Alma Mater Studiorum · Università di Bologna

Scuole di Scienze
Laurea Magistrale in Scienze Informatiche - Tecniche del software

Impossible Space circolare per favorire la locomozione naturale

Relatore:
Prof. Gustavo Marfia

Presentata da:
Ulderico Vagnoni

Co-Relatore:
Prof. Luciano Bononi

Sessione
2021/2022

Indice

Abstract

In questa relazione di tesi verrà affrontato il problema della locomozione naturale in ambienti virtuali che possiedono una grandezza maggiore dello spazio reale nel quale l'utente si muove.

Negli anni sono stati sviluppati numerosi metodi di navigazione, ma tra tutti la locomozione naturale è il sistema che porta un livello di immersione maggiore nello spazio virtuale, più di qualsiasi altra tecnica.

In questo lavoro di tesi verrà proposto un algoritmo in grado di favorire la locomozione naturale all'interno di lunghi spazi chiusi, andando a modificare la geometria dello spazio virtuale (che andrà ad assumere una forma circolare) e mantenendo comunque un certo grado di realismo dal punto di vista dell'utente.

L'obiettivo è quello di tradurre uno spazio virtuale chiuso e lungo (caso comune: un corridoio) in una forma circolare inscritta in una stanza di 2x2 metri, con un approccio simile a quello degli *Impossible Spaces*, con l'obiettivo di studiare, in futuro, entro quale percentuale di sovrapposizione l'utente si accorge di trovarsi in una geometria impossibile.

Nel primo capitolo verranno introdotti i concetti chiave della VR e dell'AR, nonché un'introduzione all'Engine Unity e al software Blender. Nel secondo capitolo si tratterà di computer graphics, quindi si introdurranno i concetti base della grafica 3D con un focus sulla matematica alla base di ogni processo grafico. Nel terzo capitolo verrà affrontano il concetto di embodiment, quindi la percezione del proprio corpo, l'importanza dell'immersitvità dei sistemi virtuali e verrà descritto un paper precedentemente realizzato riguardante tale concetto. Nel capitolo quattro si parlerà del movimento all'interno dei sistemi di realtà virtuale, della locomozione naturale e delle tecniche per rendere tale metodo più accessibile. Nel capitolo cinque infine verrà descritto l'algoritmo realizzato e verranno mostrati i risultati.

Capitolo 1

Introduzione

Cos'è la realtà virtuale?
Tecnicamente un sistema di VR (Virtual Reality) è costituito da un insieme di dispositivi informatici in grado di consentire un nuovo tipo di interazione uomo-computer.

Generando un mondo virtuale e stimolando artificialmente i sensi, il corpo viene ingannato e viene indotto ad accettare tale mondo come una nuova visione della realtà, permettendo di sperimentare situazioni al limite del reale.
Storicamente, la nascita della realtà virtuale è un argomento discusso: già dal rinascimento, con lo sviluppo della prospettiva, è nato il concetto di spazio inesistente, ossia un mondo artificiale visivamente reale e possibile, ma fisicamente inesistente.
In maniera più concreta la nascita della realtà virtuale coincide con lo sviluppo di un primo sistema chiamato Sensorama, da qui in poi si sono susseguiti vari sistemi più o meno efficaci fino ad arrivare ai tempi attuali e alla tecnologia moderna.

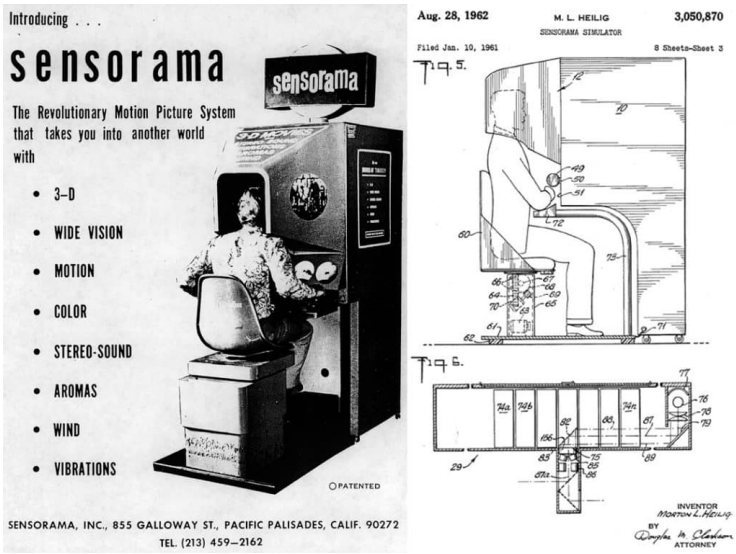


Figura 1.1: Sensorama, primo esempio di tecnologia immersiva e multi-sensoriale

La realtà virtuale è una potente tecnologia che promette di cambiare le vite di ogni persona, innescando una rivoluzione superiore alla nascita degli smartphone.
Tuttavia, l'entusiasmo nei suoi confronti decresce quando si inizia a pensare alle problematiche principali che comporta: il costo proibitivo dei sistemi hardware e l'incapacità degli ambienti di realtà virtuale di adeguarsi alle condizioni fisiche disponibili dall'utente medio sono le due problematiche principali.
Per poter godere di un'esperienza rilevante sono richiesti hardware estremamente potenti e visori di alta qualità, entrambi dal prezzo elevato, il che costituisce una forte limitazione alla diffusione della tecnologia.

Google Cardboard e Daydream sono state una parziale soluzione al problema dei costi. Questi sono progetti portati avanti da Google, che propone di utilizzare uno smartphone come dispositivo di elaborazione, display e input/output. Inserendo lo smartphone all'interno di un apposito supporto (più o meno sofisticato e dal costo irrisorio) è possibile avere tutto ciò che serve per accedere alla VR.

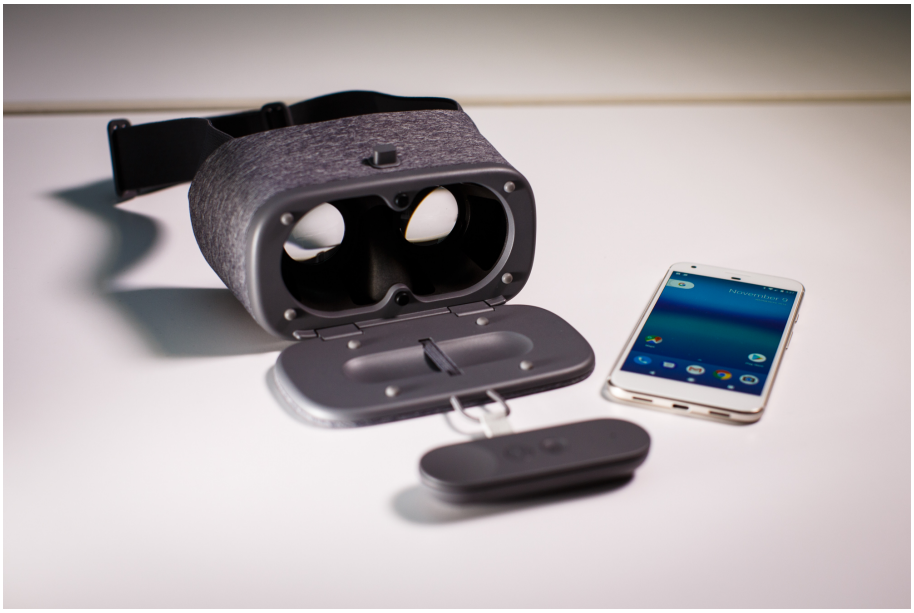


Figura 1.2: Daydream, un'alternativa economica ai moderni visori

Questa soluzione, sebbene estremamente economica, comporta significative problematiche. I telefoni non sono abbastanza potenti per elaborare ambienti virtuali di qualità e i sensori al loro interno in molti casi mancano dell'accuratezza necessaria a rendere

un sistema immersivo, il che rende questo approccio un esperimento più che un vero prodotto commerciale.

Nonostante queste difficoltà, nel corso degli ultimi anni si è assistito a un notevole sviluppo tecnologico nell'ambito della Virtual Reality (VR) e dell'Aumented Reality (AR), con la commercializzazione dei visori di nuova generazione (ad esempio Oculus Rift, HTC Vive e Varjo XR-3 per citare i più famosi) e gli investimenti fatti dalle principali aziende del settore (Google, Sony, Facebook, Microsoft) potrebbe essere l'inizio di una svolta, portando tali tecnologie alla portata di tutti.

1.1 Realtà mista: MR

Con l'espressione realtà mista (Mixed Reality) si intende tutto quello spettro di tecnologie che vanno dalla realtà aumentata a quella virtuale.
Il rapporto tra reale e virtuale determina la regione dello spettro della MR in cui ci si trova.
L'immagine sottostante, tratta da [16], rappresenta il concetto di Mixed Reality:

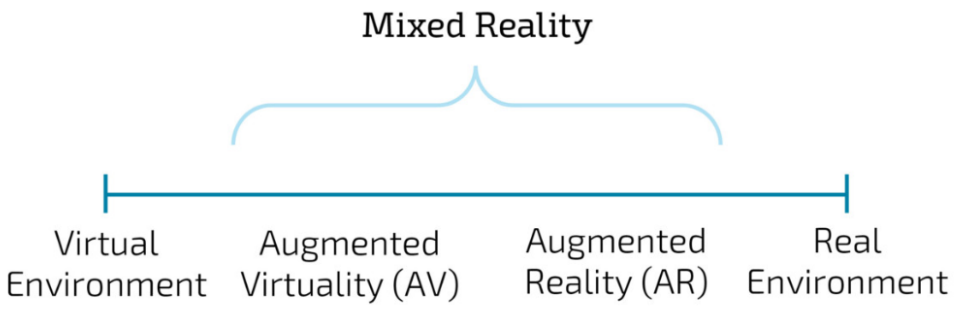


Figura 1.3: Schema riassuntivo del concetto di mixed reality

Se l'esperienza di Mixed Reality avviene nel mondo reale, il quale viene arricchito con elementi virtuali, allora si parlerà di realtà aumentata. Se l'esperienza invece avviene in un ambiente virtuale, e quindi il mondo reale è completamente rimpiazzato, allora si parlerà di realtà virtuale.

1.2 Realtà virtuale: VR

Volendo dare una definizione esaustiva di VR, possiamo definirla come l'indurre un soggetto a un comportamento mirato utilizzando una stimolazione sensoriale, senza che questo abbia consapevolezza dell'interferenza.

Il termine Virtual Reality è stato coniato da Jaron Lanier nel 1989, durante un'intervista con Kevin Kelly per la rivista Whole Earth Review.
Più recentemente, la Virtual Reality è stata concepita come un cluster di hardware e software in grado di creare animazioni 3D in tempo reale per HMD (Head Mounted Display), con le quali gli utenti interagiscono attraverso guanti e tute, con sensori che percepiscono la posizione e il movimento degli utenti nello spazio.

L'immersività è l'argomento principale della realtà virtuale e richiede continuità dell'ambiente, conformità alla visione umana, libertà di movimento e interattività.
Una volta assicurate tali caratteristiche, la realtà virtuale richiede un coinvolgimento narrativo in termini di coerenza nella trama e nella successione degli eventi: se ciò avviene, l'esperienza virtuale risultante risulta memorabile, qualunque sia il contesto in cui viene applicata.
L'immersione dipende da vari fattori, che sono la percezione dello spazio, la percezione del movimento ma sopratutto la percezione di un corpo, e tale concetto verrà approfondito nel terzo capitolo.

Per fruire della virtual reality, viene utilizzato un vasto range di dispositivi hardware, i principali sono HMD, guanti ed esoscheletri. Tutti vengono utilizzati secondo per lo stesso obiettivo: ingannare i sensi umani in modo da creare l'illusione che l'esperienza virtuale corrisponda alla realtà.
Tali strumenti devono quindi tener conto della fisiologia umana, se mancano di accuratezza e sincronismo l'esperienza perde credibilità, rischiando nel caso pessimo di far provare malessere fisico all'utente. Per esempio se le informazioni raccolte dal sistema vestibolare nell'orecchio umano non coincidono con quelle raccolte dall'apparto visivo, si sperimenta la così detta *motion sickness*.

È quindi necessario che le informazioni raccolte dai sensori siano tradotte istantaneamente nel mondo digitale, con una latenza impercettibile al cervello umano.

1.2.1 Hardware

Il primo passo per capire come la realtà virtuale funziona è il conoscere l'intero sistema VR.
Le componenti hardware fondamentali di un sistema VR sono:

- **Displays** (output): dispositivi che stimolano il senso della vista.
- **Sensori** (input): dispositivi che estraggono informazioni dal mondo reale.
- **Computers**: dispositivi che processano input e output sequenzialmente.

L'hardware produce stimoli che sovrascrivono i sensi dell'utente e utilizzando i propri sensori traccia i movimenti dell'utente al fine di regolare gli stimoli generati.
L'Head Tracking, ossia l'individuazione della posizione della testa nel mondo fisico, è il tracciamento più importante dei sistemi VR e ne rappresenta l'ingrediente fondamentale, ma un sistema può anche implementare il tracciamento degli occhi, ossia il tracciamento della direzione in cui l'utente osserva, dei controller, e quindi delle mani, o di qualsiasi altra parte del corpo.

Anche lo spazio fisico disponibile fa parte di un sistema VR. L'utente infatti, oltre agli stimoli generati dall'hardware, ottiene anche stimoli dal mondo reale.
Quindi, è erroneo pensare che l'harware e il software compongano un sistema VR, è invece ragionevole pensare che questo sia composto da hardware, software, l'organismo che fruisce tale esperienza e il mondo fisico che coesiste con il mondo virtuale.

Ogni organo di senso ha uno spazio di configurazione, che corrisponde a tutti i possibili modi in cui può essere configurato e spostato. Da qui la definizione di gradi di libertà (o DOF). Normalmente, un essere umano possiede 6 gradi di libertà, che corrispondono ai tre possibili movimenti (laterale, frontale, verticale) e le tre possibili rotazioni.
Lo stesso deve avvenire in un sistema VR immersivo attraverso il tracciamento dei movimenti.
In condizioni normali, il cervello controlla la configurazione degli organi di senso (occhi, orecchie, dita) quando riceve stimoli dal mondo esterno. Un sistema VR dirotta ogni senso rimpiazzando la stimolazione naturale con una stimolazione artificiale prodotta dall'hardware.
L'umano percepisce il mondo virtuale con i suoi occhi attraverso l'utilizzo di un display e con le sue orecchie attraverso uno speaker. Se il sistema è efficace, allora il cervello viene ingannato e l'utente crede che la stimolazione dei sensi è naturale e viene da un mondo plausibile.

Figura 1.4: Esempio di utilizzo di un visore HTC Vive Pro

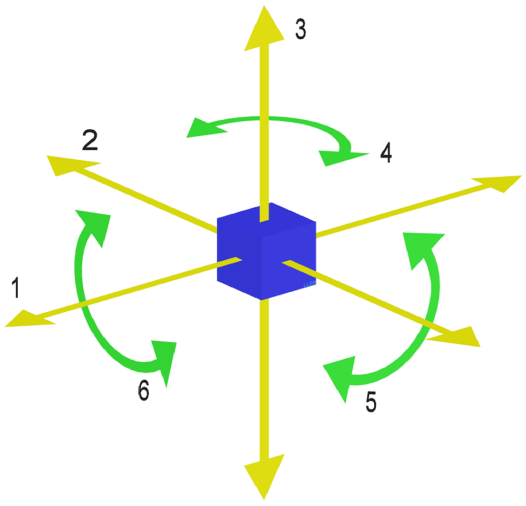


Figura 1.5: I 6 gradi di libertà possibili (6 DOF)

Audio

Quando si parla di sistemi audio, è corretto distinguere tra due configurazioni:

- **user-fixed**: dove il suono proviene da una posizione fissa con l'utente, come nel caso di una persona che ascolta suoni attraverso delle cuffie.

- **world-fixed**: dove invece il suono è prodotto da un sistema ambientale dove l'utente si trova, come nel caso di un sistema audio (per esempio un sistema Surround).

Nel caso di audio **user-fixed** quindi, si cerca di proiettare il suono direttamente nelle orecchie dell'utente, mentre nei sistemi world-fixed vengono generate fonti di suono nell'ambiente.
Entrambi i sistemi possiedono pregi e difetti (sistemi world-fixed sono più costosi e più invasivi nell'ambiente, mentre sistemi user-fixed possono essere invadenti per l'utente nei lunghi periodi), in ogni caso, in un sistema di realtà virtuale il suono deve essere gestito come nel mondo reale, mantenendo profondità, posizione e movimento del suono.

Video

Il senso della vista è molto più forte e dominante rispetto l'udito, e proprio per questo motivo è uno dei principali sensi da dover ingannare.
Negli anni sono stati sviluppati numerosi sistemi per permettere all'utente di osservare un ambiente virtuale, come il sistema VR CAVE o l'utilizzo di HMD.
Nel caso degli HeadSet, l'obiettivo da raggiungere è quello di aggiornare l'immagine visualizzata dall'utente quando questo muove la testa. Infatti, se l'utente muove la testa mentre indossa un HeadSet e l'immagine rimane la stessa, allora la percezione sarà quella di un'immagine attaccata alla sua testa.
Il tutto si traduce con un lavoro ingegneristico significativo, devono essere progettati sistemi in grado di stimare la quantità di movimento della testa e degli occhi e applicare le corrette trasformazioni in un corretto tempo e maniera.
Se ciò non avviene, l'utente sperimenta un'esperienza non convincente che può portare a emicranie e *motion sickness*.

Quindi, mentre l'audio non sembra essere una componente fondamentale dell'esperienza, la vista risulta un elemento critico.

Figura 1.6: CAVE, ambiente di realtà virtuale immersivo

1.2.2 Software

Dal punto di vista del software, la domanda più comune che ci si potrebbe chiedere è: quali componenti software sono necessarie per produrre un'esperienza VR?

Attualmente non esistono *engine* attraverso i quali creare sistemi VR, come lo sono per esempio i game engines per la creazione di videogames, come Unity o l'Unreal Engine, tuttavia, esistono SDK (Software Development Kit) resi disponibili per particolari Head-Set che facilitano il compito agli sviluppatori e che sono attualmente utilizzabili nei game engines, come per esempio:

- **OpenVR**: SDK alla base di SteamVR, la piattaforma per la realtà virtuale sviluppata da Valve.
- **Oculus SKD**: SDK per lo sviluppo di applicazioni destinate ai visori Oculus.
- **XR Toolkit**: SDK con il quale sviluppare applicazioni VR che permette la simulazione delle varie componenti del sistema.

Tutti questi sono disponibili come plug-in nel motore grafico Unity.
Ciò che si vuole realizzare è quindi un mondo virtuale, formato da modelli 3D (ognuno con il proprio materiale e proprietà fisiche che ne descrivono la sua interazione con luci,

suoni e forze), all'interno del quale vengono considerate le informazioni in input quali il tracking del movimento e gli input forniti dall'utente.

Figura 1.7: Meeting room in VRChat

Quindi, lato software, sono due gli obiettivi principali, la creazione di un mondo virtuale credibile e l'implementazione dell'interazione dell'utente con questo mondo.
Per quanto riguarda la creazione del mondo, questa può avvenire in numerosi modi, può essere generato grazie all'intervento di artisti e modellisti 3D, oppure può essere generato a partire dalle informazioni del colore e della profondità di camere utilizzando le tecniche SLAM (Simultaneous Localization And Mapping), oppure ancora, in modo procedurale attraverso codice.
Per quanto riguarda la corretta interazione tra il mondo virtuale e l'utente, l'obiettivo principale è quello di mantenere una corretta corrispondenza tra il movimento dell'utente nel mondo reale e quello virtuale.
All'interno del mondo reale, un utente si muove all'interno di una zona sicura che prende il nome di *matched zone*, una zona in cui l'utente può fisicamente muoversi e che può essere vista come un luogo dove mondo reale e virtuale sono perfettamente allineati.
Da qui derivano varie problematiche, per esempio la *matched zone* può essere significativamente più piccola della scena virtuale, oppure può essere piena di ostacoli che non permettono un corretto spostamento dell'utente al suo interno.

Questa stessa tesi è incentrata su tale problematica e nei capitoli successivi verranno descritte varie tecniche per attenuare la situazione.

In sostanza, maggiore sarà la *matched zone*, maggiore sarà la libertà dell'utente all'interno dello spazio virtuale. Altri metodi utilizzati per esplorare questi spazi virtuali con una *matched zone* eccessivamente piccola sono le così dette tecniche magiche, come il teletrasporto (tecnica molto utilizzata nei sistemi VR commerciali e mainstream), ossia delle tecniche che, seppur funzionando correttamente, rompono l'illusione all'interno del sistema virtuale, non mappando in maniera corretta il movimento reale dell'utente nella scena.

