

Tesina Image Processing and Computer Vision: Creazione di filtri usando OpenCV

Macaluso Antonio
Sergio Aurora Anna Pia



Premessa	3
Introduzione	3
Rossetto	4
Occhiali	5
Piercing	7
Septum	8
Sulle guance: blush e lentiggini	9
Barba	10
Cornici	12
GUI	12
Salvataggio Immagini	13
Limiti e riflessioni finali	14

Premessa

Questo progetto rappresenta un tentativo di applicazione di tecniche e concetti visti durante il corso di Image Processing and Computer Vision con lo scopo di realizzare qualcosa che l'utente medio di uno smartphone usa (quasi) quotidianamente nel proprio tempo libero, ossia i filtri. Ce ne sono centinaia, se non migliaia, già a disposizione degli utenti su ogni social network, ma esistono anche specifiche app dedicate all'applicazione di filtri estremamente variegati, più o meno complessi. Il nostro progetto però non ha l'obiettivo di fornire filtri migliori o più elaborati rispetto a quelli già esistenti, ma piuttosto vuole concentrarsi su ciò che è possibile realizzare dopo un corso di base sull'elaborazione di immagini, con alcuni approfondimenti che si sono resi necessari per ottenere un risultato soddisfacente. In seguito, illustreremo in generale le tecniche, le librerie e i metodi usati, per poi concentrarci sui singoli filtri.

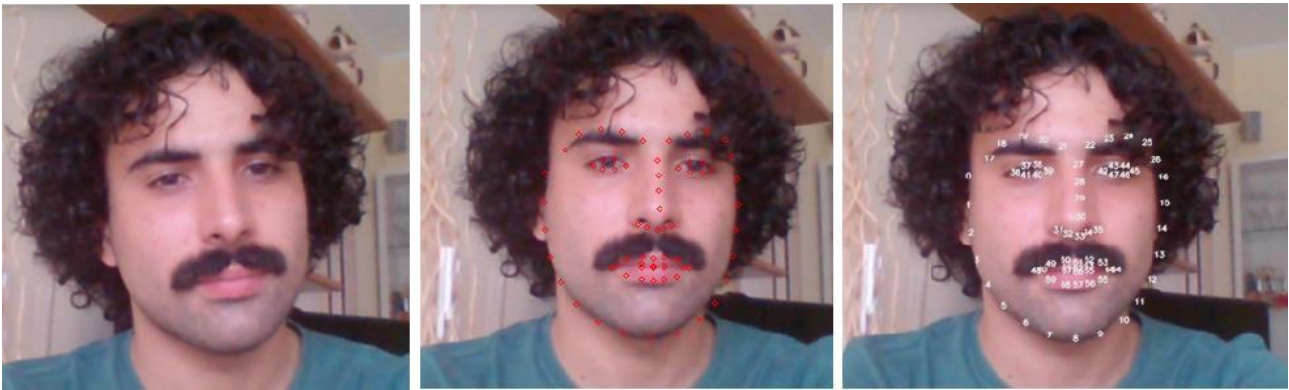


Filtri di Instagram

Introduzione

Il programma fa uso principalmente di due librerie, *opencv* e *dlib*, servendosi poi del supporto di *numpy* e *keyboard*. Tutte le operazioni sono effettuate sul flusso video acquisito dalla webcam principale; prima di tutto, ad ogni frame viene applicato un filtro gaussiano per la riduzione del rumore, mediante il metodo `cv2.GaussianBlur()`. Dopo questa operazione preliminare si passa alla fase di face detection.

In particolare, abbiamo deciso di far uso di face detector e shape predictor forniti dalla libreria *dlib*, che insieme permettono il riconoscimento di volti umani e la localizzazione di 68 punti del volto che aiutano a lavorare su zone specifiche. Il motivo per cui abbiamo preferito questo agli strumenti visti durante i laboratori dedicati al riconoscimento e tracciamento dei volti è proprio che i 68 punti forniti permettono di usare maggiore precisione al momento dell'elaborazione delle immagini, senza pesare significativamente sulle prestazioni. Qualora lo si desideri, durante l'esecuzione del programma si possono visualizzare questi punti premendo il tasto 'x' (alternando tra visualizzazione disattivata, visualizzazione dei punti, e visualizzazione dei punti numerati).



