Virtual Keyboard in an Augmented Reality Context

Faculty of Engineering of the University of Porto

Nuno Cardoso Pedro Esteves Raúl Viana

up201706162@edu.fe.up.pt up201705160@edu.fe.up.pt up201208089@edu.fe.up.pt

**ABSTRACT**

This document aims at detailing the implementation of a virtual keyboard with augmented reality techniques. The implementation includes a program to assert the initial keyboard preparation and a second one to deal with the recognition and the display of the keyed text.

This implementation uses *OpenCV* and *MediaPipe* in *Python* and integrates image processing techniques, object recognition, camera calibration and visual information rendering. The preparation program recognizes automatically the keyboard keys’ corners, but asks the user to input the different keys’ values. The recognition program uses some *OpenCV* and *MediaPipe* functions to calculate and apply the homography between the reference image and the current image in the recognition program, to find the contours and to map the finger tip to the keys position.

**KEYWORDS**

Augmented reality, homography, virtual keyboard, feature detection and matching.

1. **INTRODUCTION**

A virtual keyboard is a software program that allows users to input text entries through alternative mechanisms than the traditional mechanical keyboard. There are a variety of situations where the user couldn’t be able to use the traditional keyboard, from disability issues to specific work environments, where touching a physical keyboard could be undesirable. In these cases could be included surgeries, where the surgeon would be forbidden to touch anything including the physical keyboard, or even workers that have their hands dirty all the time, like butchers or fishermen. The virtual keyboard could be the solution, in these specific situations, to take notes and be permitted to interact with a computer in an innovative interface.

This work focuses on developing an augmented reality keyboard solution with *OpenCV* and *MediaPipe* in *Python* that represents an alternative touchless keyboard interface.

1. **USER INTERACTION**

The implementation usage flow can be described in two parts: the first one being the *preparation* and the second one the *recognition*.

The first one is needed because it is crucial to measure the correct position of the characteristic points of the keyboard image. This step is done automatically by the program, and the user just has to insert the image path as the program argument. Next, the program asks the user to interactively map each key to its value. This is a tedious task, but it's a one-time operation for each keyboard image. By pressing the left button of the mouse on the top of the key in the image, the user specifies the position of the requested key. Pressing the right button does the same task, but indicates that the key needs to have the *“shift”* key pressed at the same time in order to be accessed. When all the keys are validated the program exports the collected information into a *JSON* file.

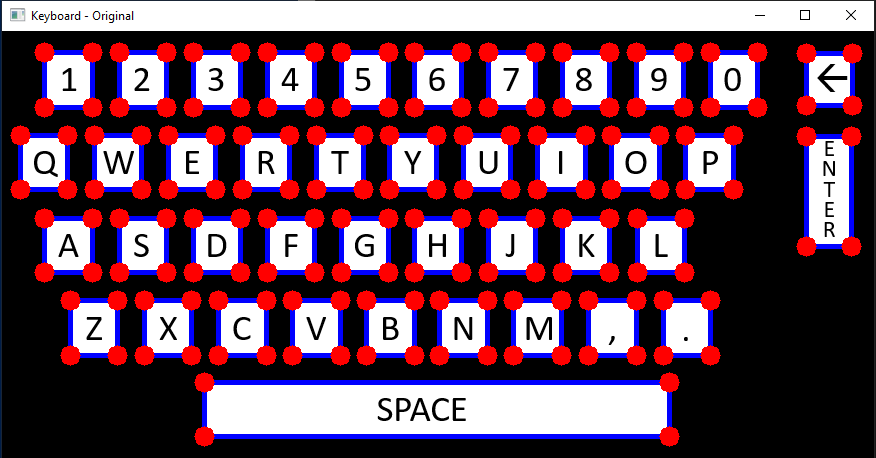
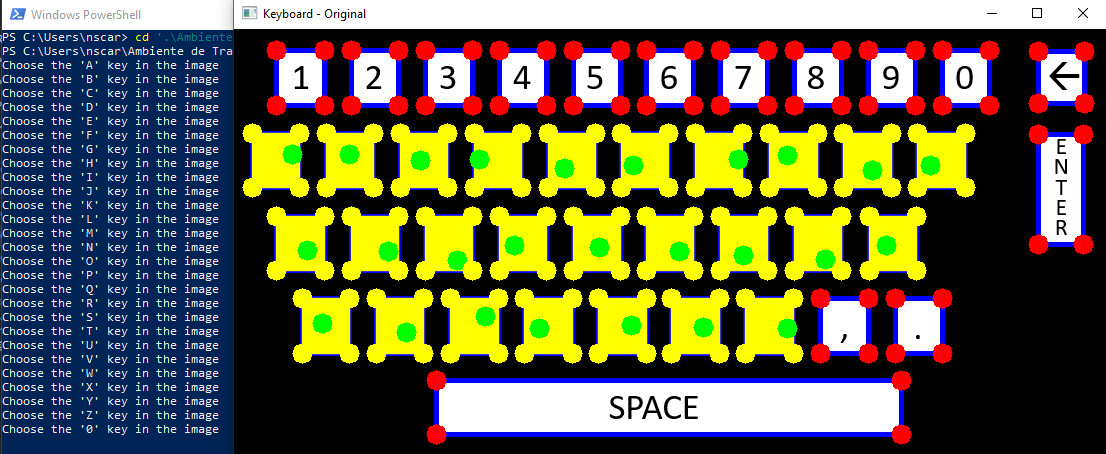


