Image Analysis and Computer Vision

Project Documentation

Mohanad Diab

10769366

Contents

1 Introduction

This project aims to solve the problem of musician hands recognition and position estimation, the plan is to use state-of-the-art computer vision techniques to accurately track and analyze the movement of a musician's hands while they play instruments such as piano, guitar, and accordion. The use of computer vision in the music industry is not a new concept, but it has come a long way since its inception. In the past, computer vision was mainly used for simple tasks such as tracking the movement of a musician's fingers on a keyboard or frets on a guitar. However, with advancements in technology, computer vision has become much more sophisticated, allowing for more detailed analysis of a musician's technique and performance.

I believe that this information can greatly benefit both novice and professional musicians by improving the learning experience. For example, by analyzing the movement of a musician's hands, we can identify areas where they may be struggling and provide targeted feedback. Furthermore, by studying the technique of accomplished musicians, we can gain a deeper understanding of what makes a great performance.

Moreover, computer vision technology in music industry can be useful in music research. By analyzing the performance of musicians, researchers can gain insights into how music is perceived and processed by the human brain. This can lead to a better understanding of how music affects our emotions and can have implications for music therapy.

I am excited to delve into the world of music and technology, and discover the potential for improving the way we learn and understand music. Through this project, I hope to contribute to the growing field of computer vision in music, and make a meaningful impact on the way we learn, perform, and appreciate music.

2 Requirements

The required outputs of this project, is to construct a model that takes the videos of musicians who are playing their respective instrument, and produces as output the motion of the fingers in their 3 dimensions, some instruments to consider like Piano, Guitar and Accordion.

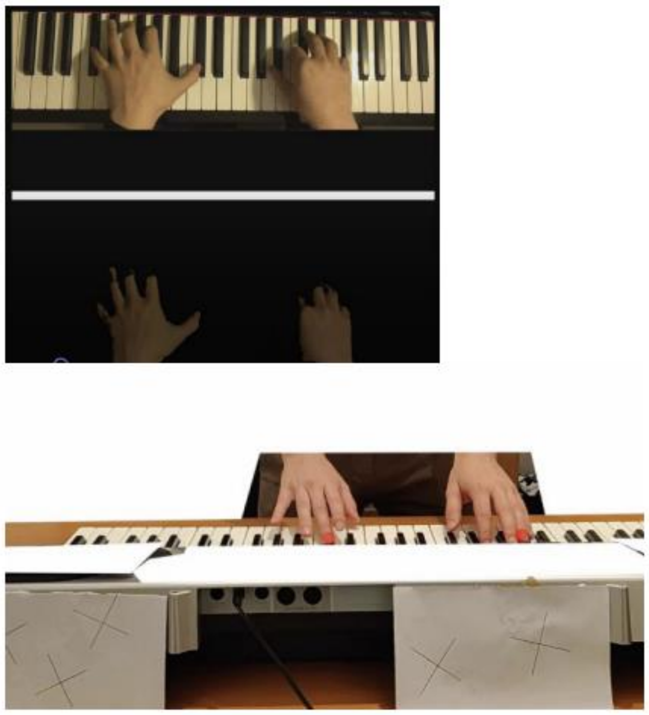


Figure (2.1): an example of the requirements

The previous explanation, it is stated that the program must accept a series of frames and try to detect the hand movements in the all the sequential frames to detect their image coordinates in their x and y components, and somehow exploit the information in the frames to build an understanding of the third z component.

The challenge relies in the fact the monocular cameras like the everyday cameras we use, do not provide stereoscopic scenarios that allow one in these cases to construct a 3D representation of reality, instead we have one video, which is mostly static and provides a singular view (one angle), which does not allow us to use the intuitive methods.

The principle of stereoscopic vision, which allows for 3D reconstruction of the reality, is used daily by human beings, where the two eyes provide 2 observation of the same object seen in 2 different angles, which allows the rays of light to intersect at the point where the surface is and ultimately producing a depth dimension to the images.

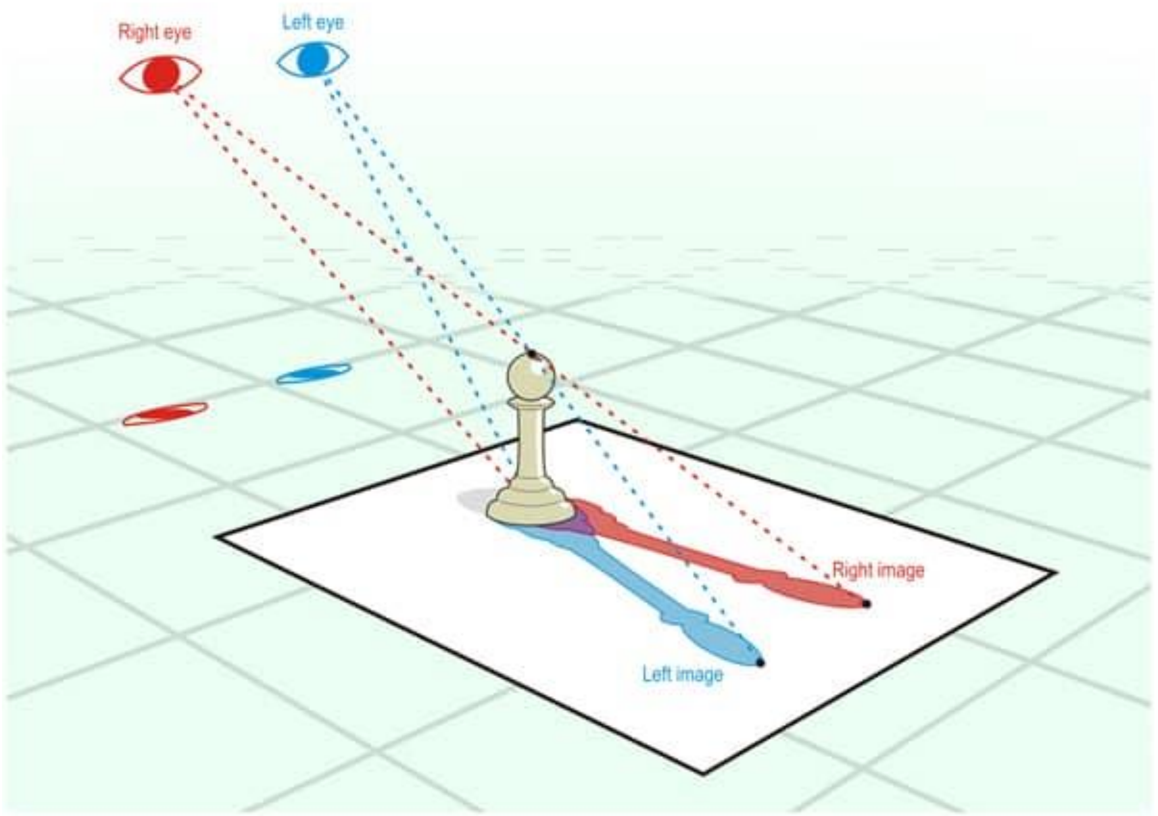


Figure (2.2): Stereoscopic vision

But in the case of a monocular camera, there are no 2 instances of light to intersect, which means there is no way to tell depth from the image, but humans do it anyways, sense their brain is used to the scenario it takes the liberty to adjust to the depth dimension compensating for the lack of the 3rd dimension in the image.

On the other hand, computers can’t do that, since they can only produce output equivalent to the input, which is in 2d.

To compensate for the lack of a second or multiple images, some methods have been advised in this project with some promising results, although the field is not yet fully ripe and requires more investigation to achieve the convention on how this problem should be approached, some of the models and methods used in this document could prove to be useful in given scenarios.

