Universidad Autónoma de Madrid

Escuela Politécnica Superior

23|24

Bachelor thesis

Hand tracking for typing pattern detection and optimization in real time

Matías Jiménez Segura

Escuela Politécnica Superior
Universidad Autónoma de Madrid
C\Francisco Tomás y Valiente nº 11

excelencia Campus Internacional UAM CSIC

www.uam.es

UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Bachelor as Ingeniería Informática Modalidad Bilingüe

BACHELOR THESIS

Hand tracking for typing pattern detection and optimization in real time

Hand tracking para la detección y optimización de patrones de tecleo en tiempo real

Author: Matías Jiménez Segura
Advisor: Carlos García Saura
Speaker: Eduardo Serrano Jerez

junio 2024

Matías Jiménez Segura
Hand tracking for typing pattern detection and optimization in real time

Matías Jiménez Segura
C\ Francisco Tomás y Valiente Nº 11

PRINTED IN SPAIN

Resumen

En el uso diario que hacemos del ordenador, la interfaz por excelencia es el teclado, que nos acompaña desde los inicios de la mecanografía y la computación. Lo que muchas personas desconocen o no tienen en cuenta es que el teclado está dividido verticalmente en zonas para ser pulsadas de manera óptima cada una con un dedo de la mano. Así pues, para optimizar la escritura no es suficiente saber qué teclas se pulsan para formar una palabra, sino también identificar con qué dedo se pulsa cada letra y cómo afecta esto a la velocidad de escritura.

Esto puede ser analizado a través de la sincronización de imágenes que muestren las manos durante el tecleo y las señales de teclado generadas. Sobre estas imágenes es posible aplicar *hand tracking* para obtener la información de la posición de las manos en cada instante. Hemos implementado este análisis a través de la librería 'MediaPipe', desarrollada por Google. Con el objetivo de utilizar la menor cantidad de recursos posibles y no depender de tener una cámara externa al ordenador, se ha usado un dispositivo impreso en 3D con un espejo que se coloca en la *webcam* del ordenador, de manera de que el vídeo capturado por la cámara enfoca directamente al teclado.

A lo largo del desarrollo del proyecto se han encontrado varios problemas. La velocidad de procesado del *hand tracking* de MediaPipe no es suficiente para capturar de manera precisa los movimientos de las manos. Para resolver este problema ha sido el procesado de un subconjunto de imágenes asociables a un evento de teclado. Otro problema es la diferencia entre el instante en el que la *webcam* captura la imagen en que se pulsa la tecla y el instante en que se captura dicho evento. Para solventar esto, se ha implementado un sistema que captura imágenes antes y después de cada evento de teclado, con el fin de poder analizar los movimientos de las manos.

Por último, se comparan los resultados obtenidos con los estudios más recientes, destacando que es posible obtener resultados representativos usando muchos menos recursos, y proponiendo nuevos enfoques para mejorar el sistema.

Palabras clave

Hand tracking, mecanografía, MediaPipe, ciclo cerrado, evento de teclado, *webcam*, *effective input*, interacción humano-máquina

Abstract

On the daily use of computer, keyboard is the main interface and has been present since the inception of typing and computation. What most people are not aware of or don't take into account is that keyboards are vertically divided in a way that each area is optimally designed to be clicked with a different finger. Therefore, to optimize typing patterns and analyze how they affect to the typing speed is not enough to know the key which is pressed, but also with which finger it has been pressed.

This can be analyzed through synchronization of images that show hands during typing and the generated keyboard signals. Hand tracking can be applied to those images to obtain information about the hands position at each moment. We have implemented this analysis through the library 'MediaPipe', developed by Google. In order to use as less resources as possible and making an inexpensive system, a mirror device has been used. This mirror device is a 3D design with a mirror which is placed upon the laptop webcam so that the captured video focuses on the keyboard.

Along the development of the project, various problems were identified: the processing rate of MediaPipe hand tracking is not enough to capture all hand movements, which are often too fast. To solve this, only the frames that are associable to a keyboard event are processed. Another obstacle encountered was that there is not a fixed delay between the instant in which the camera captures the frame of the press and the instant in which the keyboard event is captured. As a solution, more than one frame is associated to a keyboard event, so that the movement of the hands can be studied.

Finally, the obtained results are compared with the state of the art, highlighting that is possible to obtain representative outputs using less resources, and proposing system improvements for the future.

Keywords

Hand tracking, typing, MediaPipe, closed-loop, keyboard event, webcam, effective input, human-computer interaction

Table of Contents

Lists

List of algorithms

List of codes

List of figures

1

Introduction and State of the Art

1.1. Introduction

1.1.1. Motivation

In the digital age, efficient and accurate typing is key to good productivity and communication. Despite the regular and daily use of keyboards, most people don't know that there is an optimal way of typing through a keyboard, as each key has an optimal finger to be pressed with. "Column-wise, each key is assigned to one finger. Each finger has a home position in the middle row to which it returns after pressing a key" [2]. Not following these rules can be reflected in typing too slow, or in having many typing errors. While traditional typing tests and tutors [1] only provide feedback based on the accuracy and speed at which the user types, they do not have the capability to give feedback about the user's hand and finger positions.

Hand tracking technology can offer us an approach to improving typing efficiency by providing detailed insights into the user's hand and finger movement. Therefore, by analyzing which keys are being pressed incorrectly, users can gain a deeper understanding of the errors they make while typing. This could be done not only with the keys, but also with the movements, for example, a report could inform the user that he/she is pressing incorrectly the keys when the finger comes after being at 'f'.

Therefore, the motivation behind this project is to use hand tracking together with keyboard signals to enhance typing analysis and feedback, contributing to more precise, efficient and ergonomic typing practices.

[1] `https://monkeytype.com`, `https://www.keybr.com/es/index`

1.1.2. Objectives

The objective of this project is to develop an innovative system to provide feedback on hand and finger position, identifying typing errors and offering personalized recommendations for optimal key pressing techniques. The idea is that anyone can access the system in an inexpensive way, as no external devices as cameras or other sensors might be needed.

Functional Requirements

- The system must use MediaPipe hand tracking to allow the user get his hands coordinates.
- The system must be able to associate the processed frames to a keyboard event if it is within the range of time before the event.
- The system must detect key presses even when the text is not being typed in the terminal.
- The user must do a completely manual calibration of the keyboard coordinates for each key.
- The system must show the area accepted for each key.
- The system must be able to create a report personalized for each execution showing:
  - visual representations and accuracy percentages,
  - representation of presses of the different keys,
  - the most missed keystrokes.

Non-Functional Requirements

- The system must liberate all resources correctly when it finishes.
- The system must not crash if operates for long periods of time.
- The system must be compatible with macOS, GNU/Linux and Windows.

1.2. State of the art

The technology we are going to use can be divided in two big groups: Hand tracking and Key detection. Apart from this, we identified the possibility of using an external mirror device so that the computer webcam focuses on the keyboard.

1.2.1. Hand Tracking History and Applications

History

Hand tracking is a well established field of study with relevant works dating back to the 1990s [3, 4]. In these first steps of hand tracking, making the detection at a fast rate was not possible yet, as the devices were not fast enough to process all the data, but nowadays, this is possible with virtually any

device achieving a low-medium fps rate. In powerful devices it is possible to do it at a higher rate, making the detection of the movements more precise than the one in a standard device.

Nowadays, hand tracking is heavily used in Augmented Reality (AR) and Virtual Reality (VR). One of the foremost authorities of the field, Oculus, implemented hand tracking on their devices [5] with successful results. It can be used whether for typing, playing video-games, interact with a virtual interface or at education, teaching the user how to perform a task, so that it provides a more immersive experience than using remote controls... there are plenty of functionalities that hand tracking can be applied to.

Among other potential applications of hand tracking technology is teaching/training in different contexts such as learning to play the piano [2], which highlights the relevance of this approach in many different problems.

Technology

There is a variety of libraries developed in python, javascript and other languages which implement hand tracking, but the most advanced, with a regular maintenance and good reliability is the one created and used by Google: 'MediaPipe' [6]. This library can do many other things as audio processing, body tracking or facial recognition, but we are going to focus on the feature that recognizes hands and returns different parameters, as the coordinates of different points of the hand, or the probability of the detection to be a hand [7]. This library, on a standard computer, can process video at 12-14 fps more or less, which might not be enough to process hand tracking in real-time, as the movements of the hand might be faster on the average user. The coordinates calculated by MediaPipe are shown in Figure 1.1.

Application in typing

There have been many experiments to relate hand coordinates and position with typing, specially to develop keyboards on plain flat surfaces [1, 3]. On this first article, hand tracking was not implemented with bare image analysis, but with advanced infrared sensors that receive reflections from some points of two dedicated gloves developed at [9]. This can be really useful in AR/VR, as the user will see a keyboard projected on a surface, but will not have the possibility to literally touch it. As there is no physical keyboard, the problem is to correlate a frame of the camera with a possible virtual click on a key or to nothing. For this, the members developed a neural network with the same basis as an automatic speech recognition, as they have strong similarities.

[2] https://youtu.be/10-HAJbtLtg?t=580

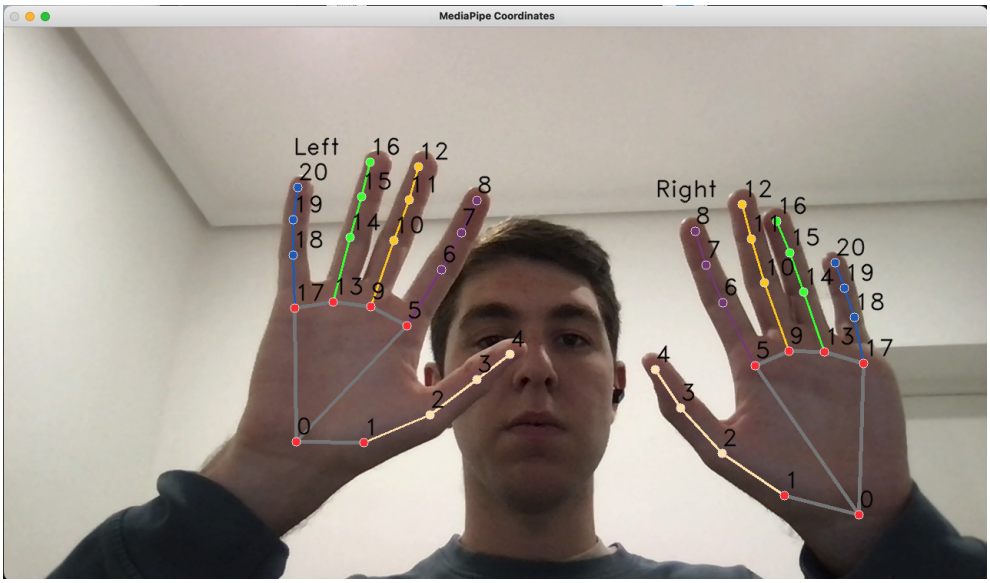


Figure 1.1: Figure which shows the hand coordinates detected by MediaPipe. Reference implementation from [8].

Also, some alternative methods have been developed. [10] explains a method in which depending on the finger the user touches with his thumb, the letter added to the text is one or another. For each finger there are some associated keys and all possible combinations are stored and discarded when more letters are typed.

1.2.2. Keyboard Events Detection

Key detection is something that might look trivial at first, but it is certainly not. There is a library in python called "pynput", which allows to manage the different events on the computer, as the mouse clicks or key pulses, but it can also simulate them. This process would need to be approved by the user, as it could be used for malicious purposes, for example, a key logger which tries to steal the users' passwords or secrets.

The part of the library we are going to use is "pynput.keyboard.listener", and it allows to launch a thread. When you create the listener, you specify what the system must do when a click or a release occur, which is implemented as a callback function. Since it is a single thread, it is recommended not to perform heavy computations or lengthy tasks within the callback function. Instead, it is advisable to execute them on a separate thread.

