



**POLITECNICO  
DI TORINO**

---

Corso di Laurea Magistrale in Ingegneria Informatica

PROGETTO DI IMAGE PROCESSING AND COMPUTER VISION

# Soluzione per l'orientamento stradale e distanziamento sociale

Progetto di:

**Andrea Bona: 277925**

**Rosario Milazzo: 280088**

**Silvia Raggi: 277628**

Prof.re:

**Bartolomeo Montrucchio**

**Luigi De Russis**

# Indice

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduzione</b>                                   | <b>2</b>  |
| 1.1      | Obiettivo . . . . .                                   | 2         |
| 1.2      | Strumenti utilizzati . . . . .                        | 2         |
| <b>2</b> | <b>Descrizione del codice</b>                         | <b>3</b>  |
| 2.1      | Traffic Light Detector . . . . .                      | 3         |
| 2.1.1    | Compute_Roi . . . . .                                 | 3         |
| 2.1.2    | Display_Roi . . . . .                                 | 5         |
| 2.1.3    | Detect_Color . . . . .                                | 5         |
| 2.1.4    | Limitazioni del Traffic Light Detector . . . . .      | 6         |
| 2.2      | Social Distancing . . . . .                           | 6         |
| 2.2.1    | Face_Recognition . . . . .                            | 6         |
| 2.2.2    | Play_Sound . . . . .                                  | 7         |
| 2.2.3    | Limitazioni del Social Distancing . . . . .           | 7         |
| 2.3      | Rilevamento dei marciapiedi . . . . .                 | 8         |
| 2.3.1    | Edge_detect . . . . .                                 | 8         |
| 2.3.2    | Roi . . . . .   | 8         |
| 2.3.3    | Run . . . . .   | 8         |
| 2.3.4    | Limitazioni del rilevamento dei marciapiedi . . . . . | 9         |
| <b>3</b> | <b>Main e Test</b>                                    | <b>10</b> |
| <b>4</b> | <b>Conclusioni</b>                                    | <b>11</b> |

# Capitolo 1

## Introduzione

Stiamo vivendo un periodo particolare in cui l'attenzione non è mai troppa. La Pandemia ci sta mettendo a dura prova, il virus è un nemico comune che dobbiamo assolutamente abbattere ma l'unico strumento a nostro favore momentaneamente è il rispetto delle regole disposte dal Ministero della Salute per evitare il contagio. Per alcune persone è più complicato poter osservare le normative per la prevenzione contro il Covid-19, per cui il nostro pensiero va alle persone non vedenti che non hanno sempre la possibilità di riuscire a mantenere il social-distancing, ormai divenuto fondamentale per la convivenza in una comunità che ha bisogno, ora più che mai, della collaborazione tra i singoli individui, anche se, purtroppo, non si può sempre far affidamento sul buon senso dei cittadini. La Computer Vision in questo caso, può diventare un ottimo alleato per coloro che hanno bisogno di un'aiuto concreto.

### 1.1 Obiettivo

Per i motivi appena illustrati abbiamo pensato di progettare, attraverso l'uso della Computer Vision, un dispositivo che abbia due fini, diversi, ma entrambi utili per la mobilità dei non vedenti:

- **Rilevamento dei marciapiedi e dei semafori** (riconoscimento dei colori).
- **Rilevamento del distanziamento sociale**, in modo da permettere al soggetto che sta usando l'applicazione di mantenere un metro di distanza dalle altre persone.

Le due funzioni saranno accompagnate da un sistema vocale in grado di segnalare le rilevazioni. Si vuole proporre una valida opzione per aiutare le persone non vedenti ad affrontare con maggiore serenità questo periodo, abbattendo per loro, dove si può, le limitazioni motorie, rendendo quindi gli spostamenti sicuri e a norma di legge, negli spazi aperti o nei luoghi chiusi come edifici pubblici e luoghi di ristoro.

### 1.2 Strumenti utilizzati

Il lavoro presentato in questo elaborato è stato creato attraverso l'ausilio del linguaggio di programmazione Python (versione 3.6), utilizzando l'IDE PyCharm CE e con l'aiuto delle seguenti librerie:

- *OpenCV* per l'uso delle degli algoritmi di Computer Vision
- *Numpy* per l'uso di funzioni matematiche
- *Winsound* per l'introduzione di suoni all'interno dell'applicazione
- *Thread* per la programmazione multi-threading
- *Scipy* per le funzioni per il calcolo della distanza



Figura 1.1: Logo di alcuni strumenti utilizzati.

