

Nutrition Reality

Relazione progetto Mixed Reality and Wearable
Vision

Salvatore Di Pietro
Deborah Maldonato

Indice

Prefazione

1. Introduzione (Salvatore Di Pietro)

1.1 L'idea

1.2 Funzionamento

2. Riconoscimento Alimenti (Salvatore Di Pietro)

2.1 QR Code

2.2 AI

2.3 Comunicazione Client-Server

3. UI (Deborah Maldonato)

3.1 Prefab

3.2 Scrittura degli script

3.3 Associazione bottoni script

3.4 Posizionamento Elem. (Salvatore Di Pietro & Deborah Maldonato)

3.5 Utilizzo dei controller e delle mani (Salvatore Di Pietro)

Prefazione

Negli ultimi decenni l'interesse per il fitness e l'alimentazione ha registrato una crescita significativa, sia a livello globale che in Italia.

Secondo le statistiche recenti della [Zippia The Carrer Expert](#), il valore di mercato dell'industria globale del fitness è di oltre 87 miliardi di dollari. Questo calcolo comprende una vasta gamma di prodotti e servizi, tra cui attrezzature per il fitness, palestre e centri, allenamenti a casa, programmi di fitness online e integratori alimentari. Con le ricerche effettuate da [TeamSystem](#), si stima che In Italia il settore del fitness ha registrato una crescita significativa. Dal 2014 al 2019, le imprese legate al fitness, allo sport e al benessere sono aumentate del 24%, con oltre 30.000 imprese e circa 70.000 addetti nel settore. Tuttavia, nonostante il periodo particolarmente critico, il lockdown e la chiusura forzata degli impianti hanno accelerato in modo importante il processo di digitalizzazione del fitness portando la popolazione a usufruire di vari servizi come: personal training online, videolezioni online, introduzione di app per l'acquisto e la fruizione di servizi legati al fitness, Social (YouTube, Instagram e TikTok) tramite contenuti pubblicati dai creator del mondo del fitness. Parallelamente, il mercato degli integratori alimentari per lo sport è in costante crescita. Nelle analisi fatte dallo studio [Businesscoot](#) si stima che negli ultimi cinque anni, il mercato globale è cresciuto di quasi 5 miliardi di dollari, con una crescente consapevolezza dell'importanza di una dieta equilibrata e del ruolo degli integratori nella performance sportiva e non solo. Contare le calorie giornaliere, capire la percentuale di micronutrienti che si è assunta e la quantità di acqua bevuta sono solo alcune delle cose principali fatte da parte della popolazione per la ricerca di uno stile di vita sano, sia per motivi personali ma anche per obiettivi sportivi. Il nostro scopo è quello di rendere pratica e veloce il modo in cui poter controllare questi aspetti dell'alimentazione sfruttando le nostre mani e il nostro sguardo con un piccolo aiuto da parte della tecnologia.

Introduzione

1.1 L'idea

Grazie anche alla crescita dell'utilizzo dei Social, molte persone hanno cominciato a comprendere l'importanza del curare il proprio corpo e la propria mente, non solo basandosi su piaceri soggettivi, ma anche su quanto l'attività fisica e seguire un'alimentazione possano portare a una maggiore consapevolezza di se stessi, dei risultati estetici importanti e una salute fisica e mentale molto forte. Un aspetto che spesso però risulta essere complicato e farraginoso è la gestione dell'alimentazione. C'è chi decide di appoggiarsi a professionisti del settore, come nutrizionisti e dietologi, chi invece decide di autogestire quali alimenti assumere e quali no, incombendo in molti casi in scelte errate ed esperienze che portano tutt'altro che i risultati attesi.

Tuttavia, ci sono persone che si documentano fino ad interessarsi agli argomenti trattati; magari mediante la lettura di libri, acquistando corsi online o sfruttando nel migliore dei modi Internet. Una cosa che accomuna tutte queste categorie di persone è il monitoraggio di tutto quello che si assume giornalmente, cercando di capire quante calorie si è assunte, che percentuali di carboidrati si è mangiati ecc... Inoltre, grazie anche alla promozione e all'importanza di non sprecare il cibo, parte della popolazione sfrutta diversi alimenti, magari anche pochi ingredienti che si hanno a disposizione in casa, per creare cibi più elaborati in modo da rendere anche un semplice spuntino già gustoso. Il tutto, spesso, viene gestito da un'applicazione scaricata sul proprio smartphone, ma che rende tutto molto complesso, portando l'utilizzatore ad abbandonare l'applicazione anche dopo pochi giorni o settimane. Lo scopo del progetto, che prende il nome Nutrition Reality, è quello di fornire un'applicazione che possa risolvere tutti i problemi descritti, permettendo all'utente di gestire la propria alimentazione in maniera facile

e comoda sfruttando la nostra essenza, ovvero il nostro corpo.

1.2 Sviluppo dell'applicazione

L'applicazione è stata sviluppata per Android attraverso **Unity (versione 6000.0.22f1)**, un game Engine utilizzato in vari ambiti, anche per la creazione di applicazioni in VR, MR, e AR che, combinata alla licenza freemium, è il motivo per cui si è deciso di optare per questo software. Unity è stato eseguito su un **MacBook Air 2020 con macOS Sequoia (versione 15.3)** poiché, oltre a essere un dispositivo prestante, è il dispositivo principale per uso personale di uno degli sviluppatori del progetto. In particolare, poiché l'applicazione è stata sviluppata principalmente per funzionare sui dispositivi Meta, si è sfruttato il package fornito dallo store di Unity (**Meta XR All-in-One SDK v.72.0.0**). L'applicazione è stata testata sul dispositivo **Meta Quest 3**, un visore VR pensato soprattutto per sfruttare la realtà mista. Infatti uno degli elementi fondamentali di questa applicazione è proprio l'utilizzo del passthrough, che consente di uscire dalla vista immersiva per visualizzare in tempo reale ciò che circonda l'utente, motivo per cui si presta perfettamente allo scopo dell'applicazione.

1.3 Funzionamento

L'applicazione è stata progettata per far sì che l'utente, una volta indossato il dispositivo, mediante i QR Code o sfruttando l'intelligenza artificiale in esecuzione, permetta a essa di riconoscere gli alimenti, garantendo all'utente, tramite degli appositi Pop-up che appariranno, di leggere tutte le caratteristiche dell'alimento e di poter interagire con gli elementi grafici forniti. Tra le scelte disponibili troviamo: aggiungere alla dispensa, consumo adesso ed esci.

Scegliendo la prima opzione l'alimento verrà aggiunto ad una dispensa virtuale, che sarà accessibile o tramite dei comandi vocali oppure tramite apposito menù, accessibile tramite un bottone presente in punto fisso nella vista dell'utente, in modo tale che possa accedervi facilmente. All'interno della dispensa sarà possibile visionare i propri alimenti con a fianco dei toggle di selezione che mostreranno le possibili ricette da preparare. Cliccando una ricetta, appariranno gli ingredienti e i passaggi dimostrativi sulla preparazione della ricetta scelta. Scegliendo l'opzione invece consumo adesso, le caratteristiche alimentari dell'alimento (calorie) verranno aggiunte ad una sezione utente, accessibile tramite una voce presente nel menù, in cui, una volta cliccato, verranno mostrate: le calorie assunte.

Riconoscimento Alimenti

Il punto cardine del progetto è proprio il riconoscimento degli alimenti che può avvenire in 2 modi:

- QR Code
- AI

2.1 QR Code

Grazie alla vasta gamma di siti presenti nel WWW che permettono la generazione di QR Code personalizzati, creati tramite una stringa specifica, l'utente sarà in grado di consumare o aggiungere alla dispensa i vari alimenti scannerizzando proprio un codice QR. I codici QR utilizzati per i test del software sono stati generati tramite il seguente sito:

<https://www.qr-code-generator.com/solutions/text-qr-code/>.

La stringa inserita per la generazione dei codici QR è la seguente:

NomeAlimento://NumeroCalorie://NutritionRealityCreateQRCode.

Le specifiche parti della stringa sono le seguenti:

- **NomeAlimento:** Specifica il nome dell'alimento (es Mela, Pera, Banana ecc...)
- **NumeroCalorie:** Specifica il numero di calorie dell'alimento (in questo caso i valori presi per i test sono dei valori medi estrapolati dalla seguente tabella:
<https://www.projectinvictus.it/calorie-alimenti/>)
- **//:**Carattere di separazione per poter manipolare la stringa
- **NutritionRealityCreateQRCode:** Stringa aggiuntiva per consentire al visore la lettura del codice QR poiché senza un numero esiguo di dati e correzione degli errori, il visore non riuscirebbe a riconoscere il codice QR.

Per la lettura dei QR Code si è sfruttata la seguente repository di GitHub:

<https://github.com/trev3d/QuestDisplayAccessDemo?tab=readme-ov-file>
<https://github.com/trev3d/QuestDisplayAccessDemo?tab=readme-ov-file>

Poiché ancora Meta non permette agli sviluppatori di accedere alla fotocamera sui Meta Quest per motivi legati alla privacy, la seguente repository consente di accedere al display del dispositivo che, grazie all'API MediaProjector di Android, è possibile copiare la sua immagine in una texture nel proprio progetto Unity quasi in tempo reale.(diversi fotogrammi di latenza). MediaProjection è un API messa a disposizione da Android che consente alle applicazioni di catturare il contenuto visualizzato sullo schermo di un dispositivo. Per i dettagli del suo funzionamento consultare: <https://developer.android.com/reference/android/media/projection/MediaProjection>. Per motivi di sicurezza e privacy, l'uso di MediaProjection richiede l'interazione dell'utente per avviare la sessione di cattura.

