Group: Digital Theremin

Members:

Name	SFU email
Daksh Patel	dpa45@sfu.ca
Dave Williams	dmw13@sfu.ca
Bobby Poulin	rrp5@sfu.ca

Guide Written by: **Dave Williams**

I give permission to Dr. Brian Fraser to make use of, and reproduce this guide.

Mediapipe Hand Landmark
How To Guide

The following is a step by step guide for how to use Google's Mediapipe Framework for real time hand tracking on the BeagleY-AI. This will cover the steps necessary to write a Python script for hand landmark tracking, suggestions for hand pose detection, and for integrating with a C module on the board. You will need a webcam for this guide.

Python Script

1.) Set up a Python environment on your board. I did this with Miniforge, which can be easily installed via the terminal. You can set up your environment using other tools as well.
2.) Create a Python environment, version=3.9 is fine, and **pip install** the following libraries: **numpy**, **opencv-python**, and **mediapipe.** Once you have those installed, that should be the entirety of the initial setup process.
3.) Create your Python script. The general procedure for implementing the Hand Tracking Module is the following:
    a.) Initialize Mediapipe
    b.) Initialize your camera with cv2
    c.) Pass each frame to Mediapipe's Hand Landmark Detection Model.
    d.) Retrieve and visualize the result.
    e.) Extract custom Gestures from the detection results (Optional)

a.) Import mediapipe. After that is done, we can set the attributes of a hand tracking model quite simply. The following code snippet loads Mediapipe's hand landmark tracking model and specifies some relevant attributes.

```
# Load Model
mp_hands = mp.solutions.hands
hands = mp_hands.Hands(
    static_image_mode=use_static_image_mode,
    max_num_hands=1,
    min_detection_confidence=min_detection_confidence,
    min_tracking_confidence=min_tracking_confidence,
)
```

static_image_mode specifies that we want to process individual frames sequentially. **max_num_hands** allows us to specify the number of hands the model can track simultaneously. The last two attributes specify detection confidence, meaning the model threshold for identifying a hand, and then tracking confidence, which is the minimum threshold for tracking hand keypoints. As the aforementioned suggests, the model is actually two models. One for hand detection, and the other landmark tracking on detected segments.

b.) To initialize your webcam you first need to import cv2 into your Python Module. Next you will need to create a **cv2.VideoCapture** instance and assign its properties like the following

```
# Camera preparation
cap = cv2.VideoCapture(cap_device)
cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter_fourcc('M', 'J', 'P', 'G'))
cap.set(cv2.CAP_PROP_FRAME_WIDTH, cap_width)
cap.set(cv2.CAP_PROP_FRAME_HEIGHT, cap_height)
cap.set(cv2.CAP_PROP_FPS,10) #limit fps
```

cap_device is a simple integer index value that references your webcam. You can find this value via the Terminal by calling: **ls /dev/video*** Your device will probably be at index 0, but it might not. On my board it was index 3. Below is a sample of results from my terminal call:

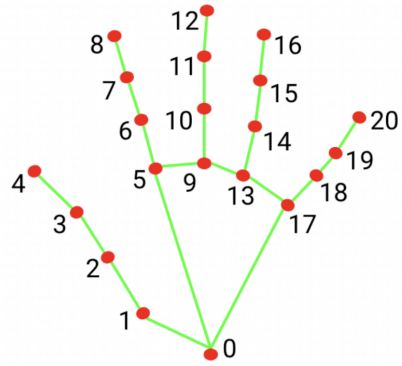
```
(test-env) davew@dmw13-beagle:~$ ls /dev/video*
/dev/video0  /dev/video1  /dev/video2  /dev/video3  /dev/video4
(test-env) davew@dmw13-beagle:~$
```

The rest of the properties defined, specify the encoding method, width, height and framerate.

c.) Next, create a loop that will call **cap.read()** to read frames from the camera feed. Once we have a frame, we will convert it to an RGB encoding from BGR, which I believe is default with opencv. After that we'll pass the frame to the hand model, and retrieve a result set:

```
# Detection implementation
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
frame.flags.writeable = False
results = hands.process(frame)
frame.flags.writeable = True
```

d.) The landmark set for a hand is a list of 21 (x,y,z) hand key-points that the model returns a prediction for. These points represent a predicted wrist position, base of pinky position, middle joint, upper joint, tip, etc.. For a full breakdown of the keypoint set, refer to Google's developer information: https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker. Here is a relevant snippet:



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

To process the result we'll do the following. First split the result set for each hand (only 1 if max_num_hands=1):

```
if results.multi_hand_landmarks is not None:
  for hand_landmarks, handedness in zip(results.multi_hand_landmarks,
                                        results.multi_handedness):
```

hand_landmarks is the landmark set for a single hand. Here we can process the set for visualization, or perhaps implement custom hand gesture detection.

d.) For visualization, we need to take the keypoint, extract the (x,y) values, then rescale to the dimensions of our camera instance. By default the values are in a [0,1] scale so this will need to be corrected. Afterward, we'll need to iterate through the keypoint set, and draw circles on our original frame at the key points rescaled (x,y) positions using opencv's drawing tools, and then draw lines between relevant key points according to the skeleton included above. Finally, in our loop we can finish by calling **cv2.imshow()** on the modified frame.

e.) For gesture detection, there are a huge number of different possible methods you could employ. Our project only needed to calculate distances between the thumb and various fingers for a set of potential hand-states. Your implementation may make use of another model, or you might extract gestures manually as well. For manual gestures, one thing I would recommend is rescaling keypoint distances relative to the distance between the wrist and base of the ring finger (0-9 on the skeleton), as distances will of course be affected by the distance of your hand to the camera. The palm height is a relatively fixed proportionality that you can scale other distances relatively with.

C Integration

For C integration, a simple solution is a Python UDP module to transmit the hand keypoint set for each frame, and a receiving UDP module in your C application. To make UDP parsing simple on the C side, it will be convenient to flatten the keypoint set and convert each value into a space separated string. These values will be in [0,1] scale, so depending on your needs you might need to rescale before sending. Our project redraws the skeleton on the Beagle's LCD module, which is **240x240px**

For how to do this, you should create a separate module and thread for custom LCD draw calls. This module should make use of the LCD and LGPIO libraries provided by the professor in class, and the relevant initialization code. From your UDP module, pass the parsed data (this can be done with **strtok()** and **atoi()**) to your new module, which you can most easily store in a buffer. Your thread should access this buffer. To draw, we'll have to make use of the functions included in GUI_Paint.c, which are included as part of the LCD library. We can call Paint_DrawCircle() to draw the keypoints, and Paint_DrawLine() to draw the skeleton. We should not have to rescale anything if the data was already pre-processed. Here are some snippets from the draw procedure:

```
static void draw_hand_screen(int points[], int size){
  assert(is_initialized);
  assert(size == NUM_HAND_KEYPOINTS);
  const int joint_radius = 3;

  Paint_NewImage(s_fb, LCD_1IN54_WIDTH, LCD_1IN54_HEIGHT, 0, WHITE, 16);
  Paint_Clear(WHITE);

  //draw joint points on scree
  for(int i = 0; i < size - 1; i+=2){
    int x = points[i];
    int y = points[i+1];
    if(x > 0 && x < LCD_1IN54_WIDTH && y > 0 && y < LCD_1IN54_HEIGHT){
      Paint_DrawCircle(x, y, joint_radius, BLACK, DOT_PIXEL_1X1, DRAW_FILL_FULL);
    }
  }

  //draw joint connections REFER TO JOINT MAP
  //wrist to thumb
  Paint_DrawLine(points[0], points[1], points[2], points[3], BLACK, DOT_PIXEL_1X1, LINE_STYLE_DOTTED); //0-1
  Paint_DrawLine(points[2], points[3], points[4], points[5], BLACK, DOT_PIXEL_1X1, LINE_STYLE_DOTTED); //1-2
  Paint_DrawLine(points[4], points[5], points[6], points[7], BLACK, DOT_PIXEL_1X1, LINE_STYLE_DOTTED); //2-3
  Paint_DrawLine(points[6], points[7], points[8], points[9], BLACK, DOT_PIXEL_1X1, LINE_STYLE_DOTTED); //3-4
```

One thing to keep in mind is there will likely be significant latency. There are a few methods for approaching this issue. One issue is limitations on the LCD's refresh rate. To fix this, I would suggest the Professors SPI Guide. The other major vector of latency issues will be with the heavy image processing of the Hand Landmark model. To improve this somewhat, we can fix a lower framerate, or decrease the cv2 camera resolution. I would recommend a very low camera resolution as it substantially improves performance and does not significantly reduce overall model performance.