# Capitolo 2

## Descrizione del codice

Le funzioni principali sono divise in tre file `.py` la cui descrizione viene portata qui di seguito.

### 2.1 Traffic Light Detector

Ottenere la detection dei Semafori stradali senza l'ausilio del Machine Learning richiede un buon gioco di squadra tra tecniche di Image-Processing e le operazioni offerte da OpenCV. Usare come base per la detection solo il riconoscimento di forme e colori potrebbe non essere la soluzione migliore, poiché il nostro codice potrebbe “confondersi” con altri elementi geometrici della scena, identificandoli erroneamente come semafori. Al fine di eseguire il riconoscimento nel modo più efficiente possibile ci sono alcuni elementi chiave da tenere in considerazione: le variazioni d’illuminazione, la sogliatura e lo spazio dei colori. L’obiettivo è quello di evitare la probabilità di intercettare “falsi semafori”.

Le operazioni sono implementate all’interno della classe `ROI_TrafficLights` che contiene tre metodi che verranno illustrati qui di seguito.

#### 2.1.1 Compute\_Roi

Una volta preso in ingresso il video, questo viene processato per ogni frame, il quale viene convertito in scala di grigi e vengono eseguite le seguenti operazioni:

1. Creazione di una prima ROI che elimina la parte inferiore del video.
2. Applicazione della trasformazione morfolologica `cv2.morphologyEx`: questa permette di evidenziare le aree più luminose della scena.
3. Con il metodo `cv2.Threshold` viene effettuata la segmentazione. Possiamo vedere dall’immagine risultante che ci sono dozzine di “luci” che riescono a passare, ma solo poche di loro sono effettivamente semafori, molte light spot provengono da aree come automobili o edifici.

```
def compute_roi(self, video): #delimita la zona dove cercare possibili semafori
    frame_transformed_to_gray = cv2.cvtColor(video, cv2.COLOR_RGB2GRAY) #conversione in scala di grigi
    mask = np.zeros(video.shape[:2], dtype=np.uint8)
    mask = cv2.rectangle(mask, (175, 0), (475, 500), (255), thickness=-1) #delimito la ROI dove posso esserci semafori
    tophat = cv2.morphologyEx(frame_transformed_to_gray, cv2.MORPH_TOPHAT, self.kernel) #segue la hat morphology per trovare le spot light
    ret, thresh = cv2.threshold(tophat, self.threshold, 255, cv2.THRESH_BINARY) #segmentazione
```

Figura 2.1: Funzione `compute_roi`



Figura 2.2: A sinistra il Frame dopo aver applicato `cv2.morphologyEx`, a destra lo stesso Frame dopo l’applicazione della segmentazione

4. Per ridurre la probabilità di intercettare i falsi semafori viene applicata la Watershed: è un algoritmo particolarmente utile quando si vogliono estrarre oggetti che si sovrappongono nelle immagini. Il primo passo da fare è usare la “distanza euclidea” (*cv2.distanceTransform*), la quale calcola la distanza dallo zero più vicino (cioè pixel di sfondo) per ciascuno dei pixel in primo piano. La funzione *cv2.ConnectedComponent* trova le parti dell’immagine che sono collegate fisicamente tra loro.

```
dist_transform = cv2.distanceTransform(thresh, cv2.DIST_L2, 5)
ret, markers = cv2.connectedComponents(np.uint8(dist_transform))
watershed = cv2.watershed(video, markers)#watershed, o "spartiacque"
watershed = cv2.normalize(watershed, None, 255, 0, cv2.NORM_MINMAX, cv2.CV_8UC1)
markers += 1
```

Figura 2.3: Procedimenti per operare la watershed

L’output di questa funzione restituisce i “marker”, ovvero le etichette che permettono di differenziare i pixel della scena: quelli che hanno lo stesso valore di etichetta appartengono allo stesso oggetto. Dopo la trasformazione possiamo notare che alcuni punti, i quali nella segmentazione potevano sembrare parti di un semaforo, vengono “inglobati” dalla sagoma dell’oggetto di cui fanno effettivamente parte. Se sono stati scelti gli opportuni parametri (kernel e threshold) i semafori dovrebbero essere identificati come cerchi distinti.