Figure 1 and 2 – Preparation Program

The second one is the proper virtual keyboard. This program starts to load the JSON file with the keyboard shape, the corners positions and the values of the keys. Then, it detects and computes the keypoints and the descriptors of the original image using *SIFT* and compares with the keypoints and descriptors of the current image. If there are enough matches, it calculates the homography between the two images. This allows us to transform the perspective, analyse the fingertip coordinates, transform them with a reverse homography and conclude if the fingertip is hovering a certain key.



Figure 3 – Keyboard (Marker) Detection

There is visual and audio feedback in the different stages of the implementation. In the preparation phase the key corners change their color, the clicked point stays marked and the key is filled with a yellow rectangle, marking it as already chosen. There is also a confirmation sound. In the recognition phase the fingertip is always marked, each time a key is activated there is a confirmation sound, its contours are highlighted, the key is drawn in the screen and the character is written in the standard output.

1. **IMPLEMENTATION**

This section describes the two main programs and shows implemented snippets of code of the different augmented reality techniques used.

1. **Semantic Model**

**list\_keys.** List of all the possible key values.

**Keyboard.** The keyboard class is the representation of the virtual keyboard. Its properties are the image name of the keyboard, its shape in an array of the type (height, width, channels) and a list of keys.

**Key.** The key class is the representation of each key of the virtual keyboard. Its properties are its value, the 2D coordinates of each of its four corners, and a boolean property indicating the need to press *“shift”* in order to be capable of pressing the key.

**Fingertip.** The Fingertip class provides the identification of the user’s finger tip location given the id of the finger used. Moreover, it identifies the key from the keyboard that the user is trying to press.

**Writer.** The Writer class is the one responsible for writing the correct character to the standard output, as well as giving the user the feedback needed to enhance user comprehension and immersion with the program. It is also responsible for counting the time needed to press a key.

**Feedback.** The Feedback class includes a series of functions used to highlight user’s interaction with the program. It includes functions to play confirmation sounds and draw color keys contours in the 2D and 3D spatial environment.

1. **Preparation Program**

**Edges and Corners Detection.** The preparation program automatically detects the keyboard and its keys’ edges through the application of the *Canny Edge Detector*. The *OpenCV* has the *Canny* function to this effect:

# Edge Detection

imgCanny = cv2.Canny(imgGrey, 50, 150, None, 3)

The *Canny Edge Detector* is an [edge detection](https://en.wikipedia.org/wiki/Edge_detection) operator that uses a multi-stage [algorithm](https://en.wikipedia.org/wiki/Algorithm) to detect a wide range of edges in images and dramatically reduces the amount of data to be processed.

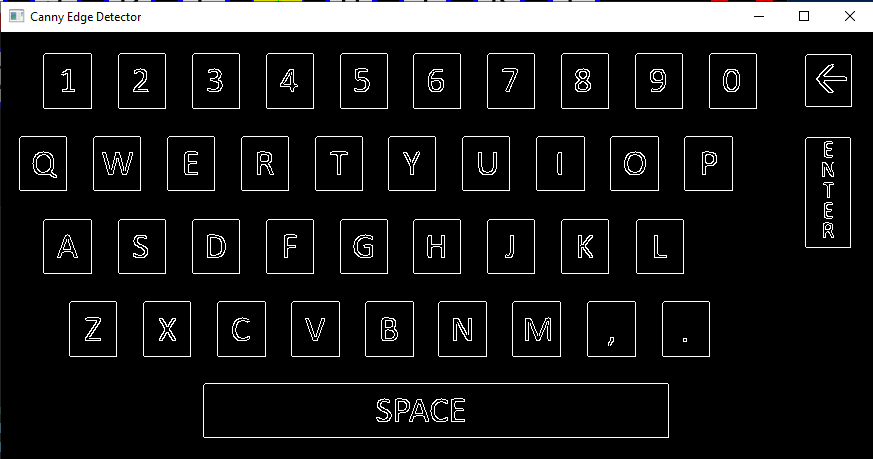


Figure 4 – Canny Edge Detection

Then, through the use of *findContours()* function from *OpenCV,* we were able to extract all the points belonging to the regions of the keys’ contours.

# Contours Detection

contours, hierarchy = cv2.findContours(imgCanny, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_NONE)

Having found the contours, then we could apply the *argmax()* and *argmin()* functions to find the extreme points of each contour, which would be the corners of each of the keys of the keyboard.

