Technical_Specification

Student 1: Shane Grouse
Student Number: 17502633

Student 2: Jack Liston
Student Number: 17497764

Supervisor: Dr. Thomas Ward
Completion Date: 06/03/20202

0. Table of contents

1. Introduction

1.1 Overview

Type touch tutor is like your traditional interactive typing game where you are given a prompt of words and you have to type those words as fast as you can with as little typos as you can. However, the twist is that we also monitor your fingers using OpenCV. Bt using Hough Transformations, we calculate the position of your fingers at the time of a keypress. Once this done we calculate which finger you have used to press the key with and then we check if the finger you have used is the correct finger according to the touch typing methodology.
It is a command-line interface game written in python that creates a game environment using the curses library. As previously mentioned we are using OpenCV to calculate the position of each finger at the moment a key is pressed. This is done by using hough transformations to look for the circle edge at the point of your finger.
There are many ways to increase your productivity, but increasing your capacity to input data into your computer, i.e typing, seems like the most rudimentary.

Touch typing is the most effective and efficient way of typing on a keyboard. Surprisingly, it is an uncommon skill that not many people have. There are also few to no opportunities to learn it. The majority of people rely on “hunt and peck” typing (typing by looking at the keyboard) rather than learning to touch type. This trend will likely continue as touch screen devices become more dominant, especially with younger generations. We hope that our project will be able to help teach touch typing to anyone and everyone, regardless of if you are a beginner or you in need of fixing your bad habits.

1.2 Glossary

Define and technical terms used in this document. Only include those with which the reader may not be familiar.

- OpenCV
    - OpenCV is a programming library used mainly image processing.
- Hough Transformation
    - Hough Transform is a technique that extracts geometric features from an image.
- Touch Typing
    - This is a method of typing in which your index fingers are anchored to the f and j keys respectively. You then align each of you remaining fingers to the keys that lead to the edge of the keyboard. Each finger is then assigned a number of keys that only it is allowed to hit.
      <img src="https://www.bucketlist127.com/uploads/images/958236e84f1432cfb33e59e32
- Words per minute (WPM)
    - Words per minute are the most common measure of somebodies typing speed.

You simply take the number of words written by the user and divide it by the time(in minutes) that you have allocated for them to type.

- Keystroke
    - This is the act of hitting a key on a keyboard to input the information you wish to pass to your device.
- Framerate
    - A video is comprised of a certain amount of static images that display per second. The amount of images shown in a given timeframe is referred to as the framerate of that video. Framerate is the most commonly measured per second and the most modern camera will record in the area of 60 frames per second, meaning that camera framerate would be 60 frames per second.
- OpenCV
    - OpenCV is a library of programming functions mainly aimed at real-time computer vision.
- Python
    - Python is an interpreted, high-level, general-purpose programming

2. System Architecture

2.1 Typing Environment

2.1.1 Curses

With the design philosophy of keeping the emphasis of the image processing and touch typing aspects of this application, we decided that we wanted to make a minimalistic and performant application that catered to super users who make use of the keyboard. With these goals in mind, a command-line game with keyboard-centric menu systems was the obvious choice. Unfortunately, a simple bash command-line app allows for little flexibility regarding user feedback like color coding and text formating.
With a clear idea of the problems that faced us, the curses library that comes standard with Python 3 was to be the backbone of our game environment. It allowed for easy text manipulation and formatting and its lack of abstraction allows for a much simpler implementation of many of our image processing features versus something like pygame.
Curses itself is a library that supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals. Originally written for BSD Unix it has since been replaced by ncurses which is an open-source project written in C which is still being updated to this day! (last updated in February 2020). The Python module is a fairly simple wrapper over the C functions provided by curses. As an environment, we have been very impressed with its performance and it has undoubtedly created the simplistic and responsive environment that we needed.

2.2 Image processing

2.2.1 OpenCV

Our idea involves taking images of a keyboard as a user is typing and later, be able to detect their fingers in those same images. To do this, we need to have a way to control an external camera that can take and save singular frames in real-time. We also need a finger detection algorithm to search these images for fingers and return information about their positions.