5. Nell’immagine ottenuta dopo l’applicazione della Watershed potrei ancora visualizzare molti punti che non sono semafori. Per questo motivo si ricorre alla detection dei Blob (Binary Large Object e si riferisce a gruppi di pixel collegati in un’immagine): l’analisi BLOB può anche aiutare a filtrare i semafori confrontando le proprietà del BLOB con quelle che ci aspettiamo siano di un semaforo. I semafori devono essere circolari e non occuperanno mai gran parte dell’immagine. Selezioniamo attraverso l’impostazione dei parametri i BLOB che sono un po ’circolari e non troppo grandi. Il prodotto finale è una serie di candidati a semaforo che soddisfano i nostri criteri di forma e dimensione.

```
self.params.minThreshold = 50
self.params.maxThreshold = 1000

self.params.filterByArea = True
self.params.minArea = 50
self.params.maxArea = 1000
```

Figura 2.4: Alcuni parametri per intercettare i Blob

6. I Blob identificati diventano i “keypoints”, ovvero i punti attorno ai quali vengono costruite le vere ROI di riferimento, all’interno delle quali viene fatta la detection dei semafori.

```
ver = (cv2.__version__).split('.')
if int(ver[0]) < 3:
    detector = cv2.SimpleBlobDetector(self.params)
else:
    detector = cv2.SimpleBlobDetector_create(self.params)
keypoints = detector.detect(watershed, mask) #la detect sulla watershed mi permette di eliminare il più possibile "falsi semafori"
blobs = cv2.drawKeypoints(watershed, keypoints, np.array([]), (0, 0, 255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS) #disegno i Blob trovati
```

Figura 2.5: Operazioni per trovare i Keypoints

Quello che dovremmo ottenere dopo questa serie di applicazioni sono le coordinate dei possibili semafori che dobbiamo intercettare all’interno del nostro video.

```
for keypoint in keypoints:#per ogni keypoint, ovvero per ogni Blob (possibile semaforo trovato) ne prendo le coordinate del centro
    x = keypoint.pt[0]
    y = keypoint.pt[1]
    coordinate_x = int(np.median(x))
    coordinate_y = int(np.median(y))
    yield coordinate_x, coordinate_y
```

Figura 2.6: Operazione per intercettare le coordinate dei keypoints



Figura 2.7: Frame dopo aver applicato il watershed e rilevamento dei Blob

### 2.1.2 Display\_Roi

Questo metodo, a cui viene passato in ingresso il video e la dimensione desiderata per le ROI, costruisce le finestre attorno alle coordinate dei Blob che sono state ottenute con il metodo `compute_roi`.

```
def display_roi(self, video, window_size):#costruisco finestre attorno ai possibili semafori, definizione delle ROI

    mask = np.zeros(video.shape, dtype=np.uint8)
    x_offset = int((window_size[0] - 1) / 2)
    y_offset = int((window_size[1] - 1) / 2)
    display_img = np.zeros(video.shape, dtype=np.uint8)

    for (x, y) in self.compute_roi(video):#richiamo la compute_roi da dove ottengo le coordinate dei blob e costruisco così le ROI
        x_min, x_max = x - x_offset, x + x_offset
        y_min, y_max = y - y_offset, y + y_offset
        mask[y_min:y_max, x_min:x_max] = 1
        display_img = np.zeros(video.shape, dtype=np.uint8)
        display_img[mask == 1] = video[mask == 1]

    return display_img
```