1.2.3. Mirror device

During COVID-19 quarantine times, many groups such as teachers, students or workers encountered difficulties in sharing what they were writing, specially in video calls. Some people were trying to find

temporary solutions so that their laptop camera could show the content of a paper over the keyboard, like Michael Peschin in [11]. It was then devised a 3D-design for a device with a small mirror which is placed onto a laptop webcam. This device can be either rigid [3] or flexible [4]. This mirror focuses the webcam on the keyboard, allowing users to display papers o anything necessary just by placing it on top of the keyboard. The combination of this device with hand tracking is very relevant, as it allows out system not to use any other external device like a camera.

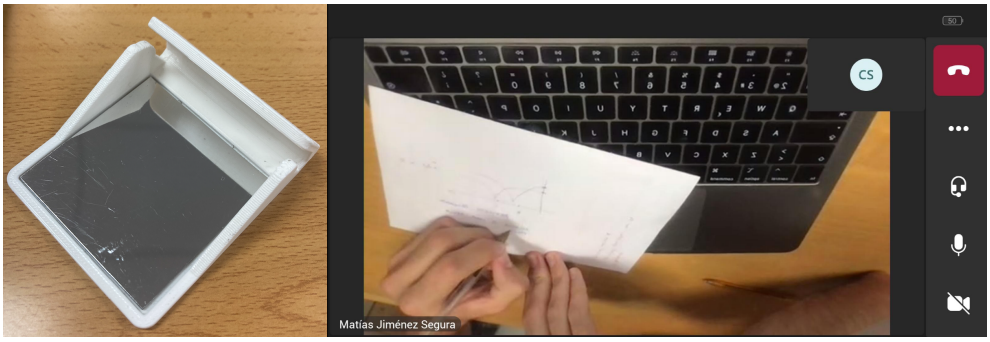


Figure 1.2: Figure which shows the mirror device to install onto the laptop webcam and for it to focus on the keyboard and it being used in a videocall, notably during the development of this project.

1.2.4. Union of Hand Tracking and Mirror device

When using both hand tracking and the mirror device, we obtained the result that can be seen at Figure 1.3. This was the first step towards our objective.

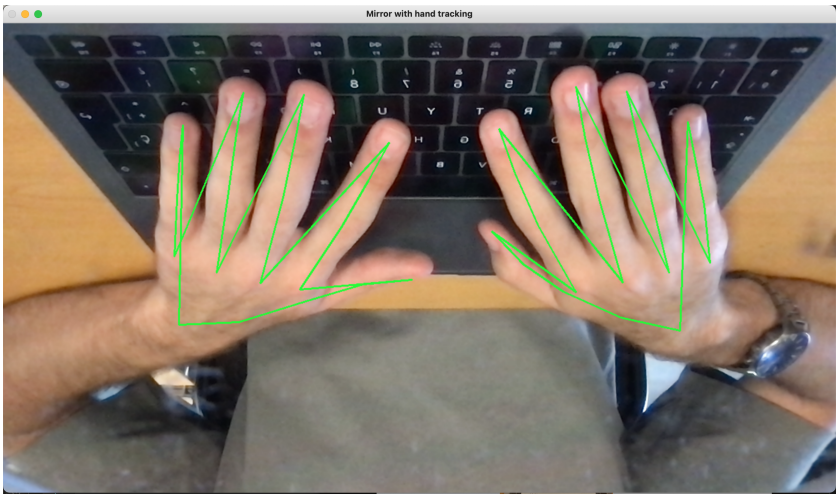


Figure 1.3: Figure which shows the proposed combination of hand tracking with the 3D-printed camera adapter, to achive hand tracking over the keyboard through laptop webcam with the mirror device.

[3] See 3D-design at: `https://www.thingiverse.com/thing:4588215`

[4] See 3D-design at: `https://www.thingiverse.com/thing:4592588`

2

System Structure, Tests and Design Decisions

2.1. Environment and Installation

The system has been developed in python for its multiplatform support in a variety of OS as MacOS, Linux or Windows. As in any modern Python development, the first step is to prepare an environment satisfying all the requirements [1] for the application. For this implementation, the required version of python for the different packages is 3.7. To start with, the environment must be created with the following commands:

```
$ python3.7 -m venv my_env
$ source my_env/bin/activate
```

The most important requirements for this project were the following packages:

- **numpy** [2]**:** provides support for multidimensional vector and matrix operations.
- **opencv** [3]**:** artificial vision library. It allows to draw lines, points... on photos or videos. The installation of the library and FFmpeg (necessary for mediapipe) for Ubuntu is done by executing:

```
$ sudo apt-get install -y \
    libopencv-core-dev \
    libopencv-highgui-dev \
    libopencv-calib3d-dev \
    libopencv-features2d-dev \
    libopencv-imgproc-dev \
    libopencv-video-dev
$ sudo apt-get install -y libopencv-contrib-dev
```

- **mediapipe** [4]**:** set of libraries and tools to apply artificial intelligence and automatic learning to python projects. Specifically, the system will use hand tracking. To install mediapipe is necessary to previously install opencv and Bazelisk [5]. The installation of mediapipe, once the mentioned libraries are installed, is made by following the steps defined in the MediaPipe manual [12]:

[1] https://github.com/GNB-UAM/HandTrackingForTypingOptimization/blob/main/requirements.txt
[2] https://numpy.org
[3] https://opencv.org
[4] https://chuoling.github.io/mediapipe/
[5] https://bazel.build/install/bazelisk?hl=es-419

1.– Clone repository.

```
$ git clone --depth 1 https://github.com/google/mediapipe.git
```

2.– Install the following dependencies.

```
$ sudo apt install python3-dev
$ sudo apt install python3-venv
$ sudo apt install -y protobuf-compiler
```

3.– In the repository folder (this step can take some time):

```
mediapipe$ pip3 install -r requirements.txt
mediapipe$ python3 setup.py install --link-opencv
```

Once this special packages are installed, the rest of the requirements can be executed by executing the following command in the project folder.

```
$ pip3 install -r requirements.txt
```

2.2. Structure of the system

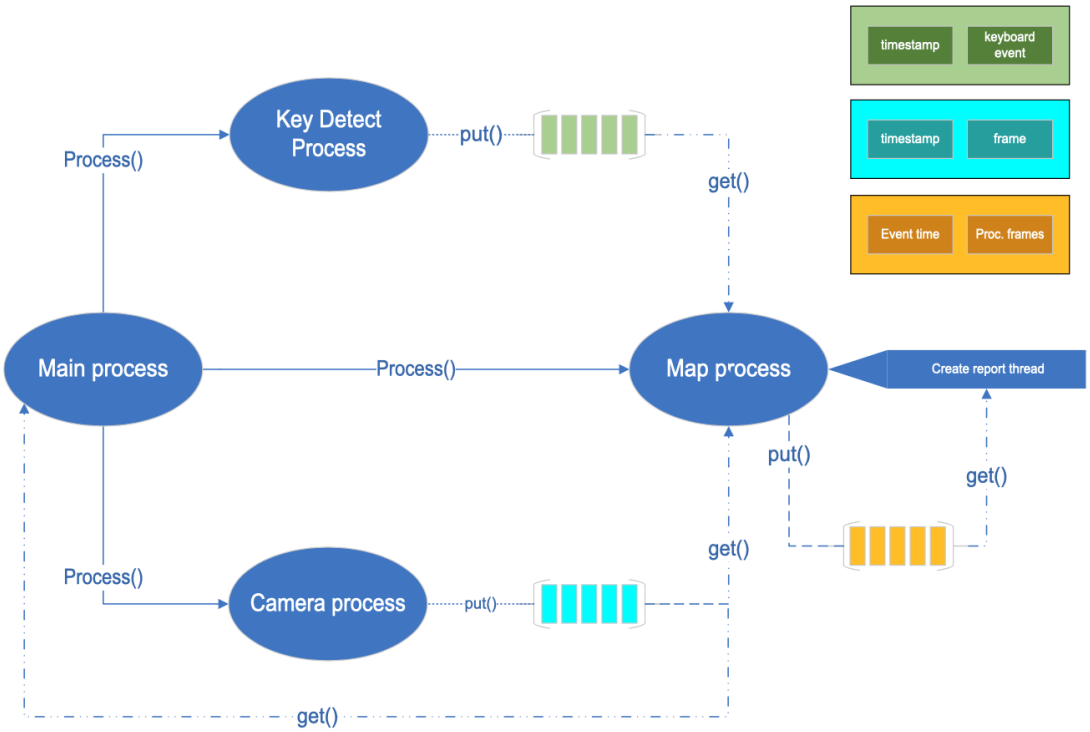


Figure 2.1: Design diagram of the system and communication between processes and threads.

The system, which is depicted in Figure 2.1, is divided in 4 processes:

- **Main or parent process:** creates and orchestrates the rest of processes.
- **Camera process:** sends tuples through the camera queue containing the frames and timestamps in real time. It uses the cv2 module for capturing video.
- **Key Detect process:** sends tuples through the key detect queue containing the key events detected and timestamps in real time. It is implemented with pynput keyboard listener. Section 1.2.2.
- **Map process:** is the process in charge of mapping the frames to the keyboard events and make hand tracking if necessary (for increased efficiency). It also contains a thread:
    - Create report thread: this thread analyzes the finished keyboard events and checks whether the keys have been pressed with the correct finger or not.

The communication between processes is made through FIFO queues with different message structures depending on each one's purpose. As these queues are part of the python multiprocessing package [6], they implement the necessary controls for managing concurrent access by multiple processes.

Queues are shared between all the processes, except for the camera queue, that is not shared with the key detect process, and vice versa, the key detect queue which is not shared with the camera process. Specifically, the queues created are the following:

- **Frame queue:** the content of this queue are tuples with the timestamp of the frame and the raw frame, this is, without having the hands detected on it. Apart from these messages, one different message is sent at the beginning of the process, that includes the resolution of the camera and a photo of the keyboard, so that the user can check the calibration of coordinates. The producer of this queue is exclusively the camera process, and messages are consumed by the Main (first message) and Map processes (timestamp and raw frames).
- **Key detection queue:** the messages of this queue follow a simple format, which, as the frame queue, contains tuples with timestamp and the keyboard event. The producer is the Key Detection process, and the consumers are the Main and Map process.
- **Finished events queue:** this queue is responsible to send keyboard events that have been processed with their respective associated frames to the report thread, so that it can check the taps and start creating the report at the same time that keyboard events are being associated with their frames.

It is also important to highlight that the arrival of SIGINT signals are always captured and managed for a controlled shutdown of the respective processes.

2.3. Classes

Two classes have been created for this system. The following subsections represent the `KeyboardEvent` and `KeyboardEventSet` respectively.

[6] `https://docs.python.org/3/library/multiprocessing.html`

2.3.1. KeyboardEvent

The KeyboardEvent (Code 2.1) class represent the keyboard event (timestamp and character), including the processed frames that must be associated to it.

Code 2.1: KeyboardEvent

```
class KeyboardEvent:
    def __init__(self, char: str, time: float):
        self.char = char
        self.time = time
        self.assoc_frames: List[Tuple[float, List]] = []
```

2.3.2. KeyboardEventSet

This class (Code 2.2) represents all the KeyboardEvents that are associable to a frame, as the frame currently being processed is within the range defined (for a keyboard event not to be discarded in a deque). The discard time is the temporal range multiplied by 1.5, so that the system is sure that no frames are analyzed -see if it has to be associated to an event or not- without taking into account possible events to which it must be associated. This class also implements a method to remove older events. As this deque is sorted by the arrival of the events at its left and the oldest ones at its right, to remove older events it is enough to look at the leftmost element until there is one that must not be discarded.

Code 2.2: KeyboardEventSet

```
class KeyboardEventSet:
    def __init__(self):
        self.events: deque[KeyboardEvent] = deque() # deque with older events at left

    def add_event(self, event: KeyboardEvent):
        self.events.append(event)

    def remove_events(self, timestamp: float) -> List[KeyboardEvent]:
        ret = []
        while self.events and not (timestamp -self.events[0].time <= config.TIME_RANGE_DISCARD):
            #pop from left
            ret.append(self.events.popleft())

        return ret
```

2.4. System Tests and Design decisions

The different parts of the structure are created as processes because the purpose of each one is different, and they do not share any content between them but the queues to communicate them. An exception is made for the report thread, as its goal is similar to its father's purpose, the mapping process.