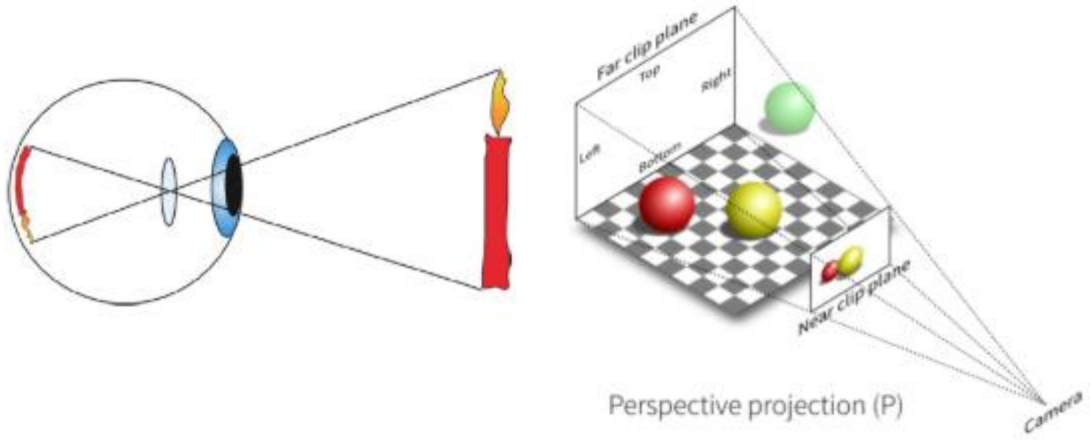


Figure (2.3): How the perspective projection views reality

3 Preliminary research

3.1 HAAR

Haar cascades are a type of object detection method that can be used for hand detection. It is based on the concept of Haar-like features, which are simple mathematical representations of an object's shape. The Haar cascade classifier works by analyzing the image in multiple scales, and at each scale, it applies a set of Haar-like features to each window of the image. The features are used to calculate a "feature vector" for each window, which represents the characteristics of the object in that window. A classifier, which is a trained machine learning model, then decides whether the window contains the object or not.

To use Haar cascades for hand detection, a dataset of hand images is needed to train a classifier. The dataset should contain a variety of hand poses and sizes, so that the classifier can generalize well to new images. Once the classifier is trained, it can be applied to new images or video streams to detect hands. The output is a set of bounding boxes that enclose the detected hands.

One of the main advantages of Haar cascades is that they are relatively simple and fast. They are a good choice for applications where real-time performance is important and computational resources are limited. Additionally, Haar cascades have been widely used in object detection and have been proven to work well in practice.

However, Haar cascades are considered as an older method of object detection, and in recent years, other methods such as CNNs and YOLO have become more popular due to their improved accuracy. Additionally, Haar cascades are not able to detect multiple objects in the same image or to detect objects with different scales, which are common limitations for this method.

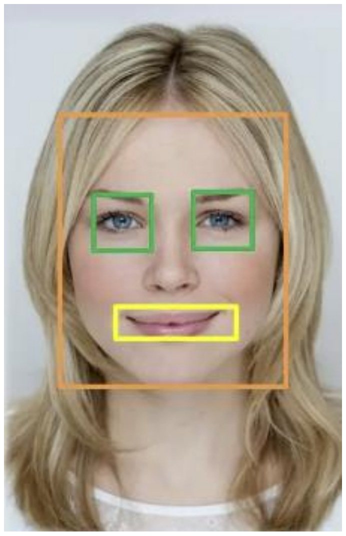


Figure (3.1): HAAR example

3.2 MediaPipe

Mediapipe is an open-source framework developed by Google that can be used for various computer vision tasks, including hand detection. The framework provides pre-trained models and a set of tools to help developers build custom computer vision applications quickly and easily.

To use Mediapipe for hand detection, developers can use the pre-trained hand tracking model provided by the framework. This model uses machine learning techniques to detect and track the movement of hands in an image or video stream. The model is trained to recognize a wide variety of hand poses and sizes, making it suitable for use in different applications.

Once the hand tracking model is integrated into the application, developers can use the Mediapipe library to process the input video and detect hands in the frames. The output of the model is a set of key points that correspond to the joints of the hands, such as the fingertips and knuckles.

Additionally, Mediapipe also provide some useful hand landmarks, like hand palm and fingers, which can be used to do some further analysis of the hand's movement, like playing a piano, guitar or any other instrument.

Figure (3.2): Mediapipe example

3.3 Yolo

YOLO (You Only Look Once) is a popular real-time object detection system that can be used for hand detection. The YOLO algorithm works by dividing an image into a grid of cells and then using a convolutional neural network (CNN) to predict the presence of objects within each cell.

To use YOLO for hand detection, developers would need to train a custom YOLO model using a dataset of hand images. The dataset would need to contain a variety of hand poses and sizes, so that the model can generalize well to new images.

Once the model is trained, it can be used to detect hands in new images or video streams. The output of the model is a set of bounding boxes that enclose the detected hands, along with a confidence score indicating how likely the model believes the detection to be correct.

One of the main advantage of YOLO is that it is real-time, meaning that it can process images very quickly, making it suitable for use in applications where fast detection is required. Additionally, YOLO is relatively simple and easy to implement, making it accessible to developers with a wide range of experience levels.

However, YOLO may not be the best choice for hand detection if high precision is required. YOLO's main purpose is to detect objects in real-time, not to get precise coordinates of the object, like in the case of hand landmarks. Additionally, YOLO's architecture is optimized to detect objects in general, not only hands, so it may not be as accurate as a model that is specifically trained to detect hands.

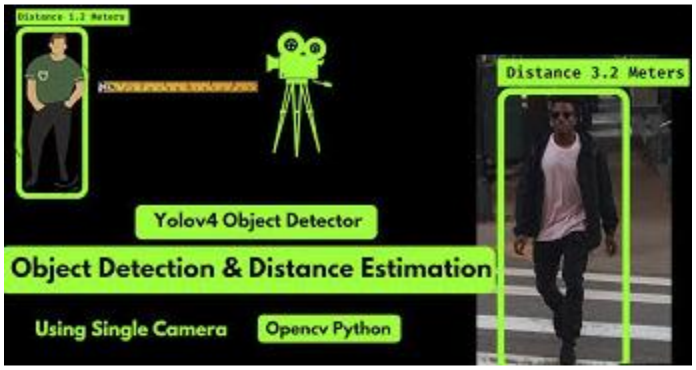


Figure (3.3): YOLOv4 example

3.4 MiDas

The Midas Machine Learning Depth Maps (MLDM) model is a machine learning-based approach for generating depth maps from monocular images. A depth map is a 2D representation of an image that encodes the distance of each pixel from the camera. The Midas MLDM model uses a convolutional neural network (CNN) to predict the depth map of an input image. The model is trained on a dataset of images and corresponding depth maps, allowing it to learn the relationship between the two. The depth maps generated by the Midas MLDM model can be used in a variety of applications, such as robotics, autonomous vehicles, and virtual reality. The model is also able to generate depth maps in real-time, making it suitable for use in real-world scenarios.

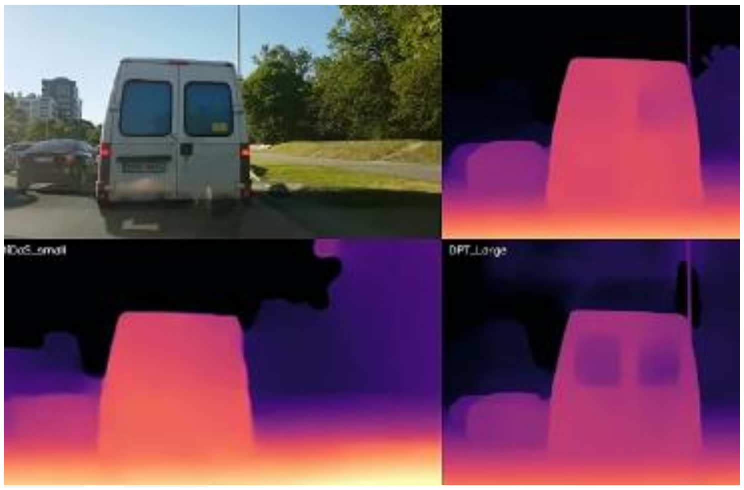


Figure (3.4): Midas example

3.5 Geometry and filters

What can also be done in this context, which also allows for a fast and simple approach, is to manually detect the objects of interest in the image with filters, getting their contours based on the area and the contrast differentiation techniques like differential filters.