1.2.3 Stato dell'arte

Attualmente sono varie le tecnologie hardware disponibili per la realtà virtuale:

- **Visori**: i conosciuti HMD, ossia caschi di realtà virtuale attraverso i quali osservare e ascoltare la scena. Questi permettono il tracciamento della testa per tradurre ogni movimento nel mondo reale in un movimento nel mondo virtuale.
- **Guanti**: guanti provvisti di sensori che permettono di manipolare oggetti virtuali in modo naturale.
- **Sistemi di tracciamento**: sistemi che permettono di tracciare la posizione dell'utente nel mondo fisico. HeadSet diversi utilizzano sistemi di tracciamento diversi, per esempio l'HTC Vive fa utilizzo del sistema *Lighthouse*, ossia delle base stations che generano fasci di laser catturati da fotorecettori disseminati sul visore e sui controller che rilevano la posizione e la rotazione dell'oggetto tracciato nello spazio.
- **Esoscheletri**: utilizzati per il tracking full-body.
- **Sistemi motore**: come tapis-roulant omnidirezionali o le robot tiles.

I visori attualmente in commercio (o in sviluppo) sono molti e possono essere suddivisi in due categorie in base al prezzo:

- Tra i visori costosi quelli di riferimento sono Oculus Rift, HTC Vive, Playstation VR e Varjo. Questi visori rappresentano lo standard qualitativo più alto per la VR, montando sensori ad alta precisione e display ad alta fedeltà permettono un'esperienza fluida e realistica.
  Tutti questi necessitano di essere collegati a un computer per funzionare, è quindi indispensabile un'elevata potenza di calcolo.

Figura 1.8: Base station utilizzata dai visori HTC per il tracking del visore

Il problema che accomuna tutti questi dispositivi è appunto il prezzo. Infatti attualmente il più economico richiede almeno 400 dollari. Aggiungendo il costo di un computer di fascia medio alta al quale il visore deve collegarsi, allora il costo diventa eccessivamente alto.

- Tra i visori più economici invece si trovano il Google Cardboard, il Google Daydream e il Gear VR di Samsung.
  Questa categoria di visori è caratterizzata dall'assenza di display e sensori, sono infatti presenti solo delle lenti e uno spazio apposito in cui inserire lo smartphone. La grande ed evidente differenza rispetto ai visori più costosi è che per funzionare questi non necessitano l'utilizzo di un computer, ma unicamente di uno smartphone.
  Vista l'elevata diffusione degli smartphone è chiaro che la spesa per accedere alla VR è, in questo caso, unicamente il prezzo del visore, il quale varia tra i 10 e i 99 dollari.
  Dall'altro lato e modalità di input possibili sono limitate, infatti oltre al movimento della testa (tracciato attraverso l'utilizzo del giroscopio integrato nella stragrande maggioranza degli smartphone) è supportato unicamente un generico tocco sullo schermo.

Figura 1.9: (a) Un visore HTC Vive Pro (b) VR Manus (c) tapis-roulant omnidirezionale in un sistema CAVE

1.2.4 Aree applicative

Negli ultimi anni, grazie all'uscita di nuovi visori commerciali di grande qualità, si è riacceso l'entusiasmo da parte del pubblico nei confronti della Virtual Reality, in particolare nell'ambito del gaming.
Oltre a questo però, la realtà virtuale ha già trovato moltissime applicazioni in diversi ambiti:

- Architettura: per esempio l'*architectual visualization*, che permette di visionare il modello 3D di un progetto.
- Medicina: utilizzata per la formazione del personale medico, come terapia per disturbi psichici o ancora per la riabilitazione motoria e cognitiva.
- Militare: utilizzata come strumento per l'addestramento, come per l'addestramento del personale medico, la simulazione di un conflitto armato, la simulazione di volo per piloti e le simulazioni di guida.

Figura 1.10: Soldato impegnato in una simulazione in VR

1.3 Realtà aumentata: AR

L'espressione Augmented Reality (AR), come suggerisce il nome, indica una tecnologia in grado di combinare informazioni virtuali con il mondo reale, andando appunto ad *aumentarlo*. Realtà fisica e realtà virtuale quindi coesistono nello stesso istante agli occhi dell'utente.
Azuma riassume il concetto di AR, indicandone tre caratteristiche fondamentali:

- Combina oggetti virtuali con l'ambiente reale.
- Gli oggetti virtuali sono disposti coerentemente nell'ambiente reale, rispettandone quindi la geometria nella prospettiva dell'utente.
- L'esperienza è in tempo reale e interattiva.

Negli ultimi anni, istituti di ricerca, università e varie imprese hanno investito nella ricerca dell'AR, pubblicando molti articoli e risultati della ricerca scientifica.
Questi risultati dimostrano la fattibilità e l'innovazione dell'AR come tecnologia HCI (human-computer interaction).

Figura 1.11: Pokemon GO, la killer application dell'AR

Sulla base di queste conoscenze, è facile capire che, mentre nella realtà virtuale l'utente è completamente immerso nell'esperienza virtuale, nella realtà aumentata l'utente si trova nel mondo reale con l'aggiunta di modelli tridimensionali, e proprio per questo motivo un display di realtà aumentata dovrebbe essere in grado di visualizzare sia il mondo reale che quello virtuale contemporaneamente.
Sono necessarie due funzioni quando si lavora con questi due tipi di dati:

- **Registrazione**: sapere dove si trovano le cose (virtuali e reali) nello spazio per fare un buon allineamento dei dati virtuali con i dati reali.
- **Tracciamento**: tracciare e mantenere il movimento di cose o fotocamera.

Un esempio di applicazione AR è Pokemon Go, un'applicazione disponibile per smartphone molto famosa e lanciata nel 2016.

1.3.1 Stato dell'arte

Con il miglioramento della potenza di calcolo dei software e degli hardware, la realtà aumentata è gradualmente passata dalla fase di ricerca teorica del laboratorio alla fase di massa e applicazione nelle aziende.
Come nel caso della VR, oggi non esiste un vero e proprio engine per la realizzazione di applicazioni AR, ma degli SDK che ne facilitano lo sviluppo, come per esempio ARK, un SKD lanciato da Apple nel 2017 per creare applicazioni AR per iPhone e iPad, ARCore (la controparte Google) o Vuforia, SDK molto popolare utilizzato in Unity come plug-in.

Attualmente, le tecnologie di visualizzazione per esperienze AR sono due:

- **Hand-helded**: l'utente tiene in mano il dispositivo attraverso il quale fruisce dell'esperienza AR, che può essere uno smart-phone o un tablet.
- **Head-Mounted**: dispositivi *wearable* che possono essere indossati come dei normali occhiali, con il vantaggio di tenere le proprie mani libere.
  Questi a loro volta possono essere di due tipi:
  - **Optical See-Through** (OST): L'utente può vedere il mondo reale attraverso uno schermo semitrasparente mentre il sistema proietta su questo contenuti virtuali.
    La retina dell'occhio unisce quindi dati virtuali e reali senza considerare lo schermo.
    Esempi più famosi di questa tecnologia sono gli occhiali Hololens e i Google Glass.
  - **Video See-Through Technologies** (VST): L'immagine del mondo viene catturata da una videocamera, come quella dello smartphone, e viene combinata con il contenuto virtuale. Il risultato viene poi mostrato attraverso il display.
    Tale tecnologia è molto meno costosa e anche molto più semplice da costruire rispetto la precedente, tuttavia la vicinanza dell'occhio umano allo schermo del cellulare comporta un calo della definizione dell'immagine.
    Un esempio di tale tecnologia sono tutte quelle applicazioni di realtà aumentata fruibili attraverso il Gear VR.

1.3.2 Aree applicative

Come la VR, anche l'AR trove numerosissime applicazioni:

Figura 1.12: Hololens, headset sviluppato da Google

- Turismo e archeologia: mostrando informazioni riguardanti attrazioni turistiche o attraverso una ricostruzione tridimensionale di siti storici.
- Industria: per l'addestramento degli impiegati e per facilitare la manutenzione di macchinari (come nel caso del maintenance remoto dove un tecnico addestrato guida a distanza un operatore che fisicamente si trova nei pressi del macchinario).
- Militare: utilizzata sia per l'addestramento ma anche come supporto nelle fasi operative, come nel caso del *Battlefield Augmented Reality System* (BARS).
- Medicina: anche qui utilizzata sia per la formazione del personale medico, ma anche per riabilitazioni motorie e in sala operatoria per la localizzazione di vene e i dati del paziente.

Figura 1.13: Applicazione AR dell'IKEA, con la quale visualizzare i mobili dal catalogo direttamente nella propria abitazione

1.4 Unity

Unity è un game-engine creato appositamente per lo sviluppo di applicazioni 2D e 3D per PC, Mac, Android, iOS, Windows Phone e le principali console in commercio.
Il motore stesso attualmente supporta la creazione di applicazioni per più di 25 diverse piattaforme, inclusi dispositivi mobili, desktop, console e lo sviluppo di applicazioni in VR.
Con Unity 3D si possono realizzare videogiochi o altri contenuti interattivi, quali visualizzazioni architettoniche, simulazioni e ambientazioni tridimensionali e i linguaggi supportati dal sistema di scripting sono due: **Javascript** e **C#**.

Il software è disponibile in due versioni: una gratuita, utilizzata per progetti individuali o per piccole compagnie, e una a pagamento, utilizzata per lo sviluppo di applicazioni massicce.
Inoltre, è presente la funzionalità Unity Hub, ossia una *standalone application* che semplifica la creazione e la gestione dei propri progetti Unity e delle versioni scaricate. Unity Hub viene quindi utilizzato per:

- La gestione del proprio account e della licenza dell'editor.

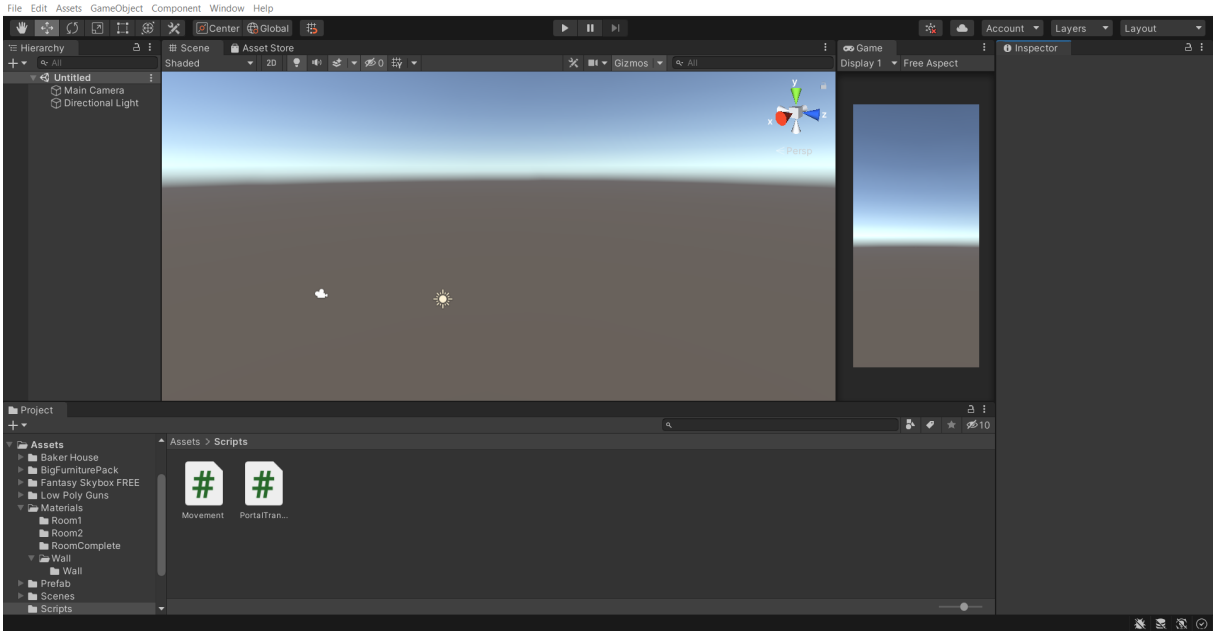


Figura 1.14: Schermata iniziale di Unity

- La creazione di un progetto e la selezione della versione dell'editor da utilizzare. È possibile inoltre selezionare un progetto template per velocizzare le tempistiche di sviluppo.

- Il download e l'installazione delle varie versioni di Unity.

Unity propone varie versioni che differiscono per supporto e funzionalità. Le versioni maggiormente utilizzate sono le LTS (long time support), ossia versioni più stabili e supportate che vantano la presenza di molti plug-in. Il motore grafico Unity è stato utilizzato all'interno di questa tesi per lo sviluppo del progetto. Perché è stato utilizzato Unity in questa tesi? Perché più del 60% di tutti i contenuti di Realtà Virtuale e Realtà Aumentata sono stati realizzati attraverso tale engine.

1.4.1 Editor e scripting

Una volta aperto Unity e creato un nuovo progetto viene aperto l'editor attraverso il quale visionare tutta la scena.
Questo è formato da varie componenti:

- **Scene**: Tab attraverso il quale visionare la scena. Questa può essere pensata come l'insieme degli oggetti creati e visualizzati sullo schermo prima che venga premuto il pulsante play.
- **Game**: Questo tab mostra l'evoluzione degli eventi nella scena. Una volta premuto il tasto play, vengono applicati gli script e la fisica a ogni oggetto presente nella scena.
- **Console**: La console viene utilizzata per il debugging della scena. Il comando più classico **Debug.Log(message)** stampa il messaggio inserito all'interno della console.
- **Hiearchy**: La gerarchia mostra tutti gli elementi che fanno parte della scena in ordine gerarchico (l'elemento figlio si troverà al di sotto dell'elemento padre e così via). Ogni elemento della gerarchia prende il nome di GameObject.
- **Project**: Da qui è possibile visionare tutti i file e le cartelle che compongono il progetto ed è possibile creare e modificare file di ogni tipologia (script, materiali, prefab, e via dicendo).
- **Inspector**: Tramite l'inspector è possibile visualizzare tutte le caratteristiche relative a un elemento della scena, quali le informazioni spaziali, la fisica applicata, il materiale o gli script assegnati.
- **Play**: Tramite questo tasto viene inizializzata la schermata Game, vengono quindi eseguiti gli script e applicate le varie componenti (come la gravità per esempio) agli elementi della scena.

Ogni elemento presente all'interno della scena prende il nome di **GameObject**, questi sono costituiti da componenti, che ne definiscono le proprietà fisiche e comportamentali. Tutto ciò che viene attribuito a un GameObject prende il nome di componente, compresi gli script. Di default, un GameObject possiede un *transform*, che ne rappresenta la posizione, la rotazione e la dimensione nello spazio.
Lo scripting rappresenta il cuore pulsante di Unity. Attraverso questo può essere fatta qualsiasi cosa.

Lo scripting di Unity supporta i linguaggi C#, Javascript e Boo, la maggior parte degli sviluppatori utilizza tuttavia il linguaggio C#. Inoltre Unity integra al suo interno l'IDE (Integrated Development Environment) VisualStudio C#.
Visual Studio è fornito di un proprio compilatore C# ed è possibile usarlo per verificare se sono presenti errori negli script.

Unity offre varie funzioni built-in per lo scripting, tra queste le principali sono:

- **Start()**: Questa funzione viene chiamata una volta, sola quando viene premuto il tasto play.
- **Update():** Questa funzione viene invece richiamata a ogni frame.
- **OnTriggerEnter():** Questa funzione viene chiamata ogni volta che un elemento provvisto di *rigidbody* entra in contatto con un elemento provvisto di un *trigger*.
- **OnCollisionEnter():** Questa funzione viene chiamata ogni volta che due elementi entrano in collisione.

Ogni volta che viene creato uno script, questo eredita automaticamente la classe **MonoBehaviour**.
Questa è una classe base per ogni script di Unity e permette di utilizzare le sopracitate funzioni.

1.5 Blender

Blender è un software di computer grafica gratuito e open-source.
Questo viene utilzzato per gli scopi più disparati nel contesto della computer graphics, come la modellazione 3D, l'animazione, il rendering di immagini tridimensionali e il rigging dei modelli.
Inoltre, con Blender è possibile realizzare anche applicazioni 3D e videogiochi.
Blender è stato sviluppato dallo studio NeoGeo e la prima versione (1.0) è stata rilasciata del gennaio 1994. A oggi, l'ultima version di Blender è la 3.3.

Blender vanta numerose modalità selezionabili dall'interfaccia principale.

Le due modalità più importanti sono:

- **Object Mode**: consente di modificare la posizione, la rotazione e la dimensione di un oggetto, considerato come un unico blocco.
- **Edit Mode**: permette di modificare le componenti di un oggetto (vertici, facce, lati). Tale funzionalità è selezionabile premendo il tasto *Tab* mentre si è nella Object Mode.

Inoltre, sono presenti altre modalità secondarie attivabili con oggetti di tipo mesh, come la *Sculpt Mode*, utilizzata per la scultura tridimensionale, la *Vertex Paint* per la

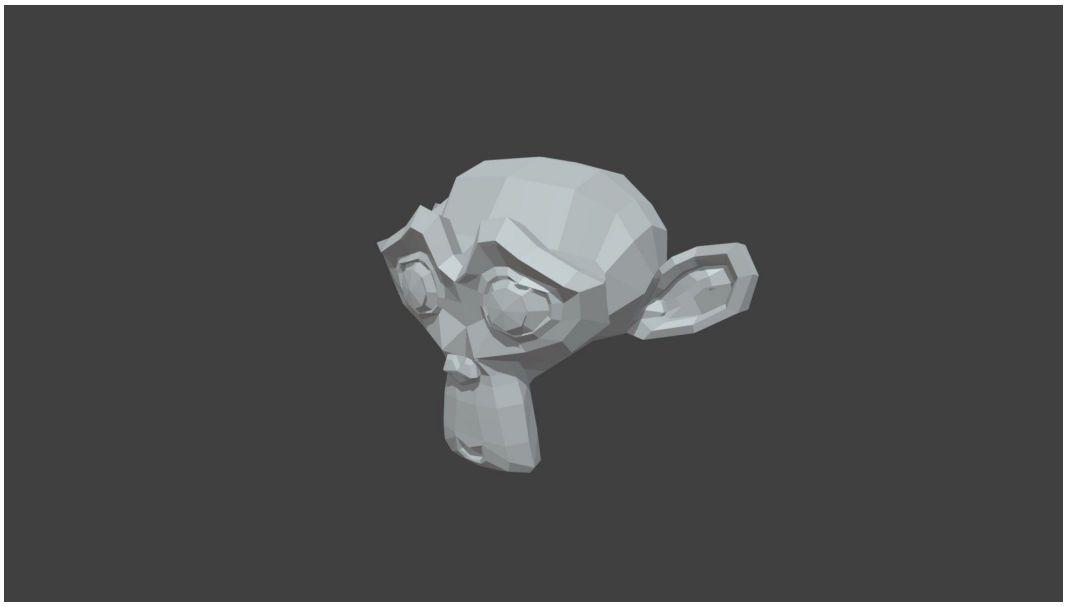


Figura 1.15: Un modello 3D di una scimmia, molto conosciuto in Blender

colorazione dei vertici, la *Texture Paint Mode* per la pittura della texture di un modello, la *Weight Paint Mode* per l'assegnazione di un peso a ogni vertice e molte altre ancora.

Capitolo 2

Computer graphics

Nel paragrafo precedente è stato introdotto il concetto di mondo virtuale, ossia un mondo artificiale visibile attraverso un display che possiede la sua geometria e la sua fisica.
Ma come viene generato e visualizzato questo mondo?
Per capire questo concetto e prima di passare al capitolo successivo, è bene introdurre il concetto di **computer grafica** e fornire tutte le nozioni necessarie a comprendere gli argomenti trattati nelle sezioni successive.

Banalmente, la computer graphics può essere definita come l'arte di disegnare immagini, linee e molto altro sullo schermo del computer utilizzando linguaggi di programmazione.

Le immagini possono essere di due tipologie:

- **Raster**: rappresentate da una griglia di pixel, l'immagine viene creata quindi accendendo dei pixel. Molto più utile nel caso di sfumature, tuttavia nel caso di operazioni come scaling, editing o ration si perde qualità.

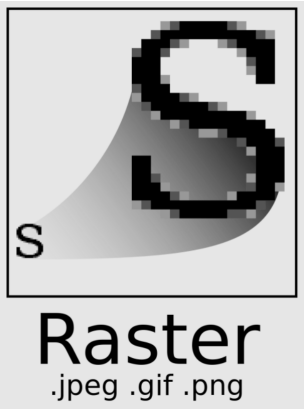


Figura 2.1: Immagine raster

- **Vector**: l'immagine viene rappresentata a partire da un comando, non ci sono istruzioni sui pixel in un file vector ma formule matematiche che catturano una

forma e costruiscono un'immagine.
L'immagine quindi risulta più nitida e tale tipologia è molto più comune nella progettazione e nel CAD. Al contrario della grafica raster, non permette sfumature ma permette un ottimo scaling e overlapping.

