UNIVERSITÀ DEGLI STUDI DI SALERNO

DIPARTIMENTO DI INFORMATICA

TESI DI LAUREA IN INFORMATICA

POSE RECOGNITION PER L'INTERAZIONE UMANA IN AMBIENTE VIDEOLUDICO

Relatore
Ch.mo Prof.
Andrea Francesco Abate

Candidato
Francesco Parisi
Matr. 0512103582

Correlatore
Ch.mo Prof.
Ignazio Passero

ANNO ACCADEMICO 2019/2020

Indice

Elenco dei Codici

Elenco delle Figure

Abstract

Questo lavoro di tesi consiste nello sviluppo di un sistema per il riconoscimento facciale, la stima della direzione del volto e la sua applicazione in ambientazioni 3D e videogames. La tesi ha come obiettivo principale quello di creare un sistema di rilevamento e riconoscimento del volto, in particolare si è voluto creare un sistema in grado di acquisire ed elaborare una sequenza di immagini allo scopo di estrarre le informazioni relative al volto riconosciuto per inviarle ad un server tcp, incaricato di decodificarle per creare una mappatura 3D e per determinarne il suo orientamento. La tesi è caratterizzata da tre fasi: la prima in cui si acquisiscono e preparano le immagini al riconoscimento, la seconda in cui avviene l'estrazione delle informazioni del volto riconosciuto e la mappatura in 3D, la terza in cui se ne determina l'orientamento. La prima fase comprende l'acquisizione di una sequenza di immagini da webcam e il riconoscimento del volto. L'acquisizione delle immagini è ottenuta grazie a un algoritmo sviluppato in python che sfrutta la libreria OpenCV e Scikit-image. Il sistema impiega una rete neurale in grado di localizzare 68 punti di riferimento del viso (landmarks). I landmarks vengono organizzati in 9 aree specifiche del viso: contorno facciale, sopracciglio sinistro e destro, naso, narici, occhio sinistro e destro, labbra e denti. La seconda fase comprende l'estrazione delle informazioni, la codifica in un formato specifico, il loro invio a un server tcp in Unity e la mappatura 3D. Le informazioni ottenute riguardano le coordinate di ciascun landmark nelle posizioni x, y e z. Tali informazioni vengono manipolate e inviate al server tcp in Unity, permettendo la creazione di una mesh 3D del viso. La terza fase comprende la stima della direzione del volto, ottenuta sfruttando 3 landmarks specifici dai quali è possibile calcolare il punto esatto in cui l'utente sta guardando, grazie alla creazione di un raggio luminoso. Infine tale lavoro è stato integrato in diversi ambienti di gioco, mettendone in evidenza il suo scopo finale.

Capitolo 1

Introduzione

Nella tesi in questione si vuole analizzare e descrivere i metodi e strumenti utilizzati per la realizzazione di un sistema orientato al riconoscimento facciale e la sua applicazione in ambienti di gioco interattivi. Così come la tecnologia moderna si concentra sul riconoscimento di volti da immagini in due dimensioni, lo sviluppo del sistema ha previsto l'utilizzo di sequenze di immagini in 2D, acquisite da una webcam, e tramite una rete neurale, nota come "Face Alignment Network", vengono estrapolate le informazioni 2D e 3D necessarie per la rappresentazione del volto. Ne sono un esempio Facial Recognition Software (FRS) [1], Face Recognition Vendor Test (FRVT) [2] e FindFace Facial Recognition [3]. In questa tesi utilizzeremo le informazioni 3D ottenute per mappare il volto riconosciuto, rappresentandolo attraverso una mesh. L'approccio utilizzato in generale non risente dei soliti problemi che affliggono il riconoscimento bidimensionale, mostrando anche una notevole capacità di resistenza a posa, risoluzione, inizializzazione e persino illuminazione. Il sistema è stato sviluppato in modo che, una volta acquisite le immagini, queste vengono pre-processate dalla rete neurale, la quale provvede a identificarne il viso e procedere alla suddivisione di esso in punti di riferimento. Tale rete neurale consente l'identificazione della struttura geometrica dei volti e un allineamento canonico del viso basato su traslazione, scala e rotazione. Una volta ottenute le informazioni necessarie, tramite la libreria Matplotlib di python, queste vengono rappresentate in grafici 2D e 3D. I grafici si presentano suddivisi principalmente in landmarks. I landmarks sono collegati gli uni agli altri in base alla zona del volto che vanno a rappresentare, infatti questo approccio permette di mostrare il grafico sul volto riconosciuto da immagine bidimensionale data in input. Le informazioni che riguardano il grafico 3D rappresentato sono poi gestite lato python e inviate, secondo un formato specifico, ad un server tcp. Attraverso questa comunicazione, il sistema da Unity è in grado di rappresentare le informazioni ricevute in una mesh 3D del volto. Questo approccio si ripete per ogni foto acquisita, permettendo un movimento in real-time del volto identificato e mappato. Oltre alla rappresentazione della mesh 3D del volto si è pensato di capire dove l'utente stesse guardando. Tale idea è stata sviluppata prendendo in considerazione alcuni landmarks specifici del volto che, utilizzati per calcolare la

normale perpendicolare alla superficie della mesh, permettono di istanziare un raggio necessario a capire dove l'utente sta realmente guardando. Il sistema sviluppato è stato poi integrato in alcuni progetti di tesi dei miei colleghi per apportare una maggiore interazione all'applicativo da loro sviluppato. In particolare, tale sistema è stato utilizzato nell'ambito dei videogames, i quali sfruttando il Deep-learning e l'Emotion recognition in real-time, ciò ha permesso un maggior coinvolgimento nel gioco da parte dell'utente, ad esempio creando oggetti in base a dove l'utente stesse guardando.

1.1 Riconoscimento Facciale

Il riconoscimento facciale è una tecnica biometrica volta a identificare in modo univoco una persona, confrontando e analizzando modelli basati sui suoi contorni facciali. È un problema importante nella ricerca sulla visione artificiale con molte applicazioni come il controllo degli accessi ad un dispositivo, il riconoscimento dell'identità nei social media e i sistemi di sorveglianza. La maggior parte dei software di riconoscimento facciale si basano interamente su immagini 2D, infatti gran parte delle fotocamere scatta foto in 2D anche se le immagini non sono molto accurate poiché si tratta di immagini piatte e senza profondità, il che non porta ad avere caratteristiche di identificazione ben precise. Con un'immagine 2D un dispositivo non è in grado di misurare la profondità di una determinata zona del viso, inoltre l'immagine si basa sulla luminosità e ciò significa che tale tecnica può presentare problematiche al buio, quindi può risultare inaffidabile in condizione di scarsa illuminazione o con una particolare prospettiva. Tali problematiche vengono risolte grazie all'impiego del riconoscimento facciale 3D che si ottiene attraverso una tecnica chiamata "lidar" [4], che è simile al sonar. Se utilizzata insieme alla tecnologia 2D, il riconoscimento facciale 3D può aumentare significativamente la precisione dell'identificazione**,** diminuendo così la possibilità di avere falsi positivi. Pochi anni fa tale tecnologia risultava particolarmente onerosa non solo a livello economico ma anche a livello prestazionale, causa di un processo di acquisizione lungo e complesso. Il progresso tecnologico ha reso il processo di acquisizione più accessibile agli utenti, permettendo così di lavorare direttamente sulla mesh 3D del volto identificato. In parte tali tecnologie hanno influenzato molto gli algoritmi relativi al riconoscimento bidimensionale, riuscendo a estrapolare le informazioni richieste a partire dall'immagine data in input, tra i quali l'algoritmo Face Detection e Haar-like features [5]. L'algoritmo, che viene utilizzato per la rilevazione dei volti e basato sulle Haar wavelet è stato elaborato da Viola-Jones nell'ambito dell'Object Detection ed è stato pensato proprio per il problema della face detection in tempo reale. A differenza dell'uomo, un computer ha bisogno di istruzioni e vincoli ben precisi per poter riconoscere un volto e per rendere il compito più gestibile, inoltre Viola – Jones richiede l'utilizzo di facce verticali-frontali a piena vista. Per essere rilevato, l'intero viso deve essere rivolto verso la fotocamera e non deve essere inclinato su entrambi i lati. Questi vincoli potrebbero ridurre l'utilità dell'algoritmo, poiché il passaggio di rilevamento è spesso seguito da un passaggio di

riconoscimento, perciò tali limiti di posa possono non essere accettabili. Molto importante per il riconoscimento facciale è anche l'utilizzo della libreria OpenCV, grazie alla quale è possibile approfondire lo sviluppo della Computer Vision, elaborando immagini/video per numerose applicazioni, tra cui Augmented Reality, Object Recognition, Gesture Recognition, Pose Recognition, Emotion Recognition e Face Recognition. Con l'avvento del Deep Learning e lo sviluppo di grandi set di dati, i recenti lavori hanno mostrato risultati di accuratezza senza precedenti anche nelle attività di visione artificiale più impegnative.

1.2 Scopo della tesi

Questa tesi ha come obiettivo quello di sviluppare un sistema che basandosi sull'elaborazione delle immagini acquisite, sia in grado di riconoscere un volto, utilizzare le informazioni ottenute per creare una mappatura 3D in Unity e determinare il suo orientamento. La tecnica implementata utilizza le informazioni 2D e 3D ottenute attraverso l'impiego della libreria Matplotlib, generando una rappresentazione grafica dei landmarks del volto identificato e quindi fornendo informazioni ben precise che ne permettono una rappresentazione 3D accettabile.
Il lavoro di tesi svolto può quindi idealmente essere suddiviso in tre fasi:

- Lo sviluppo di un algoritmo che permetta di acquisire una serie di immagini da cui rilevare e riconoscere il volto dell'utente.
- L'estrazione delle informazioni di interesse necessarie per la mappatura 3D del volto.
- Lo sviluppo di una tecnica in grado di stimare l'orientamento del volto e applicarla in ambienti di gioco interattivi.

1.3 Cenni Storici

Negli anni sono stati sviluppati diversi approcci al riconoscimento facciale. Uno dei più famosi è conosciuto anche come "Metodo delle autofacce", chiamato Eigenfaces [6] e sviluppato nel 1991 da Matthew Turk del Computer Science Department dell'Università della California e da Alex Pentland del Mit Media Laboratory. Questo algoritmo fa uso dell'analisi delle componenti principali (PCA) [7] per ridurre la dimensione dei dati da

analizzare, rimuovendo l'informazione non necessaria ed estraendo un numero arbitrario di feature che risulta minore rispetto alla dimensione della foto stessa. In questa tecnica le osservazioni sono viste come set di immagini di facce prese sotto le stesse condizioni di luce, normalizzate per allineare occhi e bocca e campionate alla stessa risoluzione. Ogni Eigenface descrive una certa caratteristica, come la linea dei capelli, la simmetria, la larghezza del viso o i contorni di quelle componenti non distinguibili. Un altro approccio, chiamato Fisherfaces [8], è stato presentato nel 1997 da Belhumeur, Hespanha e Kriegman della Yale University. Questo algoritmo è stato sviluppato in modo da essere insensibile a grandi variazioni di illuminazione e espressione facciale. Il metodo Fisherfaces si basa sulla riduzione della dimensione dello spazio facciale utilizzando il metodo PCA che insieme al metodo Fisher Discliminant Linear (FDL) [9], permette di trovare una proiezione lineare delle facce dallo spazio delle immagini. Il metodo Laplacianfaces [10] è stato sviluppato di recente per il riconoscimento automatico dei volti. Si tratta di una generalizzazione dell'algoritmo LLE (Locally Linear Embedding) [11] che aveva già dimostrato di riuscire a controllare in maniera efficace la non linearità dello spazio immagine, riducendone le dimensioni. Oltre alle tecniche descritte per il riconoscimento facciale bidimensionale, nel 2005 è stato introdotto dai ricercatori A. Bronstein, M. Bronstein e R. Kimmel il riconoscimento facciale in 3D che garantisce una minore sensibilità alle condizioni di illuminazione, una minore indipendenza della posa e informazioni geometriche sulle caratteristiche fondamentali del volto. Ad oggi l'hardware per l'acquisizione 3D risulta costoso, difficile da sostituire con quello per il riconoscimento bidimensionale ed è soprattutto ancora oggetto di grande ricerca. Con l'avvento del Machine Learning e Deep-Learning, negli ultimi anni sono stati compiuti grandi progressi nella progettazione di efficaci sistemi di riconoscimento facciale, grazie soprattutto all'impiego di dataset di immagini di grandi dimensioni, questo per garantire un corretto addestramento della macchina e una migliore accuratezza dei risultati. Con tale approccio, Nel 2018, i ricercatori dell'Army Research Laboratory degli Stati Uniti (ARL) [12] hanno sviluppato una tecnica che avrebbe permesso loro di abbinare le immagini del viso, utilizzando una telecamera termica, con quelli nei database che sono stati catturati con una fotocamera tradizionale. Spesso notiamo che tali tecnologie vengono soprattutto applicate per la sicurezza ed è per tale motivo che vengono investite grandi quantità di risorse dai governi per lo sviluppo e il miglioramento di quest'ultime. Ad esempio in diversi stati degli USA, l'FBI scansiona i database fotografici delle patenti di guida con tecnologia di riconoscimento facciale senza alcun consenso dei cittadini,

