Paper Keyboard Typer

CS 6643 – Computer Vision

Ayushi Gupta Jonathan Yang Hemanth Grandhi

1. Introduction

In this project, we use computer vision methods to identify a printed keyboard and hand gestures to transform and recreate key clicks as a real keyboard. The fingers and palm will be recognized and segmented to find their world coordinates. If a finger taps on a specific key, the paper keyboard typer will be able to trace where the finger press in the world coordinates and the corresponding key would be registered. It takes multiple frames per second of the user's hand over the keyboard using the hand tracking method to identify the pressed key. We define the keys on the paper keyboard by assigning the coordinates of each key's area. This would require the application of thresholding, edge detection, and image transformation to generate an accurate coordinate system for the keyboard. We place the camera at a position where we can record the movement of the fingers and detect the key which the user presses. The keystroke can be registered by identifying the location of the finger and coordinate of the touch.

This report covers our approach and implementation of the paper keyboard typer. We also go over alternative methods that could have been used in our system. Then, we go over our results and strengths/weaknesses of the chosen approach. Lastly, the report covers how to improve our methodology in future implementations.

2. Approach

The high-level process of the keyboard typer first requires noise reduction and filtering on the current frame. Then to perform image transformation by determining the perspective transformation matrix to generate the top-down view of the paper keyboard. The matrix is then used to generate where each key coordinate is mapped in the original frame. Meanwhile the hand coordinates of the index finger is saved and monitored until a key click is registered. Once registered, the fingertip (x,y) coordinate is converted to the top down view coordinate and the script determines which key is clicked based on the distance equation.

2.1 Noise Reduction

Noise reduction and filtering is necessary for a more accurate gradient magnitude and to ensure no other edges or noise affects the final Canny Edge Detection output. For noise removal, we applied median blurring, bilateral filtering, and lastly gaussian blurring to the captured frame for keyboard layout recognition.

These implementations can be seen below where we used cv2 libraries to implement median blurring and bilateral filtering. A median blur was given a kernel size of 5 while the bilateral filter was given a kernel size of 5 and sigma color/space of 75. The gaussian blur was implemented manually through convolution and implementing a 5 by 5 gaussian kernel across the image with sigma equal to 1. This results in the following output after the noise reduction filtering:

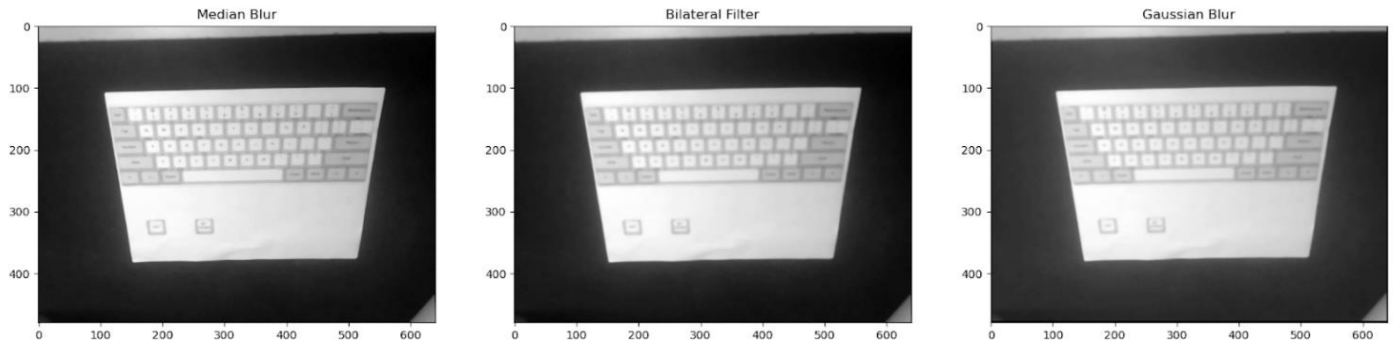


Figure 2.1: Median Blur, Bilateral Filter, and Gaussian Blur

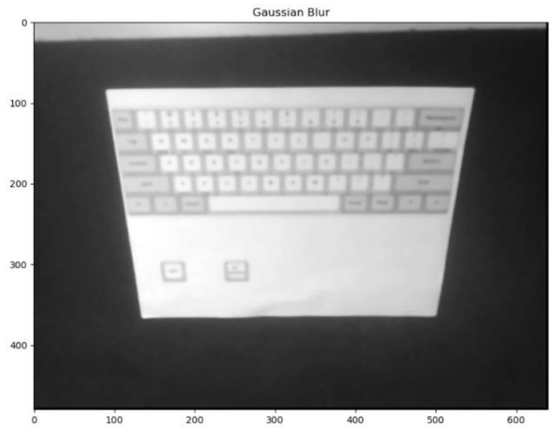
Through testing and applying different filters, this combination yielded the best final edge detection output. Since edge detection results are highly sensitive to image noise, median filter ensures all salt & pepper noise is removed while bilateral filter uses the space and intensity ranges to apply filtering without affecting the edges. Last Gaussian filter finalizes the noise and smooth the edges for a better magnitude gradient calculated next.

2.2 Magnitude Gradient

After filtering, magnitude gradient is calculated to detect the change in edge intensity and its corresponding direction. In our implementation, we used the below Sobel filters to generate the change of pixel intensities in the horizontal and vertical directions. Additionally, the slope of the gradient is also calculated using the below arctan equation:

$$G_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \qquad \theta = \tan^{-1}\left(\frac{\partial f}{\partial y} / \frac{\partial f}{\partial x}\right)$$

As a result, we are able to generate the following magnitude gradient below:



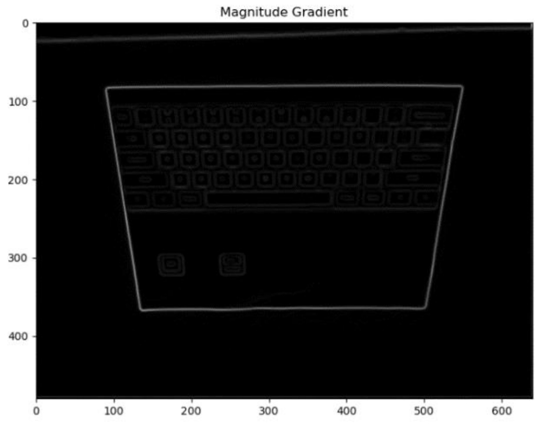


Figure 2.2: Gaussian Blur and Magnitude Gradient Image

2.3 Canny Edge Detection

After denoising the image and the gradient calculation steps, we now have the information needed to begin applying the Canny Edge Detection algorithm which can be split into 3 pieces: Non-Maximum Suppression, Double Thresholding, and Hysteresis.

2.3.1 Non-Maximum Suppression

Non-Maximum Suppression is focused on thinning the edges of the magnitude gradient so each line is only one pixel thick. This is done by looping through all the points of the magnitude gradient image and finds pixels of maximum value depending on the line's direction. This allows edge directions to be identified based on the angle value. By checking if the pixels in the same direction have higher intensities than the current pixels, we can determine the edge and determine the thin middle line.

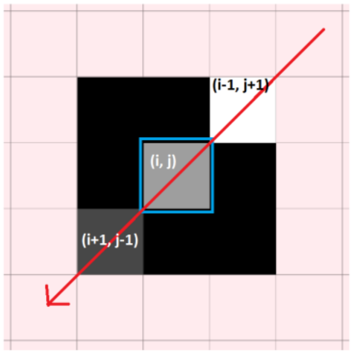


Figure 2.3: Non-Maximum Suppression Pixel Example

This check of intensities based on the edge direction can be seen above were based on the red directional line, the most intense pixel is (i-1, j+1) on the top right since it is white compared to the other black values. The current (i, j) pixel is set to 0 while if (i, j) was the most intense value in its neighborhood, it would keep its intensity value and be part of the one pixel width output line. The results of applying this Non-Maximum Suppression algorithm can be seen below compared to the Magnitude Gradient output:

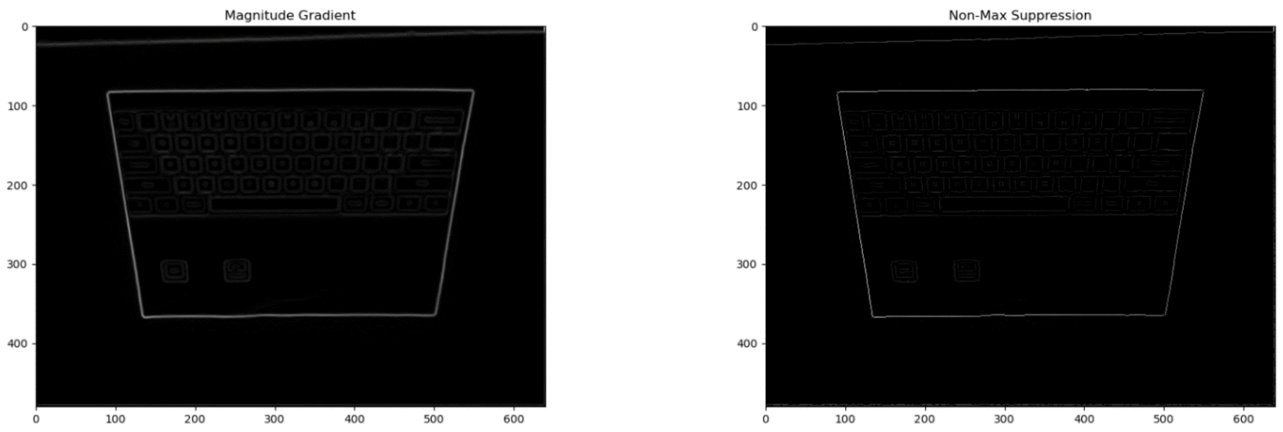


Figure 2.4: Magnitude Gradient and Non-Maximum Suppression Image

2.3.2 Double Thresholding

Thresholding is applied to identify strong and weak pixels. Based on strong and weak pixel intensity inputs, the paper keyboard typer displays the corresponding intensity values to be either of those ranges. So for pixels that have high intensities to contribute to the final edge, we will give them the strong intensity input value. For weak pixels the opposite is true so they are given low intensity values though not completely thrown away yet.