Figura 2.8: Definizione di Display\_roi

### 2.1.3 Detect\_Color

In genere le immagini sono rappresentate nello spazio colore RGB. Tuttavia, RGB mescola le informazioni del colore e sull'intensità di luce in tutti i suoi canali, ciò rende il formato RGB sensibile ai cambiamenti nell'illuminazione. Se il nostro obiettivo è rilevare i semafori, non possiamo avere variazioni nell'illuminazione (es. soleggiato, piovoso, nuvoloso, ecc.). Per contrastare questo problema, molti scelgono di convertire le immagini in spazi colore che separano la *chrominance* dalla *luminanza*. Alcuni esempi, che sono ben rappresentati in letteratura, sono HSV, HSL, CIELab e YCbCr. Convertiamo il nostro video in HSV. L'obiettivo qui è quello di intercettare i semafori attraverso l'informazione di colore: creiamo le maschere per identificare i tre colori che ci interessano (rosso, giallo e verde). Come prima, utilizziamo la *SimpleBlobDetector* per cercare i Blob che potrebbero rappresentare i semafori, ma questa volta ripetiamo l'operazione per ogni maschera di colore trovata.

```
if maskr is not None: #MASCHERA ROSSA
    ver = (cv2.__version__).split('.')
    if int(ver[0]) < 3:
        detector = cv2.SimpleBlobDetector(self.params)
    else:
        detector = cv2.SimpleBlobDetector_create(self.params)
    keypoints_r = detector.detect(redMask) #RILEVA BLOB ROSSI

    for keypoint in keypoints_r: #PER OGNI BLOB ROSSO SEGNALO UN CERCHIO E LA SCRITTA
        x = keypoint.pt[0]
        y = keypoint.pt[1]
        coordinate_x = int(np.median(x))
        coordinate_y = int(np.median(y))

        cv2.putText(img, "RED", (coordinate_x, coordinate_y), font, 1, (255, 0, 0), 2, cv2.LINE_AA)
        cv2.drawKeypoints(img, keypoints_r, np.array([]), (255, 0, 255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        redMask = cv2.drawKeypoints(redMask, keypoints_r, np.array([]), (255, 0, 255), cv2.DRAW_MATCHES_FLAGS_DRAW_RICH_KEYPOINTS)
        cv2.circle(redMask, (coordinate_x, coordinate_y), 10, (0, 255, 0), 2)
        cv2.imshow("RED MASK", redMask)
        ROI_TrafficLights.number_of_frame_red=ROI_TrafficLights.number_of_frame_red+1
```



Figura 2.9: Esempio: come viene identificata la maschera rossa per il rilevamento del semaforo

La priorità sta nel recuperare i semafori con successo, anche a discapito della precisione. L'idea è che possiamo sempre filtrare i falsi positivi in seguito, ma non possiamo recuperare una luce persa.

### 2.1.4 Limitazioni del Traffic Light Detector

Il codice, così com'è presentato può soffrire di alcune limitazioni. La rilevazione dei Blob può comportare il rilevamento erroneo di falsi semafori in qualche frame (anche se grazie alle funzioni già illustrate il problema è diminuto con ottimi risultati) per questo difetto si è cercato di applicare ulteriori accorgimenti selezionando opportunamente sia i parametri per operare la segmentazione dell'immagine, sia i parametri per identificare i blob. Indubbiamente con l'aiuto del Machine Learning si sarebbe potuto evitare questo problema, in quanto si potrebbe operare una classificazione per il riconoscimento dei semafori. Un altro problema sorge con la posizione dei semafori: il codice riesce a rilevare solo cerchi ben definiti, per questo motivo riconosce a fatica i colori dei semafori se posti lateralmente, o comunque in una posizione che non sia frontale a chi fa uso del dispositivo. Un ultimo accorgimento riguarda il range di colore che viene riconosciuto dalle maschere: poiché può capitare che il range del rosso e del giallo si confondono (dipende dal video su cui viene effettuato il test), per ovviare al problema si è deciso di emettere una voce che esclama "Fermo" quando vengono rilevati sia il giallo che il rosso, la voce incita a proseguire nel momento in cui viene rilevato il verde.

## 2.2 Social Distancing

Il compito di effettuare il social distancing è svolto da due funzioni principali, contenute nella classe `social_distancing`, i cui incarichi sono quelli di effettuare il riconoscimento dei volti mediante *Haar Cascade Classifier*, e l'altro di emettere un output sonoro per segnalare la vicinanza eccessiva di una persona al dispositivo che la rileva.