trasformando i database dipartimentali dei veicoli in una vera e propria infrastruttura di sorveglianza. Dal 2017 nella maggior parte degli aeroporti americani la polizia utilizza il riconoscimento facciale [13] per controllare i passeggeri dei voli internazionali, scattando foto ai singoli passeggeri e confrontandole con quelle dei passaporti e dei visti contenute nel database, creando anche non poche polemiche sull'utilizzo delle informazioni acquisite, soprattutto in ambito privacy. Anche i Social-Media hanno iniziato ad adottare il riconoscimento facciale, attirando l'attenzione di molti utenti. È il caso di Snapchat che risulta una dei primi social network ad utilizzare tale tecnologia. In particolare è stato sviluppato un filtro, chiamato "Face swap", in grado di riconoscere il viso di due utenti e sostituirlo. Lo stesso approccio è utilizzato anche da Instagram, che grazie all'AR (Augmented Reality) permette all'utente di utilizzare filtri in grado di alterare/modificare alcune caratteristiche del volto. L'Azienda di Cupertino, Apple nel 2017 ha sviluppato un innovativo metodo di sblocco tramite riconoscimento facciale, chiamato "Face ID", che fin da subito ha messo in evidenza le sue potenzialità. Face ID consiste in un sensore con un modulo che ha il compito di proiettare una rete di 30.000 punti sulla faccia dell'utente, generando una accurata mappatura facciale 3D, e un altro modulo che legge la mappa creata per confermare o negare l'accesso al dispositivo. Il riconoscimento facciale quindi è volto ad una continua evoluzione, trovando applicazione in realtà che oggi sono agli albori (Self-Driving, Facial Emotion, Object Recognition…) ma che sicuramente cambieranno le abitudini degli utenti con il passare degli anni.

1.4 Struttura della tesi

Questa tesi è così articolata: nel capitolo 2 verrà introdotto l'approccio usato per l'acquisizione, la gestione delle immagini e il riconoscimento del volto. Successivamente, nel capitolo 3 verrà introdotta l'implementazione delle features sviluppate, l'estrazione delle coordinate e la comunicazione con il server tcp. Nel capitolo 4 verrà introdotta la tecnica utilizzata per la mappatura 3D del volto riconosciuto, la feature che permette di determinarne l'orientamento e la sua integrazione in progetti caratterizzati da ambientazioni 3D e videogames. Nel capitolo 5 verranno esposte le conclusioni di questo lavoro di tesi e i possibili sviluppi futuri. Infine nel capitolo 6 verrà spiegato l'iter necessario per lo sviluppo della tesi, illustrando le tecnologie e gli ambienti di sviluppo utilizzati.

Capitolo 2

Preparazione dell'immagine

In questo capitolo verrà presentato l'approccio utilizzato per la preparazione delle immagini, in modo da essere pronte per il riconoscimento del volto. La preparazione delle immagini si divide in:

- Acquisizione delle Immagini
- Riconoscimento del volto

L'obiettivo di questa parte della tesi è stata la creazione di un algoritmo ad hoc per l'acquisizione in sequenza delle immagini da webcam, salvandole in uno specifico path e dandole in input alla rete FAN [14]. L'utilizzo di tale approccio è fondamentale per l'acquisizione delle informazioni necessarie per la rappresentazione dei grafici 2D e 3D e per la mappatura del viso.

2.1 Ambiente di sviluppo

Tale sistema è stato sviluppato in ambiente Windows 10 ed è stato scritto in Python. Per la creazione dell'ambiente di sviluppo e relativa gestione delle librerie è stato utilizzato Anaconda Navigator, da cui sono state installate e sfruttate le seguenti librerie OpenSource:

- OpenCV
- Skimage
- Matplotlib

La libreria OpenCV è stata principalmente utilizzata per l'apertura della fotocamera, l'acquisizione delle immagini e la rilevazione del volto. La libreria Skimage è stata utilizzata per l'elaborazione delle immagini acquisite, mentre la libreria Matplotlib è stata sfruttata per la visualizzazione dei grafici 2D e 3D generati dall'immagine e per ottenere le informazioni fondamentali per la mappatura 3D del volto.

2.2 Acquisizione delle immagini

Il primo passo attuato per lo sviluppo di tale sistema è stata l'acquisizione di una sequenza di immagini. Inizialmente si è partito da un'analisi approfondita del codice, sviluppato da Adrian Bulat [12], ricercatore di intelligenza artificiale presso l'Università di Nottingham, il quale ha realizzato un sistema per l'addestramento della rete FAN applicata a esperimenti di addestramento e allineamento facciale 2D e 3D. Costruita sulla base di un'avanzata architettura per la stima della posa, chiamata "HourGlass" (HG), la rete prende in input l'immagine RGB, ottiene i punti di riferimento 2D e genera le corrispondenti proiezioni 2D e 3D, passandole successivamente a una seconda rete, responsabile del riconoscimento vero e proprio, chiamata "Facial Recognition Network" (FRN). Nel seguente snippet di codice vediamo la procedura di acquisizione dell'immagine:

```
try:
   input_img = io.imread('../test/assets/aflw-test.jpg')
except FileNotFoundError:
   input_img = io.imread('test/assets/aflw-test.jpg')
```

Codice 2.1 Acquisizione di un' immagine

Inizialmente attraverso questo snippet si dava in input alla rete FAN il path di una singola immagine. Di seguito è mostrato l'algoritmo per l'acquisizione di più immagini:

```
cam = cv2.VideoCapture(0)
count = 0
ListImage = []
while True:
   ret, img = cam.read()
   cv2.imshow("ImageCapture", img)
   if not ret:
      break
   k=cv2.waitKey(2)
   if k%256 == 27:
      break
   else:
      print("Image "+str(count)+" saved...")
      path='../test/assets/'+str(count)+'.jpg'
      cv2.imwrite(path, img)
      ListImage.append(path)
      time.sleep(1)
      count +=1
cam.release()
cv2.destroyAllWindows()
```

Codice 2.2 Acquisizione di una sequenza di immagini

L’idea del sistema da me sviluppato si basa su un’acquisizione di più immagini simultaneamente, principalmente per poter garantire una rappresentazione real-time del movimento della mappatura 3D che si andrà a creare. È stato quindi implementato un semplice algoritmo che tramite l’utilizzo della libreria OpenCV, viene aperto il frame relativo alla fotocamera per l’acquisizione delle immagini con un intervallo di tempo per ognuna impostato ad un secondo, principalmente per permettere all’utente di poter uscire in qualsiasi momento dalla procedura. Nel momento in cui si ritiene conclusa la procedura, è stata aggiunta la possibilità di fare in modo che cliccando sul tasto “Esc”, l’utente concluda l’operazione di acquisizione. Terminando la procedura, l’immagine o sequenza di immagini acquisite vengono salvate in un path specifico e i nomi delle immagini acquisite sono inseriti in un array, incrementando di volta in volta una variabile contatore. Vediamo ora nel dettaglio dal seguente codice l’algoritmo implementato:

```
l = 0
while l<len(ListImage):
   try:
      print("Avvio Riconoscimento Volto n."+str(l))
      input_img = io.imread(ListImage[l])
   except FileNotFoundError:
      input_img = io.imread('../test/assets/aflw-test.jpg')
```

Codice 2.3 Elaborazione dell’immagine

La variabile contatore viene utilizzata per scorrere l’array, in modo tale da passare singolarmente ciascun path alla rete FAN, la quale procederà al riconoscimento e all’allineamento facciale. Nel caso in cui la rete non rilevi la presenza di un volto, è stata data come eccezione un’immagine di test, mostrando all’utente da terminale un messaggio di mancato riconoscimento facciale.

2.3 Riconoscimento del volto

Una volta acquisite le immagini si è proceduto al riconoscimento del volto, ottenuto sfruttando la rete FAN (Face-Alignment Network), in grado di ricavare le proiezioni 2D e 3D dei punti di riferimento facciali.

2.3.1 Face-Alignment Network

In tale sistema è stata impiegata la “Face-Alignment Network” (FAN). Sviluppata da Adrian Bulat, FAN è una rete neurale all’avanguardia per la localizzazione dei punti di riferimento del volto, pre-addestrata per l'allineamento di volti in 2D e 3D e valutata su centinaia di migliaia di immagini. In questo lavoro è stata proposta una migliore rappresentazione delle caratteristiche per il riconoscimento dei volti, integrando nel processo di riconoscimento le informazioni relative a posa, espressione e forma del viso, necessarie per migliorare la precisione del riconoscimento. Tale rete si è mostrata fin da subito affidabile e con un basso margine di errori, garantendo una considerevole capacità di resistenza a posa, risoluzione e illuminazione. I test eseguiti da Adrian Bulat su set di dati indipendenti, rispettivamente per il 2D, Dal 2D al 3D e il 3D, hanno mostrato ottimi risultati. Lo sviluppo del progetto non ha previsto un addestramento della macchina utilizzata, poiché la rete risulta pre-addestrata e quindi in grado di poter effettuare senza alcun problema il riconoscimento e l’allineamento dei punti facciali. Una volta ricevuta un’immagine RGB in input, viene riconosciuto un numero fissato di punti per ogni area del viso in questione.

```
preds = fa.get_landmarks(input_img)[-1]
# 2D-Plot
plot_style = dict(marker='o',
markersize=4,
linestyle='-',
lw=2)
pred_type = collections.namedtuple('prediction_type', ['slice', 'color'])
pred_types = {'face': pred_type(slice(0, 17), (0.682, 0.780, 0.909, 0.5)),
'eyebrow1': pred_type(slice(17, 22), (1.0, 0.498, 0.055, 0.4)),
'eyebrow2': pred_type(slice(22, 27), (1.0, 0.498, 0.055, 0.4)),
'nose': pred_type(slice(27, 31), (0.345, 0.239, 0.443, 0.4)),
'nostril': pred_type(slice(31, 36), (0.345, 0.239, 0.443, 0.4)),
'eye1': pred_type(slice(36, 42), (0.596, 0.875, 0.541, 0.3)),
'eye2': pred_type(slice(42, 48), (0.596, 0.875, 0.541, 0.3)),
'lips': pred_type(slice(48, 60), (0.596, 0.875, 0.541, 0.3)),
'teeth': pred_type(slice(60, 68), (0.596, 0.875, 0.541, 0.4))
}
```

Codice 2.4 Suddivisione dei punti facciali

Le informazioni ottenute da FAN sono contenute all'interno della variabile preds. La variabile "preds" è un array bidimensionale, caratterizzato dai punti di riferimento facciali individuati dalla rete, in particolare al suo interno sono contenute le coordinate di ogni punto riconosciuto. Dal codice 2.4 possiamo vedere che i punti riconosciuti sono organizzati secondo una struttura ben definita, rappresentata da pred_type. Pred_type è una struttura che permette di organizzare i punti facciali suddividendoli in 9 aree specifiche del volto, tra cui: faccia (il contorno del viso), sopracciglio sinistro, sopracciglio destro, naso, narici, occhio sinistro, occhio destro, labbra e denti. Ottenuti i punti di riferimento 2D, il compito di FAN è quello di convertirli in 3D, ovvero creare delle proiezioni a partire dai punti di riferimento facciali 2D. Per realizzare tale obiettivo è stata introdotta un'estensione della rete FAN dal 2D al 3D in grado di poter stimare la coordinata z (ovvero la profondità di ciascun punto). Una volta ottenute anche tali informazioni, il sistema procede con la realizzazione di grafici 2D e 3D applicati all'immagine in input. Di seguito vengono mostrati alcuni esempi di grafici 2D ottenuti dal riconoscimento.