The produced data is saved using the classes mentioned above and then exported to a *JSON* file.

1. **Recognition Program**

During the recognition program, the keyboard is initially loaded from the *JSON* file. After that, a *SIFT* detector is initialized and used to detect and compute the key points and descriptors of the keyboard image used in the Preparation Program.

Then, the program connects to a video camera and starts capturing. In each frame, the *SIFT* detector follows the approach described before to detect the key points and the descriptors, which will be used by a *Flann (Fast Library for Approximate Nearest Neighbors)* based matcher to match the current frame with the keyboard image. By doing this, the program is able to detect live the presence of the keyboard in the 3D world, calculate the homography and get everything ready for the detection of the user’s hand and respective key selection. Each of the techniques is described in more detail below.

**SIFT.** The image keypoints and descriptors were computed with the *detectAndCompute()* method of *OpenCV SIFT* object.

# Initiate SIFT detector

sift = cv2.SIFT\_create()

# find the keypoints and descriptors with SIFT

kp1, des1 = sift.detectAndCompute(kb\_img,None)

SIFT (Scale Invariant Feature Transform) is an algorithm that generates feature points descriptors that may be used in posterior matching. It is considered a robust identifier of uniquely feature points. Its robustness is due to its good results in adverse situations, namely variations in image scale, different orientation or rotation degrees, variations in the image lightning and presence of noise.

Initially, we tried to implement this detection with an ORB detector, but due to the observed worse results the choice fell for the SIFT in the end.

**Homography.** Homography is a geometric transformation based on a matrix that relates a collineation from one [projective space](https://en.wikipedia.org/wiki/Projective_space) to another.

The homography concept was used in the *recognition* phase to map the coordinates from the camera’s frame image being processed with the original isometric keyboard image.

In our implementation, we used a function from *OpenCV*, *findHomography(),* with the parameter *RANSAC*.

homography, \_ = cv2.findHomography(src\_pts, dst\_pts, cv2.RANSAC,5.0)

The *RANSAC* parameter forces the function to use the *Random Sample Consensus* algorithm to compute the homography. This algorithm allows a robust fitting in the presence of a considerable number of outliers, which in real-world scenarios is very common. It iteratively calculates better-fitted homographies classifying the matched points as inliers or outliers on their distance between their coordinates and the coordinates in the original image after applying the intermediate homography. It was also calculated using an inverse homography to map correctly the fingertip coordinates with the key corners.

homography\_inverse = np.linalg.inv(homography)

ft = (int(lm.x \* width), int(lm.y \* height)) self.position = cv2.perspectiveTransform(np.float32(ft).reshape(-1,1,2), homography\_inverse)[0][0]

**Hand Detector.** The hand detection was done using the *MediaPipe* *Python* library, an open-source machine learning collection that, among others, has the Hand Tracking model. This palm detection and hand landmark model computes 21 landmarks in 3D and has multi-hand support. This model is very mature and recognizes hands in very difficult situations. It even infers landmark positions when they are outside of the image.

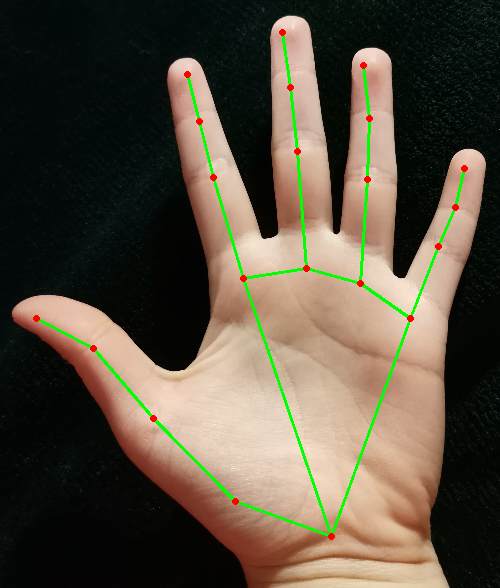
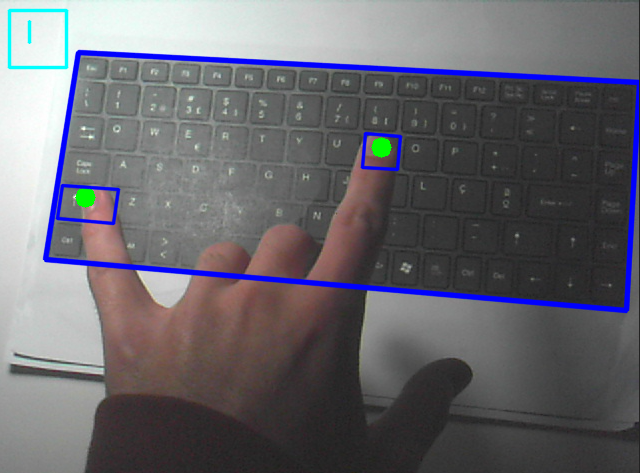


Figure 5 – Hand landmarks detected on image, *https://techtutorialsx.com*

1. **ENHANCED FEATURES**
2. **Special Keys**

The *MediaPipe* multi-hand support allowed us to implement a special key feature. The “*shift*” key modifies the behaviour of other keys being pressed at the same time. For example, when pressing “*shift”* and the “A” key, the result will be “A” and not “a”. There are also some keys which cannot be pressed unless the *“shift”* key is being pressed at the same time.