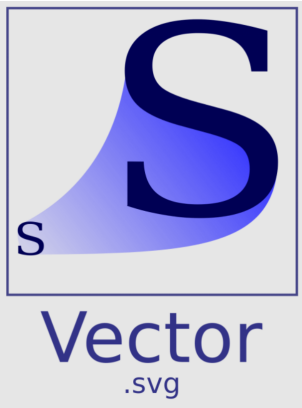


Figura 2.2: Immagine vector

La computer graphics ha un'impronta massiccia sulla vita di ogni giorno, essendo alla base dell'industrial design (costruzione di aerei, macchine, barche, e così via), del medical imaging (basti pensare a una risonanza magnetica), della visualizzazione scientifica, fino ad arrivare al mondo dell'arte e dell'intrattenimento (come il settore del cinema o video ludico).

Alla base della computer grafica c'è un sistema grafico, questo è composto da una componente hardware che rappresenta la potenza del sistema grafico e da una componente software che gestisce le risorse hardware per manipolare i dati grafici e produrre immagini.
L'hardware è composto da:

- **Input devices**: tutte quelle periferiche di input utilizzate per interagire con il sistema, come mouse, tastiera, scanner e via dicendo.
- **CPU e GPU**: processori utilizzati per il processo grafico.
- **Frame buffer**: una memoria RAM che immagazzina le immagini prima di mostrarle sul display.
- **Output devices**: dispositivi raster utilizzati per la visualizzazione delle immagini prodotte nel processo grafico.

Dal punto di vista software si può affrontare la grafica in vari livelli:

- **Application**: le varie applicazioni 3D che richiamano le API per interfacciarsi con il sistema operativo.
- **API**: Application Programming Interface, ossia un'interfaccia per programmare che comunica con il sistema operativo, come DirectX o OpenGL.
- **Sistemi operativi**: Windows, Mac, OS e così via.
- **Driver**: un primo software che interfaccia la scheda grafica con un sistema operativo.

Virtualmente, tutti i moderni sistemi grafici sono *raster based*, ossia l'immagine prodotta dalla pipeline grafica viene visualizzata attraverso l'accensione di pixel con un determinato colore.
I pixels sono contenuti in una parte della memoria RAM chiamata frame buffer, questo può essere visto come il cuore dei sistemi grafici.
La sua risoluzione, ossia il numero di pixel che il frame buffer può contenere, determina il livello di dettaglio osservabile in un'immagine.
La depth, o precisione di un frame buffer, definita come il numero di bits che sono utilizzati per ogni pixel, indica i piani disponibili per il frame buffer. Questi determinano proprietà come quanti colori possono essere rappresentati su un dato sistema. Per esempio, un frame buffer di profonidità a 1 bit permette solo due colori, mentre un frame buffer di profondità 1 byte permette la rappresentazione di 2^8 colori (256).
In un frame buffer **full color** ci sono 24 bits per pixel, tale sistema prende anche il nome di **true-color system**, o meglio conosciuto come **RGB-color system**, perché gruppi individuali di bits per ogni pixel assumono uno dei tre colori primari. Sistemi HDR (High Dynamic Range) utilizzano 12 o più bits per ogni componente di colore.
In un sistema semplice, il frame buffer contiene solamente i pixel colorati che sono mostrati sullo schermo. In sistemi più complessi invece, il frame buffer contiene molte più informazioni, come l'informazione della profondità, necessaria per creare immagini da dati tridimensionali.
In questi sistemi il frame buffer è composto da buffer multipli, uno o più dei quali sono color buffers che contengono i pixel colorati che sono mostrati.
La performance della gestione dell'immagine viene calcolata attraverso il **refresh rate**, questo, misurato in Hz, rappresenta il numero di frame (fotogrammi) che sono visualizzati sul display in un secondo. Se questi sono troppo pochi l'occhio umano noterà l'intervallo tra un frame e l'altro e percepirà un effetto di *flickering* sul display. Diverso è il frame rate, anche conosciuto come **frame frequency** e **frame per second (FPS)**,

Figura 2.3: Screen tearing durante una schermata di gioco

che viene utilizzato come benchmark per gli algoritmi utilizzati e corrisponde al numero di fotogrammi *diversi* prodotti dalla scheda grafica, quindi misura l'abilità della scheda grafica di disegnare un numero di frames nel display ogni secondo.
Se questi due rate non sono consistenti l'uno con l'altro si crea un effetto chiamato **screen tearing**, ciò accade quando non è stato ancora calcolato un nuovo frame ma viene comunque letto, quindi si hanno due fotogrammi diversi sullo stesso video.

In un semplice sistema grafico c'è solo un processore, la CPU, il quale deve sia eseguire il normale processamento che il processamento grafico. La principale funzione grafica del processore è di prendere specifiche di primitive grafiche (come linee, cerchi e poligoni) generate da application programs e assegnare valori ai pixel nel frame buffer che meglio rappresentano tali entità.
Per esempio, un triangolo è specificato dai suoi tre vertici, ma per visualizzare i tre lati che collegano i vertici, il sistema grafico deve generare una serie di pixel che appaiono come segmenti lineari allo spettatore.

La conversione di entità geometriche in pixel colorati e posizioni nel frame buffer è conosciuta come **rasterization**. Nei primi sistemi grafici il frame buffer era parte della memoria standard che poteva essere direttamente la CPU. Oggi, virtualmente ogni sistema grafico è caratterizzato dalla presenza di una *graphic processing unit*, o meglio

conosciuta come GPU, utilizzata per specifiche funzioni grafiche. La GPU può essere sia nella scheda madre che nella scheda grafica.
Le GPU si sono evolute fino a diventare più complesse della CPU. Queste sono caratterizzate sia da specifici moduli orientati alle operazioni grafiche che da un elevato livello di parallelismo: le GPU recenti contengono oltre 100 unità di elaborazione ciascuna delle quali è programmabile dall'utente.

2.1 Geometria

Qualsiasi geometria 3D creata con le modalità di modeling più disparate viene rappresentata nella pipeline di rendering come un flusso di triangoli.
Gli oggetti tridimensionali vengono quindi rappresentati utilizzando primitive lineari:

- Punti
- Linee, segmenti
- Piani, poligoni

Il fulcro delle trasformazioni geometriche è la possibilità di lavorare con tali primitive che vengono visualizzate all'interno di un sistema di coordinate.
Questo sistema può essere di due tipologie:

- **Destrorso**: sistema di coordinate utilizzato per la modellazione, anche chiamato coordinate mondo. In questo sistema, l'asse **z** è uscente dallo schermo mentre l'asse **y** è diretto verso l'altro. Per avere le idee più chiare, basti considerare la propria mano destra, il pollice indica l'asse **x**, l'indice l'asse **y** e il medio l'asse **z**.
- **Sinistrorso**: sistema di coordinate utilizzato per la prospettiva, quindi per le coordinate della telecamera impiegata nella visualizzazione della scena. Il discorso è analogo al destrorso ma con la mano sinistra appunto.

All'interno di questi sistemi di riferimento si lavora con:

- **Scalari**: valori che specificano una quantità reale.
- **Punti**: coordinate che specificano un luogo nello spazio (o nel piano).
- **Vettori**: una quantità formata da due attributi: magnitudine e direzione.

Il vettore solitamente ha un'ambiguità di rappresentazione rispetto al punto. Punti e vettori infatti si rappresentano in R^2 con due coordinate e R^3 con tre coordinate.

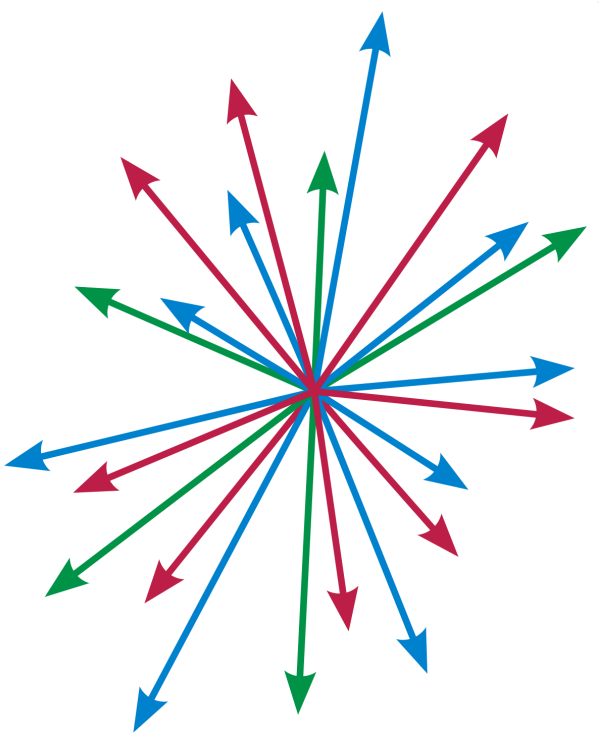


Figura 2.4: Spazio vettoriale *n-dimensionale*

Lo spazio in cui tutte queste componenti coesistono prende il nome di spazio vettoriale (o lineare), questo contiene due entità: i vettori e gli scalari. Le operazioni disponibili nello spazio vettoriale sono la somma vettoriale e la moltiplicazione di un vettore per uno scalare.

La somma vettoriale può essere eseguita sia attraverso la regola del parallelogramma che in forma algebrica (sommare le componenti corrispondenti):

$$a = [a_x a_y a_z]$$

$$b = [b_x b_y b_z]$$

$$a + b = [a_x + b_x a_y + b_y a_z + b_z]$$

All'interno di uno spazio vettoriale, dato un insieme di vettori, questi si dicono linearmente indipendenti se e solo se non esiste nessuna combinazione lineare attraverso il

quale ottenere uno dei vettori attraverso gli altri, altrimenti prendono il nome di linearmente dipendenti. Una combinazione lineare è un'espressione dove appaiono somme di vettori e moltiplicazioni tra vettori e degli scalari. Dato $v_1, v_2, ..., v_n \in V$ un insieme di vettori, una combinazione lineare può essere definita come:

$$a_1v_1 + a_2v_2 + a_2 + ... + a_nv_n$$

In uno spazio vettoriale il numero massimo di vettori linearmente indipendenti è fissato ed è chiamato dimensione dello spazio. In uno spazio n-dimensionale, un insieme di n vettori linearmente indipendenti è detto una base dello spazio. Data una base $a_1, a_2, ..., a_n$ in uno spazio vettoriale di dimensione n, un qualsiasi vettore che fa parte di tale spazio può essere espresso come:

$$v = \alpha_1a_1 + \alpha_2a_2 + ... + \alpha_na_n$$

Una base opportuna per rappresentare un qualsiasi punto dello spazio prende il nome di sistema di coordinate. Un'altra operazione molto utilizzata è il prodotto scalare, dati due vettori è possibile moltiplicarli tra loro, questa operazione restituisce come risultato appunto uno scalare:

$$a = (a_x, a_y)$$

$$b = (b_x, b_y)$$

$$< a, b >= a_xb_x + a_yb_y$$

Dove:

$$< a, b >= a \times b = \sum_{i=1}^{n} a_ib_i \in R^n$$

Il prodotto scalare induce una norma euclidea, anche detta magnitudine, ossia un modo di misurare l'entità dello spazio (la lunghezza del vettore). La norma euclidea è definita come la radice quadrata del prodotto scalare di un vettore a con sé stesso:

$$||a||_2 = \sqrt{< a, a >} = \sqrt{a_ta} = \sqrt{\sum_{i=1}^{n}(a_i)^2}$$

La magnitudine di un vettore in R3 è definita come:

$$||a||_2 = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

Un vettore di lunghezza pari a uno è detto vettore unitario. Un vettore inoltre può essere normalizzato, ossia è possibile ottenere un vettore di lunghezza pari a uno con la stessa direzione e stesso verso:

$$\frac{a}{||a||_2}$$

Dati due vettori, è possibile inoltre utilizzare il prodotto scalare per ottenere l'angolo tra due vettori:

$$a \times b = ||a||||b||cos\theta$$

Il prodotto scalare è distributivo, commutativo e bilineare. Avendo queste conoscenze è possibile andare a proiettare un vettore per ottenere una proiezione ortogonale. Dati due vettori b e c, allora è possibile definire un vettore a con la stessa direzione e verso di c che rappresenta la proiezione di b su c. Se c è un vettore non unitario allora la proiezione viene calcolata come:

$$||a|| = \frac{< b, u >}{||u||}$$

Altrimenti, se c è un vettore unitario allora si ha:

$$a =< b, u > u$$

Il prodotto scalare fornisce informazioni riguardo la relazione tra due vettori:

- Se $a \times b > 0$ allora $\theta < 90$
- Se $a \times b < 0$ allora $\theta > 90$
- Se $a \times b = 0$ allora $\theta = 90$

Un'altra operazione molto importante è il *cross product*, o anche detto prodotto vettoriale. Questo, dati due vettori a e b, restituisce il vettore normale al piano generato da a e b:

$$a = \begin{bmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{bmatrix}$$

$$aXb = [a_y b_z - a_z b_y a_z b_x - a_x b_z a_x b_y - a_y b_x]$$

Lo spazio che rappresenta punti, vettori e scalari prende il nome di spazio affine. Questo non è altro che un'estensione di uno spazio lineare che include i punti, perciò possiede due operazioni aggiuntive:

- Somma tra punto e vettore: se un punto viene sommato a un vettore, questo viene spostato e si calcola una nuova posizione.
- Differenza tra punti: la differenza tra due punti restituisce un vettore.

All'interno di uno spazio affine è possibile definire una combinazione affine, ossia una combinazione lineare di punti con coefficienti la cui somma è 1:

$$P = a_1 P_1 è a_2 P_2 + ... + a_n P_n$$

con:

$$a_1 + a_2 + ... + a_n = 1$$

I coefficienti $(a_1, a_2, ..., a_n)$ sono detti coordinate baricentriche di P nello spazio affine. Se tutti i coefficienti sono nell'intervallo $[0, 1]$ allora prendono il nome di coordinate convesse.

Dati tre punti fissi A,B,C è possibile definire un punto P come combinazione affine dei tre punti:

$$P = \alpha A + \beta B + \gamma C$$

Dove *A,B,C* non sono sulla stessa linea e $\alpha, \beta, \gamma \in R$. Se la somma delle tre coordinate baricentriche è pari a 1, allora il punto P si troverà all'interno della circonferenza. Le coordinate baricentriche vengono calcolate come:

$$P = \frac{< P, B, C >}{A, B, C} A + \frac{< P, C, A >}{A, B, C} B + \frac{P, A, B}{A, B, C} C$$

L'insieme di tutti i punti che passano per un punto P_0 nella direzione del vettore v sono rappresentabili tramite la formula:

$$l(t) = P_0 + tv$$

In cui $t \in (-\infty, \infty)$ per la retta e $t \in (0, \infty)$ per il raggio. Questa definizione rappresenta la formula parametrica per la rappresentazione di una retta. Un piano π viene definito da una normale n e da un punto del piano P_0, quindi un punto Q appartiene al piano se e solo se il prodotto scalare tra la differenza di Q e P_0 e la normale è uguale a zero, ossia la normale è perpendicolare a ogni vettore del piano.

Un sistema di riferimento, o anche chiamato frame, è definito dalla quadrupla:

$$F = (P_0, v_1, v_2, v_3)$$

Ossia un punto P_0, anche detto origine e tre vettori, che prendono il nome di assi. Ogni punto del sistema può essere definito a partire da una tripla (a_1, a_2, a_3):

$$P = P_0 + a_1 v_1 + a_2 v_2 + a_3 v_3$$

Punti e vettori vengono rappresentati entrambi in R^3 attraverso una tripla, il che li rende ambigui. Per distinguerli vengono utilizzate delle coordinate omogenee, in cui viene aggiunto uno scalare alla definizione di punto o retta che può assumere valore 1, ossia indicare un punto, o valore 0 viceversa. Perciò, punti e vettori verranno rappresentati come una quadrupla e non più come una tripla. A questo punto, un'operazione molto importante e strategica è quella di saper cambiare sistema di riferimento, ossia rappresentare un punto P attraverso due frame differenti, quindi nel caso di un sistema (x, y):

$$P = (x_p, y_p) = o + x_p x + y_p y$$

Mentre nel caso del frame (u, v):

$$P = (u_p, v_p) = e + u_p x + v_p y$$

Tale trasformazione avviene in forma matriciale descrivendo il frame:

$$\begin{bmatrix} x_p \\ y_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & x_v & x_e \\ y_u & y_v & y_e \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_p \\ v_p \\ 1 \end{bmatrix} = P_{xy} = \begin{bmatrix} u & v & e \\ 0 & 0 & 1 \end{bmatrix} P_{uv}$$

All'interno dei sistemi di riferimento, gli oggetti possono subire delle trasformazioni geometriche. Queste cambiano la posizione, l'orientamento e la dimensione degli oggetti nel sistema. L'obiettivo è quindi quello di modificare la geometria ma non la topologia. Le trasformazioni geometriche conosciute sono:

- Traslazione:
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Rotazione:
  - Attorno all'asse x:
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

  - Attorno all'asse y:
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 0 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

– Attorno all'asse z:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Scala:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

2.1.1 Mesh poligonali

Una mesh poligonale è una collezione di poligoni con un certo numero di facce che formano la pelle (skin) dell'oggetto ed è il metodo più comune di rappresentare oggetti tridimensionali in computer graphics.
Tipicamente le facce di una mesh sono triangolari, quindi vengono rappresentate attraverso l'utilizzo di tre vertici, ma possono anche essere quadrilateri. Tuttavia è possibile che i quattro vertici non siano sullo stesso piano il ché rende difficile il processo di *tessellation*, per questo motivo vengono prediletti i triangoli.
Alla base di una mesh poligonale vi è un compromesso tra l'accuratezza (maggiore è il numero di triangoli, migliore sarà il risultato) e la velocità di renderizzazione (maggiore è il numero di triangoli, maggiore sarà il tempo necessario a un qualsiasi algoritmo per renderizzarli).

Le mesh vengono utilizzate nei campi più disparati, per esempio nel medical imaging la strumentazione acquisisce dei volumi di dati che devono essere resi visibili dall'esterno attraverso l'utilizzo di mesh, oppure nella rappresentazione territoriale, dove i dati territoriali vengono rappresentati tridimensionalmente, e così via.

La generazione di una mesh avviene in tre modi:

- **Generazione a partire da punti**: vengono acquisiti una serie di punti che descrivono la mesh, tale processo prende il nome di *triangolazione*. Se si è interessati a generare anche l'interno di una mesh allora si utilizza la *tetrahedralization* e gli elementi non sono più facce triangolari ma dei tetraedri.

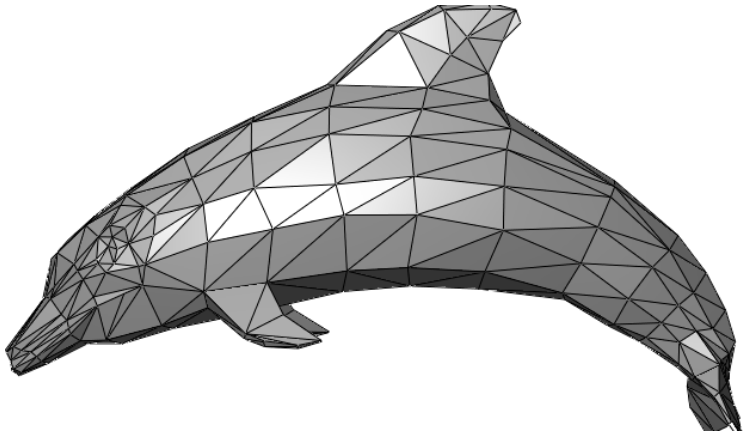


Figura 2.5: Esempio di tessellation

- **Generazione da superfici**: la superficie viene spezzata in triangoli che poi vengono essere passati alla pipeline di rendering, tale processo prende il nome di *tessellation*.
- **Generazione attraverso volumi di dati**: un volume di dati e viene ricreata la mesh che rappresenta una superficie a un solo valore, tale processo prende il nome di *polygonalization*.

Una mesh si dice **non strutturata** (o irregolare) quando i poligoni che la compongono sono diversi tra di loro, per esempio triangoli di dimensione differenti e di conseguenza angoli diversi. Si dice invece **semi-regular** quando viene ottenuta a seguito di una suddivisione regolare di una mesh irregolare, tutti i vertici sono quindi regolari a eccezione di un ridotto numero di vertici, e infine **regolare** quando tutti i vertici interni sono circondati da un numero costante di elementi, quindi tutti i poligoni hanno la stessa dimensione.