Then we can impose rules based on the size of the object with regard to the pixel size in the image and make a rough estimate of the position in terms of the observations, for instance we can take a video of a ball moving away from the camera, and meanwhile also keep track of the distance between the ball and the camera, and with a simple regression curve based on the distance to shape area in image we can determine how far is a distance from the camera the moment a frame is captured.

However, this method is very manual and specific, since the it is heavily relying in the calibration part of the distance measurement with respect to the camera, since each camera has different focal length and dimensions thus producing a different scale and image size for each camera.

The process for the calibration is also hard in the sense that this has to be done for each specific object and for each camera for all distances to get its respective function with respect to the camera.

But this can be useful in some cases where the camera is the same every time with the same conditions, and the object is the same also, then the calibration must only be done once.

4 Design

Since his project has multiple ways of approaching it, different methods are used to try and tackle the problem, the initial idea is to use the Mediapipe model to just locate the joints and determine each joint position, but this isn’t actually the case, since the way that Mediapipe works isn’t as intuitive as it looks.

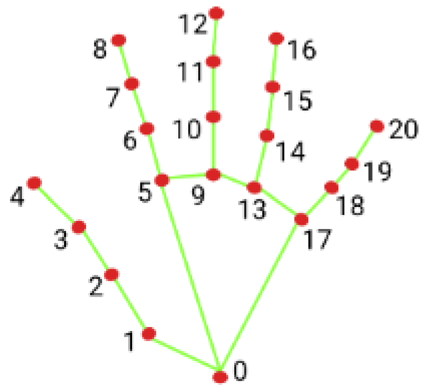
Mediapipe uses two models to produce the output, the first one being palm detection and the other is landmark detection.

4.1 Palm detection:

As the documentation found on the Mediapipe website states that a single-shot detector model was designed to detect initial hand locations in a mobile real-time manner like the face detection model in MediaPipe Face Mesh. The model is optimized for detecting hands in various sizes and in occluded and self-occluded positions, which is a complex task. To address this, the model trains a palm detector instead of a hand detector, as estimating bounding boxes of rigid objects like palms is simpler. An encoder-decoder feature extractor is also used for bigger scene context awareness and the focal loss is minimized during training to support many anchors. With these techniques, the model achieved an average precision of 95.7% in palm detection, which is significantly higher than the baseline of 86.22% without a decoder.

4.2 Landmark detection:

After the palm detection, a subsequent hand landmark model is used to perform precise key point localization of 21 3D hand-knuckle coordinates inside the detected hand regions via regression. This is done by directly predicting the coordinates. The model learns a consistent internal hand pose representation and is robust to partially visible hands and self-occlusions. To train the model, ~30K real-world images were manually annotated with 21 3D coordinates and additional supervision was provided by mapping a synthetic hand model over various backgrounds to the corresponding 3D coordinates. This was done to better cover possible hand poses and provide additional supervision on the nature of hand geometry.



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 (4.1): Mediapipe hand landmarks

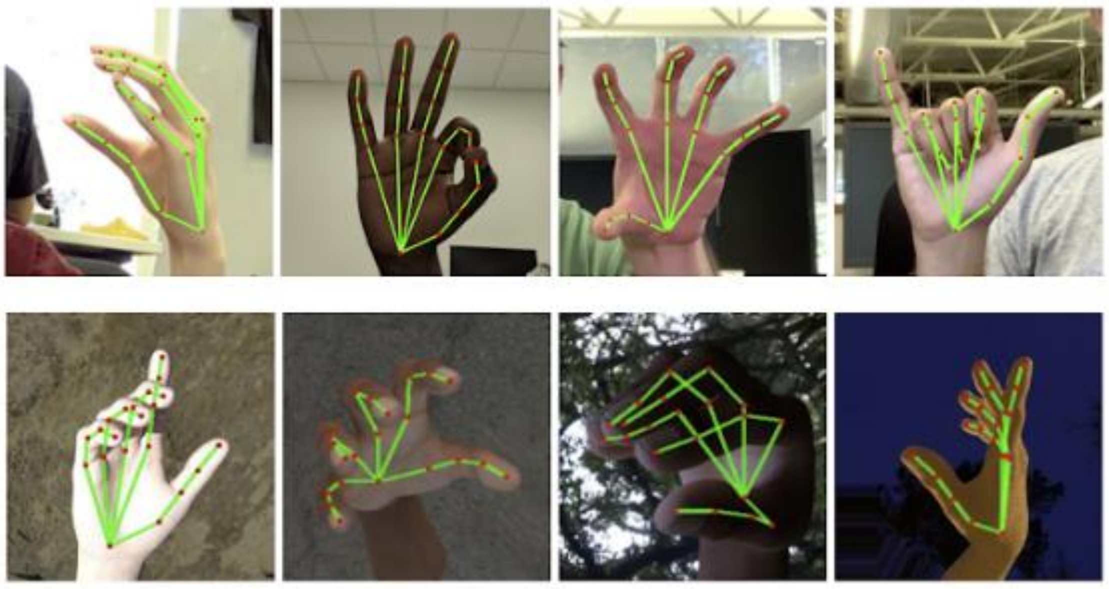


Figure (4.2): Mediapipe hand landmarks detected in different poses.

So the notion that Mediapipe can be instantly implemented to acquire a solution is farfetched, since the depth is relative to the wrist node and related to the geometric center.

Thus, a way must be found to integrate different methods with MediaPipe to produce a robust solution, one thing that I tried, was to integrate the geometrical approach where the hand center distance is measured first depending on the calibration technique measure previously, then combining it with the Mediapipe relative hand landmarks solution to produce a holistic solution.

Another way to approach the problem was to actually exploit the hand positions provided by Mediapipe and transform the 2D representation into a 3D one using Unity's game engine, where it constructs a 3D space that takes as input the coordinates and binds them to local elements that are behaving similarly to the hands of the player.

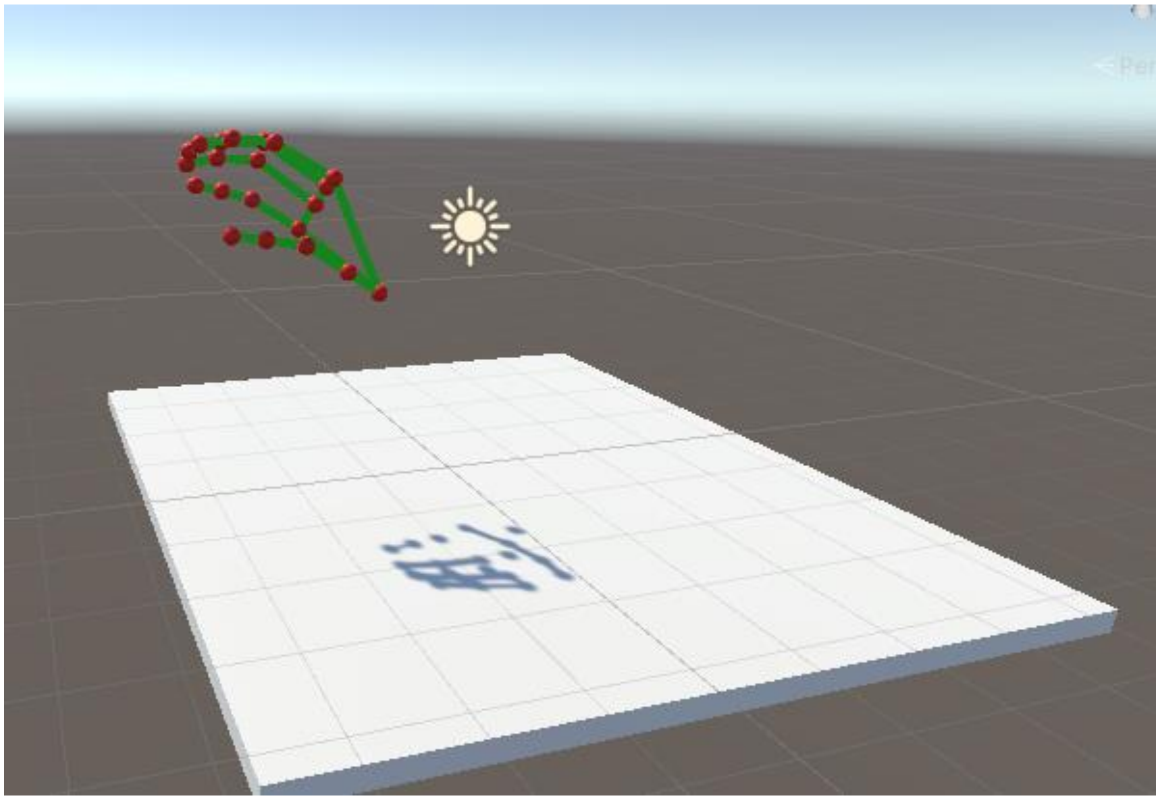


Figure (4.3): An early look at the 3D environment in Unity

5 Implementation

5.1 Technology & pipeline

The program will be designed using python since it has an implementation of Mediapipe as a module, also Opencv will be used since it is the main library to use for computer vision applications in python, and finally NumPy and math libraries will be used to handle the mathematical expressions and the 3D arrays.