Figure 6 – Using of “shift”

Besides the *“shift”* key, the “Caps Lock” key was also implemented. Therefore, if “Caps Lock” is active the alphabetic keys will be printed in capital letters unless the *“shift”* key is also pressed.

1. **Feedback**

User has feedback for every action done. In the *preparation* phase, the input is confirmed with a sound and the printing of a green dot in the point clicked inside the key area, which is also colored. Plus, on the *recognition* phase, the user fingertip is marked with a green dot and each key pressed is confirmed with a confirmation sound and its contour is highlighted.

1. **LIMITATIONS**

Although we achieved acceptable results in our implementation, there are several limitations to the used methods, due to lighting and camera position changes, and marker occlusion. These problems can affect the SIFT detector, which sometimes can’t find sufficient marker features to match the camera frame to the keyboard image, also affecting the way we detect the key pressed by the user, because of the miscalculation of the homography in some frames.

1. **CONCLUSIONS**

This work introduced the concept of a touchless virtual keyboard where marker-based augmented reality can allow a new interface and interaction with a computer program in very specific situations.

This interface relies on applied geometric relations between the current image and the reference isometric keyboard image along with their corresponding key points coordinates.

In the future, greater feedback can be added to the program, such as drawing the keys in 3 dimensions and adding different sounds. The feature detection and matching can also be improved to prevent the frequent misdetections that damage the homography calculation.

**REFERENCES**

1. OpenCV documentation [Online]. Available: <https://docs.opencv.org/4.5.1/>
2. RVA course materials
3. “NumPy” [Online]. Available: <https://numpy.org>
4. “TensorFlow” [Online]. Available: <https://www.tensorflow.org/guide/keras?hl=pt-br>
5. “MediaPipe” [Online]. Available: <https://mediapipe.dev/>
6. “playsound” [Online]. Available: <https://pypi.org/project/playsound/>

**ANNEXES**

1. **USER MANUAL**
2. Run the following command from the root project:

`$ python -m pip install -r requirements.txt`

This will install all the required dependencies.

1. Run the *preparation* program with a keyboard image path as argument, for example:

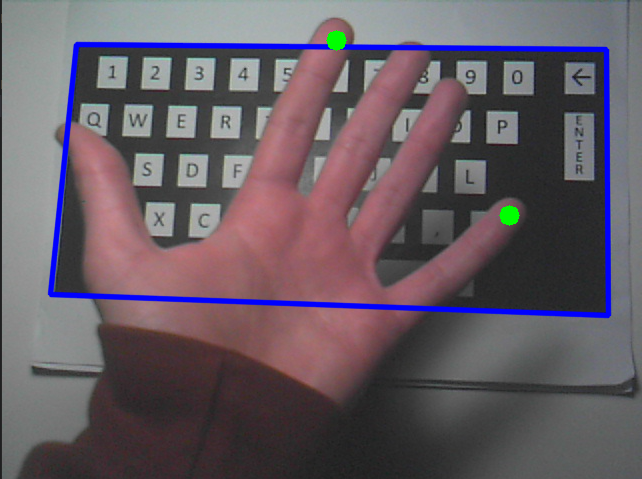
`$ python preparation.py ./keyboards/Picture1.png`

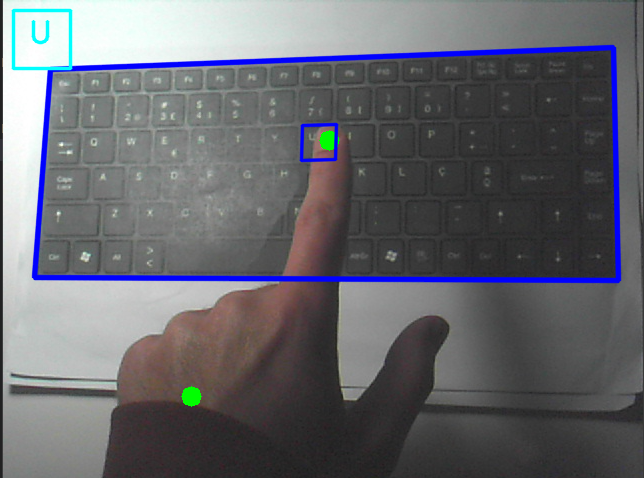
1. Run the *recognition* program with the image name used in the preparation phase as argument and enjoy. For example:

`$ python recognition.py Picture1.png

1. **PROGRAM EXECUTION IMAGES**

Example using the virtual keyboard with Caps Lock active.

