



UNIVERSITÀ DI PARMA

Dipartimento di Ingegneria e Architettura
Corso di Laurea in Ingegneria Informatica - Informatica Industriale

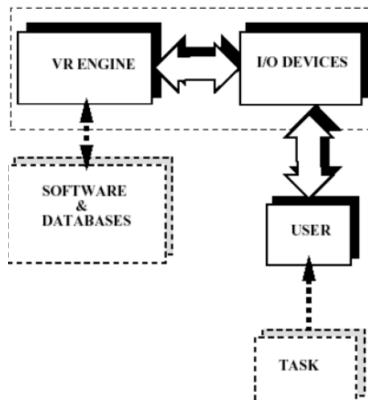
Ultraleap Demo Testing

Vincenzo Fraello (339641) - Giorgia Tedaldi (339642)

ANNO ACCADEMICO 2022/2023

1 Descrizione del progetto

I sistemi di realtà virtuale e aumentata sono dei sistemi che prevedono l'uso di software per la creazione di un ambiente artificiale, che viene poi presentato all'utente in modo tale che possa essere accettato come un ambiente reale. A differenza della realtà aumentata, nell'ambiente virtuale non è presente alcun tipo di elemento appartenente alla scena reale. Il classico schema che permette di rappresentare un sistema di realtà virtuale si compone di:



Le fasi principali che compongono l'obiettivo del progetto sono in linea con le componenti descritte nello schema sopra:

1. **Task:** testare le demo ufficiali di Ultraleap [1];
2. **Software & Databases, VR Engine:** installazione del *plug-in Ultraleap Unity* [2];
3. **I/O:** montaggio del sensore Leap [3] sul visore per VR Oculus Meta Quest 2 [4].

2 Prerequisiti

L'insieme di elementi necessari per il test delle demo:

- Una periferica *Ultraleap Hand Tracking Camera*;
- Software di monitoraggio delle mani *Ultraleap Gemini (V5.2+)*;
- *Unity 2020.3 LTS o 2021.3 LTS*.

2.1 Ultraleap Hand Tracking Camera

Il Controller Leap Motion di Ultraleap è un modulo di tracciamento ottico che cattura il movimento delle mani e delle dita degli utenti al fine di supportare l'interazione naturale tra uomo e contenuti digitali. Il controller può essere collegato a visori per realtà virtuale/aumentata. Alcuni dati dal datasheet ufficiale[5]:

- Zona di interazione: la profondità ideale è fino a 60 cm, ma garantisce fino a un massimo di 80 cm;
- Campo visivo tipico: $140 \times 120^\circ$;
- Telecamere: due telecamere nel vicino infrarosso da 640x240 pixel; distanziate di 40 millimetri l'una dall'altra. Tipicamente operano a 120Hz;
- LED: tre, posti a una certa distanza su entrambi i lati e tra le telecamere, schermati per evitare sovrapposizioni;
- Materiali di costruzione: alluminio e vetro antigraffio;
- Il tracciamento funziona in una vasta gamma di condizioni ambientali;



2.2 Ultraleap Gemini (V5.2+)

Ultraleap Gemini è una piattaforma di tracciamento delle mani che supporta soluzioni di realtà virtuale e aumentata (VR/AR) ed è stata progettata per fornire esperienze informatiche coinvolgenti, realistiche e interattive. È in grado di tracciare contemporaneamente due mani anche in condizioni difficili. Utilizzando la visione artificiale, Gemini crea delle mani virtuali dotate di dita, nocche e palmi. Inoltre consente agli sviluppatori di rilevare qualsiasi cosa: pizzichi, colpetti, prese e tanti altri gesti. Il sistema di visione artificiale di Gemini utilizza una combinazione di algoritmi di deep learning e algoritmi tradizionali di visione artificiale per tracciare le mani con un elevato grado di precisione.

Le principali caratteristiche di Ultraleap Gemini sono[6]:

- Inizializzazione rapida;
- Tiene traccia di mani di diverse dimensioni;
- Supporta l'interazione con entrambe le mani;
- Funziona in ambienti difficili.

2.3 Unity

Unity è un motore grafico di gioco multipiattaforma utilizzato per sviluppare videogiochi, app per dispositivi mobili, simulazioni architettoniche, visualizzazioni 3D e altre applicazioni interattive.

Il software Unity permette agli sviluppatori di creare giochi e applicazioni utilizzando una varietà di linguaggi di programmazione (tra cui C#) e fornisce anche un editor visuale per la creazione di scenari e personaggi in 3D.

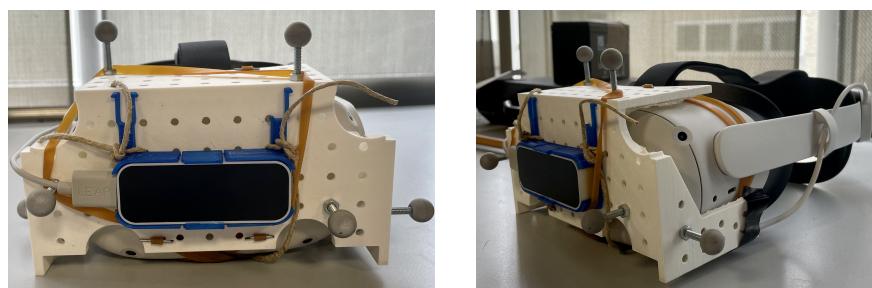
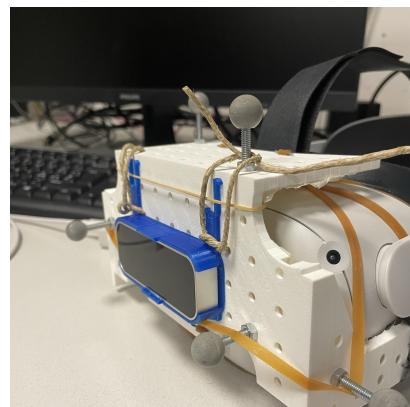
Unity non è esclusivamente un VR-Engine, ma è un motore di gioco versatile che supporta lo sviluppo di applicazioni VR tra altri tipi di giochi e simulazioni. Unity fornisce supporto integrato per una varietà di visori e piattaforme VR, come Oculus Rift.



3 Set up dell'hardware

Gli elementi principali che compongono l'hardware necessario per svolgere il progetto sono:

- Meta Quest 2 - visore VR all-in-one immersivo;
- Leap Motion Controller;
- Maschera in PLA: utilizzata per applicare il Leap motion controller sul visore.



3.1 Meta Quest 2

Il Meta Quest 2 è un visore per la realtà virtuale lanciato sul mercato nel 2020 ed è la seconda generazione di visori di realtà virtuale prodotti dall'azienda Meta. È un dispositivo stand-alone, il che significa che non richiede un PC o un telefono per funzionare. È dotato di tutto ciò che serve per creare un'esperienza di realtà virtuale completa, tra cui schermi ad alta risoluzione, sensori di tracciamento del movimento e cuffie audio integrate.



4 Set-up dell'ambiente virtuale

Il primo passo consiste nel creare un nuovo progetto **3D core** vuoto in **UnityHub**. I passi principali di configurazione del Meta Oculus Quest 2 sono:

- Nella barra principale di Unity selezionare **Edit > Project Settings** poi nella finestra di sinistra premere **XR Plug-in Management**, quindi occorre prima premere **install** e poi selezionare **Oculus** dal menù a destra.
- Accendere il visore e verificare che sia connesso al software **Oculus client**;
- Aprire **Window > Package Manager**, nell'angolo in alto a sinistra selezionare **Packages > Unity Registry**. Scorrere verso il basso e selezionare **XR Plugin Management** e nella finestra di destra premere **import**
- Aprire nuovamente **Window > Package Manager** e cliccare sul pulsante +;
- Selezionare **Add package from git URL** poi inserire nel campo il seguente percorso **com.unity.xr.interaction.toolkit**

I passi principali per la configurazione di Ultraleap Unity Plug-in sono:

- Scaricare e installare il software di handtracking **Ultraleap Gemini (V5.2+)**;
- Scaricare **Unity Modules package** da <https://github.com/ultraleap/UnityPlugin/releases/>;
- Selezionare **Assets > Import Package > Custom Package** dal menù principale di Unity;
- Trovare **Tracking.unitypackage**, **Tracking Preview.unitypackage**, **Tracking Examples.unitypackage**, e **Tracking Preview Examples.unitypackage**

Dopo aver importato il materiale necessario su Unity, tutte le demo sono contenute nel percorso **Assets > ThirdParty > Ultraleap > Tracking > Examples > XR-Examples**.

4.1 XR & XR Plug-in Management & xr.interaction.toolkit

XR è un termine generico che include i seguenti tipi di applicazioni [7]:

- Realtà virtuale (VR): l'applicazione simula un ambiente completamente diverso intorno all'utente;
- Mixed Reality (MR): l'applicazione combina il proprio ambiente con l'ambiente del mondo reale dell'utente e consente loro di interagire;
- Realtà aumentata (AR): l'applicazione sovrappone il contenuto a una visione digitale del mondo reale.

In Unity l'*XR Plug-in Management* è un pacchetto che fornisce una gestione semplice dei plug-in XR. Gestisce e offre assistenza per il caricamento, l'inizializzazione, le impostazioni e il supporto per la creazione dei plug-in XR [8].

Il pacchetto XR Interaction Toolkit [9] è un sistema di interazione di alto livello basato su componenti per la creazione di esperienze VR e AR. Fornisce un framework che rende disponibili le interazioni 3D e UI dagli eventi di input di Unity. Il sistema è basato su un insieme di componenti Interactor e Interactable di base e un Interaction Manager che unisce questi due tipi di componenti.

XR Interaction Toolkit contiene una serie di componenti che supportano le seguenti attività di interazione:

- Attività di input basate sull'uso di controller Cross-platform (es. Meta Quest (Oculus));
- Selezione, presa, passaggio;
- Feedback tattile tramite controller XR.

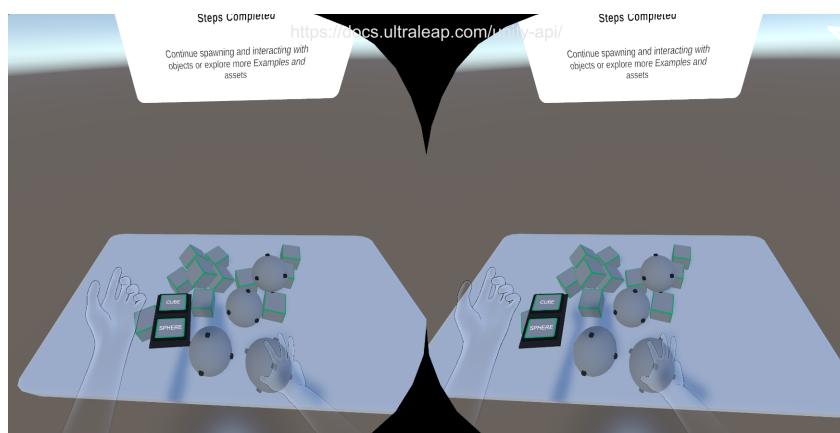
5 XR-Examples

Dopo aver introdotto gli obiettivi e le principali componenti hardware/software del progetto, di seguito sono stati riportati i dettagli di funzionamento delle principali demo fornite da Ultraleap.

5.1 Getting Started

Introduction to Hand Tracking in XR.unity

Questa prima demo fornisce un esempio di alcune delle interazioni possibili con l'*Ultraleap Tracking Plugin per Unity*. La scena contiene un tavolo, alcuni cubi con cui interagire e le mani virtuali. Ci sono due fasi, nella prima è possibile muovere, afferrare, lanciare e spingere i cubi. Nella seconda, ruotando la mano sinistra, è possibile generare nuovi cubi o sfere utilizzando un apposito menù. Durante l'interazione con qualsiasi oggetto, l'utente viene informato della selezione dell'oggetto con un cambiamento del suo colore.



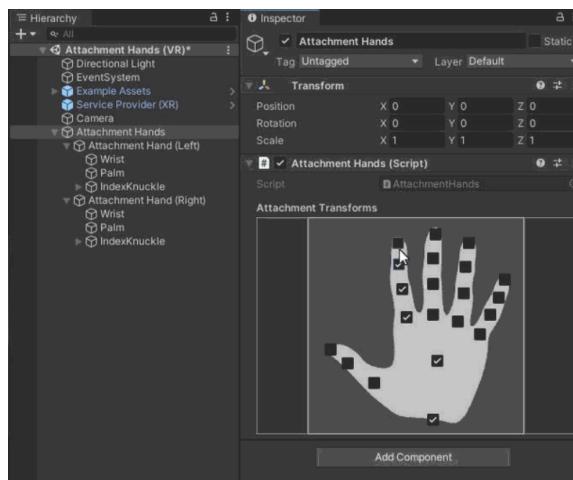
Prefabs utilizzati

Questa prima demo introduttiva è stata fornita con lo scopo di presentare gli elementi (prefab e script) principali che sono contenuti in una generica applicazione di Ultraleap: (i) **Rigged mesh hands**: mani 3D che vengono utilizzate in ambienti virtuali. Il termine *rigged* si riferisce al fatto che le mani sono dotate di un *rig*, ovvero una struttura articolata che permette loro di muoversi e di assumere diverse posizioni in modo realistico. Il termine *mesh* indica invece che le mani sono create come una rete di poligoni 3D, con texture applicate per simulare la pelle e altre caratteristiche; (ii) **Oggetti interattivi**; (iii) **Menu** vincolati alle mani 3D; (iv) **UI 3D** con cui interagire.

In *Unity*, un prefab è un oggetto o un gruppo di oggetti già creati e configurati che possono essere riutilizzati in più parti del progetto senza doverli ricostruire ogni volta da zero. Essenzialmente è un modello di oggetto salvato come file separato e che può essere trascinato e posizionato nella scena di gioco in qualsiasi momento.

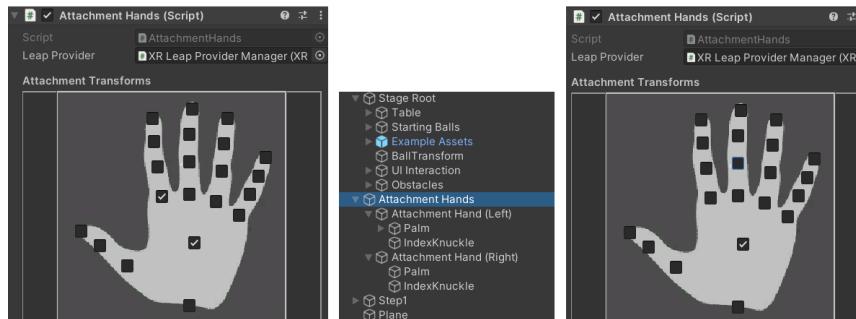
I prefab utilizzati in questa demo sono [1]:

- **XR Leap Provider Manager:** semplifica il processo di *set-up* per l'aggiunta dell'hand tracking in una scena basata sull'XR. Questo prefab gestisce l'impostazione della fonte di tracciamento della mano (es. Leap Motion controller) che l'applicazione riceve. Questo prefab dovrebbe essere utilizzato quando si creano applicazioni che usano il plug-in di Ultraleap per Unity. In particolare, **Service Provider (XR)** è un *child* di XR Leap Provider Manager. La sua funzione è quella di consentire il tracciamento delle mani di Ultraleap tramite il servizio Leap in una scena XR. Gestisce la comunicazione con l'hardware di tracciamento delle mani e trasforma i dati di tracciamento delle mani in modo che le mani vengano visualizzate nella posizione corretta rispetto al visore;
- **Hand Prefab:** è una raccolta di prefab utilizzata per visualizzare i dati di tracciamento della mano utilizzando Rigged mesh hands con vari stili di mano. Si tratta di mesh precostituite e ottimizzate che utilizzano una gerarchia di trasformazioni per muovere le mani (es. *Ghost hands*). Questi prefab consentono di aggiungere delle mani virtuali in una scena. Gli script utilizzati sono: (i) **HandModelBase**: individua automaticamente il *service provider* e si iscrive ai suoi eventi di tipo **OnUpdateFrame**;
- **Interaction Manager:** questo prefab è responsabile della gestione dell'InteractionHand, dell'InteractionController e degli oggetti interattivi. L'Interaction Manager è responsabile della creazione delle rappresentazioni delle mani fisiche dell'utente attraverso l'uso delle Interaction Hands. Queste mani corrispondono il più possibile alle dimensioni delle mani dell'utente in base ai dati di tracciamento e consentono all'utente di interagire gestendone anche la fisica. Durante il primo rilevamento delle mani la forma delle mani virtuali varia gradualmente fino ad adattarsi il più possibile a quelle reali;
- **Interaction Hands:** consente di utilizzare il motore di interazione con il tracciamento della mano;
- **Attachment Hands:** questo prefab consente di collegare facilmente oggetti virtuali a 21 possibili punti-dato tracciati in tempo reale sulla mano. Gli script associati al prefab creano un GameObject nella scena che rappresenta ogni punto tracciato della mano. Tramite l'UI dell'Inspector è possibile scegliere i punti che si desidera visualizzare nella gerarchia sotto ciascuno degli oggetti del prefab (vedi Figura sotto).