Thus, it has been very important to make tests on each part individually to learn about its limitations, conditions and results separately, and then being able to make them work together.

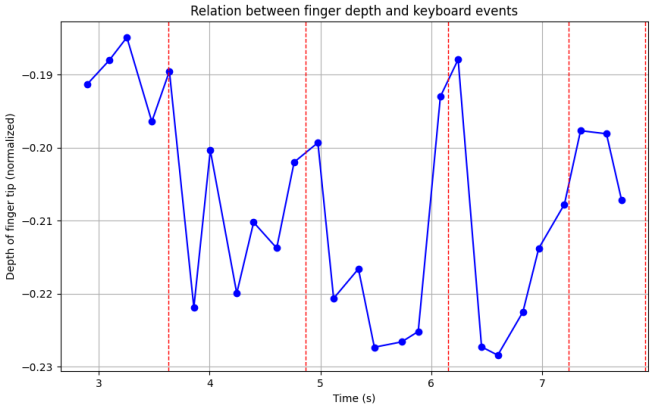
Hand detection

The camera process was going to be originally a hand detection module which processed all the frames, but this resulted in a bottleneck, as the camera could read 30 frames per second (fps) from the camera, but the MediaPipe library could only process 14fps in a regular computer. This did not give reliable result because the movement of the hands is much faster and was not be captured. To solve this, the hand tracking process turned into the camera process, and the frames are processed offline, and only if they have to be associated to a keyboard event. This lack of sufficient rate was discovered when trying to represent the depth coordinate of a finger when tapping a key. At first, when processing frames in real time, as the tap was too fast, the frames couldn't capture the moment in which the finger went down to the key, even when the movement was intentionally highly exaggerated (Figure 2.2(a)). When processing them offline, the increase in depth when pressing a key could be better appreciated (Figure 2.2(b)), as the system has almost double the information than before. Both experiments were done by raising up the finger in a non-natural way. Additionally, another test was conducted: making natural finger movements and clicking with the index finger some keys (Figure 2.2(c)). As we can see, the depth coordinate has a lot of noise and is not representative of the slight move of the finger tapping the key.

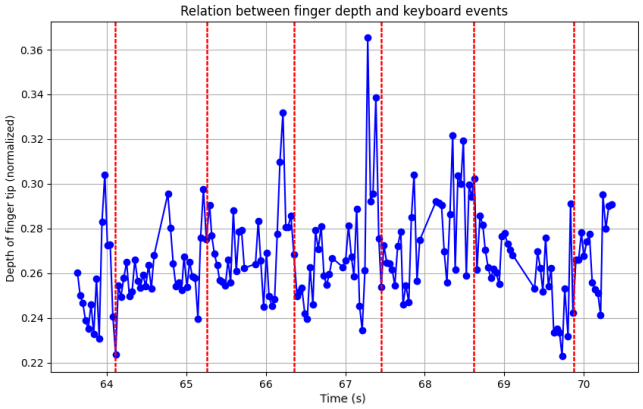
Mapping method

The original idea of the mapping process also changed, as it was to assign only the previous frames within a range of a keyboard event to it. This ran into some problems, as there is not a fixed delay, but a expected one between the keyboard event and the frame event timestamps. For this reason, frames corresponding to one later keyboard event could be assigned to a previous one. Let's see it at a graphical example (Figure 2.3).

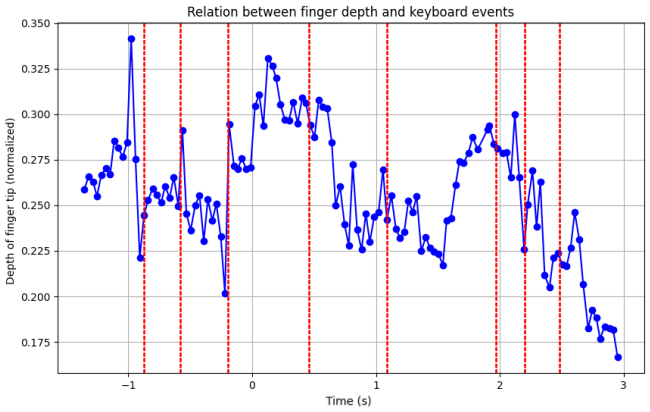
As it can be seen in this example, the system expected a delay of 200ms, this is that the difference between the timestamp marked in the frame and the one in the key tapping detection is 200 milliseconds, but the reality of the computer in which it is being executed is that this delay is 150ms. Therefore, if there are two events, one in the second 1.5 ('a') and another one in the 1.55 ('b'), the frame of the



(a) Default hand tracking implementation yielding a slow frame rate tested with large finger motions.



(b) Optimized hand tracking implementation tested with large finger motions.



(c) Optimized hand tracking implementation tested with natural finger motions.

Figure 2.2: Graph that represent finger depth over time together with keyboard events. Keyboard events are depicted with red vertical lines.

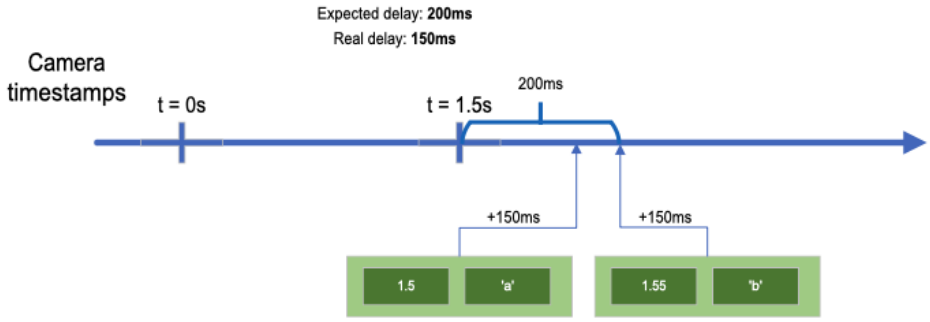


Figure 2.3: Representation of delay problem example. 'b' keyboard event frame is associated to 'a' keyboard event because of the difference between real and expected delay.

press of the 'b' key will be associated to the 'a' event (as there are no previous keyboard events, this frames will also include the 'a' tap) because the 'a' event expects to get all the previous available frames within a range until one with timestamp 1.7 (1.5 + expected delay). Therefore, as the frame of 'b' is exclusively associated to 'a', it will not be associated to its original event, 'b'.

Alternatively, a different mapping method has been developed. In this other method, the system associates frames to events if they are within a range before of after, allowing to add the same processed frame to different events. This way, it can be ensured that the keyboard event will have the frame corresponding to its press. This will also allow to study movements of the hands in higher detail, which is left as a future work.

Key coordinates detection

Although the idea was to manually calibrate the keyboard coordinates by tapping each key, a test to automatically detect the keys coordinates by edge detection was performed. The idea was simple: detecting key edges in the image and then separating each one of them and assigning the areas to keys. It was when trying this method with all types of keyboards that it became evident that it was not going to be possible, as the results were not reliable and didn't even come close to the expected ones (as it can be observed in Figure 2.4).

As this experiment resulted in outcomes that were not reliable due to variability in lighting conditions, another method had to be developed. What it was first thought was clicking on each key of the keyboard, but that would be very tedious and not user-friendly, so a simplification was developed: clicking just the extremes of each row and calculating the coordinates of all keys in between, since all the keys are the same size. For this, the left keys of the keyboard must be engaged with the index finger of user's right hand and the ones on the opposite part with the little finger of the same hand. It is checked that the hand clicking the keys is the right one; if the algorithm does not find the right hand, it asks the user to repeat the press of that specific key. For optimal calibration results, it is asked to leave the finger as still as possible over the key both before and after the press. This is because, as mentioned earlier, there

Figure 2.4: Figure which shows the experiment to automatically detect the keys coordinates by making edge detection on six different keyboards.

is no way to make sure that the expected delay is perfectly consistent between different setups during calibrating.

2.5. Data storage structures

The data structures used for storage are dictionaries with different structures depending on their purpose. There are two big dictionaries: one for storing all the finished keyboard events with their associated hand coordinates, and another for storing the information about the report.

Keyboard Events Dictionary Structure

The keys for the dictionary are strings. Most of them are the characters of all the possible keyboard events, but additionally, there are two more keys: coords_keyboard and resolution (see Code 2.3).

Keys characters

The value of each character contains an array of tuples which represent the keyboard events of its key. To have this represented, each tuple has the timestamp registered for the key detection and an array with more tuples, in which each one has the timestamp of a frame and the hands detected in it (another array in which each elements are arrays containing the landmarks of each point of the hand).

'res'

It stores the resolution of the frames, as this is necessary because the landmarks are normalized between 0 and 1, whilst the keyboard coordinates are not, because they are calculated with divisions that would result in a loss of decimals, as the values are very small.

'coords_keyboard'

The value of this key is another dictionary in which the keys are the characters and the value is their x and y coordinates. Apart from this keys, there is one more which is the average distance between keys ('avg_dist'), so that it does not have to be calculated every time we need it. This last distance is used to calculate the area in which a finger is considered to have pressed a key according to its position.

Code 2.3: Keyboard Events Dictionary Structure

```
{
    Keys characters: [
        (timestamp_kd1, [
            (timestamp_frame1, frame1_processed),
            (timestamp_frame2, frame2_processed)...]
        ),
        (timestamp_kd2, [
            (timestamp_frame1, frame1_processed),
            (timestamp_frame2, frame2_processed)...]
        )...
    ],
    'coords_keyboard':{
        Keys characters: (x_coord, y_coord),
        'avg_dist': x_avg_dist
    },
    'res': (x_resolution, y_resolution)
}
```

Report Dictionary Structure

The keys for this dictionary are exclusively the characters of the keyboard. Each value contains a dictionary in which the keys are 'OK' and 'BAD' that contain the taps that were executed correctly and the ones that not respectively in an array (see Code 2.4). The structure followed for key 'OK' values are just the x and y coordinates of the finger represented in a tuple. It comes harder when we talk about the 'BAD' values, as they have to indicate more parameters. This tuples have four values: Coordinates of the finger that should have pressed the key, name of the hand, index of the tip and coordinates of the finger that pushed the key. Any of this four values can be unknown, in which case the value introduced will be -1 in case of numbers and an empty string in case of the name of the hand.

Therefore, let's see in precision which cases can be found inside the 'BAD' and their meaning, as they might not be trivial.

- **Correct finger not found:** ((-1, -1), *, *, *). The meaning of the first tuple (coordinates of correct finger) being -1 for both x and y is that the correct hand couldn't be found in the frame.
- **Finger tip not detected within the key's area:** ((*, *), '', -1, -1). This would mean that there was not finger tip found inside the range defined as accepted for the key. This could be because the frame
- **Correct finger detected and finger that tapped the key:** everything has a valid value.