Example of the hand detection.

Example using the virtual keyboard with Caps Lock inactive.

1. **IMPLEMENTED CODE**

requirements.txt

opencv-python # OpenCV Libray

mediapipe

pyautogui # Force Key Pressing

playsound == 1.2.2 # Play Sounds

Feedback.py

from playsound import playsound # Install playsound version 1.2.2

import cv2

import numpy as np

SOUNDS\_PATH = '../sounds/'

class Feedback:

# Plays a confirmation sound

@staticmethod

def playConfirmSound():

playsound(SOUNDS\_PATH + 'confirm.mp3')

# Draws a key's contours in the 3D environment

@staticmethod

def drawKey3D(img, corners3D):

img = cv2.polylines(img,[np.int32(corners3D)],True,255,2, cv2.LINE\_AA)

# Draws a key's contours in the 2D and 3D environment

@staticmethod

def drawKey(img, key, corners3D):

Feedback.drawKey3D(img, corners3D)

h = key.corners[3][1] - key.corners[0][1]

w = key.corners[1][0] - key.corners[0][0]

if len(key.value)\*12 > w:

w = len(key.value)\*30

img = cv2.rectangle(img, (10, 10), (int(w), int(h)), (255, 255, 0), 2, cv2.LINE\_AA)

cv2.putText(img, key.value, (int((w-len(key.value)\*15) / 2), 42), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (255, 255, 0), 2, cv2.LINE\_AA)

# Draws text to indicate that Caps Lock is active

@staticmethod

def drawCapsLock(img):

cv2.putText(img, "CAPS", (img.shape[1] - 140, 60), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 255), 2, cv2.LINE\_AA)

Fingertip.py

import numpy as np

import mediapipe as mp

import cv2

class Fingertip:

# Class Constructor: Needs an Object of the type keyboard

def \_\_init\_\_(self, keyboard, id):

self.\_keyboard = keyboard

mpHands = mp.solutions.hands

self.\_hands = mpHands.Hands(static\_image\_mode=False,

max\_num\_hands=2,

min\_detection\_confidence=0.5,

min\_tracking\_confidence=0.5)

self.\_position = (0.0, 0.0)

self.\_current\_key = None

self.\_finger\_id = id

# Property Hands Getter

@property

def hands(self):

return self.\_hands

# Property Keyboard Getter

@property

def keyboard(self):

return self.\_keyboard

# Property Position Getter

@property

def position(self):

return self.\_position

@position.setter

def position(self, v):

self.\_position = v

# Property Current Key

@property

def current\_key(self):

return self.\_current\_key

@current\_key.setter

def current\_key(self, v):

self.\_current\_key = v

# Property Finger Id

@property

def finger\_id(self):

return self.\_finger\_id

@finger\_id.setter

def finger\_id(self, v):

self.\_finger\_id = v

# Analyse fingertip's coordinate

def analyseFingertipCoordinate(self, frame, homography):

frameRGB = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

results = self.hands.process(frameRGB)

height, width, \_ = frameRGB.shape

if results.multi\_hand\_landmarks:

for handLms in results.multi\_hand\_landmarks:

for id, lm in enumerate(handLms.landmark):

if id == self.finger\_id:

if Fingertip.isInsideImage(lm.x, width, lm.y, height) and (not homography is None) and homography.ndim >= 2:

homography\_inverse = np.linalg.inv(homography)

ft = (int(lm.x \* width), int(lm.y \* height))

self.position = cv2.perspectiveTransform(np.float32(ft).reshape(-1,1,2), homography\_inverse)[0][0]

cv2.circle(frame, ft, 2, (0,255,0), cv2.LINE\_AA)

else: self.\_position = (0.0, 0.0)

#is finger over a key?

return self.isFingerOverKey()

# Check if the fingertip is inside a key

def isFingertipInsideKey(self, corners): # Assuming the keys are squares

if(len(corners) != 0):

return corners[0][0] <= self.position[0] <= corners[1][0] and corners[0][1] <= self.position[1] <= corners[3][1]

else:

return False

# Check if the finger is over a key

def isFingerOverKey(self):

if self.current\_key != None:

if (not self.current\_key is None) and self.isFingertipInsideKey(self.current\_key.corners): # Still inside the same key

return True, self.current\_key

found = False

# Check if fingertip is inside other keys

for key in self.keyboard.keys:

corners = key.corners

if self.isFingertipInsideKey(corners):

found = True

self.current\_key = key

break

# Resetting key if not found

if not found:

self.current\_key = None

return found, self.current\_key

@staticmethod

def isInsideImage(x, width, y, height,):

x\_position = x \* width

y\_position = y \* height

return (x\_position > 0 and x\_position < width) and (y\_position > 0 and y\_position < height)