Modalità debug: disattivata (sx), punti (centro) e punti numerati (dx)

La libreria `opencv` è invece utilizzata massicciamente nella fase successiva, dedicata all'elaborazione dei singoli frame ed effettuata interamente nel metodo `apply_mask(image_filter, frame, landmarks)`. Questo metodo riceve come primo parametro un flag intero che serve a scegliere quale filtro applicare tra quelli implementati, il frame corrente e l'insieme delle coordinate dei punti identificati precedentemente. È possibile passare da un filtro all'altro (o alla modalità senza filtri) durante l'esecuzione del programma premendo i tasti '+' (passa al filtro successivo) o '-' (passa al filtro precedente).

In fase di esecuzione, può capitare che gli input da tastiera non vengano rilevati al primo clic sul pulsante: in tal caso, è sufficiente riprovare digitando nuovamente l'input. Capita anche che vengano rilevate due pressioni di un pulsante quando lo si preme una sola volta, saltando un filtro o una cornice.

Questo metodo svolge due azioni: la prima è l'aggiunta di testo per indicare il filtro corrente e la modalità di visualizzazione dei punti del volto, insieme a istruzioni sintetiche sull'uso del programma; la seconda è la selezione della funzione da applicare al frame per modificarlo, in base al flag ricevuto come parametro. In seguito si descrivono i vari metodi che implementano l'applicazione dei filtri, mostrandone anche i risultati.

1. Rossetto

Il metodo che implementa questo filtro è `change_lips(img, lm)` che riceve come parametri il frame corrente e l'insieme dei landmarks. Questo filtro utilizza una procedura particolare che verrà adottata solo in questa opzione. Per prima cosa, vengono creati due array di punti, `upper_lip` e `lower_lip`, contenenti le coordinate dei landmarks corrispondenti al contorno labbra. Viene poi inizializzata una maschera "vuota" delle stesse dimensioni del frame video, specificando il tipo di dato contenuto (`dtype=np.uint8`) per garantire compatibilità con il frame al momento della somma successiva. Sulla maschera viene poi usato il metodo di `opencv` `fillPoly()`, per fare in modo che tutti i punti della maschera interni ai poligoni corrispondenti alle labbra vengano "colorati" con un tono rosso. Infine il metodo restituisce il risultato di una somma pesata, effettuata mediante `cv2.addWeighted()`, in cui alla maschera viene assegnato un peso minore per fornire un risultato un po' più realistico.



Applicazione del filtro "Rossetto"

2. Occhiali

Il metodo che implementa gli occhiali, `glasses_filter(image, lens_image, sl_image, sr_image, lm)`, riceve come input il frame corrente del feed webcam, tre immagini separate a rappresentare rispettivamente le lenti e le stanghette (destra e sinistra) degli occhiali e l'insieme dei landmarks designati. All'interno del metodo sono contenuti 3 array di tuple descriventi le superfici dove applicare le immagini passate come argomento.

Per fare ciò, viene invocato per ogni sezione il metodo `perspective_image(img, foreground_path, myPoints)`, che riceve il feed della cam menzionato sopra insieme all'array di tuple e l'immagine da applicare. Il metodo accoppia i metodi di opencv `getPerspectiveTransform` e `warpPerspective`

- Il primo calcola la matrice di trasformazione M per mappare il componente nei 4 punti designati.
- Il secondo si serve di M per applicare di fatto una trasformazione prospettica.

Il risultato della trasformazione (il pezzo dei nostri occhiali correttamente trasformato e posizionato all'interno del viewport) viene successivamente unito al background. Per fare ciò, abbiamo creato un metodo ad hoc chiamato `overlay_png` di cui tratteremo più avanti.