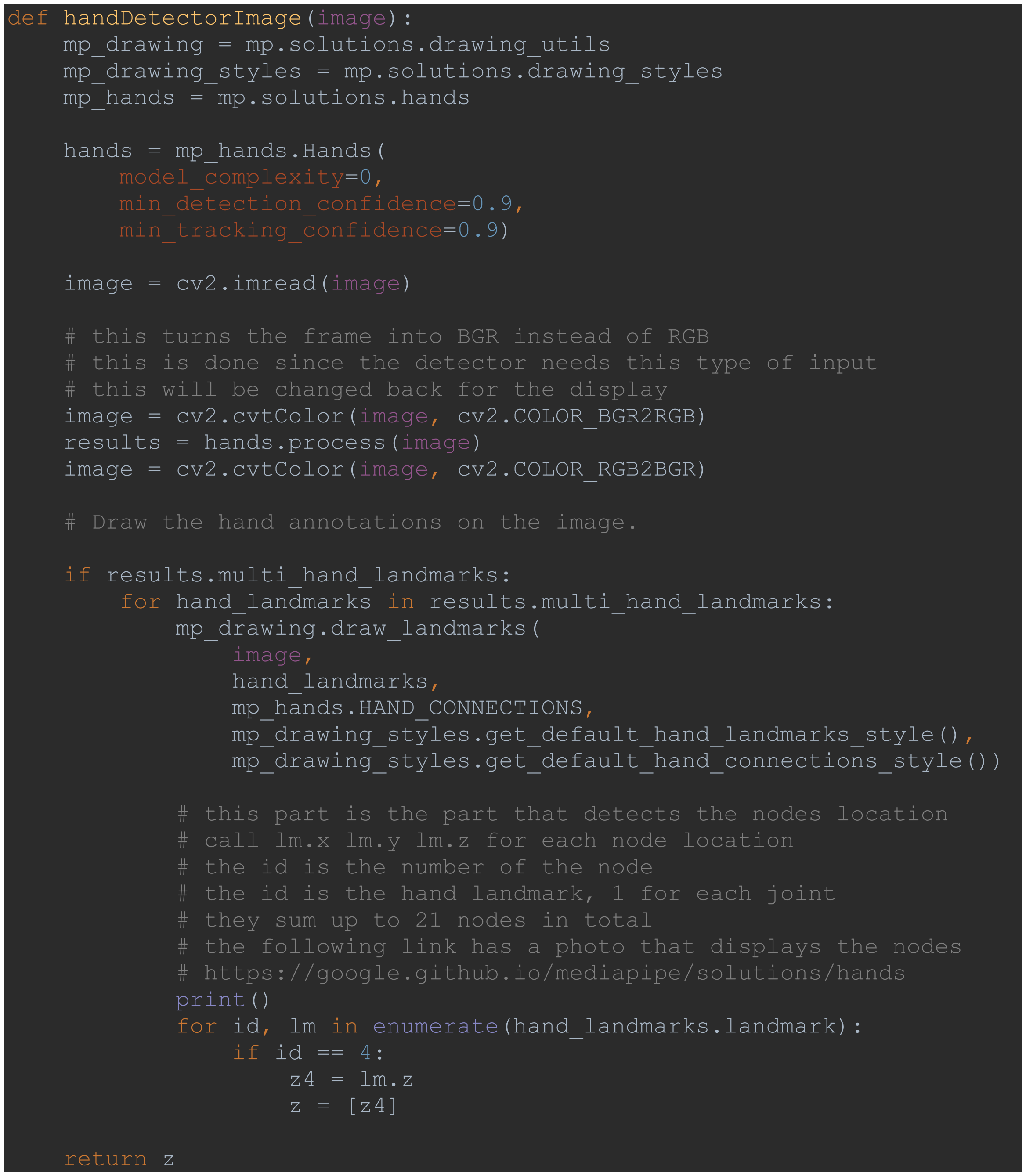
Getting a live camera feed and playing back a video:

This functionality is supported by Opencv and uses the laptops webcam, or any webcam wired to the pc, it can also playback videos.

The Opencv library offers high flexibility to work with individual frames with high performance and with great developer support.

The methods will be arranged in a module that will contain all the methods and classes to invoke during the implementation. The code is fully commented and easily understandable combined with the documentation.

An example will be made here to understand how the model works for one frame for the sake of demonstration, the actual implementation follows a similar pattern but is more complicated in size.



```python
def handDetectorImage(image):
    mp_drawing = mp.solutions.drawing_utils
    mp_drawing_styles = mp.solutions.drawing_styles
    mp_hands = mp.solutions.hands

    hands = mp_hands.Hands(
        model_complexity=0,
        min_detection_confidence=0.9,
        min_tracking_confidence=0.9)

    image = cv2.imread(image)

    # this turns the frame into BGR instead of RGB
    # this is done since the detector needs this type of input
    # this will be changed back for the display
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    results = hands.process(image)
    image = cv2.cvtColor(image, cv2.COLOR_RGB2BGR)

    # Draw the hand annotations on the image.

    if results.multi_hand_landmarks:
        for hand_landmarks in results.multi_hand_landmarks:
            mp_drawing.draw_landmarks(
                image,
                hand_landmarks,
                mp_hands.HAND_CONNECTIONS,
                mp_drawing_styles.get_default_hand_landmarks_style(),
                mp_drawing_styles.get_default_hand_connections_style())

            # this part is the part that detects the nodes location
            # call lm.x lm.y lm.z for each node location
            # the id is the number of the node
            # the id is the hand landmark, 1 for each joint
            # they sum up to 21 nodes in total
            # the following link has a photo that displays the nodes
            # https://google.github.io/mediapipe/solutions/hands
            print()
            for id, lm in enumerate(hand_landmarks.landmark):
                if id == 4:
                    z4 = lm.z
                    z = [z4]

    return z
```

Figure (5.1): A method that takes a single frame as input and return the same frame with the applied Mediapipe hand detection model on the frame and the z-coordinate.

For the part concerning Unity, some observers were written using C#, it is very basic and has the sole purpose of catching the data stream which consists of the string of the coordinates, which are dynamically published from the python socket.

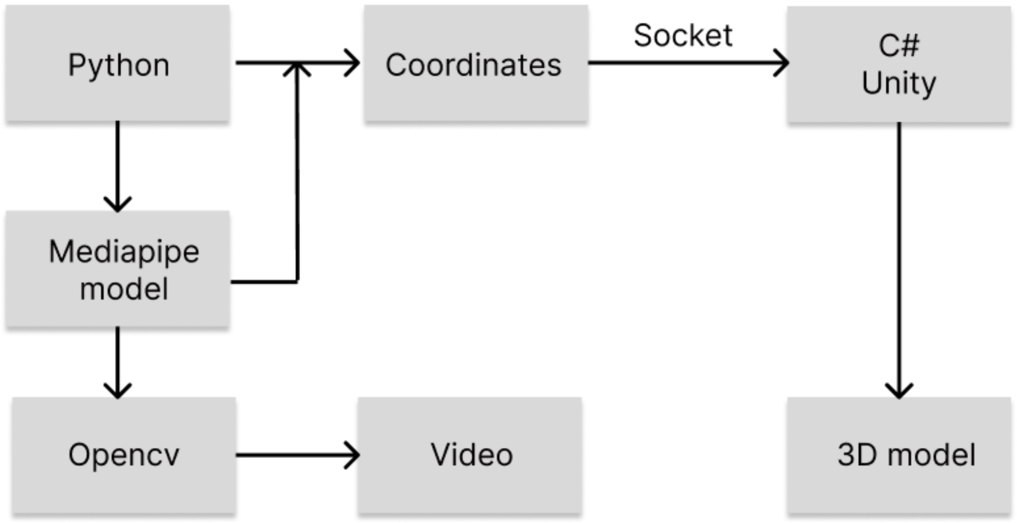


Figure (5.2): Workflow from the input to the 3D model

5.2 Geometrical formulation

For this approach, the rationale is that an object will appear bigger when it is close to the camera, and small when far. One can exploit this property to determine the distance of an object if certain information is gained.