Una volta dato il consenso il software eseguirà i vari metodi necessari alla lettura del codice QR code. Lo script che si occupa delle funzioni principali per fornire all'utente un'interfaccia con cui possa interagire è: IndicatorDriver.cs Script già presente nella repository indicata prima, è stato modificato per consentire la generazione del pop-up. In particolare le modifiche apportate sono le seguenti:

- ```
[SerializeField] private GameObject currentPopUp; //Prefab di selezione: consuma, dispensa, no
[SerializeField] private Vector3 spawnPosition; //spawnPointPopUp
```

La variabile currentPopUp è un'istanza del pop-up che appare una volta letto il codice QR. Essa è stata dichiarata private poiché deve essere utilizzata solo in questo script. Il campo è dichiarato SerializeField così da poter far apparire la variabile nell'Inspector di Unity e passargli un GameObject, in questo caso il prefab del pop-up. Mentre spawnPosition è un attributo di tipo Vector3, dichiarato private per lo stesso motivo indicato sopra. Se il GameObject, in questo caso il pop-up, non è

presente nella gerarchia, viene istanziato tramite il metodo Instantiate nel metodo ShowPopUp. Esso è un metodo della classe Object di Unity che clona l'oggetto passatogli e ritorna il suo clone consentendo di posizionarlo in un determinato punto passandogli un Vector3, una struttura che permette di specificare i punti x, y e z del GameObject da invocare.

- `Invoke(nameof(currentPopUp), time: 0.5f);`  
Il metodo Invoke() è un metodo della classe MonoBehaviour, classe base di Unity. Esso consente di invocare un metodo dopo un numero di secondi prefissato. La sintassi del metodo è la seguente: **public void Invoke(string methodName, float time)**. Poiché uno dei parametri formali del metodo è di tipo string sfruttiamo il metodo nameof() che produce il nome di una variabile, un tipo o un membro come costante della stringa, in questo caso quello del popUp. In pratica questo metodo, nel suo complesso, fa sì che, una volta letto il codice QR, il pop-up appia dopo mezzo secondo.

```
private void ShowPopUp(string result)
{
 // Se il prefab è già stato istanziato, aggiorna solo il testo e attiva il pop-up
 if (currentPopUp != null)
 {
 textToChange = currentPopUp.GetComponentInChildren<TMPro.TextMeshProUGUI>();
 if (textToChange != null)
 {
 // Imposta il testo desiderato
 textToChange.SetText(result);
 Debug.Log("[TestDebug] Testo aggiornato");
 }
 else
 {
 Debug.LogWarning("[TestDebug] Componente TextMeshProUGUI non trovato nell'istanza");
 }

 // Assicurati che il prefab sia visibile
 currentPopUp.SetActive(true);
 Debug.Log("[TestDebug] Pop-up attivato");
 }
 else
 {
 // Se non è stato ancora assegnato, crea una sola volta
 Debug.Log("[TestDebug] Prefab non assegnato, lo istanzio ora.");

 if (currentPopUp == null) // Verifica se non è ancora stato creato
 {
 currentPopUp = Instantiate(dispensaPrefab.gameObject, spawnPosition, Quaternion.identity);
 }
 ShowPopUp(result); // Richiama la funzione per impostare il testo
 }

 ress = result; // Memorizza il risultato per uso successivo
}
```

Il metodo ShowPopUp(string result) permette di mostrare il GameObject passato prima nell'Inspector, in questo caso IngredientePopUp, con le caratteristiche dell'alimento scansionato tramite il QR Code. Infatti la linea di codice 4, textToChange, tramite il metodo GetComponentInChildren, restituisce la variabile di tipo TextMeshProUGUI contenuta nel prefab passato in currentPopUp, permettendo così di modificarne il contenuto passandogli il valore letto nel codice QR Code. Questa è la linea di codice richiamata nel metodo OnTrackBarcodes dello script:

```
ShowPopUp(result.text);
```

Il parametro attuale result.text è il contenuto testuale del codice QR letto dal visore, dove result è una variabile di classe Result, implementata nello script BarcodeTraker,

dichiarata in un foreach che scorre un array I`Enumerable<BarcodeTracker.Result>`, un oggetto di tipo I`Enumerable` che restituisce una sequenza di elementi di tipo BarcodeTracker.Result. La classe Result contiene al suo interno un attributo di tipo string chiamato text, che permette appunto di tenere traccia del contenuto testuale del codice QR.

## 2.2 AI

Sfruttando sempre la repository fornita sopra, tramite un'intelligenza artificiale in esecuzione su un server in locale, per la precisione l'AI in questione è **YOLO versione 5**, il software scatta dei frame di quello che il visore vede attraverso il display inviandoli al server che si occuperà di inviare una risposta testuale dell'alimento individuato al visore. Si è deciso di usare YOLO V5 come AI poiché utilizza Pythorch per un'implementazione più veloce e precisa e quindi compatibile con il linguaggio di programmazione Python. Essa è un modello di visione computerizzata utilizzato per la rilevazione

di oggetti. Si tratta di una versione avanzata dei precedenti modelli YOLO e funziona ad alta velocità di inferenza, rendendola efficace per le applicazioni in tempo reale. Inoltre è in grado di individuare 10 classi di alimenti quali: Mela, Banana, Carota, Stuzzichini, Kiwi, Cipolla, Arancia e Pomodoro. Il file che permette di eseguire il server è ServerDef.py. Esso al suo interno contiene il seguente codice:

```
Correggi l'orientamento dell'immagine, se necessario
img = ImageOps.exif_transpose(img)

Ruota l'immagine di 90 gradi se è necessario
img = img.rotate(180, expand=True)

Esegui YOLO sull'immagine
results = model(img)
Ottieni i risultati come dizionario
detections = results.pandas().xyxy[0].to_dict(orient="records")

Esegui il rilevamento degli oggetti

food_classes = {"apple", "banana", "carrot", "finger", "kiwi", "onion", "orange", "tomato"}

names = results.names # Lista dei nomi delle classi
predictions = results.pred[0] # Predizioni fatte dal modello
filtered_detections = [d for d in detections if d["name"] in food_classes]

if filtered_detections:
 # Prendi l'alimento con la probabilità più alta
 best_food = max(filtered_detections, key=lambda d: d["confidence"])
 food_name = best_food["name"]
 return jsonify({'message': f'{food_name}'}), 200
else:
 return jsonify({'message': 'Nessun alimento rilevato'}), 400
```

Una volta che il frame scattato viene caricato sul server, dato che Unity sfrutta un **sistema di coordinate** per le texture in cui l'asse Y è invertito rispetto a molti altri formati di immagine, tramite la linea di codice `img = ImageOps.exif_trasponse(img)` corregge l'orientamento di un'immagine in base ai suoi metadati EXIF, risolvendo il problema generato da Unity. Successivamente si esegue YOLO sull'immagine da cui poi viene creato un dizionario, una struttura dati che memorizza coppie **chiave-valore**. La funzione `.to_dict(orient="records")` converte il DataFrame, generato da `.pandas()` che appunto converte i risultati in un formato pandas DataFrame in una lista di dizionari, quindi

detections sarà una lista di strutture dati di tipo dizionario. Si è scelto di utilizzare i pandas DataFrame perché rendono l'accesso ai dati semplice e veloce e possono essere convertiti facilmente in un dizionario rispetto ad utilizzare un array. Successivamente si inizializza la variabile detections per memorizzare le predizioni fatte sul modello e si filtrano tramite la riga `filtered_detections = [d for d in detections if d["name"] in food_classes]`, che crea una list comprehension utile per filtrare gli elementi della lista detections, mantenendo solo quelli il cui "name" è presente nella lista food\_classes. Se la lista esiste si considera l'alimento con probabilità più alta sfruttando una variabile best\_food, filtrando la lista creata precedentemente ed estraendo il valore di confidenza più alto utilizzando lambda come chiave poiché la funzione implementata è temporanea e viene passata tramite i parametri, che appunto prende un dizionario e ne restituisce il valore associato alla chiave "confidence". Come ultima cosa si inizializza una variabile per memorizzare il nome dell'alimento individuato che ha la probabilità più alta, inviandola poi al visore.