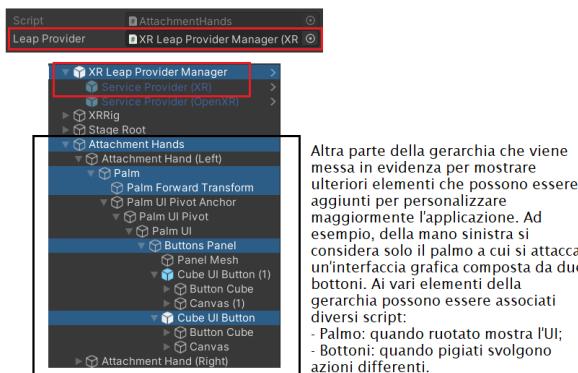


Gli script utilizzati sono:

- **InteractionBehaviours**: componente che permette al GameObject di poter essere pizzicato, afferrato e lanciato. Fornisce anche le API contenenti le callbacks in risposta a eventi di tipo **contac**, **hovering** e **grasping**;
- **AttachmentHands**: aggiungere un GameObject con questo script alla scena per avere una gerarchia di trasformazioni che segue alcuni punti importanti di una mano (segnalati con l’Inspector grafico). L’oggetto AttachmentHands gestisce due oggetti figli, uno per ogni mano del giocatore. Usare l’Inspector per personalizzare i punti che si desidera vedere nella gerarchia sotto i singoli oggetti AttachmentHand;



Dalla Figura sopra è possibile osservare il campo "Leap provider" a cui si passa il GameObject contenente lo script "XR Leap Provider Manager" per la gestione della fonte di tracciamento della mano. Tale GameObject a sua volta contiene due oggetti figli, a uno dei quali è associato lo script "LeapXRSServiceProvider" per la gestione della periferica "Leap Motion".



Altri metodi di questa classe sono:

- * Gio: guardare **Class Leap::Unity::Attachments::AttachmentHands**;
- **AttachmentHand**: classe che fornisce una insieme di metodi utili alla gestione delle mani.
 - * GetBehaviourForPoint: ritorna l'oggetto figlio **AttachmentPointBehaviour** tramite la reference a un singolo AttachmentPointFlags, o null se l'oggetto figlio non esiste;
 - * Gio: guardare **Class Leap::Unity::Attachments::AttachmentHand**;
- **AttachmentHandEnableDisable**: attiva o disattiva il GameObject AttachmentHand quando la mano viene rilevata/non rilevata;
- **AttachmentPointBehaviour**: classe contenitore per la memorizzazione di un riferimento al punto di attacco a cui corrisponde la trasformazione all'interno di una AttachmentHand. Contiene anche le mappature da un singolo flag AttachmentPointFlags all'osso corrispondente di una Leap.Hand; queste mappature sono accessibili staticamente tramite GetLeapHandPointData().

Esempio di gestione del mapping tra AttachmentPointFlags e Leap.Hand

```
public void SetTransformUsingHand(Leap.Hand hand)
{
    if (hand == null)
    {
        return;
    }

    Vector3 position = Vector3.zero;
    Quaternion rotation = Quaternion.identity;

    GetLeapHandPointData(hand, this.attachmentPoint, out position, out rotation);

    this.transform.position = position;
    this.transform.rotation = rotation;
}
```

```
public static void GetLeapHandPointData(Leap.Hand hand, AttachmentPointFlags singlePoint, out Vector3 position, out Quaternion rotation)
{
    position = Vector3.zero;
    rotation = Quaternion.identity;

    if (singlePoint != AttachmentPointFlags.None && !singlePoint.IsSinglePoint())
    {
        Debug.LogError("Cannot get attachment point data for an AttachmentPointFlags argument consisting of more than one set flag.");
        return;
    }

    switch (singlePoint)
    {
        case AttachmentPointFlags.None:
            return;

        case AttachmentPointFlags.Wrist:
            position = hand.WristPosition;
            rotation = hand.Arm.Basis.rotation;
            break;

        case AttachmentPointFlags.Palm:
            position = hand.PalmPosition;
            rotation = hand.Basis.rotation;
            break;

        case AttachmentPointFlags.ThumbProximalJoint:
            position = hand.Fingers[0].bones[1].NextJoint;
            rotation = hand.Fingers[0].bones[2].Rotation;
            break;

        case AttachmentPointFlags.ThumbDistalJoint:
            position = hand.Fingers[0].bones[2].NextJoint;
            rotation = hand.Fingers[0].bones[3].Rotation;
            break;

        case AttachmentPointFlags.ThumbTip:
            position = hand.Fingers[0].bones[3].NextJoint;
            rotation = hand.Fingers[0].bones[3].Rotation;
            break;

        case AttachmentPointFlags.IndexKnuckle:
            position = hand.Fingers[1].bones[0].NextJoint;
            rotation = hand.Fingers[1].bones[1].Rotation;
            break;

        case AttachmentPointFlags.IndexMiddleJoint:
            position = hand.Fingers[1].bones[1].NextJoint;
            rotation = hand.Fingers[1].bones[2].Rotation;
            break;

        case AttachmentPointFlags.IndexDistalJoint:
            position = hand.Fingers[1].bones[2].NextJoint;
            rotation = hand.Fingers[1].bones[3].Rotation;
            break;

        case AttachmentPointFlags.IndexTip:
            position = hand.Fingers[1].bones[3].NextJoint;
            rotation = hand.Fingers[1].bones[3].Rotation;
            break;

        case AttachmentPointFlags.MiddleKnuckle:
            position = hand.Fingers[2].bones[0].NextJoint;
            rotation = hand.Fingers[2].bones[1].Rotation;
            break;

        case AttachmentPointFlags.MiddleMiddleJoint:
            position = hand.Fingers[2].bones[1].NextJoint;
            rotation = hand.Fingers[2].bones[2].Rotation;
            break;

        case AttachmentPointFlags.MiddleDistalJoint:
            position = hand.Fingers[2].bones[2].NextJoint;
            rotation = hand.Fingers[2].bones[3].Rotation;
            break;

        case AttachmentPointFlags.MiddleTip:
            position = hand.Fingers[2].bones[3].NextJoint;
            rotation = hand.Fingers[2].bones[3].Rotation;
            break;

        case AttachmentPointFlags.RingKnuckle:
            position = hand.Fingers[3].bones[0].NextJoint;
            rotation = hand.Fingers[3].bones[1].Rotation;
            break;

        case AttachmentPointFlags.RingMiddleJoint:
            position = hand.Fingers[3].bones[1].NextJoint;
            rotation = hand.Fingers[3].bones[2].Rotation;
            break;

        case AttachmentPointFlags.RingDistalJoint:
            position = hand.Fingers[3].bones[2].NextJoint;
            rotation = hand.Fingers[3].bones[3].Rotation;
            break;

        case AttachmentPointFlags.RingTip:
            position = hand.Fingers[3].bones[3].NextJoint;
            rotation = hand.Fingers[3].bones[3].Rotation;
            break;

        case AttachmentPointFlags.PinkyKnuckle:
            position = hand.Fingers[4].bones[0].NextJoint;
            rotation = hand.Fingers[4].bones[1].Rotation;
            break;

        case AttachmentPointFlags.PinkyMiddleJoint:
            position = hand.Fingers[4].bones[1].NextJoint;
            rotation = hand.Fingers[4].bones[2].Rotation;
            break;

        case AttachmentPointFlags.PinkyDistalJoint:
            position = hand.Fingers[4].bones[2].NextJoint;
            rotation = hand.Fingers[4].bones[3].Rotation;
            break;

        case AttachmentPointFlags.PinkyTip:
            position = hand.Fingers[4].bones[3].NextJoint;
            rotation = hand.Fingers[4].bones[3].Rotation;
            break;
    }
}
```

```
case AttachmentPointFlags.MiddleKnuckle:
    position = hand.Fingers[2].bones[0].NextJoint;
    rotation = hand.Fingers[2].bones[1].Rotation;
    break;
case AttachmentPointFlags.MiddleMiddleJoint:
    position = hand.Fingers[2].bones[1].NextJoint;
    rotation = hand.Fingers[2].bones[2].Rotation;
    break;
case AttachmentPointFlags.MiddleDistalJoint:
    position = hand.Fingers[2].bones[2].NextJoint;
    rotation = hand.Fingers[2].bones[3].Rotation;
    break;
case AttachmentPointFlags.MiddleTip:
    position = hand.Fingers[2].bones[3].NextJoint;
    rotation = hand.Fingers[2].bones[3].Rotation;
    break;
case AttachmentPointFlags.RingKnuckle:
    position = hand.Fingers[3].bones[0].NextJoint;
    rotation = hand.Fingers[3].bones[1].Rotation;
    break;
case AttachmentPointFlags.RingMiddleJoint:
    position = hand.Fingers[3].bones[1].NextJoint;
    rotation = hand.Fingers[3].bones[2].Rotation;
    break;
case AttachmentPointFlags.RingDistalJoint:
    position = hand.Fingers[3].bones[2].NextJoint;
    rotation = hand.Fingers[3].bones[3].Rotation;
    break;
case AttachmentPointFlags.RingTip:
    position = hand.Fingers[3].bones[3].NextJoint;
    rotation = hand.Fingers[3].bones[3].Rotation;
    break;
case AttachmentPointFlags.PinkyKnuckle:
    position = hand.Fingers[4].bones[0].NextJoint;
    rotation = hand.Fingers[4].bones[1].Rotation;
    break;
case AttachmentPointFlags.PinkyMiddleJoint:
    position = hand.Fingers[4].bones[1].NextJoint;
    rotation = hand.Fingers[4].bones[2].Rotation;
    break;
case AttachmentPointFlags.PinkyDistalJoint:
    position = hand.Fingers[4].bones[2].NextJoint;
    rotation = hand.Fingers[4].bones[3].Rotation;
    break;
case AttachmentPointFlags.PinkyTip:
    position = hand.Fingers[4].bones[3].NextJoint;
    rotation = hand.Fingers[4].bones[3].Rotation;
    break;
```

Un esempio di codice che attacca un cubo sotto l'indice. Può essere facilmente adattato per usi simili, ad esempio attaccare un cubo in corrispondenza di ognuno dei 21 punti-dato (vedi Figura sotto):

```

1  using Leap.Unity.Attachments;
2  using UnityEngine;
3
4  public class Example_AttachmentHand : MonoBehaviour
5  {
6      //The Attachment Hand you want to attach the Cube to
7      [SerializeField] AttachmentHand attachmentHand;
8
9      //The Attachment Point you want to attach the cube to
10     [SerializeField] AttachmentPointFlags attachmentPoint = AttachmentPointFlags.IndexTip;
11
12     //The Cube that we make
13     private GameObject cube;
14
15     private void Start()
16     {
17         //Make a basic cube
18         cube = GameObject.CreatePrimitive(PrimitiveType.Cube);
19         //Adjust the scale of the cube
20         cube.transform.localScale *= 0.025f;
21     }
22
23     private void Update()
24     {
25         //Get the attachment point behaviour of the desired finger and bone
26         AttachmentPointBehaviour attachmentPointBehaviour = attachmentHand.GetBehaviourForPoint(attachmentPoint);
27
28         //If the attachment point exists, make it the parent of the cube and ensure the cube is transformed correctly below it
29         if(attachmentPointBehaviour != null)
30         {
31             cube.transform.parent = attachmentPointBehaviour.transform;
32             cube.transform.localPosition = Vector3.zero;
33             cube.transform.localRotation = Quaternion.identity;
34         }
35     }
36 }

```

```

/// <summary>
/// Returns the AttachmentPointBehaviour child object of this AttachmentHand given a
/// reference to a single AttachmentPointFlags flag, or null if there is no such child object.
/// </summary>
6 references
public AttachmentPointBehaviour GetBehaviourForPoint(AttachmentPointFlags singlePoint)
{
    AttachmentPointBehaviour behaviour = null;

    switch (singlePoint)
    {
        case AttachmentPointFlags.None: break;

        case AttachmentPointFlags.Wrist: behaviour = wrist; break;
        case AttachmentPointFlags.Palm: behaviour = palm; break;

        case AttachmentPointFlags.ThumbProximalJoint: behaviour = thumbProximalJoint; break;
        case AttachmentPointFlags.ThumbDistalJoint: behaviour = thumbDistalJoint; break;
        case AttachmentPointFlags.ThumbTip: behaviour = thumbTip; break;

        case AttachmentPointFlags.IndexKnuckle: behaviour = indexKnuckle; break;
        case AttachmentPointFlags.IndexMiddleJoint: behaviour = indexMiddleJoint; break;
        case AttachmentPointFlags.IndexDistalJoint: behaviour = indexDistalJoint; break;
        case AttachmentPointFlags.IndexTip: behaviour = indexTip; break;

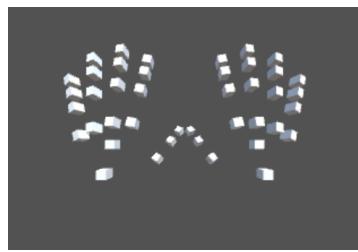
        case AttachmentPointFlags.MiddleKnuckle: behaviour = middleKnuckle; break;
        case AttachmentPointFlags.MiddleMiddleJoint: behaviour = middleMiddleJoint; break;
        case AttachmentPointFlags.MiddleDistalJoint: behaviour = middleDistalJoint; break;
        case AttachmentPointFlags.MiddleTip: behaviour = middleTip; break;

        case AttachmentPointFlags.RingKnuckle: behaviour = ringKnuckle; break;
        case AttachmentPointFlags.RingMiddleJoint: behaviour = ringMiddleJoint; break;
        case AttachmentPointFlags.RingDistalJoint: behaviour = ringDistalJoint; break;
        case AttachmentPointFlags.RingTip: behaviour = ringTip; break;

        case AttachmentPointFlags.PinkyKnuckle: behaviour = pinkyKnuckle; break;
        case AttachmentPointFlags.PinkyMiddleJoint: behaviour = pinkyMiddleJoint; break;
        case AttachmentPointFlags.PinkyDistalJoint: behaviour = pinkyDistalJoint; break;
        case AttachmentPointFlags.PinkyTip: behaviour = pinkyTip; break;
    }

    return behaviour;
}

```



Un cubo in ogni punto dato

Classe Leap.Hand

Questa classe viene utilizzata per riportare le caratteristiche fisiche di una mano rilevata. I dati di hand tracking includono: posizione e velocità del palmo; i vettori per la normale del palmo e la direzione delle dita e gli elenchi delle dita collegate. Generalmente, l'oggetto Hand viene creato a partire dai dati rilevati in un frame ricevuti dal provider service. Alcuni metodi e membri (passati al costruttore per creare l'oggetto Hand) sono:

- **Hand.Finger()**: utilizzato per ottenere un oggetto Finger collegato alla mano, utilizzando un valore ID ottenuto da un fotogramma precedente;
- **bool Equals (Hand other)**: utilizzato per comparare due oggetti Hand;
- **override string ToString ()**: utilizzato per ottenere una descrizione comprensibile all'essere umano dell'oggetto Hand;
- **int Id**: identificativo univoco assegnato all'oggetto Hand, il cui valore rimane invariato in tutti i fotogrammi consecutivi finché la mano tracciata rimane visibile. Se il tracciamento viene perso (ad esempio, quando una mano viene occlusa da un'altra mano o quando viene ritirata o raggiunge il bordo del campo visivo del controller Leap Motion), il software Leap Motion può assegnare un nuovo ID quando rileva la mano in un fotogramma futuro;
- **Frame.Hand()**: utilizzato per trovare l'oggetto Hand nei fotogrammi futuri sfruttando il valore dell'ID;
- **List<Finger> Fingers**: lista di oggetti Finger rilevati nel fotogramma che sono collegati alla mano, in ordine dal pollice al mignolo. La lista non può essere vuota;
- **Vector3 PalmPosition**: campo utilizzato per indicare il centro del palmo della mano;
- **Vector3 PalmVelocity**: campo utilizzato per indicare la velocità di cambiamento della posizione del palmo della mano;
- **Vector3 PalmNormal**: campo utilizzato per indicare il vettore normale al palmo. Se la mano è piatta, questo vettore punta verso il basso, o "fuori" dalla superficie anteriore del palmo (cioè un vettore ortogonale al palmo). È possibile utilizzare il vettore normale del palmo per calcolare l'angolo di rotazione del palmo rispetto al piano orizzontale;
- **Vector3 Direction**: campo utilizzato per indicare la direzione dalla posizione del palmo verso le dita. La direzione è espressa come un vettore unitario che punta nella stessa direzione della linea diretta dalla posizione del palmo alle dita. È possibile utilizzare il vettore direzione del palmo per calcolare gli angoli di pitch e yaw del palmo rispetto al piano orizzontale;
- **Quaternion Rotation**: campo utilizzato per indicare la rotazione della mano come quaternione;
- **float GrabStrength**: campo utilizzato per indicare la forza di una posa di presa (Grabpose) della mano. La forza è 0 per una mano aperta e passa a 1 quando viene riconosciuta una posa di presa;

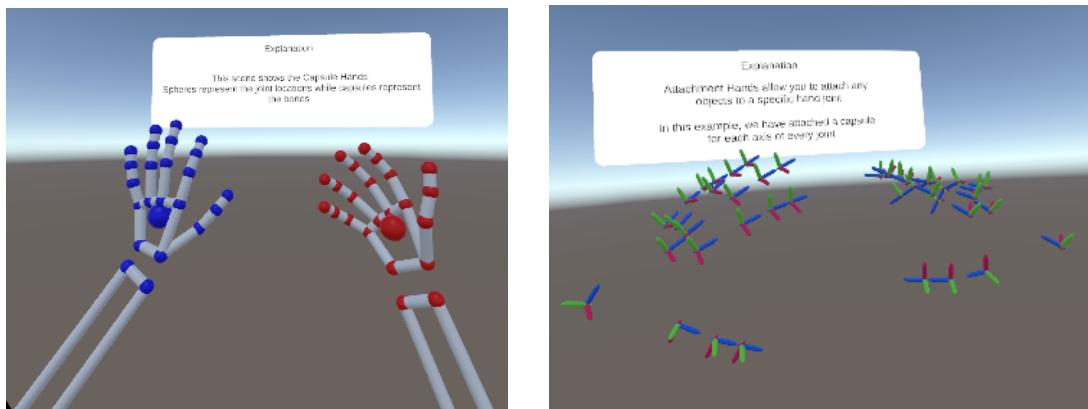
- **float PinchStrength:** campo utilizzato per indicare la forza del pinch di una posa di pinch-hand. La forza è pari a 0 per una mano aperta e 1 a uno quando viene riconosciuta una posa di pinch-hand. Il pizzico può essere effettuato tra il pollice e qualsiasi altro dito della stessa mano (es. l'indice).
- **float PinchDistance:** campo che identifica la distanza tra il pollice e l'indice in una posa di pinch-hand. La distanza è calcolata osservando la distanza più breve tra le ultime due falangi del pollice e quelle dell'indice;
- **Vector3 WristPosition:** campo utilizzato per indicare la posizione del polso della mano;
- **Arm Arm:** campo utilizzato per indicare il braccio a cui è collegata la mano;
- **LeapTransform Basis { get; set; }:** utilizzato per ottenere o settare la trasformazione della mano;
- **bool IsRight { get; set; }:** identifica se la mano è quella destra;

5.2 Building Blocks

5.2.1 Basics

Attachment Hands.unity & Capsule Hands.unity

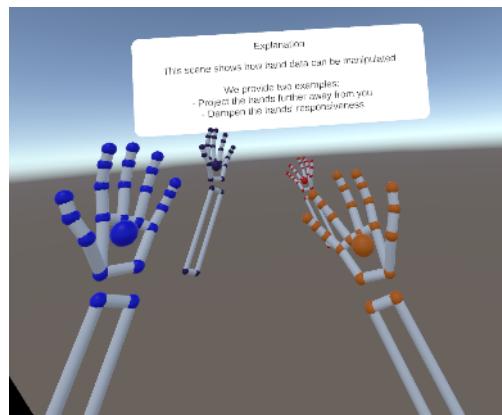
Il tracciamento delle mani di Ultraleap mappa ciascuna mano in 21 *data-point* a cui è possibile collegare più oggetti virtuali. In questa demo a ciascun punto è associata l'origine di un sistema di riferimento in cui i colori degli assi corrispondono al *Rotation Gizmo* in alto a destra della scena Unity.



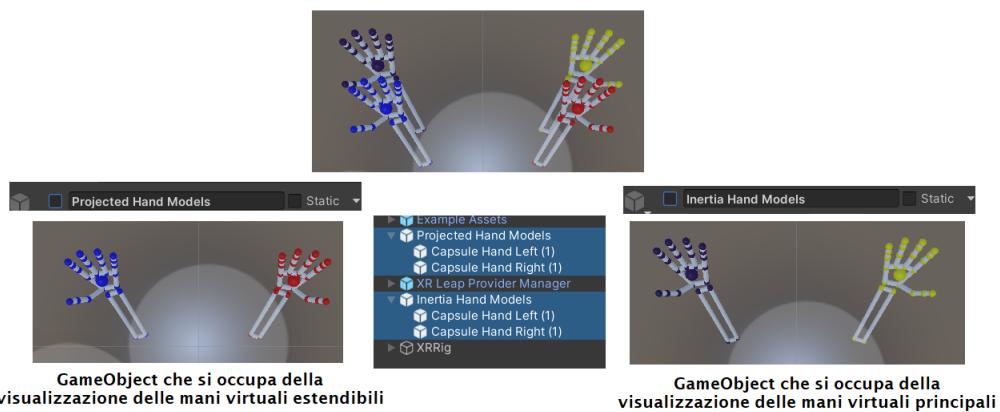
In particolare, i prefab utilizzati in queste demo sono **XR Leap Provider Manager** (5.1) e **Attachment Hands** (5.2.1).

Manipulating Hand Data.unity

Questa demo mostra un esempio di *mappatura non lineare per la manipolazione diretta in VR* che può essere usata per risolvere problemi come raggiungere qualcosa che le mani reali non possono raggiungere.

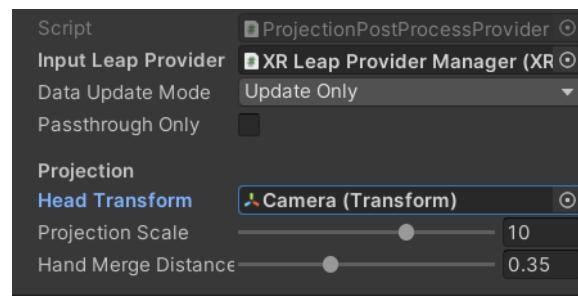


I prefab utilizzati in questa demo sono: **XR Leap Provider Manager** (5.1), **Inertia Hand Models** e **Projected Hand Models** (5.2.1).



PostProcessProvider

Un post process provider è un fornitore di dati di tracciamento delle mani generati elaborando i dati di tracciamento del frame precedente. Ad esempio, è possibile generare delle nuove mani, a partire dai dati di tracciamento, in una posizione più lontana rispetto a quella reale se le mani reali sono estese (funzione dello script `ProjectionPostProcessProvider` utilizzato in questa demo). In particolare, alle mani proiettate si applica la trasformazione Camera fornita come parametro dall'Inspector.



Un esempio di codice che permette di capire il concetto in maniera più semplice è il seguente:

```

1  using Leap;
2  using Leap.Unity;
3  using UnityEngine;
4
5  0 references
6  public class ExamplePostProcessProvider : PostProcessProvider
7  {
8      1 reference
9      public Vector3 handPosition = Vector3.up;
10
11     0 references
12     public override void ProcessFrame(ref Frame inputFrame)
13     {
14         //Get the left hand from the frame
15         Hand leftHand = inputFrame.GetHand(Chirality.Left);
16
17         //Transform the left hand so its position is driven by a variable in the inspector
18         leftHand.SetTransform(handPosition, leftHand.Rotation.ToQuaternion());
19     }
20 }

```

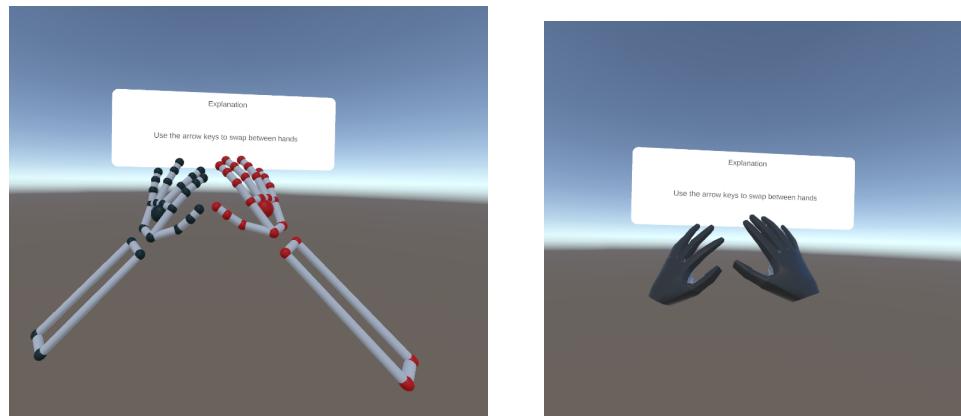
Inertia Hand Models & Projected Hand Models

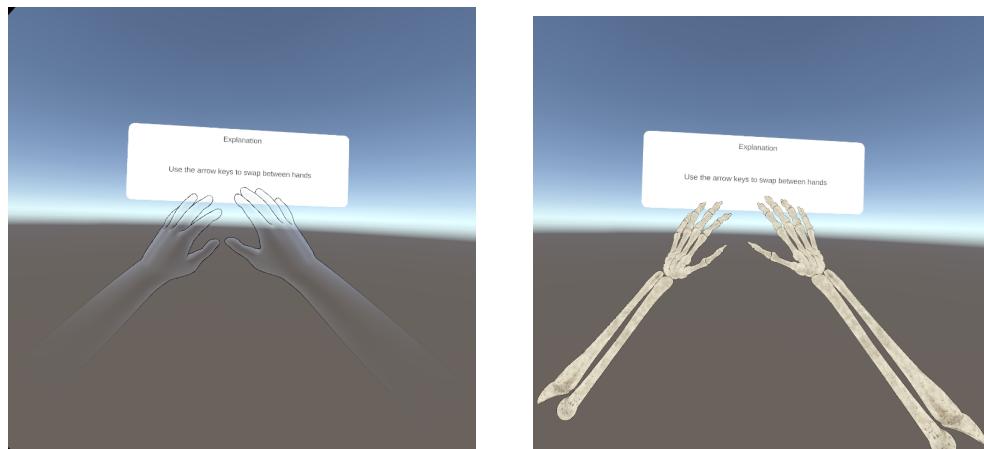
GameObject contenente due oggetti figli (uno per ogni mano) a cui è associato lo script **CapsuleHand**. Questo script è un modello di Leap-Hand di base che genera un set di sfere e cilindri per il rendering delle mani utilizzando i dati della mano Leap. La mano viene generata in modo dinamico, si ridimensiona in base alle dimensioni della mano dell'utente.

5.2.2 Pre-rigged Hands

Rigged Hands (standard).unity

In questa demo è possibile osservare degli esempi di diversi prototipi di mani virtuali che è possibile attaccare ai 21 punti mappati dall'*hand-tracking* di Ultraleap [Sezione 5.2.1]. Utilizzando le **frecce** destra e sinistra della tastiera è possibile osservare i diversi **modelli** di mani.





I prefab utilizzati in questa demo sono **XR Leap Provider Manager** (5.1) e **Hand Prefabs** (5.1).

Gli script utilizzati sono:

- **HandEnableDisable**: Questo script fornisce un componente che si occupa di gestire l’abilitazione/disabilitazione del tracking delle mani.

```
public class HandEnableDisable : HandTransitionBehavior
{
    [Tooltip("When enabled, freezes the hand in its current active state")]
    public bool FreezeHandState = false;

    protected override void Awake()
    {
        // Suppress Warnings Related to Kinematic Rigidbodies not supporting Continuous Collision Detection
        Rigidbody[] bodies = GetComponentsInChildren<Rigidbody>();
        foreach (Rigidbody body in bodies)
        {
            if (body.isKinematic && body.collisionDetectionMode == CollisionDetectionMode.Continuous)
            {
                body.collisionDetectionMode = CollisionDetectionMode.ContinuousSpeculative;
            }
        }

        base.Awake();
    }

    protected override void HandReset()
    {
        if (FreezeHandState)
        {
            return;
        }

        gameObject.SetActive(true);
    }

    protected override void HandFinish()
    {
        if (FreezeHandState)
        {
            return;
        }

        gameObject.SetActive(false);
    }
}
```

- **HandModelManager**: questo script permette la gestione dei vari oggetti **HandModels** presenti nella scena. In sostanza, è possibile **registrare** diversi tipi di **HandModelPair** (una coppia di **HandModel**, uno per la mano destra e uno per la sinistra) che vengono inseriti all’interno di una **lista**. Ogni coppia è registrata in modo **univoco** e possiede un **indice** utilizzabile per accedervi. I metodi principali utilizzati per l’esempio sono:

- **EnableHandModelPair**: il metodo riceve in ingresso l’**indice** dell’**HandModelPair** che si vuole attivare in quel momento, lo **abilita** e contemporaneamente **disabilita** tutti gli altri modelli.

```
/// <summary>
/// Enable a hand model pair in the scene
/// </summary>
/// <param name="index">The index of the pair to enable</param>
/// <param name="disableOtherHandModels">If true, disable all other active hands</param>
public void EnableHandModelPair(int index, bool disableOtherHandModels = true)
{
    EnableHandModel(Chirality.Left, index, disableOtherHandModels);
    EnableHandModel(Chirality.Right, index, disableOtherHandModels);
}
```

La fase di attivazione/disattivazione fa uso del metodo **EnableHandModel** che viene richiamato **due volte** (una per mano).