The information in focus are the size of the object and the distance measured beforehand and of different positions, fit into a function that describes the behavior.

For instance, assume we are considering the points 5 and 17 in the Mediapipe hand landmarks, as shown in the figure below:

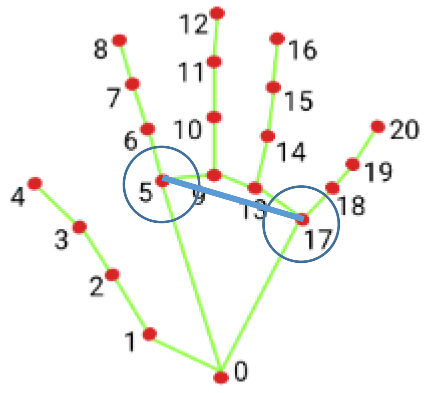


Figure (5.3): Highlighting points 5 and 17 and the distance between them.

Now, if we know the distance between these 2 points in the image, and we have already built the function that describes the relation between the distance and the depth, we can easily know the depth of the hand from the camera.

$$\boldsymbol{Depth = f(distance)} \qquad \boldsymbol{equation\ 5.1}$$

Depth function acquisition:

to acquire the function, it is very hard to do it analytically, so it has to be done empirically, by collecting data at different depths and constructing a curve to fit these lines.

Needless to say, the function can't be linear since the behavior of the projection rays is not linear in the Euclidean space (i.e., the image plane), so it is expected to be of exponentially decreasing behavior.

To acquire the function (which would be different for each camera since their intrinsic parameters are different) we must make an assumption the distance between the points 5 and 17 are equal for all hands, which would produce inaccuracies in the estimation but is somewhat acceptable since this calibration will be done for each camera separately.

First we set the hand at a distance already measured, in our case it was measured by an ordinary tape, and drawing the hand further back and registering the distance, then we can go back to the pictures and find the corresponding distance between points 5 and 17 for each distance, making a couple to approximate a function.

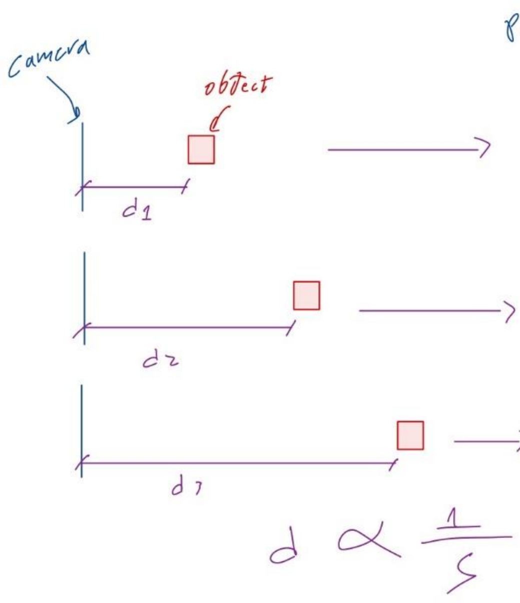


Figure (5.4): A visual representation of the relation between distance and depth where depth is expressed as size in image.

Following the explanation made earlier, the depth is inversely related to the distance of the object from the camera, which can also be interpreted in our case as the size on pixels of the distance between the points 5 and 17 in a hand in a frame, which will enable us to determine the depth of the hand in an image knowing the function (f) and determining the distance as in equation 5.1.

After going through the experience using my laptop's webcam and my hand, the following values were obtained:

```
# where x is the distance in the image
x = [300, 245, 208, 170, 145, 130, 112, 103, 93, 87, 73, 70, 67, 62,
59, 57]
# where y is the depth from the camera in cm
y = [20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95,
100]
```

Which correspond to the function shown in graph 5.5.