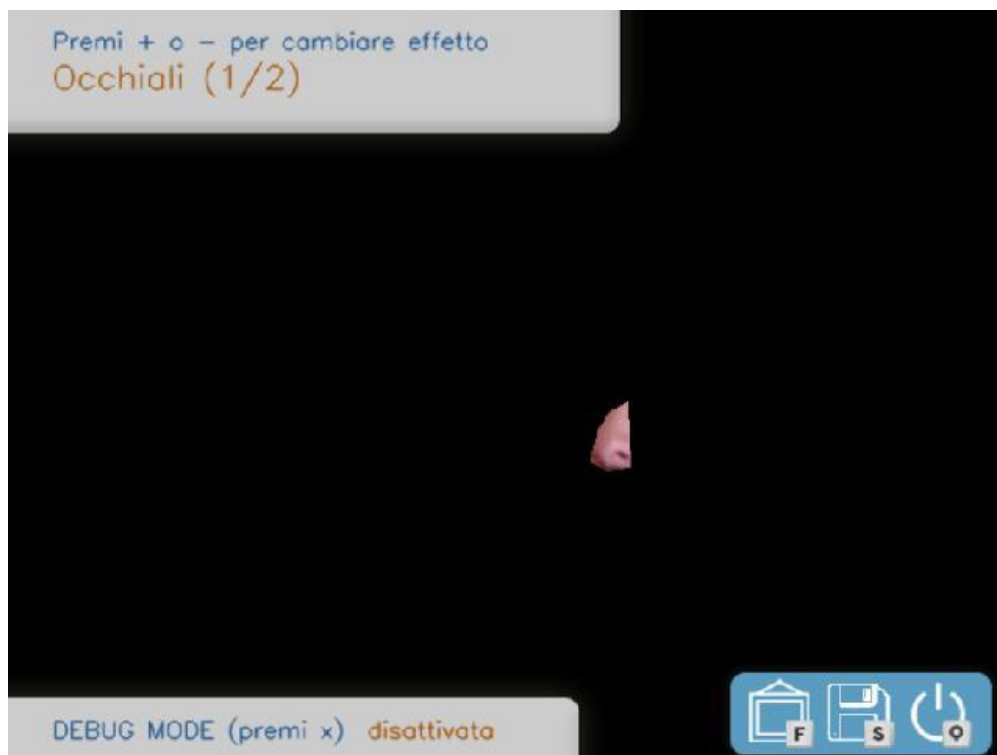
In alcuni casi limite, il risultato ottenuto presentava risultati collaterali sulle stanghette: laddove i punti dei landmarks erano eccessivamente vicini l'immagine produceva alterazioni fastidiose che si propagavano su tutto lo schermo. Per risolvere il problema, abbiamo inserito un semplice check sui landmarks in alto a sinistra ed in alto a destra: nel momento in cui la loro distanza sull'asse x è inferiore ai 5 pixel (sintomo di un'inclinazione della testa tale da celare la stanghetta), l'immagine di foreground viene rimossa dal

rendering del frame corrente. Questo piccolo accorgimento rende più realistico l'output in quanto la stanghetta dietro la testa viene rimossa dal rendering conferendo un senso di tridimensionalità.

Un altro minore accorgimento è dedicato al naso, in cui si lavora all'interno del metodo `nose_overlay`: a partire dal frame della webcam, creo una maschera per il naso sulla base di una serie di tuple grazie a `fillConvexPoly` e alla funzione di numpy `zeros_like`. L'uguaglianza `nose_mask[mask] = image[mask]` restituisce il viewport annerito ad eccezione del naso.