- **EnableHandModel**: il metodo viene inizialmente richiamato da **EnableHandModelPair** con parametri: **chirality**, **index** e **disableOtherHandModelsOfSameChirality = true**. Questo recupera l’oggetto **handModel** a seconda dell’indice che gli è stato passato; a sua volta richama un metodo omonimo **EnableHandModel** a cui passa come parametri: la **chiralità**, l’oggetto **handModel** trovato e la variabile **disableOtherHandModelsOfSameChirality**. A questo punto l’oggetto di tipo **HandModelPair** viene attivato utilizzando lo script **HandEnableDisable** descritto in precedenza; lo stesso script si occupa di disattivare gli altri modelli.

```
/// <summary>
/// Enable a single hand model in the scene, based on chirality
/// </summary>
/// <param name="chirality">The chirality to enable</param>
/// <param name="index">The index of the hand model to enable</param>
/// <param name="disableOtherHandModelsOfSameChirality">If true, disable all other active hand models of the same chirality</param>
public void EnableHandModel(Chirality chirality, int index, bool disableOtherHandModelsOfSameChirality = true)
{
    HandModelPair handModelPair = HandModelPairs[index];
    EnableHandModel(chirality, handModelPair, disableOtherHandModelsOfSameChirality);
}
```

```
/// <summary>
/// Enable a single hand model in the scene, based on chirality
/// </summary>
/// <param name="chirality">The chirality to enable</param>
/// <param name="handModelPair">The pair to enable</param>
/// <param name="disableOtherHandModelsOfSameChirality">If true, disable all other active hand models of the same chirality</param>
private void EnableHandModel(Chirality chirality, HandModelPair handModelPair, bool disableOtherHandModelsOfSameChirality)
{
    if (disableOtherHandModelsOfSameChirality)
    {
        DisableAllHandModelsByChirality(chirality);
    }

    if (chirality == Chirality.Left)
    {
        if (handModelPair.LeftEnableDisable != null)
        {
            handModelPair.LeftEnableDisable.FreezeHandState = false;
        }

        if (handModelPair.Left != null)
        {
            if (handModelPair.Left.IsTracked)
            {
                handModelPair.Left.gameObject.SetActive(true);
            }
            OnHandModelEnabled?.Invoke(handModelPair.Left);
        }
    }
    else
    {
        if (handModelPair.RightEnableDisable != null)
        {
            handModelPair.RightEnableDisable.FreezeHandState = false;
        }

        if (handModelPair.Right != null)
        {
            if (handModelPair.Right.IsTracked)
            {
                handModelPair.Right.gameObject.SetActive(true);
            }
            OnHandModelEnabled?.Invoke(handModelPair.Right);
        }
    }
}
```

- **CycleHandPairs**: lo script inizializza una variabile **currentHandID = 0** per mostrare il primo **HandModelPair** registrato. A questo punto la funzione **Update** si occupa della gestione di due eventi:

- **Input.GetKeyUp(KeyCode.RightArrow)**: la variabile **currentHandID** viene **incrementata** e viene richiamata la funzione **handModelManager.EnableHandModelPair** discussa in precedenza.
- **Input.GetKeyUp(KeyCode.LeftArrow)**: la variabile **currentHandID** viene **decrementata** e viene richiamata la funzione **handModelManager.EnableHandModelPair** discussa in precedenza.

```

public class CycleHandPairs : MonoBehaviour
{
    [SerializeField]
    private HandModelManager handModelManager;
    private int currentHandID;

    // Use this for initialization
    void Start()
    {
        if (handModelManager == null)
        {
            handModelManager = FindObjectOfType<HandModelManager>();
            if (handModelManager == null)
            {
                Debug.LogWarning("CycleHandPairs needs a HandModelManager in the scene");
                return;
            }
        }
        currentHandID = 0;
        handModelManager.EnableHandModelPair(currentHandID, disableOtherHandModels: true);
    }
}

```

```

void Update()
{
    if (handModelManager == null)
    {
        return;
    }

    if (Input.GetKeyUp(KeyCode.RightArrow))
    {
        NextHandSet();
    }

    if (Input.GetKeyUp(KeyCode.LeftArrow))
    {
        PreviousHandSet();
    }
}

private void NextHandSet()
{
    currentHandID++;
    if (currentHandID > handModelManager.HandModelPairs.Count - 1) currentHandID = 0;
    handModelManager.EnableHandModelPair(currentHandID, disableOtherHandModels: true);
}

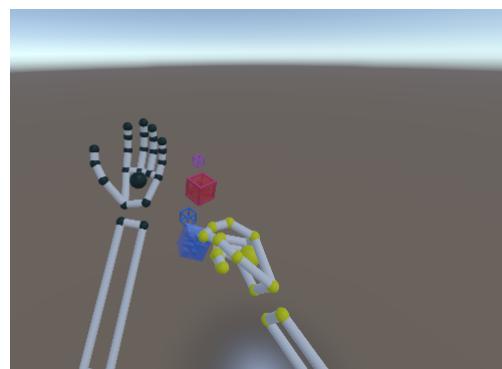
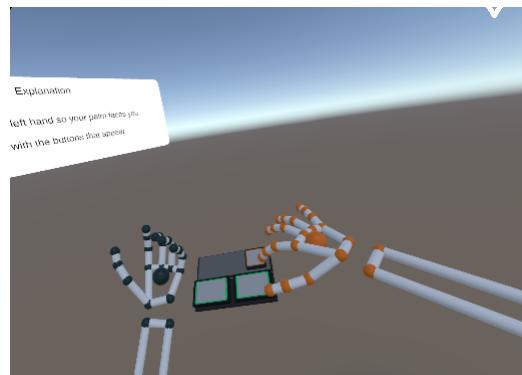
private void PreviousHandSet()
{
    currentHandID--;
    if (currentHandID < 0) currentHandID = handModelManager.HandModelPairs.Count - 1;
    handModelManager.EnableHandModelPair(currentHandID, disableOtherHandModels: true);
}

```

5.2.3 Menus & UI

Basic UI.unity - Hand Menu.unity - Dynamic UI.unity

Queste tre demo mostrano un esempio di un diverso tipo di interazione nell'ambiente virtuale basato sull'uso di oggetti che simulano dei menù manipolabili mediante l'uso di *slider*, *anchor* e pulsanti.



- **Basic UI:** in questa demo è possibile interagire con l'interfaccia grafica in diversi modi. I prefab utilizzati per la costruzione della scena sono: **XR Leap Provider Manager** (5.1), **Interaction Manager** (5.1) e **Interaction Hands** (5.1).

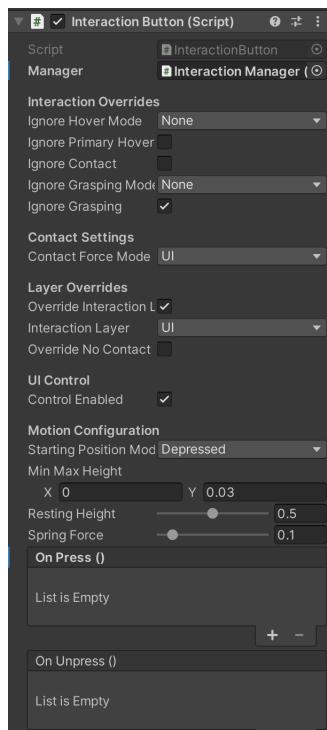
Gli script utilizzati, invece, sono:

- **IgnoreCollisionsInChildren:** viene utilizzato per fare in modo che tutti gli elementi UI non collidano tra di loro.

```
public class IgnoreCollisionsInChildren : MonoBehaviour
{
    void Start()
    {
        IgnoreCollisionsInChildrenOf(this.transform);
    }

    public static void IgnoreCollisionsInChildrenOf(Transform t, bool ignore = true)
    {
        Collider[] colliders = t.GetComponentsInChildren<Collider>();
        for (int i = 0; i < colliders.Length; i++)
        {
            for (int j = 0; j < colliders.Length; j++)
            {
                if (i == j) continue;
                Physics.IgnoreCollision(colliders[i], colliders[j], ignore);
            }
        }
    }
}
```

- **InteractionButton:** permette la gestione dell'interazione **fisica** con il bottone in VR. Fornisce metodi per gestire gli eventi di **onPress** e **onUnpress**. Per definire un'azione da compiere quando si verifica tale evento è sufficiente trascinare nell'**inspector**, all'interno della finestra relativa allo script, un **Object** a sua volta collegato a uno script in cui è definita la **funzione** da richiamare.



Per la gestione degli eventi **onPress** e **onUnpress** è possibile utilizzare metodi che restituiscono lo **stato** del bottone in quel momento:

```

    /// <summary> Gets whether the button is currently held down. </summary>
    public bool isPressed { get { return _isPressed; } }

    /// <summary>
    /// Gets whether the button was pressed during this Update frame.
    /// </summary>
    public bool pressedThisFrame { get { return _pressedThisFrame; } }

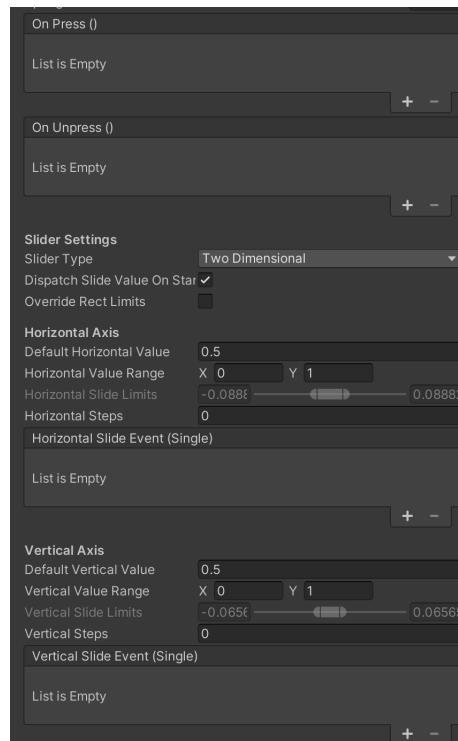
    /// <summary>
    /// Gets whether the button was unpressed this frame.
    /// </summary>
    public bool unpressedThisFrame { get { return _unpressedThisFrame; } }

    /// <summary>
    /// Gets a normalized value between 0 and 1 based on how depressed the button
    /// currently is relative to its maximum depression. 0 represents a button fully at
    /// rest or pulled out beyond its resting position; 1 represents a fully-pressed
    /// button.
    /// </summary>
    public float pressedAmount { get { return _pressedAmount; } }

    /// <summary>
    /// Returns the local position of this button when it is able to relax into its target
    /// position.
    /// </summary>
    public virtual Vector3 RelaxedLocalPosition
    {
        get
        {
            return initialLocalPosition
                   + Vector3.back * Mathf.Lerp(minMaxHeight.x, minMaxHeight.y, restingHeight);
        }
    }
}

```

- **InteractionSlider**: definisce uno **slider** attivato dalla **fisica**. Lo **sliding** è triggerato **premendo fisicamente** l’oggetto, comprimendolo, e poi muovendolo. Incrementando i limiti orizzontali e verticali si può creare uno slider 1D o 2D. **InteractionSlider** è una **sottoclasse** di **InteractionButton** da cui, quindi, eredita anche la gestione degli eventi **onPress** e **onUnpress**. Vengono definiti due ulteriori eventi, triggerati quando lo slider è **premuto**, chiamati: **horizontalSlideEvent** e **verticalSlideEvent**. La procedura che si può seguire per associare un’**azione** a questo evento è la stessa descritta per **InteractionButton**.

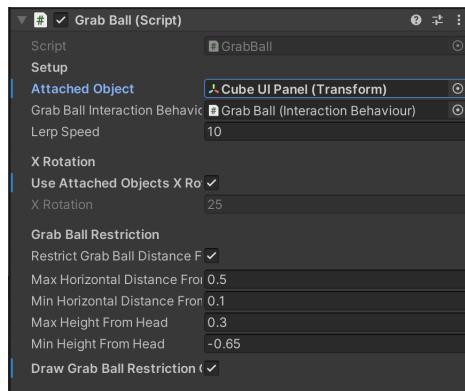


Per la gestione degli eventi è possibile utilizzare metodi quali: **HorizontalSliderValue**, **VerticalSliderValue**, **horizontalStepValue** e **wasSlid**.

```
//<summary> This slider's horizontal slider value, mapped between the values in the HorizontalValueRange. </summary>
public float HorizontalSliderValue
{
    get
    {
        return Mathf.Lerp(_horizontalValueRange.x, _horizontalValueRange.y, _horizontalSliderPercent);
    }
    set
    {
        HorizontalSliderPercent = Mathf.InverseLerp(_horizontalValueRange.x, _horizontalValueRange.y, value);
    }
}

//<summary> This slider's current vertical slider value, mapped between the values in the VerticalValueRange. </summary>
public float VerticalSliderValue
{
    get
    {
        return Mathf.Lerp(_verticalValueRange.x, _verticalValueRange.y, _verticalSliderPercent);
    }
    set
    {
        VerticalSliderPercent = Mathf.InverseLerp(_verticalValueRange.x, _verticalValueRange.y, value);
    }
}
```

- **GrabBall**: è la rappresentazione di un oggetto (una piccola sfera) a cui viene ancorato un altro oggetto (in questo caso il menù). La sfera può essere usata, per esempio, per **mostrare/restringere** l'oggetto che vi è ancorato, oppure, più semplicemente, per spostare tale oggetto in modo facile. Per definire l'oggetto da ancorare è sufficiente trascinarne la **trasformata** all'interno del campo **AttachedObject** dello script **GrabBall** che si trova nell'**Inspector**. Inoltre, da quest'ultimo, è possibile definire tutti limiti di **movimento** (sia in verticale che in orizzontale) della sfera.



Si possono utilizzare le funzioni **SetGrabBallOffset** e **SetAttachedObjectOffset** per definire rispettivamente l'**offset** della sfera relativamente all'oggetto che vi è ancorato e viceversa.

```
/// <summary>
/// Sets the Grab Ball's offset, relative to its Attached Object, moving the Grab Ball
/// The new offset is treated as a position in the Attached Object's local space.
/// </summary>
public void SetGrabBallOffset(Vector3 newOffset)
{
    _transformHelper.position = attachedObject.TransformPoint(newOffset);
    _transformHelper.rotation = attachedObject.rotation;
    _attachedObjectOffset = InverseTransformPointUnscaled(_transformHelper.transform, attachedObject.position);

    grabBallInteractionBehaviour.transform.position = _transformHelper.position;
}

/// <summary>
/// Sets the Attached Object's offset, relative its Grab Ball, moving the Attached Object
/// The new offset is treated as a position in the Grab Ball's local space.
/// </summary>
public void SetAttachedObjectOffset(Vector3 newOffset)
{
    _attachedObjectOffset = newOffset;
    UpdateAttachedObjectTargetPose();
    attachedObject.SetPose(_attachedObjectTargetPose);
}
```

- **SimpleInteractionGlow**: viene utilizzato per cambiare il colore degli oggetti durante le **interazioni**. Le interazioni possono essere di diverso tipo, ad esempio: l'**avvicinamento** dell'indice di una mano, il **contatto** vero e proprio, o ancora la **pressione** di un oggetto. Il cambio di colore non è diretto ma è **graduale** verso il **targetColor**.

```

void Update()
{
    if (_materials != null)
    {

        // The target color for the Interaction object will be determined by various simple state checks.
        Color targetColor = defaultColor;

        // "Primary hover" is a special kind of hover state that an InteractionBehaviour can
        // only have if an InteractionHand's thumb, index, or middle finger is closer to it
        // than any other interaction object.
        if (_intObj.isPrimaryHovered && usePrimaryHover)
        {
            targetColor = primaryHoverColor;
        }
        else
        {
            // Of course, any number of objects can be hovered by any number of InteractionHands.
            // InteractionBehaviour provides an API for accessing various interaction-related
            // state information such as the closest hand that is hovering nearby, if the object
            // is hovered at all.
            if (_intObj.isHovered && useHover)
            {
                float glow = _intObj.closestHoveringControllerDistance.Map(0F, 0.2F, 1F, 0.0F);
                targetColor = Color.Lerp(defaultColor, hoverColor, glow);
            }
        }

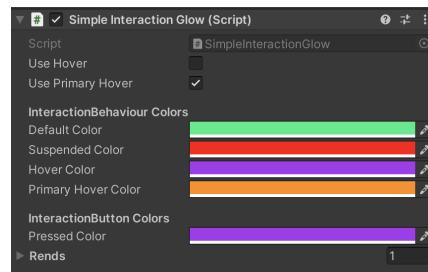
        if (_intObj.isSuspended)
        {
            // If the object is held by only one hand and that holding hand stops tracking, the
            // object is "suspended." InteractionBehaviour provides suspension callbacks if you'd
            // like the object to, for example, disappear, when the object is suspended.
            // Alternatively you can check "isSuspended" at any time.
            targetColor = suspendedColor;
        }

        // We can also check the depressed-or-not-depressed state of InteractionButton objects
        // and assign them a unique color in that case.
        if (_intObj is InteractionButton && (_intObj as InteractionButton).isPressed)
        {
            targetColor = pressedColor;
        }

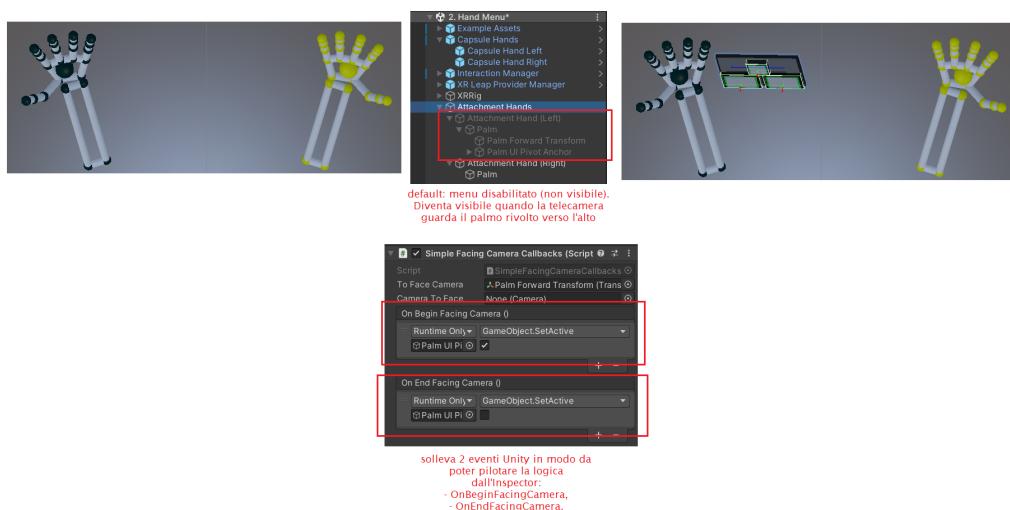
        // Lerp actual material color to the target color.
        for (int i = 0; i < _materials.Length; i++)
        {
            _materials[i].color = Color.Lerp(_materials[i].color, targetColor, 30F * Time.deltaTime);
        }
    }
}

