Barak Samesh Tom Hilton

CV Based Cursor

Introduction

In this project, we try to implement a program that manipulates the computer's cursor according to visual gestures of the user's palm of the hand, captured by the machine's camera.
The process of achieving this goal can be generally described by the following steps, which we will elaborate on later:

1) Hand recognition and feature extraction
2) Calculating desired cursor instructions
3) Executing cursor manipulation

The program is written in **Python** and depends on several additional packages: **numpy** and **math** for vectorial and trigonometric calculations, **openCV** for computational vision functionalities and **pynput.mouse** for manipulating the cursor.

1) Hand Recognition and Feature Extraction

A. Creating Mask Image

First, we determine a **region of interest (ROI)** (Figure 1.1), a subarea of the whole frame that within this area we will seek the user’s hand. To recognize the user’s hand we need to differentiate between the hand and the background. Thus, assuming the hand is darker than the background and the background is uniformly colored, we determine two thresholds of colors of HSV base, such that pixels with HSV values between the thresholds will count as being a part of the hand.

Subsequently, we create a binary picture, a **mask** (Figure 1.2)**,** in which if a pixel is part of the hand, it is colored white, else, it is colored black. Then, we apply Gaussian blur in order to smooth out unwanted noise artifacts.

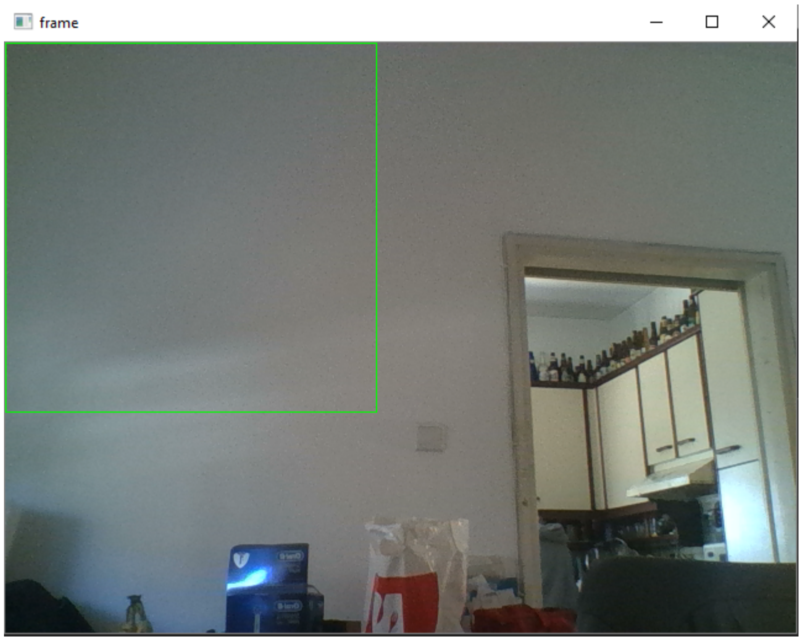

Figure 1.1 ROI (Region Of Interest) marked in green

B. Creating Curve and Convex Hull

We then calculate the curve of the mask and apply polygonal approximation by some epsilon. Now that we recognize the hand, we need to distinguish between specific gestures. To achieve this we find the convex hull polygon of the approximated curve.

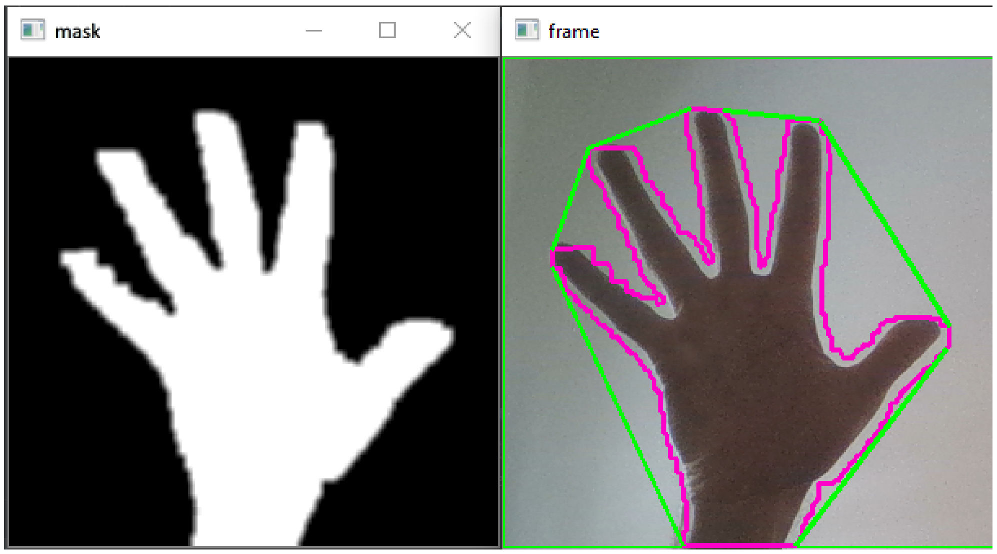

Figure 1.2 The mask of the hand (on the left), the approximated curve (marked pink) and its convex hull (marked green)