## 2.3 Comunicazione Client-Server

Per inviare i frame catturati dal visore e ricevere risposta da parte del server sull'alimento individuato si è modificato lo script: `DisplayCaptureManager.cs`. Per la comunicazione tra client e server si è sfruttato il protocollo HTTP per via della sua semplicità di implementazione. Questi sono i metodi aggiunti:

```
1 usage
private async Task SaveFrameAsync(Texture2D texture)
{
 isUploading = true;
 byte[] textureBytes = texture.EncodeToPNG();
 if (textureBytes == null) return;
 _ = UploadFrameToServer(textureBytes, uploadURL);
 isUploading = false;
}
```

- Il metodo SaveFramAsync consente di inviare il frame catturato al server. isUploading è una variabile dichiarata precedentemente per indicare che l'upload è cominciato. L'attributo textureBytes viene inizializzato in modo che la Texture di Unity venga convertita in PNG e restituisca i dati come un array di byte. Se questo attributo è vuoto, il metodo non ha niente su cui lavorare e ritorna, altrimenti esegue una coroutine che invia i byte ottenuti al server, dove uploadURL indica l'indirizzo presso cui il server è in ascolto e reimposta isUploading falso per indicare che l'upload si è concluso. Il metodo è dichiarato come async Task per eseguire il metodo su un thread separato anziché eseguirlo su quello principale per evitare un sovraccarico.

```

private async Task UploadFrameToServer(byte[] textureBytes, string serverUrl)
{
 using (UnityWebRequest request = new UnityWebRequest(serverUrl, method: "POST"))
 {
 request.uploadHandler = new UploadHandlerRaw(textureBytes);
 request.downloadHandler = new DownloadHandlerBuffer();
 request.SetRequestHeader(name: "Content-Type", "application/octet-stream");
 request.certificateHandler = new BypassCertificateHandler();

 Debug.Log("[TestDebug] Invio della richiesta al server");

 //Usa un'operazione asincrona per non bloccare l'applicazione
 await request.SendWebRequest();

 if (request.result == UnityWebRequest.Result.Success)
 {
 Debug.Log("[TestDebug] Frame inviato con successo al server!");
 Debug.Log("[TestDebug] Risposta del server: " + request.downloadHandler.text);
 HandleServerResponse(request.downloadHandler.text);
 }
 else
 {
 Debug.LogError($"[TestDebug] Errore durante l'invio del frame: {request.error}");
 }
 }
}

```

- Il metodo UploadFrameToServer carica il frame sul server. La prima riga di codice genera una richiesta HTTP POST sul server in ascolto. Quello che segue sono delle impostazioni per consentire al visore di inviare il frame scattato, ricevere le risposte dal server e specificare che si stanno inviando dei file binari. Successivamente uso un'operazione asincrona per inviare una richiesta web al server. Asincrona perché così non si blocca il thread principale, altrimenti causerebbe rallentamenti e un malfunzionamento dell'applicazione. Infine se la richiesta ha avuto successo si stampano dei log di controllo e successivamente si chiama il metodo HandleServerResponse() che elabora la risposta del server.

```

private void HandleServerResponse(string response)
{
 try
 {
 // Deserializza la risposta JSON
 var responseObject = JsonUtility.FromJson<Response>(response);

 MainThreadDispatcher.Instance.Enqueue(() => ShowMessage(responseObject.message));
 }
 catch (System.Exception e)
 {
 Debug.LogError("Errore nel parsing della risposta del server: " + e.Message);
 }
}

```

-

Il metodo `HandleServerResponse` elabora la risposta del server. All'interno del try, `responseObject` deserializza la risposta JSON, dato che HTTP comunemente invia risposte in questo formato. `JsonUtility.FromJson<T>()` è un metodo di Unity che prende una stringa JSON e la converte in un oggetto di tipo T (in questo caso `Response`). Successivamente si usa la classe `MainDispatcher` dato che essendoci diverse operazioni asincrone è importante che il metodo `ShowMessage` (identico a quello descritto per il codice QR) sia eseguito sul thread principale essendo che esegue operazioni sull'UI. Se si genera un'eccezione viene stampato un log con l'eccezione. Questo è la classe `Response`:

```
// Classe per deserializzare la risposta JSON
[System.Serializable]
[1 usage]
public class Response
{
 public string message; [Serializable]
```

```
private unsafe void OnNewFrameAvailable()
{
 if (imageData == default || isPopUpActive) return;
 screenTexture.LoadRawTextureData((IntPtr)imageData, bufferSize);
 screenTexture.Apply();

 if (flipTextureOnGPU)
 {
 Graphics.Blit(screenTexture, flipTexture, scale: new Vector2(1, -1), offset: Vector2.zero);
 Graphics.CopyTexture(src: flipTexture, dst: screenTexture);
 }

 onNewFrame.Invoke();

 float currentTime = Time.time;
 if (currentTime - lastCaptureTime >= 2f && !isUploading)
 {
 lastCaptureTime = currentTime;
 Task.Run(() => SaveFrameAsync(screenTexture));
 }
}
```

Il metodo OnNewFrameAviable era un metodo già presente in DisplayCaptureManager. Le modifiche apportate riguardano il fatto che i frame vengono inviati ogni 2 secondi. La variabile currentTime è inizializzata a Time.time, che rappresenta il **tempo in secondi** che è passato dall'inizio dell'avvio del software o dalla scena corrente. Se sono passati 2 secondi e non c'è alcun caricamento in corso (isUploading = false) e aggiorna il tempo dell'ultima cattura.

```
public class MainThreadDispatcher : MonoBehaviour
{
 private static readonly Queue<Action> _executionQueue = new Queue<Action>();

 private static MainThreadDispatcher _instance;
 # 1 usage
 public static MainThreadDispatcher Instance
 {
 get
 {
 if (_instance == null)
 {
 GameObject obj = new GameObject(name: "MainThreadDispatcher");
 _instance = obj.AddComponent<MainThreadDispatcher>();
 DontDestroyOnLoad(obj);
 }
 return _instance;
 }
 }

 # Event function
 private void Update()
 {
 while (_executionQueue.Count > 0)
 {
 _executionQueue.Dequeue()?.Invoke();
 }
 }

 # 1 usage
 public void Enqueue(Action action)
 {
 lock (_executionQueue)
 {
 _executionQueue.Enqueue(action);
 }
 }
}
```

• La classe MainDistpacher viene utilizzata per riportare le azioni sul thread principale, in particolare le azioni UI.

L'attributo `Queue<Action>` è una coda che contiene queste azioni. Questa classe viene implementata come singleton dato che il thread principale è unico. Il metodo `Update` è un metodo di Unity che viene chiamato ad ogni frame e vi è all'interno un while che controlla se ci sono processi in coda. Se ci sono li esegue. Il metodo `Enqueue` invece permette di aggiungere alla coda delle azioni che, una volta acquisito il lock, verrà eseguito nel successivo ciclo di `Update`.

Il server utilizzato è stato sviluppato in Python per poter sfruttare il modello di YOLO. In particolare il server implementato è un server realizzato con Flask. Il server è stato implementato in un ambiente virtuale per poter sfruttare le varie librerie, in particolare Pytorch, in un ambiente isolata dalla macchina locale. Il comando per poter avviare da terminale il server è il seguente: `python -m venv myenv`. Per potervi accedere si inserisce il comando: `source myenv/bin/activate`. Una volta avuto accesso si esegue il codice python con: `python serverDef.py`

Si è scelto Flask perché permette di creare applicazione web in maniera semplice e veloce.

Il codice è il seguente:

```
from flask import Flask, request, jsonify
import os
import torch
from PIL import Image, ImageOps

app = Flask(__name__)

UPLOAD_FOLDER = 'uploads'
os.makedirs(UPLOAD_FOLDER, exist_ok=True)
model = torch.hub.load('ultralytics/yolov5', 'custom', path='yolov5s.pt', force_reload=True)

@app.route('/upload', methods=['POST'])
def upload_file():
 try:
 # Ricevi i dati binari inviati
 file_data = request.data
 except:
 if not file_data:
 return 'No data received', 400

 # Crea un nome per il file
 filename = os.path.join(UPLOAD_FOLDER, f"frame_{len(os.listdir(UPLOAD_FOLDER)) + 1}.png")

 # Salva il file
 with open(filename, 'wb') as f:
 f.write(file_data)

 print(f"File salvato: {filename}") # Log per confermare che il file è stato salvato

```

```
if __name__ == '__main__':
 app.run(debug=True, host='0.0.0.0', port=5000, ssl_context=('cert.pem', 'private_key.pem'))
```

Si crea l'applicazione Flask passando il parametro `__name__` per indicare a Flask di utilizzare il modulo corrente per configurare l'applicazione. Successivamente si crea la cartella dove verranno salvati i frame scattati (`uploads`) prima dichiarandone il nome e poi, utilizzando la funzione `makedirs` della libreria `os`, creare la cartella. Se non esiste viene creata, altrimenti il secondo parametro di quest'ultima funzione evita di sollevare un'eccezione. In seguito si richia, tramite `torch.hub.load`, un modello pre-addestrato YOLOv5 da un repository pubblico di PyTorch Hub. Il modello caricato è specificato come `custom` e caricato tramite il file `yolov5s.pt`, che rappresenta il modello addestrato.

