Robot Vision Phase 2

Thomas Kong and Amolak Plaha

Main Topics Used in Phase 2:

- Camera Calibration
- Masking Image
- ARUCO Detection
- ORB Feature Detection

Similarities to Phase 1:

- Measure Object in 2D image
- Uses Camera Calibration
- Use Aruco Detection

New Features in Phase 2:

- Uses only one image for Measurement
- Can identify objects in image
- Can measure multiple objects in image

Basic Procedure:

1. Code performs camera calibration for distortion coefficients and k matrix
2. Take image input for measurement and finds Aruco Marker in image
3. Use Aruco Marker as reference for measurement (must know Marker's length)
4. Perform masking to find contours within image and create bounding box for each notable figure

```
# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((6*8,3), np.float32)
objp[:,:2] = np.mgrid[0:8,0:6].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# use images captured by camera
images = glob.glob("opencv*.jpg")

for fname in images:
    img = cv2.imread(fname)

    # Resize to make sure the image does not contain too much pixels for Opencv to find patterns

    img = cv2.resize(img, (640,480))

    #if the above resizing is causing distortion, please uncomment this line and comment out the previous naive resizing method
    #img = image_resize(img, height=640)

    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    # Find the chess board corners, we use 8*6 grid
    ret, corners = cv2.findChessboardCorners(gray, (8,6),None)
    # If found, add object points, image points (after refining them)
    if ret == True:
      objpoints.append(objp)

      # cv2.cornerSubPix() increases the accuracy of the corner coordinates
      corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
      imgpoints.append(corners2)

      # Draw and display the corners
      img = cv2.drawChessboardCorners(img, (8,6), corners2,ret)
#         cv2.imshow('img',img)
#         cv2.waitKey(500)

# cv2.destroyAllWindows()
```

```python
# function for finding the midpoint
def midpoint(A, B):
  return ((A[0] + B[0]) * 0.5, (A[1] + B[1]) * 0.5)
```

```python
# Reference object dimensions
apriltag_img = img

# load Tag36h11 in aruco dictionary
ARUCO_DICT = {"DICT_APRILTAG_36h11": cv2.aruco.DICT_APRILTAG_36h11}
arucoDict = cv2.aruco.Dictionary_get(ARUCO_DICT["DICT_APRILTAG_36h11"])

arucoParams = cv2.aruco.DetectorParameters_create()
(image_points, ids, rejected) = cv2.aruco.detectMarkers(apriltag_img, arucoDict,  parameters=arucoParams)

# by default, the four corners are (x,y) coordinates in the image, with pixel as unit.
# The order of the four corners are top-left, top-right, bottom-right, and bottom-left

# Ensure that at least one tag is detected, then prepare the corners for drawing:
if len(image_points) > 0:
  # flatten the ArUco IDs list
  ids = ids.flatten()
  # loop over the detected ArUCo corners
  for (markerCorner, markerID) in zip(image_points, ids):
    # extract the marker corners
    image_points = markerCorner.reshape((4, 2))
    (topLeft, topRight, bottomRight, bottomLeft) = image_points
    # convert each of the (x, y)-coordinate pairs to integers
    topRight = (int(topRight[0]), int(topRight[1]))
    bottomRight = (int(bottomRight[0]), int(bottomRight[1]))
    bottomLeft = (int(bottomLeft[0]), int(bottomLeft[1]))
    topLeft = (int(topLeft[0]), int(topLeft[1]))

  # draw the bounding box of the ArUCo detection
cv2.line(apriltag_img, topLeft, topRight, (0, 255, 0), 4)
cv2.line(apriltag_img, topRight, bottomRight, (0, 255, 0), 4)
cv2.line(apriltag_img, bottomRight, bottomLeft, (0, 255, 0), 4)
cv2.line(apriltag_img, bottomLeft, topLeft, (0, 255, 0), 4)

cv2_imshow(apriltag_img)
```

```
(tMidX, tMidY) = midpoint(topLeft, topRight)
(bMidX, bMidY) = midpoint(bottomLeft, bottomRight)
(lMidX, lMidY) = midpoint(topLeft, bottomLeft)
(rMidX, rMidY) = midpoint(topRight, bottomRight)

# calculate the Euclidean distances between the midpoints
dA = dist.euclidean((tMidX, tMidY), (bMidX, bMidY))
dB = dist.euclidean((lMidX, lMidY), (rMidX, rMidY))
```

```
# Pixel Ratio
Real_measurement = 35 # 3.5 cm or 35 mm
pixel_ratio = dB / 35
```

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (7, 7), 0)
```

```
edge = cv2.Canny(gray, 15, 100) #play w/min and max
edge = cv2.dilate(edge, None, iterations=1)
edge = cv2.erode(edge, None, iterations=1)
```

```
# find contours in the edge map
cntours = cv2.findContours(edge.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cntours = imutils.grab_contours(cntours)
# sort contours left-to-right
(cntours, _) = contours.sort_contours(cntours)
```

Basic Procedure:

5. For each bounding box, find length and width of each box to find object's dimension based on reference measurements
6. Create new image by cropping each bounding box
7. Use Orb Detection on crop image and compare it to list of inputted images
8. Image that best matches crop image is that object's "Identity"
9. Write object's identity on original image