C. Calculating Convexity Defects

The convex hull can give us a good feature to consider when analyzing the shape of the hand: **Convexity Defect-** A cavity in the contour which creates an area which is included in the convex hull but excluded from the area of the contour. Each defect is characterized by several parameters- start (point), end (point), far (point) and distance (scalar). These can be seen in Figure 1.3, taken from OpenCV documentation.

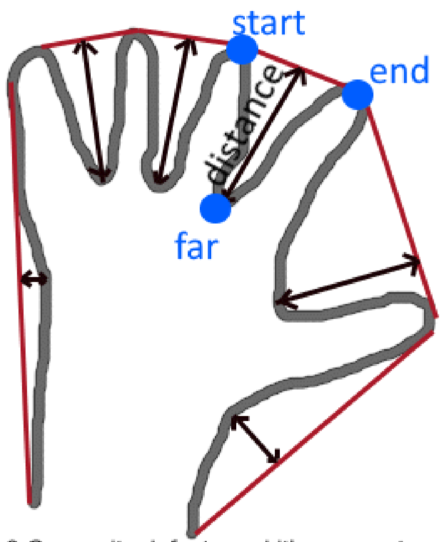

Figure 1.3 Convexity defects and it's parameters created by a typical contour of a humanoid hand, taken from the OpenCV documentation.

From all the defects found, we extract the relevant defects by these criterias:

- The distance should be greater than a determined threshold (defects with small distance are probably a result of noise)

- The angle created by the points start-far-end should be greater than a determined threshold (defects with small angle are probably a result of noise)
- The defect should be oriented upwords, that is, for a given defect, the middle point of start and end should be higher than the far point of the defect (we would want to only consider defects between fingers)

We can use the relevant defects to distinguish between gestures based on the number of relevant defects and also determine a reference point for the hand based on the average point of the relevant defects far points.

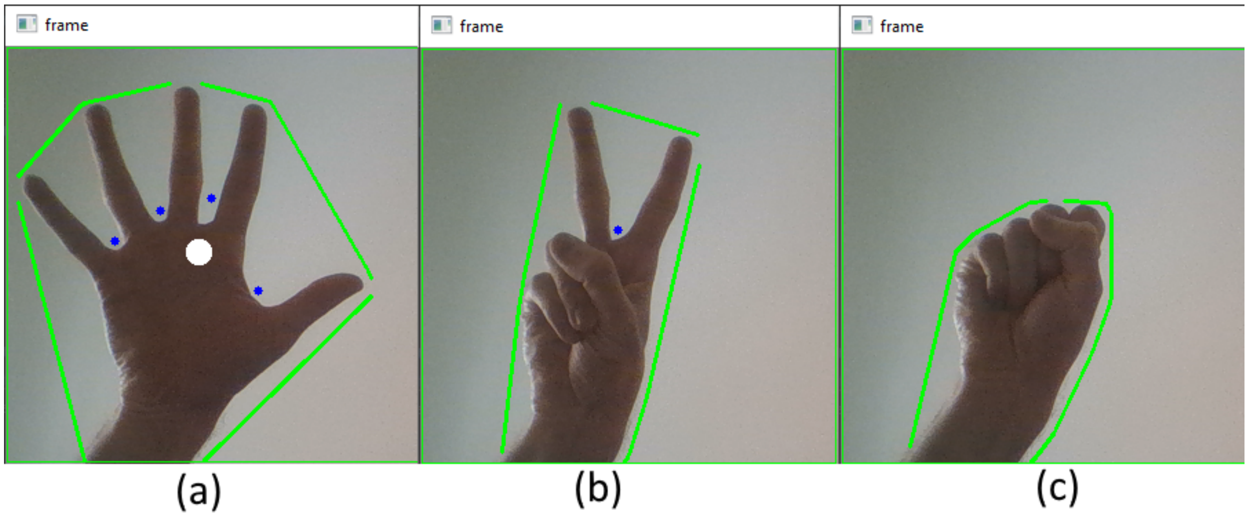

Figure 1.4 Different gestures distinguished by number of relevant defects. The **far points** of the defects are marked with blue dots. On (c) the reference point of the hand is marked by a white circle.