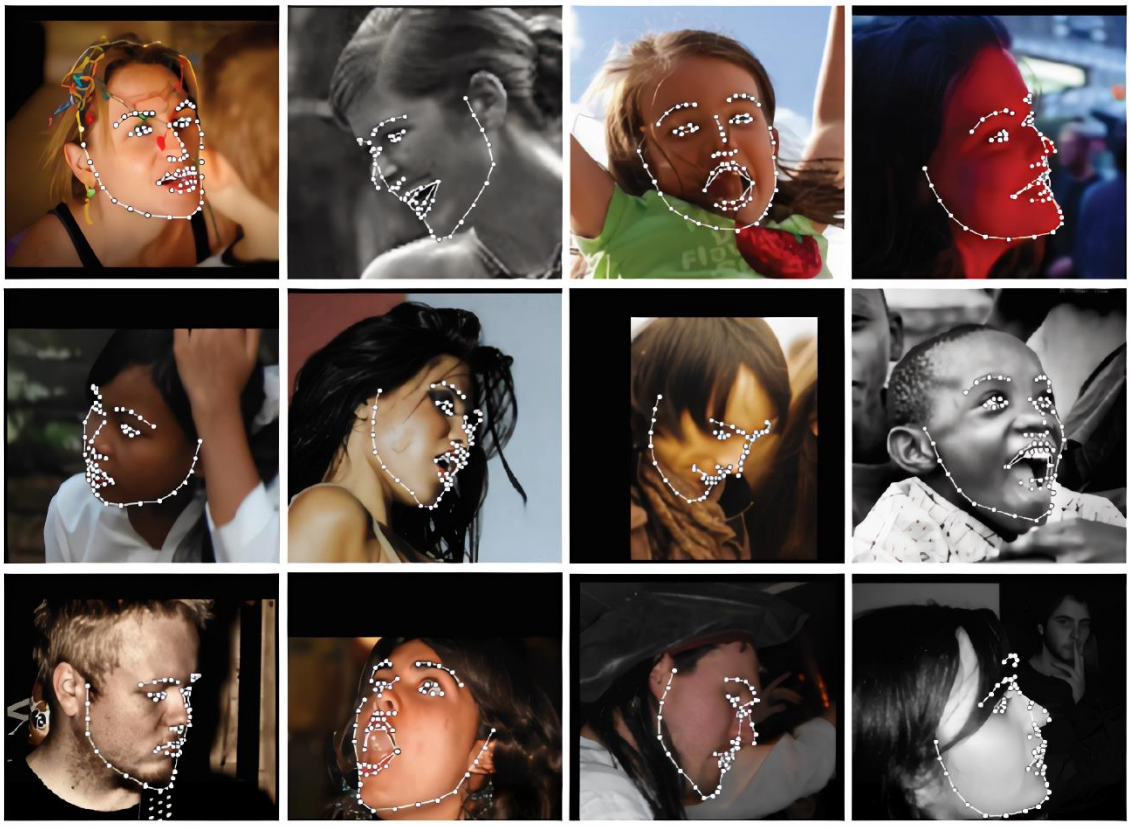

Figura 2.3 Grafici 2D ottenuti dal riconoscimento facciale

Capitolo 3

Implementazione delle features

Nei capitoli precedenti sono state esaminate le tecniche utilizzate per l'acquisizione delle immagini, il relativo riconoscimento e allineamento facciale. Il risultato ottenuto dalle tecniche utilizzate ha permesso la creazione di grafici 2D e 3D del volto riconosciuto e l'estrazione di informazioni che verranno successivamente manipolate per la creazione della mappatura 3D del volto in Unity.

3.1 Rappresentazione del volto riconosciuto

Una volta effettuate le operazioni iniziali di acquisizione e riconoscimento, il sistema prevede la visualizzazione a schermo del risultato ottenuto, caratterizzato da una finestra contenente l'immagine 2D presa in input, il grafico 2D presente sul volto riconosciuto, e il grafico 3D.

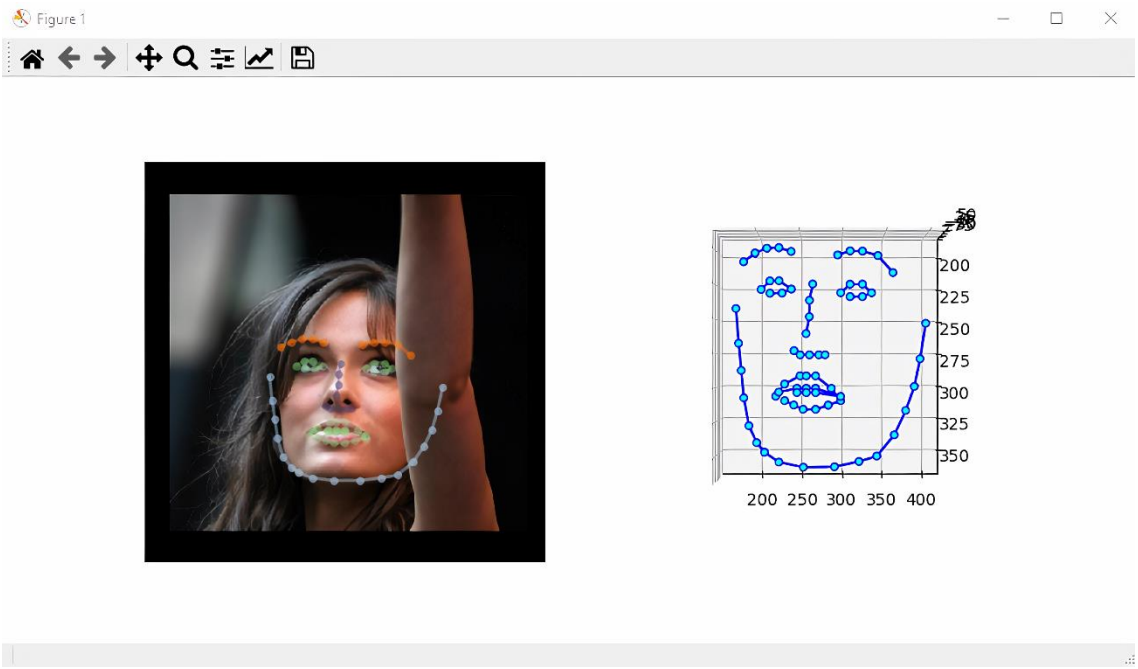

Figura 3.1 Rappresentazione grafici del volto

La figura 3.1 mostra il risultato finale dell'utilizzo di FAN e FRN. Il volto viene riconosciuto, strutturato in landmark e generati due grafici, rispettivamente un grafico 2D applicato sul viso dell'utente e un grafico 3D separato per permettere all'utente di visionarne la profondità. Tale rappresentazione è ottenuta sfruttando la libreria Matplotlib, in particolare il modulo pyplot e il toolkit Axes3D. Matplotlib è una libreria per il tracciamento di grafici a partire dal tipo di informazioni date in input, quindi particolarmente adatta per applicazioni di calcolo scientifico, inoltre utilizza il modulo Numpy per ottimizzare le prestazioni in caso di elaborazione di dati ad elevata dimensionalità. Il suo impiego ha permesso di capire in maniera approfondita l'utilizzo delle coordinate dei landmarks del volto, facilitando lo sviluppo del sistema. Per la rappresentazione dei grafici si procede fornendo il seguente codice:

```
fig = plt.figure(figsize=plt.figaspect(.5))
ax = fig.add_subplot(1, 2, 1)
ax.imshow(input_img)

for pred_type in pred_types.values():
ax.plot(preds[pred_type.slice, 0],
preds[pred_type.slice, 1],
color=pred_type.color, **plot_style)

ax.axis('off')
```

Codice 3.1 Rappresentazione Grafico 2D

Il codice 2.4 mette in evidenza la variabile "plot_style", ovvero un dizionario contenente le proprietà di visualizzazione dei landmark, tra cui: forma, dimensione, stile di linea e spessore. Come mostrato nel codice 3.1, la rappresentazione del grafico 2D ha previsto l'utilizzo di "plt.figure", una funzione che calcola la larghezza e l'altezza della figura con un rapporto d'aspetto specificato, nel nostro caso impostato di default a 0.5. Prima della visualizzazione è stata aggiunta una sottotrama all'immagine di input attraverso la funzione add_subplot e sono stati presi in considerazione i valori contenuti in pred_types.values(), inoltre utilizzando la funzione ax.plot ciò ha permesso la creazione del grafico 2D su immagine, disattivando gli assi x e y del grafico ottenuto con ax.axis('off'). Per la rappresentazione del grafico 3D si è proceduto applicando la stessa tecnica utilizzata per il 2D.

```
# 3D-Plot
ax = fig.add_subplot(1, 2, 2, projection='3d')
surf = ax.scatter(preds[:, 0] * 1.2,
                  preds[:, 1],
                  preds[:, 2],
                  c='cyan',
                  alpha=1.0,
                  edgecolor='b')
for pred_type in pred_types.values():
    ax.plot3D(preds[pred_type.slice, 0] * 1.2,
              preds[pred_type.slice, 1],
              preds[pred_type.slice, 2], color='blue')
ax.view_init(elev=90., azim=90.)
ax.set_xlim(ax.get_xlim()[::-1])
plt.show()
```

Codice 3.2 Rappresentazione Grafico 3D

Come mostrato nel codice 3.2, anche in questo caso viene usata la variabile ax per l'aggiunta della sottotrama, con la differenza che si prende in considerazione la proiezione 3D del grafico creato. I landmarks del volto sono collegati tra loro grazie alla funzione ax.plot3D che permette di organizzarli per zona del volto grazie anche al metodo slice, ovvero un potente metodo per la suddivisione di elenchi, stringhe e tuple. L'impiego di slice() ha permesso di gestire i diversi punti di riferimento del volto, i quali vengono collegati ciascuno da un arco in modo da mostrarne la zona interessata e quindi creare il grafico 3D, visualizzato attraverso l'uso della funzione plt.show().

3.1.1 Imprecisioni riconoscimento

Il sistema sviluppato mostra una notevole precisione nel riconoscimento del volto, tuttavia si sono presentati casi in cui il volto non è stato riconosciuto in modo adeguato. Inizialmente il lavoro era focalizzato sull'elaborazione di una singola immagine, solitamente passata in input con una buona risoluzione. Introducendo il real-time e acquisendo una sequenza di immagini scattate da webcam, in alcuni casi questo ha causato situazioni in cui il volto non è stato riconosciuto correttamente. Sfruttando la webcam di un pc possono esserci situazioni che alterano l'immagine acquisita, come la risoluzione e l'illuminazione. La rete neurale FAN è in grado di riconoscere l'utente entro

una certa distanza, infatti è stato provato che nel caso in cui l'utente si allontani dalla webcam di circa due metri, può capitare che non venga più preso in considerazione il volto, riconoscendo eventualmente un oggetto e di conseguenza creando grafici 2D e 3D errati. Una possibile soluzione è quella di posizionarsi ad una distanza ravvicinata, permettendo al dispositivo di rilevare senza difficoltà il solo volto, oppure addestrare la rete neurale anche a tali particolari condizioni di posa. Questo sistema, pur utilizzando una webcam ha ottenuto ottimi risultati, riuscendo nella maggior parte dei casi a riconoscere e rappresentare graficamente il volto, testandolo in diverse condizioni che possono indurre a problemi comuni come: posa del volto, particolari condizioni di luminosità e distanza dalla webcam. Di seguito vengono riportati alcuni risultati ottenuti dai test effettuati.