Next a high and low threshold value is given to identify strong and irrelevant pixels. If a pixel intensity is higher than the given high threshold value, it will be shown in the threshold output. Any pixel intensities below the low threshold are thrown away and pixels between the thresholds will be kept for the next step for a final check to see if that pixel is part of the edge.

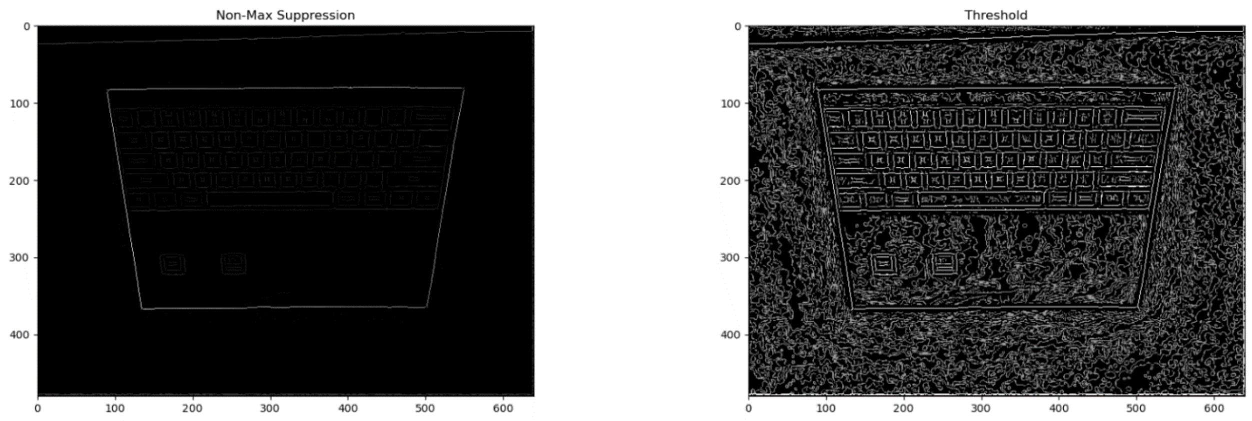


Figure 2.5: Non-Maximum Suppression Image and Double Threshold Image

2.3.3 Hysteresis

The last step for the final Canny edge detection is hysteresis which consists of checking each weak pixel and determining if it should be set as a strong pixel based on neighboring pixels. If there is a neighboring pixel around coordinate (i, j) as per the diagram in 2.3.1 that's strong, it will be rewritten with the strong pixel intensity. Otherwise, it will be thrown away.

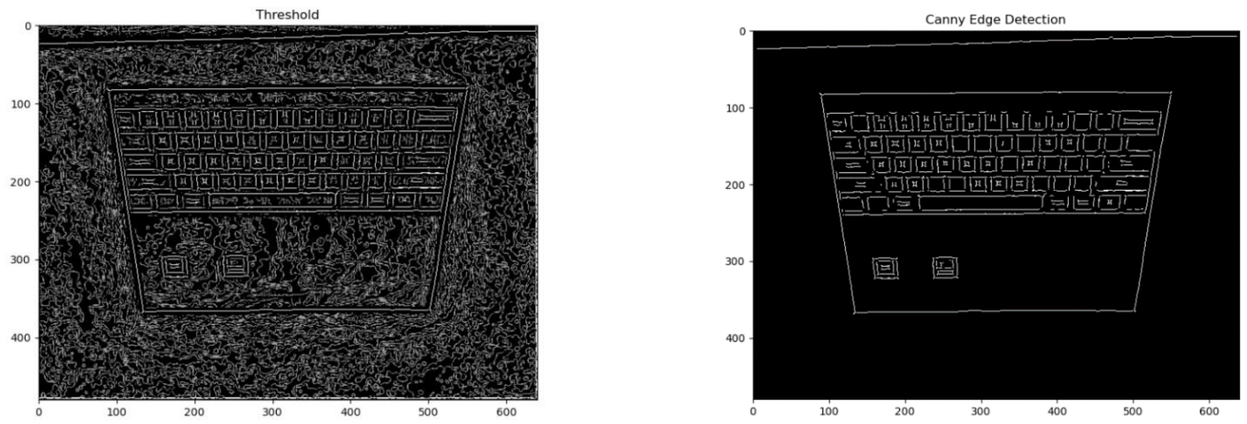


Figure 2.6: Double Threshold Image and Canny Edge Detection Image

2.4 Morphological Transformation

That last step to complete our edge detection is using the closing morphological transformation function. By taking the Canary Edge Detection output image and applying a dilation followed by erosion transformation, it will close any gaps on the object which is helpful for our final contour line. Since our Canary Edge detection is not always perfect due to noise or resolution issues, this morphological closing step is required to ensure the edges of our paper are closed so we can find the endpoints of the paper.

Erosion removes the boundaries of a foreground object by applying a convolution of a kernel full of ones. All pixels near edges/boundaries will be discarded based on the size of the kernel.

Dilation uses the opposite of erosion so if a pixel element in the kernel is one, the rest of the pixel elements in that kernel is also 1.

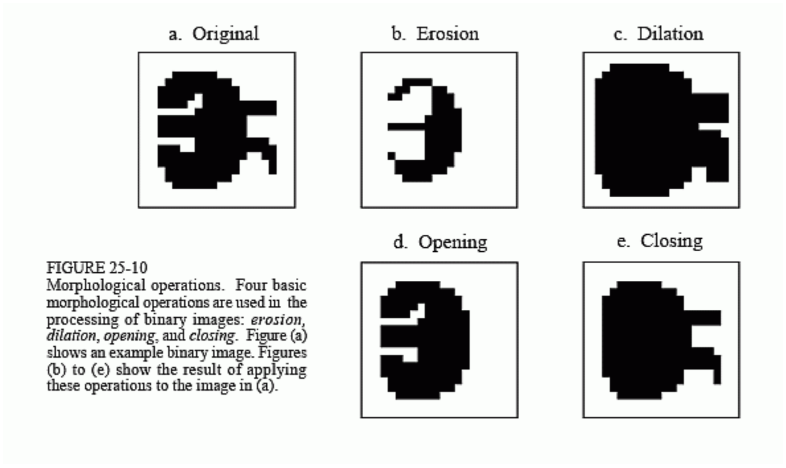

a. Original
b. Erosion
c. Dilation
d. Opening
e. Closing
FIGURE 25-10
Morphological operations. Four basic morphological operations are used in the processing of binary images: erosion, dilation, opening, and closing. Figure (a) shows an example binary image. Figures (b) to (e) show the result of applying these operations to the image in (a).

Figure 2.7: Erosion, Dilation, Opening, and Closing Example

By combining both methods by first applying dilation followed by erosion, we can close small holes without bulking up the edges and to give us the final connected Canary Edge output shown below.

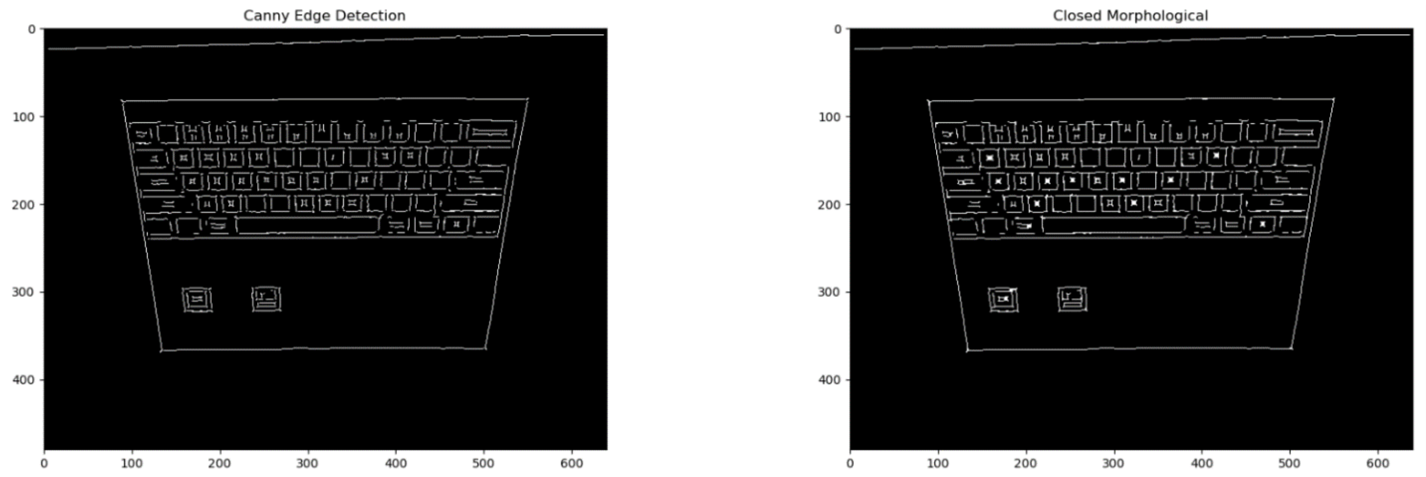


Figure 2.8: Canny Edge Detection and Closed Morphological Transformation

2.5 Finding Contours in Frame

With this final closed Canny Edge Detection output image, we can now find all contours in the frame. Contours are seen as a curve joining all continuous points with the same color or intensity, which in our case would be the white lines in the output frame. By finding all the contours in the image, we will grab all closed loops such as individual squares of keycaps. Our approach has us finding all the contours in the frame, with the biggest contour being the entire paper layout of the keyboard. Since each contour's perimeters can be calculated, we can approximate the largest contour as the entire paper keyboard layout so we find the largest perimeter for our result.