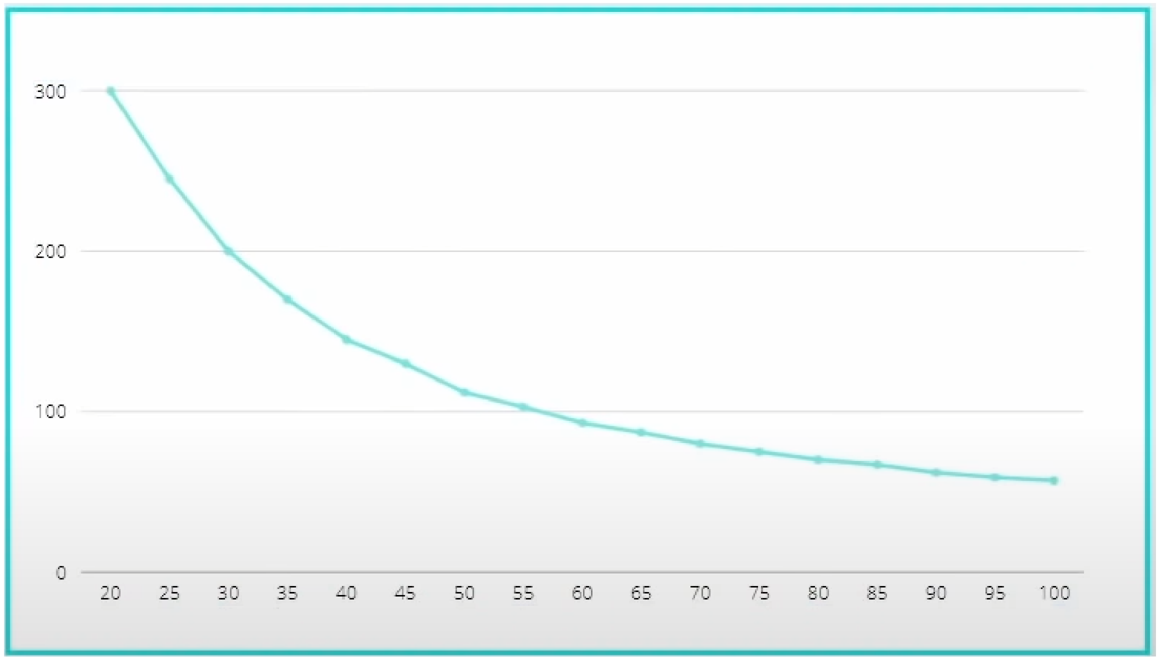


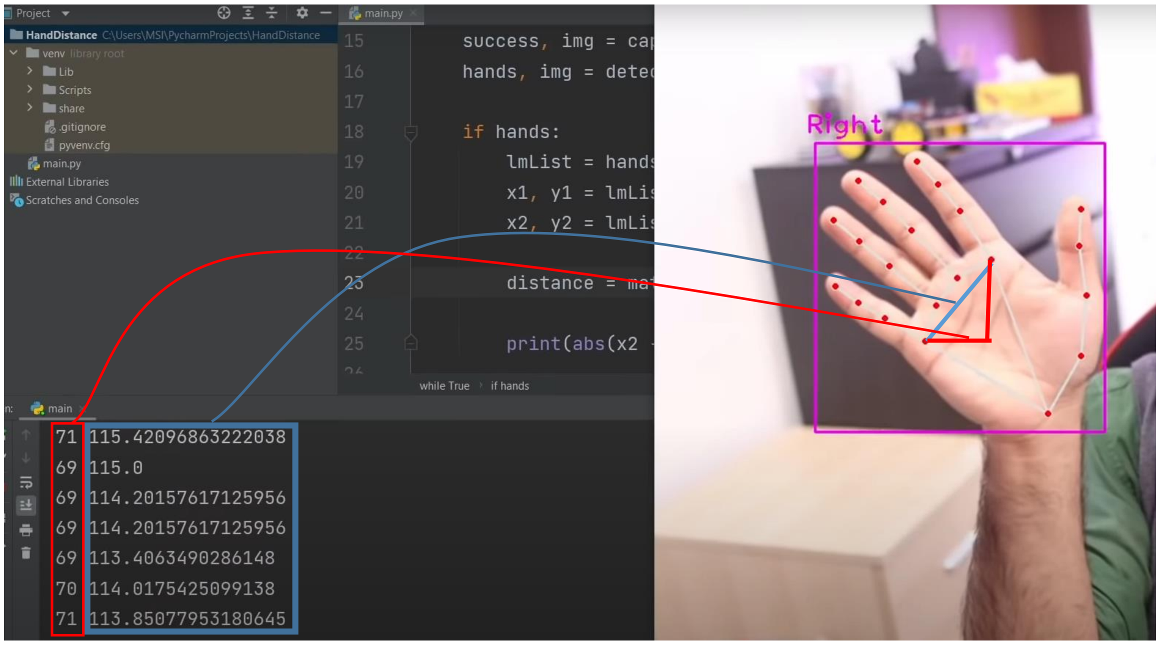
Figure (5.5): The graph of the function obtained from the values reported above.

Next, we can use the values to obtain the function, a regression model would seem a lot of work for a simple 2D function, so we can use numpy.polyfit() function:

```
# The values correspond to a quadratic equation
# y = Ax^2 + Bx + c
# We can use NumPy to get the coefficients

coefficients = np.polyfit(x, y, 2)
```

Now that we have the function, another problem should be solved, how can we determine the distance between the 2 points in the image? one solution would be the difference in their x coordinates, but that introduces another problem, what if the hand is tilted in some angle, so we have to use the vector between the points, since a vector’s length is robust to orientation, in 2D the length of the vector is obtained simply by applying the Pythagorean rule to the x and y coordinates.



```
    success, img = cap
    hands, img = detec

    if hands:
        lmList = hands
        x1, y1 = lmLis
        x2, y2 = lmLis

        distance = ma

        print(abs(x2
```

```
71 115.42096863222038
69 115.0
69 114.20157617125956
69 114.20157617125956
69 113.4063490286148
70 114.0175425099138
71 113.85077953180645
```

Figure (5.6): Diagonal vs Horizontal distance estimation difference.

For the implementation in python:

```
x1, y1, z1 = lmlist[5]
x2, y2, z2 = lmlist[17]
distance = math.sqrt((y2 - y1) ** 2 + (x1 - x2) ** 2)
A, B, C = coefficients
depth = A*distance**2 + B*distance + C
```

where the variable “lmList” is a list that contains the coordinates of all 21 landmarks (points) in a single frame, so we only need points 5 and 17 at the moment.

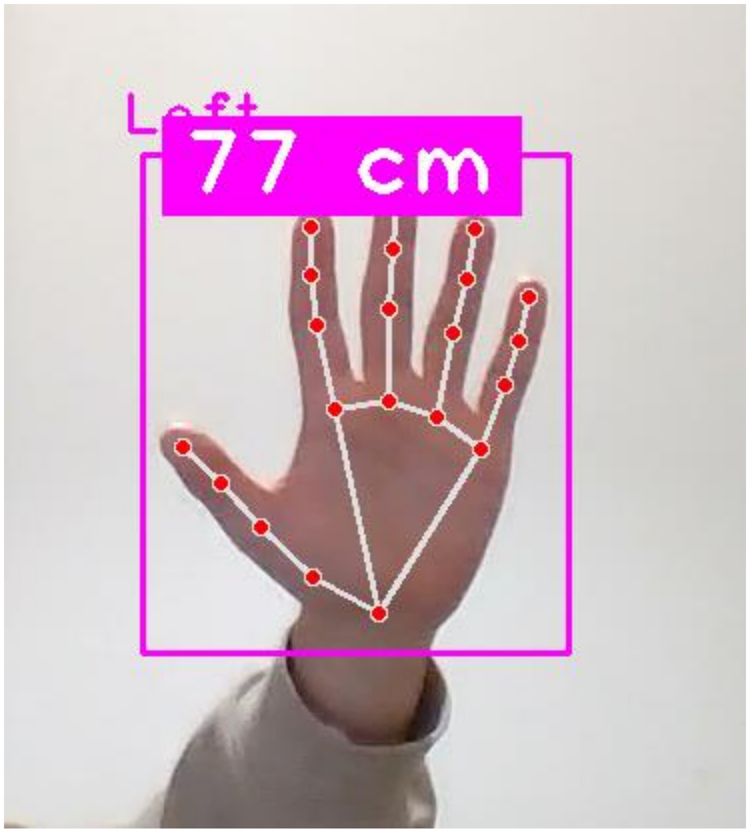


Figure (5.7): The final output of the distance of the hand from the camera.

5.3 Articulation depth estimation

Now that we have the distance of the palm of the hand we can measure the distance of the articulations of each finger following the same rationale, if the distance to the palm of the hand is expressed as d, then the distance to all three articulations of all fingers should also be the same in the case where the hand is spread open, since they are all lying on the same plane.

This justification only works if the palm's plane is parallel to the camera's plane of course, but how do know the distance otherwise? The approach from here on will also be very simple, since the hand in its natural movement can either be spread open or fully contracted, we can also infer the distance from the attitude of the nodes of each finger.

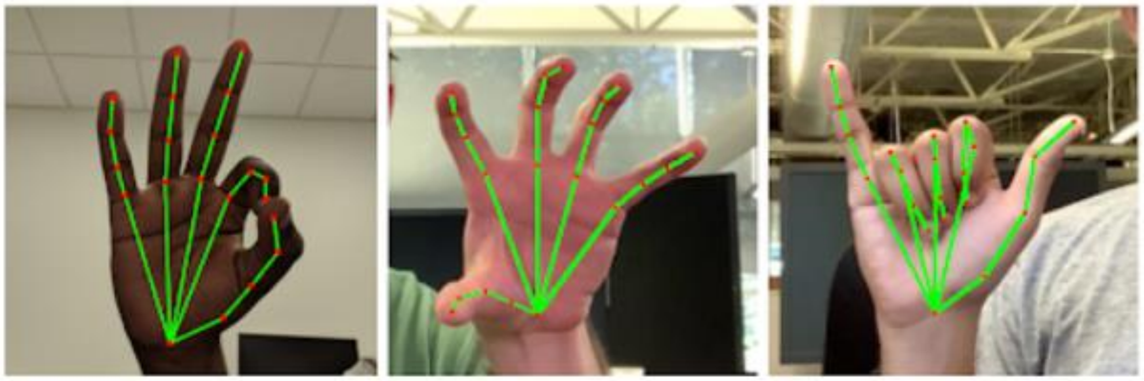


Figure (5.8): Hand attitude cases ordered from left to right as 1, 2, 3 respectively.

As shown in figure's 5.8 attitude 1, we can tell the position of the pinky, ring, and middle finger instantly since they're on the same plane as the palm, for the index and the thump on the other hand, some considerations must be taken.

In the index case, we can make relations combining the distance between each consecutive node, for instance the distances between the nodes in the middle finger are the maximum possible, then they're at the same plane of the palm, meaning at offset zero.

But for the index, the distances between its nodes are nearing the minimum distance possible, meaning they are further than the palm with an offset.

The way we can determine the offset is also experimental and depends on real world values, but the values in the image don't matter this time, since what we care about is the attitude of the articulations rather than their distance, but the distance in the image translated as the line vectors joining each joint are in fact representative of the depth offset from the palm.

This method will be used to determine the position of the three articulations of each finger in the special case where we know the depth of the palm in the image, and that the plane of the palm is parallel to the plane of the image.

$$\boldsymbol{a1, a2, a3 = f(l1, l2, l3, d)} \qquad \boldsymbol{equation\ 5.2}$$

Where a1, a2, and a3 are the three articulations of a finger.

L1, l2, and l3 are the lines joining these joints with the palm node.