```

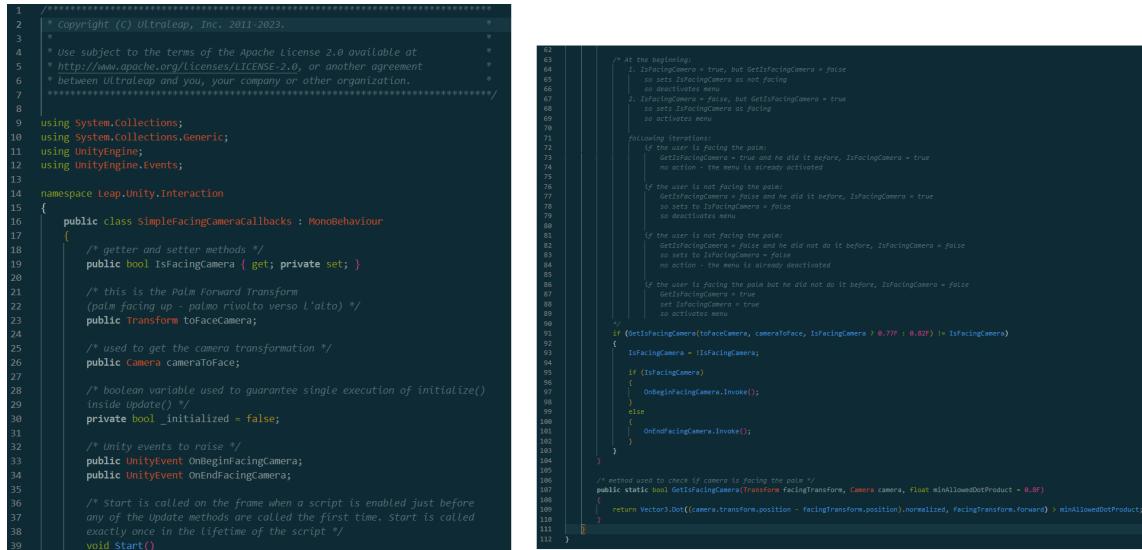
I diversi colori possono essere scelti nelle configurazioni dello script all'interno dell'inspector.



- Hand Menu:** questa demo fornisce un tipo di interazione simile alla precedente, ma con l'UI vincolata al palmo della mano e shiftata rispetto a questa. I prefab utilizzati per la costruzione della scena sono: **XR Leap Provider Manager** (5.1), **Interaction Manager** (5.1), **Attachment Hands** (5.2.1) e **HandModels** (5.2.1). I principali elementi all'interno della gerarchia sono:



Lo script **SimpleFacingCameraCallbacks** si occupa di attivare/disattivare il GameObject associato all'UI menu:



```

1  /* copyright (C) Ultraleap, Inc. 2011-2023.
2   * Use subject to the terms of the Apache License 2.0 available at
3   * http://www.apache.org/licenses/LICENSE-2.0, or another agreement
4   * between Ultraleap and you, your company or other organization.
5   */
6
7
8
9  using System.Collections;
10 using System.Collections.Generic;
11 using UnityEngine;
12 using UnityEngine.Events;
13
14 namespace Leap.Unity.Interaction
15 {
16     public class SimpleFacingCameraCallbacks : MonoBehaviour
17     {
18         /* getter and setter methods */
19         public bool IsFacingCamera { get; private set; }
20
21         /* this is the Palm Forward Transform
22          (palm facing up - palmo rivolto verso l'alto) */
23         public Transform ToFaceCamera;
24
25         /* used to get the camera transformation */
26         public Camera CameraToFace;
27
28         /* boolean variable used to guarantee single execution of initialize()
29         inside update() */
30         private bool _initialized = false;
31
32         /* Unity events to raise */
33         public UnityEvent OnBeginFacingCamera;
34         public UnityEvent OnEndFacingCamera;
35
36         /* Start is called on the frame when a script is enabled just before
37         any of the Update methods are called the first time. Start is called
38         exactly once in the lifetime of the script */
39         void Start()
40         {
41             if (ToFaceCamera != null) Initialize();
42         }
43
44         /* method used to get the Main Camera from the Scene */
45         private void Initialize()
46         {
47             if (CameraToFace == null) { CameraToFace = Camera.main; }
48             /* Set IsFacingCamera to be whatever the current state ISN'T, so that we are
49             guaranteed to fire a UnityEvent on the first initialized update(). */
50             IsFacingCamera = !GetIsFacingCamera(ToFaceCamera, CameraToFace);
51             _initialized = true; // to execute Initialize() only once in update
52         }
53
54         void Update()
55         {
56             /* if ToFaceCamera != null and Initialize() it has never been invoked */
57             if (ToFaceCamera != null && !_initialized)
58             {
59                 Initialize();
60             }
61             if (_initialized) return; /* returns if ToFaceCamera==null because _initialized is still true */
62
63
64         /* At the beginning:
65            1. IsFacingCamera = true, but GetIsFacingCamera = false
66            so deactivates menu
67            2. IsFacingCamera = false, but GetIsFacingCamera = true
68            so sets IsFacingCamera as facing
69            so activates menu
70
71         following iterations:
72         if the user is facing the palm:
73             GetIsFacingCamera = true and he did it before, IsFacingCamera = true
74             no action - the menu is already activated
75
76             if the user is not facing the palm:
77                 GetIsFacingCamera = false and he did it before, IsFacingCamera = true
78                 no action - the menu is already deactivated
79
80             if the user is facing the palm but he did not do it before, IsFacingCamera = false
81                 GetIsFacingCamera = true and he did not do it before, IsFacingCamera = false
82                 so sets IsFacingCamera = false
83                 no action - the menu is already deactivated
84
85             if the user is not facing the palm:
86                 GetIsFacingCamera = false and he did not do it before, IsFacingCamera = false
87                 no action - the menu is already deactivated
88
89         */
90
91         if (GetIsFacingCamera(ToFaceCamera, CameraToFace, IsFacingCamera > 0.77f ? 0.82f : 0.85f) != IsFacingCamera)
92         {
93             IsFacingCamera = !IsFacingCamera;
94
95             if (IsFacingCamera)
96             {
97                 OnBeginFacingCamera.Invoke();
98             }
99             else
100             {
101                 OnEndFacingCamera.Invoke();
102             }
103         }
104
105         /* method used to check if camera is facing the palm */
106         public static bool GetIsFacingCamera(Transform facingTransform, Camera camera, float minAllowedDotProduct = 0.8f)
107         {
108             Vector3 dot = (camera.transform.position - facingTransform.position).normalized * facingTransform.forward;
109             return dot.Dot(minAllowedDotProduct);
110         }
111     }
112 }
```

- **Dynamic UI:** questa demo fornisce un diverso tipo di interazione rispetto alle precedenti. Quando l'utente volge il palmo verso l'alto viene mostrato un menù contenitore di oggetti cubi che possono essere afferrati, trascinati al di fuori della box e infine rilasciati nello spazio dell'ambiente virtuale. Il processo inverso consente di riporli negli appositi contenitori con lo stesso colore. I prefab utilizzati in questa demo sono **Attachment Hands** (5.2.1), **XR Leap Provider Manager** (5.1) e **HandModels** (5.1).

Gli script utilizzati sono:

- **AnchorableBehaviour:** in Ultraleap, AnchorableBehaviour è una classe che consente di fissare un oggetto virtuale in una posizione specifica nello spazio 3D utilizzando gli Anchor. Quando si utilizza la classe AnchorableBehaviour, è possibile definire un Anchor per un oggetto virtuale e specificare la sua posizione relativa rispetto ad altri oggetti nello spazio. Una volta che l'oggetto virtuale è stato fissato in posizione utilizzando la classe AnchorableBehaviour, esso rimarrà in quella posizione anche quando l'utente si muove all'interno dell'ambiente circostante.

I metodi e i campi principali della classe sono:

- * **SetAnchor(Anchor anchor):** consente di definire l'Anchor per l'oggetto virtuale;
- * **ClearAnchor():** rimuove l'Anchor dall'oggetto virtuale;
- * **IsAnchored():** restituisce un valore booleano che indica se l'oggetto virtuale è attualmente ancorato;
- * **GetAnchor():** restituisce l'Anchor attualmente associato all'oggetto virtuale;
- OnAnchorUpdated():** questo metodo viene chiamato ogni volta che l'Anchor viene aggiornato;
- * **anchor:** il riferimento all'Anchor associato all'oggetto virtuale;

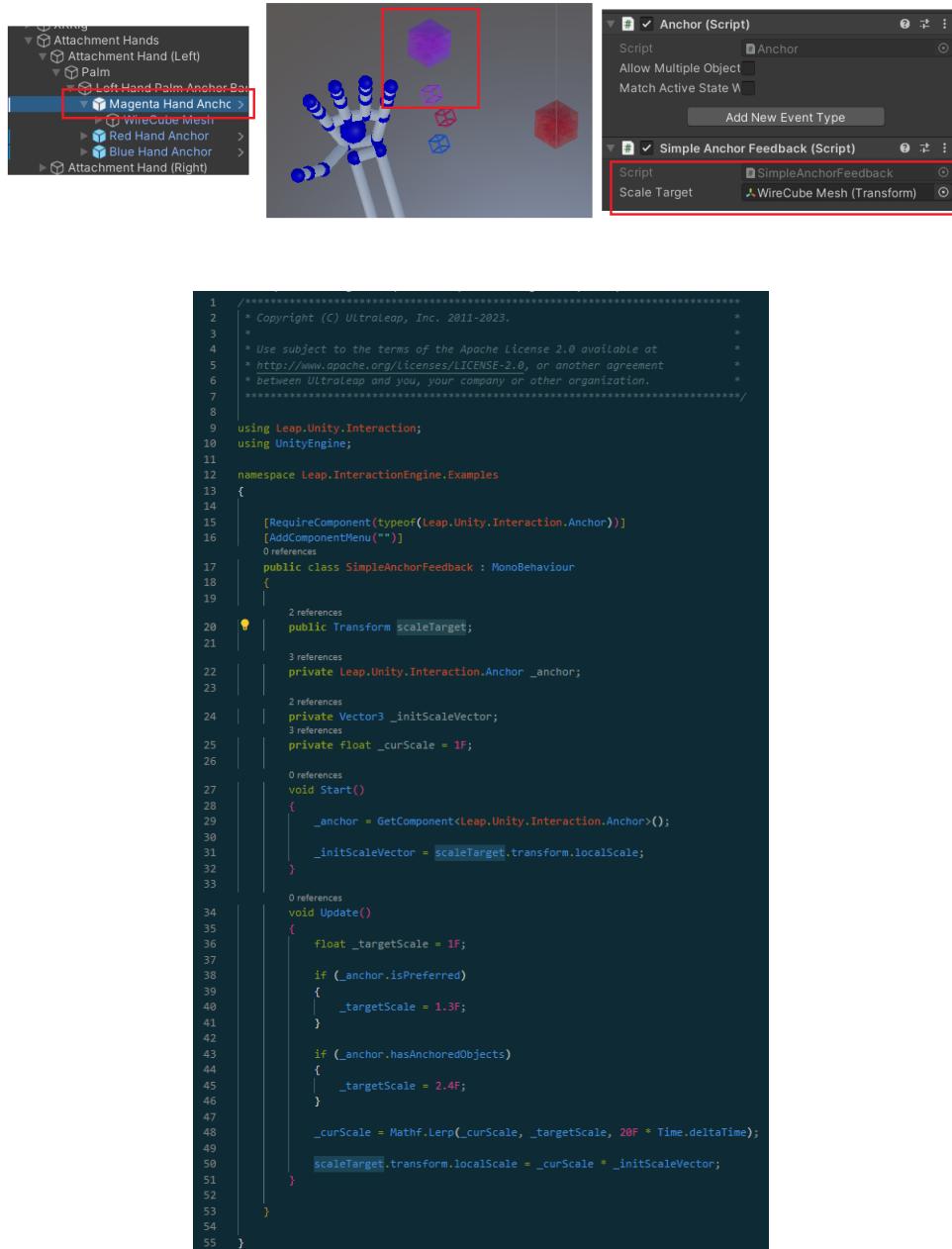
- * **isAnchored:** un valore booleano che indica se l'oggetto virtuale è attualmente ancorato;
- * **positionOffset:** la posizione relativa dell'oggetto virtuale rispetto all'Anchor;
- * **rotationOffset:** la rotazione relativa dell'oggetto virtuale rispetto all'Anchor;
- * **scaleOffset:** la scala relativa dell'oggetto virtuale rispetto all'Anchor.
- **AnchorGroup:** in Ultraleap, la classe AnchorGroup consente di creare gruppi di Anchor e di gestirli in modo coordinato, in modo che gli oggetti virtuali che utilizzano questi Anchor possano essere spostati e posizionati insieme, come se fossero parte di un unico oggetto. La classe AnchorGroup fornisce anche una serie di metodi che consentono di manipolare e interagire con gli Anchor all'interno del gruppo. Ad esempio, è possibile creare nuovi Anchor all'interno del gruppo, rimuovere gli Anchor esistenti o aggiornare la loro posizione o rotazione. Inoltre, la classe AnchorGroup consente di impostare proprietà globali per tutti gli Anchor all'interno del gruppo, come la loro visibilità o la capacità di interagire con gli oggetti virtuali che utilizzano questi Anchor.

I principali campi e metodi di questa classe sono:

- * **AddAnchor(Anchor anchor):** consente di aggiungere un nuovo Anchor al gruppo;
- * **RemoveAnchor(Anchor anchor):** consente di rimuovere un Anchor dal gruppo;
- * **Clear():** rimuove tutti gli Anchor dal gruppo;
- * **GetAnchor(int index):** restituisce l'Anchor in posizione index all'interno del gruppo;
- * **anchors:** la lista di Anchor presenti nel gruppo;
- * **isEnabled:** un valore booleano che indica se il gruppo è abilitato o disabilitato;
- **Anchor:** In Ultraleap, un Anchor è un punto nello spazio 3D che viene utilizzato per fissare un oggetto virtuale in una posizione specifica e mantenere la sua posizione relativa rispetto ad altri oggetti nello spazio. Gli Anchor vengono creati utilizzando un sistema di tracciamento a infrarossi, che rileva i movimenti delle mani e degli oggetti nell'ambiente circostante. Quando viene creato un Anchor, viene definita una posizione nello spazio 3D e viene fissato un oggetto virtuale in quella posizione.