### 2.2.1 Face\_Recognition

Per il riconoscimento dei volti si è scelto, come già detto, di utilizzare *Haar Cascade Classifier*.

```
face_model = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
```

Figura 2.10: Haar Cascade Classifier

La funzione `face_model.detectMultiScale` va a cercare delle zone di luci e ombre compatibili con volti all'interno della *Region of Interest* definita. Queste vengono salvate in un vettore `face_cor`; si procede poi con la stampa dei quadrati che delimitano il volto e del punto blu che ne indica il centro. Infine, per ognuno di questi volti ad ogni frame viene calcolata la distanza stimata dalla telecamera e viene stampata a schermo. La distanza, avendo a disposizione una sola telecamera, è calcolata tramite una formula che mette in correlazione lunghezza focale, altezza stimata del volto, altezza dell'immagine in pixel, altezza del sensore e altezza totale dell'immagine:

$$Distance = \frac{focallength \times imageheight \times referencefaceheight}{faceheight \times sensorheight} \quad (2.1)$$

dove *focal length* è in *mm*, *image height* in *pixel*, *reference face height* in *mm*, *face height in pixel* e *sensor height* anch'esso in *mm*. La stima della distanza è quindi fortemente limitata e legata alle qualità della telecamera con cui si sta effettuando la ripresa.

Per stampare in output la distanza stimata, si effettua infine una divisione per 10, per riportare i valori in centimetri, misura ritenuta più utile per la valutazione di un giusto distanziamento.

Per limitare il più possibile il rilevamento di falsi positivi (zone di luce e ombra che vengono erroneamente rilevate come volti) si è adottato uno stratagemma che consiste nel memorizzare, ad ogni frame, un vettore con i volti che sono stati rilevati (`facecheck`). Al frame successivo, prima di visualizzare e calcolare le distanze (solo se inferiori ai 3 metri), la funzione si occupa di verificare se esista una sovrapposizione tra il nuovo volto e il volto rilevato nel frame precedente (con un margine di spostamento di 10 pixel al più):

```
social_distancing.D1 = ((reference_focal_length * reference_face_height * height) / (face_height * sensor_height)) / 10
if social_distancing.D1<300:
    for j in range(0,1):
        if social_distancing.facenum[j] != 0:
            x1_j = social_distancing.facecheck[j][0]
            y1_j = social_distancing.facecheck[j][1]
            x2_j = social_distancing.facecheck[j][0] + social_distancing.facecheck[j][2]
            y2_j = social_distancing.facecheck[j][1] + social_distancing.facecheck[j][3]
            if x1 > x1_j - 10 and y1 > y1_j - 10 and x2 < x2_j + 10 and y2 < y2_j + 10:
                social_distancing.facenum[j] = social_distancing.facenum[j] + 1 # la faccia è nel range di quella precedente
                social_distancing.faceflag[j]=1 #flag per evitare falsi positivi
        else:
            social_distancing.facenum[i] = social_distancing.facenum[i]+1
```

Figura 2.11: Controllo sulla face recognition per evitare falsi positivi

Se ciò avviene, viene incrementata una variabile relativa a tale volto, e, dopo un numero fissato di frame (da noi posto a 10), il volto verrà effettivamente considerato tale e ne verrà calcolata la distanza; in caso contrario,

con l'utilizzo di un vettore di flag, la variabile di questo volto viene riazzerata:

```

if social_distancing.facenum[i] > 10:
    social_distancing.D = ((reference_focal_length * reference_face_height * height)/(face_height*sensor_height))/10
if social_distancing.D < 300:
    output = cv2.circle(output, (mid_x, mid_y), 3, [255,0,0], -1)
    output = cv2.rectangle(output, (x1, y1) , (x2,y2) , [0,255,0] , 2)
    output = cv2.putText(output, str(round(social_distancing.D,2)) + " cm", (x1, y2+30), cv2.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2, cv2.LINE_AA)
    if social_distancing.D<150 and social_distancing.D!=0:
        output = cv2.putText(output, "Attenzione!!", (100, 400), cv2.FONT_HERSHEY_SIMPLEX, 2, [0,0,255] , 4)
else:
    social_distancing.D = 0
    social_distancing.facenum[i] = 0

```

Figura 2.12: Rilevamento dei volti e stampa della distanza

Le rilevazioni effettuate vengono infine salvate in una variabile di output, che verrà poi utilizzata nel main per sommare tutte le informazioni e mostrare a video.