Una mesh è composta da una tripla (V, E, F) dove:

- Vertici(V): punti nello spazio $V = 1, ..., N_v$
- Edges (E): linee che collegano due vertici $E = (i, j) \in VxV : X_j \in N(X_i)$
- Faces (F): triangoli generati a partire da vertici e lati

  $F = (i, j, k) \in VxVxV : (i, j), (i, k), (k, j) \in E$

Essendo la mesh una rappresentazione lineare di una superficie, nel caso in cui si volesse rappresentare una sfera si andrà ad aumentare il livello di dettaglio, utilizzando quindi

dei triangoli sempre più piccoli che rappresentano la mesh (supponendo di avere una mesh regolare dove tutti i triangoli sono della stessa grandezza) e approssimando così la forma della sfera. Quindi più le facce della mesh sono piccole, maggiore è l'accuratezza di rappresentazione.
Questa apporssimazione ha un'errore dell'ordine

$$O(h^2)$$

.

2.1.2 Geometria procedurale

Per geometria procedurale si intende la generazione di una geometria (una mesh) attraverso un insieme di regole, o un algoritmo che, nel caso di Unity, è realizzato attraverso uno script.
All'interno di questo progetto di tesi, il concetto di mesh procedurale è stato largamente utilizzato, in quanto lo stesso progetto ha richiesto la modifica di una mesh attraverso l'utilizzo di codice.
Il concetto di geometria procedurale è largamente conosciuto specialmente dell'ambito video ludico, dove l'utilizzo di geometrie generate a *run-time*, utilizzando specifici algoritmi, ha permesso di creare mondi di gioco infiniti e sempre diversi.

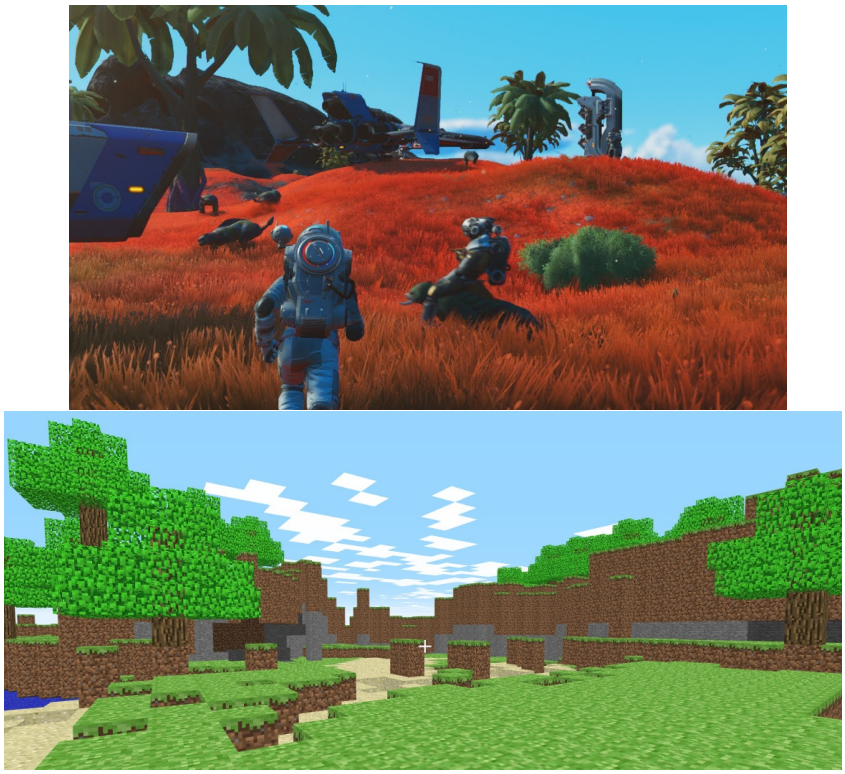


Figura 2.6: (a) No-Man-Sky (b) Minecraft, esempi di videogiochi che sfruttano geometrie procedurale per la generazione di mondi

All'interno di Unity, la mesh di un GameObject viene visualizzata nella scena grazie a due componenti:

- **MeshFilter**: componente che indica quale mesh renderizzare.
- **MeshRenderer**: componente che renderizza la mesh indicata dal **MeshFilter**.

Inoltre, **Mesh** è una classe in Unity che permette di creare e modificare mesh attraverso script.
Come specificato nella sezione precedente, una mesh è descritta da vertici, spigoli e facce, nel caso di Unity si fa riferimento unicamente ai vertici e alle facce (che in questo caso sono necessariamente triangoli).
Per creare una mesh dal nulla in Unity, la pipeline da seguire è sempre la stessa:

- Definire i vertici.
- Assegnare i vertici.
- Assegnare i triangoli.

Nel caso in cui si voglia generare un semplice triangolo, il codice è il seguente:

```
public class Triangle : MonoBehaviour
{

    Vector3[] vertices;
    int[] triangles;

    void Start()
    {
        vertices = new Vector3[3];
        triangles = new int[3];

        vertices[0] = new Vector3(0, 0, 0);
        vertices[1] = new Vector3(1, 0, 0);
        vertices[2] = new Vector3(0, 0, 1);

        triangles[0] = 0;
        triangles[1] = 1;
        triangles[2] = 2;

        Mesh mesh = new Mesh();
        gameObject.AddComponent<MeshFilter>();
        gameObject.AddComponent<MeshRenderer>();
        gameObject.GetComponent<MeshFilter>().mesh = mesh;
        mesh.vertices = vertices;
```

```
25          mesh.triangles = triangles;
26      }
27  }
```

E il risultato dell'esecuzione è il seguente:

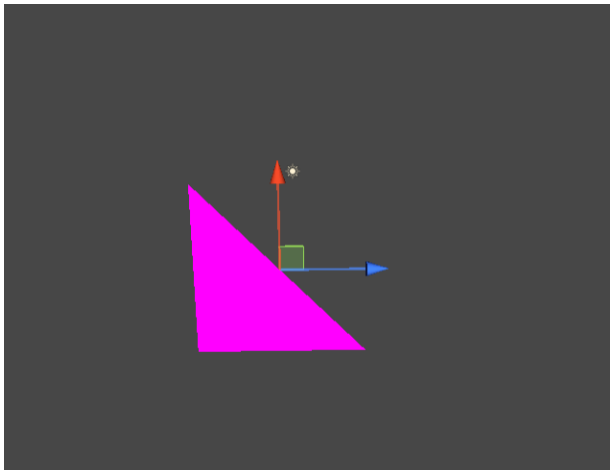


Figura 2.7: Triangolo privo di materiale, risultato del codice

La visualizzazione della mesh avviene seguendo il senso orario dei vertici, occorre quindi definire i triangoli considerando i vertici in senso orario dal punto di vista dell'osservatore, altrimenti questi verranno renderizzati dal lato opposto.

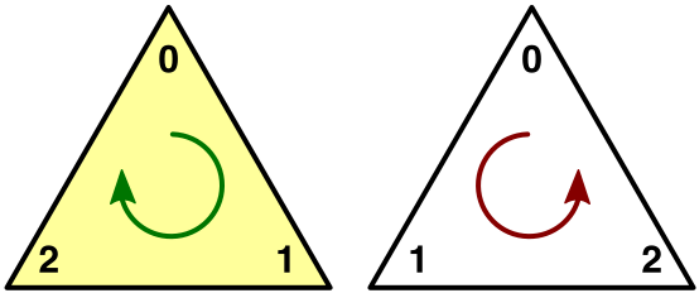


Figura 2.8: Ordine di definizione dei triangoli

Inoltre, le mesh in Unity sono descritte da altre due componenti fondamentali:

- **Normali**: le normali di ogni vertice utilizzate per calcolare dove la luce incide sulla mesh.
- **UV**: proiezione di una texture 2D su un modello 3D.

La geometria procedurale non si limita alla creazione di mesh, ma può essere utilizzata anche per la modifica di queste, come è appunto avvenuto in questa tesi.
Per esempio, dato un cubo sufficientemente complesso, quindi con un numero non banale di vertici, è possibile modificarlo in una sfera di raggio 1 metro.
Il codice è il seguente:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System.Linq;

public class Rounded : MonoBehaviour
{

    public Mesh mesh;
    Vector3[] vertices;
    int[] triangles;
    Vector3[] normals;

    void Awake()
    {
        mesh = gameObject.GetComponent<MeshFilter>().mesh;
        vertices = mesh.vertices;
        triangles = mesh.triangles;

        for(int i = 0; i < vertices.Length; i++)
        {
            vertices[i] = pointAlongSphere(transform.TransformPoint(
   vertices[i]));
        }

        mesh.vertices = vertices;
        mesh.triangles = triangles;
        mesh.RecalculateNormals();
        mesh.triangles = mesh.triangles.Reverse().ToArray();

    }

    Vector3 pointAlongSphere(Vector3 pointVert)
```

```
33      {
34          Debug.Log(pointVert.x);
35          float t1 = Mathf.Sqrt(1 / (Mathf.Pow(pointVert.x,2) + Mathf.Pow
    (pointVert.y, 2) + Mathf.Pow(pointVert.z, 2)));
36          float t2 = -Mathf.Sqrt(1 / (Mathf.Pow(pointVert.x, 2) + Mathf.
    Pow(pointVert.y, 2) + Mathf.Pow(pointVert.z, 2)));
37
38          Vector3 pointOne = new Vector3(pointVert.x * t1, pointVert.y *
    t1, pointVert.z * t1);
39          Vector3 pointTwo = new Vector3(pointVert.x * t2, pointVert.y *
    t2, pointVert.z * t2);
40
41
42          float distance1 = Vector3.Distance(pointOne, pointVert);
43          float distance2 = Vector3.Distance(pointTwo, pointVert);
44
45          if(distance1 < distance2)
46          {
47              return pointTwo;
48          }
49          else
50          {
51              return pointOne;
52          }
53      }
54  }
```

Questo semplice script, data una mesh rappresentante un cubo, proietta tutti i vertici del cubo sulla sfera di raggio 1, modificando la posizione dei vertici originali per assegnargli una nuova posizione.
Ciò avviene in maniera molto semplice, data l'equazione parametrica della retta passante per due punti (un punto corrispondente al vertice e un punto rappresentante il centro della sfera) vengono calcolati i due punti di intersezione tra la retta e la sfera, andando a considerare infine solo il punto più vicino (calcolato attraverso la distanza cartesiana tra due punti) al vertice originale.

Il risultato è il seguente:

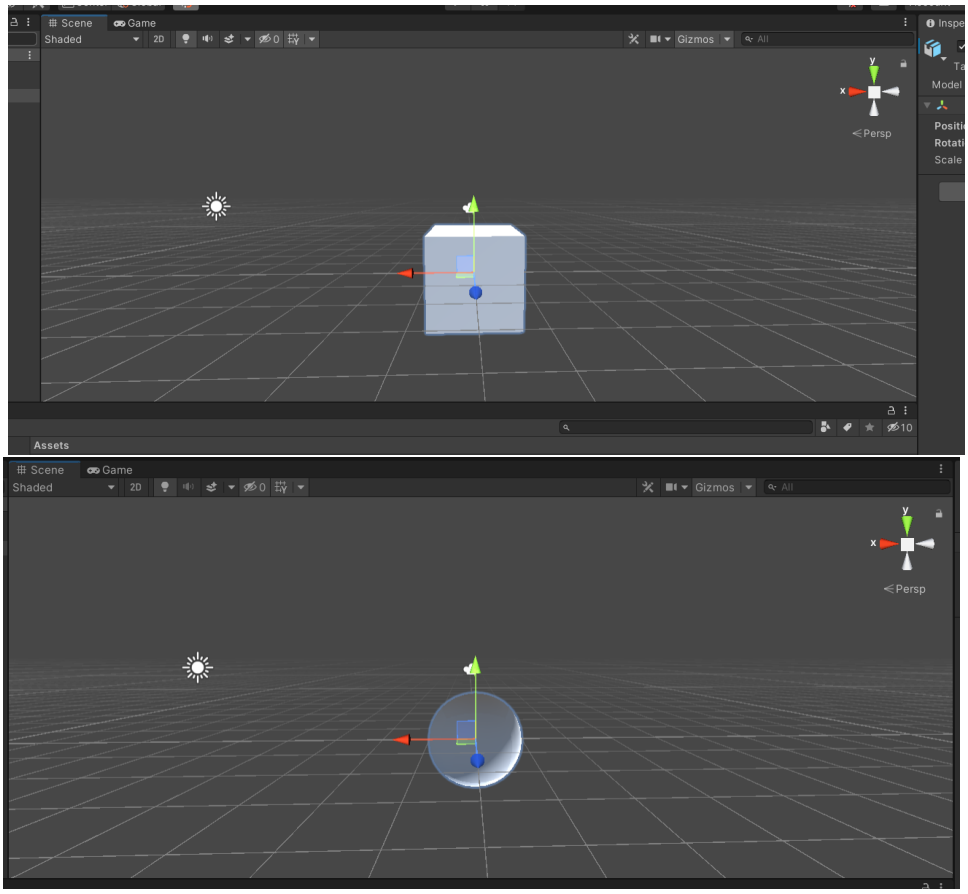


Figura 2.9: Cubo prima e dopo l'esecuzione del codice

Ciò può essere applicato a ogni tipologia di mesh:

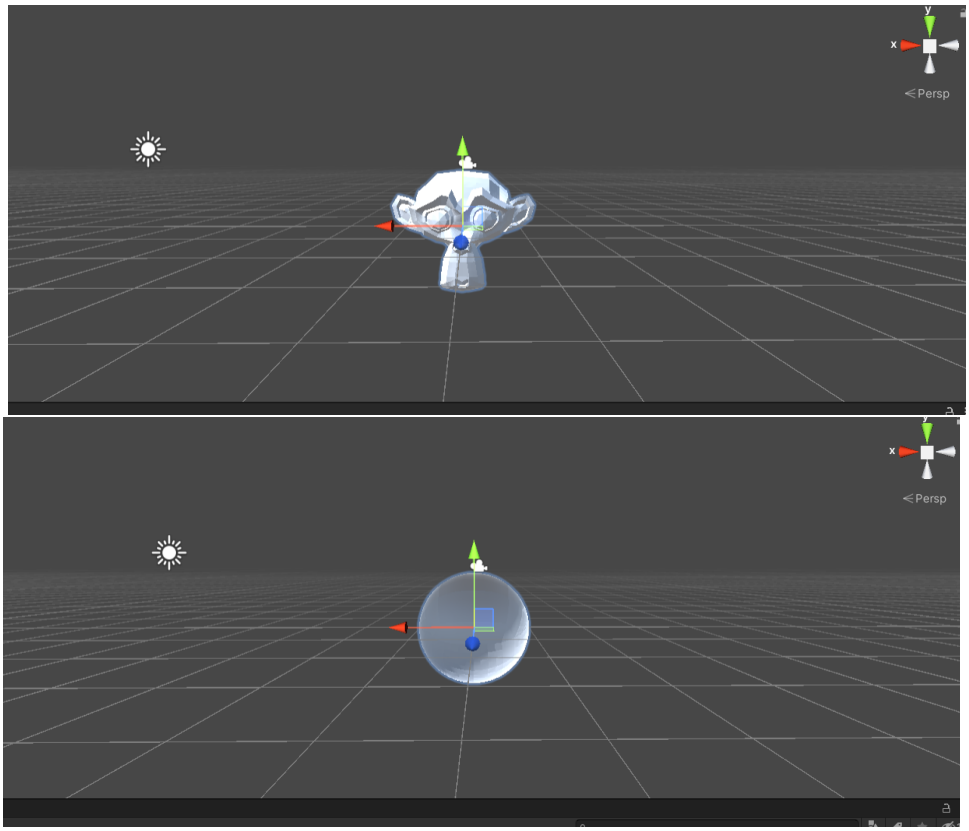


Figura 2.10: Scimmia prima e dopo l'esecuzione del code

2.1.3 Rendering

Con il termina *rendering* si indica la creazione di un'immagine 2D partendo da una scena 3D e da una camera virtuale. Una scena 3D è composta da geometrie, sorgenti di luce, proprietà materiali delle mesh e texture.
La camera virtuale decide cosa finirà nell'immagine finale, la renderizzazione quindi, a partire da una scena 3D, genera un'immagine 2D corrispondente al punto di vista della camera.
A oggi esistono due principali processi di rendering:

- **Pipeline Based Rendering (forward rendering)**: la pipeline di base che corrisponde a una sequenza di passi eseguiti dalla CPU e dalla GPU che vanno dall'applicazione fino alla scrittura sul frame buffer.
- **Ray-Tracing (backward rendering)**: processo opposto al precedente, a partire infatti dal 2D (la vista camera) invia raggi e assegna un colore ai pixel a seconda dell'intersezione con il raggio fuoriuscito.

2.2 Illuminazione e Shaders

Generalmente, assegnando un colore a tanti pixel viene visualizzata sullo schermo una scena. Questa scena può apparire spoglia e, considerando solo il colore degli oggetti, si perde il senso della profondità. Aggiungendo le luci si raggiunge una visualizzazione realistica, creando quel senso di profondità e rendendo il tutto visivamente reale.

Immaginando una scena in cui si ha un cubo con le facce dello stesso colore, senza l'applicazione delle luci questo cubo avrà la seguente forma:

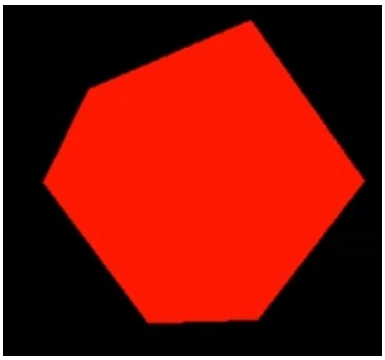


Figura 2.11: Immagine di un cubo senza l'applicazione di luci

Aggiungendo invece le luci, si avrà questo risultato:

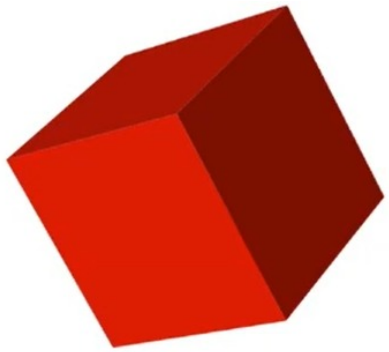


Figura 2.12: Dopo l'applicazione delle luci

Si può intuire quindi come le luci definiscono la tridimensionalità della scena.
Nella scena virtuale, le luci si comportano in maniera simile al mondo reale, dove il colore di un oggetto è il risultato della luce incidente che lo colpisce e il modo in cui la superficie di questo la riflette.

Le componenti fondamentali dell'illuminazione sono due:

- Sorgenti di luce: definite da emittance spectrum (colore), geometria (posizione) e direzione.
- Proprietà del materiale: definite invece da reflectance spectrum (colore), geometria (posizione, orientamento e microstruttura) e assorbimento.

Attualmente, ci sono due modelli di illuminazione:

- **Locale**: il colore di una superficie dipende solo dalla sorgente luminosa, ignorando gli altri oggetti vicini. Questo porta benefici dal punto di vista delle performance, in quanto la scena viene renderizzata molto più velocemente ma viene pagato un costo in termini di realismo.
- **Globale**: il colore di una superficie dipende sia dalla sorgente luminosa che dal riflesso degli altri oggetti in scena che colpisce la superficie.

Le sorgenti luminose a loro volta si dividono in quattro tipologie:

- **Luce ambientale**: una sorgente di luce con intensità fissa che colpisce equamente tutti gli oggetti. Proviene da tutte le direzioni e viene utilizzata per illuminare uniformemente l'intera scena simulando una luce indiretta.

- **Luce puntiforme**: una sorgente di luce che fuoriesce da un punto verso tutte le direzioni con intensità costante.
- **Luce direzionale**: illumina tutti gli oggetti equamente ma da una direzione specifica. Viene spesso posizionata molto lontano per simulare sorgenti di luce distanti come il sole.
- **Spotlight**: la luce è emessa da un singolo punto nello spazio e si diffonde a cono il cui apice è un punto con una direzione e la cui larghezza è determinata da un angolo θ. L'intensità è concentrata nel centro del cono e si attenua del tutto in θ e $-\theta$.

Un modello di iluminazione locale che considera la luce proveniente direttamente dalle sorgenti luminose e non dalla luce riflessa dagli oggetti adiacenti prende il nome di equazione di *Phong*. Secondo questo modello, l'intensità di luce in un punto I è definita dalle seguenti componenti:

$$I_\lambda = k_e + I_a K_a + I_d K_d + I s K_s$$

Ossia una componente emissiva (se il punto considerato è esso stesso una sorgente luminosa), una componente ambiente, una diffusiva e una speculare, con i vari k definiti come:

$$0 \leq k_e, k_a, k_d, k_s \leq 1$$

In quanto indicano la frazione dei contributi diversi e sono perciò dei numeri reali compresi tra zero e uno.

Contrariamente alla fase di **lighting**, ossia il processo di computazione dell'intensità luminosa di un punto 3D,la fase di **shading** è il processo che assegna un colore ai vari frammenti a partire dal colore dei vertici stessi. Dopo aver calcolato l'illuminazione sui vertici di una geometria, con lo shading viene attribuito un colore ai vari frammenti attraverso un'interpolazione del colore sui vertici.
Mentre l'illuminazione avviene nel geometry stage, la fase di shading avviene nel rasterization stage, in quanto effettuata sui frammenti.
Lo shading è una componente fondamentale del rendering e ogni oggetto visualizzato sullo schermo avviene mediante l'utilizzo di shaders.
Con la nascita dell prime GPU programmabili, uno shader può essere visto come una serie di istruzioni eseguite sulla GPU.

2.2.1 Shaders in unity

In Unity, gli shader vengono definiti tramite appositi file con estensione **.shader**. Questi file sono strutturati in modo annidato e organizzano le informazioni in varie strutture:

- **Properties**: costituisce un insieme di dati in input, come colori, valori, texture, e indirettamente la mesh relativa a tali valori.
- **SubShader**: permette di definire multipli *shader programs* per *use cases* differenti. È inoltre possibile avere più SubShader all'interno di uno stesso file shader. Un SubShader a sua volta contiene:
  - **Pass**: contiene il vero e proprio codice che viene eseguito nella GPU. Una SubShader può contenere molteplici Pass, i quali contengono:
    - **Vertex Shader**: uno shader che viene applicato per ogni vertice di una mesh e che ne può modificare la posizione.
    - **Fragment Shader**: uno shader che viene applicato per ogni frammento di una geometria, e ne decide quale sarà il colore (e quindi per ogni pixel nello schermo).

Il codice all'interno del Pass rappresenta il vero e proprio shader ed è scritto nel linguaggio HLSL. Le strutture all'interno delle quali si trova il Pass invece è una sintassi propria di Unity scritta con il linguaggio ShaderLab e utilizzata per descrivere uno **Shader Object**. Un semplice Shader di tipo Unlit (senza l'utilizzo di luci) appare in Unity come:

```
Shader "Unlit/7"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        LOD 100

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            // make fog work
```

```
18              #pragma multi_compile_fog
19
20              #include "UnityCG.cginc"
21
22              struct appdata
23              {
24                  float4 vertex : POSITION;
25                  float2 uv : TEXCOORD0;
26              };
27
28              struct v2f
29              {
30                  float2 uv : TEXCOORD0;
31                  UNITY_FOG_COORDS(1)
32                  float4 vertex : SV_POSITION;
33              };
34
35              sampler2D _MainTex;
36              float4 _MainTex_ST;
37
38              v2f vert (appdata v)
39              {
40                  v2f o;
41                  o.vertex = UnityObjectToClipPos(v.vertex);
42                  o.uv = TRANSFORM_TEX(v.uv, _MainTex);
43                  UNITY_TRANSFER_FOG(o,o.vertex);
44                  return o;
45              }
46
47              fixed4 frag (v2f i) : SV_Target
48              {
49                  // sample the texture
50                  fixed4 col = tex2D(_MainTex, i.uv);
51                  // apply fog
52                  UNITY_APPLY_FOG(i.fogCoord, col);
53                  return col;
54              }
55              ENDCG
56          }
57      }
58  }
```

In Unity, è presente una particolare tipologia di Shader che prende il nome di **Stencil Shader**. Questo utilizza uno **Stencil Buffer**, ossia una memoria che contiene un valore di 8-bit per pixel, con lo scopo di renderizzare o meno un oggetto nella scena.

Lo Stencil Shader assegna un valore all'oggetto al quale è assegnato, la GPU a sua volta comparerà il valore corrente nello Stencil Buffer con tale valore. Questa operazione prende il nome di *stencil test*. Se lo stencil test viene passato, la GPU esegue un *depth test*, se invece viene fallito allora la GPU salterà il resto del processamento per tale pixel e non renderizzerà l'oggetto.

In Unity, uno Stencil Shader appare come:

```
Shader "Custom/RenderOrNot"
{
    Properties
    {
        _MainTex ("Texture", 2D) = "white" {}
    }
    SubShader
    {

        ZWrite Off
        ColorMask 0

        Stencil{
            Ref 1
            Comp always
            Pass replace
        }