Code 2.4: Report Dictionary Structure

```
{
    Keys charaters: {
        OK:
        [
            coord_OK_tap1,
            coord_OK_tap2,
            coord_OK_tap3...
        ],
        BAD:
        [
            (correct_tip_coords_tap1, hand_name_tap1, finger_tap1_index, finger_tap1_coords),
            (correct_tip_coords_tap2, hand_name_tap2, finger_tap2_index, finger_tap2_coords)...
        ]
    }
}
```

2.6. Algorithms

2.6.1. Camera algorithm

The camera algorithm (Algorithm 2.1) is an infinite loop that sends the frames read from the webcam if the stop event is not set. This event is controlled by the Main process.

```
input: queue, stop_event
cap ← VideoCapture( 0 );
while True do
    if not stop_event is set then
        frame ← cap.read();
        timestamp ← time.time();
        queue.put( ( timestamp, frame) );
    end
end
```

Algorithm 2.1

2.6.2. Mapping algorithm

The mapping algorithm (Algorithm 2.2) is an infinite loop which reads the frames from the queue until a wait time for one single frame is larger than 5 seconds (**Main loop**), which means that the camera process has finished. For each frame, it waits until any more associable keyboard events can enter the queue (**Loop 1**). If a frame is processed before a keyboard event to which it must be associated is read from the queue, it won't be in the event keyboard set and, therefore, the frame will not be mapped to that event. This is done by making a loop until the difference between the actual time of the system and the frame timestamp is bigger than the association time range times 1.5. Once it has exited Loop 1, it removes the old enough events from the set and sends them to the report thread(**Loop 2**). After, it checks if the frame must be associated to any keyboard event (**Loop 3**) and if it does, the hands of the frame are detected and added to the list of associated frames of the corresponding Events(**Loop 4**). When all the frames have been processed, the keyboard events must be removed from the set with their current information, so the argument sent to the remove events function is a float with infinite value, so that all the events are considered old enough. All this removed events are sent to the report thread through the event queue.

```
1  event_set ← new KeyboardEventSet();
   ; /* Main Loop                                                    */
2  while queue get wait <5s do
3  |  frame_time, frame ← camera_queue.get();
   |  ; /* Loop 1                                                    */
4  |  while time.time() - frame_time < TIME_RANGE_DISCARD do
5  |  |  event_time, event ← kd_queue.get_no_wait();
6  |  |  if event then
7  |  |  |  event_set.add_event(new KeyboardEvent(event_time, event));
8  |  |  end
9  |  end
10 |  removed_events ← event_set.remove_old_events(frame_time);
   |  ; /* Loop 2                                                    */
11 |  for event in removed_events do
12 |  |  event_queue.put(event)
13 |  end
14 |  associable ← False;
   |  ; /* Loop 3                                                    */
15 |  for event in event_set.events do
16 |  |  if abs(event.time- frame_time) < TIME_RANGE then
17 |  |  |  associable ← True;
18 |  |  |  break;
19 |  |  end
20 |  end
21 |  if associable is True then
22 |  |  detection ← detect_hands(frame);
   |  |  ; /* Loop 4                                                 */
23 |  |  for event in event_set.events do
24 |  |  |  if abs(event.time- frame_time) < TIME_RANGE then
25 |  |  |  |  event.assoc_frames.add_event(detection);
26 |  |  |  end
27 |  |  end
28 |  end
29 end
30 removed_events ← event_set.remove_old_events(float('inf'));
31 for event in removed_events do
32 |  event_queue.put(event)
33 end
```

Algorithm 2.2

2.6.3. Report algorithm

The report algorithm (Algorithm 2.3) is an infinite loop (**Main loop**) which reads the events from the queue until a sentinel message (*None*) is sent from its father, the Map process. For each one of the iterations, the algorithm look for the specific frame with the expected delay defined in the system (**Loop 1**). Once it finds this frame, it will stop looking for other frames for this keyboard event (line 24). For that specific frame, the system will analyze the position of the hands and will check whether the right finger tip for that key are inside the accepted range for it (**Loop2**); if it is, it will add the coordinates of the finger to the dictionary in that specific letter, and in the 'OK' array (line 11). Otherwise, if the finger is not found inside the area accepted by the key, the algorithm will look for any finger tip which is inside the area and nearest to the specified coordinates (**Loop 3**). The position of the tip inside the area accepted by a key will probably mean that the key was tapped with that finger. After, a tuple of four elements will be added to that character 'BAD' array in the dictionary: coordinates of correct finger (1), name of the hand (2), index (3) and coordinates (4) of the finger that tapped the key. If any of this elements is not found, a -1 or a '' will be inserted in its place.

```
input: keyboard coordinates
// Main Loop
while not last event (None) do
    event ← event.get();
    ; /* Loop 1                                                    */
    for frame_time, frame in event.assoc_detected_frames do
        correct_finger ← False;
        if frame_time - event.time > EXPECTED_DELAY then
            correct_hand, optimal_finger ← info[ event.char ];
            coords_char ← coords_keyboard[ event.char ];
            ; /* Loop 2                                            */
            for hand in assoc_detected_frames do
                if correct_hand( hand ) is True then
                    if distance( optimal_finger_coords, coords_char ) is in range
                     then
                        report[ event.char ][OK].add_event( optimal_finger_coords
                         );
                        correct_finger ← True;
                    end
                end
            end
            if not correct_finger then
                nearest_dist ← infinite;
                ; /* Loop 3                                        */
                for hand in assoc_detected_frames do
                    for tip in ALL_TIPS do
                        if distance( tip, coords_char ) is in range then
                            nearest_dist ← distance( tip, coords_char );
                            nearest_tip ← tip;
                            hand_name_click ← hand_name( hand ) nearest_coords
                             ← tip
                        end
                    end
                end
                if found one then
                    report[ event.char ][BAD].add_event( optimal_finger_coords,
                     hand_name, nearest_tip, nearest_tip_coords );
                else
                    report[ event.char ][BAD].add_event( optimal_finger_coords, ",
                     -1, (-1, -1) );
                end
            end
            break
        end
    end
end
```

Algorithm 2.3

3 Results

3.1. Coordinates keyboard calibration experiment

This experiment shows the result of executing the calibration repeatedly (Figure 3.1): the circles represent the average coordinate of each key and the color means the finger the key has to be pressed with. Apart from this, each character is shown with the color assigned to its row exclusively for this experiment, so that it is easier to distinguish between correct and incorrect detections. As it can be seen, there are some keys that are not at their correct position, that's why it is important that the user is able to repeat the calibration as many times as he needs.

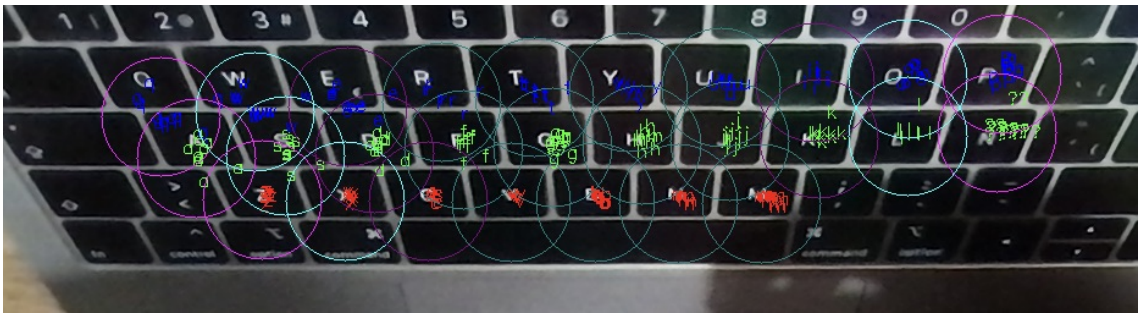


Figure 3.1: Figure which shows all the coordinates calculated in 11 repetitions (each letter) and the average coordinates (circles) for all the executions.

It is important to insist that this represents both good and bad outputs. Therefore, when calibrating, the user can obtain a proper representation of the keyboard coordinates or not, and have the possibility to repeat the calibration until he is satisfied with it.

3.2. Expected delay approximation experiment

The expected delay could be approximated by looking at the distance to a key over the associated frames to it and representing the offset equally for various keyboard events (Figure 3.2). To try this, the key 'a' was tapped with the index finger of the right hand. The finger was moved far away from the key

so that the distance could be appreciated better in two ways: fast and slow moves. As it can be seen, the fast moves generate a more pronounced 'V' in the graph, while the slow moves generate a more opened 'V', but both get the minimal distance to the key with a similar offset from the keyboard event. From this type of analysis, together with the difference between keyboard timestamps, the speed of each key typing could be calculated and be added to the personalized report.

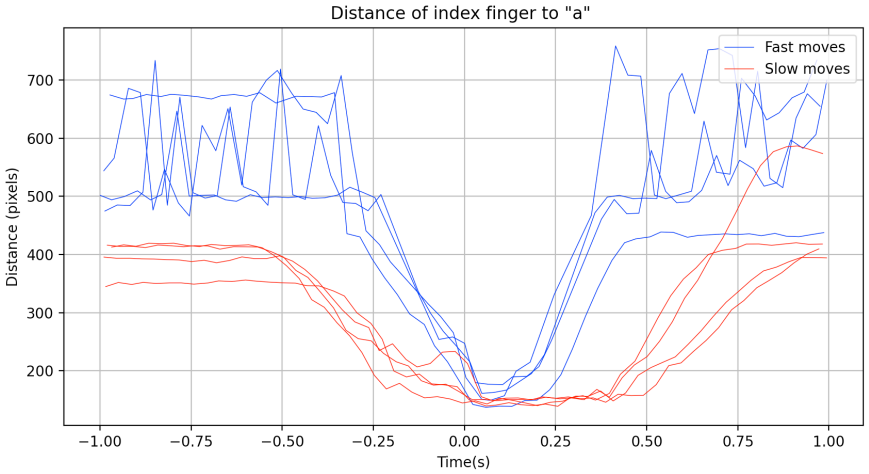


Figure 3.2: Distance of finger index to key 'a' over the time. The time is the offset between the keyboard event time and the frame timestamp, therefore, 0 is the moment when the event was captured.

3.3. Personal report generation

First, the keyboard coordinates detected with the calibration method are shown to the user (Figure 3.3). Once he approves them, the system starts registering the keyboard events and camera frames. It also starts the mapping process and a personal report thread is created. The information used in this section is obtained directly from some text written for this memory during the execution of the program.

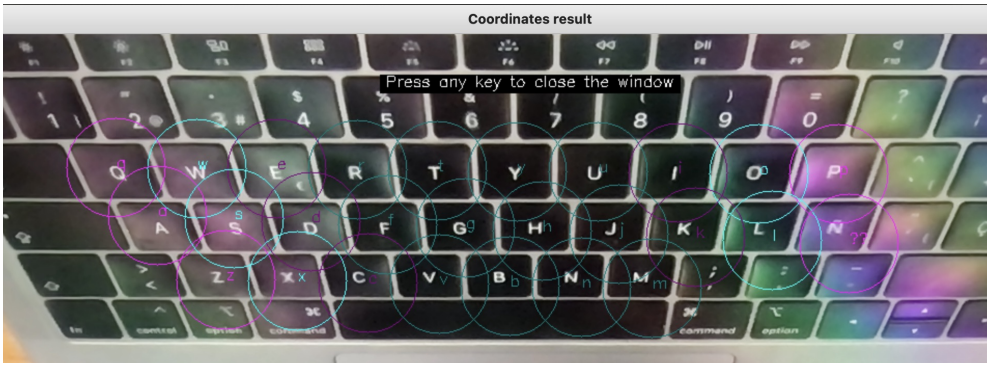


Figure 3.3: Cropped screenshot of what the system shows the user to set or not the calculated coordinates as valid.

The Figure 3.4 is a visual representation of the report generated by the type of some text in a real example. We can see the following types of indicators:

- **Green letters:** indicate that the key has been pressed correctly for that keyboard event.
- **Red letters:** indicate that the key has not been pressed correctly for that keyboard event.
- **'+':** represent the finger that might have tapped the key instead of the one that should have (legend of color represented in the bottom left part of the image).