Figura 2.13: Esempio di applicazione della face recognition

La face recognition è risultata efficace sia per il riconoscimento dei volti con la mascherina indossata che senza (vedi video test e screen nella cartella allegata).

### 2.2.2 Play\_Sound

Questa funzione si occupa, mediante la libreria *winsound*, di emettere dei bip tanto più frequenti quanto più l'altra persona è vicina. Per fare ciò, il tempo di ripetizione dei bip è legato alla distanza calcolata dall'altra persona mediante un'espressione matematica, più precisamente

$$Time = Distance \times e^{1.3} \quad (2.2)$$

$e^x$  è una funzione monotona crescente, quindi tanto più aumenta la distanza, tanto più i bip saranno meno frequenti.

Chiaramente si è ritenuto utile far sì che il suono venisse emesso solo laddove la distanza stimata sia insufficiente a garantire un adeguato distanziamento sociale (circa 150 cm).

```

def play_sound(self):
    D_new = int(social_distancing.D*math.exp(1.3))
    if self<150 and self!=0:
        winsound.Beep(social_distancing.frequency, D_new)

```

Figura 2.14: Codice per i suoni prodotti tramite winsound

### 2.2.3 Limitazioni del Social Distancing

Questa funzione di rilevamento volti e distanziamento sociale presenta alcune criticità legate a limitazioni proprie della natura del progetto. L'*Haar Cascade Classifier*, infatti, per quanto efficiente non riesce ad essere troppo preciso e tende a rilevare volti immaginari anche dove non ve ne siano. In più, il rilevamento della distanza è effettuato mediante una formula inversa che dipende, come già detto, da diversi fattori, propri della telecamera: richiede quindi una configurazione prima del primo utilizzo (o l'implementazione di ulteriori librerie e funzioni che automatizzino il processo, ritenute non fondamentali nell'ambito di questo progetto). Infine, la libreria *winsound* scelta per effettuare l'output sonoro è fortemente legata all'ambiente Windows; questo rende il progetto meno universale (seppur tale libreria possa facilmente essere reperita e installata su altre piattaforme). La scelta di utilizzare *winsound* invece di altre scelte più universali è legata alla possibilità di regolare frequenza e durata del suono. Un'altra limitazione che si è scelto di imporre è di non rilevare volti troppo distanti, poiché questo comprometterebbe il funzionamento delle altre parti di codice (semafori e marciapiedi). Solo i volti sotto i 3 metri di distanza stimata saranno rilevati e segnalati a schermo e tramite audio;

questa scelta si è rivelata necessaria anche e soprattutto a causa del difetto dell'*Haar Cascade Classifier*, che rileva volti “fantasma” anche molto distanti.

## 2.3 Rilevamento dei marciapiedi

Il rilevamento dei marciapiedi si basa sul riconoscimento delle linee, ma questo comporta molte limitazioni poiché non vengono rilevate solamente le linee dei marciapiedi e quindi è necessario svolgere, attraverso il codice, un’operazione di *pruning* in modo tale da scartare tutte le linee estranee all’obiettivo imposto.

### 2.3.1 Edge\_detect

Inizialmente avviene il rilevamento delle linee, attraverso il metodo blur si effettua una riduzione del rumore, in seguito si procede con il riconoscimento dei bordi usando il metodo *Canny*.

```
def edgeDetect(img): #rilevamento bordi
    img = cv2.GaussianBlur(img, (3, 3), 0)
    edges = cv2.Canny(img, 100, 150)
    return edges
```

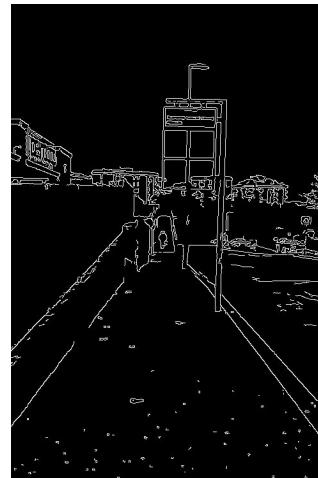


Figura 2.15: Frame dopo l’applicazione di Canny

### 2.3.2 Roi

Successivamente viene estratta una ROI così da elaborare le linee situate solamente nella parte inferiore del video, in cui si dovrebbe trovare il marciapiede, in questo modo vengono eliminate le prime linee di disturbo.

```
def roi(img, vert): #estrai una region of interest
    mask = np.zeros_like(img)
    cv2.fillPoly(mask, vert, 255)
    return cv2.bitwise_and(img, mask)
```