2.6 Finding Endpoints/Corners of the Paper

The end points of the image are calculated by going through all the set contours in the frame and determining which is the largest rectangular contour. By getting the maximum and minimum of each contour, we can find the endpoints of that contour and in a frame where the largest rectangle would be the paper itself, that would return the corners of the keyboard layout template.

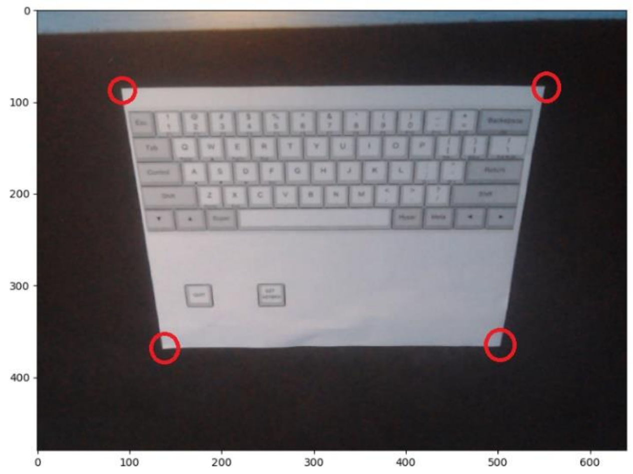
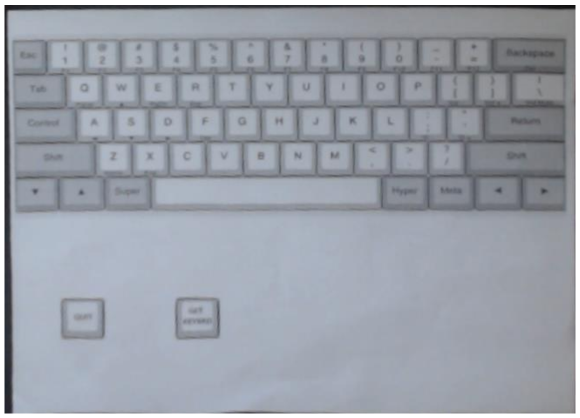
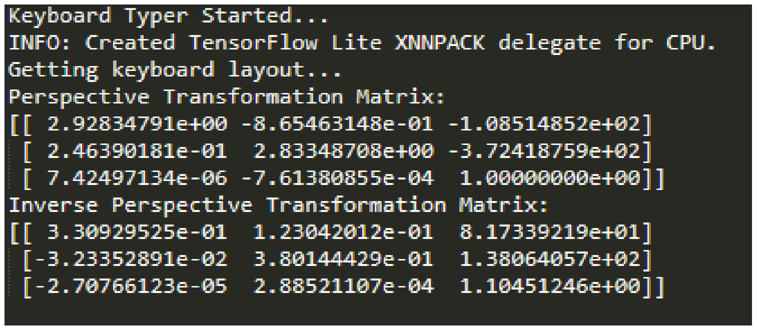


Figure 2.9: Endpoint Landmarks of Paper Keyboard Layout

2.7 Projection Matrix

Using the endpoints of the keyboard layout found in the frame, we can map those points to another coordinate system which we defined as 1400 width and 1000 height, the same ratio of a typical 8.5”x11” paper which our keyboard layout exists on. These endpoints essentially serve as our landmarks to map one coordinate system to the other. By defining and transforming those endpoint coordinates, we are able to generate the projection matrix needed to change from the original camera frame to the top-down view with geometric transformations.

```
Keyboard Typer Started...
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Getting keyboard layout...
Perspective Transformation Matrix:
[[ 2.92834791e+00 -8.65463148e-01 -1.08514852e+02]
 [ 2.46390181e-01  2.83348708e+00 -3.72418759e+02]
 [ 7.42497134e-06 -7.61380855e-04  1.00000000e+00]]
Inverse Perspective Transformation Matrix:
[[ 3.30929525e-01  1.23042012e-01  8.17339219e+01]
 [-3.23352891e-02  3.80144429e-01  1.38064057e+02]
 [-2.70766123e-05  2.88521107e-04  1.10451246e+00]]
```

Figure 2.10: Top-Down View and corresponding Perspective Transformation Matrix

2.8 Key Coordinates

Key coordinates are pre-defined in the "keycoordinates.py" file based on the top-down coordinate system. Since they are defined early, we can map them over the top-down view generated from the above transformation matrix. Using the inverse of that matrix to apply the perspective transformation algorithm gives us each key coordinate of the original input frame as shown below.

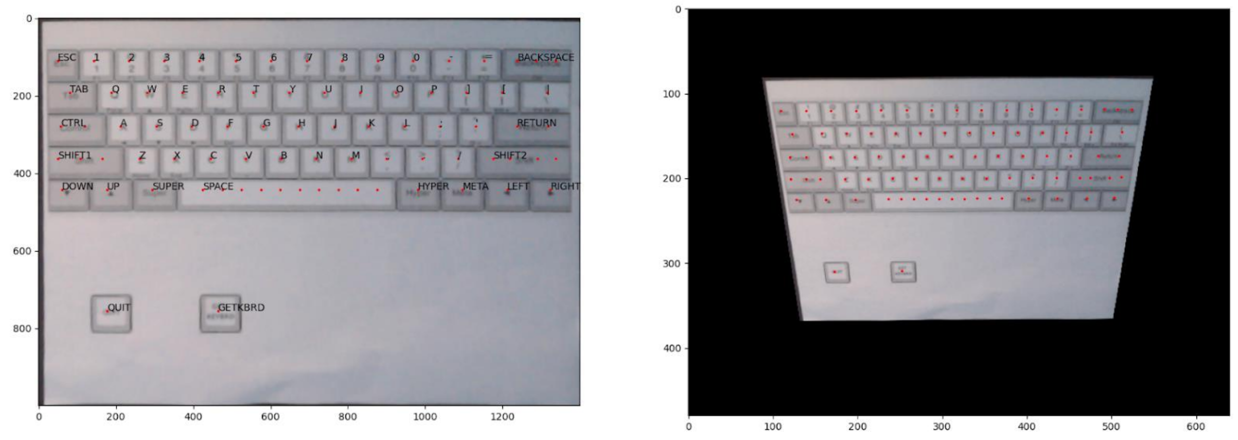


Figure 2.11: Top Down and Inverse Transformed Views of Key Coordinates

2.9 Hand Detection and Finger Tracking

To improve user experience, for the keyboard typer project, accurate hand/finger detection is vital to ensure each key is clicked properly. For example, if the user is going to keep the index finger on the key coordinates around the key "J", then the fingertip coordinates captured must be accurate to correctly infer "J" was clicked. This approach to use MediaPipe Hands was selected due to the robustness of MediaPipe and how accurately they were able to generate hand skeleton frames. Additional research using other approaches such as Convex Hull can be seen later in this report, along with its limitations.

MediaPipe Overview

MediaPipe Hands is a high-fidelity hand and finger tracking solution. It employs machine learning to infer the following 3D landmarks of a hand from just a single frame and can even scale across multiple hands.

