**Lab 08: Augmented Reality**

**2020 Spring**

**Multi-Media & Lab**

**Name/ID: 201533661 이승수**

**Dept: Software**

**[Code]**

**<ar.py>**

#the main program

import cv2

import numpy as np

import math

from object\_module import \*

import sys

import aruco

A = [[1019.37187, 0, 618.709848], [0, 1024.2138, 327.280578], [0, 0, 1]] #hardcoded intrinsic matrix for my webcam

A = np.array(A)

obj = three\_d\_object('data/3d\_objects/low-poly-fox-by-pixelmannen.obj', 'data/3d\_objects/texture.png')

def get\_extended\_RT(A, H):

#finds r3 and appends

# A is the intrinsic mat, and H is the homography estimated

H = np.float64(H) #for better precision

A = np.float64(A)

R\_12\_T = np.linalg.inv(A).dot(H)

r1 = np.float64(R\_12\_T[:, 0]) #col1

r2 = np.float64(R\_12\_T[:, 1]) #col2

T = R\_12\_T[:, 2] #translation

#ideally |r1| and |r2| should be same

#since there is always some error we take square\_root(|r1||r2|) as the normalization factor

norm = np.float64(math.sqrt(np.float64(np.linalg.norm(r1)) \* np.float64(np.linalg.norm(r2))))

r3 = np.cross(r1,r2)/(norm)

R\_T = np.zeros((3, 4))

R\_T[:, 0] = r1

R\_T[:, 1] = r2

R\_T[:, 2] = r3

R\_T[:, 3] = T

return R\_T

def main():

marker\_colored = cv2.imread('data/m1\_flip\_new.png')

marker\_colored = cv2.resize(marker\_colored, (480,480), interpolation = cv2.INTER\_CUBIC )

marker = cv2.cvtColor(marker\_colored, cv2.COLOR\_BGR2GRAY)

cv2.namedWindow("webcam")

vc = cv2.VideoCapture(0)

h,w = marker.shape

#considering all 4 rotations

marker\_sig1 = aruco.get\_bit\_sig(marker, np.array([[0,0],[0,w], [h,w], [h,0]]).reshape(4,1,2))

marker\_sig2 = aruco.get\_bit\_sig(marker, np.array([[0,w], [h,w], [h,0], [0,0]]).reshape(4,1,2))

marker\_sig3 = aruco.get\_bit\_sig(marker, np.array([[h,w],[h,0], [0,0], [0,w]]).reshape(4,1,2))

marker\_sig4 = aruco.get\_bit\_sig(marker, np.array([[h,0],[0,0], [0,w], [h,w]]).reshape(4,1,2))

sigs = [marker\_sig1, marker\_sig2, marker\_sig3, marker\_sig4]

rval, frame = vc.read()

h2, w2, \_ = frame.shape

h\_canvas = max(h, h2)

w\_canvas = w + w2

while rval:

rval, frame = vc.read() #fetch frame from webcam

key = cv2.waitKey(20)

if key == 27:

break

canvas = np.zeros((h\_canvas, w\_canvas, 3), np.uint8) #final display

canvas[:h, :w, :] = marker\_colored #marker for reference

success, H = aruco.find\_homography\_aruco(frame, marker, sigs)

# success = False

if not success:

# print('homograpy est failed')

canvas[:h2 , w: , :] = np.flip(frame, axis = 1)

cv2.imshow("webcam", canvas )

continue

R\_T = get\_extended\_RT(A, H)

transformation = A.dot(R\_T)

augmented = np.flip(augment(frame, obj, transformation, marker, True), axis = 1) #flipped for better control

canvas[:h2 , w: , :] = augmented

cv2.imshow("webcam", canvas)

if \_\_name\_\_ == '\_\_main\_\_':

main()

**<aruco.py>**

#hepler functions to detect the aruco marker

import numpy as np

import cv2

import sys

def display(img, f = 1):

#takes an image as input and scaling factor and displays

img = scale(img, f)

cv2.imshow('dummy', img)

cv2.waitKey(0)

cv2.destroyAllWindows()

def scale(image, f):

#scales an image acc to f

h = int(image.shape[0]\*f)

w = int(image.shape[1]\*f)

return cv2.resize(image, (w,h), interpolation = cv2.INTER\_CUBIC )

def get\_bit\_sig(image, contour\_pts, thresh = 127):

ans = []

#getting all the 4 corners of the quad

a, b = contour\_pts[0][0]

c, d = contour\_pts[1][0]

e, f = contour\_pts[3][0]

g, h = contour\_pts[2][0]

for i in range(8):

for j in range(8):