Keyboard.py

import json

# Constants

JSON\_FILE\_PATH = "../files/"

KEYBOARD\_IMAGES\_PATH = "../keyboards/"

class Keyboard:

# Class constructor: Needs the Image Name

def \_\_init\_\_(self, image\_name, shape = [], keys = []):

self.\_image\_name = image\_name

self.\_shape = shape

self.\_keys = keys

# Keyboard Shape Property (Height, Length, Channels)

# Gets the shape property

@property

def shape(self):

return self.\_shape

# Sets the shape property

@shape.setter

def shape(self, v):

self.\_shape = [int(\_v) for \_v in v]

# Keyboard Keys Property. It should be an array of Key objects

# Gets the keys property

@property

def keys(self):

return self.\_keys

# Sets the keys property

@keys.setter

def keys(self, v):

self.\_keys = v

# Keyboard Image Name Property. It should be a string

# Gets the image\_name property

@property

def image\_name(self):

return self.\_image\_name

# Sets the image\_name property

@image\_name.setter

def image\_name(self, v):

self.\_image\_name = v

# OTHER CLASS METHODS

# Gets the keyboard image size (length, height)

def getSize(self):

return (self.shape[1], self.shape[0])

# Gets the keyboard image path

def getImagePath(self):

return KEYBOARD\_IMAGES\_PATH + self.image\_name

# Converts the Keyboard class into the JSON format, and store it in a file

def toJSON(self):

json\_path = JSON\_FILE\_PATH + self.image\_name + ".json"

try:

f = open(json\_path, "x") # Tries to create the file

except FileExistsError:

f = open(json\_path, "w") # If the file already exists, it overwrites the current file

f.write(json.dumps({ "shape": [str(shape) for shape in self.shape], "keys": [key.toJSON() for key in self.keys] }))

f.close()

# Loads the Keyboard class from a JSON file

def loadFromJSON(self):

f = open(JSON\_FILE\_PATH + self.image\_name + ".json") # Opens the file

dict = json.load(f)

keys = [Key(key['value'], key['needs\_shift'], key['corners']) for key in dict['keys']]

self.shape = dict['shape']

self.keys = keys

class Key:

# Class Constructor: Needs the Key value

def \_\_init\_\_(self, value, needs\_shift = False, corners = []):

self.\_value = value

self.\_needs\_shift = needs\_shift

self.\_corners = [list(map(float, sublist)) for sublist in corners]

# Key Value Property. It should be a char value

# Gets the value property

@property

def value(self):

return self.\_value

# Sets the value property

@value.setter

def value(self, v):

self.\_value = v

# Key Corners Property. It should be an array of pairs of floats, with the key corners coordinates

# Gets the corners property

@property

def corners(self):

return self.\_corners

# Sets the corners property

@corners.setter

def corners(self, v):

self.\_corners = v

# Key Needs Shift Property. It should be a boolean

# Gets the needs\_shift property

@property

def needs\_shift(self):

return self.\_needs\_shift

# Sets the needs\_shift property

@needs\_shift.setter

def needs\_shift(self, v):

self.\_needs\_shift = v

# OTHER CLASS METHODS

# Convert the Key class into the JSON format

def toJSON(self):

return {"value": self.value, "needs\_shift": self.needs\_shift, "corners": [[str(corner[0]), str(corner[1])] for corner in self.corners]}

# Operator ==

def \_\_eq\_\_(self, other):

return isinstance(other, Key) and self.value == other.value

# List of Keys accepted by our program

list\_keys = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M", "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "ENTER", "SPACE", "BACKSPACE", "CAPSLOCK", "SHIFT", ",", ".", "|", ">"]

Preparation.py

import cv2

import sys

from keyboard import Keyboard, Key, list\_keys, KEYBOARD\_IMAGES\_PATH

import pyautogui

from feedback import Feedback

# Draw corners of a key

def drawCorners(vertices, img, color):

cv2.circle(img, vertices[0], 2, color, cv2.LINE\_AA)

cv2.circle(img, vertices[1], 2, color, cv2.LINE\_AA)

cv2.circle(img, vertices[2], 2, color, cv2.LINE\_AA)

cv2.circle(img, vertices[3], 2, color, cv2.LINE\_AA)

# Get the corners of a key. Assumes the image is in a front view

def getCorners(contourn):

extLeft = tuple(contourn[contourn[:, :, 0].argmin()][0])

extRight = tuple(contourn[contourn[:, :, 0].argmax()][0])

extTop = tuple(contourn[contourn[:, :, 1].argmin()][0])

extBot = tuple(contourn[contourn[:, :, 1].argmax()][0])

point1 = (extLeft[0], extTop[1])

point2 = (extRight[0], extTop[1])

point3 = (extRight[0], extBot[1])

point4 = (extLeft[0], extBot[1])

return [point1, point2, point3, point4]