```
for c in cntours:
  if cv2.contourArea(c) < 5000:
    continue

  orig = img.copy()
  bbox = cv2.minAreaRect(c)
  bbox = cv2.cv.boxPoints(bbox) if imutils.is_cv2() else cv2.boxPoints(bbox)
  bbox = np.array(bbox, dtype="int")

  # order the contours and draw bounding box
  bbox = perspective.order_points(bbox)
  cv2.drawContours(orig, [bbox.astype("int")], -1, (0, 0, 255), 5)

  # unpack the ordered bounding bbox; find midpoints
  (tl, tr, br, bl) = bbox
  (tMidX, tMidY) = midpoint(tl, tr)
  (bMidX, bMidY) = midpoint(bl, br)
  (lMidX, lMidY) = midpoint(tl, bl)
  (rMidX, rMidY) = midpoint(tr, br)

  cv2.circle(orig, (int(tMidX), int(tMidY)), 5, (255, 0, 0), -1)
  cv2.circle(orig, (int(bMidX), int(bMidY)), 5, (255, 0, 0), -1)
  cv2.circle(orig, (int(lMidX), int(lMidY)), 5, (255, 0, 0), -1)
  cv2.circle(orig, (int(rMidX), int(rMidY)), 5, (255, 0, 0), -1)

  # compute the Euclidean distances between the midpoints
  dX = dist.euclidean((tMidX, tMidY), (bMidX, bMidY))
  dY = dist.euclidean((lMidX, lMidY), (rMidX, rMidY))

  # use pixel_ratio
  distX = round(dX / pixel_ratio)
  distY = round(dY / pixel_ratio)

  # draw the object sizes on the image
  cv2.putText(orig, "{:.1f} mm".format(distX), (int(tMidX - 25), int(tMidY - 25)), cv2.FONT_HERSHEY_DUPLEX,1, (0, 0, 0), 1)
  cv2.putText(orig, "{:.1f} mm".format(distY), (int(rMidX + 25), int(rMidY)), cv2.FONT_HERSHEY_DUPLEX,1, (0, 0, 0), 1)
  # show the output image

  cropped_img = orig[int(tl[1] - 25):int(bl[1] + 25), int(tl[0] - 25):int(tr[0] + 25)]
  cv2_imshow(cropped_img)

  id = Identify(cropped_img, IDlist)
  if (id >= 0):
    cv2.putText(orig, img_name[id], (int(bMidX + 25), int(bMidY + 25)), cv2.FONT_HERSHEY_DUPLEX, 1, (0, 0, 0), 1)
  # show the output image
  cv2_imshow(orig)
```

```
# INPUT

img = cv2.imread("PennyTest.jpg")
# img = cv2.resize(img, (960, 1280))
```

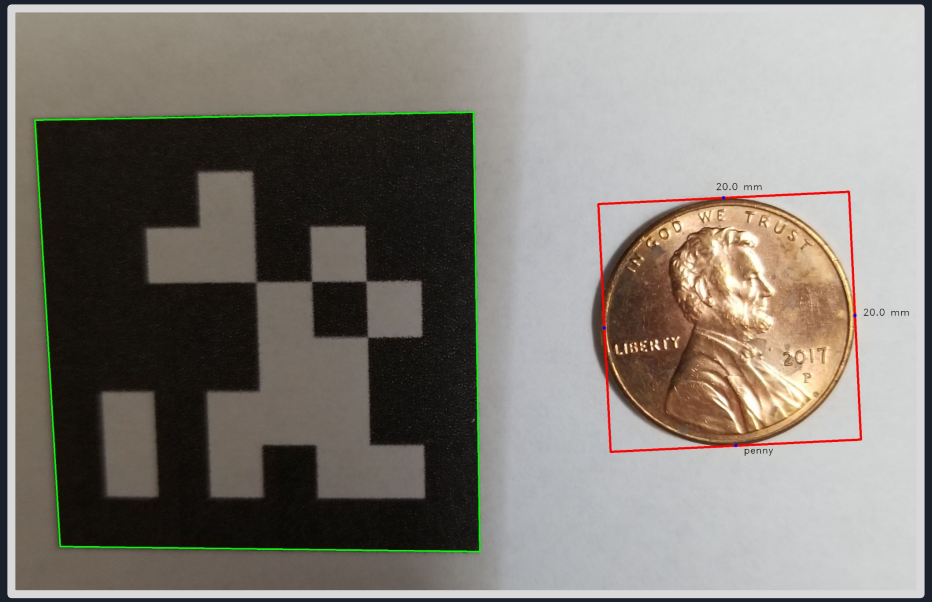
```
penny = cv2.imread("penny.jpg")
nickel = cv2.imread("nickel.jpg")
dime = cv2.imread("dime.jpg")
quarter = cv2.imread("quarter.jpg")

images = []

images.append(penny)
images.append(nickel)
images.append(dime)
images.append(quarter)

img_name = ["penny", "nickel", "dime", "quarter"]