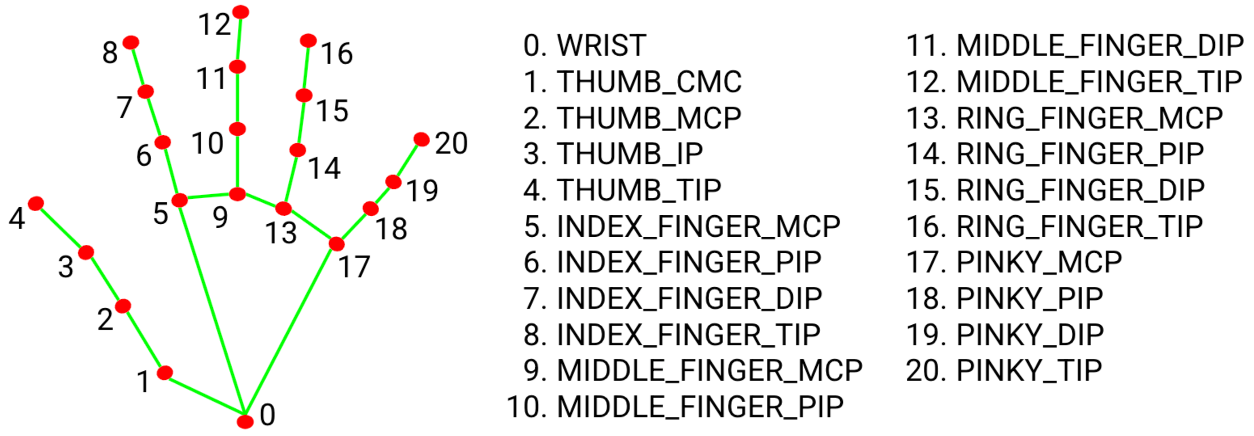

0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Figure 2.12: MediaPipe Hands 3D Landmarks

MediaPipe Hands utilizes their ML pipeline consisting of multiple models captured to create the hand skeleton. These pipelines include a palm detection model and a hand landmark model. The palm detection model works with the entire frame and returns the oriented hand bounding box, allowing for hand recognition and feeds the approximate hand information for the hands pipeline. The hand landmark pipeline then uses that cropped hand bounding box to read and return the suggested 21 3D landmarks shown above. This frees space for us to avoid overhead costs of data augmentation and focus on grabbing the coordinates we need to continue our project with accuracy.

The palm detector was first trained to allow accurate estimation of bounding boxes around the entire hands since it's much simpler to detect entire hands/palms instead of individual fingers. Since palms are smaller objects, MediaPipe used the non-maximum suppression algorithm for edge detections and used square bounding boxes to ignore aspect ratios. An encoder-decoder was used for scene context awareness and focal loss from the camera was also minimized as well.

The hand landmarks are then implemented afterwards given the bounding box of the palm detector, which uses ~30K real-world images with the 21 3D Landmark coordinates for each, even taking into account z coordinate value. A synthetic hand model over various backgrounds were also rendered to cover any additional hand poses to ensure landmark accuracy.

MediaPipe Hands Integration

Our testing of MediaPipe's implementations showed to be very accurate with its palm detection models and hand landmark models. We were accurately able to generate the (x,y) coordinates of each landmark and even the z-coordinates if given a better perspective and more optimal context, surroundings, and lighting.

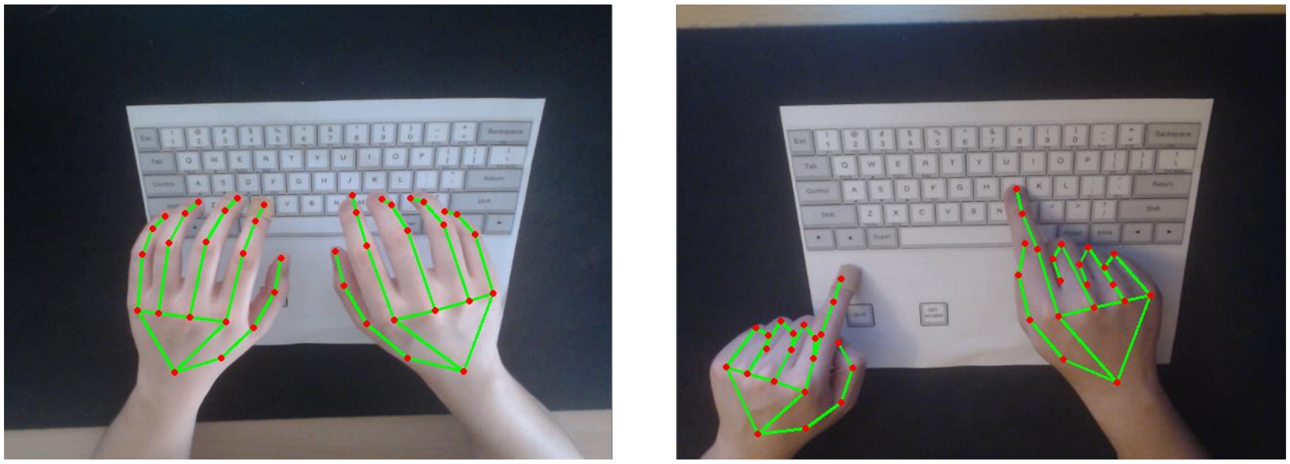
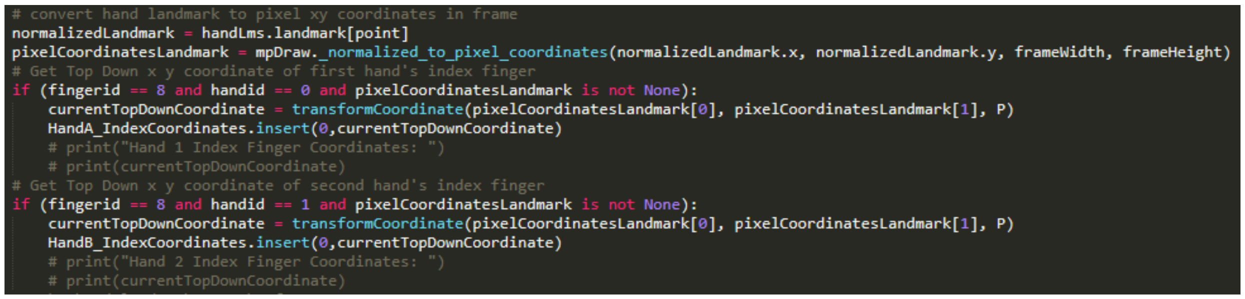


Figure 2.13: MediaPipe Hands Keyboard Typer Testing

Since we were only focused on using the index fingers to determine key clicks, we only had to follow the index fingers of both hands. While initially we had wanted the ability to type with all fingers, the changes in (x,y) coordinates of each finger landmark and inaccurate z-coordinate readings made it difficult to do so. This is further expanded upon in 3.4 of the report. Instead we are capturing both hands and filtering them out by landmarks to determine the (x,y) coordinates of each index finger. As per the figure above, this landmark happens to be given the ID number 8, and is filtered as shown in the code below:



```
# convert hand landmark to pixel xy coordinates in frame
normalizedLandmark = handLms.landmark[point]
pixelCoordinatesLandmark = mpDraw._normalized_to_pixel_coordinates(normalizedLandmark.x, normalizedLandmark.y, frameWidth, frameHeight)
# Get Top Down x y coordinate of first hand's index finger
if (fingerid == 8 and handid == 0 and pixelCoordinatesLandmark is not None):
    currentTopDownCoordinate = transformCoordinate(pixelCoordinatesLandmark[0], pixelCoordinatesLandmark[1], P)
    HandA_IndexCoordinates.insert(0,currentTopDownCoordinate)
    # print("Hand 1 Index Finger Coordinates: ")
    # print(currentTopDownCoordinate)
# Get Top Down x y coordinate of second hand's index finger
if (fingerid == 8 and handid == 1 and pixelCoordinatesLandmark is not None):
    currentTopDownCoordinate = transformCoordinate(pixelCoordinatesLandmark[0], pixelCoordinatesLandmark[1], P)
    HandB_IndexCoordinates.insert(0,currentTopDownCoordinate)
    # print("Hand 2 Index Finger Coordinates: ")
    # print(currentTopDownCoordinate)
```

Figure 2.14: Code Snippet of Hand Landmarks

2.10 Determining Key Clicks

Key clicks are determined by the pixel coordinates of the index fingers of both hands. The script is constantly saving the last 75 frames (x,y) coordinates of each index finger. If an index finger sits in an area for those 75 frames (given some leniency of 5 frames in case there the hand recognition piece glitches during that time since it's not perfect), we then transform those coordinates to the top-down view and find the closest key coordinate within the index fingers range. This is done by implementing using the distance formula, where it looks for the closest key within a range of 100 pixels in the top down view.