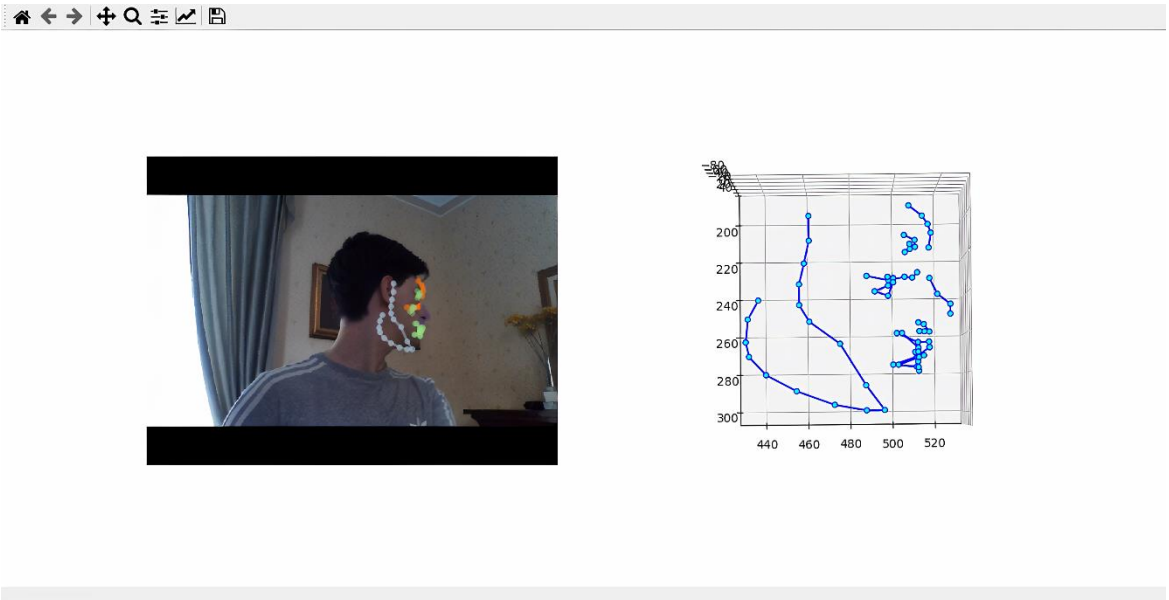

Figura 3.2 Riconoscimento volto di profilo

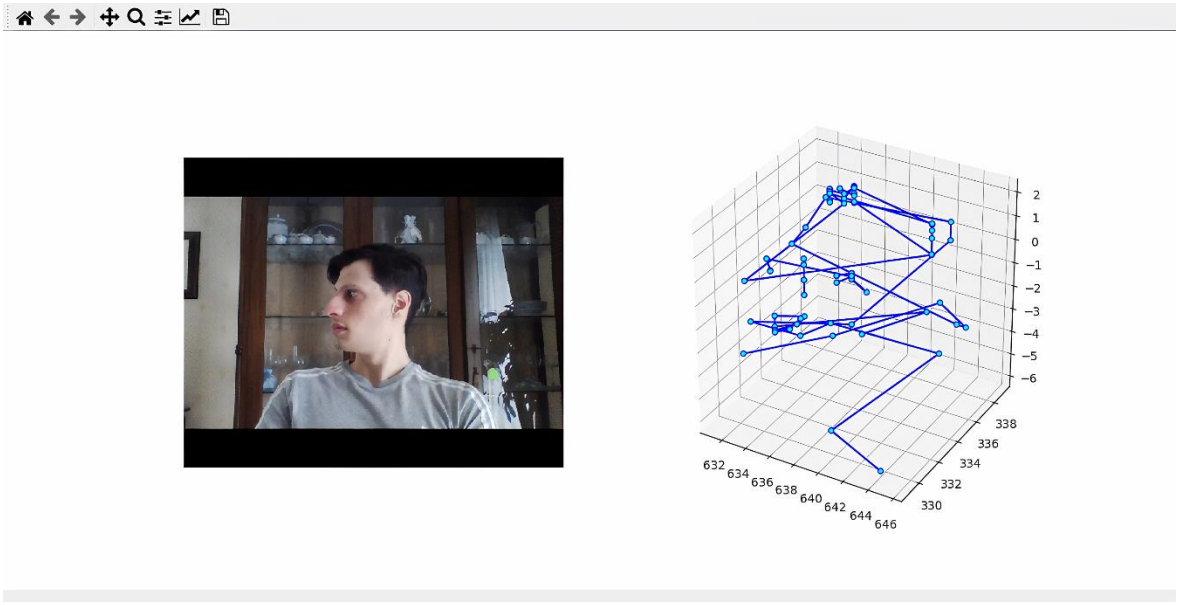

Figura 3.3 Errore riconoscimento volto

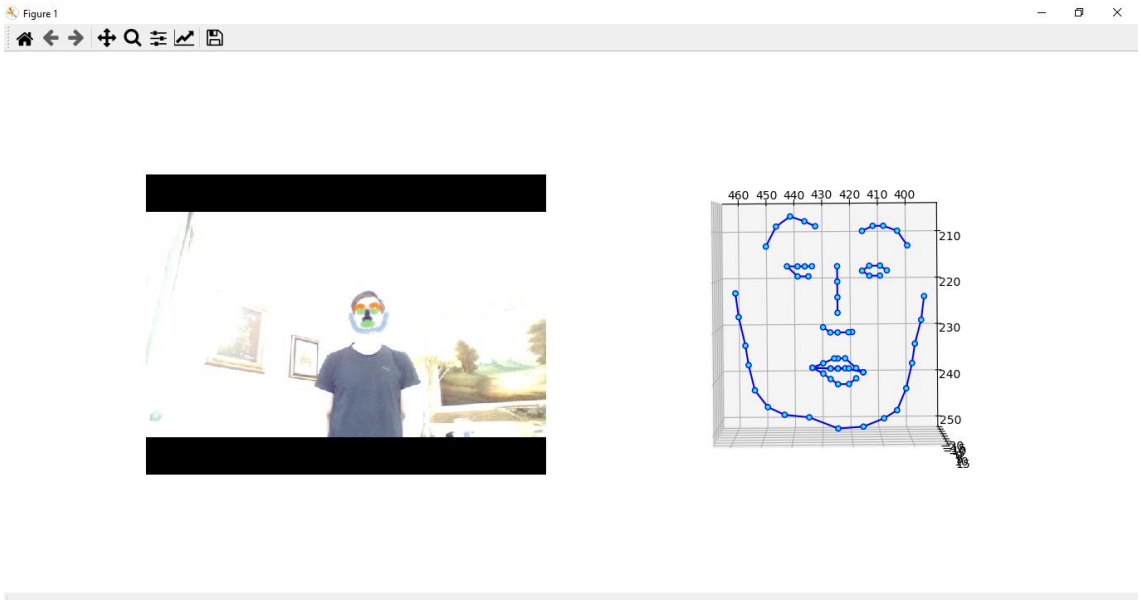

Figura 3.4 Rappresentazione frontale con particolare condizione di luce

3.2 Manipolazione dei landmarks

Il processo di riconoscimento del volto avviene quindi in due passi principali: per prima cosa, data l'immagine acquisita, ne vengono riconosciuti i landmarks tramite l'utilizzo della rete FAN. Successivamente tali landmarks vengono utilizzati per la realizzazione dei grafici 2D e 3D. Per la creazione di una mappatura 3D del volto è necessario gestire le coordinate dei landmarks, cioè convertirle in uno specifico formato per poi essere inviate al server tcp, che riceverà tali coordinate, verranno decodificate e utilizzate per istanziare i vari landmarks in Unity, ottenendo così una mesh 3D del volto riconosciuto.

3.2.1 Estrazione delle coordinate

Una volta ottenuti i landmarks dal volto riconosciuto, è necessario estrarre le coordinate di ciascuno. Analizzando la variabile preds, in particolare dal codice 2.4 risulta che preds è un'array bidimensionale in cui sono contenute le coordinate tridimensionali (x, y, z) di ciascun landmark, per un totale di 68 landmarks. Tale risultato è ottenuto utilizzando la funzione "get_landmarks(self, image_or_path, detected_faces=None)" contenuta nel file api.py, dove al suo interno sono presenti tutte le funzioni necessarie per la rilevazione e il riconoscimento dei landmarks. Dal codice 3.3 possiamo vedere che tale funzione ha come risultato il valore ottenuto dalla funzione get_landmarks_from_image(image_or_path, detected_faces=None):

```
def get_landmarks(self, image_or_path, detected_faces=None):
    """Deprecated, please use get_landmarks_from_image

    Arguments:
        image_or_path {string or numpy.array or torch.tensor} -- The input image or
path to it.

    Keyword Arguments:
        detected_faces {list of numpy.array} -- list of bounding boxes, one for each
face found
        in the image (default: {None})
    """
    return self.get_landmarks_from_image(image_or_path, detected_faces=None)
```

Codice 3.3 Funzione riconoscimento landmarks

Stampando la variabile preds otteniamo un array del tipo:

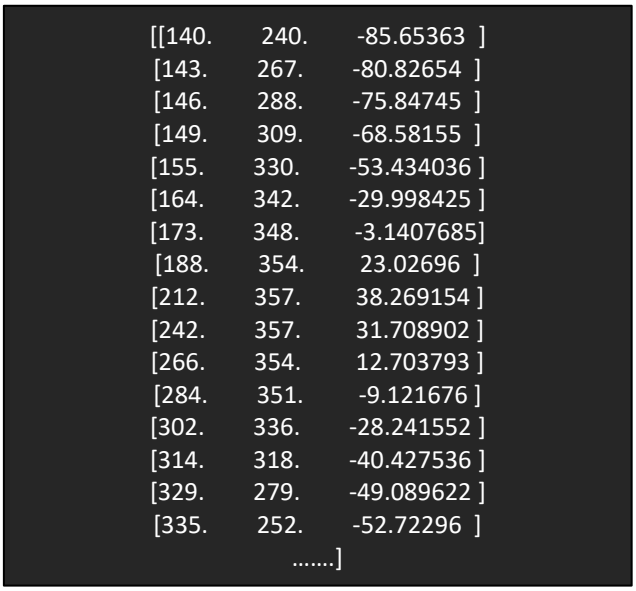

Figura 3.5 Stampa della variabile preds

Si nota fin da subito che preds presenta un formato specifico in cui ogni array di coordinate è separato dagli altri dal carattere '\n'. Tale array verrà successivamente manipolato per ottenere le coordinate in un formato valido da mandare al server tcp in Unity.

3.2.2 Comunicazione server tcp

Per poter mappare il volto in Unity è necessario l'utilizzo di un server tcp per la ricezione delle coordinate dei landmarks inviate lato pyton. In Unity il server è stato sviluppato integrando "slessans" [15], che consente ai client di connettersi al server e inviare token, cioè stringhe contenenti le coordinate di ciascun landmark seguite da un separatore "&". Per la creazione del server è stato utilizzato un gameObject persistente (sfera) a cui è stato allegato lo script "SCL_PositionalControllerInput.cs", che permette di collegarsi ad un client, leggere i 3 valori di posizione (x, y, z) per poi convertirli da formato stringa a float. Dal codice 3.4, a partire dal metodo Start() sono state istanziate la variabile maxClients relativa al massimo numero di connessioni, separatorString relativa al separatore e portNumber relativo al numero di porta. Una volta iniziato l'ascolto, quest'ultimo verrà interrotto al raggiungimento del numero fissato di maxClients.

```
void Start()
{
SCL_IClientSocketHandlerDelegate clientSocketHandlerDelegate = this;
controlledObject = Resources.Load<GameObject>("marker");
int maxClients = 9;
string separatorString = "&";
int portNumber = 13000;
Encoding encoding = Encoding.UTF8;

this.socketServer = new SCL_SocketServer(
clientSocketHandlerDelegate, maxClients, separatorString, portNumber, encoding);

this.socketServer.StartListeningForConnections();

Debug.Log(String.Format(
"Started socket server at {0} on port {1}",
this.socketServer.LocalEndPoint.Address, this.socketServer.PortNumber));
}
```

Codice 3.4 Creazione server

Il metodo ClientSocketHandlerDidReadMessage, chiamato dal server quando riceve i dati da un client, passa il token ricevuto come stringa. La stringa letta viene divisa dal metodo split() in presenza del carattere spazio e viene inserita in un array di tipo stringa. Successivamente viene effettuato un controllo per verificare se la lunghezza dell'array è uguale a 3, questo principalmente per distinguere le coordinate di tipo x, y e z. Se tale condizione è verificata si procede nel convertire ciascun elemento dell'array in float per poi stampare il risultato in console e salvare tali coordinate con this.SetPositionValue (float x, float y, float z). Nel caso in cui il formato non sia valido si rimanda a FormatException, mentre se la lunghezza non è uguale a 3 viene stampato un messaggio di errore lunghezza.

```
public void ClientSocketHandlerDidReadMessage(SCL_ClientSocketHandler handler, string message)
        {
                string[] pieces = message.Split(' ');
                if (pieces.Length == 3)
                {
                        try
                        {
                                float x = float.Parse(pieces[0], CultureInfo.InvariantCulture);
                                float y = float.Parse(pieces[1], CultureInfo.InvariantCulture);
                                float z = float.Parse(pieces[2], CultureInfo.InvariantCulture);

                                Debug.Log(String.Format("x: {0}, y: {1}, z: {2}", x, y, z));
                                this.SetPositionValue(x, y, z);
                        }
                        catch (FormatException)
                        {
                                Debug.Log("Invalid number format: " + message);
                        } }
```

Codice 3.5 Conversione degli elementi dell'array

L’invio delle coordinate al server tcp in Unity è stato gestito in python mediante l’utilizzo della libreria socket. Una volta completato l’applicativo server lato Unity, si è proceduto con la creazione del client.