`force_reload=True` forza il ri-caricamento del modello ogni volta che viene eseguito il codice. `@app.route('/upload', methods=['POST'])` definisce una route (un endpoint) dell'applicazione Flask ed è utilizzato per associare un URL (endpoint) a una funzione. In questo caso, l'endpoint è `/upload` e accetta solo richieste POST. Un endpoint è il punto d'accesso che i client utilizzano per comunicare con un server o un'applicazione web, inviare richieste e ottenere risposte. Si dichiara una funzione dal nome `upload_file()` dove all'interno si crea il nome per un file univoco incrementando un contatore basato sul numero di file già presenti, apre il file in modalità scrittura binaria e ci scrive i dati sopra. In fine l'`if` su `__name__` fa sì che, se il file python è eseguito direttamente, il valore `__name__` è impostato su `__main__`. `app.run()` avvia il server Flask, dove il flag `debug = true` avvierà il server in modalità sviluppo, così che ad ogni modifica del codice il server si ricarica automaticamente. L'indirizzo 0.0.0.0 indica che il server è in ascolto su ogni indirizzo disponibile sulla macchina su cui il server è in esecuzione. Il 5000 indica che la porta del server è la 5000. `ssl_context` abilità HTTPS utilizzando i certificati SSL. Per la generazione di `cert.pem` e `private_key.pem` si è utilizzato il comando da terminale: `openssl req -x509 -newkey rsa:4096 -keyout private_key.pem -out cert.pem -days 365 -nodes`. Esso consente di creare un

certificato autofirmato. Poiché il Meta Quest 3 non permette di utilizzare i certificati autofirmati è stato implementato la seguente callback:

```
private bool ValidateCertificate(object sender, X509Certificate certificate, X509Chain chain, System.Net.Security.SslPolicyErrors sslPolicyErrors)
{
 // Ignora gli errori del certificato
 return true;
}
```

Essa ignora completamente gli errori SSL e accetta qualsiasi certificato, anche se non valido dove: sender è l'oggetto che ha generato la richiesta, certificate il certificato ricevuto dal server che deve essere validato, chain la catena di certificati che collega il certificato ricevuto a un'autorità di certificazione attendibile e sslPolicyErrors che indica eventuali errori sul certificato. Si è implementata un callback così che la funzione venga richiamata solo quando c'è un certificato da validare. La callback viene effettuata nell'Awake con il seguente codice:

```
ServicePointManager.ServerCertificateValidationCallback = ValidateCertificate;
```

ServicePointManager è una classe di C# che gestisce la connessione di rete HTTP.

Tramite ServerCertificateValidationCallBack si passa il metodo creato prima per far sì che gli errori SSL vegano ignorati.

L'URL da inserire per poter far comunicare il server con il visore è dato da questa riga:

```
private string uploadURL = "https://192.168.10.28:5000/upload";
```

L'URL è ricavabile anche avviando il server:

```
(myenv) (base) ture14@MacBook-Air-di-Salvatore TheNutritionReality % python3 serverDef.py
Downloading: "https://github.com/ultralytics/yolov5/zipball/master" to /Users/ture14/.cache/torch/hub/master.zip
YOLOv5 🚀 2025-2-19 Python-3.12.8 torch-2.5.1 CPU

Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs
Adding AutoShape...
* Serving Flask app 'serverDef'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on https://127.0.0.1:5000
* Running on https://192.168.10.28:5000
Press CTRL+C to quit
* Restarting with stat
Downloading: "https://github.com/ultralytics/yolov5/zipball/master" to /Users/ture14/.cache/torch/hub/master.zip
YOLOv5 🚀 2025-2-19 Python-3.12.8 torch-2.5.1 CPU

Fusing layers...
YOLOv5s summary: 213 layers, 7225885 parameters, 0 gradients, 16.4 GFLOPs
Adding AutoShape...
* Debugger is active!
* Debugger PIN: 351-007-738
```

# UI

Una volta parlato di come si è riusciti a far riconoscere gli alimenti, non rimane che sviluppare l'effettiva applicazione che consentirà all'utente di usufruire del nostro servizio. Per realizzare l'Interfaccia Utente abbiamo diviso il lavoro in quattro macro-sezioni:

- Creazione dei prefab
- Scrittura degli script
- Associazione script-bottoni
- Ancoraggio

## 3.1 Prefab

Sono stati necessari diversi prefab per il corretto funzionamento dell'applicazione, vediamoli uno ad uno:

### StartPopUp

Il punto di partenza, contiene il bottone di avvio “dell'applicazione”, ciò vuol dire in realtà che caricherà la scena effettiva dove avverrà la scansione e la scelta degli ingredienti.



SceneChanger . LoadNextScene()



### PermessoPopUp

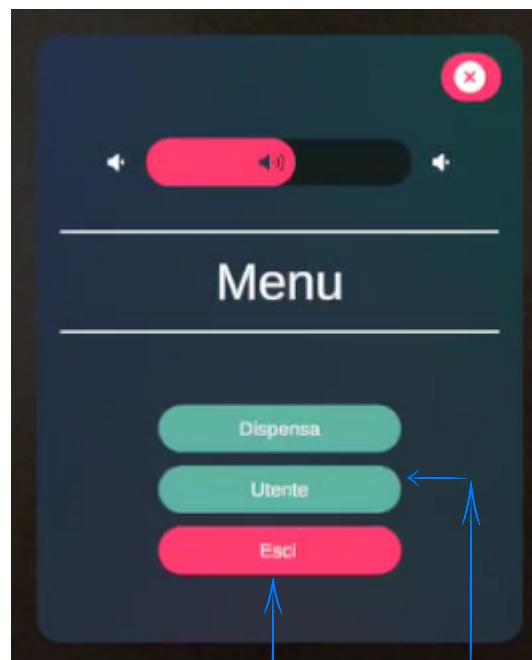
Informa l'utente che sarà necessario permettere la registrazione dello schermo per utilizzare l'applicazione



User.AddCal  
Ingrediente.  
Ingrediente() keepIngredient()

### IngredientePopUp

Apparirà non appena la registrazione troverà un ingrediente (o il suo QR-code) e permetterà di scegliere se: aggiungerlo in dispensa, consumarlo (quindi sommarlo al conteggio di calorie) o non farci nulla.



User.showCalories()  
Exit.ExitApplication()

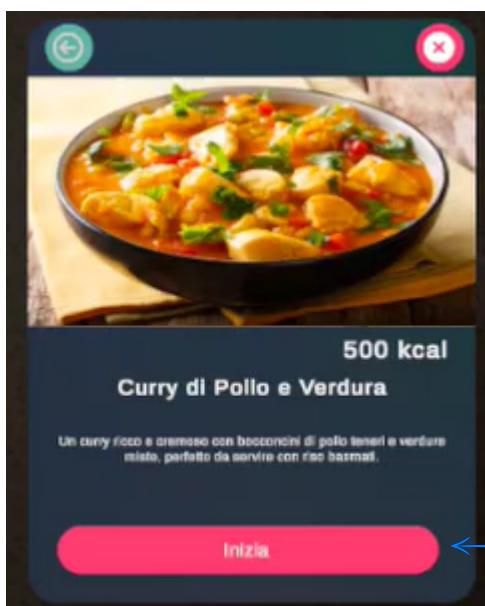
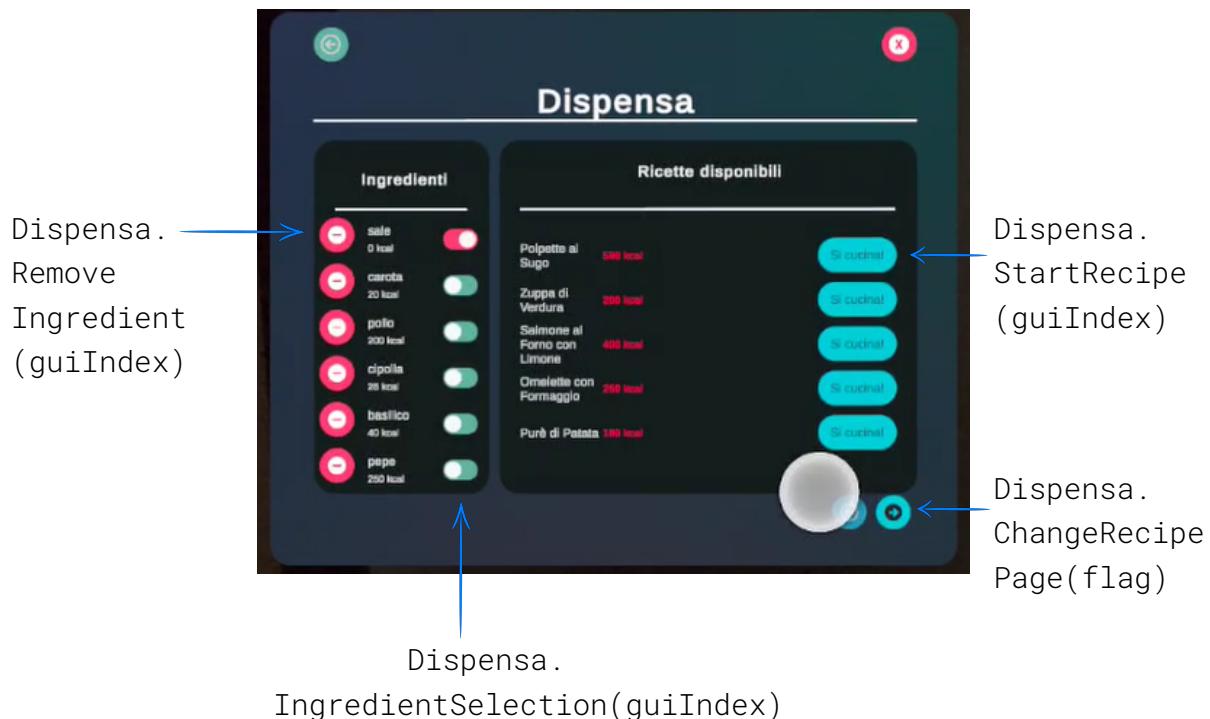
### MenuPopUp

Fa accedere al DispensaPopUp, UserPopUp e fa uscire dall'applicazione. Vi si accede tramite la Menulcon →