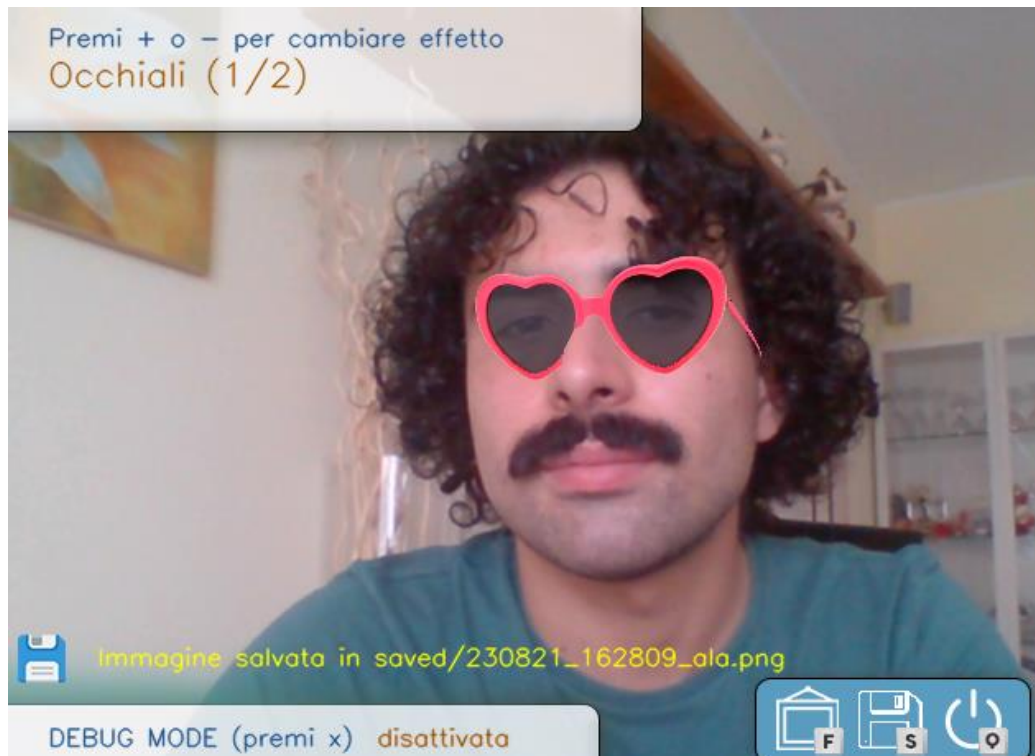
Per conferire al naso un aspetto tridimensionale abbiamo deciso di lavorare su diversi array di tuple uniti a comporre una maschera il più soddisfacente possibile:

- il primo array si occupa della sezione frontale del naso, operante su 5 punti di cui 4 interpolati come valori intermedi
- il secondo e il terzo array si occupano delle facce laterali (in maniera cautelare) per evitare che tracce degli occhiali possano comparire sovrapposte sul naso ad inclinazioni significative del viso



Maschera del naso (acquisita in fase di debug): a viso inclinato, il naso sporge anche lateralmente, nonostante non vi siano landmarks, grazie ai calcoli svolti sulle frazioni intermedie

La sezione finale del metodo di serve dei metodi di `thresholding`, `split` e `merge` offerti da `openCV` per fornire alla maschera un canale alpha e rimpiazzare così la componente nera dell'immagine con una trasparente. Al termine di questa operazione, posso finalmente unire all'immagine ottenuta la componente `frame + filtro` così da simulare (seppur in maniera semplificata) la profondità del naso rispetto al resto del viso.



L'inclinazione cela la stanghetta a sinistra dell'osservatore. Si può inoltre notare come il naso venga renderizzato sopra gli occhiali conferendo un senso di profondità.

Per adempiere alle numerose sovrapposizioni di immagini in png abbiamo selezionato un piccolo metodo chiamato `overlay_png` che normalizza i canali alpha di background e foreground nell'intervallo (0-1), sovrappone i colori e il canale alpha aggiustati e denormalizza all'intervallo originale (0-255). Per svolgere questa operazione le immagini devono essere memorizzate con il canale alpha.

- Il foreground è già caricato come argomento in formato png usando `cv2.imread` con il flag `cv2.IMREAD_UNCHANGED`
- Il background riceve una conversione dello spazio di colore utilizzando `cv2.cvtColor` con il flag `cv2.COLOR_RGB2RGBA`, che aggiunge un canale alpha all'immagine RGB

3. Piercing

Il metodo `add_eyebrow_piercing()` apre un'immagine in formato png raffigurante il piercing da applicare e inizializza una maschera come descritto in precedenza. Procede poi a identificare alcuni punti a partire dai landmarks. In particolare:

- `eyebrow_up` è uno dei punti del sopracciglio destro, quasi alla fine
- `ref_pt` è un punto intermedio le cui coordinate sono calcolate come media aritmetica di quelle di due punti consecutivi dell'occhio destro, preso come riferimento per i passi successivi
- infine, il punto `center` ha coordinate calcolate come media aritmetica delle coordinate dei due punti precedenti.

