Logo

Description automatically generated with medium confidence

**Image Analysis and Computer Vision**

**Project Documentation**

**Mohanad Diab**

**10769366**

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.

Graphical user interface, website

Description automatically generated

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.

Diagram

Description automatically generated

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.

Diagram

Description automatically generated

Figure (2.2): How the perspective projection views reality

3 Preliminary research

**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.

A picture containing person, person, toy, lady

Description automatically generated

Figure (3.1): HAAR example

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.

A picture containing text

Description automatically generatedA picture containing indoor, keyboard

Description automatically generated

Figure (3.1): Mediapipe example

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.

Graphical user interface, text, application, chat or text message

Description automatically generatedHowever, 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 (3.1): YOLOv4 example

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 are 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.

**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.

**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.

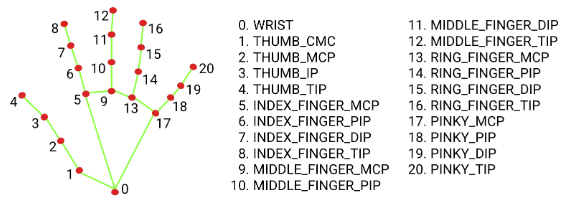


Figure (4.1): Mediapipe hand landmarks

A picture containing different, colorful, several

Description automatically generated

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.

Graphical user interface

Description automatically generated

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

**5 Implementation**

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.

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.

Diagram

Description automatically generated

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

**Geometrical approach**

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 are 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.

Chart

Description automatically generatedFor 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.

**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.

Diagram

Description automatically generatedFirst 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]

Chart, line chart

Description automatically generatedWhich 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.

**Graphical user interface, website

Description automatically generated**

Figure (5.4): 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

A picture containing text, indoor

Description automatically generatedwhere 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 (5.5): The final output of the distance of the hand from the camera.