And d is the depth of the palm.

We will create a study case of 1 finger and generalize the findings into the remaining ones, with some differences of course which will be discussed during the explanation.

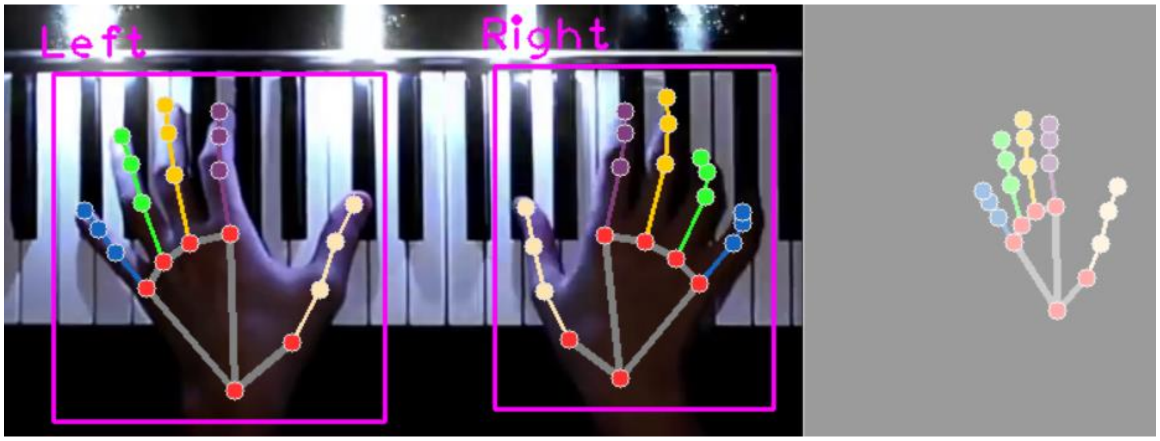


Figure (5.9): Finger articulations estimation.

As can be seen in the above figure, the left hand currently is pressing on 2 keys with the thump and the pinky, when all the other fingers are in rest, and the index is preparing to press a black key.

The middle key and the ring key can be seen to be on the plan of the hand so the depth of their articulations are the same for the palm.

The pinky as discussed earlier is pressing a key, so the lines of in between the nodes seem to have less vectorial values than usual.

The same method used to determine the depth of the hand with respect to the camera will be used, but it will be in terms of line between nodes with respect to the distance between nodes 5 and 17.

The idea is that we don’t have to make another empirical estimation again since we have fixed the distance of the baseline between the points 5 and 17, so we have the “scaling” factor of the depth from the camera, and now we only care about the relative distances from the nodes of the stem of each finger.

However, the distance is unique for each finger, since the length of each finger is unique, also for each finger different ratio values will be tested to determine the distance in the script.

To determine the relations between the joints and the lines connecting them to determine the position of the joints of each finger with respect to the palm, we must make a few assumptions, these assumptions do not reduce the accuracy to an unsatisfactory level, and will make developing the detection algorithm much faster.

To start off, we will refer to figure 5.10 and will adopt the following naming convention where the line connecting two joints p1, p2 will be named L_{12} and so on.

For example, we consider the index finger, which contains three joints corresponding to the 3 articulations of the finger, and a base knuckle joint, in the index finger case referring to figure 5.10 the joints are 8, 7, 6, 5, where the joints are the three articulations and the knuckle joint respectively.

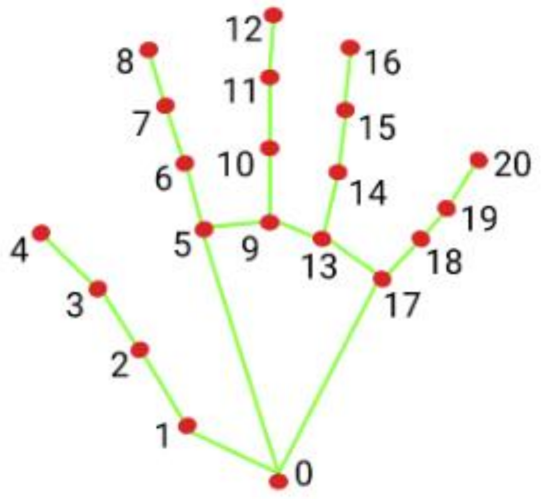


Figure (5.10): Mediapipe hand landmarks

The first operation that is adopted, is that the knuckle joint corresponding to point will have the same depth as the palm since it is the point used in the process.

The second assumption would be that all lines joining the joints of a finger have the same length, for the index finger this means that:

$$L_{56} = L_{67} = L_{78}$$

This would lead to a very simplified solution that would produce satisfactory results, so in an ideal scenario where all lines have the same length, the hand is completely flat or the finger is completely pointing down as shown in figure 5.11.

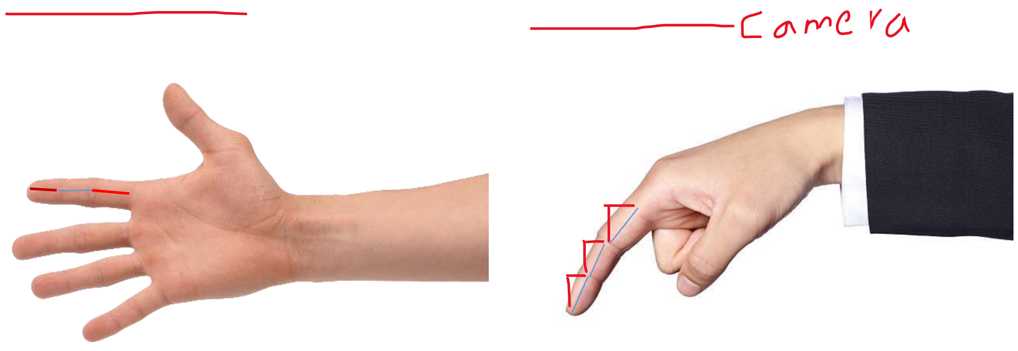


Figure (5.11): Positions where the lines of the articulations would be equal.

The second case is where the lines are nor equal, this would lead to different scenarios where it is necessary to define the finger behavior in its movement to determine how the cases would look like and how to interpret them.

The fingers of the human hand can move sideways in a limited amount, but this wouldn't affect the depth of the joints of the finger, only the x and y components which are tracked with the Mediapipe detector, so we don't consider this case.

The fingers can also move up and down, in the down case, which we are interested in, the finger behaves in a similar way all the time, especially with musicians who adopt a very specific technique when they play, the movement consists of moving the 2 outermost lines together to go the first step, then finally the third line moves down to go to the maximum extent.

In the example of the index finger, this would mean that in order for the finger to descend from the plan of the palm, the first movement would cause the lines L_{87} and L_{76} to decrease in length simultaneously, the final move would be that the third line would also go down with them, bringing them all the case discussed above where all lines are of the same length (but much less than the planer mode).

So we will use the relationships discussed above to determine the position of each joint in the finger at all times, one thing that should be discussed is that we assumed the lines connecting consecutive joints in the same finger to be equal, but this is not the case in cross-fingers lines, since the fingers all have different lengths compared to each other.

This can be solved by measuring the hand and finding the lengths of each finger separately or finding a study that averages the hand size for a test sample, for our applications we'll adopt the average obtained as the average thumb is 35 to 65 millimeters long. The index is 50 to 88 millimeters long and the ring finger is 42 to 96 millimeters long. The middle finger is 57 to 100 millimeters long and the pinky is 40 to 74 millimeters long.

So basically, we'll adopt a convention where the following values are taken into consideration, thump length 50 mm, index 67 mm, middle finger 88 mm, ring finger 68 mm, and finally the pinky 57 mm.

For the implementation in python, the length of each line will be measure according to the method described for the points 5, 17 in the depth estimation technique, and will be done for each line which will make the sum to be 21 lines, then the lines will be compared to the lines that belong to the same finger and the values will be stored.

After the values are stored, an algorithm will iterate through the values in order to determine the ratios described above, and will determine the position the finger is taking, and will calculate the depth of each articulation of each finger, and finally all the results will be stored as a series.

