# GitHub Repository Link

https://github.coventry.ac.uk/5062CEM/STUDENTID\_IPA

# recogniser.py

"""  
The Object Recogniser class. This class will consist of the methods that will be used to  
detect an object from a given frame.  
  
For example, we will use my staff identification card as a training image to be detected in  
the camera whenever it is shown.  
"""  
import time  
  
import cv2  
import numpy  
  
  
class Recogniser:  
 """  
 The class initializer is used to set the feature detector of the object recognition class. There  
 are two types of detectors used:  
  
 \* BRISK - Provide an explanation of BRISK  
 \* SURF - Provide an explanation of SURF  
 """  
  
 def \_\_init\_\_(self, detector):  
 if detector == "BRISK":  
 # The default parameters for BRISK are used, i.e. 30, 3, 1.0. These can be changed  
 # accordingly to increase (or decrease) the performance of the feature detector.  
 self.feature\_detector = cv2.BRISK\_create(30, 3, 1.0)  
 elif detector == "SURF":  
 # The default parameters for SURF are used, i.e. 100, 4, 3, FALSE and FALSE. These can  
 # be changed accordingly to increase (or decrease) the performance of the feature detector.  
 self.feature\_detector = cv2.xfeatures2d.SURF\_create(100, 4, 3, False, False)  
 else:  
 # If there was no feature detector set, then it will raise an exception and the application  
 # will no longer continue.  
 raise Exception("No Feature Detector")  
  
 """  
 This function is used to detect the key areas of interest, otherwise known as keypoints, form an   
 image or a frame grabbed from the camera or video.  
 """  
  
 def get\_keypoints(self, \_img):  
 tmp\_keypoints = self.feature\_detector.detect(\_img)  
 return tmp\_keypoints  
  
 """  
 This function is used to extract the descriptor from an image, or frame grabbed from the camera   
 or video. It will use the key areas of interest, otherwise known as keypoints, to form this   
 descriptor.  
 """  
  
 def get\_descriptor(self, \_img, \_keypoints):  
 tmp\_keypoints, tmp\_descriptor = self.feature\_detector.compute(\_img, \_keypoints)  
 return tmp\_keypoints, tmp\_descriptor  
  
 """  
 This function will match the descriptors that were extracted from the image, an a frame grabbed   
 from the camera or video. Depending upon the type of feature detector used, there may be two types   
 of descriptors extracted. This means that their may be two types of 'matchers', BruteForce for those   
 that are an 8-bit integer, or Flann for those that are 32-bit floats.  
  
 The matching process uses the k-nearest-neighbour algorithm, with k set as 2. These initial matches   
 are then refined using a ratio test (set to 0.7) to filter out the 'good matches'. These matches are   
 then used to determine whether the object has been found in the image or frame grabbed from the camera   
 or video.  
 """  
  
 @staticmethod  
 def match(\_d1, \_d2):  
 matcher = None  
 if \_d1.dtype == "uint8":  
 # BRISK  
 matcher = cv2.DescriptorMatcher.create(cv2.DescriptorMatcher\_BRUTEFORCE)  
 elif \_d1.dtype == "float32":  
 # SURF  
 matcher = cv2.DescriptorMatcher.create(cv2.DescriptorMatcher\_FLANNBASED)  
  
 knn\_matches = matcher.knnMatch(\_d1, \_d2, 2)  
 ratio\_thresh = 0.7  
 good\_matches = []  
 for i in range(len(knn\_matches)):  
 if knn\_matches[i][0].distance < ratio\_thresh \* knn\_matches[i][1].distance:  
 good\_matches.append(knn\_matches[i][0])  
 return good\_matches  
  
 """  
 This function will calculate the FPS in which the camera, or video file is running at. This is purely   
 for debugging purposes only; and ensures that I can see my object recogniser is running in a real-time   
 constraint. For example, if my camera feed is 30FPS, then I expect it to continue running at 30FPS.  
 """  
  
 @staticmethod  
 def calculate\_fps(\_frame\_number, \_time):  
 return int(\_frame\_number // (time.time() - \_time))  
  
 """  
 This function will detect the object from the frame grabbed from the camera or the video.  
 """  
  