Figura 2.16: Frame dopo aver definito la ROI

### 2.3.3 Run

Ed infine usando la trasformata di Hough probabilistica si esegue il rilevamento delle linee. Per quanto riguarda i parametri usati in questi metodi la scelta è stata fatta in modo empirico .

Una volta ottenute le linee candidate al riconoscimento dei marciapiedi attraverso i loro punti iniziali e finali vengono calcolate la lunghezza di ogni linea, l’angolo sotteso tra ogni linea e l’asse orizzontale ed il coefficiente angolare di ogni linea.

Attraverso l’angolo le linee possono essere divise in linee pressoché verticali e linee pressoché orizzontali. Usando la posizione dei punti iniziali e finali delle linee può avvenire una suddivisione in due parti, linee che si trovano nella parte sinistra del frame e linee che si trovano nella parte destra.

Combinando questi due fattori le linee vengono suddivise in tre liste: linee verticali sinistre, verticali destre e orizzontali. In questo modo vengono scartate le linee oblique che possono essere una fonte di disturbo. Le liste di linee appena ottenute vengono poi ordinate in base alla loro lunghezza seguendo un ordine decrescente e infine si tronca la lista ai primi valori così da mantenere le linee più grandi, ovvero le principali linee candidate come marciapiedi.

```

slope=math.atan2(p2[1]-p1[1], p2[0]-p1[0]) #calcolo angolo tra la linea e l'asse orizzontale
if p2[1]-p1[1] !=0:
    m=(p2[0]-p1[0])/(p2[1]-p1[1]) #calcolo coefficiente angolare linea
else:
    m=-np.inf

#divisione in liste delle linee in base a angolo e posizione del punto più in basso
if(p2[0]<=250 and slope > math.radians(95) and slope< math.radians(150)):
    leftList.append([distance,p1,p2,slope,m])

elif (p2[0]>250 and slope > math.radians(30) and slope< math.radians(85)):
    rightList.append([distance,p1,p2,slope,m])

elif(((slope > math.radians(0) and slope<math.radians(15)) or (slope > math.radians(165) and slope<math.radians(180))) and distance>200):
    horizontal.append([distance, p1, p2, slope, m])

```

Figura 2.17: Funzione che definisce le linee del marciapiede

Le linee sinistre e destre, prese a due a due, vengono prolungate fino a intersecarsi (nel punto rosso) oppure fino al limite superiore del frame, successivamente si esegue una media dei valori ottenuti. Il risultato di quest'ultima operazione viene usato per definire se è opportuno girare leggermente a sinistra (punto di intersezione nella parte sinistra del frame) o girare leggermente a destra (punto di intersezione nella parte destra del frame) o infine continuare dritto (punto di intersezione più o meno al centro), a seconda della condizione verificata si incrementa il relativo contatore e si azzerano gli altri.

Le linee orizzontali vengono confrontate con le linee che si trovano nella parte sinistra e destra. Se tutte le linee orizzontali si trovano sopra le altre ci troviamo in una situazione in cui il marciapiede volge alla conclusione e quindi è necessario fermarsi e viene incrementato un contatore in caso contrario viene azzerato. Si è scelto di usare dei contatori poiché in alcuni casi vi erano dei falsi positivi i quali davano un risultato errato, con il contatore è possibile definire una soglia minima di frame in cui tale condizione deve essere vera così da avere risultati più attendibili. Superata la soglia definita in precedenza attraverso un output sonoro si comunica all'utente l'azione da eseguire.