#using bilinear interpolation to find the coordinate using fractional contributions of the corner 4 points

f1 = float(i)/8 + 1./16 #fraction1

f2 = float(j)/8 + 1./16 #fraction2

#finding the intermediate coordinates

upper\_x = (1-f1)\*a + f1\*(c)

lower\_x = (1-f1)\*e + f1\*(g)

upper\_y = (1-f1)\*b + f1\*d

lower\_y = (1-f1)\*(f) + f1\*(h)

x = int( (1-f2)\*upper\_x + (f2)\*lower\_x )

y = int( (1-f2)\*upper\_y + (f2)\*lower\_y )

#thresholding

if image[y][x] >= 127:

ans.append(1)

else:

ans.append(0)

return ans

def match\_sig(sig1, sig2, thresh = 62):

# print(sum([ (1- abs(a - b)) for a, b in zip(sig1, sig2)]))

if sum([ (1- abs(a - b)) for a, b in zip(sig1, sig2)]) >= 62:

return True

else:

return False

def find\_pattern\_aruco(image, aruco\_marker, sigs):

#converting image to black and white

gray = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

#adaptive thresholding for robustness against varying lighting

thresholded = cv2.adaptiveThreshold(gray, 255,cv2.ADAPTIVE\_THRESH\_MEAN\_C, cv2.THRESH\_BINARY,11,10)

h, w = aruco\_marker.shape

contours ,\_= cv2.findContours(thresholded, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

for cnt in contours :

approx = cv2.approxPolyDP(cnt, 0.01 \* cv2.arcLength(cnt, True), True)

if approx.shape[0]==4:

x1 = approx[0][0][0]

x2 = approx[1][0][0]

y1 = approx[0][0][1]

y2 = approx[1][0][1]

norm = (x1 - x2)\*\*2 + (y1 - y2)\*\*2

#constraint on minimum edge size of quad

if norm > 100:

temp\_sig = get\_bit\_sig(gray, approx)

match1 = match\_sig(sigs[0], temp\_sig)

match2 = match\_sig(sigs[1], temp\_sig)

match3 = match\_sig(sigs[2], temp\_sig)

match4 = match\_sig(sigs[3], temp\_sig)

if (match1 or match2 or match3 or match4):

dst\_pts = approx

if match1:

src\_pts = np.array([[0,0],[0,w], [h,w], [h,0]])

if match2:

src\_pts = np.array([[0,w], [h,w], [h,0], [0,0]])

if match3:

src\_pts = np.array([[h,w],[h,0], [0,0], [0,w]])

if match4:

src\_pts = np.array([[h,0],[0,0], [0,w], [h,w]])

cv2.drawContours(image, [approx], 0, (0, 0, 255), 2) #mark red outline for found marker

return src\_pts, dst\_pts, True

#reaching here implies nothing was found

return None, None, False

def find\_homography\_aruco(image, aruco\_marker, sigs):

src\_pts, dst\_pts, found = find\_pattern\_aruco(image, aruco\_marker, sigs)

H = None

if found:

H, mask = cv2.findHomography(src\_pts.reshape(-1,1,2), dst\_pts.reshape(-1,1,2), cv2.RANSAC,5.0)

if H is None:

return False, None

else:

return True, H

**<object\_module.py>**

#contains code to the .obj file and augment the object

import cv2

import numpy as np

def augment(img, obj, projection, template, color=False, scale = 4):

# takes the captureed image, object to augment, and transformation matrix

#adjust scale to make the object smaller or bigger, 4 works for the fox

h, w = template.shape

vertices = obj.vertices

img = np.ascontiguousarray(img, dtype=np.uint8)

#blacking out the aruco marker

a = np.array([[0,0,0], [w, 0, 0], [w,h,0], [0, h, 0]], np.float64 )

imgpts = np.int32(cv2.perspectiveTransform(a.reshape(-1, 1, 3), projection))

cv2.fillConvexPoly(img, imgpts, (0,0,0))

#projecting the faces to pixel coords and then drawing

for face in obj.faces:

#a face is a list [face\_vertices, face\_tex\_coords, face\_col]

face\_vertices = face[0]

points = np.array([vertices[vertex - 1] for vertex in face\_vertices]) #-1 because of the shifted numbering

points = scale\*points

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

dst = cv2.perspectiveTransform(points.reshape(-1, 1, 3), projection)#transforming to pixel coords

imgpts = np.int32(dst)

if color is False:

cv2.fillConvexPoly(img, imgpts, (50, 50, 50))

else:

cv2.fillConvexPoly(img, imgpts, face[-1])

return img

class three\_d\_object:

def \_\_init\_\_(self, filename\_obj, filename\_texture, color\_fixed = False):

self.texture = cv2.imread(filename\_texture)

self.vertices = []

self.faces = []