Additionally, the system generates a text output (Figure 3.5)

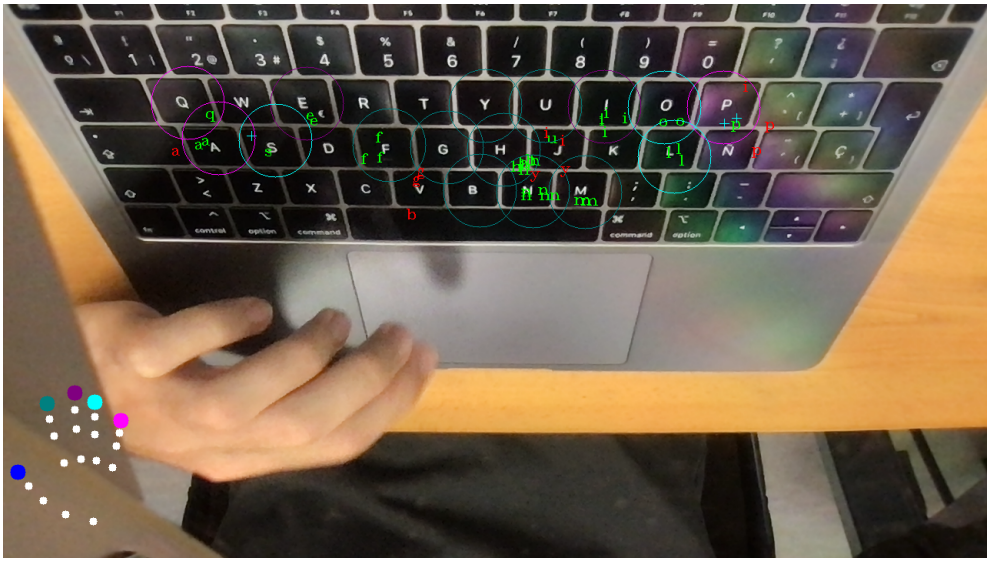


Figure 3.4: Visual representation of key presses during typing. This representation removes the keyboard events in which the system does not have information about the correct finger for that moment. Green letters represent correct presses, red means bad and '+' represent the finger that tapped the key when the optimal finger didn't.

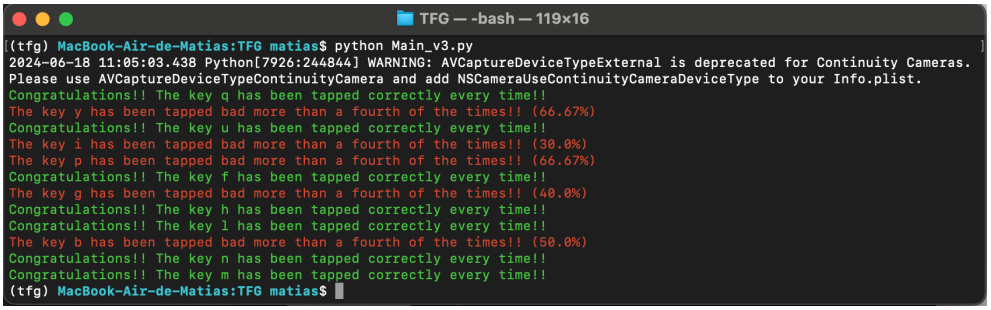


Figure 3.5: Text output generated by the system when writing the mentioned text.

From both reports (3.4, 3.5), it can be deducted, for example, that the user did not press correctly the letter 'p', as it was surely clicked incorrectly a 66 % of the times. It can be seen also that the user failed on this because he had to click it with the little finger (pink) and usually did it with the ring finger (light blue).

Additionally, the movement of the tip of the fingers during all the execution of the program can be

seen (Figure 3.6). To do this, an algorithm to sort the events and concatenate frames has been created (Code A.18). This algorithm sorts the keyboard events by their time arrival, as they are not sorted anymore (they are divided by characters) and concatenates the frames in order. It is possible that the system may lose some movements because if the user does not press a key in 2 times the time range to associate a frame to a keyboard event, the frame is discarded and, therefore, lost. It must also be taken into account that when the user takes a hand away from the camera range or the library does not detect it, there will be a discontinuity in the representation of the movements. As it can be seen, the most frequented areas are the ones near the natural positions of the hands, which are the keys 'a', 's', 'd', 'f', 'ñ', 'l', 'k', 'j'.

Figure 3.6: Movement of the finger tips (excluding thumbs) during a short session of typing.

Another figure that can be obtained from the stored data about the execution is the representation of the finger that should tap a key before this key is tapped (Figure 3.7). For this figure, 'n' and 'o' keys have been selected, but this images without zoom are generated individually for all the keys that have been pressed during the execution. The presence of some lines that do not end onto the key can be due to the finger with which the user tapped the key was not the correct one or because the frame corresponding to the tap is slightly later than the chosen one.

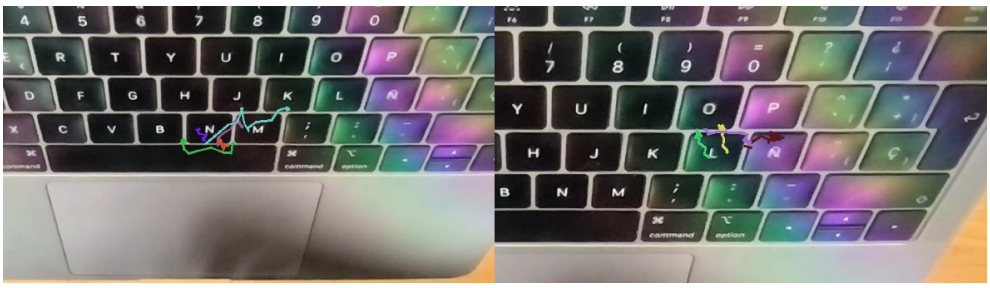


Figure 3.7: Previous movements of the finger supposed to tap 'n' (up) and 'o' (down).

4

Conclusions and Future Work

In conclusion, this system allows to map keyboard events and their associated hand movements in an optimal way so that typing patterns can be studied. This has been achieved by using a mirror device placed onto the laptop webcam and making hand detection on the frames captured by it. This system enables the user to get a personal report of its correct and incorrect keyboard taps, but it could also determine correct or incorrect finger movements when typing. This approach is not only applicable to improving typing techniques, as diagnoses of different psychomotor diseases [13] could be made from studying the hand movements.

Overall, we believe that the outcome has satisfactory achieved all the defined objectives. The movements representations are quite near to the real moves of the hand and it helps the user to learn good typing techniques. We believe that even though hand coordinates are sometimes lost because of the MediaPipe detections. This could be solved by implementing a more complex evaluation system. Not only the basis of this system would be really strong and useful for this implementation but also it would be relatively easy to add this extra functionality just by changing the report method. We decided to continue developing the project in this way to keep the original idea of making a system in which you do not need any more devices than your personal laptop and the mirror device so that it is easily accessible.

It is good to highlight that we achieved the reproduction of data that other systems obtain with more resources. For example, if we compare our results to the ones obtained at [1] we can see that we get similar content with much less resources. For example, at Figure 4.1 we can see the previous movements to letter 'o' in this project's system (down) and at Richardson's one (top). Our system is less precise that the other one such as, at the depth coordinate: we don't represent it at Figure 4.1 because it has a lot of noise, but they do represent it, as they need that coordinate to determine if a movement is a key tap, but we do have this knowledge, so the depth coordinate is not necessary.

This code has been openly shared at 'Grupo de Neurocomputación Biológica UAM's github [1] for the benefit of the science community.

[1] https://github.com/GNB-UAM/HandTrackingForTypingOptimization.git

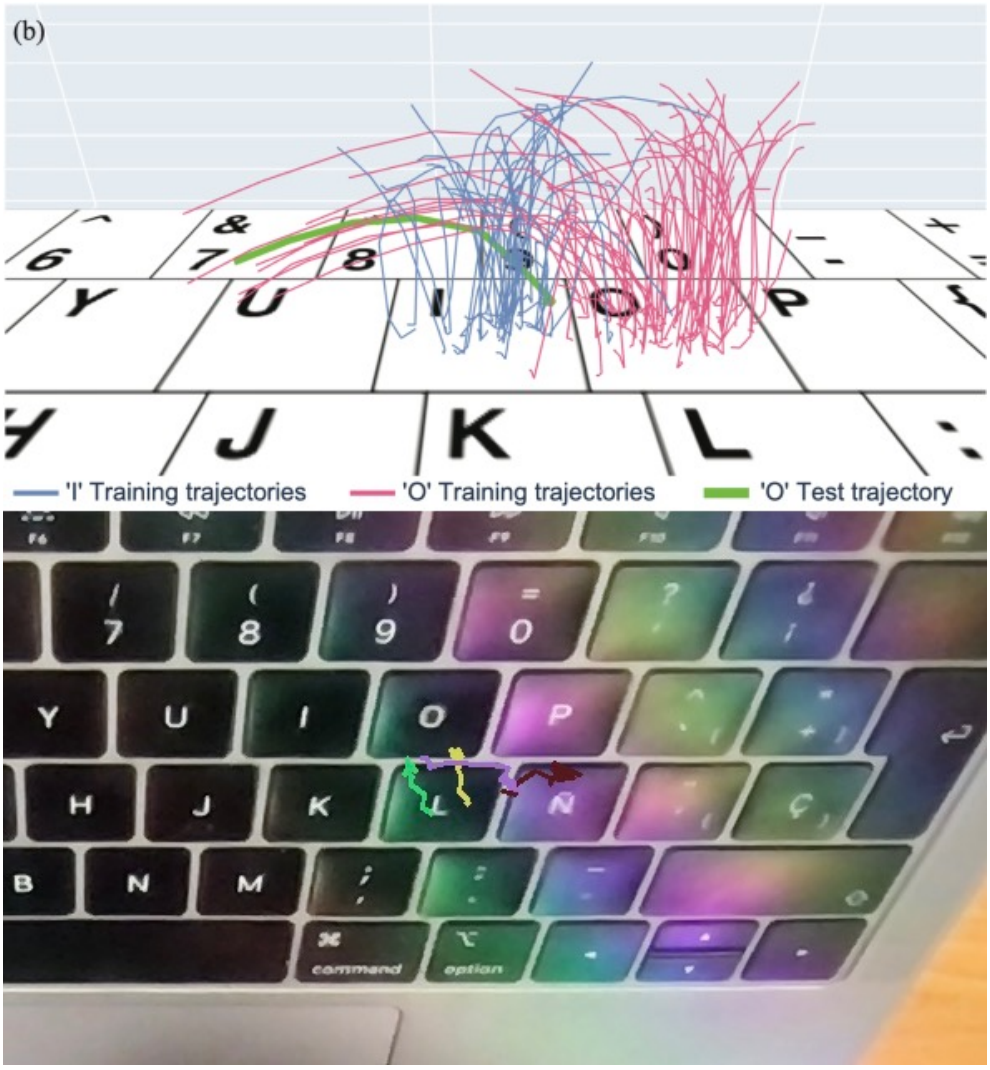


Figure 4.1: Comparison between previous movements representation of a tap to 'o' key in this project's system and the one developed at [1]

4.1. Objectives and Additional Implementations

4.1.1. Functional Requirements

- **The system must use MediaPipe hand tracking to allow the user get his hands coordinates:** Accomplished.
- **The system must be able to associate the processed frames to a keyboard event if it is within the range of time before the event:** Accomplished with Algorithm 2.2
- **The system must detect key presses even when the text is not being typed in the terminal:** Accomplished with 'pynput' library.
- **The user must do a completely manual calibration of the keyboard coordinates for each key:** Accomplished.
- **The system must show the area accepted for each key:** Accomplished (Figure 3.3).
- **The system must be able to create a report personalized for each execution showing:** Accomplished 3.3.
    - visual representations and accuracy percentages,
    - representation of presses of the different keys,
    - the most missed keystrokes.

4.1.2. Non-Functional Requirements

- **The system must liberate all resources correctly when it finishes:** Accomplished
- **The system must not crash if operates for long periods of time:** Accomplished.
- **The system must be compatible with macOS, GNU/Linux and Windows:** Accomplished with python.

4.1.3. Additional implementations

In addition the original objectives, the system improved or implemented the following ones:

- The user must do a completely manual calibration of the keyboard coordinates for each key $\rightarrow$ The user must do a calibration but clicking only the extreme keys of each row.
- The system must be able to associate the processed frames to a keyboard event if it is within the range of time before the event $\rightarrow$ The system can associate events previous or after to the keyboard event so that the hand movements can be better analyzed for any keyboard event.
- The system must allow the user to repeat the calibration until he approves it as valid.

4.2. Future work

We believe that this system is very innovative and can be a powerful tool if more features are implemented and improved. To start with, it would be crucial for deeper studies that a system could make predictions of the coordinates of the hands which are not detected. This could be calculated from the coordinates that the system do have, as making a better hand tracking library than the one

implemented by Google is pretty difficult. This prediction system could be easily implemented to this system since it would be necessary just to change the report method.