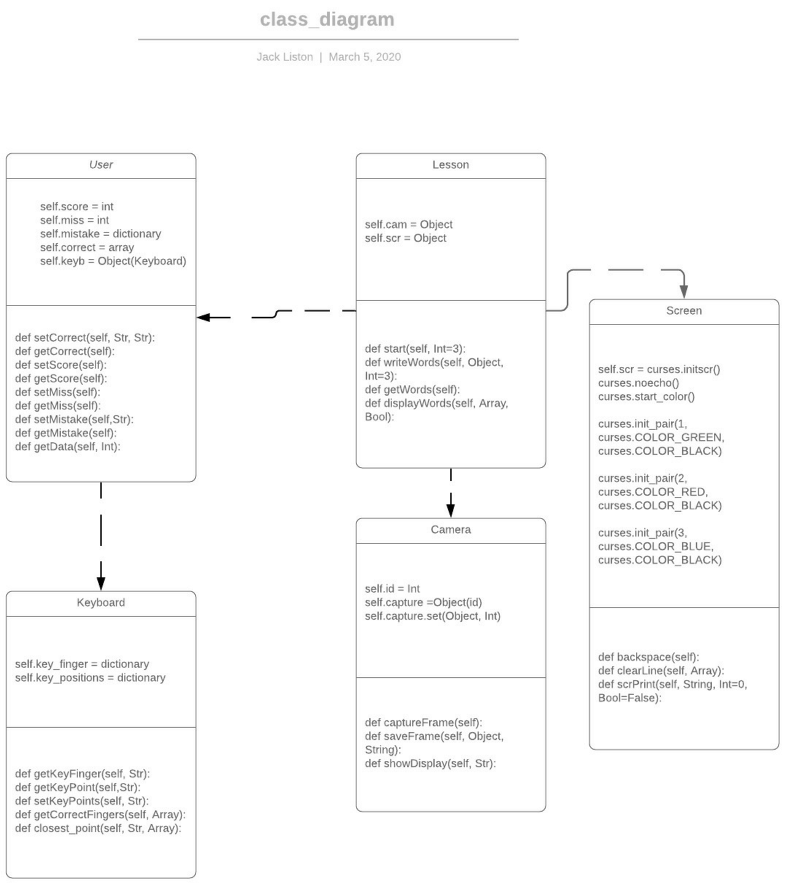
OpenCV is an open-source, computer vision library that provides support for real-time applications. It is written in C++ but has official interfaces and wrappings for many other languages, including Python, Java, MATLAB. For our project, the Python interface of OpenCV was a perfect fit. Firstly, connecting and controlling an external camera is very easy using the libraries *VideoCapture* class. It accepts the ID of the camera you would like to use and returns an object to represent the camera. This object allows you to take images and change the settings of the camera. Once you have an image, you can use it with OpenCVs wide array of functions, including saving and editing. OpenCV also had functions for Hough Transforms, which is used in our program to detect fingers

2.2.2 Hough Transforms

After taking the images, we need a way to identify fingers in them. For this, we used Hough Transforms. This is used in image processing to detect select features or shapes from an image. It begins by using Canny edge detection, which uses an algorithm and thresholds (used to somewhat define lines, higher thresholds result in fewer lines) to define the lines that are present in the image. Every pixel that is part of a line is white, while the rest of the pixels are black. This makes it a lot easier to scan the image and is a staple in computer vision. Next, the image is searched for the requested feature. Whatever feature is being searched for is represented as a mathematical equation. The edge detected image is then searched to see if any lines can correctly fit the features equation. If the line can fit the equation, an instance of the feature has been found. Hough Transforms are a built-in function in OpenCV and we used a transform for circles to detect fingers.