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
   s.connect(('127.0.0.1', 13000))
   i = 0
   while i < len(preds):
      string = str(preds[i])+'&'
      msg = bytes((string), 'utf-8')
      s.sendall(msg)
      time.sleep(0.2)
      i = i + 1
```

Codice 3.6 Invio coordinate lato client

Dal codice 3.6, socket.socket è stato utilizzato per creare un oggetto socket, mentre gli argomenti passati specificano la famiglia di indirizzi, ovvero IPv4, e il tipo di socket. Il client chiama connect(“127.0.0.1”,13000) per stabilire una connessione al server attraverso un host e una porta unica. In questo caso solo il client può raggiungere il server, e non viceversa. Una volta collegato il client al server, si è gestito l’array preds attraverso l’uso di una variabile contatore e un ciclo che prendendo in input la lunghezza dell’array, viene scorso convertendo le coordinate di ciascun landmark in formato stringa e tramite s.sandall() inviato come oggetto byte al server . Tale operazione si ripete per ogni landmark riconosciuto, con un intervallo di invio impostato a 0.2 secondi. Di seguito è riportato il risultato ottenuto su console Unity:

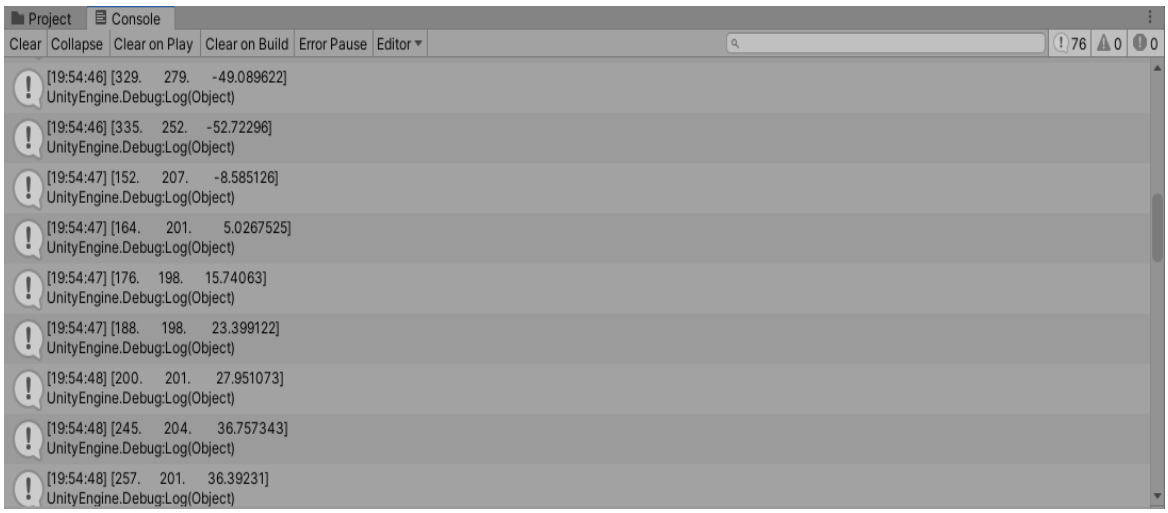

Figura 3.6 Coordinate ricevute su console

3.2.3 Codifica e decodifica delle coordinate

Per ogni token ricevuto è opportuno stabilirne un formato specifico, in modo che tali coordinate possano essere utilizzate per istanziare ciascun landmark riconosciuto in Unity. Dal codice 3.6 è possibile vedere che le coordinate che vengono inviate al server presentano un formato caratterizzato dalla presenza di caratteri che non ne permettono l'utilizzo. Per rimuovere questi caratteri fondamentale è stato l'utilizzo del modulo re. Il modulo re fornisce operazioni per la corrispondenza delle espressioni regolari. Tali espressioni regolari usano il carattere ('\') per indicare forme speciali o per consentire l'uso di caratteri speciali senza invocarne il significato, inoltre le funzioni in questo modulo consentono di verificare se una determinata stringa corrisponde a una certa espressione regolare. La soluzione proposta prevede l'utilizzo di re.sub(), che a seconda del pattern dato in input permette di sostituire i caratteri non richiesti, in questo caso le parentesi quadre. Di seguito è riportato il codice modificato:

```
while i < len(preds):
    string = re.sub('[^A-Za-z0-9 .\-+\']+', '', str(preds[i]))
    msg = bytes(man_string(string), 'utf-8')
    s.sendall(msg)
    time.sleep(0.02)
    i = i + 1
```

Codice 3.7 Utilizzo di re.sub() per rimozione caratteri

Successivamente è stata implementata la funzione man_string(string), in grado di assegnare a ciascuna stringa un formato valido. La funzione man_string() riceve in input la stringa ottenuta da re.sub(). Tale stringa è concatenata con il carattere "&" affinché il server riconosca il token correttamente. Dalla figura 3.5 si nota soprattutto la presenza di un certo numero di spazi tra le posizioni x, y e z e questo rappresentava un problema per il formato, così si è proceduto effettuando un ulteriore modifica alla stringa richiamando re.sub() per fare in modo che in presenza di una sequenza di spazi, questa venga sostituita da un singolo spazio, permettendo allo script SCL_PositionalControllerInput.cs di identificare correttamente le 3 variabili e convertirle in float. Per evitare la presenza di eventuali errori di formato è stato poi aggiunto un ulteriore controllo, ovvero "string.strip()" per verificare se il token presenti spazi iniziali e finali, in tal caso rimossi. Questa funzione sviluppata permette di inviare al server le coordinate di ciascun landmark

sottoforma di stringa e con un formato valido per la decodifica e conversione in Unity. Di seguito viene riportato il risultato finale utilizzando la funzione man_string(string):

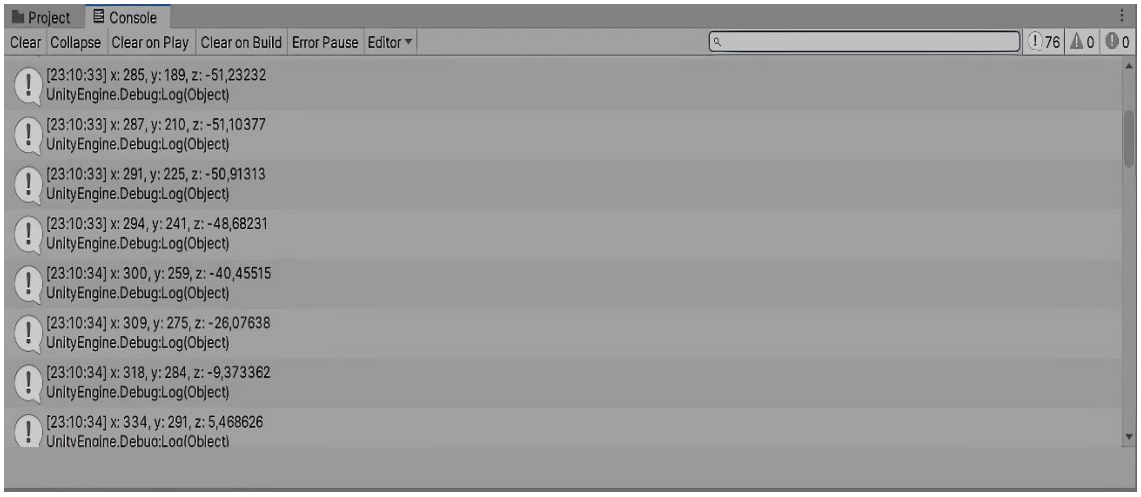

Figura 3.7 Formato valido delle coordinate su console

3.3 Riconoscimento secondo volto

Nei paragrafi precedenti si è visto come ottenere, da un'immagine 2D il riconoscimento di un singolo volto e l'estrazione delle relative informazioni dei landmarks. Lo sviluppo del sistema ha messo in evidenza anche la possibilità di riconoscere un secondo volto. Per effettuare il riconoscimento di un secondo volto si è pensato di apportare alcune modifiche al codice rimanendo pressoché identica la procedura di riconoscimento. La tecnica utilizzata è stata quella di creare un nuovo array, chiamato predt, e fare in modo che le informazioni relative al secondo volto riconosciuto vengano inserite al suo interno, stessa cosa fatta con preds.

```
preds = fa.get_landmarks(input_img)[-1]
predt = fa.get_landmarks(input_img)[0]
```

Codice 3.8 Aggiunta nuovo array di coordinate

Si nota che predt, così come preds, presenta un indice intero posto alla fine. La rete FAN, nel caso in cui ci sia più di un volto presente nell'immagine, grazie alla presenza di questo indice permette di stabilire la priorità del volto da riconoscere. Se alla variabile predt passiamo l'intero -1, ciò indica a FAN di effettuare il riconoscimento del primo volto a distanza ravvicinata, per poi passare a quello successivo. Una volta istanziato predt si è

proceduto con la duplicazione dello stesso codice utilizzato per la rappresentazione grafica.

```
for pred_type in pred_types.values():
   ax.plot(preds[pred_type.slice, 0],
         preds[pred_type.slice, 1],
         color=pred_type.color, **plot_style)

for pred_type in pred_types.values():
   ax.plot(predt[pred_type.slice, 0],
         predt[pred_type.slice, 1],
         color=pred_type.color, **plot_style)

ax.axis('off')
```

Codice 3.8.1 Duplicazione codice rappresentazione grafico 2D del secondo volto

Il codice 3.8.1 mostra la duplicazione del ciclo necessario per la rappresentazione grafica 2D, ottenendo così come sul primo, anche il grafico 2D su immagine del secondo volto riconosciuto. Per la rappresentazione 3D è stata svolta la stessa operazione di duplicazione di codice come nella rappresentazione del grafico 2D, in particolare dal codice 3.8.2 si può notare l'aggiunta della variabile surg, che allo stesso modo di surf permette la creazione di un grafico a dispersione utilizzando in questo caso l'array predt. Per la rappresentazione finale del grafico 3D è stata fatta un'ulteriore duplicazione di codice necessaria per la rappresentazione dei relativi landmarks riconosciuti.