IDlist = getID(images)
```

```python
def getID(img):
  IDList = []

  orb = cv2.ORB_create()
  for image in img:
    kp, des = orb.detectAndCompute(image,None)
    IDList.append([image,des,kp])

  return IDList
```

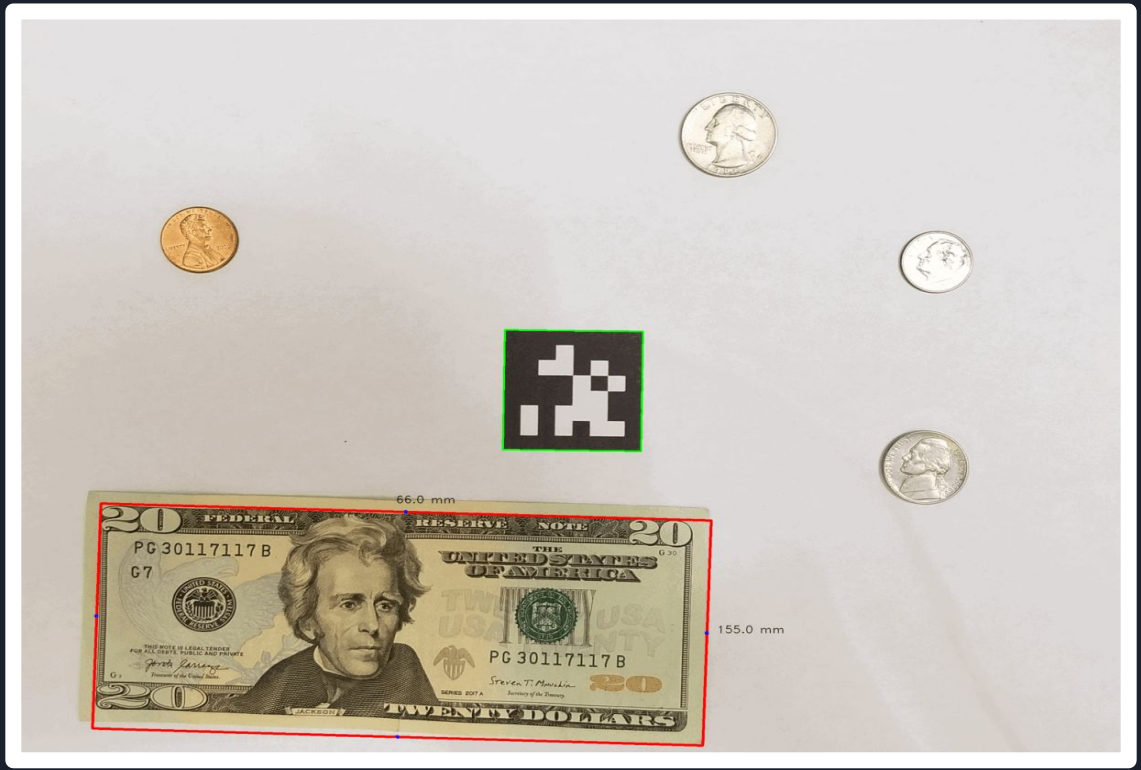
```python
def Identify(img2, IDlist):
  MIN = 10
  orb = cv2.ORB_create()

  kp2, des2 = orb.detectAndCompute(img2,None)

  BF = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
  check = []
  correct = -1
  try:
    for img, des, kp in IDlist:
      matches = BF.match(des,des2)
      img_match = cv2.drawMatches(img,kp,img2,kp2,matches,None,flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

      pts1_obs = []
      pts2_obs = []

      for m in matches:
        pts1_obs.append(kp[m.queryIdx].pt)
        pts2_obs.append(kp2[m.trainIdx].pt)

      pts1_obs = np.array(pts1_obs)
      pts2_obs  = np.array(pts2_obs)


      _, E, R, t, mask = cv2.recoverPose(pts1_obs, pts2_obs, K, distorC, K, distorC)
      mask = mask.ravel().astype(bool)
      matches_valid = [matches[i] for i in range(len(mask)) if mask[i]]
      check.append(len(matches_valid))

  except:
    pass

  print(check)
  if len(check) > 0:
    if max(check) > MIN:
      correct = check.index(max(check))


  return correct
```

Results:

- Image shows an Aruco Marker and a Penny
- Marker's length is 3.5 cm or 35 mm
- Typical Penny's length and width are 19 mm (some error)



Results for image with many objects:



Limitations:

- Image should have white background for better output
- Difficulty measuring Height of an object
- Images used for ORB feature must be clear
- Some error in measurements and in detection

Conclusion:

- Output for Identification heavily depends on input images
- Code can be improved in terms of ORB detection and Measurement
- Possible to adapt code to work with video rather than image

Contributions

- Both Thomas and Amolak brainstormed/ contributed to the ideas to make phase 1 more automated.
- Thomas focussed on the physical coding aspect of the project.
- Amolak focussed on the slides and drawing connections between phase 1 and 2.

Thank you!