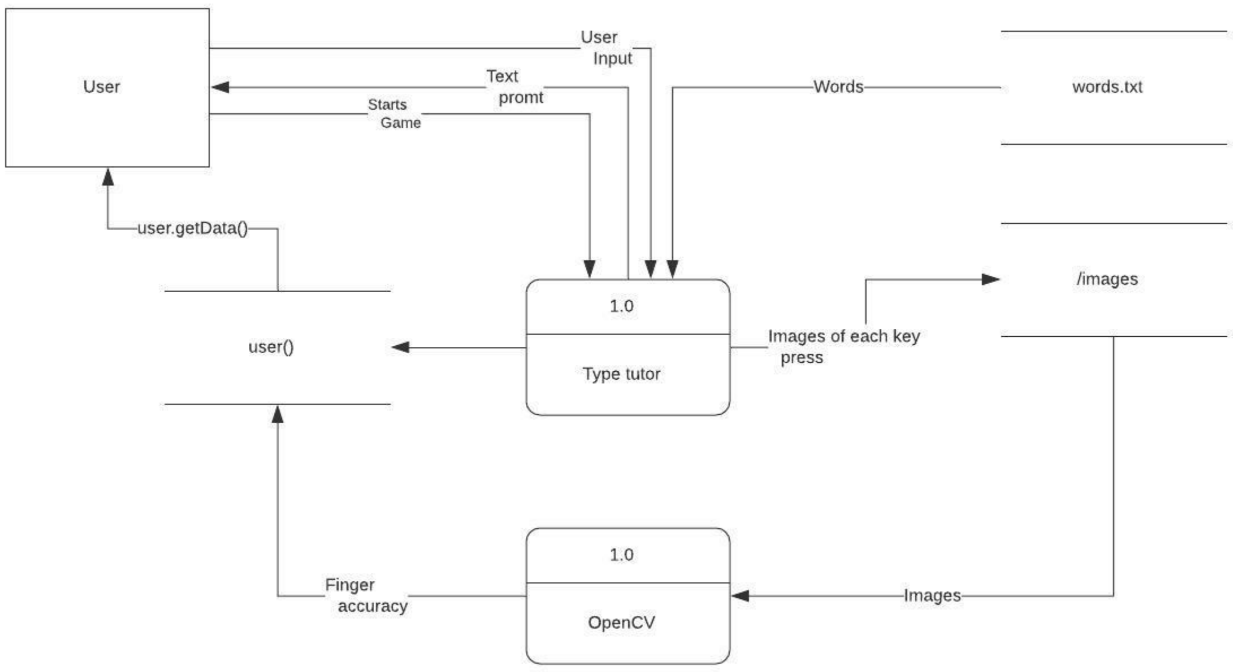
3. High-Level Design

3.1 Class Diagram


class_diagram
Jack Liston | March 5, 2020
User
self.score = int
self.miss = int
self.mistake = dictionary
self.correct = array
self.keyb = Object(Keyboard)
def setCorrect(self, Str, Str):
def getCorrect(self):
def setScore(self):
def getScore(self):
def setMiss(self):
def getMiss(self):
def setMistake(self,Str):
def getMistake(self):
def getData(self, Int):
Lesson
self.cam = Object
self.scr = Object
def start(self, Int=3):
def writeWords(self, Object, Int=3):
def getWords(self):
def displayWords(self, Array, Bool):
Screen
self.scr = curses.initscr()
curses.noecho()
curses.start_color()
curses.init_pair(1, curses.COLOR_GREEN, curses.COLOR_BLACK)
curses.init_pair(2, curses.COLOR_RED, curses.COLOR_BLACK)
curses.init_pair(3, curses.COLOR_BLUE, curses.COLOR_BLACK)
def backspace(self):
def clearLine(self, Array):
def scrPrint(self, String, Int=0, Bool=False):
Camera
self.id = Int
self.capture =Object(id)
self.capture.set(Object, Int)
def captureFrame(self):
def saveFrame(self, Object, String):
def showDisplay(self, Str):
Keyboard
self.key_finger = dictionary
self.key_positions = dictionary
def getKeyFinger(self, Str):
def getKeyPoint(self,Str):
def setKeyPoints(self, Str):
def getCorrectFingers(self, Array):
def closest_point(self, Str, Array):