```
for pred_type in pred_types.values():
   ax.plot3D(preds[pred_type.slice, 0] * 1.2,
         preds[pred_type.slice, 1],
         preds[pred_type.slice, 2], color='blue')

for pred_type in pred_types.values():
   ax.plot3D(predt[pred_type.slice, 0] * 1.2,
         predt[pred_type.slice, 1],
         predt[pred_type.slice, 2], color='blue')
```

Codice 3.8.2 Duplicazione codice rappresentazione grafico 3D del secondo volto

Tali modifiche apportate hanno permesso, seppur non ottimizzando il codice e impiegando maggior tempo di elaborazione, un altrettanto affidabile riconoscimento di un secondo volto. Di seguito vengono riportati alcuni risultati ottenuti da tale sviluppo:

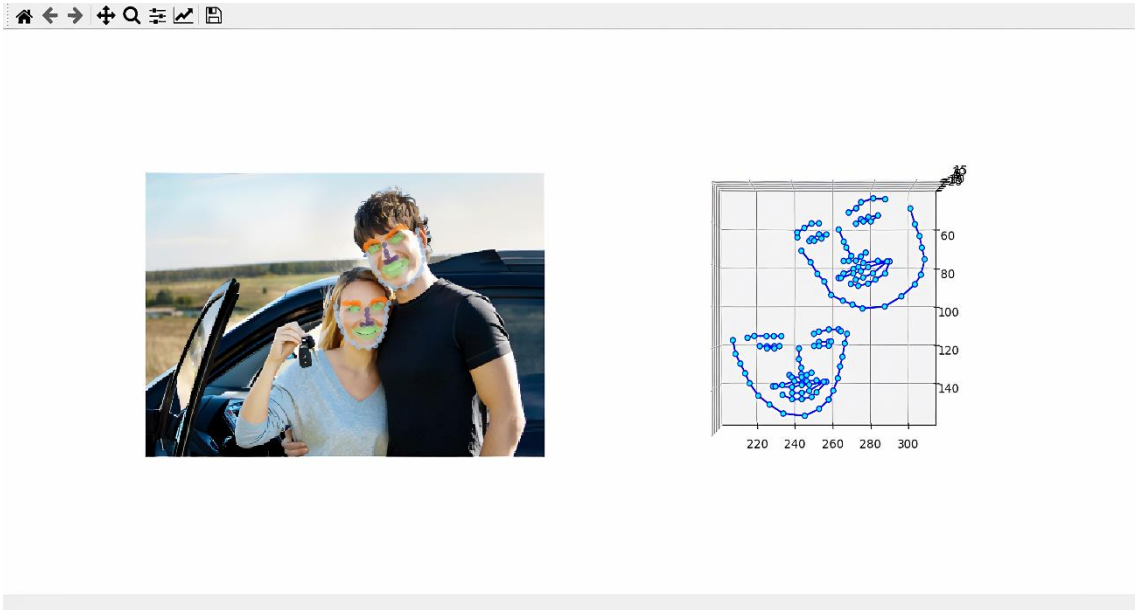

Figura 3.8 Riconoscimento di due volti vicini e rappresentazione grafici

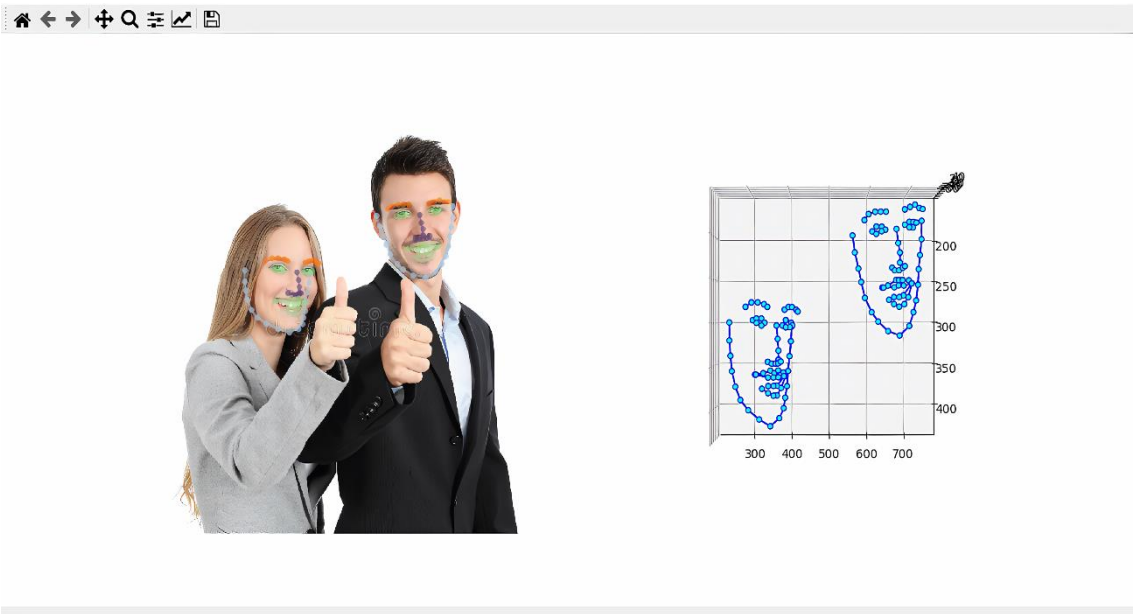

Figura 3.9 Riconoscimento di due volti e rappresentazione grafici

3.3.1 Distanza Euclidea

Ottenuto il riconoscimento di un secondo volto, una feature aggiuntiva è stata la possibilità di confrontare la distanza tra i landmarks dei due volti. Questa feature si basa sull'idea che ogni volto abbia le sue dimensioni e che analizzandole si possano ottenere importanti informazioni per distinguere un volto dall'altro. Oltre ad usare le posizioni tra i punti per realizzare la mappatura 3D, si è pensato di effettuare un calcolo sulle distanze tra coppie di punti appartenenti ai due rispettivi volti. Tale procedura è stata realizzata utilizzando la formula per il calcolo della distanza euclidea [16] tra due punti dei due volti riconosciuti. La distanza è calcolata prendendo in considerazione due punti in tre dimensioni, ovvero dati P = (px, py, pz) e Q = (qx, qy, qz), abbiamo che:

$$\text{Distanza tri-dimensionale} = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}.$$

Per applicare tale formula al listato python si è proceduto sviluppando una funzione apposita. Innanzitutto è stata importata la libreria math, necessaria per calcoli matematici, e istanziata la variabile lenght. Analizzando il listato si è pensato di gestire diversi casi di riconoscimento, ovvero il riconoscimento di un solo volto oppure il riconoscimento di due volti. Nel caso di un solo volto riconosciuto, il sistema procede con la sua esecuzione così come spiegato nei paragrafi precedenti. Nel caso di due volti riconosciuti, viene utilizzata la variabile lenght = len(fa.get_landmarks(input_img)) per determinare il numero di volti presenti nell'immagine. Grazie a lenght è possibile effettuare un controllo per verificare se il suo valore è maggiore di uno e se tale condizione è soddisfatta si procede applicando alla stringa un formato valido per l'invio al server, mentre per ogni coppia di landmarks dei due volti riconosciuti si applica la funzione "Distance".

```
# Calcolo Distanza Tri-dimensionale tra punti di due volti
def Distance(a, b, preds, predt):
    x = pow(preds[a][0] - predt[b][0], 2)
    y = pow(preds[a][1] - predt[b][1], 2)
    z = pow(preds[a][2] - predt[b][2], 2)
    p = x + y + z
    rp = math.sqrt(p)
    if (rp > 0):
        print("Distanza Landmarks n." + str(a) + ": " + str(rp))
```

Codice 3.9 Funzione per calcolo distanza tri-dimensionale

Così come illustrato dalla formula della distanza tri-dimensionale, dal codice 3.9 la funzione Distance prende in considerazione le variabili x,y, z. Sottraendo rispettivamente ciascun elemento di preds con ciascun elemento di predt, per poi elevare il tutto al quadrato, si sommano gli elementi x, y e z ottenuti e si calcola la radice quadrata del risultato finale. Di seguito è riportato il risultato dall'applicazione di tale funzione:

```
Distanza Landmarks n.1: 57.63055828245541
Distanza Landmarks n.2: 57.16896519637128
Distanza Landmarks n.3: 58.374728846543206
Distanza Landmarks n.4: 57.192428703262415
Distanza Landmarks n.5: 60.04944606424076
Distanza Landmarks n.6: 61.033991436272174
Distanza Landmarks n.7: 61.626863082969905
Distanza Landmarks n.8: 63.49813443537879
Distanza Landmarks n.9: 65.89596182749787
Distanza Landmarks n.10: 68.73997907548234
Distanza Landmarks n.11: 70.1652137905419
Distanza Landmarks n.12: 71.84231010005273
Distanza Landmarks n.13: 70.75499121311347
Distanza Landmarks n.14: 70.60122588577043
Distanza Landmarks n.15: 68.30056493502187
Distanza Landmarks n.16: 69.7102488611134
Distanza Landmarks n.17: 53.617211837913764
                    ...
```

Figura 3.10 Distanza tri-dimensionale di ciascun landmark tra due volti

L' applicazione di questa formula ha quindi reso possibile comprendere la distanza tra i 68 landmarks dei due volti riconosciuti.

Capitolo 4

Mappatura 3D del volto

In questa fase i dati provenienti dall'estrapolazione delle coordinate dei landmarks riconosciuti verranno utilizzati per la mappatura 3D finale del volto. Lo scopo di questa fase è principalmente quello di creare in real-time una mappatura 3D per ogni volto riconosciuto per visualizzarne il movimento. Fondamentale è stata la feature relativa alla stima della direzione del volto in quanto integrata poi in altri progetti di tesi.

4.1 Rappresentazione volto in Unity

Una volta estrapolate le coordinate dai landmarks riconosciuti, lo sviluppo del sistema si è poi concentrato sulla loro rappresentazione grafica in Unity, ottenendo lo stesso grafico generato nella GUI di matplotlib. Si e proceduto utilizzando le coordinate di formato x,y,z di ciascun landmark per istanziare un gameObject di tipo sfera. La sfera, chiamata "marker", è un gameObject di tipo prefab. Per poter istanziare correttamente ogni landmark è stato utilizzato lo script CreaOggetti.cs. Innanzitutto sono state istanziate le variabili controllerInput, che permette di comunicare con SCL_PositionalControllerInput per prendere in input le coordinate ricevute e decodificate, e "proiettile" per istanziare il gameObject sfera. Affinchè ogni prefab possa essere istanziato, è stato utilizzato il metodo Update() in quanto essendo richiamato ad ogni frame risulta il più adatto per ricevere in real-time le coordinate. Per garantire la corretta istanziazione di ciascun prefab è stato aggiunto un controllo che evita la creazione di più cloni di un prefab con le stesse coordinate. Tale controllo permette di confrontare, a partire dalla prima coordinata ricevuta, i valori (x,y,z) con il vettore di posizione vec e trasform.position, inizializzati entrambi a 0. Dal codice 4.1 è possibile vedere che il controllo permette di gestire singolarmente ogni coordinata, assegnando alla variabile trasform.position le coordinate comunicate da controllerInput.GetPositionValue() e istanziando così ciascun prefab tramite metodo Istanziate(proiettile, transform.position, transform.rotation). Tale tecnica permette la realizzazione di una singola mappatura 3D del volto riconosciuto. Nel caso di più volti riconosciuti, più mappature 3D vengono create e sovrapposte, quindi non

riuscendo a distinguere un volto dall'altro. Questo problema è stato risolto impiegando una variabile contatore che viene incrementata per ogni coordinata ricevuta. L'idea è stata quella di impiegare un controllo che nel momento in cui il contatore raggiunge il valore 68, cioè il numero di landmarks totali del volto, tutti i prefab con il tag "marker" vengono aggiunti a un array e cancellati immediatamente, inizializzando nuovamente il contatore a 0.

```
   void Update()
   {
      if (controllerInput.GetPositionValue() != vec && transform.position !=
controllerInput.GetPositionValue())
      {
         if (count == 68)
         {
            GameObject[] arr = GameObject.FindGameObjectsWithTag("marker");

            foreach (GameObject marker in arr)
               GameObject.Destroy(marker);