I meotdi e i campi principali forniti dalla classe sono:

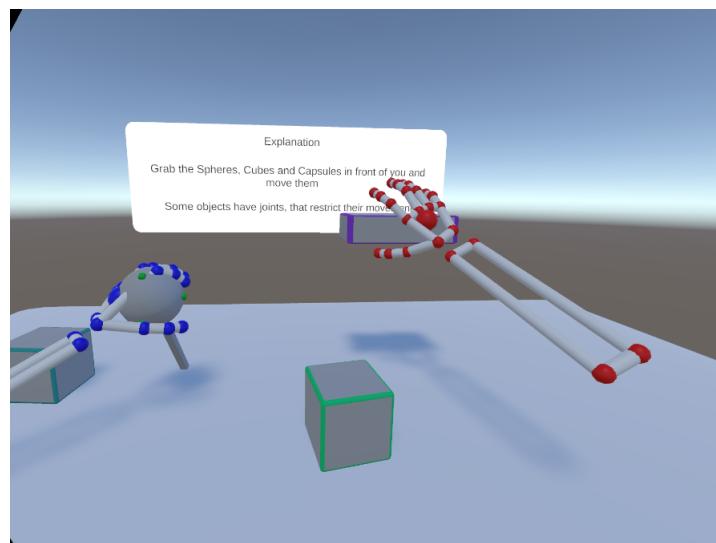
- * **Action OnAnchorPreferred = () => { }:** metodo invocato non appena un oggetto ancorabile preferisce questo ancoraggio se cerca di attaccarsi a un ancoraggio;
- * **Action OnAnchorNotPreferred = () => { }:** metodo invocato quando un oggetto ancorabile non preferisce più l'anchor;
- * **Action OnAnchorablesAttached = () => { }:** metodo invocato non appena un oggetto ancorabile viene collegato a all'ancoraggio;
- * **Action OnNoAnchorablesAttached = () => { }:** metodo invocato quando non ci sono oggetti ancorabili collegati all'ancoraggio;
- * **HashSet<AnchorableBehaviour> anchoredObjects { get; set; }:** ottiene e setta l'AnchorableBehaviours collegato a questo anchor.
- **SimpleAnchorFeedback:** fornisce un feedback visivo per gli Anchor. Quando viene utilizzato insieme a un oggetto virtuale che è stato ancorato in una posizione specifica, lo script SimpleAnchorFeedback cambia la scala del contenitore dell'oggetto virtuale per indicare quando l'Anchor è stato acquisito e quando è stato perso. In particolare, quando l'oggetto virtuale è posizionato correttamente sull'Anchor, la scala del contenitore diventa più grande per indicare che l'Anchor è stato acquisito correttamente. Al contrario, quando l'oggetto virtuale viene spostato via dall'Anchor, la scala si riduce.



5.2.4 3D Interaction

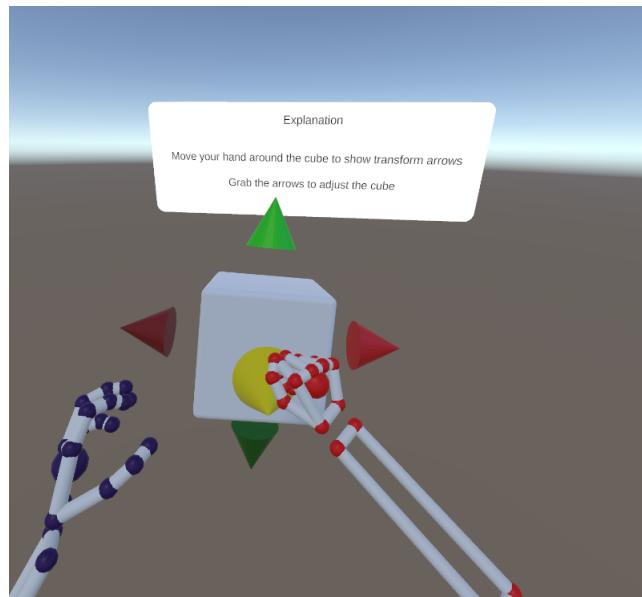
Interactable Objects.unity

Questa demo evidenzia la possibilità, oltre a poter creare oggetti di diverse forme, di aggiungere vincoli e giunti agli oggetti. In particolare si possono osservare cubi, sfere e parallelepipedi. Componendo oggetti, vincoli e giunti è possibile creare strumenti come ad esempio leve.



Interaction Events.unity

Questa demo mostra l'utilizzo del tool *TransformHandle* che l'user può usare a runtime per controllare posa e orientamento dell'oggetto. Questo è possibile grazie a delle frecce che compaiono quando la mano virtuale si trova in prossimità dell'oggetto. Afferrandole è possibile applicargli delle rototraslazioni.



I prefab utilizzati sono: **XR Leap Provider Manager** (5.1), **Interaction Manager** (5.1) e **HandModels** (5.1) con **CapsuleHands** (5.2.1).

Gli script utilizzati:

- **TransformHandle:** questa è la classe base per la gestione degli eventi: **OnShouldShowHandle** (quando mostrare la maniglia), **OnShouldHideHandle** (quando nascondere la maniglia), **OnHandleActivated** (azione da compiere quando si usa la maniglia), **OnHandleDeactivated** (azione da compiere quando la maniglia non è più in uso).

```

private void onGraspBegin()
{
    _tool.NotifyHandleActivated(this);
    OnHandleActivated.Invoke();
}

private void onGraspEnd()
{
    _tool.NotifyHandleDeactivated(this);
    OnHandleDeactivated.Invoke();
}

```

L'implementazione di tali **eventi** sfrutta le callback **onGraspBegin** e **onGraspEnd** (ereditate dalla classe **InteractionBehaviour** 5.1)

- **TransformTool:** lo script si occupa di aggiornare la **posa** dell'oggetto. Qui vengono accumulate tutte le informazioni relative al **movimento** e alla **rotazione** di tutte le **maniglie**.
- **TransformTranslationHandle:** questo script utilizza le informazioni relative alla **traslazione** dell'oggetto (prima e dopo l'azione dell'utente) e le utilizza per calcolare di quanto l'utente vuole spostare la maniglia lungo la sua direzione (x,y o z). Questa informazione viene poi passata a **TransformationTool** che gestisce l'aggiornamento dell'oggetto.

```

public class TransformTranslationHandle : TransformHandle
{
    public TranslationAxis axis;

    protected override void Start()
    {
        // Populates _intObj with the InteractionBehaviour, and _tool with the
        // TransformTool.
        base.Start();

        // Subscribe to OnGraspedMovement; all of the logic will happen when the handle is
        // moved via grasping.
        _intObj.OnGraspedMovement += onGraspedMovement;
    }

    private void onGraspedMovement(Vector3 presolvePos, Quaternion presolveRot,
        Vector3 solvedPos, Quaternion solvedRot,
        List<InteractionController> controllers)
    {
        /*
        * OnGraspedMovement provides the position and rotation of the Interaction object
        * before and after it was moved by its grasping hand. This callback only occurs
        * when one or more hands are grasping the Interaction object. In this case, we
        * don't care about how many or which hands are grasping the object, only where
        * the object is moved.
        *
        * The Translation Handle uses the pre- and post-solve movement information to
        * calculate how the user is trying to move the object along this handle's forward
        * direction. Then the Translation Handle will simply override the movement caused
        * by the grasping hand and reset itself back to its original position.
        *
        * The movement calculated by the Handle in this method is reported to the Transform
        * Tool, which accumulates movement caused by all Handles over the course of a frame
        * and then moves the target object and all of its child Handles appropriately at
        * the end of the frame.
        */

        // Calculate the constrained movement of the handle along its forward axis only.
        Vector3 deltaPos = solvedPos - presolvePos;
        Vector3 handleForwardDirection = presolveRot * Vector3.forward;
        Vector3 deltaAxisPos = handleForwardDirection * Vector3.Dot(handleForwardDirection, deltaPos);

        // Notify the tool about the calculated movement.
        _tool.NotifyHandleMovement(deltaAxisPos);

        // In this case, the tool itself will accumulate delta positions and delta rotations
        // from all handles, and will then synchronize handles to the appropriate positions and
        // rotations at the end of the frame.

        // Because the Tool will be the one to actually move this Handle, all we have left to do
        // is to undo all of the motion caused by the grasping hand.
        _intObj.rigidbody.position = presolvePos;
        _intObj.rigidbody.rotation = presolveRot;
    }
}

```

- **TransformRotationHandle:** funziona nello stesso modo di **TransformTranslationHandle** solamente che in questo caso si utilizzano le informazioni relative alla **rotazione** dell'oggetto (prima e dopo l'interazione con l'utente).

```

public class TransformRotationHandle : TransformHandle
{
    protected override void Start()
    {
        // Populates _intObj with the InteractionBehaviour, and _tool with the TransformTool.
        base.Start();

        // Subscribe to OnGraspedMovement; all of the logic will happen when the handle is moved via grasping.
        _intObj.OnGraspedMovement += onGraspedMovement;
    }

    private void onGraspedMovement(Vector3 presolvePos, Quaternion presolveRot, Vector3 solvedPos, Quaternion solvedRot, List<InteractionController> controllers)
    {
        /*
        * The RotationHandle works very similarly to the TranslationHandle.
        *
        * We use OnGraspedMovement to get the position and rotation of this object
        * before and after it was moved by its grasping hand. We calculate how the handle
        * would have rotated and report that to the Transform Tool, and then we move
        * the handle back where it was before it was moved, because the Tool will
        * actually move all of its handles at the end of the frame.
        */

        // Constrain the position of the handle and determine the resulting rotation required to get there.
        Vector3 presolveToolToHandle = presolvePos - _tool.transform.position;
        Vector3 solvedToolToHandleDirection = (solvedPos - _tool.transform.position).normalized;
        Vector3 constrainedToolToHandle = Vector3.ProjectOnPlane(solvedToolToHandleDirection, (presolveRot * Vector3.up)).normalized * presolveToolToHandle.magnitude;
        Quaternion deltaRotation = Quaternion.FromToRotation(presolveToolToHandle, constrainedToolToHandle);

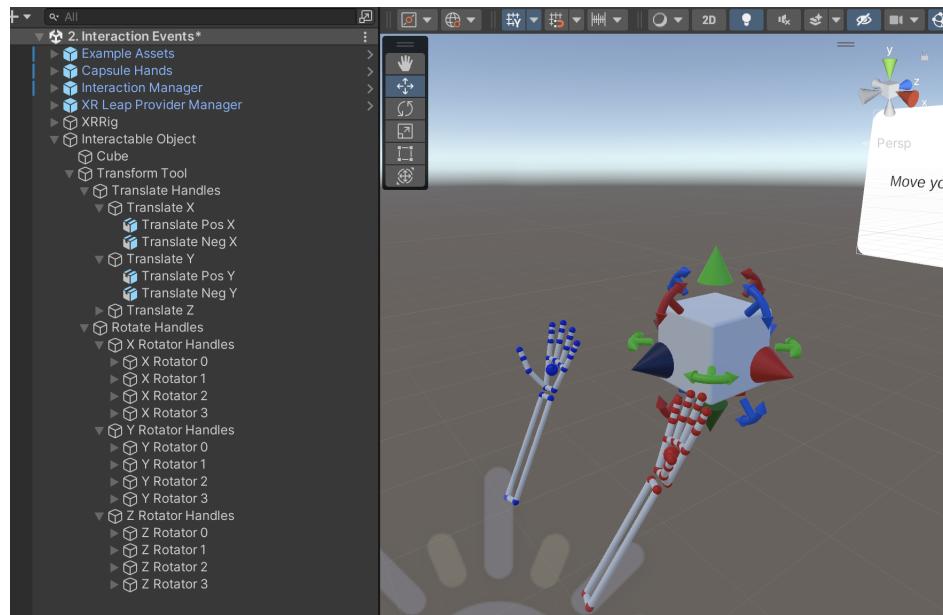
        // Notify the tool about the calculated rotation.
        _tool.NotifyHandleRotation(deltaRotation);

        // Move the object back to its original position, to be moved correctly later on by the Transform Tool.
        _intObj.rigidbody.position = presolvePos;
        _intObj.rigidbody.rotation = presolveRot;
    }
}

```

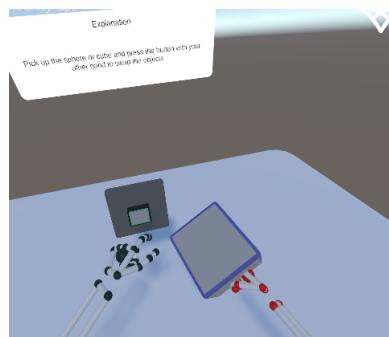
Le maniglie sono organizzate **gerarchicamente**.

- maniglie per la **traslazione**: una coppia per ogni asse;
- maniglie per la **rotazione**: quattro per ogni asse;



Swap Grasp.unity

Questa demo combina due tipologie di interazioni: l'utente può afferrare l'oggetto (cubo o sfera), quindi sfruttando un pulsante gli è possibile cambiare il tipo di oggetto afferrato.



Lo script utilizzato è **SwapGraspExample**. Premendo il bottone con una mano, mentre con l'altra si sorregge uno dei due oggetti , è possibile scambiarli. Supponendo che l'oggetto preso sia la sfera, allora il cubo **erediterà** la posa che la sfera possedeva nel momento in cui è stato invocato lo scambio. Viceversa, il cubo erediterà la posa della sfera.

```
public class SwapGraspExample : MonoBehaviour
{
    public IntObj objA, objB;
    public InteractionButton swapButton;
    private bool _swapScheduled = false;

    void Start()
    {
        swapButton.OnUnpress += scheduleSwap;
        // Wait for just after the PhysX update to swap a grasp;
        // this allows the swapped object to inherit the _latest_rigidbody position and
        // rotation from the original held object (which needs the PhysX update to receive
        // scheduled force / MovePosition / MoveRotation changes from the grasped movement
        // system).
        PhysicsCallbacks.OnPostPhysics += onPostPhysics;
    }

    private void scheduleSwap()
    {
        _swapScheduled = true;
    }

    private void onPostPhysics()
    {
        //Swapping when both objects are grasped is unsupported
        if (objA.isGrasped && objB.isGrasped)
        { return; }

        if (_swapScheduled && (objA.isGrasped || objB.isGrasped))
        {

            // Swap "a" for "b"; a will be whichever object is the grasped one.
            IntObj a, b;
            if (objA.isGrasped)
            {
                a = objA;
                b = objB;
            }
            else
            {
                a = objB;
                b = objA;
            }

            // (Optional) Remember B's pose and motion to apply to A post-swap.
            var bPose = new Pose(b.rigidbody.position, b.rigidbody.rotation);
            var bVel = b.rigidbody.velocity;
            var bAngVel = b.rigidbody.angularVelocity;

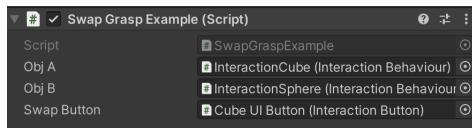
            // Match the rigidbody pose of the originally held object before swapping.
            // If it exists, always use the latestScheduledGraspPose to perform a SwapGrasp!
            // This prevents subtle slippage with non-kinematic objects that may experience
            // gravity forces, drag, or hit other objects, which can leak into the new
            // grasping pose when the SwapGrasp is performed.
            if (a.latestScheduledGraspPose.HasValue)
            {
                b.rigidbody.position = a.latestScheduledGraspPose.Value.position;
                b.rigidbody.rotation = a.latestScheduledGraspPose.Value.rotation;
            }
            else
            {
                b.rigidbody.position = a.rigidbody.position;
                b.rigidbody.rotation = a.rigidbody.rotation;
            }

            // Swap!
            a.graspingController.SwapGrasp(b);

            // Move A over to where B was, and for fun, let's give it B's motion as well.
            a.rigidbody.position = bPose.position;
            a.rigidbody.rotation = bPose.rotation;
            a.rigidbody.velocity = bVel;
            a.rigidbody.angularVelocity = bAngVel;
        }

        _swapScheduled = false;
    }
}
```

Si possono impostare i due oggetti da scambiare e il bottone a cui far riferimento dall'inspector.



Turntable And Pullcord.unity

Questa demo permette, attraverso l'uso di una specifica *gesture* (*pinch*), di attivare un oggetto. In particolare, è possibile ancorare il cubo al tavolo, farlo ruotare facendo scorrere la mano sulla corona circolare della base su cui è poggiato e suddividerlo in cubi più piccoli tirando una pullcord elastica.