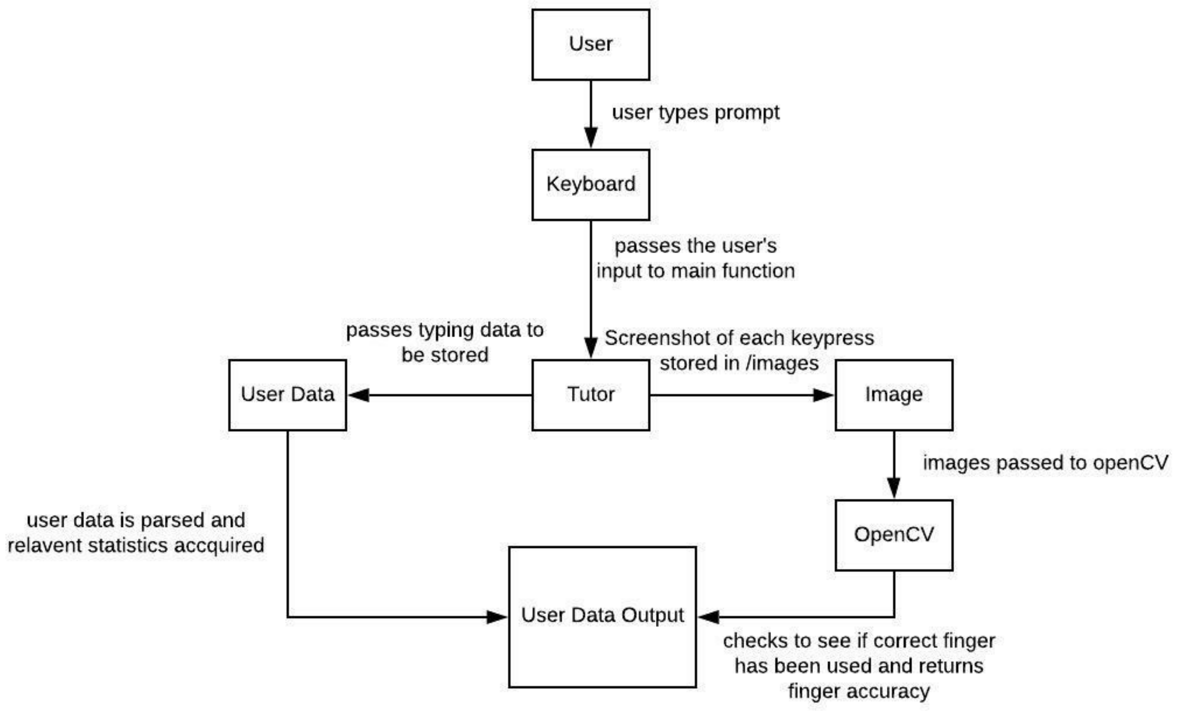
Above is a class diagram illustrating the links between each of our classes. It shows what is being initialized in each of the classes as well as the functions that the classes contain. It also describes the types of input that each function takes in. This really helped us to visualize the design structure of our application and to reduce the overlapping of functionality in our code. Having a class diagram also enabled us to prioritize the module nature of our design. As low latency is imperative for this application to be effective we had to optimize each class to the best of our ability and this modular design philosophy really contributed to that.

3.2 Data Flow Diagram



Similarly, the data flow diagram that is shown above illustrates our philosophy of a modular design while also illustrating our drive to keep the flow of data as simplistic as possible. We chose to leave the image processing to the end of the game as it would minimize the input delay while typing. While altering the parameters of our loose search, we ran into some delay issues but as we had left this until after the game it didn't affect the typing experience. If we were processing the images while the user was typing then that man introduces input lag which would desync the image capture and completely void the image processing.

3.3 Context Flow Diagram


User
user types prompt
Keyboard
passes the user's input to main function
passes typing data to be stored
Screenshot of each keypress stored in /images
User Data
Tutor
Image
images passed to openCV
OpenCV
user data is parsed and relavent statistics accquired
User Data Output
checks to see if correct finger has been used and returns finger accuracy

As you can seem from the above context diagram, our system architecture mainly flows to and from a main hub of functionality. We prioritized keeping both our planned architectural structure and our code structure as modular as possible. We decided to go for a structure that consists of the main script that uses five different objects.

3.4 Per class design analysis

3.4.1 Lesson

The lesson class is passed a camera and screen environment when it is first initialized. It is the class that manages the main game state. It is responsible for starting the game, reading in the target words, returning them as a prompt for the user to see and taking input from the user. It manages the environment by using the screen object that has been passed to it. Additionally, it uses the camera object that has been passed to it to take a screenshot each time the correct key is pressed. Once the key is pressed the image is saved and stored in /images to be assessed by the keyboard class and our hough functions. Finally, the lesson class passes user data such as correct and incorrect keystrokes to the user class. It calls the user functions user.setMiss and .setCorrect when a correct pr incorrect key is pressed Additionally it passes the keys that you have miss pressed to the user class to later be used in user.getData

3.4.2 Screen

The screen class is mainly responsible for the user interface. It makes use of the curses library that is native to python. It allows us to display words, highlight them in specified colors, create the typing game environment and format text in ways that a standard CLI app would not allow. Additionally, this class allows us to create our menus in the application as the getKey() function allows us to check for specified user input. When first initialized it creates the environment and the functions it contains are used to alter the text shown to the end-user.