        Pass
        {
        }

    }
}
```

Dove, con **ZWrite** indica se renderizzare o meno l'oggetto, **ref** indica il *reference value* che verrà confrontata dalla GPU con il contenuto dello Stencil Buffer, **Comp** lo stencil test e con **Pass** viene indicata l'operazione effettuata quando viene passato lo stencil test.

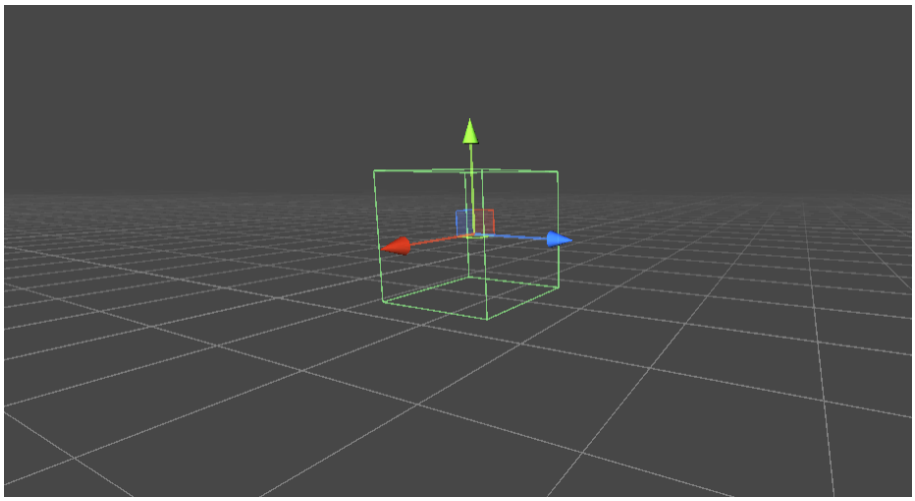


Figura 2.13: Cubo non renderizzato nella scena utilizzando ZWrite off

Capitolo 3

Psicologia e immersività

Il corpo umano non è stato progettato per la realtà virtuale, al contrario, la realtà virtuale è progettata per l'uomo; applicando stimolazioni artificiali inganna e bypassa le operazioni biologiche dell'individuo e quindi altera la sua percezione.
La percezione umana è infatti una sintesi delle informazioni ottenute a partire da stimoli esterni, catturati da recettori degli organi di senso, attraverso un processo cognitivo che produce infine un'informazione.
Gli organi di senso sono molteplici e, anche se la vista è il senso dominante e nella VR è l'elemento fondamentale per la fruizione dell'esperienza, la percezione non è limitata all'occhio umano, ma bensì a tutti i sensi.

La percezione avviene quando gli organi di senso convertono lo stimolo ricevuto in impulsi elettrici e li inviano al cervello. Secondo recenti studi il corpo umano contiene circa 86 miliardi di neuroni. Circa 20 miliardi sono nella corteccia cerebrale e gestiscono le percezioni e le altre funzioni di alto livello come l'attenzione, la memoria, il linguaggio e la coscienza.

Per propriocezione si fa riferimento invece alla capacità dell'individuo di percepire le posizioni nello spazio delle proprie parti del corpo e della quantità di sforzo muscolare necessaria per spostarle (per esempio, basti chiudere gli occhi, muovere un arto, automaticamente ci si renderà conto di sapere perfettamente dove questo arto si trova rispetto al mondo).

Le informazioni degli organi di senso e la propriocezione vengono processati dalla struttura neurale. Normalmente, il cervello elabora e interpreta gli input in maniera coerente e consistente e tutti i tentativi di interferire con queste operazioni rischia di causare un'incongruenza tra i dati ricevuti dai sensi, ossia un conflitto sensoriale.
In alcuni casi il corpo può adattarsi a questi conflitti, in altri casi il cervello si accorgere dello scenario inconsistente e, nel caso pessimo, uno sforzo eccessivo del cervello in uno scenario inconsistente può portare alla **VR Sickenss**. Nella maggior parte dei casi risponde con stanchezza e mal di testa ma nel peggiore dei casi vi è la nausea.

Uno degli esempi più importanti di conflitto sensoriale nella VR è la *vection*, ossia una circostanza in cui il senso della vista avverte un'accelerazione, ma il senso dell'equilibro informa di essere immobili. Ciò accade utilizzando un controller per la navigazione, di conseguenza la vista avvertirà il movimento nella scena virtuale, mentre in realtà l'utente è immobile.

Tuttavia, una caratteristica fondamentale del sistema sensoriale umano è l'adattamento. Quando una persona viene sottoposta costantemente a un certo stimolo, questa tende ad adattarsi (il rumore dell'aereo che decolla o della macchina che si muove tenderanno a dileguarsi poco tempo dopo la partenza). L'adattamento è fondamentale per la VR. Attraverso un'esposizione continua, gli utenti tendono ad essere più a loro agio rispetto a nuovi utente che sperimentano la scena virtuale per la prima volta, subendo così meno effetti collaterali.

La corretta coordinazione dei sensi umani con gli impulsi percepiti in uno scenario virtuale generano un'esperienza corretta.
Il risultato massimo viene raggiunto quando, oltre alla coerenza tra percezione umana e realtà virtuale percepita, vi è anche una alta immersività nella scena.

L'immersività psicologica è la misura dell'efficacia delle esperienze mediali più importante, dalla visione di un'immagine generata al computer fino all'essere immersi in un ambiente di realtà virtuale.
Questa dipende da tre fattori principali:

- Percezione nello spazio
- Percezione del movimento
- Percezione di un corpo

La natura multidimensionale di questa esperienza, unita alla presenza continua del proprio corpo biologico, rendono problematici gli approcci sia teorici che sperimentali. Da qui inoltre sorge una domanda: è possibile sperimentare le stesse sensazioni verso un corpo virtuale, all'interno di un sistema virtuale immersivo, attraverso il proprio corpo?

3.1 Embodiment

Una delle domande principali della scienza cognitiva è: come una persona sperimenta sé stessa all'interno di un corpo che interagisce continuamente con l'ambiente?
Le persone sperimentano sé stesse in un corpo, e più specificatamente in un corpo che

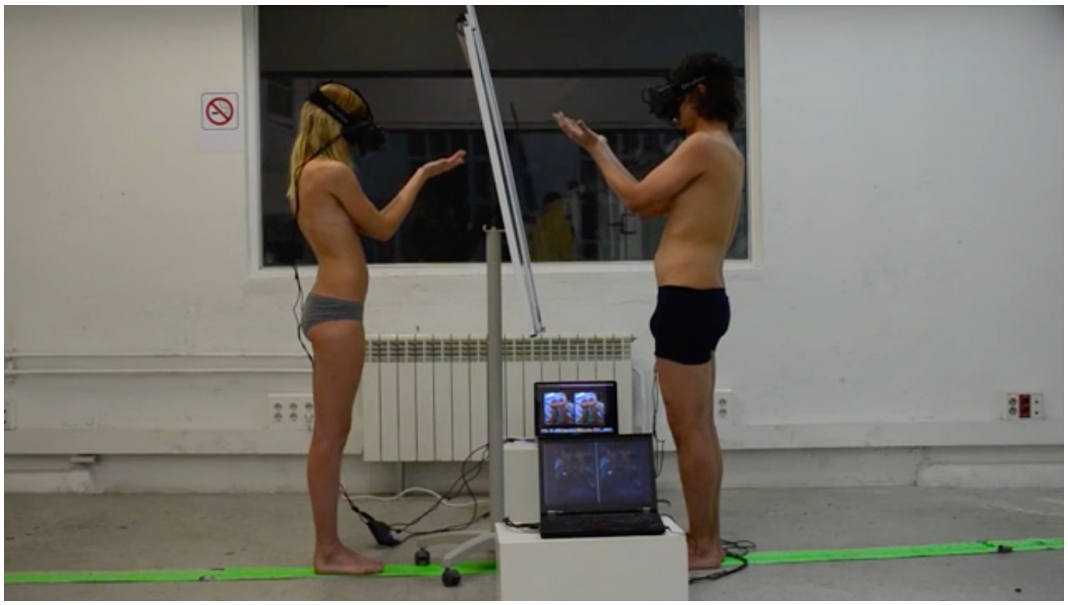


Figura 3.1: Body-Swapping, due soggetti sperimentano il corpo dell'altro attraverso l'utilizzo di visori

percepiscono come loro, che si muove secondo le loro intenzioni e obbedisce alle loro volontà.
Nella vita quotidiana queste sensazioni sono normalmente accoppiate insieme, percepite come emergenti da un solo corpo, ossia quello biologico, dando coerenza al proprio *io* e alla rappresentazione del proprio corpo.

La manipolazione sperimentale del senso di embodiment è problematica dal momento che il proprio corpo è sempre presente e apparentemente non può essere dissociato dal proprio io.
Tuttavia, gli studi sulla percezione del corpo rilevano un approccio alternativo a questa esperienza, manipolando l'identità di una parte del corpo.

Secondo *De Vignemont*, una persona incarna un oggetto E se e solo se alcune proprietà di E sono processate allo stesso modo delle proprietà del proprio corpo. Questa definizione è in linea con *Blanke* e *Metzinger*, i quali hanno dichiarato che l'embodiment include esperienze soggettive dell'utilizzo e del possesso di un corpo.

Perciò, per definire il senso dell'embodiment viene globalmente accettata questa definizione:

> *Il senso di embodiment verso un corpo B è il senso che emerge quando le proprietà di B sono processate come se fossero delle proprietà del proprio corpo biologico*

Le proprietà biologiche di un corpo fisico possono essere descritte da tre sensi:

- **Senso di self-location**: determinazione del volume di spazio in cui una persona sente di essere. Normalmente, self-location e body-space coincidono tra loro, in quanto ci si sente presenti all'interno di un corpo fisico, tuttavia, questa sensazione può essere rotta nel momento in cui qualcuno sperimenta delle esperienze out-of-body (OBE). È da notare il fatto che per self-location si intende solo la percezione della posizione di sé rispetto al proprio corpo, e non in base al mondo (presenza).
- **Senso di agency**: percezione di avere controllo globale, incluse le esperienze di azione soggettive, il controllo e l'intenzione. Il sense of agency è presente nei movimenti attivi, e, per esempio, non è presente nel caso di pazienti che soffrono della sindrome della mano anarchica.
- **Senso di body ownership**: auto attribuzione di un corpo, ciò porta a pensare che il corpo è la sorgente di tutte le sensazioni sperimentate. Un paziente che soffre di somatoparafrenia tende a non sviluppare questo senso nei confronti dell'arto indesiderato.

Dopo aver introdotto queste tre caratteristiche, è possibile riformulare la definizione precedentemente fornita:

> *Un soggetto A sperimenta un sense of embodiment nei confronti di un corpo B, se, almeno in minima intensità: si sente self-located in B, si sente agent di B o se percepisce B come il suo corpo*

Quando l'utente sperimenta almeno uno di questi tre sensi in minima intensità allora il soggetto sperimenta un sense of embodiment.
Se l'utente sperimenta tutti e tre i sensi a massima intensità allora questo sperimenta un'esperienza completa e massima di sense of embodiment.

3.2 Avatar

Finora la Virtual Reality è stata espressa solamente in funzione della sua capacità di creare un mondo parallelo con il quale un utente interagisce.
Tuttavia, la comunicazione e la socialità sono argomenti molto ampi che estendono il concetto di VR.

Interazioni sociali in VR, o social VR, consentono alle persone di connettersi nella maniera più pura, e questo è il potenziale più grande della realtà virtuale, implementare un nuovo tipo di comunicazione, molto più simile a quella reale rispetto ai moderni sistemi di comunicazione.
Più persone possono condividere lo stesso ambiente virtuale, interagendo tra loro, comunicando, in un modo simile al mondo reale.
Ma come dovrebbe quindi apparire agli altri un utente in un sistema VR? Attraverso una rappresentazione di sé stesso, ossia attraverso un avatar.

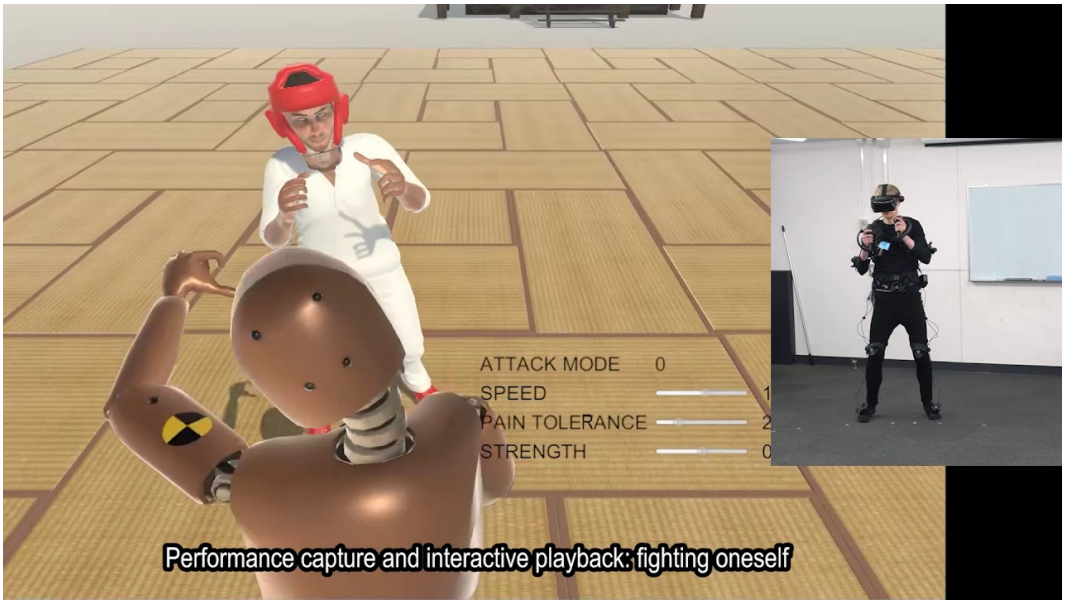
Un avatar è una rappresentazione grafica (tridimensionale o bidimensionale) di un utente che può corrispondere visivamente, uditivamente e comportamentalemnte all'utente. Comunemente questi sono utilizzati nei social-network o nel mondo video ludico. Tuttavia un avatar ha un impatto molto più forte di quanto si può vedere in superficie.
Uno degli aspetti più appetibili di un avatar è l'anonimità. Come visto in [27], esiste un fenomeno chiamato **Effetto Proteus**, ossia un fenomeno secondo il quale il comportamento di una persona cambia in base alle caratteristiche dell'avatar, secondo i preconcetti sociali diffusi. Per esempio, persone che impersonano avatar più alti o di bell'aspetto dimostrano un grado di confidenza più alto rispetto a persone che impersonano avatar bassi o di aspetto sgradevole.
Un avatar permette anche di sviluppare empatia, sperimentando infatti il mondo dalla prospettiva di un avatar che appare differente in termini di razza, sesso, altezza, peso ed età, è possibile vivere realtà diverse.
Gli avatar arricchiscono l'esperienza virtuale aumentando il livello di immersività dell'utente.
Steed e Schroeder identificano il realismo di un avatar come uno dei fatti principali che influiscono sull'interazione sociale e la co-presenza nella realtà virtuale [22].

Con il crescere dell'attenzione del pubblico nei confronti del Metaverso, è lecito immaginare una realtà virtuale dove gli utenti utilizzano un HDT come avatar, ossia uno **Human Digital Twin** che ne rappresenta una copia virtuale fedele all'originale.
Il grado di similarità di avatar può essere espresso attraverso tre metriche di giudizio:

- **Visual appearence**: quanto l'avatar è visivamente simile all'utente.
- **Auditory appearence**: quanto il suono proveniente dall'avatar coincide con la voce e al linguaggio della persona.
- **Behavioral appearence**: quanto il movimento dell'avatar corrisponde al linguaggio del corpo, i gesti, le espressioni facciali e altri movimenti della persona.

Per il primo caso, esistono numerose tecniche per convertire un volto (o un corpo) fisico in formato tridimensionale, come per esempio il 3D scanning, che tuttavia possiede costi elevati e una manodopera considerevole per il postprocessing del risultato.
Tuttavia, un realismo troppo spinto o un mismatch troppo ampio tra realismo fotografico e comportamentale possono portare a effetti indesiderati quale l'**Uncanny Valley**, ossia la sensazione di familiarità e piacevolezza sperimentata da un campione di persone e generata da robot, automi e avatar virtuali che aumenta al crescere della loro somiglianza con la figura umana, fino a un certo punto in cui l'estremo realismo rappresentativo produce repulsione e inquietudine.

Per quanto riguarda la parte uditiva, è molto semplice registrare e riprodurre messaggi vocali in VR, facendo in modo che la parte uditiva coincida con la rappresentazione visiva, per esempio l'audio deve uscire dalla bocca dell'avatar e si deve considerare la distanza per la produzione di suoni con una certa profondità. Infine, per quanto riguarda la rappresentazione comportamentale, esistono molti *capture systems* in grado di mappare fedelmente il movimento del corpo dell'utente all'interno della scena, e la tecnologia head-tracking permette anche di mappare il movimento della testa umana all'interno della scena. Tuttavia, acquisire il volto umano è molto difficile in quanto coperto dell'headset, e il tracking del corpo può essere un'operazione molto costosa e non troppo accurata.



Body-Tracking completo

Nella vita di ogni giorno, le persone comunicano attraverso un'interazione *one-on-one*, ossia un'interazione tra due persone differenti. Un aspetto importante di questo tipo di

comunicazione è la tipologia di relazione tra le due persone, questa può essere:

- Simmetrica: una relazione in cui le persone hanno lo stesso status.
- Complementare: una relazione in cui una persona occupa una posizione superiore, come nel caso del capo e dell'impiegato.

Ma la comunicazione può avvenire anche tra più persone e anche qui in modo simmetrico, un pranzo di famiglia per esempio, o complementare, come una lezione frontale tra insegnante e studenti. Tutte queste casistiche sono facilmente riproducibili in un ambiente virtuale attraverso la VR. Numerosi sono gli esempi di community online emerse che connettono milioni di utenti, come per esempio i videogiochi multigiocatore di massa (MMORPG), o la piattaforma Second Life.

Figura 3.2: Second-Life, mondo virtuale molto conosciuto

La VR inoltre permette di trasformare le interazioni sociali, permettendo situazioni impossibili nel mondo reale. Per esempio, nel caso di una lezione online svolta in VR, è possibile modificare la scena in modo tale che ogni studente veda il volto del professore che lo osserva. Ciò è impossibile nel mondo fisico, un professore non può contemporaneamente fissare tutti gli studenti contemporaneamente, ma in un mondo virtuale ciò è possibile.

3.3 Who will trust my digital twin?

Come visto nella sezione precedente, le interazioni sociali posso essere proiettate all'interno di uno scenario virtuale in varie forme, dalla *one-on-one* alle interazioni sociali di più grande scala.
L'efficacia di queste interazioni dipende da numerosi fattori: il sense of embodiment sperimentato, il realismo degli avatar e il grado di empatia provato nei confronti di un avatar.
Nel corso del lavoro di tesi è stato svolto uno studio preliminare che esamina l'efficacia delle interazioni **HDT-human** nel contesto di un negozio di moda. Lo studio è stato condotto presso il Laboratorio di Realtà Virtuale dell'Università di Bologna, attraverso tale studio è stato poi pubblicato un articolo.

L'esperimento è stato svolto utilizzando un Digital Twin il quale, interfacciandosi con il gestore di un negozio di abbigliamento, esprime delle richieste riguardo prodotti da acquistare.
I soggetti partecipanti all'esperimento impersonavano la figura del gestore del negozio, e al termine della conversazione, hanno risposto a un questionario esprimendo il proprio livello di confidenza con tale interlocutore virtuale.

Figura 3.3: Scansione tridimensionale del modello

Con l'avanzata del Metaverso, le interazioni con avatar virtuali diventeranno sempre più

comuni e saranno presenti nei campi più disparati, dalle simulazioni lavorative fino a meeting online.
Questo lavoro mira perciò a esplorare l'accettazione che un HDT riscuote nel campo dello scenario di una piattaforma di *x-commerce* al fine di analizzare quali sensazioni prova una persona fisica durante l'interazione con uno Human Digital Twin.

3.3.1 Svolgimento

Per realizzare tale esperimento è stato utilizzato uno scanner **Artec Eva 3D** per l'acquisizione del modello tridimensionale dell'attore, il quale si è prestato in questo lavoro. Inoltre, è stato utilizzato il software Open-Source Blender per il postprocessing del modello acquisito e per realizzare le animazioni necessarie utilizzate nella scena.
Il funzionamento dello scanner si basa sull'irradiazione di una luce bianca e sulla triangolazione della camera e fornisce una precisione di 0.1mm, il che permette di acquisire modelli 3D molto complessi e di ottima qualità. Per condurre l'esperimento, sono stati realizzati tre video differenti:

- Il primo video mostra il modello reale.
- Nel secondo e terzo video viene mostrato il Digital Twin del modello, rispettivamente con una voce naturale e con una voce robotica.

I tre video realizzati rappresentano la stessa scena: in ognuno di questi il soggetto parlante richiede al commesso uno specifico prodotto. Per analizzare come la comunicazione con l'HDT viene percepita, è stato reclutato un gruppo di 32 partecipanti, 13 maschi e 19 femmine, dai 21 ai 58 anni.
I partecipanti hanno visionato i tre video, ognuno dei quali della durata di 11 secondi.
Ogni partecipante ha poi risposto a un questionario fornendo un'impressione preliminare degli HDT e, dopo essersi immedesimato in un impiegato, ha risposto a un ulteriore numero di domande. Dopo aver completato i questionari di consenso e demografici, i partecipanti hanno risposto a domande preliminari sulla loro occupazione (studente, lavoratore, commesso di un negozio di moda) e i loro rapporti con avatar digitali (specificando se hanno avuto nella loro vita almeno un contatto con un avatar, la frequenza e e il grado di confort nei loro confronti). A seguire, l'esperimento ha avuto luogo.
Ogni partecipante è stato coinvolto in una sessione *within-subject*, chiedendo loro di identificarsi come impiegati di moda e rispondere a specifiche richieste dei clienti.

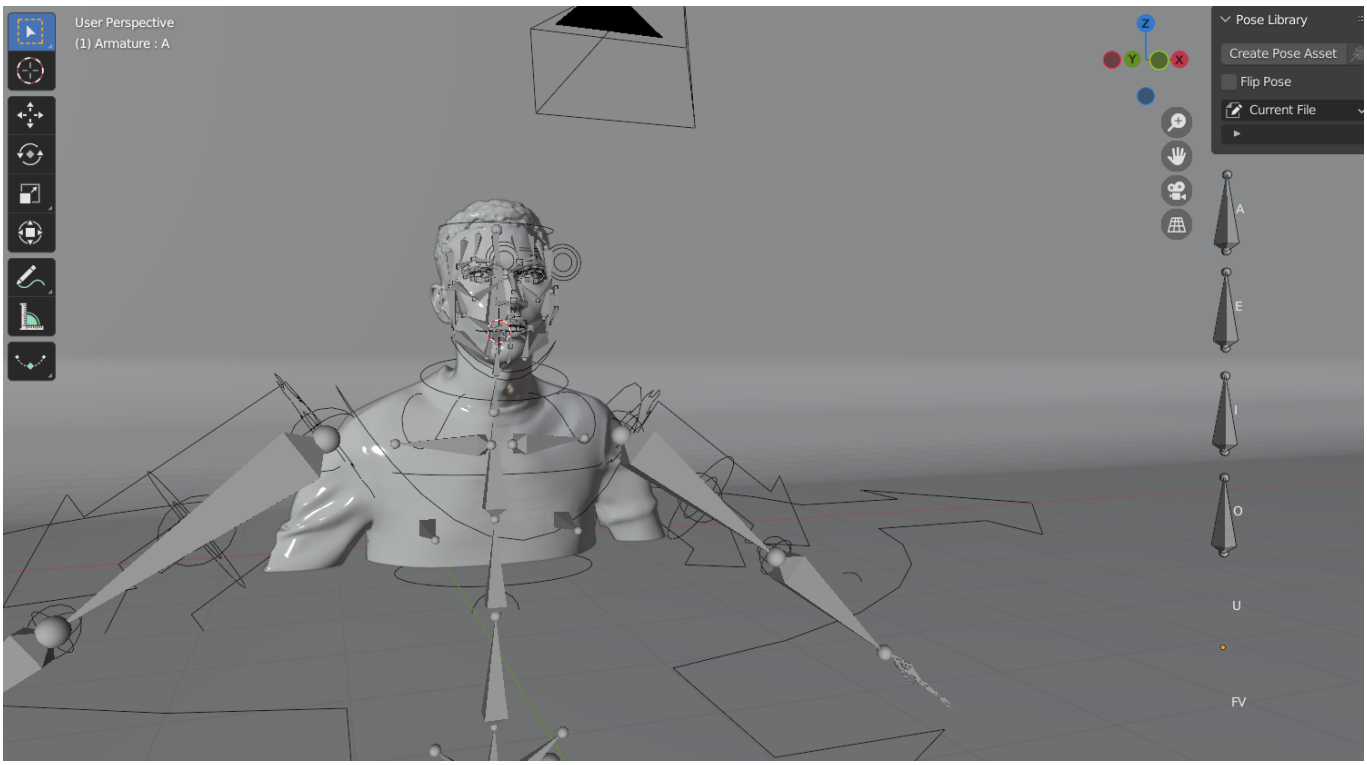


Figura 3.4: Post-processing in Blender

La prima richiesta è stata:

> *Buongiorno, avrei bisogno di una camicia da indossare con un pantalone di lana di colore grigio casual. Di solito indosso la taglia M, ma se lo vuole le posso fornire il mio modello 3D.*

Tale richiesta viene effettuata in tre configurazioni differenti:

- Persona fisica che utilizza la voce naturale (NN)
- HDT che utilizza la voce naturale (AN)
- HDT che utilizza una voce robotica (AA)

Dopo aver visualizzato ognuno dei tre video, ai soggetti è stato richiesto di rispondere a domane aperte in non più di un minuto, per facilitare il loro sato di elaborazione e per concentrarsi sulla richiesta invece che sull'aspetto visivo del cliente.
Poi, ogni partecipante ha risposto a domande differenti riguardo la propria esperienza, adottando un scala di Likert a 7 punti:

- (Q1) Comprendere la richiesta del cliente (complesso o facile)
- (Q2) Quanto naturale/spontanea l'interazione con il cliente era considerata (tanto o poco)
- (Q3) Se durante la trattativa ci si è sentiti a proprio agio o no

Finalmente, ogni partecipante ha risposto a una serie di domande riguardo il proprio interesse a lavorare in un negozio di moda in un contesto di *extended reality*, anche qui misurato attraverso una scala di Likert a 7 punti:

- (I1) Quanto favorevolmente si lavorerebbe in uno store virtuale
- (I2) Quanto favorevolmente si interagirebbe con un HDT

3.3.2 Risultati

Analizzando la media e la deviazione standard, è possibile vedere come gli utenti preferiscono interfacciarsi con la persona fisica e non gradiscono invece l'avatar con una voce robotica.
La configurazione dell'avatar con la voce normale ottiene valori di gradimento più bassi rispetto alla persona reale, tuttavia la differenza non risulta significante, il che dimostra che tale configurazione ha ottenuto comunque dei buoni risultati.

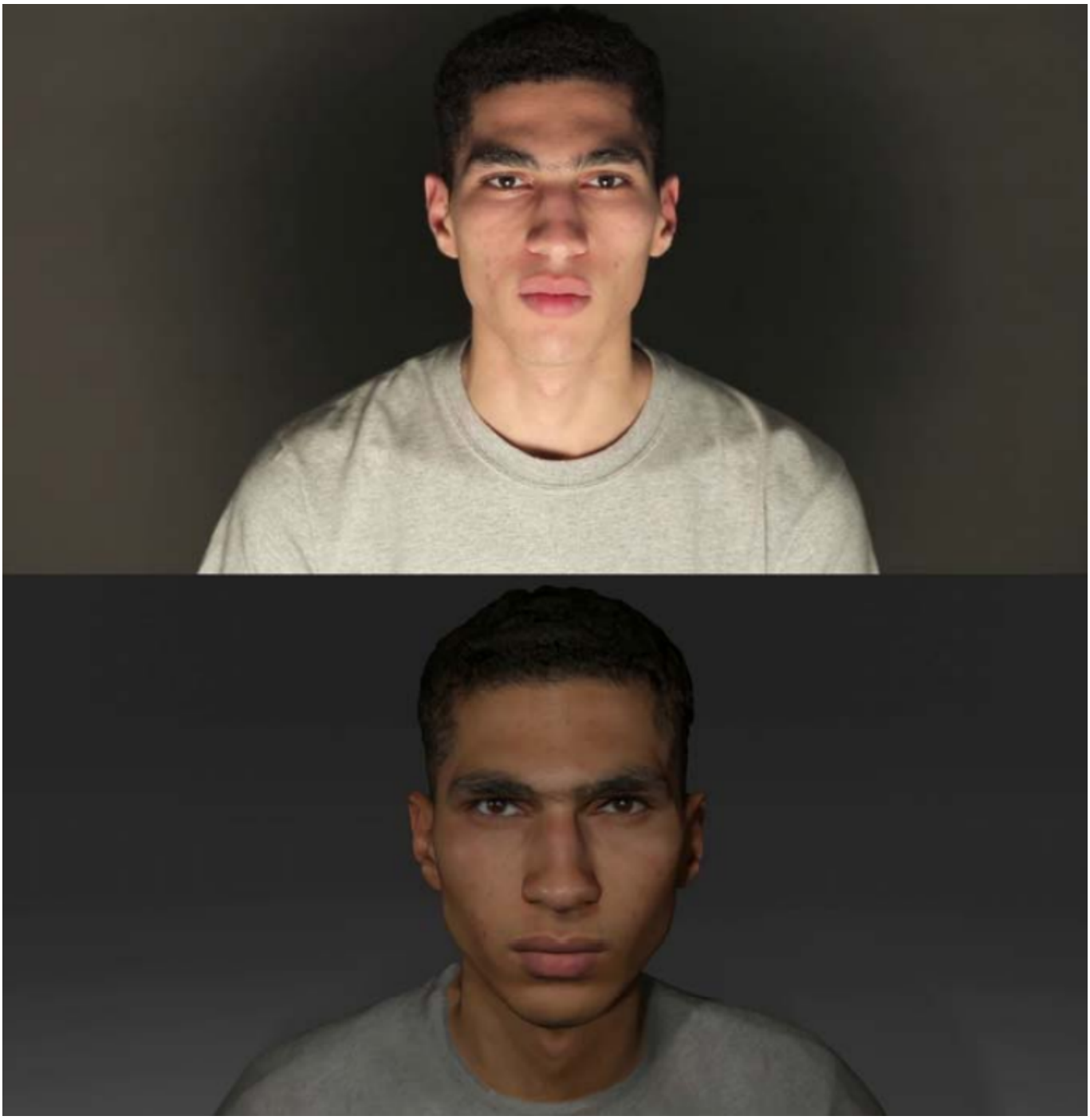


Figura 3.5: Modello reale e virtuale

In conclusione, è stato presentato un esperimento preliminare in cui si è valutato il possibile favore degli HDT come clienti in un negozio fisico.
I risultati suggeriscono che l'adozione di tale tecnologia è possibile, tuttavia, molti aspetti richiedono ulteriori indagini per supportarne una qualsiasi interazione sociale tra umani e HDT.
Nei lavori futuri, si mirerà a migliorare il design sperimentale, esaminando diversi aspetti e contestualizzando maggiormente HDT e umani in contesti strutturati all'x-commerce.

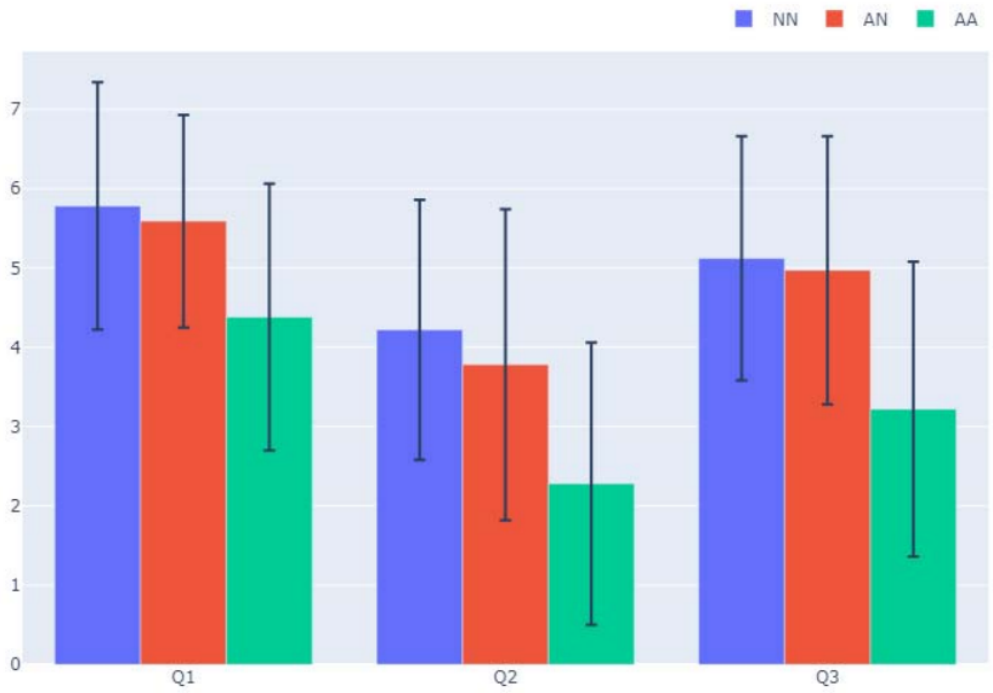


Figura 3.6: Risultati dell'esperimento

Capitolo 4

Movimento

La locomozione, l'atto di muoversi da un luogo all'altro, è considerata una delle attività più fondamentali e universali eseguita durante l'interazione con la Virtual Reality.
Per la maggior parte delle persone, la locomozione è l'attività più triviale e frequente nella propria vita, tuttavia, permettere agli utenti di muoversi liberamente attraverso un ambiente virtuale presenta una sfida considerevole.
Idealmente, il movimento dell'utente all'interno del VE (Virtual Environment) dovrebbe essere vincolato solo dalla topografia e dall'architettura virtuale, e non dalla grandezza dello spazio fisico.
Purtroppo, questa casistica è possibile solo all'interno di ambienti virtuali immersivi che si adattano all'interno dei confini dello spazio fisico dell'utente.