## DispensaPopUp

E' il cuore dell'applicazione, qui troviamo gli ingredienti stati aggiunti tramite scan, la loro gestione (rimozione e selezione) e le ricette che ci permettono di cucinare! Queste ultime infatti verranno visualizzate tramite il pop-up **Recipe**

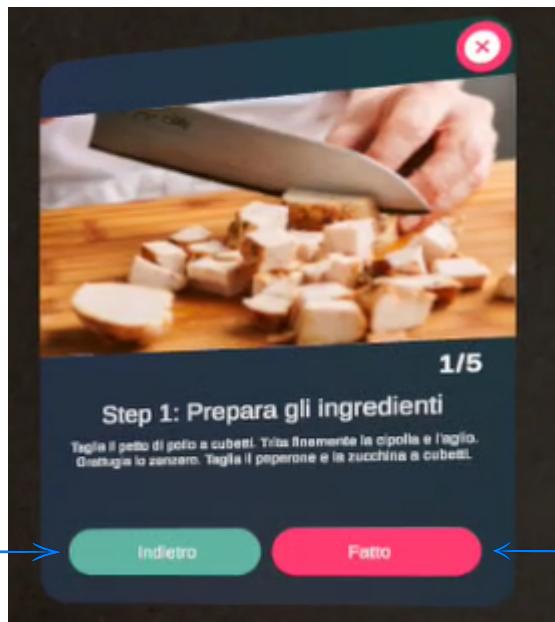


## Recipe

Da il vero e proprio via alla ricetta, mostrandone una breve introduzione, per poi passare a **Step**.

## Step

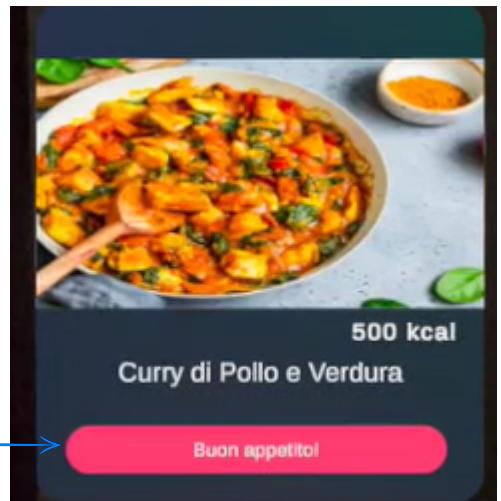
Da qui avremo modo di visionare i passaggi della ricetta mentre cuciniamo.

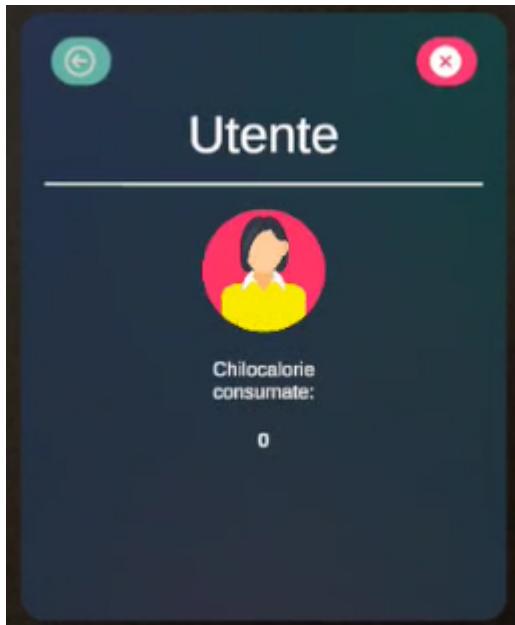


Alla fine passerà al pop-up End.

StepManager .  
NextStep()

**End**  
Contrassegna la fine della ricetta e si occupa di conservarne le calorie nel contatore totale.



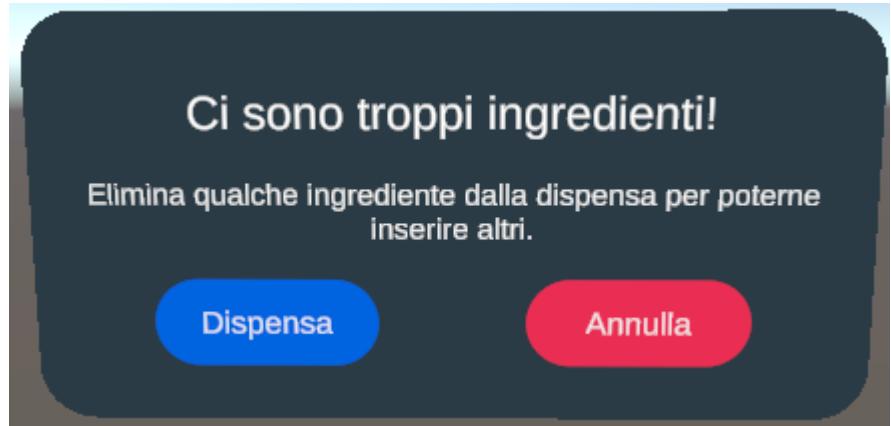


### UserPopUp

Questo è il pop-up che interesserà principalmente all'utente; vi si accede tramite il menù e si occupa di tenere conto delle calorie assunte mano a mano che vengono consumati ingredienti e ricette (che vengono mostrate tramite la funzione che richiama il menù alla pressione del tasto "Utente").

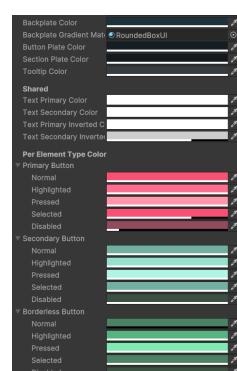
### Overload

Infine, troviamo l'Overload, esso si occupa di avvertirci quando ci sono troppi ingredienti in dispensa per poterne aggiungere altri.



### Nota:

Per il tema dei prefab ho deciso di realizzarne uno mio, per renderlo il più adatto possibile ai fini dell'applicazione, utilizzando colori allegri e coesi tra loro. Per fare ciò ho modificato uno degli asset di base che fornisce Meta per i pop-up dell'UI Set (`UIThemeQuest.asset`) con la palette qui di lato.



Infine l'ho assegnato a tutti i prefab.

## 3.2 Scrittura degli script

Per il funzionamento dei prefab appena visti è stata necessaria una cospicua quantità di linee di codice (in C#), vediamo principalmente le funzioni associate ai bottoni:

### LoadNextScene()

Si trova nello script *SceneChanger*, si occupa semplicemente di caricare la scena successiva (`currentScene + 1`) nell'ordine impostato nel building profile scelto.

```
0 references
public void LoadNextScene()
{
 int currentScene = SceneManager.GetActiveScene().buildIndex;
 SceneManager.LoadScene(currentScene + 1);
}
```

### ExitApplication()

```
0 references
public class Exit : MonoBehaviour
{
 0 references
 public void ExitApplication(){
 Application.Quit();
 }
}
```

Script *Exit*, si occupa di chiudere l'applicazione (`Application.Quit`).

### AddCalIngrediente()

Nello script *User*, prende il numero delle calorie dalla GUI dell'IngredientePopUp (`calorieIngrediente`) e lo somma al totale delle calorie dell'utente (`calTot`).

```
0 references
public void addCalIngrediente() {
 string text = calorieIngrediente.text.Trim();
 calTot += Int32.Parse(text.Where(char.IsDigit).ToArray());
 Debug.Log($"User: Sto passando le calorie dell'ingrediente consumato.{calorieIngrediente.text}");
}
```

## KeepIngredient()

Script *Ingrediente*, entra in gioco quando si sceglie di voler conservare l'ingrediente in dispensa; infatti richiama proprio una funzione di quest'ultima  
(dispensa.AddIngredient(nomeIngrediente, numeroCalorie)).

```
0 references
public void keepIngredient()
{
 // Aggiunge alla lista di ingredienti in dispensa, passando sia il nome che le calorie
 dispensa.AddIngredient(nomeIngrediente, numeroCalorie); // AddIngredient ora normalizza internamente
 Debug.Log($"Ingrediente '{nomeIngrediente}' ({numeroCalorie} cal) passato alla dispensa.");
}
```

## AddIngredient(nomeIngrediente, calorieIngrediente)

Script *Dispensa*, in breve:

- “Pulisce” il nome dell’ingrediente che deve analizzare (1)
- Verifica che l’ingrediente non sia già in dispensa (2)
- Lancia il pop-up Overload se ci sono troppi ingredienti in dispensa (3)
- Se è tutto in regola, conserva l’ingrediente in dispensa salvandolo in un file json (dispensa\_data.json) e ne calcola le ricette disponibili (se selezionato) (4).

```
0 references
public void AddIngredient(string nomeIngrediente, string calorieIngrediente)
{
 Debug.Log($"Dispensa: Tentativo di aggiungere ingrediente: '{nomeIngrediente}'.");

 // Pulizia stringa risultato ingrediente
 string normalizedName = nomeIngrediente.Trim().ToLowerInvariant(); // Trim() rimuove gli spazi

 // Verifica se l’ingrediente è già presente nella dispensa
 if (ingredients.Exists(i => i.name == normalizedName))
 {
 Debug.Log($"Dispensa: Ingrediente '{normalizedName}' è già presente in dispensa. Non aggiunto nuovamente.");
 Output();
 return; // Esci se l’ingrediente è un duplicato
 }

 if (ingredients.Count < 6) // Controllo se ci sono troppi ingredienti (limite 6 per la GUI)
 {
 Ingredient i = new Ingredient(normalizedName, calorieIngrediente);
 ingredients.Add(i); // Aggiunge l’ingrediente scansionato alla lista
 SaveData(); // Salva i dati dopo aver aggiunto un ingrediente

 Debug.Log($"Dispensa: Ingrediente '{normalizedName}' aggiunto alla lista 'ingredients'. Numero ingredienti in dispensa: {ingredients.Count}.");

 // Pulisce la lista recipesWithIngredients della dispensa per prepararla al ricalcolo
 recipesWithIngredients.Clear();
 recipeTot = 0;
 RicetteManager.Instance.UpdateDispensaWithAvailableRecipes();
 }
 else
 {
 Debug.Log($"Dispensa: Troppi ingredienti, impossibile aggiungere " + nomeIngrediente);
 showOverload(); // Mostra il popup di sovraccarico
 }
}
```