Now we can use the reference point to determine the location of the hand within the ROI and the number of relevant defects to distinguish between gestures.
First and zero order moments were considered for calculating the center of mass of the mask image, using it as the reference point, but the idea was dropped after realizing the forearm is also considered as the hand on the mask, giving the center of mass undesired behaviour as a reference point for the hand (Figure 1.5).

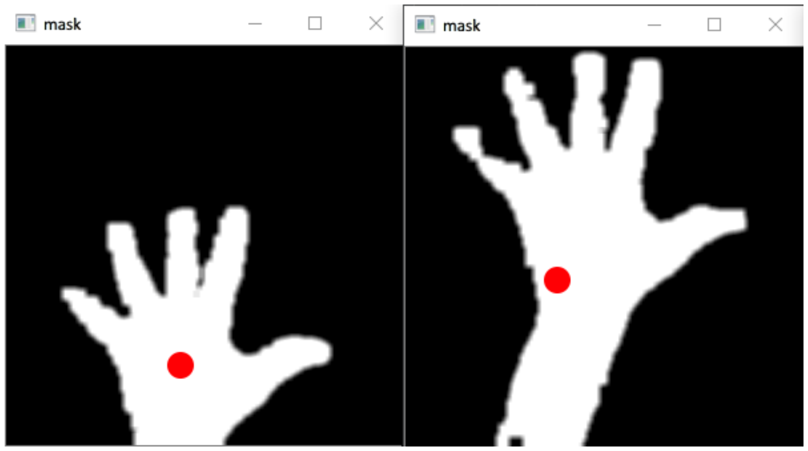

Figure 1.5 Two mask images with the center of mass of each marked by red dots. When the forearm is included in the mask, the center of mass is lower, related to the palm of the hand, in reference to the location of the center of mass when the forearm is excluded from the mask.

<u>2) Calculating Desired Cursor Instructions</u>

A. Determining the Cursor State

To know how to manipulate the cursor, we need to define the way that the hand should affect the cursor, in terms of velocity and state.

Determining the state is the low hanging fruit- We can conveniently determine the gestures that would indicate the state of the cursor according to figure 1.4:

- Open hand (a) for mobilizing the cursor (nothing clicked)
- Two fingers raised (b) for right button clicked
- closed fist (c) for left button clicked

Now we define how the velocity of the cursor would be affected by the location of the reference point of the hand within the ROI. First, we will address the speed (magnitude of the velocity) and then we define the behaviour of the direction.

Constants would be written in bold and italic, ***like so***.

B. Determining the Cursor Speed

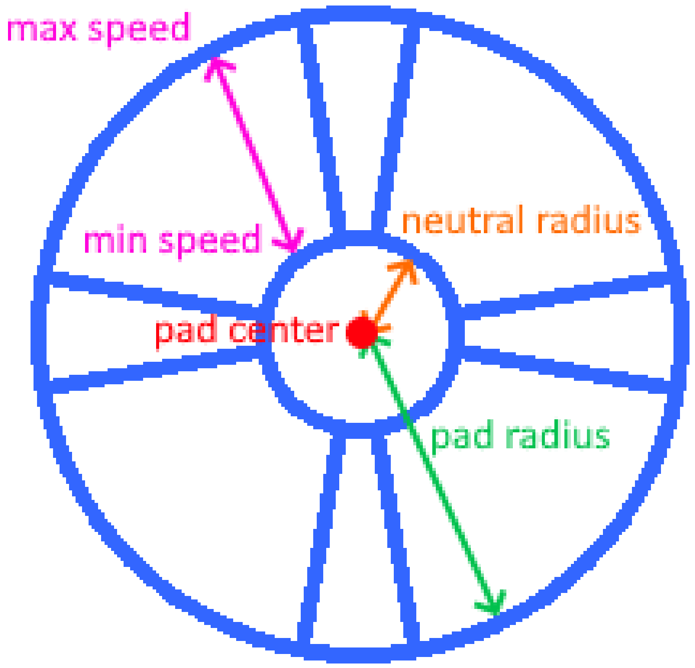

Figure 2.1 The constants mantiond on section 2B visualized geometrically

The region of the mouse pad is a circle with the radius of ***pad radius*** centered at ***pad center (cx, cy)***. Within the mouse pad there is a neutral area which is a circle with the radius of ***neutral radius*** centered at ***pad center***. When the reference point is inside the neutral area or outside the mouse pad, the speed of the cursor is 0.
Otherwise, the value of the speed moves between ***min speed*** and ***max speed***. When the reference point is at distance ***neutral radius*** from ***pad center***, the speed would be ***min speed*** and when the distance is ***pad radius***, the speed would be ***max speed***. Between those two distances, the speed linearly interpolates between ***min speed*** and ***max speed***.