            count = 0;
            stop = false;
         }

         transform.position = controllerInput.GetPositionValue();
         Instantiate(proiettile, transform.position, transform.rotation);
         count++;
         if (count == 40)
         {
            a = transform.position;
         }
         if (count == 43)
         {
            b = transform.position;
         }
         if (count == 52)
         {
            c = transform.position;
         }
```

Codice 4.1 Istanziare prefab da coordinate

Quest'ultima modifica ha permesso di gestire correttamente ogni mappatura 3D creata, mostrando in real-time il movimento del volto riconosciuto. Di seguito viene mostrato il risultato ottenuto da una sequenza di volti riconosciuti e mappati dalle immagini acquisite:

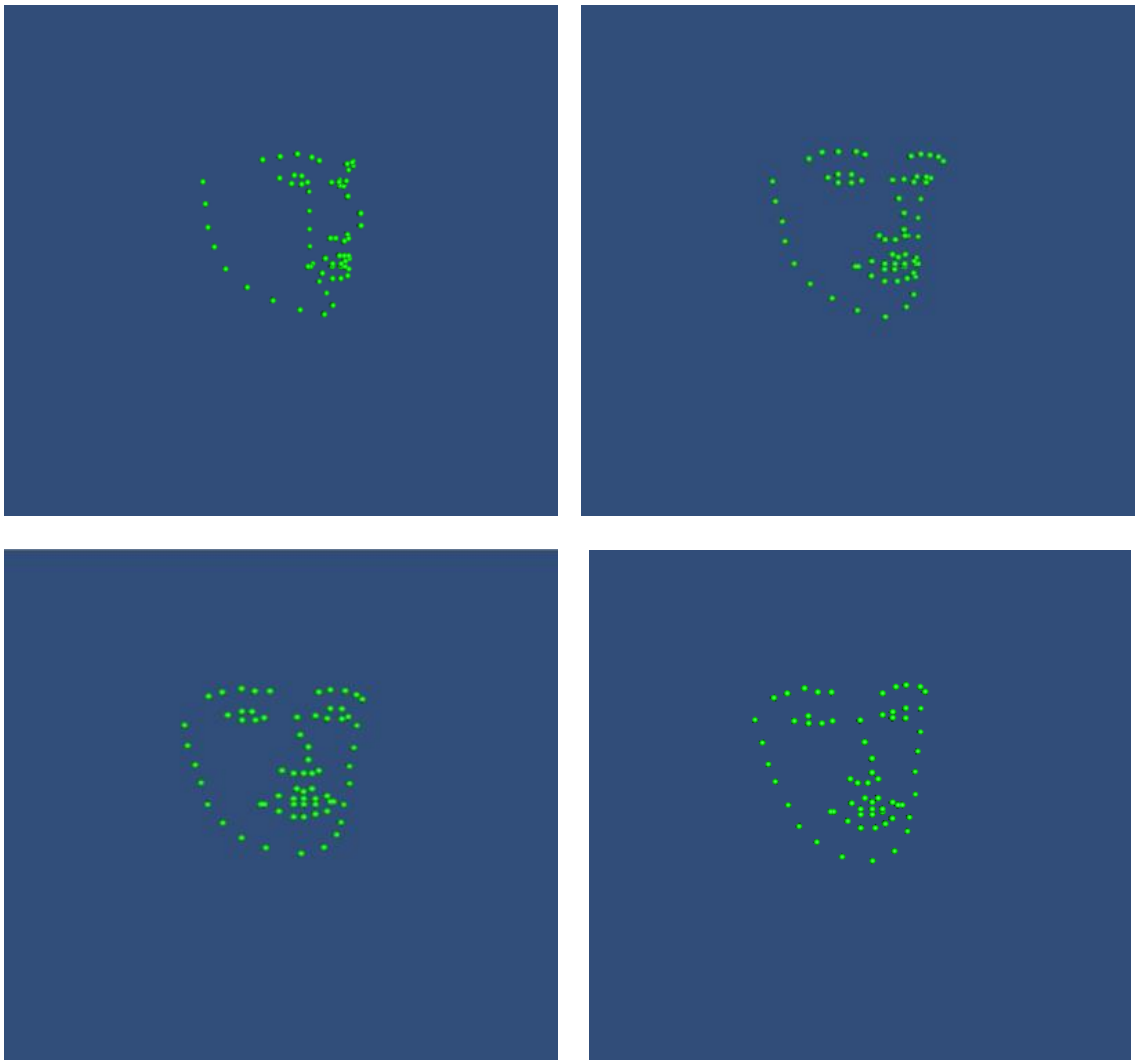

Figura 4.1 Mappatura 3D del volto riconosciuto in Unity

4.2 Orientamento del volto

Ottenuta la mappatura 3D del volto si è pensato di aggiungere come feature aggiuntiva la possibilità di determinarne l'orientamento. Per riconoscere l'orientamento del volto è stato utilizzato lo script rettaPerpendicolare.cs, il quale è stato aggiunto ad un gameObject con mesh filter, necessario per la generazione di un raggio luminoso indicante la direzione. L'orientamento del volto è calcolato prendendo in considerazione le coordinate di tre landmarks specifici del volto: Landmark occhio sinistro, Landmark occhio destro, Landmark labbro superiore. Dato che l'acquisizione delle coordinate è svolta in real-time, per salvare le coordinate dei tre landmarks, fondamentale è stato il ruolo del contatore. Tale contatore in precedenza è stato introdotto in CreaOggetti.cs, come da codice 4.1 per evitare sovrapposizioni di più volti mappati in 3D in Unity, mentre in tal caso si è pensato di riutilizzarlo applicando dei controlli sul valore, per fare in modo che se la condizione sia rispettata allora le coordinate di quel landmark vengono salvate in un nuovo vettore di posizione. Una volta salvato il vettore di posizione di ciascun landmark, lo script rettaPerpendicolare.cs richiama tali vettori per istanziare tre gameObject sfera, necessari per la generazione del raggio. Così come in CreaOggetti.cs, anche in questo script la procedura viene svolta in Update(), in cui viene effettuato un controllo per verificare se tutti i prefab del volto sono stati istanziati correttamente. A questo punto, per poter generare il raggio è stata richiamata la variabile mesh, a cui sono stati passati come vertici i vettori di posizione precedenti e utilizzato il metodo RecalculateNormals() per ricalcolare le normali della mesh da triangoli e vertici. Da questo si è proceduto istanziando la componente Linerender lr, la quale permette di generare una linea retta nello spazio 3D circostante ed è stato aggiunto un materiale affinché sia visibile dalla rappresentazione finale.

