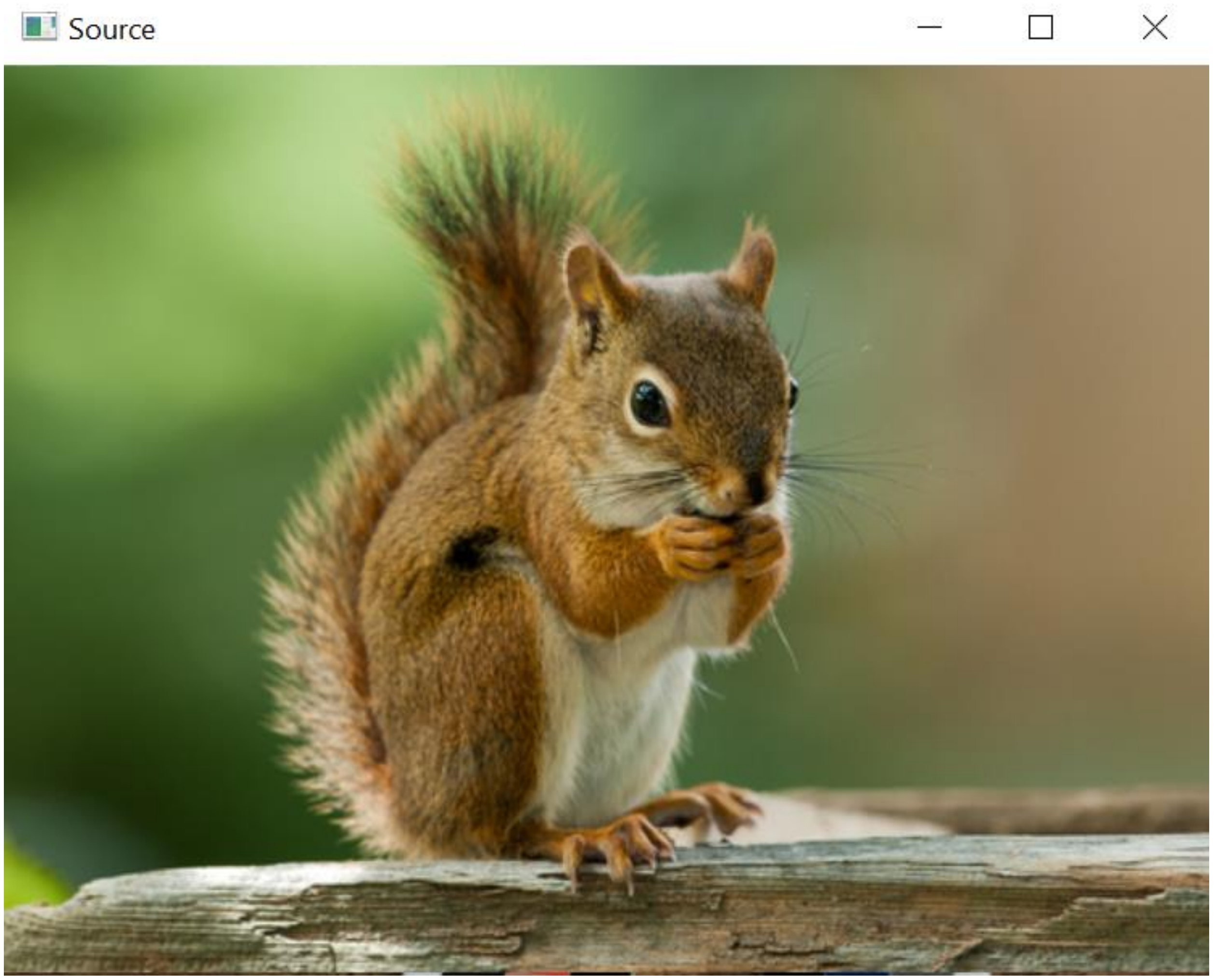
10) Script:The result is shown below.

```
# show the input image, source image, output of our augmented
reality
cv2.imshow("Input", image)
cv2.imshow("Source", source)
cv2.imshow("OpenCV AR Output", output)
cv2.waitKey(0)
```