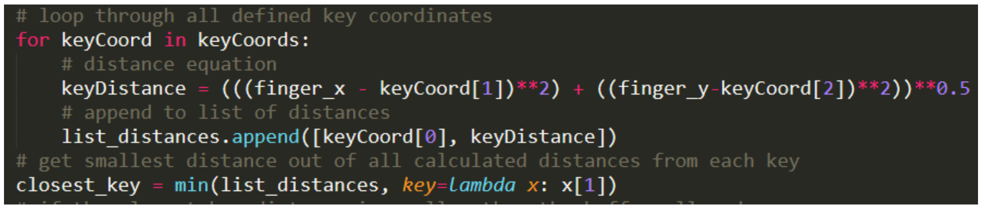
$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

d = distance

(x_1, y_1) = coordinates of the first point

(x_2, y_2) = coordinates of the second point

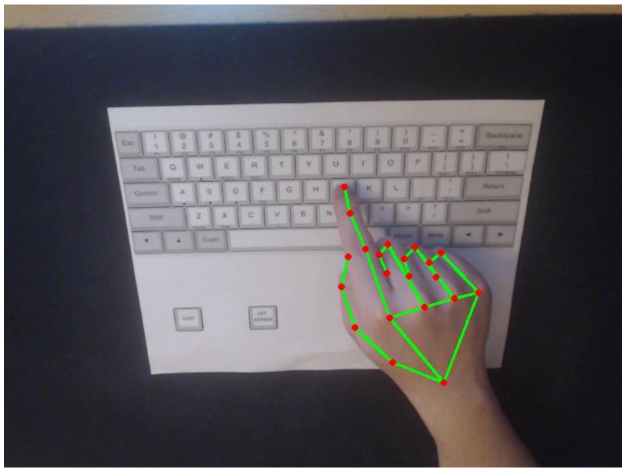
Figure 2.15: Distance Formula Overview

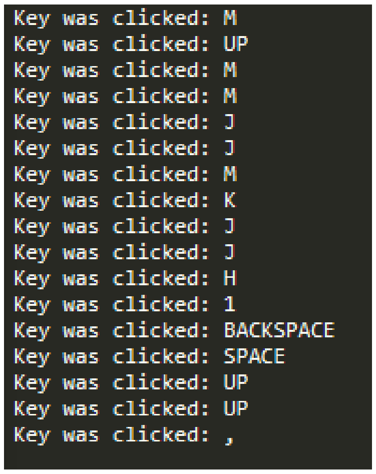


```
# loop through all defined key coordinates
for keyCoord in keyCoords:
    # distance equation
    keyDistance = (((finger_x - keyCoord[1])**2) + ((finger_y-keyCoord[2])**2))**0.5
    # append to list of distances
    list_distances.append([keyCoord[0], keyDistance])
# get smallest distance out of all calculated distances from each key
closest_key = min(list_distances, key=lambda x: x[1])
```

Figure 2.16: Code Snippet of Distance Formula

If the finger coordinate is within range of that key coordinate, it will consider that button pressed and reset the coordinate frames. This is then printed to the command line or output panel to show which button was pressed.





```
Key was clicked: M
Key was clicked: UP
Key was clicked: M
Key was clicked: M
Key was clicked: J
Key was clicked: J
Key was clicked: M
Key was clicked: K
Key was clicked: J
Key was clicked: J
Key was clicked: H
Key was clicked: 1
Key was clicked: BACKSPACE
Key was clicked: SPACE
Key was clicked: UP
Key was clicked: UP
Key was clicked: ,
```

Figure 2.17: Demo of the Paper Keyboard Typer

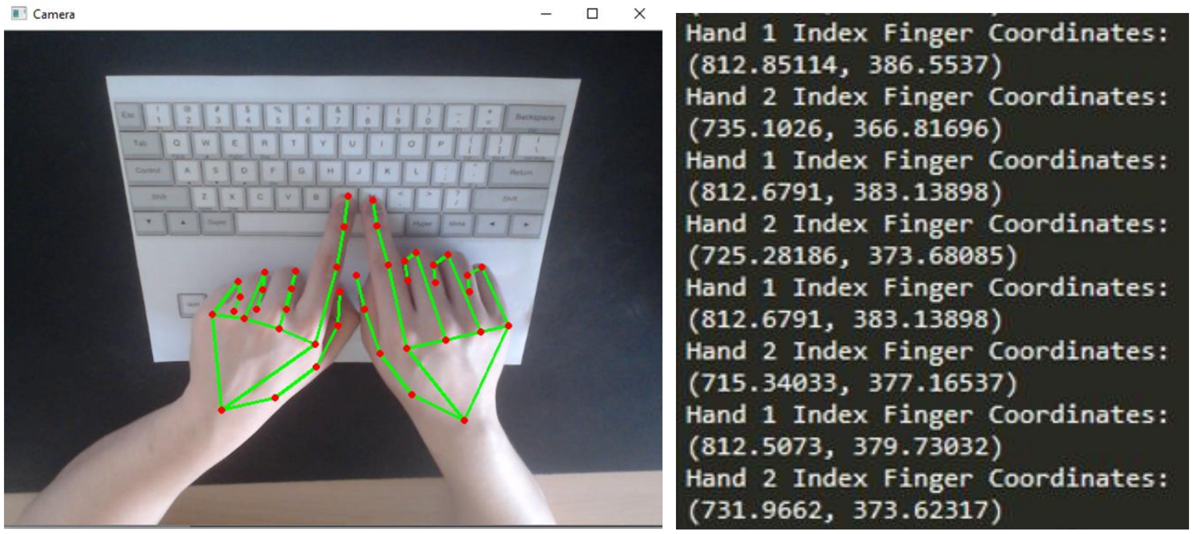
2. **Alternative Approaches**

We have tried several approaches to improve the accuracy of our finger and key detection so that it gives the correct keypress when the fingers are moving. We faced several challenges which were mainly due to limitations in the quality of the camera and lack of multiple cameras.

3.1 Tracking sudden changes in the finger coordinates

Our initial idea to accurately detect which finger is currently moving to press a key was to closely track the coordinates of each of the fingers, and define a threshold change value to conclude whether the finger pressed a key or not. However, this method did not prove effective as it was unrealistic to define a threshold that can balance the speed of taking the image frames as well as correctly detect a finger position change which was meant as a movement to click.

If we set the threshold value to a low change, it led to the incorrect deduction that a key was pressed even when the user was moving within the same key range or repositioning his hands. Moreover, when the threshold value is set too low, the movements would be very small, and this, in turn, would require a fast frame data, where we take many frames per second and analyze the change in finger coordinates in close intervals, and map it back to the key if the change is above the threshold, i.e, we track the finger at very high intervals so that we don't miss any movement data. Due to the high number of false positives, we decided that we need to set a higher threshold.



```
Hand 1 Index Finger Coordinates:
(812.85114, 386.5537)
Hand 2 Index Finger Coordinates:
(735.1026, 366.81696)
Hand 1 Index Finger Coordinates:
(812.6791, 383.13898)
Hand 2 Index Finger Coordinates:
(725.28186, 373.68085)
Hand 1 Index Finger Coordinates:
(812.6791, 383.13898)
Hand 2 Index Finger Coordinates:
(715.34033, 377.16537)
Hand 1 Index Finger Coordinates:
(812.5073, 379.73032)
Hand 2 Index Finger Coordinates:
(731.9662, 373.62317)
```

Figure 3.1: Checking the changes in finger positions

When we use a higher threshold, we observe more accurate keypress recognition, however, this requires that the user might have to move their fingers more than they usually do while typing when the threshold is too high. We also saw some issues when the user is pressing nearby keys that the program did not recognize as correct presses because the change was not enough to trigger the threshold. Hence it is required in this approach to find a good balance for the threshold.

Ultimately, we decided not to go with this approach because each individual's fingers are of different lengths and sizes and setting a common threshold would not work. Moreover, people type at different speeds, and their movements are personal traits. It could however be possible to set these as more complex parameters, which could be calibrated at the start of the program to achieve more accurate results and could be taken up as future work.

3.2 Convex hull and contours for hand detection

We also experimented with using contours and Convex hulls in our implementation. The main idea was that we would use contours to define the finger shapes and in turn get the coordinates of the tips. We take a set of contour points and then set the correct parameters to get the appropriate contour curve, which in this case is the outline of the hand. To make this process easier, we used canny edge detection to make the image frames binary and then used active contour snakes to get the corresponding contour of the hand. We then used the concept of convex hulls to make a boundary. The convex hull of an object is defined as the minimum boundary that can completely wrap the object.

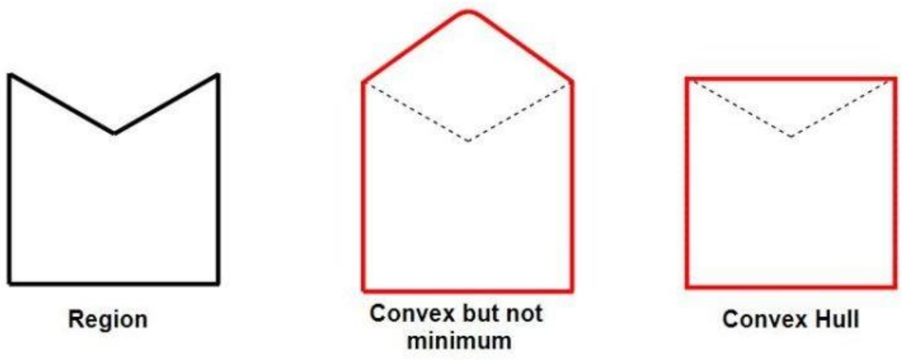


Figure 3.2: Convex Hull Idea

Using our convex hull data for the image frame, we can then easily get the contours of the hand, and in turn the fingers. Using the contours of the hand and the convex hull, we can find the convexity defects for each image frame, this allows us to find out which sections are the fingers and in turn differentiate the fingers. We can see the usefulness of this from the figure below, which shows the contours, convex hull, and convexity defects.