In molte occasioni gli utenti si trovano a muovere in un ambiente virtuale notevolmente più grande dello spazio fisico disponibile. È infatti ragionevole pensare che gli utenti fruiscano di tale tecnologia all'interno delle proprie abitazioni e magari nelle proprie stanze.
Sono state proposte numerose tecniche che hanno agevolato la situazione permettendo di muoversi in uno spazio virtuale più grande, la maggior parte di queste però si basano su approcci irrealistici, i metodi di locomozione oggi più impiegati infatti sono:

- **Movimento attraverso analogico**: movimento classico che tuttavia ha il più alto tasso di motion sickness.
- **Teletrasporto**: l'utente indica un punto nel terreno attraverso il controller nel quale terrà teletrasportato.
- **Portali**: l'utente crea due portali attraverso i quali spostarsi nella scena.
- **Arm-swing**: l'utente si muove oscillando le proprie braccia simulando la camminata o la corsa.

Questi approcci, seppur permettono di muoversi in uno spazio virtuale molto grande, rovinano l'esperienza rompendo il senso di immersività tanto ricercato.

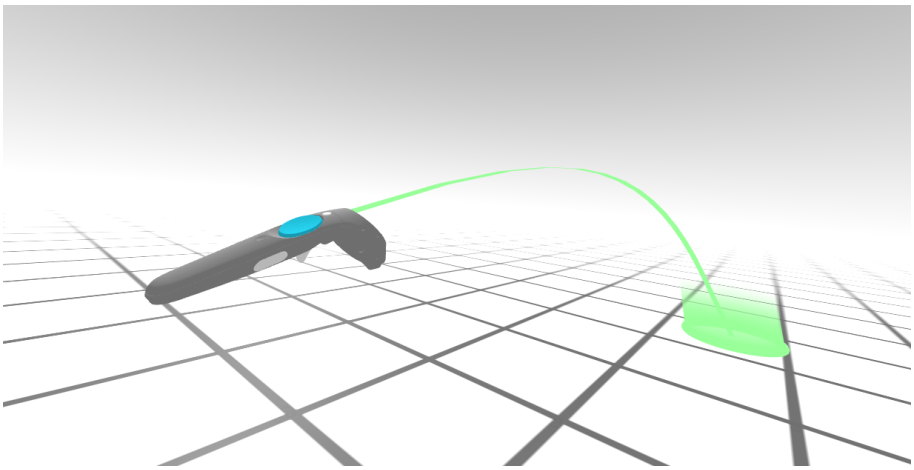


Figura 4.1: Il teletrasporto è la soluzione più adottata nelle applicazioni VR

Tuttavia, nel corso degli anni sono state sviluppate tecniche che mirano all'utilizzo della locomozione naturale in ambienti ristretti, anche chiamate **tecniche di ridirezione** (RDW, redirected walking) [17].
Nonostante le forme di RDW proposte sono molto numerose, tutte queste si basano su due forme di manipolazione:

- Tecniche che manipolano le proprietà fisiche del movimento dell'utente all'interno dello spazio virtuale.
- Tecniche che manipolano l'ambiente virtuale stesso.

In entrambi i casi, l'obiettivo è quello di massimizzare lo spazio fisico disponibile.
Il vantaggio più considerevole delle tecniche di ridirezione è il fatto che l'utente si muove fisicamente, e per questo sperimenta stimolazioni propriocettive, cinestetiche e vestibolari nel modo corretto.

Una tecnica RDW ideale deve soddisfare quattro criteri:

- **Impercettibile**: l'utente non deve percepire che la ridirezione sta avvenendo.
- **Sicura**: l'utente non deve essere messo in pericolo durante la camminata.
- **Generalizzabile**: deve poter essere applicabile in qualsiasi ambiente virtuale con un qualsiasi numero di utenti.
- **Priva di effetti collaterali**: deve evitare effetti indesiderati come motion sickness o interferenza con i task principali dell'esperienza virtuale.

Nonostante a oggi nessun tipo di RDW può vantare tutti questi quattro criteri contemporaneamente, negli anni sono stati fatti grandi passi avanti.

4.1 Stato dell'arte

Le prime tecniche di ridirezione sono state ideate più di 15 anni fà.
Come detto nella sezione precedente, attualmente tutte le RDW si basano su due forme di manipolazione:

- Manipolazione del mapping tra lo spostamento e la rotazione reali dell'utente
- Manipolazione delle proprietà architetturali dell'ambiente virtuale

A loro volta, le tecniche di ridirezione si suddividono in due categorie principali:

- **Subtle manipulations**: tecniche impercettibili all'utente.
- **Overt manipulations**: tecniche percepite dall'utente.

4.1.1 Manipolazione del movimento dell'utente nel mondo virtuale

La prima implementazione delle RDW si basa sulla manipolazione del mapping tra la reale rotazione e quella virtuale.
Nello specifico, questa tecnica si basa sull'applicazioni di rotazioni impercettibili che spingono l'utente inconsciamente a cambiare la propria traiettoria.
Tale tecnica prende il nome di *rotation gain* e, successivamente, sono state proposte tecniche simili che sfruttano lo stesso principio.
Attualmente le tecniche che sfruttano il principio del *gain* sono:

- **Rotation gains**: come spiegato, il *rotation gain* va a modificare il mapping tra la rotazione reale e quella virtuale, ossia la rotazione dell'utente nel mondo reale può essere tradotta come una rotazione più o meno grande nel mondo virtuale. Jerald e altri indicano che gli utenti sono meno propensi a notare cambiamenti se il *gain* viene applicato nella stessa direzione in cui la testa ruota.
  Specificatamente, la rotazione può essere aumentata dell'11.2% o diminuita del 5.2% senza che l'utente si accorga di nulla. Per quanto riguarda la rotazione full-body, Bruder e altri hanno scoperto che la rotazione può essere aumentata del 30.9% o diminuita del 16.2%.

- **Translation gains**: si basa sullo scalare il movimento dell'utente. Steinicke è arrivato alla conclusione che il movimento dell'utente può essere amplificato del 26% o attenuato del 14%.

- **Curvature gains**: la traiettoria dell'utente viene curvata senza che questo ne sia consapevole. Nel caso ottimo questi approcci portano l'utente a camminare in uno spazio circolare nonostante nel mondo virtuale possieda una traiettoria rettilinea. È stato suggerito che gli utenti possono essere reindirizzato lungo una circonferenza di raggio 22 metri, 11.6 metri o 6.4 metri.

- **Bending gains**: in maniera simile al curvature gain, qui la traiettoria dell'utente, già curvilinea, viene curvata ancora di più. Langhebehm ha scoperto che è possibile flettere una curva fino a 4.4 volte il suo raggio nel mondo reale.

È da notare il fatto che sia il curvature gain che il bending gain richiedono una conoscenza a propri del percorso dell'utente, o una buona predizione del path seguito.

4.1.2 Tecniche di ridirezione overt

Le tecniche precedentemente descritte fanno parte della categoria *subtle*, in quanto l'utente non è in grado di percepire tali modifiche mentre si trova nella scena virtuale.
Tuttavia, nonostante un RDW dovrebbe essere idealmente impercettibile, sono state proposte anche overt techinques come per esempio la *seven league boots* che permette all'utente di abilitare o disabilitare traslazioni percettibili premendo un pulsando del controller [12] o tecniche come quelle descritte da Williams e altri [26] che mirano a intervenire innescando una *overt tecnicque* nel momento in cui l'utente raggiunge i limiti dello spazio fisico.
Le tecniche utilizzate in questo caso sono tre:

- **Freeze-backup** dove l'esperienza virtuale viene congelata e l'utente viene guidato al centro della stanza per poi riprendere dal punto in cui è stato interrotto.

- **Freeze-turn** dove l'immagine nel display viene congelata e il sistema richiede all'utente di voltarsi verso il centro della stanza per poi riprendere.

- **2:1 turn** dove anche qui l'esperienza viene congelata e viene chiesto all'utente di fare un giro su sé stesso. La velocità di rotazione viene raddoppiata in modo tale che l'utente ruota nella scena virtuale di 360°, ruotando solo di 180° nel mondo reale.

Inoltre, come studiato da [7] l'utilizzo di distrattori può essere una buona tecnica per modificare la posizione dell'utente nello spazio virtuale.
I distrattori sono elementi o personaggi virtuali il cui obiettivo è quello di catturare l'attenzione dell'utente mentre il sistema li dirotta inconsciamente lontani dai limiti fisici.
Le tre tipologie di distrattori studiati sono:

- **Looking**: distrattore che richiedeva di essere semplicemente osservato.
- **Touching**: distrattore che richiedeva di essere toccato.
- **Interacting**: distrattore che richiedeva un'interazione, come per esempio il risolvere un semplice task.

Questi sono stati un'ottima tecnica per modificare la traiettoria dell'utente, riportando un risultato migliore nel caso dell'interazione con il distrattore.
Infine, uno studio condotto da Simeone e altri ha dimostrato come l'architettura virtuale influenza il comportamento dell'utente. Per esempio, un utente che si trova di fronte un ostacolo nell'ambiente virtuale, tende a cambiare la propria traiettoria e a evitare tale ostacolo.

4.1.3 Manipolazione dell'ambiente virtuale

Come specificato nella sezione precedente, la seconda categoria di RDW si basa sull'alterazione dell'architettura della scena virtuale e, anche in questo caso, l'alterazione può essere *subtle* o *overt*. Adalberto e altri hanno proposto una tecnica *overt* chiamata **Space Bender** che si basa sulla manipolazione dell'ambiente virtuale andando a flettere la geometria stessa ogni qual volta che l'utente arriva in prossimità dei limiti fisici dello spazio reale. Quando l'utente raggiunge il bordo dello spazio fisico, la scena viene flessa di 90° gradi, in modo da permettere all'utente di continuare a muoversi.

Suma e altri hanno proposto un approccio di ridirezione radicalmente differente: l'architettura dell'ambiente virtuale viene modificata sfruttando l'inabilità dell'utente di osservare i cambiamenti nello spazio. Questi hanno cambiato la posizione di porte e corridoi alle spalle degli utenti, andando così a modificare il loro percorso.
Nello studio condotto, solo uno di 77 partecipanti in due *user studies* è riuscito a notare la manipolazione.
Questa tipologia di manipolazione permette agli utenti di navigare all'interno di un grande ufficio virtuale di circa 219 metri quadrati, senza lasciare uno spazio di $4.3 * 4.3$ metri.

Figura 4.2: Risultato dello Space Bender, di Simeone e altri

4.2 Impossible Spaces

La tecnica appena descritta prende il nome di **Impossible Spaces**, una tecnica che si basa esclusivamente sulla manipolazione architettonica dell'ambiente virtuale.
Gli Impossible Spaces consentono di comprimere ambienti virtuali molto grandi in spazi fisici più piccoli, per mezzo di architetture *self-overlapping*, ossia geometrie sovrapposte. Due *user studies* hanno rivelato come piccole stanze possono sovrapporsi fino al 56% senza che l'utente ne sia consapevole, e stanze più grandi mappate in uno spazio fisico di 9.14 x 9.14 metri possono sovrapporsi fino al 31%.
Se l'obiettivo non è quello di replicare un ambiente reale, allora la tecnica chiamata **Flexible Spaces** può essere utilizzata per fornire una camminata illimitata all'interno del VE generato dinamicamente.

Il lavoro di Vasylevska e Kaufmann ha dimostrato che è possibile aumentare la soglia di detection e la quantità di sovrapposizione tra stanze virtuali collegandole attraverso un corridioio molto lungo, inoltre corridoio leggermente curvilinei possono essere più vantaggiosi per la compressione spaziale.
Nonostante questo metodo sia efficace, lo svantaggio più grande è l'impossibile di applicare tale tecnica in ambienti esterni.

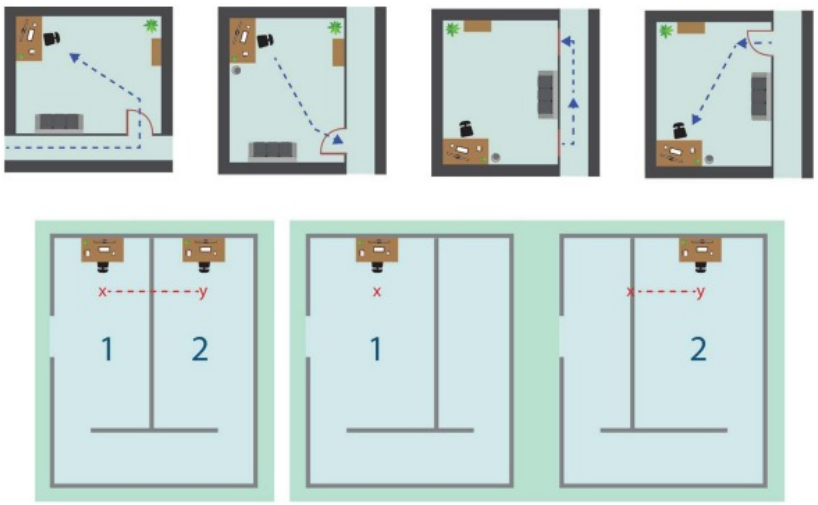


Figura 4.3: Schema riassuntivo del meccanismo degli impossible space

4.3 Effetti collaterali indesiderati

Le tecniche RDW creano intenzionalmente delle discrepanze tra i sensi spaziali, distorcono la mappatura dei movimenti, o riorganizzano l'architettura dell'ambiente virtuale, tutto ciò in modo *subtle*, quindi di nascosto all'utente, o in modo *overt*, quindi palesemente sotto gli occhi dell'utente.
Queste potenzialmente possono:

- Indurre motion sickness: in base alla quantità di gain applicata durante la sessione, l'utilizzo di tali RDW può provocare un aumento della probabilità di occorrenza di motion sickness, proporzionale alla quantità di gain applicata.

- Interferire con l'apprendimento spaziale e con la memoria: a oggi sono pochi i dati empirici rigardo le conseguenze delle RDW sulla cognizione spaziale e sulla memoria. Hodgson e altri hanno esaminato l'effetto di queste senza trovare alcun impatto negativo sulla memoria. Questo risultato è in conflitto con lo studio di Williams e altri i quali hanno affermato che i conflitti senso-motori che si creano nella scena hanno un impatto negativo nell'apprendimento spaziale.

- Portare un carico cognitivo maggiore rispetto al semplice camminare nel mondo reale: in questo caso è stato analizzato come il carico cognitivo aumenta nel momento in cui però i cambiamenti sono percepibili dall'utente, mentre prima di superare tale soglia non si giunge a un carico eccessivamente pesante.

4.4 Impossible spaces: lavoro svolto

Durante la scrittura di questa tesi, è stato svolto un periodo all'estero della durata di 3 mesi presso l'università di Leuven (Belgio), KU Leuven.
Nel corso di questo periodo è stato condotto uno studio riguardo i problemi della locomozione naturale, legati all'incongruenza tra spazio fisico e spazio virtuale, e gli Impossible Spaces.
È stato inoltre portato avanti un progetto sulla loro possibile applicazione all'interno di uno ambiente virtuale circolare.
Il lavoro svolto consiste nell'applicazione della tecnica degli impossible spaces in una stanza circolare al fine di massimizzare lo spazio disponibile percorribile da un utente.

4.4.1 Svolgimento

Al fine di ottenere tale risultato, è stata realizzata una stanza circolare attraverso il software Blender, la quale è stata poi decorata con vari mobili, ricavati da un set di prefab gratuiti disponibili sullo store di Unity.
Al centro della stanza è stata posizionata una colonna, e, alle due estremità della colonna, sono stati posizionati due pannelli non renderizzati attraverso i quali l'utente è in grado di osservare il resto della stanza.

Se l'utente osserva la stanza attraverso il portale di sinistra, allora osserverà una stanza, altrimenti, osservando la stanza attraverso il portale di destra, viene renderizzata l'altra.

Implementazione

L'implementazione di tale progetto avviene mediante l'utilizzo di più Stencil Shader, concetto introdotto al termine del capitolo due.
Nella scena vengono generati due Plane, posti alle estremità della colonna centrale. A entrambi è assegnato uno Stencil Shader che assegna un numero di riferimento (1 per il primo portale, 2 per il secondo) e indica di non renderizzare i due Plane.

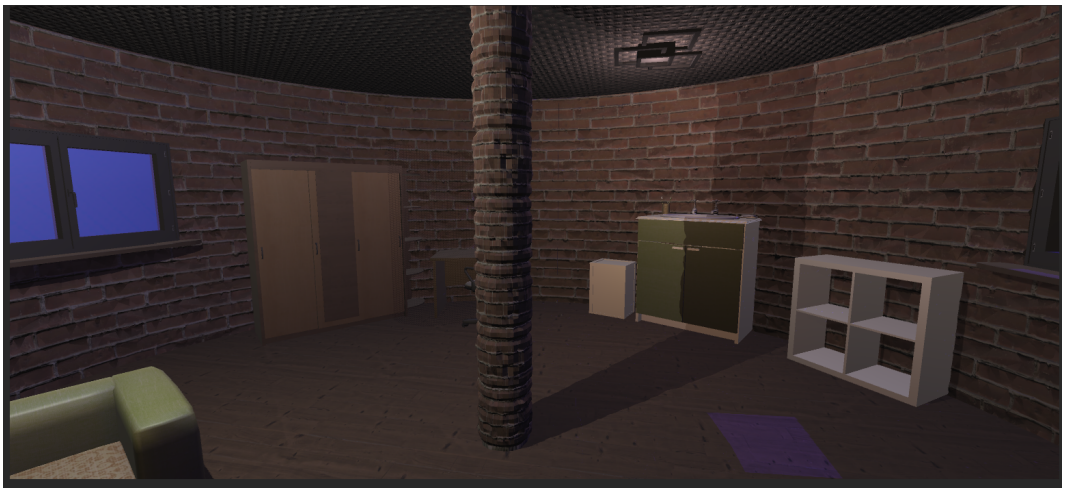


Figura 4.4: Foto della stanza, a sinistra si può notare la stanza da letto mentre a destra si può notare il bagno

Qui è definita una *pass operation* che inserisce nel frame buffer il proprio numero di riferimento.
Sono stati poi generati due shaders per ogni oggetto presente nelle due stanze, uno per ognuna delle due appunto. Anche qui, in entrambi gli shaders è stato assegnato un *ref number* (sempre 1 e 2) e specificando uno *stencil test* attraverso **Comp**. Questo viene definito attraverso un *Enum* e può essere di due tipologie: **Equal** o **NotEqual**. Nel primo caso, verranno renderizzati solo gli oggetti che possiedono lo stesso numero di riferimento, viceversa il secondo caso.
Inoltre, per entrambi i Plane è stato definito uno script, chiamato **PortalTransport** il quale, ogni volta che avviene un evento di tipo *trigger* con la camera dell'utente, va a modificare lo stencil test dei materiali relativi alla stanza di fronte la quale il Plane è posto.

Quello che è accade è dunque questo: se l'utente guarda attraverso un portale, allora vedrà unicamente gli oggetti che hanno lo stesso reference number del portale.

Se l'utente attraversa il portale, allora verrà modificato lo stencil test per ogni materiale, facendo in modo che l'oggetto venga renderizzato solo se di fronte il portale.

Il risultato è dunque una forma *overt* di Impossible Space. Lo spazio disponibile viene aumentato della metà della grandezza della stanza, tuttavia tale cambiamento è palese agli occhi dell'utente.
Seppure il caso di una stanza circolare è estremamente raro, studi futuri possono essere

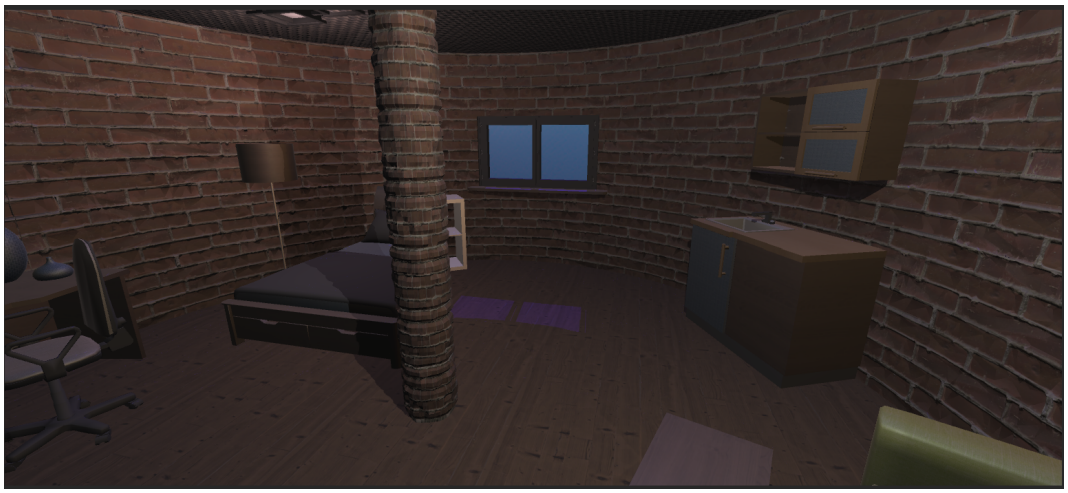


Figura 4.5: Punto di vista differente

condotti sulla percentuale di sovrapposizione che la stanza virtuale può subire senza che l'utente si accorga di tale cambiamento.

Figura 4.6: (a) Stanza da letto (b) Bagno

Capitolo 5

Progetto di tesi

Questo lavoro di tesi si pone l'obiettivo di favorire la locomozione naturale all'interno di lunghi spazi chiusi virtuali, come per esempio dei corridoi, avendo a disposizione uno spazio fisico ristretto e nettamente inferiore rispetto allo spazio virtuale.
Per raggiungere tale obiettivo è stato implementato un algoritmo in grado di manipolare i modelli 3D disponibili, in modo da mutare un corridoio rettilineo in una forma circolare di un diametro pari alla larghezza della stanza in cui l'utente si muove, in questo caso 2 metri, ossia la larghezza della test chamber del laboratorio di Realtà Virtuale dell'università di Bologna.
Il passaggio da una forma rettilinea a una circolare è una forma overt di manipolazione dell'architettura dell'ambiente virtuale in quanto l'utente si accorge immediatamente del cambiamento, tuttavia, viene anche svolta una manipolazione subtle in quanto, se il modello 3D è più lungo della circonferenza di raggio r (dove r è idealmente la metà della lunghezza del lato della stanza nella quale l'utente si muove) allora verranno generate più forme circolari di circonferenza $2r\pi$, ognuna della quali rappresenta una porzione del corridoio e, senza che l'utente si accorge di nulla, mentre questo si muove nel nuovo corridoio circolare verranno aggiunte nuove porzioni di corridoio nel mentre che le precedenti vengono rimosse.
L'obiettivo di questo lavoro è quindi quello di massimizzare lo spazio fisico disponibile per l'utente che si muove all'interno di uno spazio ristretto.
Inoltre, un secondo obiettivo è quello in futuro di analizzare la *threshold* di sovrapposizione del corridoio che un utente può subire senza accorgersi di nulla.

5.0.1 SteamVR

Steam VR è un software di realtà virtuale rilasciato dall'azienda Valve nel marzo 2015 in collaborazione con HTC. Attraverso questo è possibile sperimentare esperienze VR utilizzando varie tipologie di hardware, essendo infatti questo compatibile con la maggior parte dei visori di realtà virtuale attualmente in commercio (HTC Vive, Oculus Rifr, Varjo, e così via). Una volta collegato il proprio visore e avviato Steam VR, questo permette di configurare il proprio ambiente, andando prima a controllare se tutte le com-

ponenti sono attive (headset, controller e base stations) e permettendo poi di eseguire un setup della stanza, delimitando i bordi dello spazio percorribile. Per implementare la componente VR in questo progetto di tesi è stato utilizzato Steam VR all'interno di Unity, attraverso lo specifico plug-in.
Una volta importato, tale plug-in permette di inserire all'interno della scena il *[CameraRig]*, ossia una componente rappresentante l'area di gioco percorribile e contenente la camera, la cui vista viene renderizzata nel visore, e i due controller.
Il movimento in questo caso è avvenuto mediante la locomozione naturale attraverso il tracking dell'headset tramite le base stations.

Figura 5.1: Steam VR Home

5.1 Metodi

Per la realizzazione di questo lavoro è stato utilizzato il motore grafico Unity 3D, versione **2020.3.32f1**, mentre per la fruizione dell'esperienza in realtà virtuale è stato utilizzato l'headset HTC Vive Pro con due base stations utilizzate per il tracking dell'HMD all'interno della stanza.
Al fine di ottenere il risultato desiderato è stata seguita la seguente pipeline:

- Prima di modificare la mesh, viene considerata una circonferenza di raggio r (in questo caso, definito in modo da inscrivere la circonferenza all'interno della stanza del laboratorio). Il modello verrà modificato andando a seguire la traiettoria della circonferenza generata in questo script.
- Una volta importato il modello 3D desiderato, viene applicato uno script chiamato ***Cut***, il compito di questo script è quello di suddividere la mesh in segmenti di lunghezza pari alla lunghezza della circonferenza diviso 10. Una volta coperta tutta la circonferenza, se ci sono altri segmenti allora verranno posizionati con un'altezza maggiore al fine di essere nascosti dalla vista dell'utente.
- Eseguito il taglio della mesh, viene applicato uno script chiamato ***Remesh*** a ognuna delle porzioni generate. Questo script ha il compito di modificare la geometria delle mesh, andando a mappare i vertici lungo la circonferenza precedentemente generata.
- Una volta modificata la geometria delle mesh, viene applicato un ultimo script chiamato ***TeleportMeshes***. Questo, ogni volta che l'utente attraversa uno dei segmenti, in base alla sua posizione andrà a modificare l'altezza dei segmenti, andando ad *aggiungere* il segmento successivo e *rimuovendo* il segmento precedente, posti a una distanza tale da rendere tale modifica impercettibile all'utente e mantenendo un'esperienza fluida.

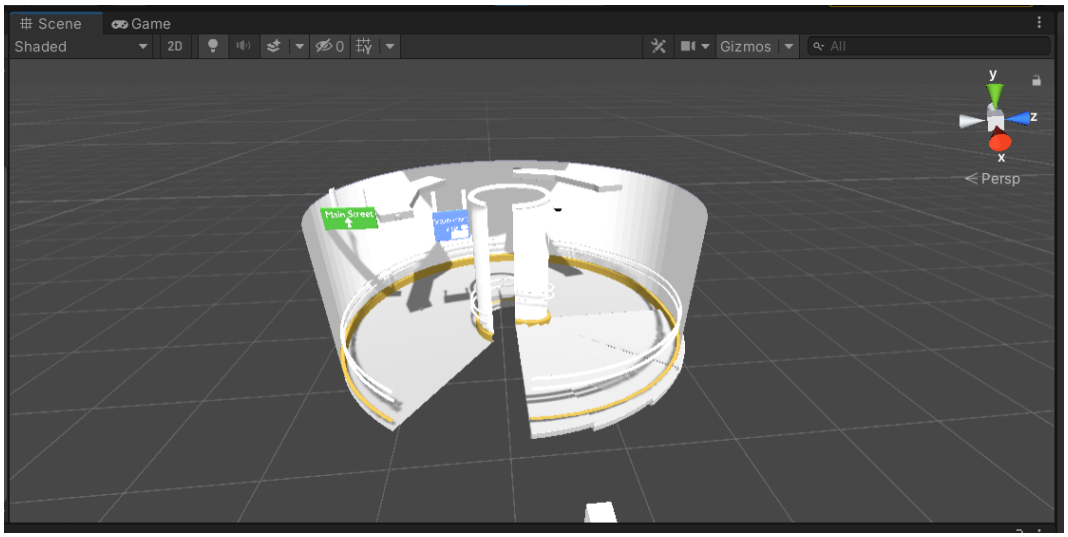


Figura 5.2: Prima del cambiamento

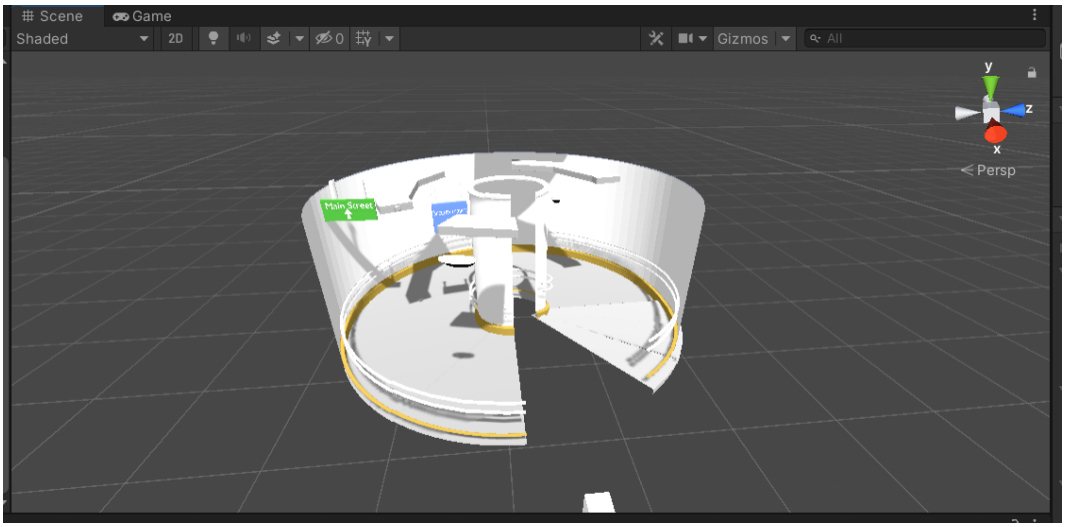


Figura 5.3: Dopo il cambiamento

5.2 Codice

All'interno di questa sezione verrà mostrato il codice relativo all'algoritmo precedentemente descritto.
Il codice in questione è suddiviso in vari script, verranno quindi descritti separatamente.

5.2.1 Circle