 @staticmethod  
 def detect\_object(\_training\_image, \_training\_keypoints, \_frame\_keypoints, \_filtered\_matches):  
 tmp\_object = numpy.float32(  
 [\_training\_keypoints[m.queryIdx].pt for m in \_filtered\_matches]  
 )  
  
 tmp\_frame = numpy.float32(  
 [\_frame\_keypoints[m.trainIdx].pt for m in \_filtered\_matches]  
 )  
  
 try:  
 homography, mask = cv2.findHomography(tmp\_object, tmp\_frame, cv2.RANSAC, 3)  
 except cv2.error:  
 return None  
  
 height, width = \_training\_image.shape[:2]  
 boundaries = numpy.float32([  
 [0, 0],  
 [0, height],  
 [width, height],  
 [width, 0]  
 ]).reshape(-1, 1, 2)  
  
 try:  
 boundaries = numpy.int32(cv2.perspectiveTransform(boundaries, homography))  
 x, y = 0, 0  
 for a in boundaries:  
 for b in a:  
 x += b[0]  
 y += b[1]  
 except cv2.error as e:  
 print(Exception(e))  
 return None  
 except UnboundLocalError:  
 print(Exception(UnboundLocalError.\_\_str\_\_))  
 return None  
 return boundaries

# main.py

import time  
import cv2  
from recogniser import Recogniser  
  
# Create an instance of the class using the SURF feature detector  
obj\_rec = Recogniser("SURF")  
  
# Load the training image that we want to use to detect  
training\_img = cv2.imread("img.png", cv2.IMREAD\_REDUCED\_COLOR\_2)  
  
# Grab some key areas of interest from the training image  
training\_keypoints = obj\_rec.get\_keypoints(training\_img)  
  
# Extract a descriptor from the training image using the key areas of interest  
training\_keypoints, training\_descriptor = obj\_rec.get\_descriptor(training\_img, training\_keypoints)  
  
# Set the video capture method to use our in-built webcam  
cap = cv2.VideoCapture(0)  
  
# Check whether we have opened the camera or not.  
if cap.isOpened() is False:  
 raise Exception("The camera is already open.")  
  
# Change the camera settings to 1280x720 (720p) resolution  
cap.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 1280)  
cap.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 720)  
  
# Get the time when we begun the application  
start\_time = time.time()  
frame\_number = 0  
  
# Use an infinite loop to grab frames from the webcam, we can break this later on  
while True:  
 # Read a frame from the camera, and a return value on whether it is grabbing a frame  
 ret, frame = cap.read()  
 # If the frame is none (i.e. empty) then we can throw an error  
 if frame is None:  
 raise Exception("Error reading from the camera.")  
  
 # Find key areas of interest from the frame of the camera  
 keypoints = obj\_rec.get\_keypoints(frame)  
 # Generate a descriptor from the key areas of interest  
 keypoints, descriptor = obj\_rec.get\_descriptor(frame, keypoints)  
 # Perform a match between the descriptor of the training image and the frame to determine if  
 # the object can be found.  
 matches = obj\_rec.match(training\_descriptor, descriptor)  
  
 # Get the detected boundaries of the object  
 detected\_boundaries = obj\_rec.detect\_object(training\_img, training\_keypoints, keypoints, matches)  
  
 # Draw the detected boundaries onto the frame  
 cv2.polylines(frame, [detected\_boundaries], True, (0, 255, 0), 1, cv2.LINE\_AA, 0)  
  
 # Put the FPS in the top-left corner of the image  
 frame\_number += 1  
 cv2.putText(frame, str(obj\_rec.calculate\_fps(frame\_number, start\_time)), (0, 15),  
 cv2.FONT\_HERSHEY\_COMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE\_AA)  
  
 # Displays just the camera, with a bounding box around the detected image.  
 cv2.imshow("Window", frame)  
  
 # Sets a wait key for one second, and listens for ESC key to break the while loop  
 if cv2.waitKey(1) == 27:  
 # Releases the camera when the while loop has ended  
 cap.release()  
 # Destroys any windows that were created  
 cv2.destroyAllWindows()  
 # Now lets break  
 break