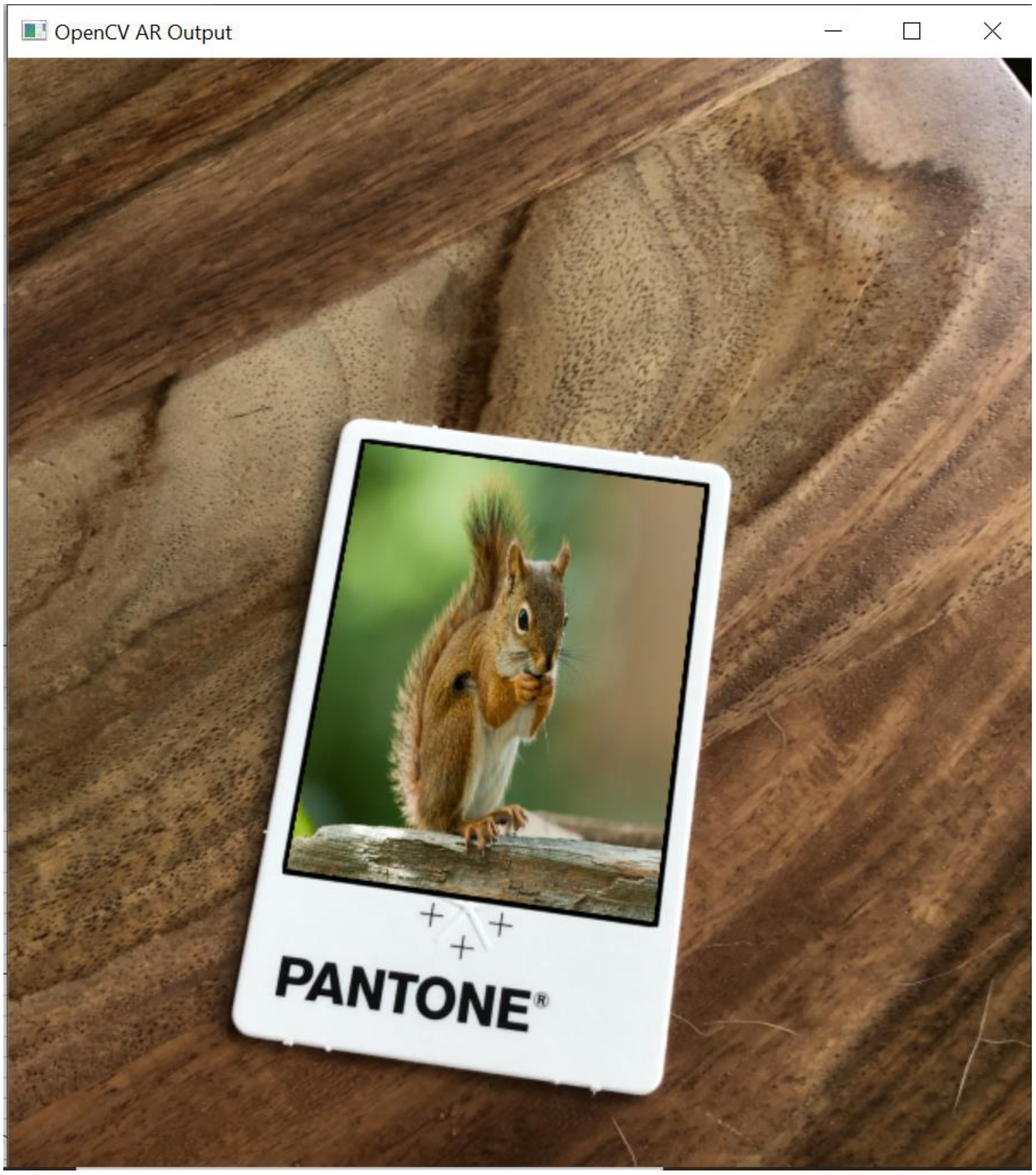
Result:



Input Image



Source Image



Resulting Image