3.4.3 User

This class is used to store, parsed and calculate the user's data that the typing tutor as gathered. When first initialized the user is assigned a score of 0 and a miss of 0. These correspond to how many correct keypresses and how many typos the user made while taking the test. The user is assigned a dictionary that is used to store any keys they have made a typo on as a key and the value is the number of times the user has missed that letter. They are given an array call correct which is made up of tuples. These tuples represent each correct keypress and they contain two strings. The first string is the key that was pressed and the second key is the name of the image that corresponds to the moment that key was pressed. This array is passed to our hough.identifyFingers() function which will return an array of the coordinates of each of the fingers in that each as tuples (x, y). The functions tied to this class are mainly for editing the parameters tied to the user. However, the user.getData() function is responsible for calculating the data that is displayed at the end of the game.

3.4.4 Keyboard

The keyboard class is responsible for storing all data related to the keyboard as well as preforming the geometric calculations related to the position of fingers relative to the keys. It contains a dictionary that has each key as a key and a tuple as a value. This tuple contains a string and another tuple. The String is referring to the correct finger to be used when pressing a key. The strong is a combination for the hand and the index of the finger. For example, your pinky finger on your left hand would be "l0". The second item in the value tuple is another tuple made of two integers. These refer to the x and y coordinates of the key on the image frame. They are used to calculate which finger is closest to that key at the point it was pressed. The functions contained in this class are the getters for the dictionary, a setter function which is used to print the alignment add onto the camera when you first start the application, and the final two functions the getCorrectFingers() and closest_point() functions. The getCorrectFinger() function takes in the user.correct array. It iterates through the tuples and passes the image to hough.identify fingers. It gets back an array of two arrays. Each sub-array contains four tuples. These represent the four fingers on each hand. Once it has the position of the eight fingers it will now pass the key that was pressed and the position of the four fingers on that had to closest_point(). Closest_point() simply checks which of the four fingers is closest to the position of the key at the point of the keypress and returns a bool on whether that was the correct finger or not. If the bool is true then getCorrectFingers() increments it's score. The score is then divided by the total correct answers to return a percentage of the correct finger presses.

3.4.5 Camera

When initialize you can either pass a specified video node or the class will use the default video node of 0. When using an external webcam with a laptop that has an internal webcam you will likely have multiple available video nodes. Your laptops operating system will dictate which node is set to default. If you are having issues with the wrong video input being displayed, manually input your desired video node when starting the program. e.g `python3 typeTutor.py 1`. Once the node id has been configured openCV's video capture function is initialized and a video buffer of one frame is applied. The class contains 5 functions. First is the captureFrame which returns a tuple with bool and numpy array. The bool refers to if the image was taken successfully and the numpy array is the data that makes up that frame. The saveFrame function takes that numpy array and writes it to an image. The showFrame is used to show an individual frame and the showDisplay shows a live video feed that is used to align the keyboard. The close function closes any frames of displays that are being output to the user.

3.5 Finger Identification

For identifying fingers within an image, we used OpenCVs in-built Hough circle function. This function performs a Hough Transform to search for circles within an image. Our first attempt was to try to recognize the user's fingernails. This was not possible as the image should be somewhat blurred before the search. This blur removed the nail's details and meant the nail could not be detected. Instead, the transform was able to detect the outline of the user's fingertips, achieving our goal.

After many attempted searches, it was obvious that one single search was not going to work (See Problems and Resolution). Instead, we are using two searches per image. The first is the *"confident"* search, which is designed to find guaranteed fingers. It has high blurring and edge detection to ignore keyboard and background lines. If not all the fingers have been detected, the second search is done. This is a *"loose"* search and has less blurring to capture more detail. It should identify all fingers but is expected to detect some false positives. A heuristic is then generated from the result of confidence. This heuristic is used to distinguish between fingers and false positives inside of loose.