#each face is a list of [lis\_vertices, lis\_texcoords, color]

self.texcoords = []

for line in open(filename\_obj, "r"):

if line.startswith('#'):

#it's a comment, ignore

continue

values = line.split()

if not values:

continue

if values[0] == 'v':

#vertex description (x, y, z)

v = [float(a) for a in values[1:4] ]

self.vertices.append(v)

elif values[0] == 'vt':

#texture coordinate (u, v)

self.texcoords.append([float(a) for a in values[1:3] ])

elif values[0] == 'f':

#face description

face\_vertices = []

face\_texcoords = []

for v in values[1:]:

w = v.split('/')

face\_vertices.append(int(w[0]))

if len(w) >= 2 and len(w[1]) > 0:

face\_texcoords.append(int(w[1]))

else:

color\_fixed = True

face\_texcoords.append(0)

self.faces.append([face\_vertices, face\_texcoords])

for f in self.faces:

if not color\_fixed:

f.append(three\_d\_object.decide\_face\_color(f[-1], self.texture, self.texcoords))

else:

f.append((50, 50, 50)) #default color

# cv2.imwrite('texture\_marked.png', self.texture)

def decide\_face\_color(hex\_color, texture, textures):

#doesnt use proper texture

#takes the color at the mean of the texture coords

h, w, \_ = texture.shape

col = np.zeros(3)

coord = np.zeros(2)

all\_us = []

all\_vs = []

for i in hex\_color:

t = textures[i - 1]

coord = np.array([t[0], t[1]])

u , v = int(w\*(t[0]) - 0.0001), int(h\*(1-t[1])- 0.0001)

all\_us.append(u)

all\_vs.append(v)

u = int(sum(all\_us)/len(all\_us))

v = int(sum(all\_vs)/len(all\_vs))

# all\_us.append(all\_us[0])

# all\_vs.append(all\_vs[0])

# for i in range(len(all\_us) - 1):

# texture = cv2.line(texture, (all\_us[i], all\_vs[i]), (all\_us[i + 1], all\_vs[i + 1]), (0,0,255), 2)

# pass

col = np.uint8(texture[v, u])

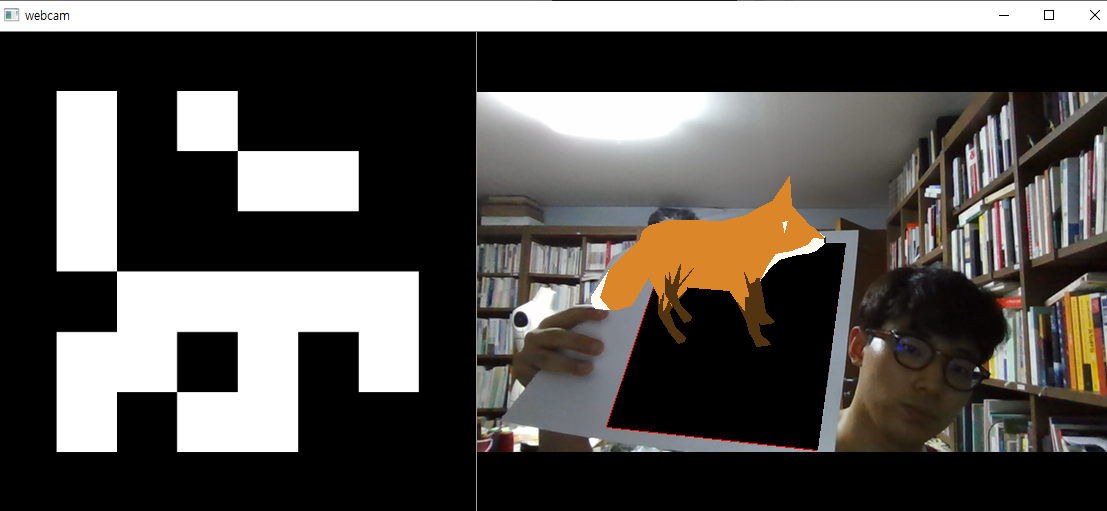
col = [int(a) for a in col]

col = tuple(col)

return (col)

**[Result]**

Screenshot capture)



File elements)