C. Determining the Cursor Direction

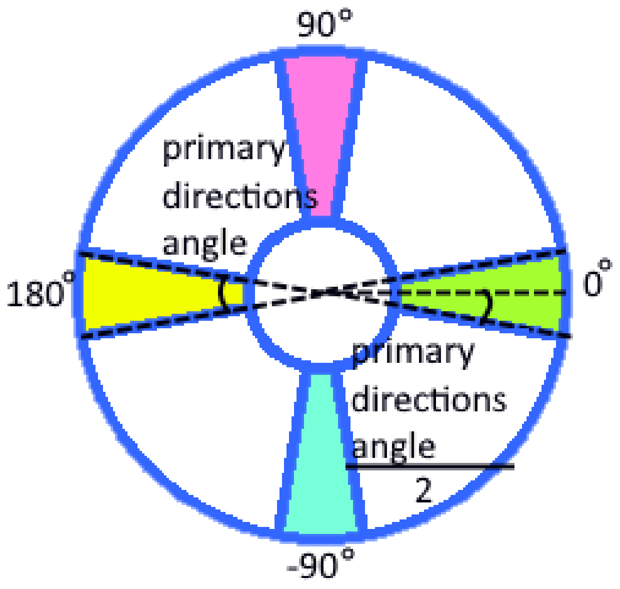

Figure 2.2 Visual representation of the slices of the primary directions

We would refer up (90°), down (-90°), left (180°) and right (0°) as the primary directions. “Around” each primary direction we allocated a slice of the mouse pad with the angle of ***primary directions angle*** (the area of each slice lies between the radius of the primary direction + ***primary directions angle/2*** and the radius of primary direction - ***primary directions angle/2***). When the reference point of the hand is in a slice of a primary direction, the direction of the cursor's velocity will be the corresponding primary direction. Otherwise, The direction would be the direction of the vector (reference point - ***pad center***)***.*** For Example, when the reference point is in the pink area on Figure 2.2, the direction of the velocity would be up (90°).

D. Calculating Vertical Speed and Horizontal Speed

We have the speed and the direction of the velocity. Thus,

$Velocity = speed \cdot dir$

To manipulate the cursor we need to seperate the x and y ingredients from the Velocity vector. This could be done by simple geometry, given **α** as the angle of the velocity:

$xspeed = speed \cdot cos\alpha$

$yspeed = speed \cdot sin\alpha$

3) Executing cursor manipulation

A. Defining the "mouse control" class parameters

First, we have created a class that assembles all of the functions that control the cursor.
The class contains two parameters which are an instance of the object that controls the cursor which is named "**mousec**" and another parameter "**cool_down**".
"cool_down" is essential when we perform a "right click" or "left click" because it prevents Multiple executions of the left or right click in a row, Because the program received more than one frame in a row when the user performs the gesture.
This is a solution for a slow user that hesitates after performing a gesture.

We used a time gap between two clicks that will count as separate gestures, the reason is that it does not prevent the real user to perform the same click multiple times if this is his real intention.

B. Defining the “mouse control” class functions

This class four functions:

1. Constructor - simply initialize the parameters.
2. Move_mouse - given 2 integers which indicate how many pixels from the current location the cursor should move in X and Y.
    a. X- positive : right
    b. X- negative: left
    c. Y- positive: down
    d. Y- negative : up
3. Gesture- given a string that indicates if it should perform a right click or left click it performs the corresponding click.
4. Control_mouse - this function wraps the other function (“move_mouse” and “gesture”) and by receiving a string that decides which function is needed and 2 integers for the case we want to move the cursor. The function decides which function to call and calls it with its input.

C. connect the “mouse control” class to the other code

Initialization of the class is used and then we use variables that we calculated in the previous section as input to the instance of this class’s method “control_mouse”.

Conclusions

Computer vision is a vast domain and for most problems in this domain exists more than one way to approach it. Each problem can be solved using different tools and one should pick the best tool for his task.
Each tool uses and emphasizes different features of the given data you have to work with, and just like in choosing the right representation, choosing the right approach may ease the way to the goal and choosing the wrong one could lead you to a rough ride.
We implemented a solution step by step, completing one task at a time, and finally we got a program that achieves the goal we set for ourselves.

Demonstration Video

https://www.youtube.com/watch?v=zGID2HO6JJo&feature=youtu.be