Figura 2.18: Risultato finale

### 2.3.4 Limitazioni del rilevamento dei marciapiedi

La principale limitazione di questo codice è il riconoscimento delle linee. Infatti spesso vengono riconosciute delle linee non appartenenti ai marciapiedi come ad esempio pali delle indicazioni stradali, bici, tombini o altro che si trova situato nei pressi del marciapiede. Un altro problema è dovuto anche allo stato dei marciapiedi come ad esempio marciapiedi dismessi oppure pieni di foglie come nei periodi autunnali. Attraverso il codice è stato possibile limitare questi inconvenienti ma non eliminarli del tutto.

# Capitolo 3

## Main e Test

La funzione *main* svolge il compito di congiungere le funzioni dei tre file *.py* appena descritti; cattura il video in input e, tramite il metodo *cap.read()*, rende disponibile il frame attuale alle altre funzioni, le quali svolgono il loro compito.

Sono stati effettuati diversi test, utilizzando video di natura diversa, tra cui:

- video scaricati da internet
- video registrati con iPhone 6
- video registrati con iPhone 8
- test con la webcam del computer

per ogni test bisogna apportare piccole modifiche, che dipendono dalla natura del dispositivo che ha effettuato la registrazione video: vengono infatti operate rotazioni (laddove è necessario), conversioni tra lunghezza e altezza per il riconoscimento facciale e vengono cambiate le caratteristiche relative alla fotocamera.

La gestione dei suoni è affidata a tre diversi thread, questo per evitare che la riproduzione di un suono faccia rallentare eccessivamente il programma (in un'esecuzione puramente sequenziale il programma tende ad aspettare che il suono sia stato completamente eseguito prima di procedere al frame successivo, causando notevoli rallentamenti).

Le funzioni dei file sono state gestite in modo tale da non consentire la loro esecuzione contemporanea, poiché comporterebbe un rallentamento eccessivo dell'esecuzione: tramite una variabile di controllo (che funge anche da peso per la somma pesata delle tre funzioni) si può gestire il comportamento dell'applicativo: se in un frame viene rilevato un volto o un semaforo, viene impedito il rilevamento delle linee del marciapiede.

Qui di seguito vengono riportati alcuni screen sui test eseguiti.

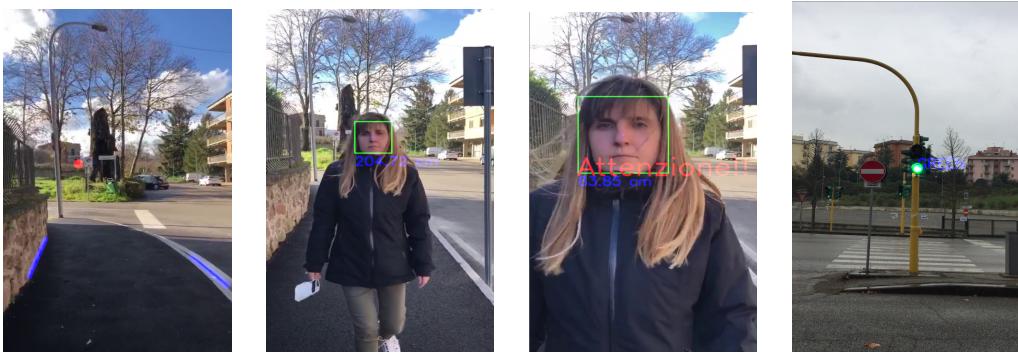


Figura 3.1: Alcuni test effettuati su video registrati con iPhone 6 e iPhone 8

# Capitolo 4

## Conclusioni

Grazie alla realizzazione di questo progetto il nostro gruppo è stato in grado di comprendere le potenzialità offerte da OpenCV e come la Computer Vision offra una potenza di calcolo tale da trovare terreno fertile tra i più svariati campi di applicazione. La realizzazione del progetto inoltre ci ha permesso di studiare e approfondire altri tipi di librerie e di sperimentare anche le tecniche legate al campo dell'Image Processing. Oltre a ciò abbiamo potuto testare anche i limiti che queste applicazioni comportano: infatti, come già è stato accennato in questo elaborato, si potrebbe pensare di rielaborare in un futuro il codice apportando diverse migliorie, come l'impiego del Machine Learning. Una volta risolti i difetti, questa applicazione può essere pensata per essere installata in un dispositivo mobile "wearable" con una camera incorporata, come ad esempio un cappello o una fascia per la testa.