4. Problems and Resolution

4.1 KeyBoard Mapping.

Issue: How to mapping to co-ordinates of each key on the keyboard
Solution: This was an issue that, along with the finger detection was going to be a priority to solve. Our initial concept behind how we would measure which finger pressed which key was fairly rudimentary, at the moment a key is pressed, which finger is closest to that key. Logically, the closest finger is the one that pressed the key but with that concept comes two tasks, gathering the position of the fingers and gathering the position of the keys. Thankfully the dimensions of the qwerty keyboard are standardized when looking at the alpha characters. Of course, there are qwerty layouts like ANSI and OSI which have alternatively formatted enter kets, shift key, etc, bt the alpha keys are always the same distance apart. So, we decided that when the user starts the application, if we provide them with an alignment tool to ensure their keyboard is aligned correctly, we can calculate the position of a key relative to the four corner keys, q, p, z, and m. We mapped the co-ordinates of the keys out in a dictionary and this has actually worked brilliantly for our needs. Unfortunately, alternative alpha layouts like Dvorak will not work with this software but this implementation solved the problem that we had excellently. Now, provided the user aligns the keyboard as per our alignment aid, grabbing the location of a specific key's co-ordinates is as simple as calling a value from a dictionary!

4.2 Finger Recognition

Issue: Difficulty identifying all fingers in a single Hough Transform
Solution: We attempted to find the best parameters for our Hough search that would find all fingers without any false positives. Different values of blurring and thresholds (for the Canny edge detection) were used, but one problem consisted. It was a battle between too much detail versus too little. Too few lines would result in not all the fingers being detected. Too many lines result in false positives. There was no middle ground either, as we would start to have false positives before detecting all of the fingers. The solution was to have two searches, one with very little detail and one with a moderate amount of detail. Finger detection is determined by a combination of the results from these two searches.

4.3 Typing Environment

Issue: Flask not being as efficient and as easy to implement as original research suggested.
Solution: Curses is a library that supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals. Originally written for BSD Unix it has since been replaced by ncurses which is an open-source project written in C which is still being updated to this day! (last updated in February 2020). The Python module is a fairly simple wrapper over the C functions provided by curses. As an environment, we have been very impressed with its performance and it has undoubtedly created the simplistic and responsive environment that we needed. After much research and deliberation, we have decided to switch our typing environment to the more simplistic and performant curses as it will both allow us to focus on the more exciting aspects of our project and provide us with an environment that meets all of the design philosophies that we wish to adhere to.