This project has been focused on the technical work but it would be interesting to develop a user-friendly interface which implements a more advanced human-computer interaction.

Last but not least, we believe a strategy to make sure that the system does not study a frame which is not the correct one (because of the problem with the expected and real delay between camera and key detection timestamps) should be developed. One option is to make an automatic approximation method as the one explained at 3.2 or to analyze the hand positions in more than one frame per keyboard event, analyzing its movement and, as done in [1], determine whether a movement is classified as a tap to a key or not. This last idea can be implemented on this system just by changing the report method, since all the finger trajectories are stored (see Figure 4.1).

Additionally, talking about functionality, a good idea would be to study the progress so that the user can not only see the report of one execution but also see how he has improved his typing technique since his first session. We consider it would also be beneficial to implement a method in which the text that the user writes is defined and check whether he writes it correctly or he makes mistakes, just as the current typing trainers do, but taking into account the hands position as well.

Bibliography

[1] M. Richardson, M. Durasoff, and R. Wang, "Decoding surface touch typing from hand-tracking," in *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology*, UIST '20, (New York, NY, USA), p. 686–696, Association for Computing Machinery, 2020.

[2] A. M. Feit, D. Weir, and A. Oulasvirta, "How we type: Movement strategies and performance in everyday typing," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, (New York, NY, USA), p. 4262–4273, Association for Computing Machinery, 2016.

[3] J. M. Rehg and T. Kanade, "Visual tracking of high dof articulated structures: An application to human hand tracking," in *Computer Vision — ECCV '94* (J.-O. Eklundh, ed.), (Berlin, Heidelberg), pp. 35–46, Springer Berlin Heidelberg, 1994.

[4] W. Westerman, *Hand tracking, finger identification, and chordic manipulation on a multi-touch surface*. Citeseer, 1999.

[5] SpectreX, "Brief history of hand tracking in virtual reality." *https://spectrexr.io/blog/news/brief-history-of-hand-tracking-in-virtual-reality*, 2022. Last Accessed at 20/05/2024.

[6] C. Lugaresi, J. Tang, H. Nash, C. McClanahan, E. Uboweja, M. Hays, F. Zhang, C. Chang, M. G. Yong, J. Lee, W. Chang, W. Hua, M. Georg, and M. Grundmann, "Mediapipe: A framework for building perception pipelines," *CoRR*, vol. abs/1906.08172, 2019.

[7] F. Zhang, V. Bazarevsky, A. Vakunov, A. Tkachenka, G. Sung, C. Chang, and M. Grundmann, "Mediapipe hands: On-device real-time hand tracking," *CoRR*, vol. abs/2006.10214, 2020.

[8] O. Talmage, "A tutorial on finger counting in real-time video in python with opencv and mediapipe." *https://medium.com/@oetalmage16/a-tutorial-on-finger-counting-in-real-time-video-in-python-with-opencv-and-mediapipe-114a988df46a*, 2023. Last Accessed at 05/04/2024.

[9] S. Han, B. Liu, R. Wang, Y. Ye, C. D. Twigg, and K. Kin, "Online optical marker-based hand tracking with deep labels," *ACM Trans. Graph.*, vol. 37, jul 2018.

[10] J. Fashimpaur, K. Kin, and M. Longest, "Pinchtype: Text entry for virtual and augmented reality using comfortable thumb to fingertip pinches," in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI EA '20, (New York, NY, USA), p. 1–7, Association for Computing Machinery, 2020.

[11] J. Hill and B. Sandalow, "Back-to-school hack shares students' handwritten work - and teacher response - in real time." *https://www.mccormick.northwestern.edu/news/articles/2020/08/back-to-school-hack-shares-students-handwritten-work-and-teacher-response-in-real-time.html*, 2020. Last Accessed at 10/04/2024.

[12] Google, "Mediapipe installation." *https://github.com/google-ai-edge/mediapipe/blob/master/docs/*, 2023. Last Accessed at 30/03/2024.

[13] L. Giancardo, A. Sánchez-Ferro, I. Butterworth, C. S. Mendoza, and J. M. Hooker, “Psychomotor impairment detection via finger interactions with a computer keyboard during natural typing,” *Scientific Reports*, vol. 5, Apr. 2015.

APPENDICES

A

Code

Code A.1: config.py: Part 1

```
from typing import Callable, Tuple, Dict, Any, List

'Time_range_to_assign_a_frame_to_a_keyboard_event'
TIME_RANGE = 0.5
TIME_RANGE_DISCARD = TIME_RANGE *1.5

EXPECTED_DELAY = 0.1

'Configuration_of_keyboard_by_rows'
ALL_KEYS_BY_ROWS = [
    ["q", "w", "e", "r", "t", "y", "u", "i", "o", "p"],
    ["a", "s", "d", "f", "g", "h", "j", "k", "l", "ñ"],
    ["z", "x", "c", "v", "b", "n", "m"]
]

'All_keys_of_keyboard_without_rows'
ALL_KEYS = [key for row in ALL_KEYS_BY_ROWS for key in row]

'Indexes_for_each_tip_in_the_coordinates_detected_by_MediaPipe'
THUMB_TIP = 4
INDEX_FINGER_TIP = 8
MIDDLE_FINGER_TIP = 12
RING_FINGER_TIP = 16
PINKY_TIP = 20
ALL_TIPS = [THUMB_TIP, INDEX_FINGER_TIP, MIDDLE_FINGER_TIP,
     RING_FINGER_TIP,PINKY_TIP]

'Dictionary_to_get_the_name_of_the_finger_from_the_index'
TIP_TO_FINGER: Dict[int, str] = {
    THUMB_TIP: 'thumb',
    INDEX_FINGER_TIP: 'index_finger',
    MIDDLE_FINGER_TIP: 'middle_finger',
    RING_FINGER_TIP: 'ring_finger',
    PINKY_TIP: 'pinky'
}

'Dictionary_to_represent_each_finger'
TIP_TO_COLOR: Dict[int, Tuple[int, int, int]] = {
    THUMB_TIP: (255, 0, 0),
    INDEX_FINGER_TIP: (128, 128, 0),
    MIDDLE_FINGER_TIP: (128, 0, 128),
    RING_FINGER_TIP: (255, 255, 0),
    PINKY_TIP: (255, 0, 255)
}

'Function_to_know_if_it_is_Left_hand'
def left(coords: List) -> bool:
    return coords[THUMB_TIP].x -coords[PINKY_TIP].x > 0
```

Code A.2: config.py: Part 2

```
'Function_to_know_if_it_is_Right_hand'
def right(coords: List) -> bool:
    return coords[THUMB_TIP].x -coords[PINKY_TIP].x < 0

'Function_to_get_name_of_the_hand'
def hand_name(coords: List) -> str:
    if coords[THUMB_TIP].x -coords[PINKY_TIP].x < 0:
        return 'right'
    else:
        return 'left'

'Dictionary_to_get_the_hand_function_and_the_index_of_the_finger_it_must_be_pressed_with_of_each_key'
TypeCallable = Callable[[List], bool]
OPTIMAL_CLICK: Dict[Any, Tuple[TypeCallable, int]] = {
    'q': (left, PINKY_TIP),
    'w': (left, RING_FINGER_TIP),
    'e': (left, MIDDLE_FINGER_TIP),
    'r': (left, INDEX_FINGER_TIP),
    't': (left, INDEX_FINGER_TIP),
    'y': (right, INDEX_FINGER_TIP),
    'u': (right, INDEX_FINGER_TIP),
    'i': (right, MIDDLE_FINGER_TIP),
    'o': (right, RING_FINGER_TIP),
    'p': (right, PINKY_TIP),
    'a': (left, PINKY_TIP),
    's': (left, RING_FINGER_TIP),
    'd': (left, MIDDLE_FINGER_TIP),
    'f': (left, INDEX_FINGER_TIP),
    'g': (left, INDEX_FINGER_TIP),
    'h': (right, INDEX_FINGER_TIP),
    'j': (right, INDEX_FINGER_TIP),
    'k': (right, MIDDLE_FINGER_TIP),
    'l': (right, RING_FINGER_TIP),
    'ñ': (right, PINKY_TIP),
    'z': (left, PINKY_TIP),
    'x': (left, RING_FINGER_TIP),
    'c': (left, MIDDLE_FINGER_TIP),
    'v': (left, INDEX_FINGER_TIP),
    'b': (left, INDEX_FINGER_TIP),
    'n': (right, INDEX_FINGER_TIP),
    'm': (right, INDEX_FINGER_TIP)
}

'Macro_to_write_green_text'
GREEN_INIT = "\033[32m"
GREEN_END = "\033[0m"

'Macro_to_write_red_text'
RED_INIT = "\033[31m"
RED_END = "\033[0m"
```

Code A.3: main.py: Part 1

```
1  from keyboard_sensor import launch_listener
2  from camera import record
3  from aux_module import get_keyboard_coordinates
4  from processing_loop import map
5  import config
6  from pynput import keyboard as kb
7  import multiprocessing
8  import cv2
9  import os
10 import signal
11 import time
12 from mediapipe import solutions
13 from mediapipe.framework.formats import landmark_pb2
14 import numpy as np
15 import json
16 import threading
17
18 def main(verbose):
19     multiprocessing.set_start_method('spawn')
20     camera_queue = multiprocessing.Queue(maxsize=-1)
21     kd_queue = multiprocessing.Queue(maxsize=-1)
22
23     map_process: multiprocessing.Process = None
24     cam_process: multiprocessing.Process = None
25     kd_process: multiprocessing.Process = None
26
27     def handler(signum, frame):
28         stop_camera.set()
29         while not camera_queue.empty():
30             _, _ = camera_queue.get()
31         if kd_process:
32             if kd_process.is_alive():
33                 kd_process.join()
34         if cam_process:
35             if cam_process.is_alive():
36                 cam_process.join()
37         if map_process:
38             if map_process.is_alive():
39                 map_process.join()
40         try:
41             exit(0)
42         except:
43             pass
44
45     signal.signal(signal.SIGINT, handler)
```

Code A.4: main.py: Part 2

```
47      #Create processes for camera and key detect
48      stop_camera = multiprocessing.Event()
49      cam_process = multiprocessing.Process(target=record, args=(camera_queue, stop_camera, ))
50      kd_process = multiprocessing.Process(target=launch_listener, args=(kd_queue, ))
51
52      # Start detection
53      cam_process.start()
54      kd_process.start()
55
56      stop_camera.set()
57
58      resolution, keyboard = camera_queue.get()
59      cv2.imwrite(f'output/keyboard.png', keyboard)
60
61      #Calibration loop
62      coords_keyboard = None
63      new_keyboard = False
64      while True:
65          if not new_keyboard:
66              try:
67                  with open("keyboard.json", 'r') as json_file:
68                      coords_keyboard = json.load(json_file)
69              except:
70                  new_keyboard = True
71          if new_keyboard or coords_keyboard is None:
72              stop_camera.clear()
73              coords_keyboard = get_keyboard_coordinates(camera_queue, kd_queue, resolution)
74              if coords_keyboard is None:
75                  # Wait for the processes to finish
76                  stop_camera.set()
77                  kd_process.join()
78                  os.kill(cam_process.pid, signal.SIGINT)
79                  while not camera_queue.empty():
80                      _, _ = camera_queue.get()
81                  cam_process.join()
82                  return 0
83
84              stop_camera.set()
85              with open("keyboard.json", 'w') as json_file:
86                  json.dump(coords_keyboard, json_file)
87
88          keyboard_copy = keyboard.copy()
89
90          for key in config.ALL_KEYS:
91              x_char, y_char = coords_keyboard[key]
92              _, tip = config.OPTIMAL_CLICK[key]
93              radius = int(coords_keyboard['avg_dist'] *2 / 3)
94              cv2.circle(keyboard_copy, (int(x_char), int(y_char)), radius = radius, color =
                    config.TIP_TO_COLOR[tip], thickness=1)
95              cv2.putText(keyboard_copy, key, (int(x_char), int(y_char)),
                        cv2.FONT_HERSHEY_SIMPLEX, 0.5, config.TIP_TO_COLOR[tip], 1)
```