Contemporaneamente vengono calcolati anche i valori di `height` e `width`, scelti uguali tra loro e pari alla metà della distanza verticale tra `eyebrow_up` e `ref_pt`. L'immagine letta viene a questo punto ridimensionata mediante `cv2.resize()` in modo che il piercing assuma dimensione `width x height`. A questo punto si passa alla selezione di una regione di interesse sulla maschera, di forma rettangolare: in verticale, la roi va dal sopracciglio (`center - height`) al `center`, mentre in orizzontale parte da `center + half_width` e prosegue fino a `center + width`. La somma successiva effettuata con `cv2.add()` modifica la roi appena descritta usando la seconda metà (in orizzontale) dell'immagine del piercing; in questo modo si dà l'impressione che il piercing applicato sia un mezzo anello intorno al sopracciglio. La nuova roi modificata va a sostituire la vecchia roi nella maschera, e il metodo ritorna il risultato della somma dell'immagine di partenza con la maschera costruita come descritto.

4. Septum

Il metodo `add_septum()` funziona in modo molto simile al metodo `add_eyebrow_piercing()`: cambiano esclusivamente i punti considerati e la regione di interesse ritagliata dalla maschera e dall'immagine raffigurante il piercing (la stessa usata precedentemente). I punti usati sono quelli corrispondenti ai limiti interni delle due narici, `midnose_right` e `midnose_left`, la punta del naso, `nosetip`, il punto centrale del labbro superiore (usato come riferimento per il calcolo di un centro e dell'altezza dell'immagine), `center_upper_lip`, e un centro che regola la posizione del piercing.

In questo caso l'ampiezza del piercing viene calcolata come la distanza orizzontale tra le due narici (`width = midnose_right[0] - midnose_left[0]`), mentre la sua altezza sarà la metà della distanza verticale tra la punta del naso e il centro del labbro superiore. Il centro, a questo punto, è stato scelto a metà tra le due narici e a circa due terzi dell'altezza, partendo dalla punta del naso (ovvero `nosetip[1] + height - one_third_height`, scritto in questo modo per garantire che dopo la divisione tra interi non venga "sprecato" il resto, che altrimenti determinerebbe leggere differenze tra le dimensioni delle matrici coinvolte che causerebbe l'arresto del programma in corrispondenza delle somme).

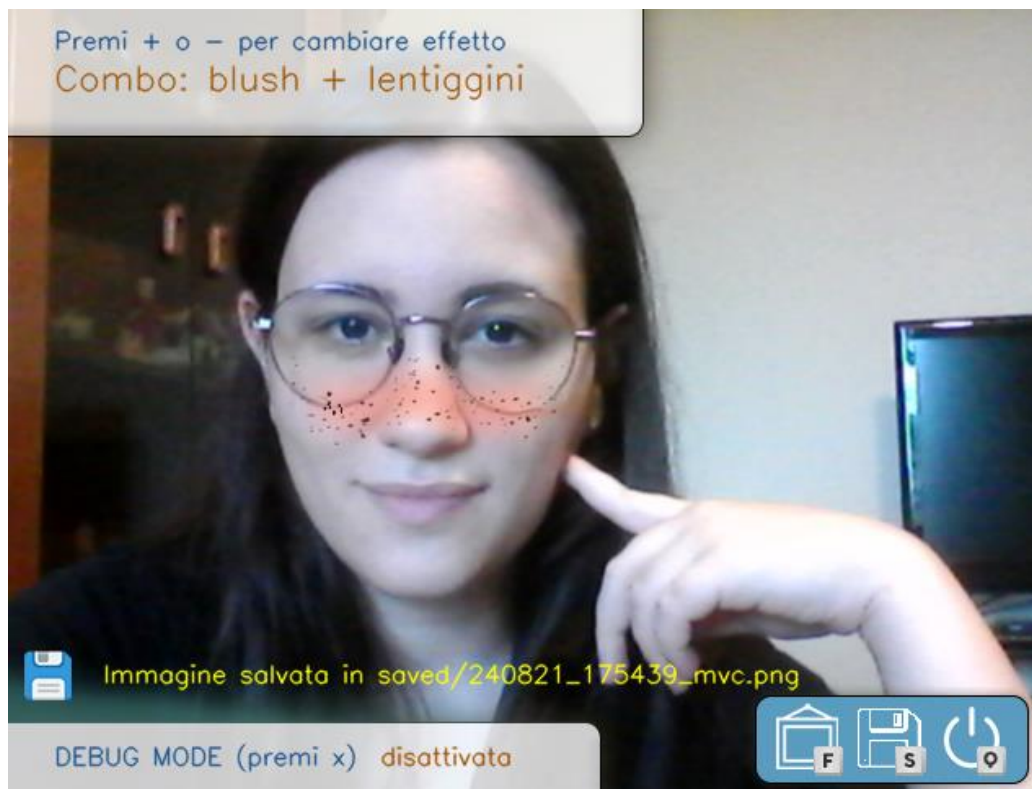
A questo punto si effettua un `resize` dell'immagine del piercing, e poi si seleziona la regione di interesse della maschera: a partire dal centro (`center`), verso l'alto per un terzo dell'altezza e verso il basso per i restanti due terzi; in orizzontale, sempre partendo dal centro, per metà verso sinistra e per il resto verso destra. Questa viene sommata a `piercing`, e il risultato va a sostituire la vecchia roi ma non interamente: solo i due terzi inferiori della roi vengono modificati. In questo modo si ottiene un effetto simile alla presenza di un septum.