1

2

4

3

## showCalories()

Script *User*, semplicemente manda nella GUI del UserPopUp (`calTotGUI`) la calorie totali calcolate finora (`calTot`).

```
0 references
public void ... showCalories(){
 calTotGUI.SetText(calTot.ToString());
}
```

## RemoveIngredient(guiIndex)

Script *Dispensa*, rimuove l'ingrediente di cui è stato premuto l'apposito bottone nel DispensaPopUp. Per farlo:

- Rimuove effettivamente l'ingrediente dalla lista che li conserva (`ingredients`) (1)
- Deseleziona tutti gli ingredienti in modo che non si creino problemi con il calcolo delle ricette (2)
- Aggiorna il RicetteManager in modo che vengano rimosse tutte le ricette che contengono l'ingrediente (3).

```
0 references
public void RemoveIngredient(int guiIndex)
{
 Debug.Log($"Dispensa: Tentativo di rimuovere ingrediente all'indice GUI: {guiIndex}");
 string removedIngredientName = ingredients[guiIndex].name;
 ingredients.RemoveAt(guiIndex);
 for (int i = 0; i < 6; i++) bottoneIngrediente[i].isOn = false; //Deseleziona tutto anche nella GUI
 recipeIngredients.Clear(); // Deseleziona tutti gli ingredienti quando ne elimina uno
 SaveData();

 // Pulisco il ricetteManager
 Debug.Log($"Dispensa: Ingrediente '{removedIngredientName}' rimosso da 'ingredients'. Numero ingredienti in dispensa: {ingredients.Count}");
 recipesWithIngredients.Clear();
 recipeTot = 0;
 Debug.Log("Dispensa: Lista recipesWithIngredients pulita.");
 RicetteManager.Instance.UpdateDispensaWithAvailableRecipes();
}
```

1  
2  
3

## IngredientsSelection(s)

Script *Dispensa*, gestisce la selezione dell'ingrediente da parte dell'utente e ne determina le ricette da mandare in output nel DispensaPopUp.

Se l'ingrediente è stato deselezionato allora lo rimuove dalla lista degli ingredienti di una possibile ricetta (1) se no lo aggiunge (2). Alla fine aggiorna il RicetteManager.

```
0 references
public void ingredientsSelection(int s)
{
 string selectedIngredientNameGUI = nomeIngrediente[s].text;
 Debug.Log($"Dispensa: Nome ingrediente originale dalla GUI: '{selectedIngredientNameGUI}'. Indice: {s}");

 // Pulizia stringa del nome ingrediente selezionato
 string normalizedSelectedIngredientName = selectedIngredientNameGUI.Trim().ToLowerInvariant();

 // Controlla se l'ingrediente è già stato selezionato
 if (recipeIngredients.Contains(normalizedSelectedIngredientName))
 {
 // Se già presente lo deseleziona
 recipeIngredients.Remove(normalizedSelectedIngredientName);
 Debug.Log($"Dispensa: Deselezionato ingrediente: '{normalizedSelectedIngredientName}' .");
 }
 else
 {
 // Se non presente, lo seleziona
 recipeIngredients.Add(normalizedSelectedIngredientName);
 Debug.Log($"Dispensa: Selezionato ingrediente: '{normalizedSelectedIngredientName}' .");
 }

 // Il RicetteManager aggiorna le ricette disponibili in base alla selezione
 Debug.Log("Dispensa: Chiamo RicetteManager per ricalcolare ricette dopo selezione/deselezione.");
 recipesWithIngredients.Clear();
 recipeTot = 0;
 Debug.Log("Dispensa: Lista recipesWithIngredients pulita.");
 RicetteManager.Instance.UpdateDispensaWithAvailableRecipes(); // Chiede al RicetteManager di ripopolarla
}
```

1

2

## startRecipe(s)

Script *Dispensa*, recupera la ricetta stata scelta dall'utente e la manda allo StepManager in modo che ne mandi in video il Recipe pop-up e, successivamente, gli Step pop-up.

```
0 references
public void startRecipe(int s)
{
 Debug.Log($"Dispensa: startRecipe() - Indice ricetta selezionata: {s}.");

 // Recupera l'oggetto Ricetta dalla lista
 Ricetta selectedRicetta = recipesWithIngredients[s];
 Debug.Log($"Dispensa: Avvio ricetta '{selectedRicetta.nome}'.");

 StepManager stepManager = GameObject.FindObjectOfType<StepManager>(); //
 stepManager.guiRecipe(selectedRicetta);
}
```

## ChangeRecipePage(move)

Script *Dispensa*, richiama il PageManager che si occuperà di stampare ricette successive (o precedenti, in base a move) a quelle della pagina corrente.

```
0 references
public void ChangeRecipePage(int move)
{
 PageManager.Instance.changePage(move); // Chiede al PageManager di cambiare pagina
 Output(); // Output della nuova pagina
 Debug.Log("Dispensa: Pagina ricette cambiata e GUI aggiornata.");
}
```

## StartRecipeSteps()

Script *StepManager*, si occupa di far partire gli Step pop-up dopo il Recipe pop-up, aggiorna la GUI tramite `UpdateCurrentStepUI()`.

```
public void StartRecipeSteps()
{
 Debug.Log("StepManager: StartRecipeSteps chiamato. Passando al primo step.");
 if (ricetta == null || ricetta.step == null || ricetta.step.Count == 0)
 {
 Debug.LogWarning("StepManager: Impossibile iniziare gli step. Nessuna ricetta caricata o senza step.");
 hideRecipeOverview();
 return;
 }

 // Nasconde il popup di inizio ricetta
 hideRecipeOverview();
 stepNumber = 0;

 // Mostra il popup del primo step
 stepPopUp.SetActive(true);
 Debug.Log("StepManager: Popup Step attivato (primo step).");

 // Aggiorna la GUI con il primo step
 UpdateCurrentStepUI();
}
```

## NextStep() e PreviousStep()

Script *StepManager*,

```
public void NextStep()
{
 stepNumber++;

 if (ricetta == null || ricetta.step == null || ricetta.step.Count == 0)
 {
 Debug.LogWarning("StepManager: Nessuna ricetta caricata o step mancanti.");
 return;
 }

 // Se siamo all'ultimo step o oltre
 if (stepNumber >= ricetta.step.Count)
 {
 hideStep();
 showEnd(); // Mostra il popup di fine ricetta
 return;
 }

 // Aggiorna la GUI
 UpdateCurrentStepUI();
}
```

una funzione passa allo step successivo  
(`stepNumber++`)

l'altra a quello precedente  
(`stepNumber--`)

```
public void PreviousStep()
{
 stepNumber--;

 if (stepNumber < 0) // Se siamo tornati prima del primo step
 {
 stepNumber = 0; // Resetta per sicurezza
 hideStep();
 guiRecipe(ricetta); // Torna al riepilogo della ricetta
 return;
 }

 // Aggiorna la GUI
 UpdateCurrentStepUI();
}
```

semplicemente aggiornando la GUI sempre dello stesso Step pop-up.

### [addCalRicetta\(\)](#)

Script *User*, riceve in input le calorie della ricetta completata nell'End pop-up (*calorieRicetta*) e le somma al conteggio totale dopo aver ripulito la stringa.

```
public void addCalRicetta()
{
 string text = calorieRicetta.text.Trim();
 calTot += Int32.Parse(new string(text.Where(char.IsDigit).ToArray()));
 Debug.Log($"User: Sto passando le calorie della ricetta consumata.{calorieIngrediente.text}");
}
```

### [Tutti gli script, in breve](#)

Tutte le funzioni mostrate finora sono solo una porzione del totale delle linee di codice state scritte, per questo riporto un riassunto di cosa fanno in generale tutti i singoli script necessari per il corretto funzionamento dei prefab:

- *Dispensa*: gestisce il singleton della dispensa, si occupa della GUI di ingredienti e ricette, della selezione degli ingredienti, dell'overload, della rimozione e aggiunta degli ingredienti e del loro salvataggio nel file json.
- *Exit*: chiude l'applicazione.
- *Ingrediente*: si occupa dell'IngredientePopUp e della sua GUI e conserva gli ingredienti in dispensa.
- *PageManager*: gestisce il suo singleton, cambia le pagine nel DispensaPopUp aggiornando la sua GUI.
- *Ricetta*: ha il costruttore e controlla se la ricetta corrisponde agli ingredienti selezionati o meno.
- *RicetteData*: conserva i dati delle ricette memorizzate nell'app (in locale).
- *RicetteManager*: singleton, aggiorna il DispensaPopUp con le ricette corrispondenti agli ingredienti selezionati.
- *SceneChanger*: carica le scene.