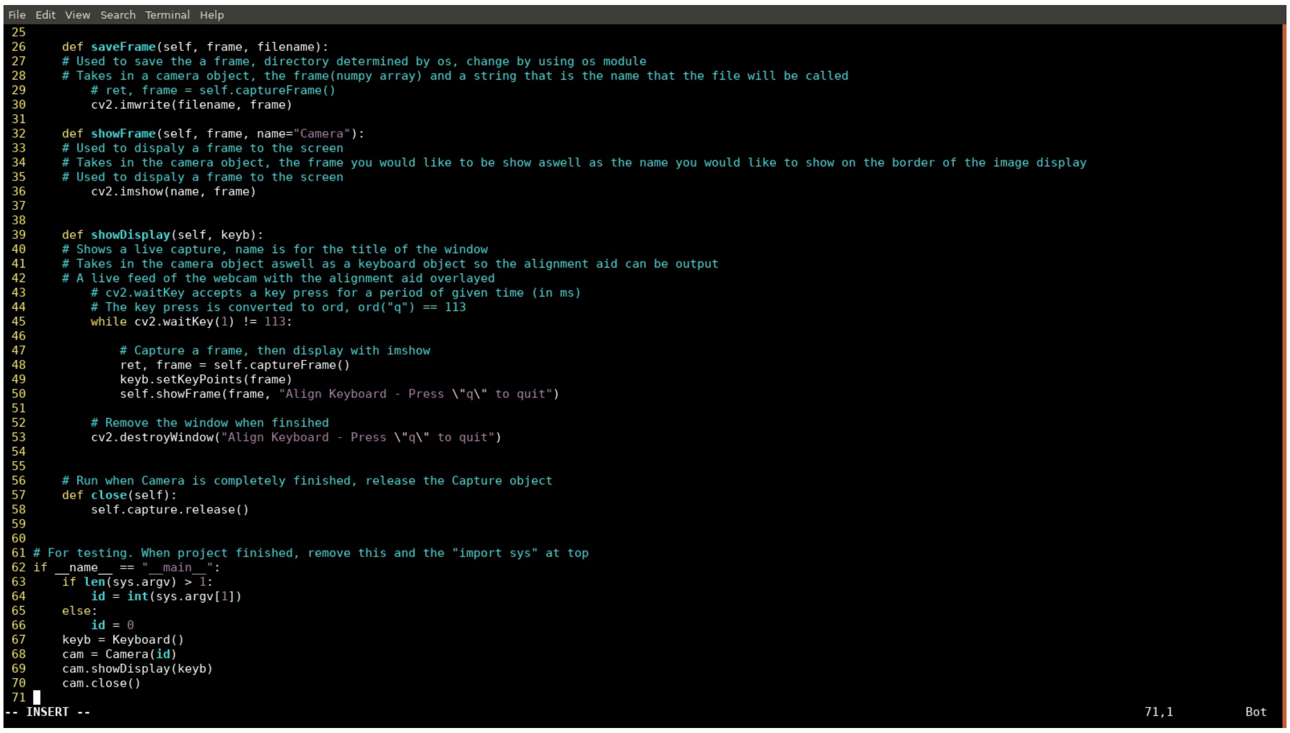
5. Installation Guide

To install and run typeTutor all you have to is clone our repository and install what is in our requirements.txt. You can do so by typing the following into your terminal.
`git clone https://gitlab.computing.dcu.ie/grouses3/2020-ca326-sgrouse-typetutor cd 2020-ca326-sgrouse-typetutor pip install -r requirements.txt cd code python3 typeTutor.py`

Testing

Functional Testing

Testing was a topic that we found difficult in a way as we have no true way to automate the main functionality. It creates a new prompt to be typed at each game and the positioning of the fingers is bound to change per game. With this in mind, we decided to make use of our modular design by doing the majority of our testing each class individually. Each class has a set of tests within the file that will be run if you simply run the file that contains that class.



```

    def saveFrame(self, frame, filename):
    # Used to save the a frame, directory determined by os, change by using os module
    # Takes in a camera object, the frame(numpy array) and a string that is the name that the file will be called
        # ret, frame = self.captureFrame()
        cv2.imwrite(filename, frame)

    def showFrame(self, frame, name="Camera"):
    # Used to dispaly a frame to the screen
    # Takes in the camera object, the frame you would like to be show aswell as the name you would like to show on the border of the image display
    # Used to dispaly a frame to the screen
        cv2.imshow(name, frame)


    def showDisplay(self, keyb):
    # Shows a live capture, name is for the title of the window
    # Takes in the camera object aswell as a keyboard object so the alignment aid can be output
    # A live feed of the webcam with the alignment aid overlayed
        # cv2.waitKey accepts a key press for a period of given time (in ms)
        # The key press is converted to ord, ord("q") == 113
        while cv2.waitKey(1) != 113:

            # Capture a frame, then display with imshow
            ret, frame = self.captureFrame()
            keyb.setKeyPoints(frame)
            self.showFrame(frame, "Align Keyboard - Press \"q\" to quit")

        # Remove the window when finsihed
        cv2.destroyWindow("Align Keyboard - Press \"q\" to quit")


    # Run when Camera is completely finished, release the Capture object
    def close(self):
        self.capture.release()


# For testing. When project finished, remove this and the "import sys" at top
if __name__ == "__main__":
    if len(sys.argv) > 1:
        id = int(sys.argv[1])
    else:
        id = 0
    keyb = Keyboard()
    cam = Camera(id)
    cam.showDisplay(keyb)
    cam.close()

-- INSERT --                                                        71,1          Bot
```

As you can see from the screenshot above, in each of our class files we included tests to ensure that they are correctly executing their functionality. In this case, if you run camera.py, a camera a keyboard is initialized and the keyboard is passed to the camera. Then align method is then shown and proven to be working.

Similarly, the hough.py file that stores our image processing functions can be run independently to ensure that it is working correctly. We have the script outputting the circles it finds on each file and they are even color-coded to show green, for confident and blue and red circles for the loose search, blue being the circles chosen to be the other fingers as they have been heuristically evaluated to be the best fit.