Applicazione di piercing e septum

5. Sulle guance: blush e lentiggini

Dal momento che sia le lentiggini che i due blush vengono applicati alla stessa regione, è bastato un unico metodo per realizzarli tutti, passando tra i parametri anche il path dell'immagine in formato png da aprire. I punti rilevanti per questa categoria di filtri sono quattro: tempie destra e sinistra (`left_cheekbone` e `right_cheekbone`) e altri due punti ottenuti dai primi due e dai punti corrispondenti all'inizio della mandibola: `down_left` ha come coordinata orizzontale la media delle coordinate orizzontali di `left_cheekbone` e della mandibola, e in verticale ha la stessa coordinata della mandibola; `down_right` è analogo a `down_left` ma considera i punti sul lato destro del volto. Questi quattro punti sono poi immagazzinati in un vettore passato, insieme al frame corrente ed al path dell'immagine corrispondente al filtro scelto, a `perspective_image()`, di cui si è già parlato nella sezione dedicata agli occhiali.



Combinazione di blush + lentiggini

6. Barba

Il metodo `add_beard()` fa nuovamente uso di `perspective_image` per mappare il template della barba sul viso.



beard.png



beard_dx.png

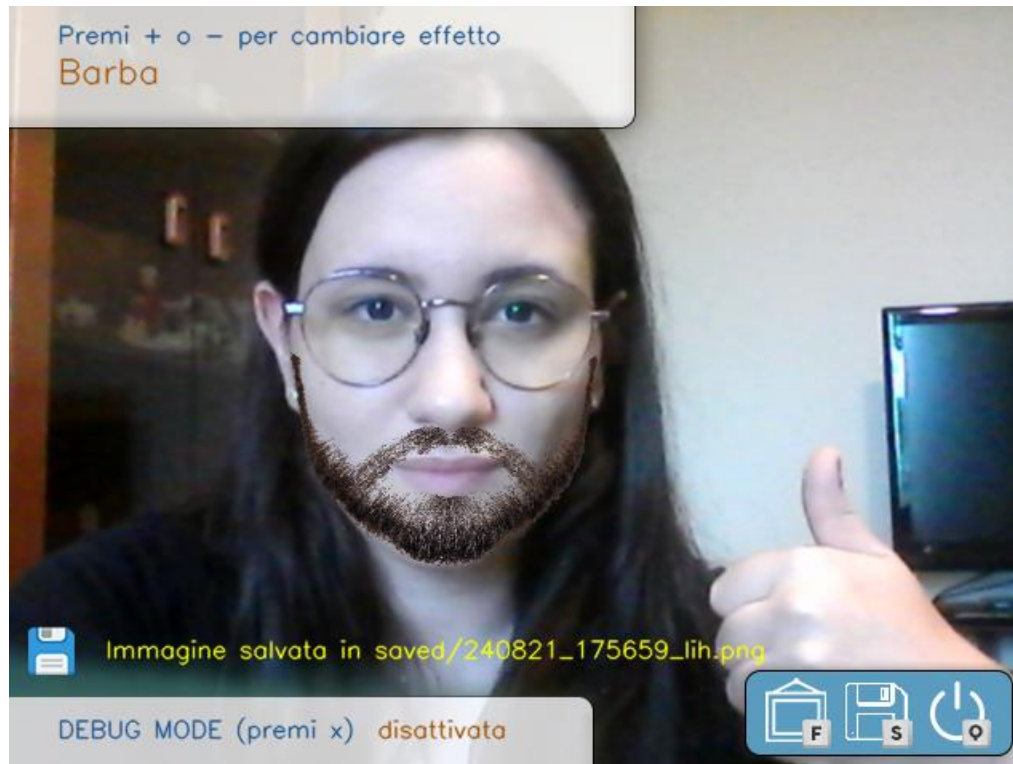


beard_sx.png

- Durante una prima stesura del codice, abbiamo provato ad utilizzare una singola mappatura del file `beard.png` su 4 punti con risultati non soddisfacenti: il movimento del viso (specialmente quando inclinato) non seguiva quello della barba in virtù dei punti selezionati, solo 4 ed interpolati sulla base dei landmarks già forniti dal dataset. Pertanto, la barba appariva monodimensionale.
- In un secondo momento abbiamo deciso di separare la barba nella sua componente destra e sinistra e operare su 6 punti differenti (`beard_sx.png` e `beard_dx.png` mappati in due piani