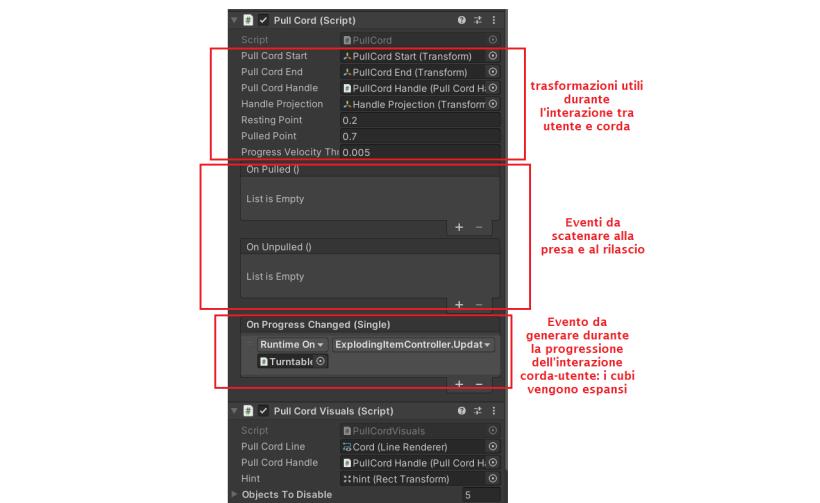


I prefab utilizzati in questa demo sono: **XR Leap Provider Manager** (5.1), **Interaction Manager** (5.1) e **Pull Cord and Turntable**.

Gli script utilizzati sono:

- **Turtable:** è responsabile per la rotazione del turtable ogni volta che i polpastrelli si intersecano con l'oggetto (contatto determinato con **IsPointInsideTurntable(Vector3 localPoint)**). Richiama inoltre **TurntableVisuals.UpdateVisuals()** quando i parametri del giradischi sono cambiati;
- **TurntableVisuals:** crea un cerchio con un rendering di linee e tratteggi con **drawMeshInstanced**. Prende informazioni come l'altezza e il raggio del piatto rotante da **Turntable**. Lo script è progettato per aggiungere effetti visivi all'oggetto 3D che viene ruotato tramite lo script "Turntable". Questo script consente di creare effetti visivi come ombre, effetti di luce e particelle che si attivano durante la rotazione dell'oggetto. I metodi principali sono **UpdateVisuals()** e **DrawTurntable()**;
- **PullCord:** questo componente di interazione è progettato per gestire l'interazione dell'utente con un oggetto simulando la trazione di una corda.. Aggiorna la posizione di proiezione dell'impugnatura della corda, aggiorna la posizione di riposo dell'impugnatura, fornisce un valore di trazione per l'avanzamento della corda e aggiorna gli elementi che esplodono. Il funzionamento dello script "PullCord" di Ultraleap avviene attraverso tre fasi principali: 1. "Grabbing": Quando la mano dell'utente entra nella zona di interazione dell'oggetto, viene effettuato un "grab" (presa) virtuale della corda. In questa fase, la corda viene attaccata alla mano dell'utente, che può quindi muoverla in qualsiasi direzione; 2. "Pulling": L'utente può quindi "tirare" la corda, ovvero muoverla in una direzione specifica per attivare l'oggetto associato all'azione. Durante questa fase, lo script calcola la direzione e la forza del movimento dell'utente sulla corda e aggiorna di conseguenza la posizione dell'oggetto; 3. "Releasing": Quando l'utente rilascia la corda, lo script rilascia la "presa" virtuale della corda e l'oggetto torna alla sua posizione iniziale;
- **PullCordHandle:** tiene traccia dell'impugnatura della corda e la muove in accordo con l'attrazione della mano e il pinching;

- **PullCordVisuals:** aggiora la visuale della pull cord dipendentemente dallo stato della pullcord.



```
1 reference
private Vector3 MidpointLeap.Hand hand)
{
    return (hand.GetIndex().TipPosition + hand.GetThumb().TipPosition) / 2f;
}

1 reference
private void UpdatePinchingLeap.Hand hand, float distanceToRestingPos)
{
    float distance = Vector3.Distance(hand.GetIndex().TipPosition, hand.GetThumb().TipPosition);

    // Distanza di allontanamento della maniglia dalla sua posizione di riposo (_stretchThreshold)
    // distanceToRestingPos = Vector3.Distance(midpoint, RestingPos)
    // La posizione in cui la maniglia salta se rilasciata (RestingPos)
    // Vector3 midpoint = Midpoint(_intBeh.primaryHoveringHand)
    if (_isPinching && (distance > _pinchDeactivateDistance || distanceToRestingPos > _stretchThreshold))
    {
        StopPinching();
    }

    // only start pinching if the midpoint is close enough to the resting position
    else if (distanceToRestingPos < _distanceToHandThreshold)
    {
        if (!_isPinching && distance < _pinchActivateDistance)
        {
            StartPinching();
        }
    }
}
```

```
1 reference
private void StartPinching()
{
    if (OnStateChanged != null) OnStateChanged.Invoke(PullCordState.Pinned);

    _isPinching = true;
}

3 references
private void StopPinching()
{
    if (OnStateChanged != null) OnStateChanged.Invoke(_intBeh.isPrimaryHovered ? PullCordState.Hovered : PullCordState.Default);

    if (OnPinchEnd != null) OnPinchEnd.Invoke();

    _isPinching = false;
}
```

```
14 references
public enum PullCordState
{
    Default,
    Hovered,
    Pinched,
    Inactive
}


/// OnStateChanged is invoked when the pull cord handle state changes.

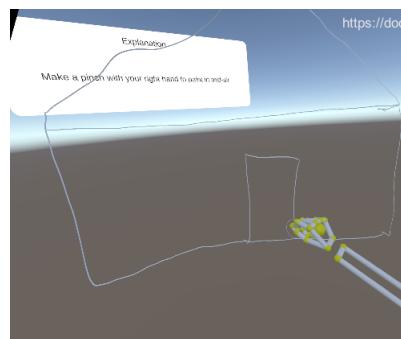
11 references
public UnityEvent<PullCordState> OnStateChanged;

/// OnPinchEnd is invoked when the pull cord handle was pinched, but has been let go off this frame.

4 references
public UnityEvent OnPinchEnd;
```

Pinch To Paint.unity

Con questa demo mostra come è possibile utilizzare la detection della *gesture* (*pinch*) per disegnare nello spazio 3D.

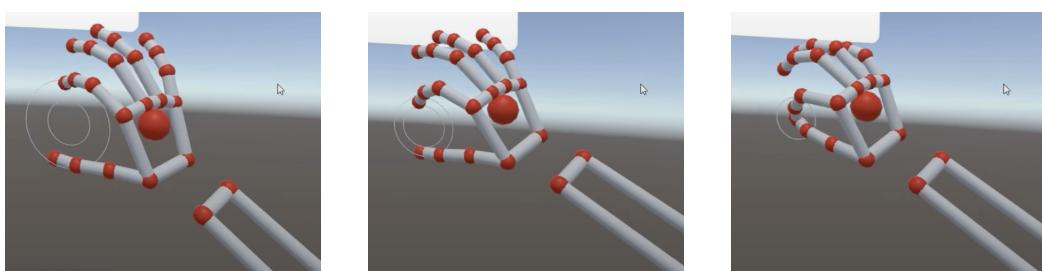


I prefab utilizzati sono: **XR Leap Provider Manager** (5.1), **Interaction Manager** (5.1), **HandModels** (5.1) utilizzato con **CapsuleHands** (5.2.1).

Gli script utilizzati sono:

- **PinchDetector:** questa classe serve per la gestione/creazione di azioni basate sul **pinch**. Quando viene utilizzata insieme ad un **HandModel** permette il **detect** di pose di tipo **pinch**. In particolare, nel caso di questa demo, il detector è stato associato alla **mano destra**.
- **PaintCursor:** si occupa della parte **grafica** del cursore (posizione, spessore e colore) e tiene traccia del corrente stato di **pinch** e di **painting**, riferendosi a un oggetto **PinchDetector**.
- **RectToroid:** serve a generare una **mesh toroidale** che viene utilizzata per il **paint cursor**. Infatti, guardando la mano destra, si possono notare due cerchi concentrici entrambi appartenenti alla classe **RectToroid** :
 - **RectToroidPinchTarget:** è fisso con un certo raggio
 - **RectToroidPinchState:** si **restringe** quando l'indice e il pollice della mano destra si avvicinano.

Affinchè il disegno inizi i due cerchi devono essere **sovraposti**: in questo modo il cursore ci fornisce un'indicazione su quanto **pinch** è ancora necessario prima di arrivare a poter disegnare.



- **PinchStrokeProcessor:** è associato a un **PaintCursor** per ottenere le informazioni sullo stato di **pinch** della mano e la posizione della stessa. Si occupa di aggiungere nuovi **tratti** di linee nelle posizioni corrette. Si occupa anche di gestire l'audio che simula il contatto di una matita con un foglio, in funzione della **velocità di disegno**.

Nella funzione **start** viene creato un oggetto di tipo **StrokeProcessor** che si occupa della **logica** vera e propria di creazione delle linee di disegno.

Nella funzione **Update** viene controllato se si sta cercando di disegnare in un punto accettabile, ovvero visibile dal punto di vista (**FOV**) dell'utente. A questo punto si passa alla gestione dell'azione di disegno.

```
void Update()
{
    // check whether we are trying to draw within an acceptable camera FOV
    float angleFromCameraLookVector = Vector3.Angle(Camera.main.transform.forward,
    _paintCursor.transform.position - Camera.main.transform.position);
    bool withinAcceptableCameraFOV = angleFromCameraLookVector < _acceptableFOVAngle;

    if (_paintCursor.Tracked && !_strokeProcessor.BufferingStroke && withinAcceptableCameraFOV)
    {
        BeginStroke();
    }

    if (_paintCursor.DidStartPinch && withinAcceptableCameraFOV && !_strokeProcessor.ActualizingStroke)
    {
        StartActualizingStroke();
    }

    if (_paintCursor.Tracked && _strokeProcessor.BufferingStroke)
    {
        UpdateStroke();
    }

    if ((!_paintCursor.Tracked || !_paintCursor.pinchDetector.Active) && _strokeProcessor.ActualizingStroke)
    {
        StopActualizingStroke();
    }

    if ((!_paintCursor.Tracked || !_strokeProcessor.ActualizingStroke && !withinAcceptableCameraFOV) &&
    _strokeProcessor.BufferingStroke)
    {
        EndStroke();
    }
}
```

- **StrokeProcessor:** un oggetto **Stroke** (il tratto disegnato) è composto da più oggetti **StrokePoints**.

```
public struct StrokePoint
{
    public Vector3 position;
    public Vector3 normal;
    public Quaternion rotation;
    public Quaternion handOrientation;
    public float thickness;
    public Color color;
}
```

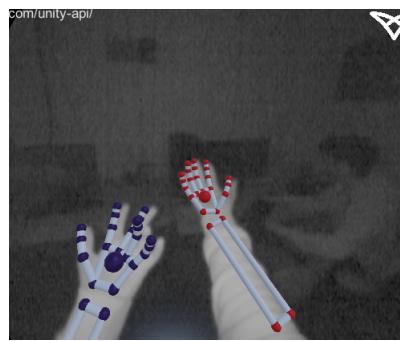
I punti di uno **Stroke** vengono aggiunti ad una lista nuova (**StartStroke**, se la posizione dell'ultimo **StrokePoint** è **lontana** da quella attuale) o ad una già esistente (**UpdateStroke**). Si appoggia all'utilizzo di un **ThickRibbonRenderer** per il display della linea (che viene aggiornato o creato nelle stesse funzioni appena menzionate).

- **ThickRibbonRenderer:** fornisce un **Renderer** per le linee disegnate. Queste in realtà sono dei veri e propri 'nastri' di un certo spessore, quindi aventi **tre dimensioni**.

5.3 Troubleshooting

Troubleshooting/Infrared Viewer.unity

Questa demo evidenzia l'accuratezza del tracking del Leap, in quanto consente di visualizzare sia le mani virtuali che quelle reali acquisite tramite la telecamera del visore.



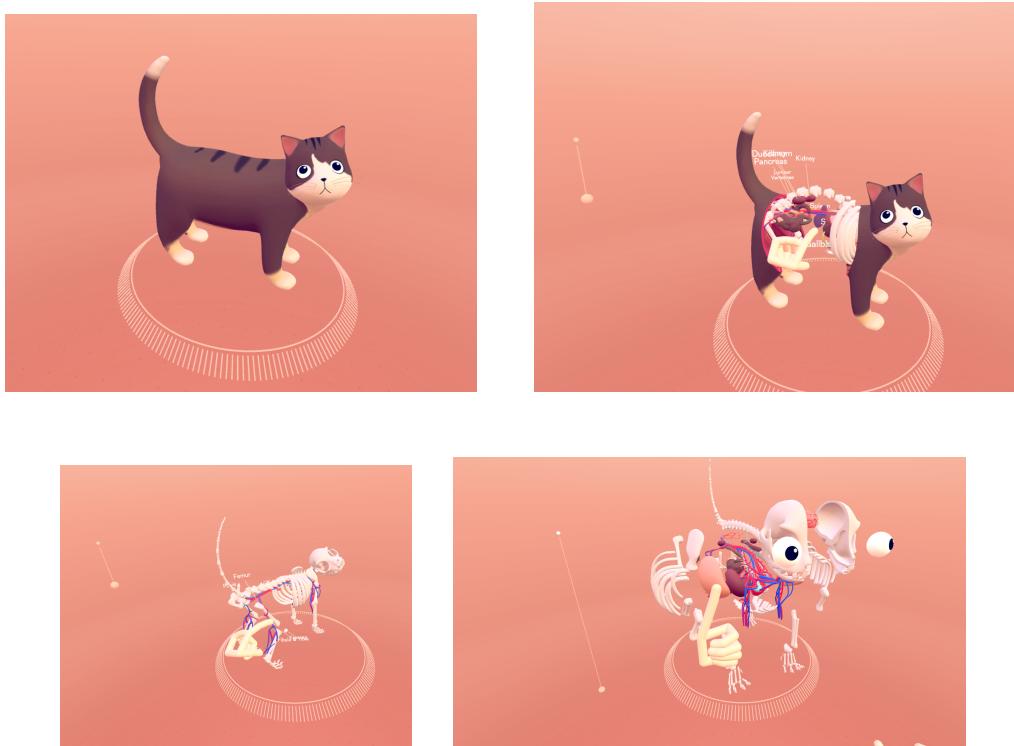
I prefab utilizzati in questa demo sono **Service Provider XR** (5.1) e **VR Infrared Viewer**. Non è necessario scrivere codice, occorre solo trascinare l'Image Retriever nella scena e configurarlo. Lo script utilizzato **LeapImageRetriever** viene attaccato al GameObject Camera.

5.4 Ultraleap Demos

I paragrafi sottostanti sono un esempio d'uso delle demo-base precedenti, sfruttate per la realizzazione di applicazioni più complesse.