Code A.5: main.py: Part 3

```
97              text_size, _ = cv2.getTextSize("Press any key to close the window",
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, 1)
98              init_text_x = int(resolution[0]/2 -text_size[0]/2)
99              init_text_y = 50
100             init_rect = (init_text_x-5, int(init_text_y-text_size[1]/2-5))
101             end_rect = (init_text_x+text_size[0]+5, int(init_text_y+text_size[1]/2))
102             color = (0, 0, 0)
103             keyboard_copy = cv2.rectangle(keyboard_copy, init_rect, end_rect, color, cv2.FILLED)
104             color = (255, 255, 255)
105             text = "Press any key to close the window"
106             cv2.putText(keyboard_copy, text, (init_text_x, init_text_y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 1)
107
108             cv2.imshow('Coordinates result',keyboard_copy)
109             cv2.waitKey(0)
110             cv2.destroyAllWindows()
111             cv2.waitKey(1)
112
113             # Discard keys to close the window or others
114             time.sleep(0.5)
115             while not kd_queue.empty():
116                 _, _ = kd_queue.get()
117
118             while True:
119                 print("Are the coordinates ok? y/n")
120                 _, key = kd_queue.get()
121                 if key == kb.KeyCode.from_char('y'):
122                     print("Yes")
123                     new_keyboard = False
124                     break
125                 elif key == kb.KeyCode.from_char('n'):
126                     print("NO")
127                     new_keyboard = True
128                     break
129             if not new_keyboard:
130                 break
131
132         stop_camera.clear()
133         # Start mapping process
134         map_process = multiprocessing.Process(target=map, args=(kd_queue, camera_queue,
                (int(resolution[0]), int(resolution[1])), coords_keyboard, verbose))
135         map_process.start()
136
137         # Wait for the processes to finish
138         kd_process.join()
139         os.kill(cam_process.pid, signal.SIGINT)
140         cam_process.join()
141         map_process.join()
142
143 if __name__ == "__main__":
144         #Uncomment to save videos associated to keyboard events
145         #main(verbose = True)
146         main(verbose = False)
```

Code A.6: camera.py

```
import cv2
import mediapipe as mp
import numpy as np
import time
import signal
import sys
from multiprocessing import Queue, Event
from pynput import keyboard as kb
import config
import json

def record(frame_queue: Queue, pause_event: Event):
    def handler(signum, frame):
        cap.release()
        cv2.destroyAllWindows()
        cv2.waitKey(1)
        sys.exit()

    signal.signal(signal.SIGINT, handler)

    cap = cv2.VideoCapture(0)
    n_frames = 0

    print("Taking photo of the keyboard...")
    time.sleep(2)
    ret, frame = cap.read()
    frame_queue.put(((cap.get(3), cap.get(4)), cv2.flip(frame, 1)))
    print("Done!")

    while True:
        if not pause_event.is_set():
            ret, frame = cap.read()
            n_frames += 1
            if not ret:
                break

            frame_queue.put((time.time(), cv2.flip(frame, 1)))
```

Code A.7: keyboard_sensor.py

```
from pynput import keyboard as kb
import time
from multiprocessing import Queue
import signal

def pulse(key, queue: Queue):
    timestamp = time.time()
    if hasattr(key, 'char') and key.char is not None:
        lower_char = key.char.lower()

        key = kb.KeyCode.from_char(lower_char)
    queue.put((timestamp, key))
    if key == kb.Key.esc:
        return False

def launch_listener(queue: Queue):
    def handler(signum, frame):
        #Sentinel message
        queue.put((None, None))
        queue.close()
        queue.join_thread()
        exit(0)

    signal.signal(signal.SIGINT, handler)
    listener = kb.Listener(lambda key: pulse(key, queue))

    listener.start()
    while listener.is_alive():
        pass
```

Code A.8: aux_module.py: Part 1

```python
from pynput import keyboard as kb
from typing import Dict, Tuple
import queue
import multiprocessing
import cv2
import os
import time
from config import *
from mediapipe import solutions
from mediapipe.framework.formats import landmark_pb2
from mediapipe.tasks.python import vision
import mediapipe as mp
import numpy as np

def get_keyboard_coordinates(camera_queue: multiprocessing.Queue, kd_queue: multiprocessing.Queue, resolution) -> Dict[str, Tuple[float, float]]:
    coords_keyboard: dict[str, tuple[float, float]] = dict()
    essential_keys = ['q', 'p', 'a', 'ñ', 'z', 'm']

    options = mp.tasks.vision.HandLandmarkerOptions(
        base_options = mp.tasks.BaseOptions(model_asset_path="hand_landmarker.task"), # path to model
        running_mode = mp.tasks.vision.RunningMode.IMAGE, # running on a live stream
        num_hands = 2, # track both hands
        min_hand_detection_confidence = 0.3, # lower than value to get predictions more often
        min_hand_presence_confidence = 0.3, # lower than value to get predictions more often
        min_tracking_confidence = 0.3, # lower than value to get predictions more often
    )
    # initialize landmarker
    detector = vision.HandLandmarker.create_from_options(options)

    while not kd_queue.empty():
        kd_queue.get()

    # go through essential keys in pairs
    for i in range(0, len(essential_keys), 2):
        first_key = essential_keys[i]
        second_key = essential_keys[i + 1]
        cont_loop1 = True
        while cont_loop1:
            print("Press '" + first_key.upper() +"' with the forefinger of your right hand.")
            while True:
                try:
                    # It has to keep emptying the queue
                    instant_frame, frame = camera_queue.get()
                    bool_right_hand = False
```

Code A.9: aux_module.py: Part 2

```
45                      try:
46                          instant_kd, key_pressed = kd_queue.get_nowait()
47                          if key_pressed == kb.KeyCode.from_char(first_key):
48                              if (instant_frame -instant_kd) < EXPECTED_DELAY:
49                                  while (instant_frame -instant_kd) < EXPECTED_DELAY:
50                                      instant_frame, frame = camera_queue.get()
51
52                              mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=frame)
53                              result = detector.detect(mp_image)
54                              for hand_coords in result.hand_landmarks:
55                                  # Right hand
56                                  bool_right_hand = right(hand_coords)
57                                  if bool_right_hand:
58                                      coords_keyboard[first_key] =
                                            hand_coords[INDEX_FINGER_TIP].x *resolution[0],
                                            hand_coords[INDEX_FINGER_TIP].y *resolution[1]
59                                      cont_loop1 = False
60                                      break
61                              if not bool_right_hand:
62                                  print("Couldn't make a detection of the right hand. Please try again")
63                              break
64                          elif key_pressed == kb.Key.esc:
65                              return None
66                          else:
67                              print(f"You didn't press the key {first_key}")
68                              break
69                      except queue.Empty:
70                          pass
71
72                  except queue.Empty:
73                      pass
74
75          cont_loop1 = True
76          while cont_loop1:
77              print("Press '" + second_key.upper() +"' with the forefinger of your right hand.")
78              while True:
79                  try:
80                      # It has to keep emptying the queue
81                      instant_frame, frame = camera_queue.get()
82                      try:
83                          bool_right_hand = False
84                          instant_kd, key_pressed = kd_queue.get_nowait()
85                          if key_pressed == kb.KeyCode.from_char(second_key):
86                              if (instant_frame -instant_kd) < EXPECTED_DELAY:
87                                  while (instant_frame -instant_kd) < EXPECTED_DELAY:
88                                      instant_frame, frame = camera_queue.get()
89
90                              mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=frame)
91                              result = detector.detect(mp_image)
```

Code A.10: aux_module.py: Part 3

```
                            for hand_coords in result.hand_landmarks:
                                # Right hand
                                bool_right_hand = right(hand_coords)
                                if bool_right_hand:
                                    coords_keyboard[second_key] = hand_coords[PINKY_TIP].x *
                                        resolution[0], hand_coords[PINKY_TIP].y *resolution[1]
                                    cont_loop1 = False
                                    break
                            if not bool_right_hand:
                                print("Couldn't make a detection of the right hand. Please try again")
                            break
                        elif key_pressed == kb.Key.esc:
                            return None
                        else:
                            print(f"You didn't press the key {second_key}")
                            break
                    except queue.Empty:
                        pass

                except queue.Empty:
                    pass

    avg_dist_x = 0
    for row in ALL_KEYS_BY_ROWS:
        if row[0] == 'q':
            interval_x = (coords_keyboard['q'][0] -coords_keyboard['p'][0])/(len(row)-1)
            interval_y = (coords_keyboard['q'][1] -coords_keyboard['p'][1])/(len(row)-1)
            avg_dist_x += interval_x
        elif row[0] == 'a':
            interval_x = (coords_keyboard['a'][0] -coords_keyboard['ñ'][0])/(len(row)-1)
            interval_y = (coords_keyboard['a'][1] -coords_keyboard['ñ'][1])/(len(row)-1)
            avg_dist_x += interval_x
        elif row[0] == 'z':
            interval_x = (coords_keyboard['z'][0] -coords_keyboard['m'][0])/(len(row)-1)
            interval_y = (coords_keyboard['z'][1] -coords_keyboard['m'][1])/(len(row)-1)
            avg_dist_x += interval_x
        for idx in range(len(row)):
            coords_keyboard[row[idx]] = (coords_keyboard[row[0]][0] -(interval_x *idx),
                coords_keyboard[row[0]][1] -(interval_y *idx))
    coords_keyboard['avg_dist'] = abs(avg_dist_x/3)
    return coords_keyboard
```

Code A.11: structs.py

```
1  from typing import List, Tuple
2  import config
3  from collections import deque
4  import time
5
6  class KeyboardEvent:
7      def __init__(self, char: str, time: float):
8          self.char = char
9          self.time = time
10         self.assoc_frames: List[Tuple[float, List]] = []
11
12         self.clear_frames = [] #only used with verbose
13
14     def __str__(self):
15         return f"Keyboard Event: {self.char} {self.time}"
16
17 class KeyboardEventSet:
18     def __init__(self):
19         self.events: deque[KeyboardEvent] = deque() # deque with older events at left
20
21     def add_event(self, event: KeyboardEvent):
22         self.events.append(event)
23
24     def remove_events(self, timestamp: float) -> List[KeyboardEvent]:
25         ret = []
26         while self.events and not (timestamp -self.events[0].time <= config.TIME_RANGE_DISCARD):
27             #pop from left
28             ret.append(self.events.popleft())
29
30         return ret
```

Code A.12: processing_loop.py: Part 1

```
from mediapipe.framework.formats import landmark_pb2
from structs import KeyboardEvent, KeyboardEventSet
from mediapipe.tasks.python import vision
from pynput import keyboard as kb
from typing import List, Tuple
from math import sqrt
from config import *
import multiprocessing
import queue
import threading
import cv2
import time
import datetime
import mediapipe as mp
import pickle
import json
import signal

report = {}
for key in ALL_KEYS:
    report[key] = {'OK': [], 'BAD': []}

def create_report(event_queue: multiprocessing.Queue, coords_keyboard: dict, resolution: Tuple[int, int]):
    while True:
        event = event_queue.get()
        if event is None:
            break
        for time_frame, processed_hands in event.assoc_frames:
            correct_finger = False
            dif = time_frame -event.time
            init = EXPECTED_DELAY -0.05
            end = EXPECTED_DELAY + 0.05
            if dif > init and end < 0.15:
                hand_function, correct_finger_idx = OPTIMAL_CLICK[event.char]
                x_char, y_char = coords_keyboard[event.char]
                avg_dist = coords_keyboard['avg_dist']
                x_finger = -1
                y_finger = -1
```