da 4 punti con 2 punti in comune nell'asse del viso): con questo sistema la barba segue con maggior fedeltà il viso rilevato, anche con inclinazioni più significative.

- La nuova mappatura presentava tuttavia un nuovo problema: l'immagine della barba copriva spesso le labbra. Per risolvere, abbiamo scritto il metodo `mouth_overlay` che opera sul frame in maniera analoga al precedente `nose_overlay` ma, come suggerisce il nome, su landmarks differenti: il metodo rimappa le labbra del frame originale su quello elaborato in modo tale da cancellare i peli facciali nelle zone indesiderate.



Barba

7. Cornici

Tra le piccole funzionalità di contorno vi è la possibilità di aggiungere una cornice alla nostra immagine, selezionando con la pressione del tasto **f** tra la serie di opzioni offerte. Il metodo invocato è il già menzionato `overlay_png()` a cui fornisco come input sia l'acquisizione webcam unita alle elaborazioni precedenti che il filtro in png (caricato con il metodo `imread` con annesso flag `cv2.IMREAD_UNCHANGED` per preservare il canale alpha e con esso le informazioni sulla trasparenza).



8. GUI

Per semplificare l'esperienza di navigazione il programma prevede la presenza di una GUI contenente un layout di base realizzato con Photoshop ed una serie di scritte contestuali che variano in base ai filtri e alle maschere utilizzate.

L'output presentato su schermo è il frutto di una serie di sovrapposizioni di layers contenenti i vari stadi di elaborazione dell'immagine, contenuti all'interno del main ed aggiornati in maniera ciclica ogni volta che il nuovo frame viene catturato dal feed della webcam.

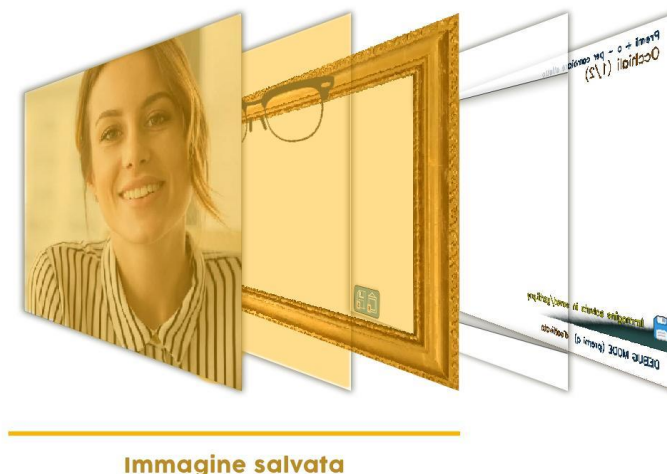
- Il primo layer contiene l'**acquisizione della webcam**
- Il secondo layer contiene, in caso di rilevamento di uno o più visi, la **maschera selezionata**
- Il terzo layer contiene la **cornice selezionata** dall'utente
- Il quarto layer contiene la **GUI**
- Il quinto layer contiene tutte le **informazioni testuali**