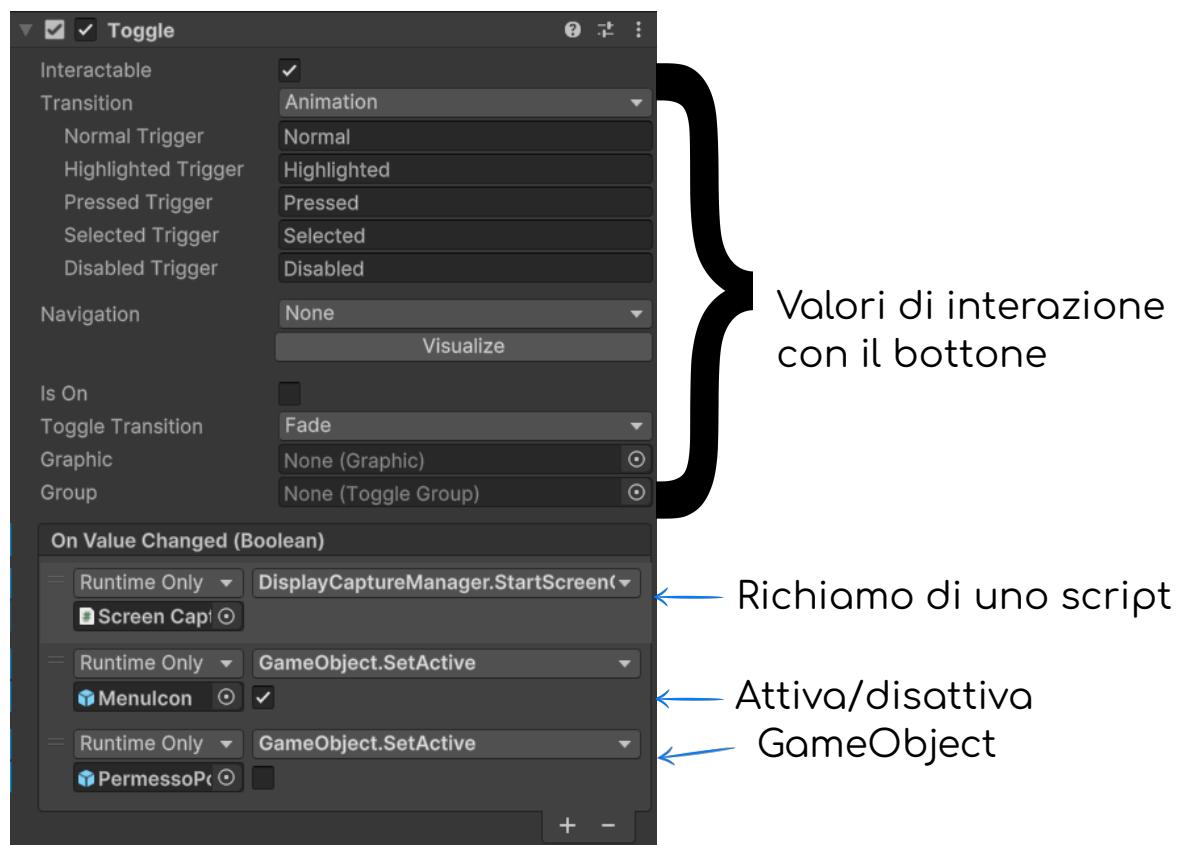
- *StepManager*: singleton, gestisce il Recipe, Step ed End pop-up e le loro GUI in base alla fase corrente della ricetta in esecuzione.
- *User*: gestisce la GUI del UserPopUp e ha le funzioni per il calcolo delle calorie totali.

### 3.3 Associazione bottoni script

Abbiamo visto sia i prefab che gli script a essi associati, ma come avviene effettivamente il collegamento tra i due?

Semplicemente, in una sezione dedicata dell'inspector. Considerando che i prefab sono stati utilizzati grazie al toolkit fornito da Meta per gli elementi grafici (in particolare la scena importata UI Set), i componenti di tipo "Button" e "ToggleSwitch" hanno al loro interno una sezione che si occupa di cosa fare accadere quando vengono premuti, la sezione Toggle.

Di seguito un esempio (del permessoPopUp):



Questa operazione è stata svolta in tutti i prefab visti in precedenza, facendo in modo che non si accavallino tra di loro durante le azioni dell'utente.

### 3.4 Posizionamento Elementi

Per quanto riguarda il posizionamento dei GameObject alla testa dell'utente si sono usati 2 script: HeadLockedUI.cs e HeadLockedUIX.cs

Il primo viene utilizzato per quegli elementi dell'UI che possono deformarsi lungo tutto gli assi, come l'icona del menù e l'icona dell'area utente.

Il codice è il seguente:

```
◊ 12 asset usages
public class HeadLockedUI : MonoBehaviour
{
 [SerializeField] private Transform headTransform; ◊ Changed in 4 assets
 [SerializeField] private Vector3 localOffset = new Vector3(-0.3f, 0, 1f); // Regola la posizione rispetto alla testa

 ◊ Event function
 void Update()
 {
 if (headTransform != null)
 {
 transform.position = headTransform.position + headTransform.TransformDirection(localOffset);
 transform.rotation = headTransform.rotation;
 }
 }
}
```

L'attributo headTransform è un attributo di tipo Transform e rappresenta la testa dell'utente (in questo caso come riferimento si è preso il CenterEye del buildingblock CameraRig). È dichiarato come SerializeField proprio per poter passare il riferimento della testa dell'utente dall'inspector di Unity. localOffset regola la posizione dell'UI rispetto alla testa, anche questo può essere modificato dall'inspector. Dato che l'aggiornamento deve essere fatto ad ogni frame, si sfrutta il metodo Update di Unity, dove, se la il riferimento alla testa dell'utente non è nullo, si imposta la posizione nello spazio 3D (transform.position) uguale alla posizione globale dell'headTransform convertendo l'offset locale in uno spazio globale, così che l'UI rimanga sempre

nella direzione giusta della testa. Infine si allinea la rotazione della testa all'UI, in modo che ne segua i movimenti.

Il secondo invece è il seguente:

```
public class HeadLockedUIElementX : MonoBehaviour
{
 public Transform headTransform; // Changed in 3 assets
 public Vector3 localOffset = new Vector3(-0.3f, 0, 1f); // Regola la posizione rispetto alla testa // Serializable

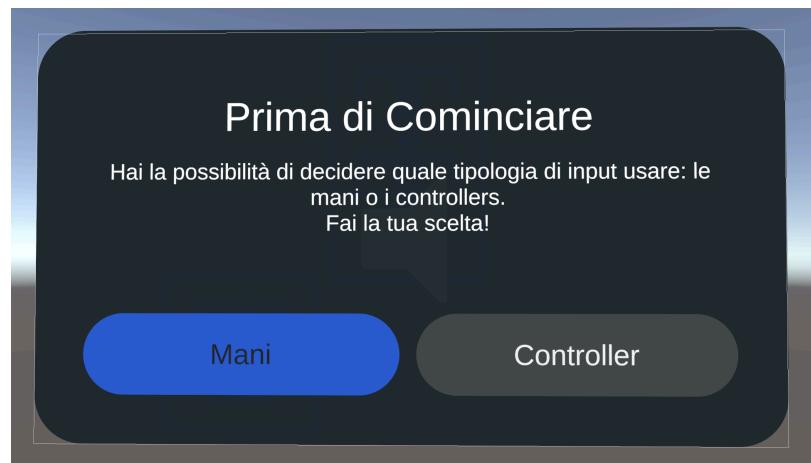
 // Event function
 void Update()
 {
 if (headTransform != null)
 {
 // Imposta la posizione relativa alla testa
 transform.position = headTransform.position + headTransform.TransformDirection(localOffset);

 // Mantiene la UI sempre dritta
 Vector3 lookDirection = transform.position - headTransform.position;
 lookDirection.y = 0; // Rimuove la componente verticale per evitare inclinazioni
 transform.rotation = Quaternion.LookRotation(lookDirection);
 }
 }
}
```

La differenza rispetto al codice precedente è in questo la y rimane fissa per evitare inclinazioni. lookDirection rappresenta la direzione in cui l'UI deve guardare, annullando poi la componente verticale e che crei una rotazione in modo che guardi in direzione di lookDirection(Quaternion.LookRotation(lookDirection)), ovvero che l'UI continui a guardare sempre in direzione verso l'utente

### 3.5 Utilizzo delle mani e dei controller

Poiché la maggior parte dei visori sul mercato permette di potervi interagire usando le mani o usando i controller in dotazione, è stato deciso di consentire all'utente quale approccio usare. Questa scelta viene effettuata mediante un prefab che appare subito dopo la schermata start.



La scena che gestisce questa scelta è ***SceneChoiceHandController***. Il bottone con la descrizione ***Mani*** carica la scena che permette all’utente di interagire con il software mediante l’utilizzo delle mani, mentre il bottone ***Controller*** ha la stessa funzione del bottone descritto precedentemente, ma prevede l’utilizzo dei controller da parte dell’utente.

Le scene sono: ***IntroController***, ***Intro***.