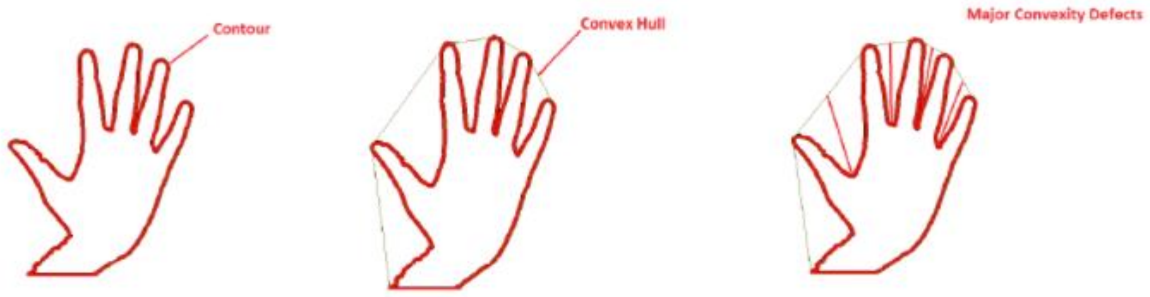


Figure 3.3: Contours, Convex Hull and Convexity Defects

While we were able to correctly get the fingers and differentiate them, the program did not work as expected when we used it on top of a paper. It included the background of the content present on the paper when calculating the convexity defects, this could lead to incorrect results when using the keyboard paper. The results varied greatly depending on the quality of the image as well as the lighting.

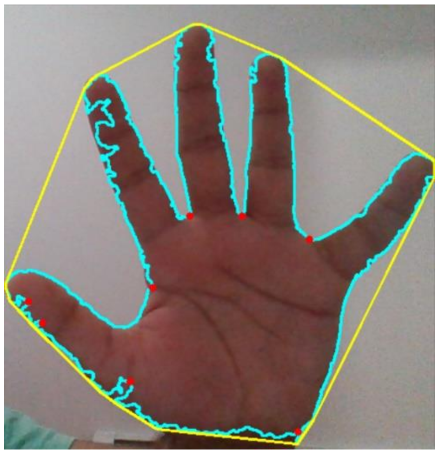


Figure 3.4: Result of Convex Hull approach

As we can see from the figure below, the black image on the paper is causing the results of the points, represented in red to be incorrect. This affects the coordinates which we are calculating to find the position of the fingers, in order to decide which key should be recorded as the input. While we could blur the background, for the keyboard type, we would need the small outlines of the paper keys to figure out where the keys are, and ultimately which key is being pressed, so we need to be careful that this blurring does not affect the detection of the keyboard outline itself or the individual keys present on the paper.

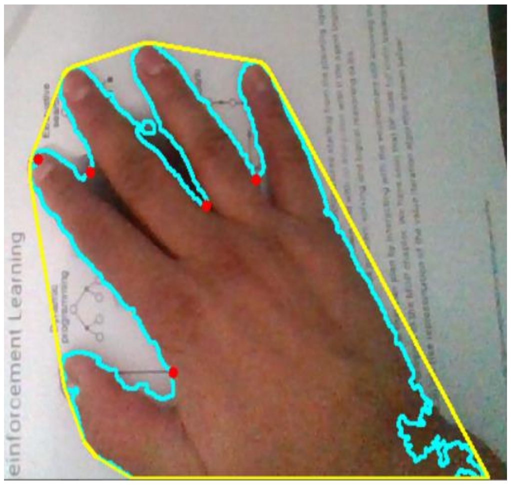


Figure 3.5: Issues with Convex Hull approach

Due to the unreliability of the results, we decided not to go with this approach. However, it is worth noting that with a better camera and appropriate blurring of the background, it could be possible to achieve better results.

3.3 Optical Character Recognition

We also explored the possibility of using OCR to detect the key locations. We know that the keys are of fixed size on the paper, hence we could use OCR to find the letters on the keyboard, as well as their location. We could then use the knowledge of the finger locations at the current time, to figure out which key should be pressed. Another approach would be to use OCR to detect the keys at the current time, and we know the list of keys on the paper keyboard. Using these two as information, we can then conclude that the key which is not detected by the OCR, is one being covered by a finger, and hence should be recorded.

In practice, this did not work, as the user's hand covers a bigger part of the keyboard than just the current key which is being pressed. There is also the issue that the camera is often at an angle, and might not always be a top view, and this could hinder the OCR’s ability to effectively recognize the keys at certain angles. Also, the quality of the keyboard image and the image frame affect the results. As we can see from the image below, the ‘Shift; and similar other keys cannot be recognized easily using OCR, this would cause issues with our approach.

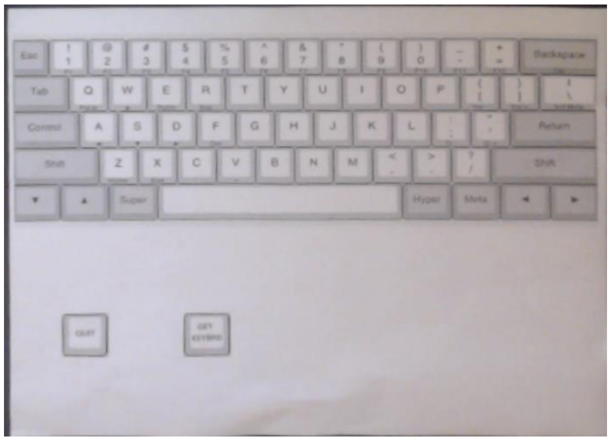


Figure 3.6: Poor image quality affecting OCR

Hence, we decided not to use OCR to detect the keys, however, this led to another solution. As we know that the paper image will be constant, we can then transform the image using geometric transformations. This results in a top view of the keyboard. Using this we can define regions for each key, taking the corners of the paper keyboard as correspondences. Now that we know the area of each key and which position it is with respect to the keyboard edges, using the location of the fingers, we can then safely say that the finger is over the area of a certain key, and hence should be pressed. Although the OCR approach resulted in not working, it helped us find a way to a solution we can use. It is also possible to take up this OCR method as future work. With a better camera and a setup of multiple cameras, we can possibly still implement the OCR method to detect the keys and decide which one should be pressed.

3.4 Two webcam approaches to determine finger click

The issue we ran with our current implementation was that with only one camera, we can generate only one angle of the image frame, i.e, top-view of the keyboard and hands. This leads to a huge problem. With only the top-view, we can only accurately get the *(X, Y)* coordinates of the fingers, and we don't have any real insight into the '*Z*' coordinate value. Without knowing this, we cannot effectively determine whether the finger is simply above the key or if it should actually be pressed.

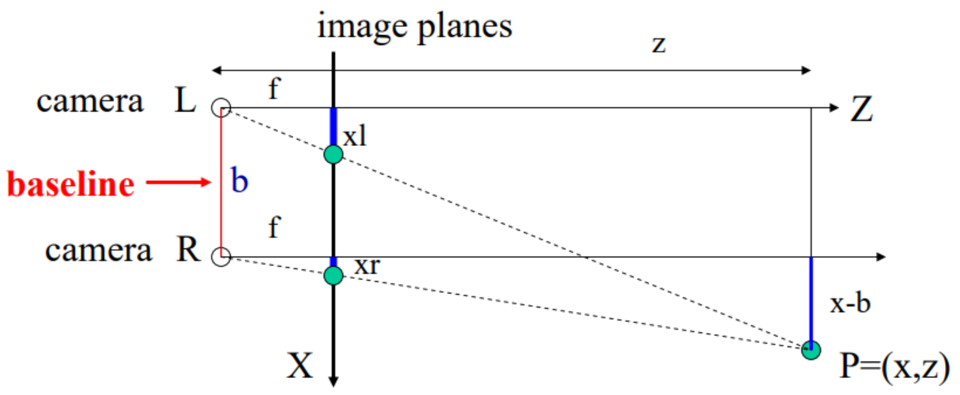


Figure 3.7: Two cameras set up

Our idea was that, with the help of two cameras, we could use the concept of Stereo cameras to calculate the disparity image at a certain image frame time. For example, at a time *t*, we can take two images of the hand, over the paper keyboard. Using the two images, we can calculate the disparity, which in turn helps us calculate the depth of the image. Using the depth, we can find the Z coordinate of the respective fingers. Setting the paper keyboard as the origin, we can now conclude that if the finger is at *(X, Y)* over a key area, and the *Z* value is at zero, the user is over a certain key, and is also touching the paper keyboard, i.e, a press was made and that it should be recorded.

$$disparity = (distance\,between\,cameras * focal\,length)\,/\,depth$$

Figure 3.8: Formula for calculating the disparity

Another simpler approach would be to have one camera perpendicular to the other, that is one watches the *(X, Y)* plane and the other watches the *(Y, Z)* plane. At a time, *t*, camera one can find the position of the finger over the keyboard, and camera two calculates the distance between the fingers and the paper keyboard. Again, knowing that the finger is over a certain key, as well as that it's touching the paper keyboard, we can then safely and accurately conclude that a key is indeed being pressed and hence should be recorded. This method is also viable as it does not require the calculation of the *Z* coordinate, and hence save time in computing.

However, due to not having the required hardware, we were not able to implement this approach. We are confident that this approach would be the best choice of action to improve our current model, and would greatly increase the flexibility of usage as well as accuracy. We would definitely focus on this for our future work, as it seems very promising and would enable us to add more complex features such as multi-finger typing, touch typing, and various keyboard layouts.

3. Experiments

4.1 Data

For the keyboard layout, we used a specific keyboard layout used to determine which key location is at which point. The pixel coordinates data for each key is saved in the script itself and the expectation is that the user will be able to print the exact keyboard document as below with the same layout.