After doing some experimentation with the algorithm, it has been found that our initial hypothesis regarding the lengths of the line segments joining the joint representing the finger articulations are not necessarily equal, and would produce some differences depending on the hand ratios which are different from person to person.

To navigate through this problem, some empirical values will be derived from experiment from the value of the ratios used in the estimation process.

The process discussed above will be implemented using the acquired ratios as the upper and lower bound, and with the help of linear interpolation, one can find the intermediate location of the articulations throughout al positions.

Since the ratio is robust to both depth and orientation, establishing a relative depth index from the palm plane exploiting the ratio of the line segments allows for easy and computationally inexpensive solution.

The same process is followed for all of the fingers, and the estimated positions of the articulations are saved into a text file for the video of the piano shot from the top, also the hand detection module also provides a script that will showcase the live demonstration.

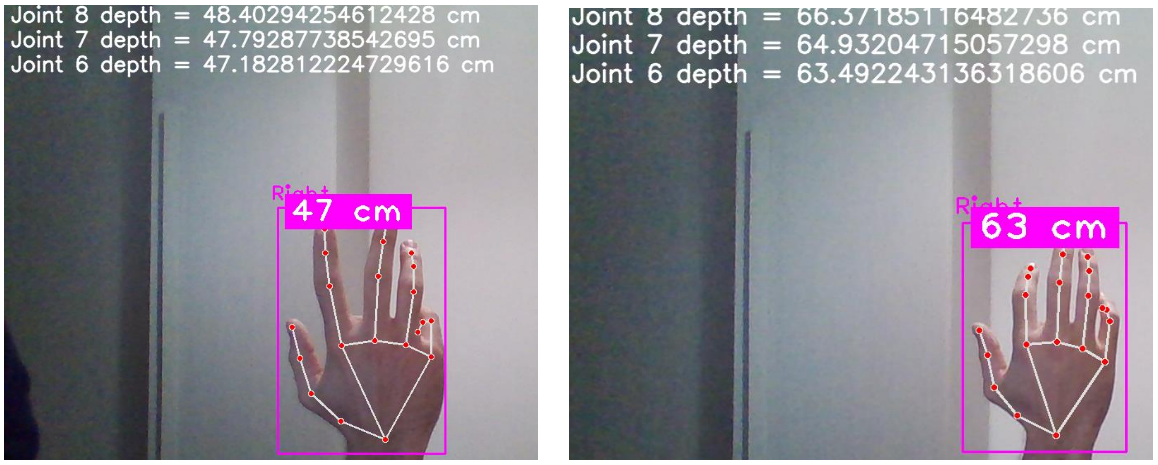
As an example mentioned here, an experiment will be made on the index finger to test out the correctness of the hypothesis, the ratios of the lines joining the 3 outer joints representing the three articulations of the finger are obtained empirically.

The ratios were 0.9 for the case where the hand was completely flat and in the palm plane, the second ratio was when the finger was in the pressing pose, which is equivalent to the piano key pressing motion which the value was 0.45, interpolating this function to count for all cases in between, the following function was obtained:

$$Depth\ tip = index\ length * (0.9 - ratio) \qquad \boldsymbol{equation\ 5.3}$$

$$Depth\ second\ joint = index\ length * \frac{(0.9 - ratio)}{2} \qquad \boldsymbol{equation\ 5.4}$$

The estimation based on the previous equations have produced satisfying results, and the result have been assigned with its own python file, named "HandDetectionGeometryIndex.py" which can be found in the file directory, running this script will produce an output that tells the user the depth of the palm, and the 3 articulations depth of the index finger, represented in the naming convention used in the Mediapipe hand landmarks.



```
Joint 8 depth = 48.40294254612428 cm
Joint 7 depth = 47.79287738542695 cm
Joint 6 depth = 47.182812224729616 cm
```

```
Joint 8 depth = 66.37185116482736 cm
Joint 7 depth = 64.93204715057298 cm
Joint 6 depth = 63.492243136318606 cm
```

Figure (5.12): Difference in depth of the index articulations based on the pose of the finger (left: hand spread) (right: index pressing).

This method has been generalized and used on all the fingers in a video of a musician playing a piano, and all the positions of the articulations of each finger for each hand have been saved in a text file to be used conveniently for analysis in the next step of the process.

At this point, the goal of the project has been achieved, and the final output is consistent and coherent with the initial requirements.

6 Extras

As an extra step, which can be useful to visualize the output in 3D rather than making graphs and looking the logged values, a Unity project is developed in 3D using the coordinates for each finger and its articulations to output a 3D model for the hand in a 3D environment.

This can be done thanks to python's flexibility, where we can deploy the values live and place them in a socket that streams the data as strings, meanwhile, Unity will subscribe to the stream and "listen" to the values broadcasted by the socket.

But Unity uses C#, so some scripts have been written in C# to catch the stream of strings and decode it to the floating numbers of the positions of each node, then the values are fed to the Unity engine, where a game like environment is set up, and the elements corresponding to the nodes and the lines are assigned, then the values are defined to each node which are defined as spheres in Unity, and the lines are linking these spheres.

This way, we can run the python project, and as it is running, the Unity engine catches the stream, and we can run it in parallel to project the 3D hand position.

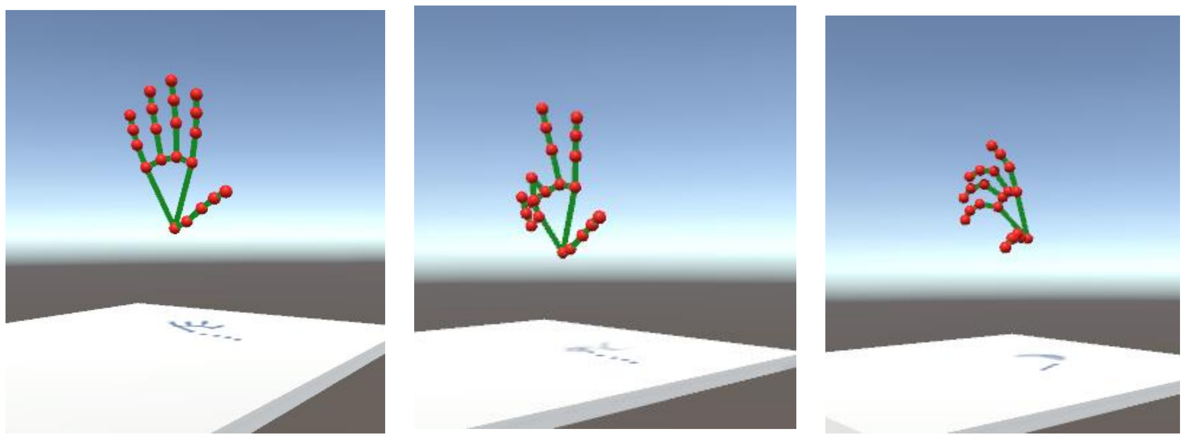


Figure (6.1): Example of hand positions in 3D environment.

7 Conclusion & recommendations:

In conclusion, this computer vision project successfully utilized deep learning and geometry to determine the position of a hand and individual fingers as well as their articulations.

The use of deep learning algorithms allowed for high accuracy in detecting and identifying the hand and fingers, while the incorporation of geometry provided a more robust understanding of the hand's position and movement.

The results of this project demonstrate the potential for these techniques to be applied in various practical applications such as robotics, virtual and augmented reality and Human-computer interaction.

Further research could focus on improving the robustness of the system in different lighting conditions and increasing the number of hand gestures that can be recognized. Overall, this project represents a step forward in the field of computer vision and its potential to enhance human-computer interaction.

Also further recommendations would be to take into account different scenarios where for instance the plane of the hand is not parallel to the plane of the camera or perpendicular or include all the cases in between.

And the problem where the musicians are playing instruments like the guitar and an accordion are still unsolved since the hands follow more complicated movements.

8 References:

1 – Mediapipe website : Home | mediapipe (google.github.io)

2 – Mediapipe hand documentation: Hands | mediapipe (google.github.io)

3 – Hand landmarks: hand_landmarks.png (1543×538) (mediapipe.dev)

4 – Midas: MiDaS | PyTorch

5- Yolov5: https://github.com/ultralytics/yolov5

9 Appendix

9.1 List of figures