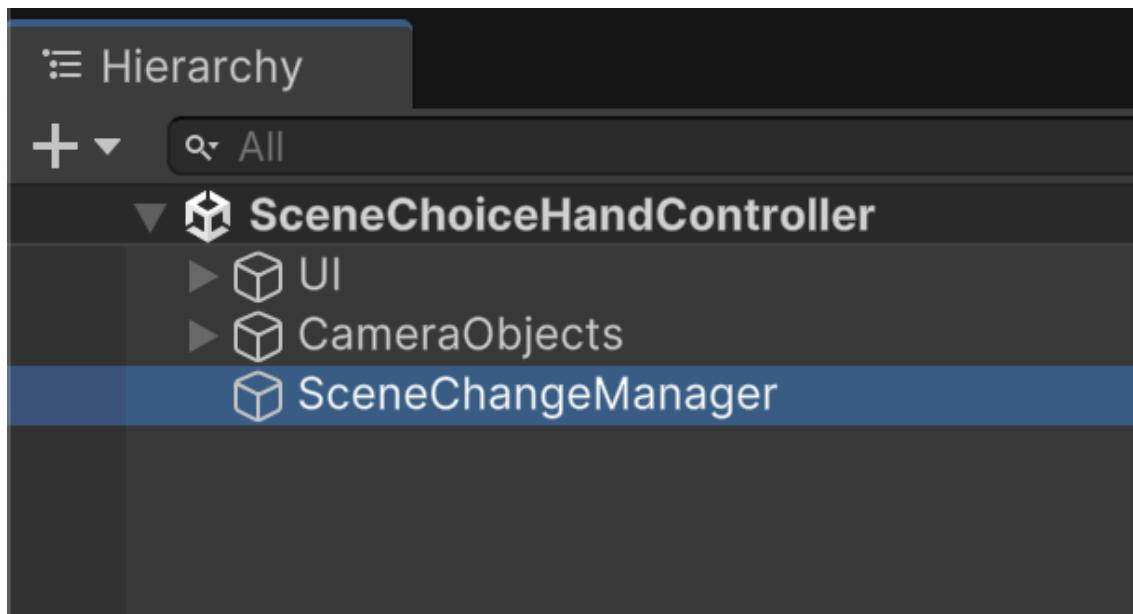
A differenza della scena ***Intro***, dove è presente il building block delle mani e per il loro tracciamento, nella scena di ***IntroController*** vi è inserito solamente il building block che consente di tracciare i controller.

| Used by           | Block              | Status    |
|-------------------|--------------------|-----------|
| Interactions Rig  | 1 Block installed  | Installed |
| Virtual Hands     | 2 Blocks installed | Installed |
| Hand Interactions | 1 Block installed  | Installed |
| Passthrough       | 1 Block installed  | Installed |
| Hand Tracking     | 2 Blocks installed | Installed |

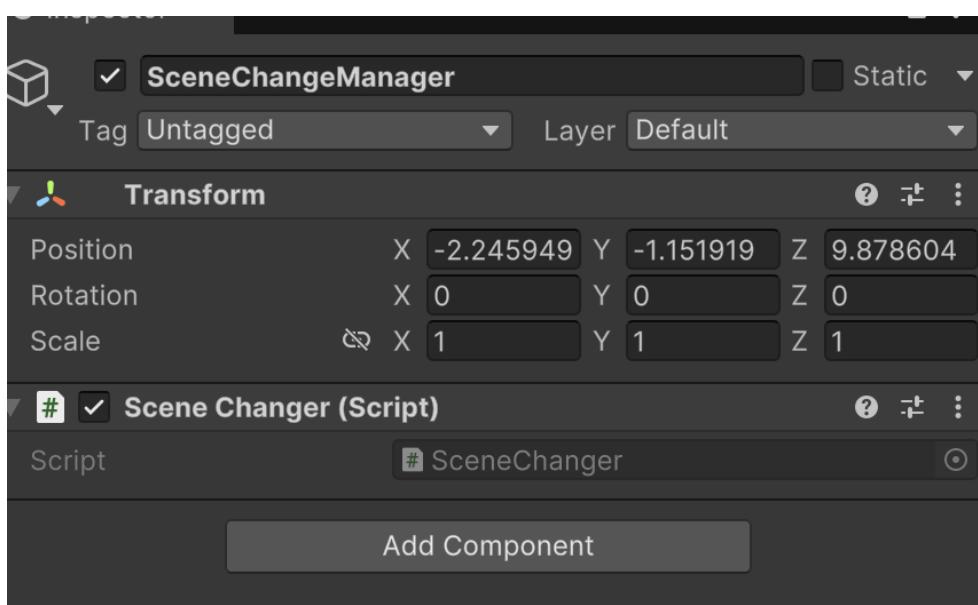
Elementi CameraRig scena ***Intro***.

Elementi CameraRig scena ***IntroController***.

Nella scena *SceneChoiceHandController*, lo script SceneChanger.cs è inserito nell'elemento SceneChangeManager che permette di passare da una scena all'altra in base alla scelta effettuata dall'utente.



L'elemento che gestisce il cambio scena.



Script inserito nell'elemento presente nella gerarchia.

Questo è lo script che gestisce il cambio della scena:

```
using System;
using UnityEngine;
using UnityEngine.SceneManagement;
using TMPro;
using UnityEngine.Serialization;

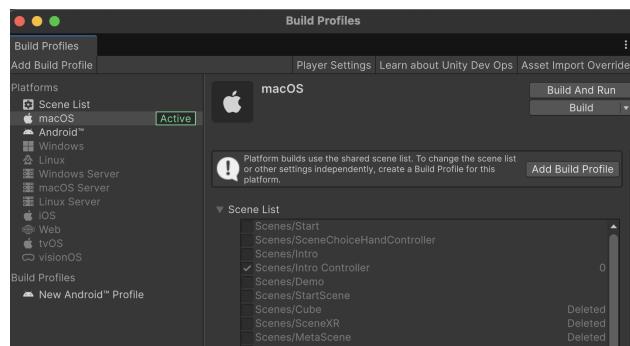
namespace Oculus.Interaction
{
 public class SceneChanger : MonoBehaviour
 {
 //Carica scena successiva, semplice incremento
 public void LoadNextScene()
 {
 int currentScene = SceneManager.GetActiveScene().buildIndex;
 SceneManager.LoadScene(currentScene + 1);
 }

 public void LoadChosenScene(int sceneChoice) // sceneChoice = 2 Mani, sceneChoice = 3 Controller
 {
 SceneManager.LoadScene(sceneChoice);
 }

 //scena iniziale
 public void LoadStartScene()
 {
 SceneManager.LoadScene(0);
 }
 }
}
```

Al suo interno vi sono due 2 metodi: *LoadStartScene*, *LoadNextScene*, *LoadChosencene*.

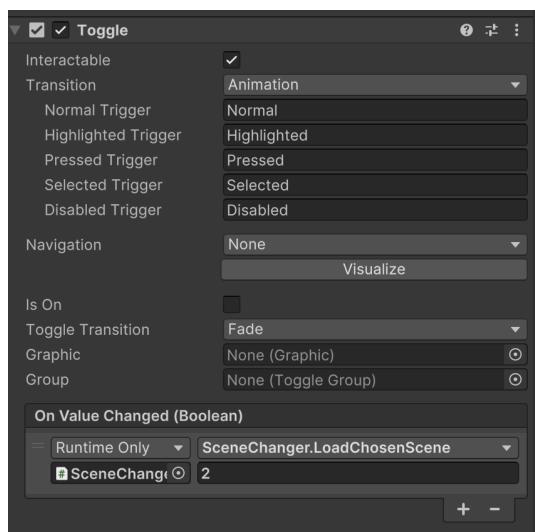
Il primo metodo permette di caricare la scena iniziale, impostata nella finestra di Build Profiles di Unity.



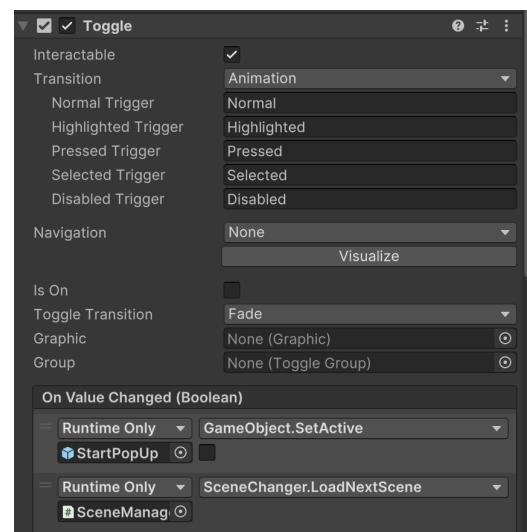
Poiché le scene sono identificate anche da un numero, mediante il metodo LoadScene del package SceneManagement, carichiamo la prima scena in assoluto nella lista, identificata dal numero 0.

Il secondo metodo invece permette di caricare le scene secondo l'ordine stabilito nella finestra di Build Preferences. Una volta individuato l'indice della scena attualmente attiva, si carica la scena successiva mediante il metodo LoadScene. L'ultimo metodo invece contiene un parametro che permette di passare la scena che si desidera caricare. Al metodo LoadScene viene passato infatti questo parametro, consentendo di caricare la scena desiderata.

Questi metodi sono associati ai bottoni dell'UI nel seguente modo:



Associazione bottone *Mani* nella scena SceneChoiceHandController



Associazione bottone *Start* nella scena Start