```
GameObject raggioVisivo = new GameObject();
        raggioVisivo.tag = "normal";
       // raggioVisivo.transform.position = (pos1+pos2)/2;
        raggioVisivo.AddComponent<LineRenderer>();
        LineRenderer lr = raggioVisivo.GetComponent<LineRenderer>();
       // lr.material = new Material(Shader.Find("Particles/Alpha Blended Premultiply"));
        lr.startColor=Color.cyan;
        lr.startWidth=2.1f;
        lr.endWidth=2.9f;
        lr.SetPosition(0, (c.a + c.b) / 2);
        lr.SetPosition(1, mesh.normals[0]*-200 + transform.position);
```

Codice 4.2 Creazione e posizione del raggio

Come mostrato dal codice 4.2, la posizione iniziale della linea è stata calcolata individuando il punto medio tra i due occhi, mentre la posizione finale è ottenuta sommando alle normali della mesh il vettore di posizione del gameObject in cui lo script è stato aggiunto. Il risultato finale è mostrato come segue:

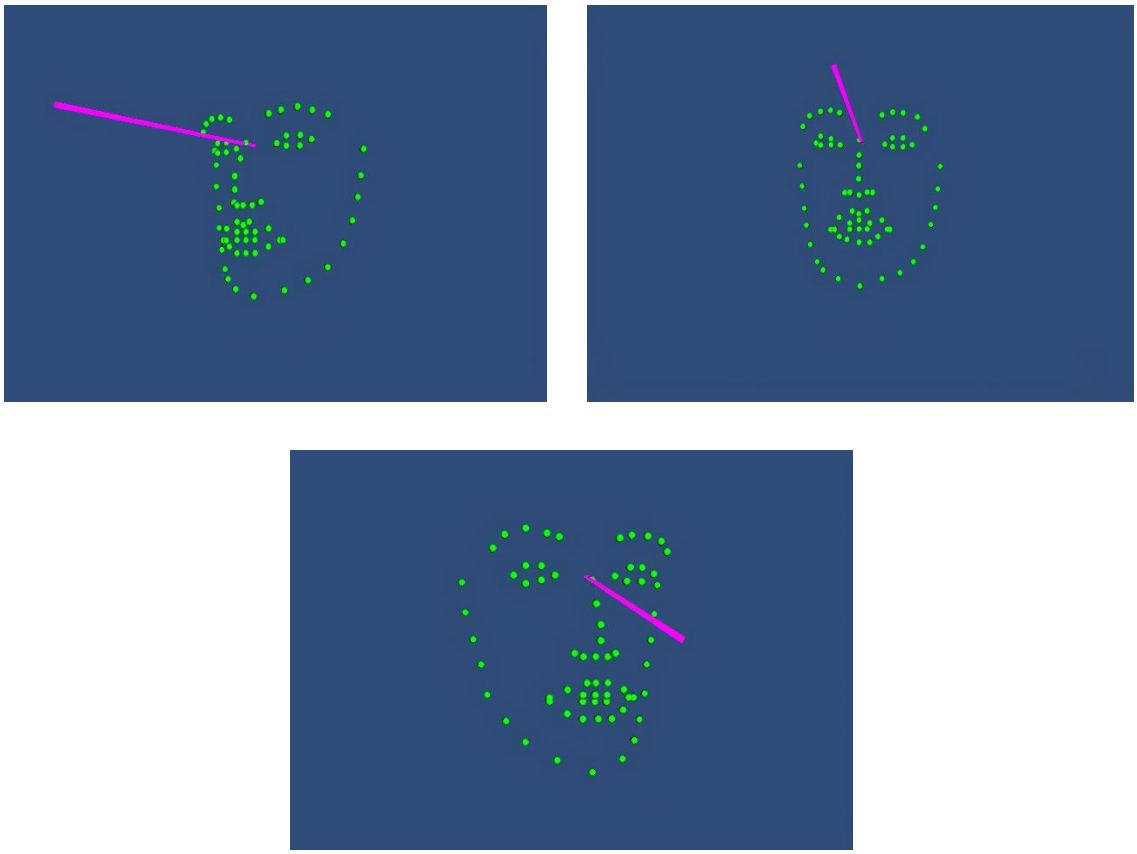

Figura 4.2 Orientamento del volto mappato in 3D

Grazie a questa feature, per ogni volto mappato in Unity viene generata una retta ben visibile che permette di stabilirne l'orientamento in maniera precisa.

4.3 Integrazione in altri progetti

Realizzato tale applicativo si è pensato poi di sfruttarlo in alcuni progetti di tesi sviluppati dai miei colleghi. Trattandosi di giochi e ambientazioni 3D basate sul riconoscimento dell'emozione dell'utente, l'idea è stata quella di poter integrare questo progetto per coinvolgere maggiormente l'utente nella sessione di gioco o nell'ambientazione circostante. Per mettere in pratica tale idea si è proceduto analizzando i modi e le dinamiche con cui l'utente possa interagire maggiormente con ogni applicativo sviluppato, rendendolo divertente e avvincente. Lo sviluppo ha previsto l'integrazione di tale progetto per fare in modo che mentre l'utente sta giocando oppure se si trova in una particolare ambientazione, viene riconosciuto il suo volto, identificato l'orientamento e da questo generati oggetti che vanno a ostacolare/sorprendere l'utente. Lo spawn degli oggetti è stato realizzato in modo che quando l'utente guarda in una direzione, l'oggetto viene generato dalla direzione opposta. Analizzando la direzione del volto e in particolare il raggio generato, si è pensato di dividere l'area visiva dell'utente in tre sezioni: sinistra, centro e destra. Tale tecnica ha previsto l'utilizzo della variabile meshX, contenente le normali necessarie al calcolo della direzione, come visto nel codice 4.2, e sono stati fatti dei controlli sul suo valore, in particolare se compreso tra -20 e 20 indica che l'utente sta guardando frontalmente, mentre se maggiore di 20 l'utente sta guardando a sinistra e se minore di 20 l'utente sta guardando a destra. L'informazione ottenuta è passata al server e comunicata allo script relativo allo spawn degli oggetti. In Update() viene controllata la direzione espressa con "SCL_PositionalControllerInputDirections.direzione" e se il risultato è uguale a left viene richiamato il metodo spawnRight() per la creazione di oggetti a destra, mentre se uguale a "right" viene richiamato il metodo spawnLeft() per la creazione di oggetti a sinistra.

Nel progetto del collega Domenico Rossi, un videogame incentrato sull'Air Hockey, gli oggetti che vengono creati sono pietre, le quali diventano parte attiva del game. Le pietre nel caso in cui l'utente guarda a sinistra o destra vengono fatte cadere sul tavolo da gioco impedendo il passaggio del dischetto e disturbando l'utente nel gioco. Di seguito viene mostrato il risulto finale:

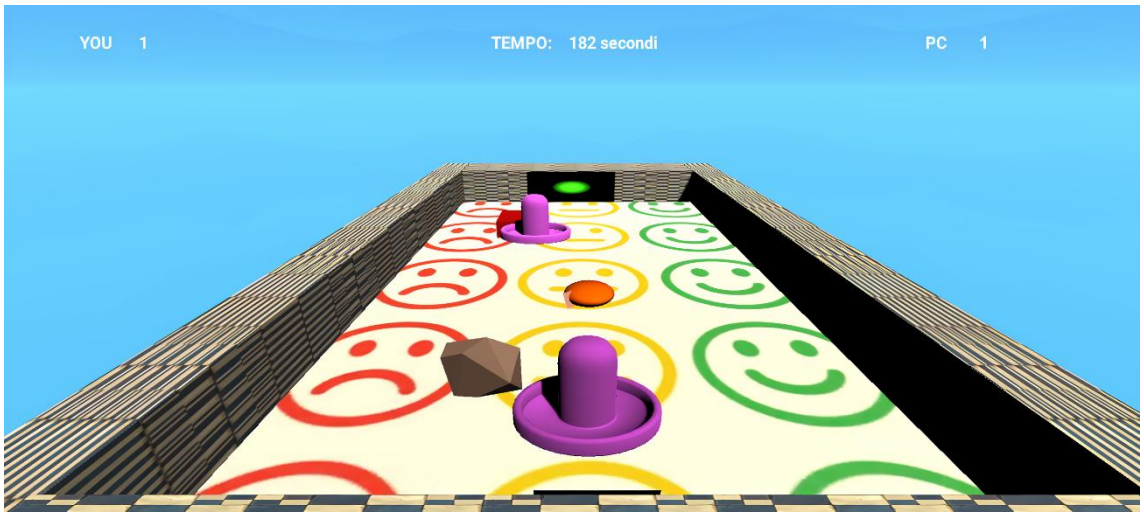

Figura 4.3 Videogame Air Hockey

Nel progetto dei colleghi Andrea Sorrentino, Ciro Maione e Antonio Saporito, la tecnica utilizzata è stata la stessa, ma essendo un applicativo con diverse ambientazioni 3D, la generazione degli oggetti è risultata singolare per ognuno. Nell'ambientazione spazio, per cercare di rendere l'utente sorpreso è stato fatto in modo che gli oggetti creati fossero caratteristici, generando un'astronave con effetti luminosi nel caso in cui l'utente guarda a sinistra, e un robottino nel caso in cui l'utente guarda a destra.

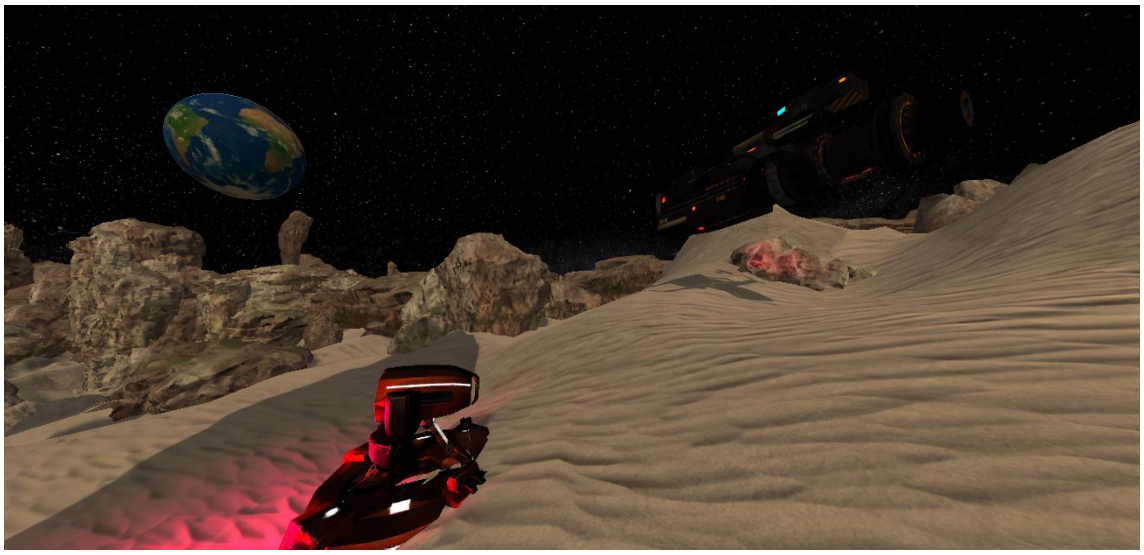

Figura 4.4 Ambientazione spazio

Nell'ambientazione Natura l'utente si trova in un ambiente tranquillo, caratterizzato da un paesaggio con alberi e colline. Per far sì che l'utente possa interagire maggiormente, gli oggetti che vengono creati in base all'orientamento del volto sono animali, in particolare viene generato un coniglio se l'utente guarda alla sua sinistra e un maialino se l'utente guarda alla sua destra.

Figura 4.5 Ambientazione natura

L’integrazione fatta con tali progetti, cui ringrazio tutti i miei colleghi per l’aiuto dato, ha permesso di mostrare le potenzialità del sistema sviluppato e la flessibilità con cui può adattarsi ad altri progetti.

Capitolo 5

Conclusioni

L'obiettivo di questa tesi era lo sviluppo di un applicativo in grado di riconoscere attraverso l'impiego delle reti neurali un volto da una sequenza di immagini 2D per poter creare la mappatura 3D di ogni volto riconosciuto e stimarne la posa. La rete neurale Face-Alignment Network sfrutta le informazioni 2D messe a disposizione, che hanno permesso di affrontare diversi problemi di cui risentono le immagini bidimensionali acquisite: variazione di illuminazione, pose non frontali, distanza. In particolare l'informazione 3D ottenuta dalla rete neurale è stata fondamentale per lo sviluppo del progetto. La rete neurale ha permesso di riconoscere in maniera efficiente i volti, localizzando 68 keypoint su occhi, naso, bocca, sopracciglia e contorno del viso. Grazie all'acquisizione in condizioni ottimali delle immagini RGB da webcam, questi punti sono stati semplicemente estrapolati e mappati in 3D in Unity. La rete neurale proposta presenta però alcune limitazioni. Essendo una rete neurale pre-addestrata, utilizzata così com'è può in qualche caso particolare non riconoscere correttamente il volto, come nella figura 3.3, e provocare un insuccesso del sistema, oppure mapparlo in maniera errata. Importante è stata soprattutto l'aggiunta della feature per stabilire il punto esatto in cui l'utente sta realmente guardando. Tale feature è risultata fondamentale per l'integrazione in altri progetti in cui si richiedeva un maggior coinvolgimento degli utenti con l'applicativo, evidenziando la propria flessibilità e validità.

5.1 Sviluppi futuri

Eventuali sviluppi futuri di tale lavoro possono riguardare:

- Estrazione di informazioni più resistenti alle variazioni di luce e ai cambiamenti della posa.
- Addestramento analitico delle reti neurali per permettere un maggior apprendimento di ogni volto e quindi una riduzione degli errori di riconoscimento.

- Integrazione del sistema con un dispositivo di acquisizione delle immagini più evoluto per garantire una migliore qualità delle immagini e di conseguenza una migliore gestione delle condizioni di luce.
- Rendere più robuste la reti neurali riguardo la stima della posa, infatti riconoscono correttamente il volto entro una distanza di 2 metri, tuttavia soggetti non posizionati entro tale distanza potrebbero non essere riconosciuti correttamente. Per risolvere questo problema si potrebbe pensare di addestrarle con un dataset specifico.
- Ottimizzazione del codice python per la codifica delle informazioni 3D da inviare al server in Unity, poiché attualmente rende l'esecuzione onerosa in termini di elaborazione.
- Integrazione di una feature robusta che permetta il riconoscimento di volti multipli.

Capitolo 6

Preparazione ambiente di sviluppo

In questo capitolo viene mostrata la procedura iniziale per gestire l'installazione e l'esecuzione dell'applicativo sviluppato.

6.1 Tecnologie utilizzate

Seguendo la procedura presente su github [17], lo sviluppo del progetto ha previsto principalmente l'impiego di Python 3.7, Unity 2019.3.6f1(64-bit), l'installazione da riga di comando del progetto sviluppato da Adrian Bulat e Pytorch 1.5.0:

Installazione Pytorch	Installazione Progetto
conda install -c pytorch pytorch	conda install -c 1adrianb face_alignment

6.1.1 Anaconda Navigator

Per l'esecuzione del listato python è stato utilizzato Anaconda Navigator [18]. Anaconda Navigator è un'interfaccia utente grafica desktop (GUI) che consente di avviare applicazioni e gestire pacchetti conda, ambienti e canali senza utilizzare i comandi della riga di comando. Affinché il progetto sia utilizzato correttamente sono stati installati diversi packages essenziali per l'acquisizione e gestione delle immagini, la gestione degli array e la rappresentazione grafica delle informazioni. I packages utilizzati sono:

- **Matplotlib.** È una libreria per la creazione di grafici per il linguaggio di programmazione Python e la libreria matematica NumPy. Fornisce API orientate agli oggetti che permettono di inserire grafici all'interno di applicativi usando toolkit GUI generici. C'è anche una interfaccia "pylab" procedurale progettata per assomigliare a quella di matlab.

- **OpenCV.** È una libreria software multipiattaforma nell'ambito della visione artificiale in tempo reale. La libreria permette una semplice gestione di immagini trattandole come "matrici di pixel **",** alle quali è possibile accedervi in maniera molto semplice e rapida.
- **Scikit-image.** È una raccolta di algoritmi per l'elaborazione delle immagini.
- **Numpy.** È una libreria open source per il linguaggio di programmazione Python, che aggiunge supporto a grandi matrici e array multidimensionali insieme a una vasta collezione di funzioni matematiche di alto livello per poter operare efficientemente su queste strutture dati.

Bibliografia

[1] Facial Recognition System as a Maritime Security Tool. Jyri Rajamäki, Tuomas Turunen, Aki Harju, Miia Heikkilä. January 2009.

[2] 2D and 3D face recognition: A survey. Andrea F. Abate, Michele Nappi, Daniel Riccio, Gabriele Sabatino. Pattern Recognition Letters: Volume 28, Issue 14, 15 October 2007, Pages 1885-1906.

[3] Face recognition on partially occluded images using compressed sensing. A.Morelli Andrés, S.Padovani, M.Tepper, J.Jacobo-Berlles. Pattern Recognition Letters: Volume 36, 15 January 2014, Pages 235-242.

[4] Three-Dimensional Face Recognition. Alexander Bronstein,Michael Bronstein; Ron Kimmel. International Journal of Computer Vision volume 64, pages 5–30(2005).

[5] Facial feature detection using Haar classifiers. Phillip Ian Wilson, Dr John Fernandez. Journal of Computing Sciences in Colleges, April 2006.

[6] Eigenfaces vs. Fisherfaces: Recognition Using Class Specific Linear Projection. Belhumeur, Peter N., Hespanha, João P. e Kriegman, David J. 7, s.l. : IEEE Transactions on pattern analysis and machine intelligence, 1997, Vol. 19.

[7] Face recognition using kernel principal component analysis. Kwang In Kim, Keechul Jung, Hang Joon Kim. IEEE Signal Processing Letters (Volume: 9, Issue: 2, Feb. 2002).

[8] Face Recognition Using Fisherface Method. Mustamin Anggo, La Arapu. Journal of Physics Conference Series 1028(1):012119 · June 2018.

[9] Two-dimensional FLD for face recognition. Huilin Xiong, M.N.S.Swamy, M.O.Ahmad. Pattern Recognition: Volume 38, Issue 7, July 2005, Pages 1121-1124.

[10] Two-dimensional Laplacianfaces method for face recognition. BenNiu, Qiang Yang, Simon Chi Keung Shiu, Sankar KumarPalc. Pattern Recognition: Volume 41, Issue 10, October 2008, Pages 3237-3243.

[11] Face recognition using discriminant locality preserving projections. Weiwei Yu, Xiaolong Teng, Chong qing Liu. Image and Vision Computing: Volume 24, Issue 3, 1 March 2006, Pages 239-248.

[12] Face recognition by statistical analysis of feature detectors. P. Kalocsai, C. von der Malsburg, J. Hornc. Image and Vision Computing: Volume 18, Issue 4, 1 March 2000, Pages 273-278.

[13] Face recognition as an airport and seaport security tool. Rajamäki Jyri, Turunen Tuomas, Harju Aki, Heikkilä Miia, Hilakivi Maarit, Rusanen Sami. Volume 6 2009.

[14] How Far Are We From Solving the 2D & 3D Face Alignment Problem?. Adrian Bulat, Georgios Tzimiropoulos; Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pp. 1021-1030.

[15] Slessans/Unity-socket-server: https://github.com/slessans/unity-socket-server/

[16] Distanza Euclidea. Wikipedia: https://it.wikipedia.org/wiki/Distanza_euclidea

[17] Face-Alignment. https://github.com/1adrianb/face-alignment

[18] Anaconda Navigator. https://docs.anaconda.com/anaconda/navigator/install