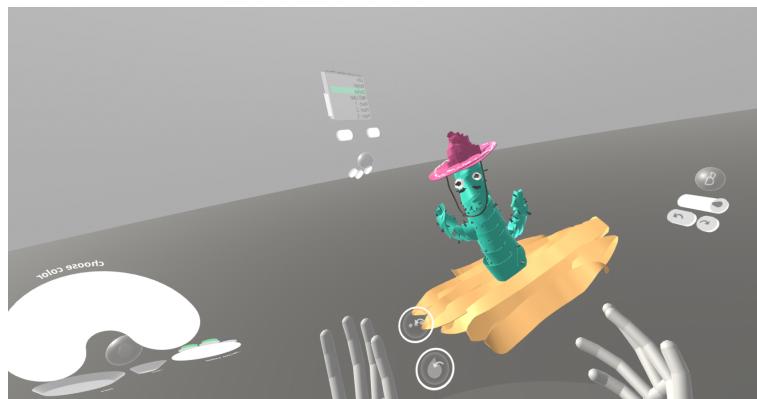
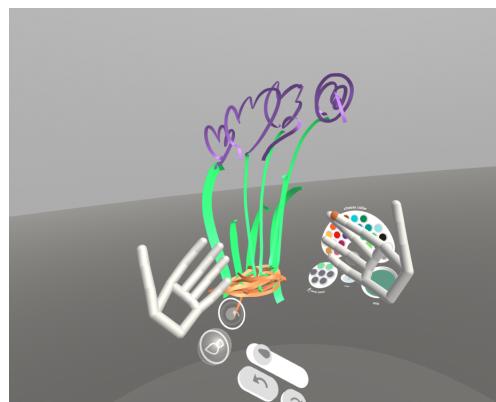
Cat Explorer

Le tecnologie VR risultano essere più coinvolgenti e intuitive rispetto ad altre tecnologie e possono essere utilizzate a scopo educativo per rendere possibili scenari che non lo sarebbero nella realtà. In questa demo è possibile interagire con un gatto per esplorarne l'anatomia. Attraverso un menù (implementato come nella sezione 5.2.3) è possibile selezionare un diverso livello di dettaglio della composizione anatomica (pelle, apparato muscolare, organi, apparato circolatorio, scheletro). Inoltre, è possibile considerare una specifica sezione definibile attraverso il *pinching* di una finestra scorrevole. Si può far ruotare il gatto posto su una basetta come quella presentata nel paragrafo 5.2.4. Infine, attraverso il *pinching* di una *pullcord* (paragrafo 5.2.4) è possibile ottenere un esploso dell'animale. Muovendosi nell'ambiente reale è possibile avvicinarsi al gatto per cogliere più dettagli, toccandolo ogni componente viene dotata di un'etichetta (nome).



Paint

Questa demo offre la possibilità di creare disegni in tre dimensioni sfruttando la detection della *gesture pinch*, ed è un'espansione della versione basica proposta nel paragrafo 5.2.4. Infatti, in questa versione è disponibile un numero maggiore di strumenti e feature: palette di colori completamente personalizzabile, possibilità di cambiare spessore della linea tracciata con il dito, possibilità di disposizione degli strumenti nello spazio circostante (come nella demo Dynamic UI descritta nel paragrafo 5.2.3). Altre feature aggiuntive sono: salvataggio del disegno, cancellazione degli errori e caricamento di un modello esistente.



6 Sviluppo della demo

Al termine della fase di testing delle demo offerte dalla libreria Ultraleap e dell'analisi del relativo codice, abbiamo deciso di provare a mettere in pratica ciò che abbiamo imparato realizzando una nostra demo. Nello specifico abbiamo realizzato un prototipo di gioco in un ambiente virtuale che simula "Boccette" una specialità del biliardo all'italiana. Lo scopo principale è quello di posizionare la propria biglia più vicina al pallino rispetto a quella dell'avversario.

6.1 Passi necessari per l'uso

I passi principali che sono necessari per utilizzare la nostra demo sono:

1. Verificare di aver soddisfatto i prerequisiti (Sezione 2);
2. Configurare l'hardware necessario (Sezione 3);
3. Configurare correttamente l'ambiente virtuale (Sezione 4);
4. Inserire il contenuto di "*UltraleapPoolGameDemo.zip*" all'interno di **directory del progetto creato al Passo 3 > Assets**. Nel file zip sono contenute le seguenti direcotry: Scenes, Materials, Prefabs e Scripts.

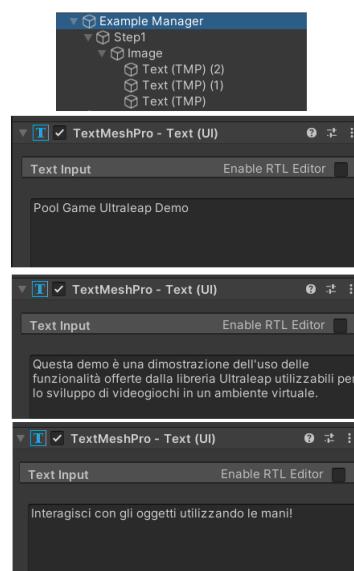
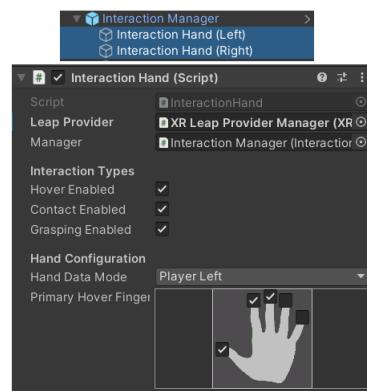
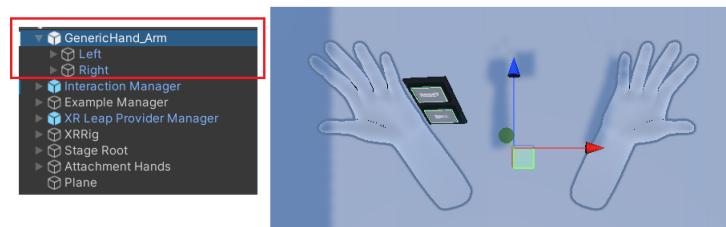
6.2 Prefab UltraleapPoolGameDemo

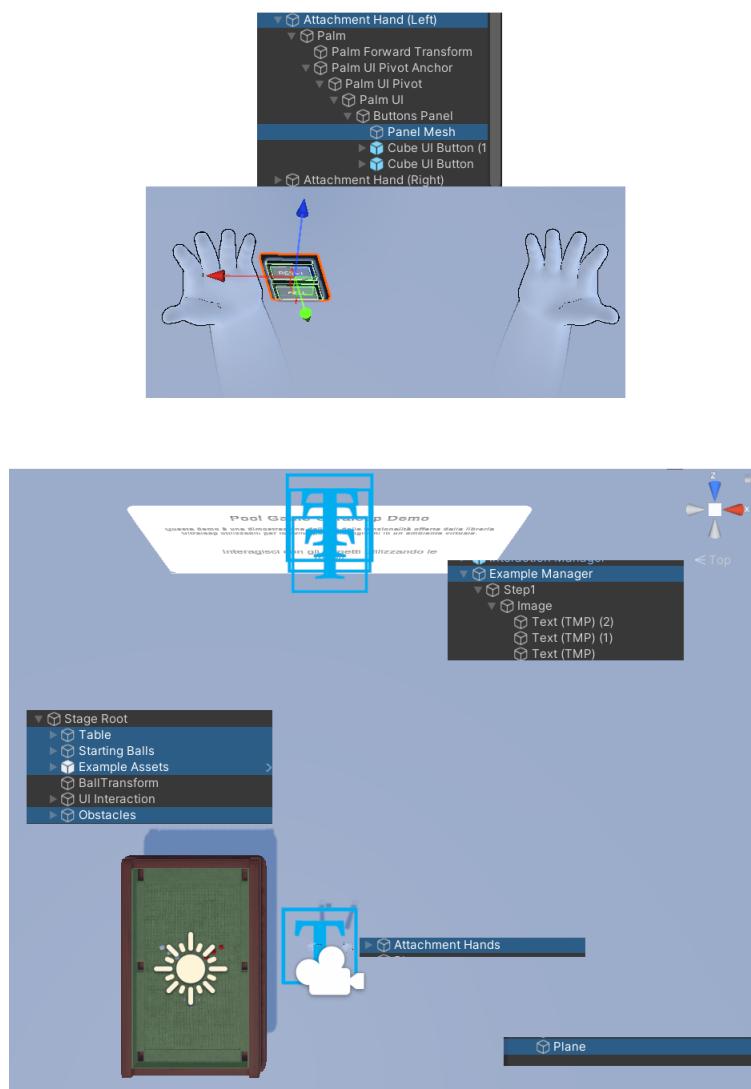
I prefab utilizzati in questa demo sono:

- **GenericHand_Arm;**
- **Interaction Manager;**
- **Example Manager;**
- **XR Leap Provider Manager;**
- **XRRig;**
- **Attachment Hands;**
- **Custom prefab tavolo da gioco.**

Gli script utilizzati sono:

- **CameraOffset;**
- **LeapXRServiceProvider;**
- **InteractionBehaviour;**
- **SimpleInteractionGlow;**
- **InteractionHand;**
- **ResetScene;**
- **SpawnRandomObjectAtPosition;**
- **IgnoreCollisionsInChildren;**
- **SimpleFacingCameraCallbacks.**





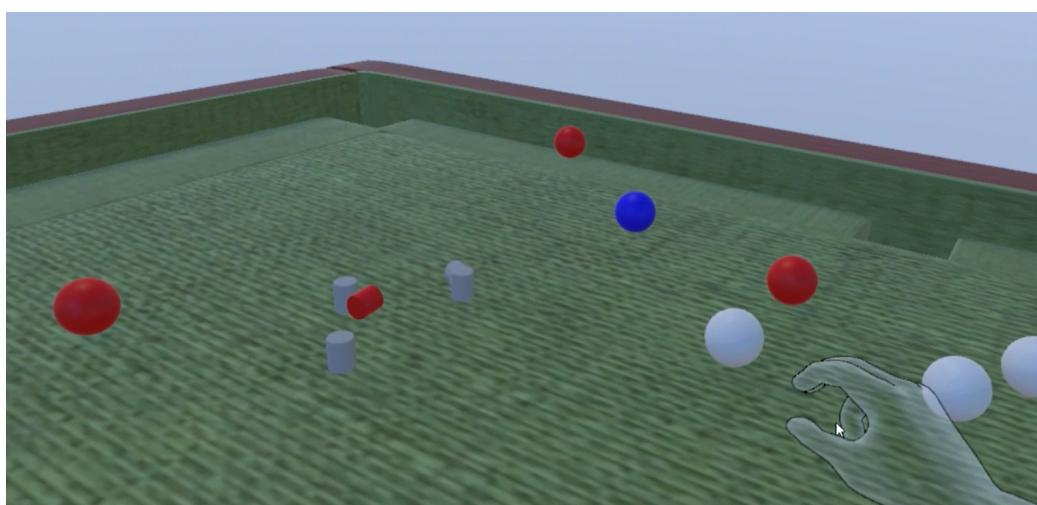
I GameObject inseriti nella scena sono:

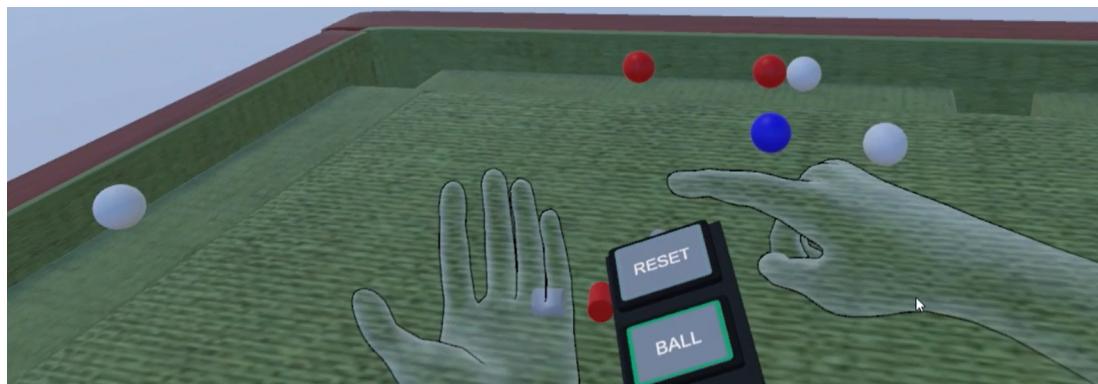
- Table:** tavolo da gioco creato sfruttando i modelli 3D "Ultraleap_Cube" (piano, sponde) e "Ultraleap_Cylinder" (gambe del tavolo) a cui sono stati applicati dei materials specifici per le parti in legno e per il tappetino da gioco in verde. È un parent GameObject che al suo interno contiene altri GameObject a cui si associano i modelli;
- Starting Balls:** parent GameObject che contiene altri GameObject che rappresentano le palle da gioco. Sono state realizzate sfruttando il modello 3D "Ultraleap_Sphere_Interactable". Fondamentalmente sono delle sfere a cui è applicato lo script "InteractionBehaviour" e un material per il colore;
- Obstacles:** parent GameObject che contiene altri GameObject che rappresentano gli ostacoli da gioco. Sono stati realizzati sfruttando il modello 3D "Ultraleap_Cylinder". Fondamentalmente sono dei cilindri a cui è applicato lo script "InteractionBehaviour" e un material per il colore;
- Attachment Hands:** parent GameObject utilizzato per contenere ulteriori GameObject per inserire un menu con due bottoni nella scena. I bottoni permettono di resettare la scena o di generare delle nuove palle da gioco in una posizione casuale all'interno del tavolo da gioco (tramite gli appositi script ResetScene e SpawnRandomObjectAtPosition).



palle da gioco:
create sfruttando il
prefab
"InteractionBall"

realizzati utilizzando gli
"Interactable_cylinder"





6.3 Descrizione script custom

Oltre agli script forniti da Ultraleap abbiamo realizzato due script per inserire delle funzionalità ai bottoni dell'UI menu:

- Reset della scena;

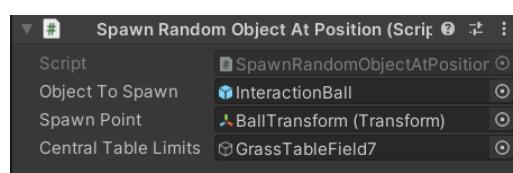
```

1 1 using System.Collections;
2 2 using System.Collections.Generic;
3 3 using UnityEngine;
4 4 using UnityEngine.SceneManagement;
5 5 public class ResetScene : MonoBehaviour
6 6 {
7 7     //Reset current scene
8 8     public void ResetMainScene()
9 9     {
10 10         SceneManager.LoadScene(SceneManager.GetActiveScene().name);
11 11     }
12 12 }
```

- Creazione di nuove palle da gioco in una posizione casuale all'interno del tavolo da gioco.

```

1 1 using UnityEngine;
2 2
3 3 public class SpawnRandomObjectAtPosition : MonoBehaviour
4 4 {
5 5     public GameObject objectToSpawn;
6 6     public Transform spawnPoint;
7 7     public GameObject centralTableLimits;
8 8
9 9     public void SpawnRandomObject()
10 10    {
11 11        var size = GameObject.Find("GrassTableField7").GetComponent<Collider>().bounds.size;
12 12
13 13        Vector3 center = new Vector3(centralTableLimits.transform.position.x, 1, centralTableLimits.transform.position.z);
14 14
15 15        var px = Random.Range(-size.x / 2, size.x / 2);
16 16        var pz = Random.Range(-size.z / 2, size.z / 2);
17 17
18 18        Vector3 pos = center + new Vector3(px, spawnPoint.position.y, pz);
19 19        Instantiate(objectToSpawn.transform, pos, Quaternion.identity, spawnPoint);
20 20    }
21 21 }
```



6.4 Creazione dell'eseguibile

I passi necessari affinchè possa essere creato un'applicazione eseguibile per Windows/Mac/Linux sono:

1. Aprire il progetto;
2. File > Build Settings > Add open scenes;
3. Selezionare Scenes/PoolGame Main;
4. Build;
5. Selezionare la cartella.

Riferimenti

- [1] *Ultraleap - XR Examples*. URL: <https://docs.ultraleap.com/unity-api/The-Examples/XR/index.html>.
- [2] *Ultraleap for Developers*. URL: <https://developer.leapmotion.com/unity>.
- [3] *Ultraleap Leap Motion*. URL: ultraleap.com/product/leap-motion-controller/.
- [4] *Meta Quest 2*. URL: <https://www.meta.com/it/quest/products/quest-2/>.
- [5] *Ultraleap - datasheet*. URL: https://www.ultraleap.com/datasheets/Leap_Motion_Controller_Datasheet.pdf.
- [6] *Ultraleap Gemini*. URL: <https://developer.leapmotion.com/about-gemini>.
- [7] *Unity - XR*. URL: <https://docs.unity3d.com/Manual/XR.html>.
- [8] *Unity - XR Plugin Management*. URL: <https://docs.unity3d.com/Manual/com.unity.xr.management.html>.
- [9] *Unity - XR Interaction Toolkit*. URL: <https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.3/manual/index.html>.