Per motivi di sintesi, il quinto layer nell'immagine contiene anche la coppia di layer contestuali (GUI+testo) dedicati all'avviso di immagine salvata correttamente (trattata nel paragrafo successivo).





9. Salvataggio Immagini

Durante l'esecuzione del programma, la pressione del tasto **s** permette di memorizzare nella cartella "saved" all'interno del programma una copia in .png della nostra immagine modificata. L'immagine da salvare viene conservata ad ogni esecuzione del frame in una variabile apposita prima che il ciclo di sovrapposizione dei layers venga completato (in particolare, prima che venga aggiunta la GUI in modo tale che l'utente non abbia livelli superflui nel suo file salvato in memoria). Un livello di GUI contestuale avvisa l'utente del successo dell'operazione.



L'immagine viene salvata col comando di opencv `cv2.imwrite(filename, saving_frame)`

- `saving_frame` contiene il frame da esportare
- `filename` si serve delle librerie `random` e `datetime` per generare il filename contenente data (DDMMYY), ora (HHMMSS) ed una stringa randomica di tre caratteri per prevenire una sovrascrittura nel caso di due immagini salvate in rapida successione (nello stesso secondo).

Name	Date	Type
 220821_124523_cso.png	22/08/2021 12:45	PNG File
 220821_124523_elo.png	22/08/2021 12:45	PNG File
 220821_124523_mqj.png	22/08/2021 12:45	PNG File
 220821_124527_wpm.png	22/08/2021 12:45	PNG File

10. Limiti e riflessioni finali

Durante il processo di realizzazione del progetto ci siamo accorti delle imperfezioni e dei limiti di quest'ultimo. Quando possibile, ci siamo adoperati per contenerli:

- Abbiamo riscontrato come alcuni dei movimenti laterali della testa producessero risultati non realistici a causa dell'effetto sul naso. Usando `nose_overlay()` e `perspective_image()`, rispettivamente per mostrare il naso quando questo è davanti al filtro e per fare in modo che il filtro segua la prospettiva data dalla posizione del volto, il risultato è più fedele a quello desiderato (e desiderabile). Tuttavia si possono vedere anche situazioni in cui questo non basta, ad esempio per movimenti molto ampi del volto. Per questi probabilmente servirebbe un modello di riconoscimento delle feature facciali più elaborato di quello da noi usato, oppure bisognerebbe sottoporre un modello già esistente ad un processo di training per migliorarne i risultati.
- Un altro problema che risulta fastidioso è che i landmarks non sono ben stabili, di conseguenza si vedono spesso dei tremolii sui filtri. In questo senso, già l'uso del filtro gaussiano per la rimozione del rumore ha aiutato a ridurre tale effetto rispetto alla versione del programma che agiva direttamente sul frame così come veniva ricevuto dalla webcam, ma per minimizzarlo di nuovo bisognerebbe ricorrere a tecniche di machine learning più avanzate che portino il riconoscimento facciale a fornire un risultato più stabile.
- Infine abbiamo notato che il frame rate è piuttosto basso, ma riteniamo sia inevitabile per via di tutta l'elaborazione che viene fatta sull'immagine e pertanto non abbiamo potuto intervenire da questo punto di vista.

Rimaniamo comunque soddisfatti del lavoro svolto, poiché ha richiesto da parte nostra non solo la "semplice" applicazione di quanto visto a lezione, ma anche la ricerca di altro materiale e un processo di sperimentazione fino ad ottenere il risultato migliore possibile; è stato quindi un progetto stimolante da svolgere e che ha arricchito e consolidato le capacità acquisite durante il corso.

Grazie per l'attenzione