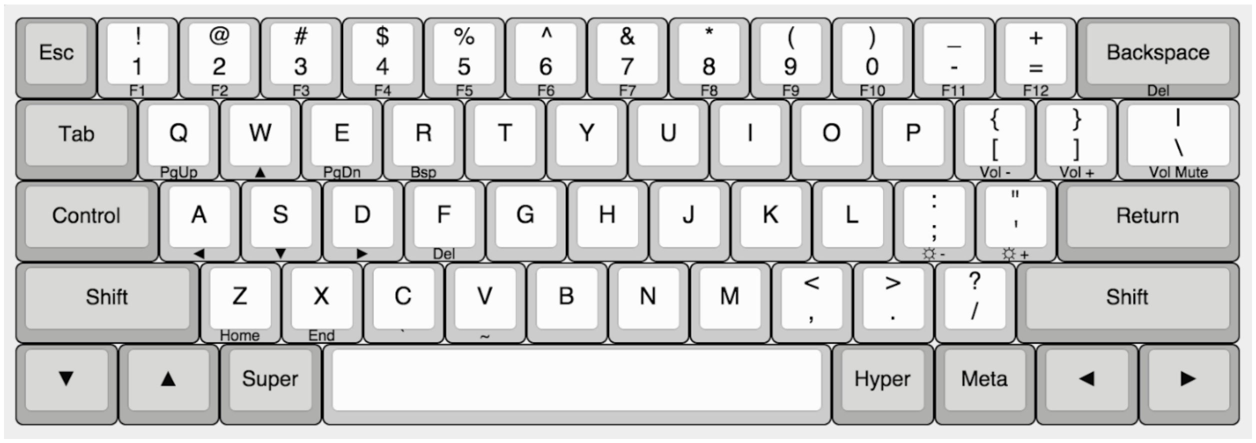


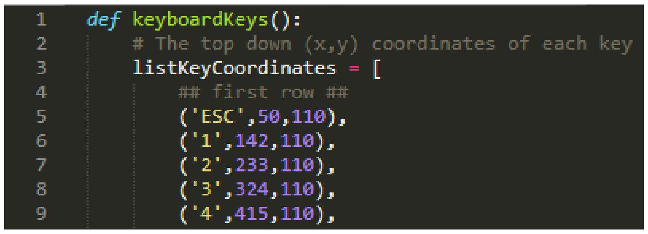
Figure 4.1: Keyboard Layout

Additional keycaps were also added for a Quit and Get Keyboard functionality to allow the user to quit out of the Keyboard Typer script or re-grab the keyboard location.

QUIT
GET KEYBRD

Figure 4.2: Additional Keys

Using this exact layout as saved in the word document in the code, the python script will always know the location of all 80 key coordinates as the top-down view will always be generated and hold the key coordinates in the same locations. This list is seen in the "keycoordinates.py" file.



```
def keyboardKeys():
    # The top down (x,y) coordinates of each key
    listKeyCoordinates = [
        ## first row ##
        ('ESC',50,110),
        ('1',142,110),
        ('2',233,110),
        ('3',324,110),
        ('4',415,110),
```

Figure 4.3: Top-Down Key Coordinates

4.2 Results

4.2.1 Success Cases:

In most cases we are able to determine the coordinates of each key thanks to the script being able to identify the end points of the paper. In our testing, we were able to use a black background below the white paper keyboard layout resulting in accurate canny edges. From there contours were generated and the endpoints are determined based on the largest contour, which would be the keyboard layout. As shown in our below results, this generated an accurate top down view where our key coordinates are listed in the expected locations. Additionally our transformation matrix is additionally tested as shown how we were able to transform back to our original frame and how each key coordinate is displayed in the correct position.

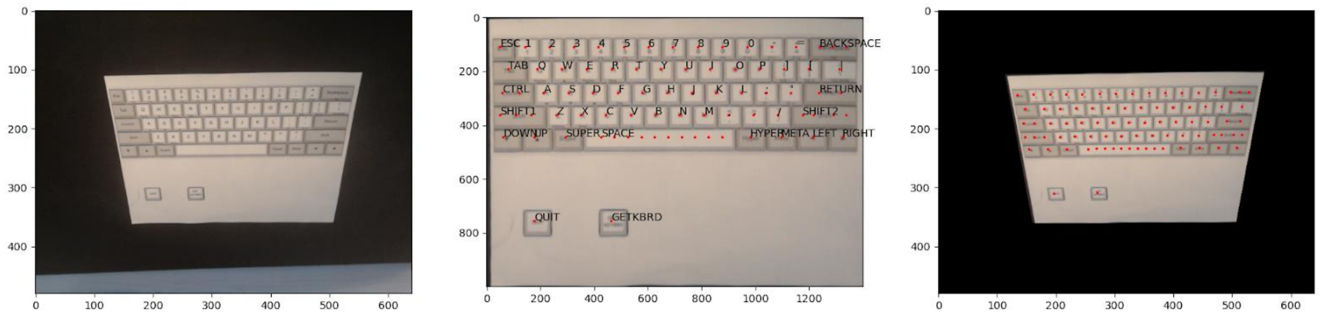


Figure 4.4: Successful Keyboard Transformation

4.2.2 Issues:

Because of the methodology chosen on how we're processing the input frame, there are definitely limitations on how we are determining which contour rectangle is actually the keyboard. In the example below, because the paper is cut off the methodology used to determine the end points of the entire paper pointed to a specific key since that itself had a contour. In our example below, just the "Y" key is seen as the keyboard since it is a closed contour, resulting in all the key coordinates being saved around the Y key.

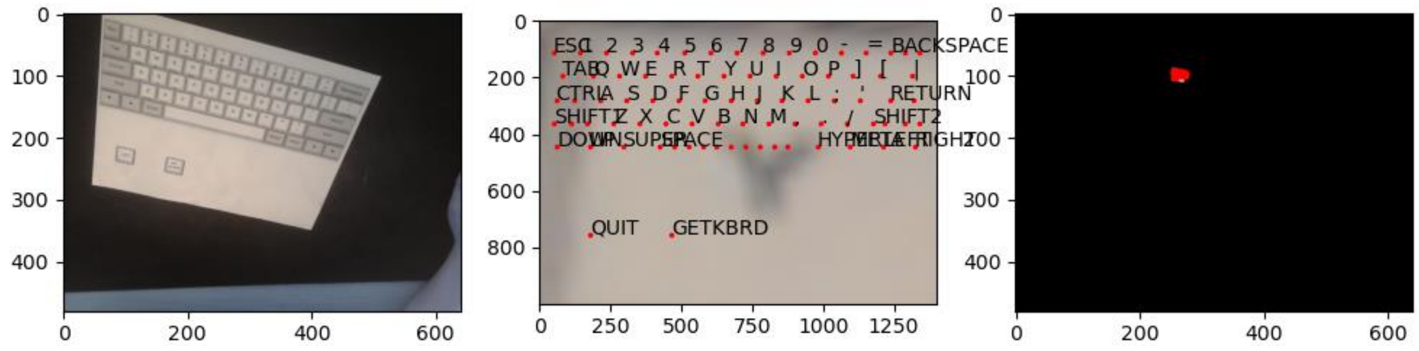


Figure 4.5: Corners detected on "Y" key

This below example shows what could happen if the keyboard is completely sideways. Since we are determining the endpoints of the paper based on horizontal and vertical maximum and minimum, it does not take into account how the paper is positioned. As a result, the sideways keyboard layout was mapped to the defined top-down coordinate system which we defined as 1400 by 1000 pixels and the projection matrix is calculated as such.

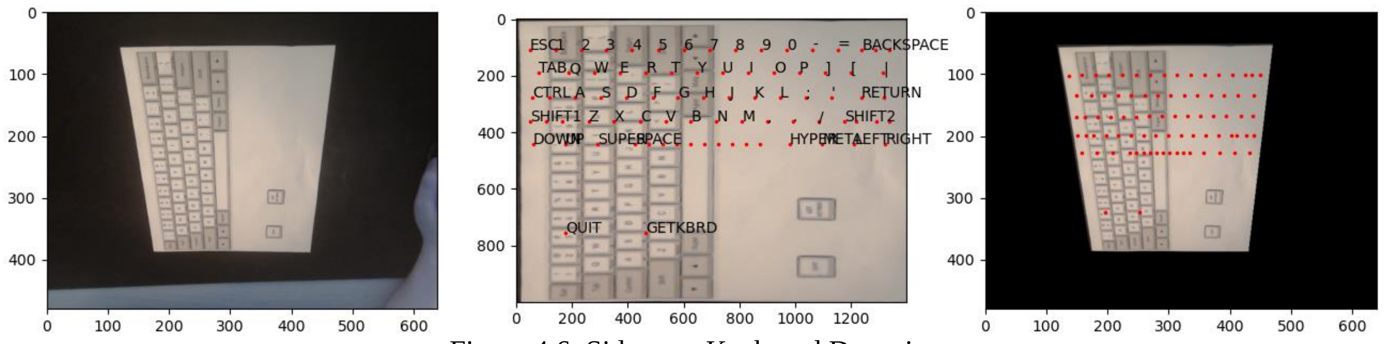


Figure 4.6: Sideways Keyboard Detection

4.2.3 Testing:

While difficult to show in report form, we were able to test every single key click needed from the keyboard to show. The confidence of how accurate our edge detection is also supported by the photo output of the key coordinates on top-down view of the frame itself.