# Check if the image click is inside a key's corners

def isInsideCorners(point, extLeftTop, extRightBot):

return point[0] >= extLeftTop[0] and point[0] <= extRightBot[0] and point[1] >= extLeftTop[1] and point[1] <= extRightBot[1]

# Draw the click in a key

def drawClick(clickPoint, img, corners, colorClick, colorCorners):

for cs in corners:

if isInsideCorners(clickPoint, cs[0], cs[2]):

cv2.rectangle(img, cs[0], cs[2], colorCorners, -1, cv2.LINE\_AA)

cv2.circle(img, clickPoint, 2, colorClick, cv2.LINE\_AA)

cv2.circle(img, cs[0], 2, colorCorners, cv2.LINE\_AA)

cv2.circle(img, cs[1], 2, colorCorners, cv2.LINE\_AA)

cv2.circle(img, cs[2], 2, colorCorners, cv2.LINE\_AA)

cv2.circle(img, cs[3], 2, colorCorners, cv2.LINE\_AA)

return [[c[0], c[1]] for c in cs]

return []

def main():

# Check if the user specified the keyboard image name

if sys.argv.\_\_len\_\_() <= 1:

print("Error: You need to specify the keyboard image name!")

return

img\_name = str(sys.argv[1]) # Get name of the Keyboard

img\_path = KEYBOARD\_IMAGES\_PATH + img\_name

global img

img = cv2.imread(img\_path) # Read the image

# Check if the image exists

if not hasattr(img, "\_\_len\_\_"):

print("Error: Invalid image!")

return

# Covert to grey scale

imgGrey = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Edge Detection

imgCanny = cv2.Canny(imgGrey, 50, 150, None, 3)

# Contours Detection

contours, hierarchy = cv2.findContours(imgCanny, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_NONE)

cv2.drawContours(img, contours, -1, (255, 0, 0), 3)

# Get vertices for all contours

global corners

corners = []

for c in contours:

points = getCorners(c)

drawCorners(points, img, (0, 0, 255))

corners.append(points)

# Get the key coordinates

global current\_key

keys = []

for key in list\_keys:

print("Choose the '" + key + "' key in the image")

# Creates a key and stores it in the current\_key variable

# The other values of the key will be set inside the selectKey callback

current\_key = Key(value=key)

cv2.imshow("Keyboard - Original", img)

# Sets the Mouse Callback

cv2.setMouseCallback("Keyboard - Original", selectKey)

# Waits for a key pressing

cv2.waitKey(0)

keys.append(current\_key)

# Create and store the keyboard

keyboard = Keyboard(img\_name, [value for value in img.shape], keys)

keyboard.toJSON()

# Report purposes

# cv2.imshow("Canny Edge Detector", imgCanny)

# cv2.waitKey(0)

cv2.destroyAllWindows()

# Callback to Select Keys

def selectKey(event, x, y, flags, param):

if event == cv2.EVENT\_LBUTTONDOWN or event == cv2.EVENT\_RBUTTONDOWN:

# Get corners for the click point

points = drawClick((x, y), img, corners, (0, 255, 0), (0, 255, 255))

# Check if the user pressed the correct key

if len(points) <= 0:

return

current\_key.corners = points # Set the current key corners

if event == cv2.EVENT\_RBUTTONDOWN:

current\_key.needs\_shift = True

Feedback.playConfirmSound()

pyautogui.press("enter") # Force a key pressing

if \_\_name\_\_ == "\_\_main\_\_":

main()

Recognition.py

import cv2

import numpy as np

import sys

from keyboard import Keyboard

from fingertip import Fingertip

from writer import Writer, Feedback

# CONSTANTS

KNN\_MATCH\_K = 2

ESC\_KEY = 27

BITWISE\_AND\_CONST = 0xFF

FLANN\_INDEX\_KDTREE = 1

MIN\_MATCH\_COUNT = 10

# Shows a camera frame

def showFrame(frame):

cv2.imshow("Camera", frame)

return cv2.waitKey(1)

def main():

# Check if the user specified the keyboard image name

if sys.argv.\_\_len\_\_() <= 1:

print("Error: You need to specify the keyboard name!")

return

kb\_name = str(sys.argv[1]) # Get name of the Keyboard

# Create a keyboard by loading it from the JSON file

keyboard = Keyboard(kb\_name)

try:

keyboard.loadFromJSON()

except FileExistsError:

print("Error: File doesn't exist!")

return -1

fingertip = Fingertip(keyboard, 8)

fingertipAux = Fingertip(keyboard, 20)

writer = Writer()

kb\_img = cv2.imread(keyboard.getImagePath(), 0)

# Initiate SIFT detector

sift = cv2.SIFT\_create()

# find the keypoints and descriptors with SIFT

kp1, des1 = sift.detectAndCompute(kb\_img,None)

index\_params = dict(algorithm = FLANN\_INDEX\_KDTREE, trees = 5)

search\_params = dict(checks = 50)

flann = cv2.FlannBasedMatcher(index\_params, search\_params)

image\_capture = cv2.VideoCapture(0)

dst = []

capslocked = False

while True:

\_, frame = image\_capture.read() # Read Camera frame

kp2, des2 = sift.detectAndCompute(frame,None)

if not hasattr(kp2, '\_\_len\_\_') or len(kp2) <= 0 or KNN\_MATCH\_K > len(des2): # If the image is black, skip frame

showFrame(frame)

continue

matches = flann.knnMatch(des1,des2,KNN\_MATCH\_K)

# Store all the good matches as per Lowe's ratio test.

good = []

for m,n in matches:

if m.distance < 0.7\*n.distance:

good.append(m)

if len(good)>MIN\_MATCH\_COUNT:

src\_pts = np.float32([ kp1[m.queryIdx].pt for m in good ]).reshape(-1,1,2)

dst\_pts = np.float32([ kp2[m.trainIdx].pt for m in good ]).reshape(-1,1,2)

homography, \_ = cv2.findHomography(src\_pts, dst\_pts, cv2.RANSAC,5.0)

h,w,\_ = keyboard.shape

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

try:

dst = cv2.perspectiveTransform(pts,homography)

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

except cv2.error:

pass

# The Perspective Transform must return an array with 4 points

if len(dst) == 4:

found, key = fingertip.analyseFingertipCoordinate(frame, homography)

\_, keyAux = fingertipAux.analyseFingertipCoordinate(frame, homography)

if found:

writer.processKeyAux(frame, keyAux, homography)

capslocked = writer.processKey(frame, key, homography)

else:

writer.key = None

if capslocked:

Feedback.drawCapsLock(frame)

else:

print( "Not enough matches are found - {}/{}".format(len(good), MIN\_MATCH\_COUNT) )

# Stops the program when the user presses the ESC key

if showFrame(frame) & BITWISE\_AND\_CONST == ESC\_KEY:

break

image\_capture.release()

cv2.destroyAllWindows()

if \_\_name\_\_ == "\_\_main\_\_":

main()

Writer.py

import time

import numpy as np

import cv2

from feedback import Feedback

TIME\_PROCESS\_KEY = 0.5 # 1 second input to process key

class Writer:

def \_\_init\_\_(self):

self.\_key = None

self.\_time\_processing = time.time()

self.\_text = ""

self.\_capslocked = False

self.\_shiftlocked = False

# Key Property

@property

def key(self):

return self.\_key

@key.setter

def key(self, v):

self.\_key = v

# Time Processing Property

@property

def time\_processing(self):

return self.\_time\_processing

@time\_processing.setter

def time\_processing(self, v):

self.\_time\_processing = v

# Text Property

@property

def text(self):

return self.\_text

@text.setter

def text(self, v):

self.\_text = v

# Capslocked Property

@property

def capslocked(self):

return self.\_capslocked

@capslocked.setter

def capslocked(self, v):

self.\_capslocked = v

# shiftlocked Property

@property

def shiftlocked(self):

return self.\_shiftlocked

@shiftlocked.setter

def shiftlocked(self, v):

self.\_shiftlocked = v

# Get char to write for keys

def getText(self, key):

if key.value == 'CAPSLOCK':

self.capslocked = not self.capslocked

return self.text + ""

values = {

'SPACE': self.text + ' ',

'ENTER': self.text + '\n',

'BACKSPACE': self.text[:-1]

}

if (((not self.capslocked) and (not self.shiftlocked)) or (self.capslocked and self.shiftlocked)) and len(key.value) == 1 and key.value.isalpha():

return values.get(key.value, self.text + chr(ord(key.value) + 32))

return values.get(key.value, self.text + key.value)

# Process key detection

def processKey(self, frame, key, homography):

if (self.key != key):

self.key = key

self.\_time\_processing = time.time()

if time.time() - self.time\_processing > TIME\_PROCESS\_KEY and ((key.needs\_shift and self.shiftlocked) or not key.needs\_shift):

self.text = self.getText(key)

self.time\_processing = time.time()

Feedback.playConfirmSound()

print(self.text)

self.key = None

try:

pts = np.float32([ key.corners[0],key.corners[1],key.corners[2],key.corners[3] ]).reshape(-1,1,2)

dst = cv2.perspectiveTransform(pts,homography)

Feedback.drawKey(frame, key, dst)

except cv2.error:

pass

return self.capslocked

# Process key detection

def processKeyAux(self, frame, keyAux, homography):

if keyAux and keyAux.value == "SHIFT":

ptsShift = np.float32([ keyAux.corners[0],keyAux.corners[1],keyAux.corners[2],keyAux.corners[3] ]).reshape(-1,1,2)

dstShift = cv2.perspectiveTransform(ptsShift,homography)

Feedback.drawKey3D(frame, dstShift)

self.shiftlocked = True

else: self.shiftlocked = False