Code A.13: processing_loop.py: Part 2

```
41                  for hand in processed_hands:
42                      correct_hand_bool = hand_function(hand)
43                      if correct_hand_bool:
44                          x_finger = hand[correct_finger_idx].x *resolution[0]
45                          y_finger = hand[correct_finger_idx].y *resolution[1]
46                          if sqrt((x_char -x_finger) **2 + (y_char -y_finger) **2) < avg_dist*2/3:
47                              report[event.char]['OK'].append((x_finger, y_finger))
48                              correct_finger = True
49
50                  if not correct_finger:
51                      nearest_dist = float('inf')
52                      nearest_tip = None
53                      hand_name_click = ''
54                      #look for finger that tapped the key
55                      for hand in processed_hands:
56                          hand_name_v = hand_name(hand)
57                          for finger_tip_idx in ALL_TIPS:
58                              x_tip = hand[finger_tip_idx].x *resolution[0]
59                              y_tip = hand[finger_tip_idx].y *resolution[1]
60                              dist = sqrt((x_char -x_tip) **2 + (y_char -y_tip) **2)
61                              if dist < avg_dist/2 and dist < nearest_dist:
62                                  nearest_dist = dist
63                                  nearest_tip = finger_tip_idx
64                                  hand_name_click = hand_name_v
65                                  x_near = x_tip
66                                  y_near = y_tip
67                      if nearest_dist < float('inf'): #if found one
68                          report[event.char]['BAD'].append(((x_finger, y_finger), hand_name_click,
                                nearest_tip, (x_near, y_near)))
69                      else:
70                          report[event.char]['BAD'].append(((x_finger, y_finger), '', -1, (-1, -1)))
71                  break
```

Code A.14: processing_loop.py: Part 3

```
74      for key in ALL_KEYS:
75          good = 0
76          bad = 0
77          not_sure = 0
78          if len(report[key]['OK'])>0 or len(report[key]['BAD'])>0:
79              for click in report[key]['OK']:
80                  good += 1
81              for click in report[key]['BAD']:
82                  finger_correct = click[0]
83                  if finger_correct[0] > 0:
84                      bad += 1
85                  else:
86                      not_sure += 1
87              total = good + bad + not_sure
88              percentages[key] = {'good': good/total, 'bad': bad/total, 'not_sure': not_sure/total}
89          else:
90              percentages[key] = {'good': -1, 'bad': -1, 'not_sure': -1}
91
92      for key in ALL_KEYS:
93          if percentages[key]['good'] > -1:
94              if percentages[key]['bad'] > 0.25:
95                  print(f"{RED_INIT}The key {key} has been tapped bad more than a fourth of the
                        times!! ({percentages[key]['bad']}){RED_END}")
96              elif percentages[key]['good'] == 1:
97                  print(f"{GREEN_INIT}Congratulations!! The key {key} has been tapped correctly
                        every time!!{GREEN_END}")
98
99      with open("output/report.json", 'w') as f:
100         json.dump(report, f)
```

Code A.15: processing_loop.py: Part 4

```
104     # Ignore SIGINT Signals. This process will end when it stop receiving signals
105     signal.signal(signal.SIGINT, signal.SIG_IGN)
106
107     dictionary = dict()
108     dictionary['res'] = resolution
109     dictionary['coords_keyboard'] = coords_keyboard
110
111     all_keys_code = []
112     for key in ALL_KEYS:
113         dictionary[key] = []
114         all_keys_code.append(kb.KeyCode.from_char(key))
115
116     event_set = KeyboardEventSet()
117
118     options = mp.tasks.vision.HandLandmarkerOptions(
119         base_options = mp.tasks.BaseOptions(model_asset_path="hand_landmarker.task"), # path to
                model
120         running_mode = mp.tasks.vision.RunningMode.IMAGE, # running on a live stream
121         num_hands = 2, # track both hands
122         min_hand_detection_confidence = 0.3, # lower than value to get predictions more often
123         min_hand_presence_confidence = 0.3, # lower than value to get predictions more often
124         min_tracking_confidence = 0.3, # lower than value to get predictions more often
125     )
126     # initialize landmarker
127     detector = vision.HandLandmarker.create_from_options(options)
128
129     if verbose:
130         coordinates = (50, 50)
131         font = cv2.FONT_HERSHEY_SIMPLEX
132         scale = 1
133         thickness = 2
134
135     event_queue = multiprocessing.Queue(maxsize=-1)
136     report_thread = None
137     report_thread = threading.Thread(target=create_report, args=(event_queue, coords_keyboard,
            resolution, ))
138     report_thread.start()
```

Code A.16: processing_loop.py: Part 5

```
141         try:
142             instant_frame, frame = frame_buffer.get(timeout=5)
143             while True:
144                 try:
145                     instant_kd, key = kd_queue.get_nowait()
146                     if key == kb.Key.esc:
147                         break
148                     if key in all_keys_code:
149                         event = KeyboardEvent(key.char, instant_kd)
150                         event_set.add_event(event)
151                 except queue.Empty:
152                     pass
153 
154                 # Condition to stop adding possible events that are interesting for this frame
155                 if time.time() -instant_frame > TIME_RANGE_DISCARD:
156                     break
157 
158             # Remove old enough events
159             removed_events = event_set.remove_events(instant_frame)
160             for event_removed in removed_events:
161                 event_queue.put((event_removed))
162                 dictionary[event_removed.char].append((event_removed.time,
                        event_removed.assoc_frames))
163                 if verbose: #write video to disk the clear video
164                     fourcc = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
165                     out = cv2.VideoWriter(f'videos/{event_removed.char}{event_removed.time}.avi',
                            fourcc, 27.0, frameSize=(int(resolution[0]), int(resolution[1])))
166                     for instant_frame_event_removed, frame_event_removed in
                            event_removed.clear_frames:
167                         dim_text, _ = cv2.getTextSize(f"{instant_frame_event_removed}", font, scale,
                                thickness)
168                         coords_rect = (coordinates[0], coordinates[1] -dim_text[1])
169                         frame_event_removed = cv2.rectangle(frame_event_removed, coords_rect,
                                (coordinates[0] + dim_text[0], coordinates[1]), (255, 255, 255), cv2.FILLED)
170                         frame_text = cv2.putText(frame_event_removed,
                                f"{instant_frame_event_removed}", (50, 50),
                                cv2.FONT_HERSHEY_SIMPLEX, 1, (255,0,0), 2)
171                         out.write(frame_text)
172                     out.release()
```

Code A.17: processing_loop.py: Part 6

```
175             # check if there is any associable in the range
176             valid_frame = False
177             for event in event_set.events:
178                 if abs(event.time -instant_frame) < TIME_RANGE:
179                     valid_frame = True
180                     break
181 
182             if valid_frame:
183                 mp_image = mp.Image(image_format=mp.ImageFormat.SRGB, data=frame)
184                 result = detector.detect(mp_image)
185                 for event in event_set.events:
186                     if abs(instant_frame -event.time) < TIME_RANGE:
187                         #inside range --> append processed frame to keyboard event
188                         event.assoc_frames.append((instant_frame, result.hand_landmarks))
189                         if verbose:
190                             event.clear_frames.append((instant_frame, frame))
191 
192         except queue.Empty:
193             break
194 
195     # Remove old enough events
196     removed_events = event_set.remove_events(float('inf'))
197     for event_removed in removed_events:
198         event_queue.put((event_removed))
199         dictionary[event_removed.char].append((event_removed.time, event_removed.assoc_frames))
200         if verbose:
201             fourcc = cv2.VideoWriter_fourcc('M', 'J', 'P', 'G')
202             out = cv2.VideoWriter(f'videos/{event_removed.char}{event_removed.time}.avi', fourcc, 27.0,
                      frameSize=(1080, 720))
203             for instant_frame, frame in event_removed.clear_frames:
204                 dim_text, _ = cv2.getTextSize(f"{instant_frame}", font, scale, thickness)
205                 coords_rect = (coordinates[0], coordinates[1] -dim_text[1])
206                 frame = cv2.rectangle(frame, coords_rect, (coordinates[0] + dim_text[0], coordinates[1]),
                          (255, 255, 255), cv2.FILLED)
207                 frame_text = cv2.putText(frame, f"{instant_frame}", (50, 50),
                          cv2.FONT_HERSHEY_SIMPLEX, 1, (255,0,0), 2)
208                 out.write(frame_text)
209             out.release()
210 
211     event_queue.put(None)
212 
213     report_thread.join()
214     with open(f"output/all_data.pickle", "wb") as file:
215         pickle.dump(dictionary, file)
```

Code A.18: hands_mov.py: Part 1

```
1  import cv2
2  import pickle
3  from config import *
4  import numpy as np
5
6  def hands_mov(dictionary):
7      keyboard = cv2.imread('output/keyboard.png')
8      sorted_frames = []
9      sorted_frames_time = []
10     last_frame_introduced = -1
11
12     # look for last event
13     last_event = float('-inf')
14     for char in ALL_KEYS:
15         key_events_char = dictionary[char]
16         for time_event, frames in key_events_char:
17             if time_event > last_event:
18                 last_event = time_event
19
20     prev_event_time = float('-inf')
21     while prev_event_time != last_event:
22         next_event_time = float('inf')
23         for char in ALL_KEYS:
24             key_events_char = dictionary[char]
25             for time_event, frames in key_events_char:
26                 if time_event < next_event_time and time_event > prev_event_time:
27                     next_event_time = time_event
28                     next_event_frames = frames
29
30         prev_event_time = next_event_time
31         for time_frame, frame in next_event_frames:
32             if time_frame > last_frame_introduced:
33                 sorted_frames.append(frame)
34                 sorted_frames_time.append(time_frame)
35                 last_frame_introduced = time_frame
36
37     movs_index = [[], []]
38     movs_middle = [[], []]
39     movs_ring = [[], []]
```

Code A.19: hands_mov.py: Part 2

```
41
42      for frame in sorted_frames:
43          for index, hand in enumerate(frame):
44              bool = right(hand)
45              if bool:
46                  index = 0
47                  if int(hand[PINKY_TIP].x*dictionary['res'][0]) < 600:
48                      continue
49              else:
50                  if int(hand[PINKY_TIP].x*dictionary['res'][0]) > 600:
51                      continue
52                  index = 1
53              movs_pinky[index].append((int(hand[PINKY_TIP].x*dictionary['res'][0]),
                    int(hand[PINKY_TIP].y*dictionary['res'][1])))
54              movs_ring[index].append((int(hand[RING_FINGER_TIP].x*dictionary['res'][0]),
                    int(hand[RING_FINGER_TIP].y*dictionary['res'][1])))
55              movs_middle[index].append((int(hand[MIDDLE_FINGER_TIP].x*dictionary['res'][0]),
                    int(hand[MIDDLE_FINGER_TIP].y*dictionary['res'][1])))
56              movs_index[index].append((int(hand[INDEX_FINGER_TIP].x*dictionary['res'][0]),
                    int(hand[INDEX_FINGER_TIP].y*dictionary['res'][1])))
57      keyboard_copy = keyboard.copy()
58      for index in range(0, 2):
59          print(index)
60          hand_contour = np.array(movs_pinky[index], dtype=np.int32)
61          cv2.polylines(keyboard_copy, [hand_contour], isClosed=False,
                color=TIP_TO_COLOR[PINKY_TIP], thickness=2)
62
63          hand_contour = np.array(movs_ring[index], dtype=np.int32)
64          cv2.polylines(keyboard_copy, [hand_contour], isClosed=False,
                color=TIP_TO_COLOR[RING_FINGER_TIP], thickness=2)
65
66          hand_contour = np.array(movs_middle[index], dtype=np.int32)
67          cv2.polylines(keyboard_copy, [hand_contour], isClosed=False,
                color=TIP_TO_COLOR[MIDDLE_FINGER_TIP], thickness=2)
68
69          hand_contour = np.array(movs_index[index], dtype=np.int32)
70          cv2.polylines(keyboard_copy, [hand_contour], isClosed=False,
                color=TIP_TO_COLOR[INDEX_FINGER_TIP], thickness=2)
71
72      cv2.imwrite(f'hand_movements.png', keyboard_copy)
73
74
75  if __name__ == '__main__':
76      with open('output/all_data.pickle', 'rb') as f:
77          dictionary = pickle.load(f)
78
79      hands_mov(dictionary=dictionary)
```


UÁM
Universidad Autónoma
de Madrid