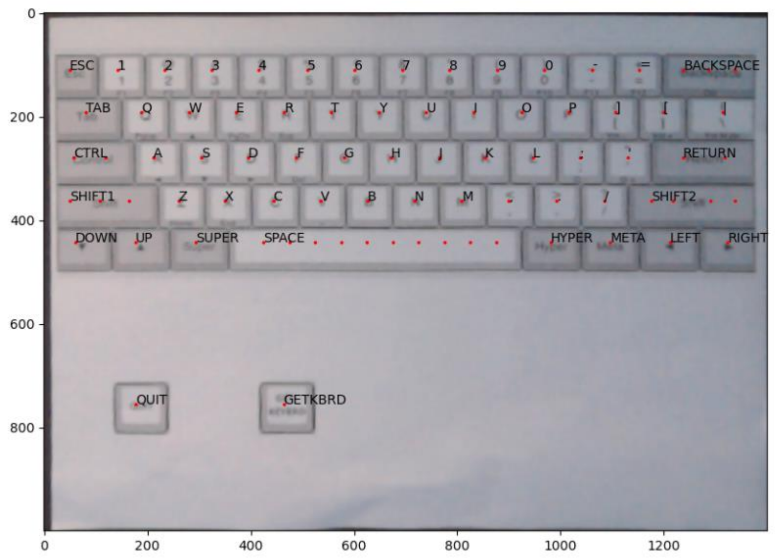
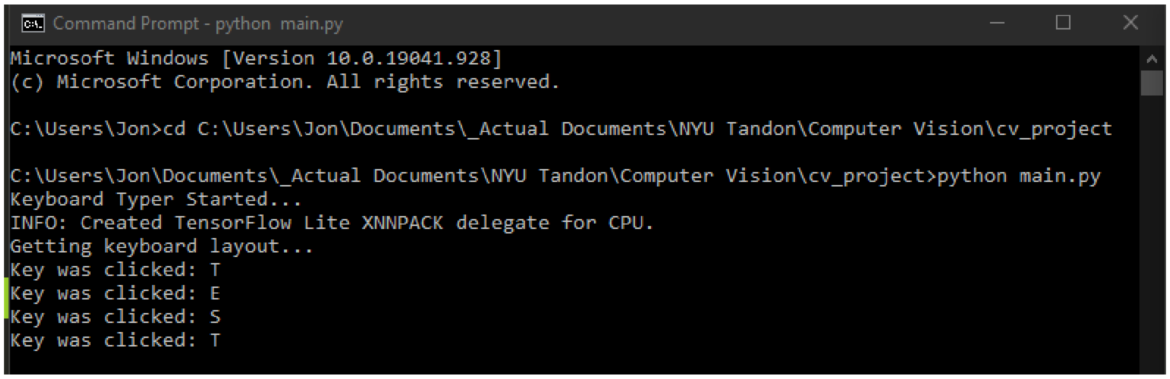


Figure 4.7: Key Coordinates on Top-Down View

While we tested every single key coordinate, below are examples of certain letters being typed can be seen below. The words “TEST” and “HELLO (Return)” can be seen generated by the paper keyboard typer project.

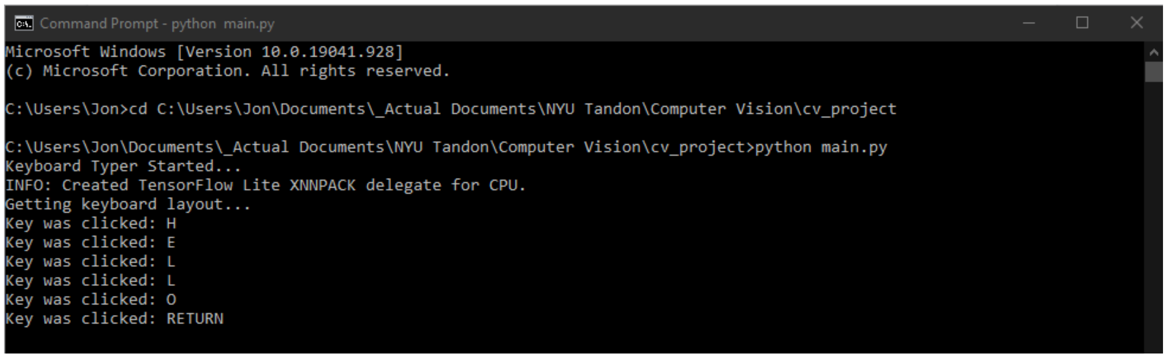


```
Microsoft Windows [Version 10.0.19041.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Jon>cd C:\Users\Jon\Documents\_Actual Documents\NYU Tandon\Computer Vision\cv_project

C:\Users\Jon\Documents\_Actual Documents\NYU Tandon\Computer Vision\cv_project>python main.py
Keyboard Typer Started...
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Getting keyboard layout...
Key was clicked: T
Key was clicked: E
Key was clicked: S
Key was clicked: T
```

Figure 4.8: Test Output “TEST”



```
Microsoft Windows [Version 10.0.19041.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Jon>cd C:\Users\Jon\Documents\_Actual Documents\NYU Tandon\Computer Vision\cv_project

C:\Users\Jon\Documents\_Actual Documents\NYU Tandon\Computer Vision\cv_project>python main.py
Keyboard Typer Started...
INFO: Created TensorFlow Lite XNNPACK delegate for CPU.
Getting keyboard layout...
Key was clicked: H
Key was clicked: E
Key was clicked: L
Key was clicked: L
Key was clicked: O
Key was clicked: RETURN
```

Figure 4.9: Test Output “HELLO”

Additionally, we can confirm that our matrix and transformation is working as intended by the output figures shown in the “supplemental” folder of the code along with the above images in the “Approach” section of this report.

4. Conclusion

We were able to complete the working implementation of our idea but faced several challenges throughout the implementation. We were able to take in the video feed from our webcam, take image frames at certain intervals, and with the help of edge detection and contours, detect fingers in the image. We were also able to take image frames of the paper keyboard, and using geometric transformations, enabled us to accurately assign areas for each key on the paper keyboard. Combining these two methods, we were able to produce an implementation that tracks the fingers and decides which key should be recorded.

While we were able to create a functioning paper keyboard, it is far from being a viable product. We still need to greatly improve the number of features and flexibility of its usage. Currently, our implementation only allows the use of only one finger at a type to type, and it is quite slow. It would not be useful for a fast type or someone interested in using it as a full-time solution. However, it is still useful for certain situations, and slow typers. It could be functional for critical situations when a keyboard is not present and you need to get some typing done. It could be integrated with a phone camera, or a mobile application, which can essentially allow us to always have a simple keyboard in our pockets.

Strengths:

- It is a simple implementation that can be packaged very easily.
- Easy to convert into a mobile application.
- User friendly and not complicated to run.
- Useful in certain situations when an impromptu keyboard is needed
- It is quick and gets the task done as expected.
- Due to its simplicity, it is also robust and works without errors.
- Can be customizable.
- Less hardware needed, only a camera.

Weaknesses:

- Only covers single finger typing.
- Bit slow due to the wait time to register a press.
- Lighting could greatly affect the results.

We did learn a great deal about the various concepts used in building the program and look forward to adding several features and improving the current version, this is discussed in the future work section of this report.

5. Future Work

We could make several improvements to our project. The main focus of our future work would be to use the two-camera approach we discussed earlier to set up a system where we can calculate the 3D coordinates of each of the fingers. This will allow us to add the feature of multiple finger typing, which would be a major improvement. It would then act as a full-fledged keyboard.

Secondly, we wish to add more features such as a paperless keyboard, and get rid of the keyboard altogether, this can be achieved by having only four corner fiducials, and then the user can set up those on any surface. We can then show a virtual keyboard on the screen, and just track the fingers using the fiducials for reference. Features we wish to add are multiple layouts, different language keyboards, and even add trackpad capabilities. This would reduce the need for hardware all together, and would only require a camera. We also wish to explore the possibility of using a single camera to take two pictures, from different angles, by changing the camera properties such a focal length and aperture and see if we can calculate the 3D coordinates using the approach, this would eliminate the need for two cameras, and make it more portable, or even executable via just a smartphone.

We also wish to test and experiment with the results using better quality cameras and see if that helps reduce some of the false positives, and also not miss any keystrokes. We will also explore the addition of using deep learning to train a model to accurately decide where the key has been pressed or if it is a false positive, this would entail generating data as well. The use of machine learning can greatly improve the reliability of our program.

Lastly, in real world situations, there are often interferences, and the hand movements differ from person to person, we would like our application to be robust to the point that anyone can use it, and that it is flexible enough to handle interferences and situations it was not specifically developed for. We believe that with the use of more complex computer vision techniques and machine learning algorithms, this is achievable and look forward to traversing this path of development.

6. References

- https://medium.com/analytics-vidhya/contours-and-convex-hull-in-opencv-python-d7503f6651bc
- https://towardsdatascience.com/a-gentle-introduction-to-ocr-ee1469a201aa
- https://google.github.io/mediapipe/solutions/hands.html
- https://courses.cs.washington.edu/courses/cse455/09wi/Lects/lect16.pdf
- https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html
- https://www.dspguide.com/ch25/4.htm
- https://google.github.io/mediapipe/solutions/hands.html
- https://github.com/google/mediapipe