```
1
2  public class Circle : MonoBehaviour
3  {
4
5      float radius = 1f;
6
7      const float convertToRadius = 0.017453292519f;
8
9      public GameObject GameObject;
10
11     public float GetCircunference()
12     {
13         return (radius * Mathf.PI * 2);
14     }
15
```

```
16      public Vector3 PositionAlong(float t)
17      {
18          float angleArc = (((t) * 360f)/(float)(Mathf.PI * 2 * radius))
    * convertToRadius;
19          Vector3 position = new Vector3(
20              0.4f * Mathf.Cos(angleArc),
21              0,
22              0.4f * Mathf.Sin(angleArc)
23              );
24
25          return position;
26      }
27  }
```

All'interno di questo script viene si considera una circonferenza di raggio 1 metro (*radius* appunto). Qui, sono state definite due funzioni:

- **GetCircunference()**: utilizzata per restituire la lunghezza della circonferenza.
- **PositionAlong()**: tale funzione è stata utilizzata per mappare la mesh originale lungo la circonferenza. Dato un punto lungo l'asse z chiamato Z_i, viene considerata la posizione dell'arco descritto da tale punto lungo la circonferenza ($[0, Z_i]$) e, attraverso la formula inversa del calcolo dell'arco di circonferenza avendo a disposizione il raggio e l'angolo espresso in gradi, viene calcolato l'angolo generato dall'arco corrispondente e, attraverso le formule trigonometriche di seno e coseno, viene calcolata la posizione del punto Z_i lungo la circonferenza. Al seno e coseno viene poi moltiplicata la distanza dal centro desiderata.

Cut

Questo è lo script utilizzato per il taglio della mesh. Tale operazione avviene grazie a cinque funzioni:

- **Awake()**: nella funzione Awake vengono inizializzate le variabili utilizzate in tutto lo script, soprattutto la lunghezza della circonferenza e la posizione del piano del taglio. Questo viene definito come un piano parallelo agli assi x e y con una componente z uguale a un'unità chiamata *segmentUnit*, pari alla lunghezza della circonferenza diviso 10. Dopo ogni taglio effettuato, la componente z del piano viene incrementata di un'unità.
- **AdjustMesh()**: in questa funzione avviene il riposizionamento della mesh. Vengono considerati tutti i vertici della geometria e a questi viene attribuito un valore

maggiore di zero sia lungo l'asse x che lungo l'asse z, per fare in modo che ogni vertice abbia coordinate positive lungo gli assi:

```
1
2         for (int i = 0; i < Vertices.Length; i++)
3         {
4             if (Vertices[i].z < rightMostZ)
5             {
6                 rightMostZ = Vertices[i].z;
7             }
8         }
9
10
11        for (int i = 0; i < Vertices.Length; i++)
12        {
13            if (Vertices[i].x < rightMostX)
14            {
15                rightMostX = Vertices[i].x;
16            }
17        }
18
19        rightMostPoint = new Vector3(rightMostX, 0, rightMostZ);
20
21        float distanceX = -(transform.TransformPoint(
    rightMostPoint).x - 0);
22        float distanceZ = -(transform.TransformPoint(
    rightMostPoint).z - 0);
23
24        Vector3 distance;
25
26        distance = new Vector3(distanceX, 0, distanceZ);
27
28        for (int i = 0; i < Vertices.Length; i++)
29        {
30            Vertices[i] += distance;
31        }
32
33        newMesh.vertices = Vertices;
34
35
```

- **IntersectionLinePlane()**: questa prende in input due punti e la posizione di un piano parallelo agli assi x e y e restituisce i punti di intersezione tra la retta definita dai due punti e il piano di equazione $z = segmentSize$:

```
1
```

```
        float t = ((planePosition - point1.z) / (point2.z - point1
    .z));
        float x = (point1.x + (point2.x - point1.x) * t);
        float y = (point1.y + (point2.y - point1.y) * t); ;
        float z = (point1.z + (point2.z - point1.z) * t); ;

        intersectionPoint = new Vector3(x, y, z);
        return intersectionPoint;


```

- **Cut()**: questa funzione, prendendo in input una mesh e lo stesso piano della funzione precedente, esegue il taglio della mesh in segmenti lunghi quanto la componente z del piano. Data una mesh di una certe lunghezza l, questa viene suddivisa in un numero $\boldsymbol{n}$ di segmenti pari alla lunghezza circonferenza diviso 10 attraverso un piano parallelo all'asse x e y che scorre lungo la mesh. La suddivisione avviene in questo modo: per ogni submesh relativa alla mesh originale, attraverso un ciclo **for** vengono considerati tutti i vertici della submesh tre a tre (per andare appunto a considerare i triangoli) e per ognuno di questi tre vertici si considera la posizione rispetto al piano del taglio, andando a considerare due mesh di origine:
  - Se tutti e tre i vertici hanno un valore z inferiore alla posizione del piano allora viene aggiunto il triangolo, le normali e gli UV relativi a tali vertici all'interno della prima mesh.
  - Se solamente un vertice ha un valore z inferiore alla posizione del piano allora prima si verifica di quale vertice si tratta, poi vengono generati altri due vertici attraverso la funzione **intersectionLinePlane()** e vengono generati tre nuovi triangoli, il primo, e tutte le informazioni relative ai vertici, verrà aggiunto alla prima mesh, i rimanenti 2 verranno aggiunti alla seconda mesh.
  - Se due vertici hanno un valore z inferiore alla posizione del piano allora, come nel caso precedente, verranno generati altri due vertici e verranno definiti altri tre triangoli, due verranno aggiunti alla prima mesh e uno verrà aggiunto alla seconda.
  - Se infine tutti e tre i vertici hanno un valore z superiore alla posizione del piano di taglio, allora il triangolo e le relative informazioni dei vertici verranno aggiunti alla seconda mesh.

  Infine, verrà richiamata la funzione **MakeGameObj()** per generare due nuovi GameObject, uno relativo alla prima mesh e uno relativo alla seconda:

```
1
2         for (int j = 0; j < count; j++)
3         {
4
5             foreach (var i in meshReq.GetTriangles(j))
6             {
7                 subMeshTri.Add(i);
8             }
9
10            for (int i = 0; i < subMeshTri.Count; i += 3)
11            {
12                bool sideA = (transform.TransformPoint(Vertices[
    subMeshTri[i]]).z >= segSize);
13                bool sideB = (transform.TransformPoint(Vertices[
    subMeshTri[i + 1]]).z >= segSize);
14                bool sideC = (transform.TransformPoint(Vertices[
    subMeshTri[i + 2]]).z >= segSize);
15
16                var sideCount = (sideA ? 1 : 0) + (sideB ? 1 : 0)
    + (sideC ? 1 : 0);
17
18                if (sideCount == 0)
19                {
20                    AddTriangle(2, j, transform.TransformPoint(
    Vertices[subMeshTri[i]]), transform.TransformPoint(Vertices[
    subMeshTri[i + 1]]), transform.TransformPoint(Vertices[
    subMeshTri[i + 2]]),
21                        Normals[subMeshTri[i]], Normals[subMeshTri
    [i + 1]], Normals[subMeshTri[i + 2]],
22                        UV[subMeshTri[i]], UV[subMeshTri[i + 1]],
    UV[subMeshTri[i + 2]]);
23                }
24                else if (sideCount == 1)
25                {
26                    if (sideA == true)
27                    {
28                        intersection1 = intersectionLinePlane(
    transform.TransformPoint(Vertices[subMeshTri[i]]), transform.
    TransformPoint(Vertices[subMeshTri[i + 1]]), segSize);
29                        intersection2 = intersectionLinePlane(
    transform.TransformPoint(Vertices[subMeshTri[i]]), transform.
    TransformPoint(Vertices[subMeshTri[i + 2]]), segSize);
30                        AddTriangle(1, j, transform.TransformPoint
    (Vertices[subMeshTri[i]]), intersection1, intersection2,
31                            Normals[subMeshTri[i]], Normals[
    subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
```

```
32                              UV[subMeshTri[i]], UV[subMeshTri[i +
    1]], UV[subMeshTri[i + 2]]);
33                          AddTriangle(2, j, intersection1, transform
    .TransformPoint(Vertices[subMeshTri[i + 1]]), transform.
    TransformPoint(Vertices[subMeshTri[i + 2]]),
34                              Normals[subMeshTri[i]], Normals[
    subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
35                              UV[subMeshTri[i]], UV[subMeshTri[i +
    1]], UV[subMeshTri[i + 2]]);
36                          AddTriangle(2, j, transform.TransformPoint
    (Vertices[subMeshTri[i + 2]]), intersection2, intersection1,
37                              Normals[subMeshTri[i]], Normals[
    subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
38                              UV[subMeshTri[i]], UV[subMeshTri[i +
    1]], UV[subMeshTri[i + 2]]);
39                          inte.Add(intersection1);
40                          inte.Add(intersection2);
41                      }
42                      else if (sideB == true)
43                      {
44                          ...
45                      }
46                      else
47                      {
48                          ...
49                      }
50                  }
51                  else if (sideCount == 2)
52                  {
53                      if (sideA == sideB)
54                      {
55                          intersection1 = intersectionLinePlane(
    transform.TransformPoint(Vertices[subMeshTri[i + 2]]),
    transform.TransformPoint(Vertices[subMeshTri[i]]), segSize);
56                          intersection2 = intersectionLinePlane(
    transform.TransformPoint(Vertices[subMeshTri[i + 2]]),
    transform.TransformPoint(Vertices[subMeshTri[i + 1]]), segSize)
    ;
57                          AddTriangle(1, j, intersection1, transform
    .TransformPoint(Vertices[subMeshTri[i]]), transform.
    TransformPoint(Vertices[subMeshTri[i+1]]),
58                              Normals[subMeshTri[i]], Normals[
    subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
59                              UV[subMeshTri[i]], UV[subMeshTri[i +
    1]], UV[subMeshTri[i + 2]]);
60                          AddTriangle(1, j, transform.TransformPoint
```

```
(Vertices[subMeshTri[i + 1]]), intersection2, intersection1,
                        Normals[subMeshTri[i]], Normals[
subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
                        UV[subMeshTri[i]], UV[subMeshTri[i +
1]], UV[subMeshTri[i + 2]]);
                    AddTriangle(2, j, transform.TransformPoint
(Vertices[subMeshTri[i + 2]]), intersection1, intersection2,
                        Normals[subMeshTri[i]], Normals[
subMeshTri[i + 1]], Normals[subMeshTri[i + 2]],
                        UV[subMeshTri[i]], UV[subMeshTri[i +
1]], UV[subMeshTri[i + 2]]);
                    inte.Add(intersection1);
                    inte.Add(intersection2);
                }
                else if (sideA == sideC)
                {
                    ...
                }
                else if (sideB == sideC)
                {
                    ...
                }
            }
            else
            {
                AddTriangle(1, j, transform.TransformPoint(
Vertices[subMeshTri[i]]), transform.TransformPoint(Vertices[
subMeshTri[i + 1]]), transform.TransformPoint(Vertices[
subMeshTri[i + 2]]),
                    Normals[subMeshTri[i]], Normals[subMeshTri
[i + 1]], Normals[subMeshTri[i + 2]],
                    UV[subMeshTri[i]], UV[subMeshTri[i + 1]],
UV[subMeshTri[i + 2]]);
            }
        }
        subMeshTri.Clear();
    }
    MakeGameObj(2, this);
    MakeGameObj(1, this);


```

- **AddTriangle()**: questa funzione, prendendo in input tre vertici e le relative informazioni (normali, UV e il triangolo descritto),la mesh nella quale inserirli (la prima o la seconda mesh precedentemente introdotte) e la relativa *subMesh* andrà

a immagazzinare tali informazioni in delle liste che verranno poi lette nella funzione **MakeGameObj()**:

```
        if (side == 1)
        {
            _Triangles[submesh].Add(_Verticies.Count);
            _Verticies.Add(vert1);
            _Triangles[submesh].Add(_Verticies.Count);
            _Verticies.Add(vert2);
            _Triangles[submesh].Add(_Verticies.Count);
            _Verticies.Add(vert3);
            _Normals.Add(normal1);
            _Normals.Add(normal2);
            _Normals.Add(normal3);
            _UVs.Add(uv1);
            _UVs.Add(uv2);
            _UVs.Add(uv3);
        }
        else
        {
            _Triangles2[submesh].Add(_Verticies2.Count);
            _Verticies2.Add(vert1);
            _Triangles2[submesh].Add(_Verticies2.Count);
            _Verticies2.Add(vert2);
            _Triangles2[submesh].Add(_Verticies2.Count);
            _Verticies2.Add(vert3);
            _Normals2.Add(normal1);
            _Normals2.Add(normal2);
            _Normals2.Add(normal3);
            _UVs2.Add(uv1);
            _UVs2.Add(uv2);
            _UVs2.Add(uv3);
        }

```

- **MakeGameObj()**: infine, questa funzione va a generare il GameObject relativo alle informazioni inserite . Inoltre, se la posizione del piano di taglio è maggiore della lunghezza della circonferenza, allora i futuri GameObject che verranno generati da questa funzione avranno un'altezza maggiore rispetto i precedenti, con l'obiettivo di nascondere tali porzioni all'utente per poi mostrarli quando questo si avvicinerà sempre di più.

```
        if (number == 1)
```

```
3         {
4             if (segmentSize >= circunference)
5             {
6                 heightNextLevel = 20;
7             }
8
9             string name = nameMesh + "1" + max;
10            max -= 1;
11            GameObject = new GameObject(name);
12            GameObject.transform.position = new Vector3(original.
    transform.position.x, heightNextLevel, original.transform.position.z
    );
13
14            GameObject.transform.rotation = original.transform.rotation
    ;
15            GameObject.transform.localScale = original.transform.
    localScale;
16
17            var mesh = new Mesh();
18            mesh.indexFormat = UnityEngine.Rendering.IndexFormat.UInt32
    ;
19            mesh.name = original.GetComponent<MeshFilter>().mesh.name;
20
21            mesh.vertices = _Verticies.ToArray();
22            mesh.subMeshCount = count;
23
24            finalTri1 = new int[_Triangles.Count][];
25
26            for (var i = 0; i < _Triangles.Count; i++)
27                finalTri1[i] = _Triangles[i].ToArray();
28
29            for (var i = 0; i < finalTri1.Length; i++)
30                mesh.SetTriangles(finalTri1[i], i, true);
31
32            mesh.normals = _Normals.ToArray();
33            mesh.uv = _UVs.ToArray();
34
35
36            var renderer = GameObject.AddComponent<MeshRenderer>();
37            renderer.materials = original.GetComponent<MeshRenderer>().
    materials;
38
39            var filter = GameObject.AddComponent<MeshFilter>();
40            filter.mesh = mesh;
41
42
```

```
43              mesh.RecalculateNormals();
44              mesh.RecalculateBounds();
45
46
47              if (max != 0)
48              {
49                  _Verticies.Clear();
50                  _Normals.Clear();
51                  _UVs.Clear();
52                  _Triangles.Clear();
53
54                  _Verticies2.Clear();
55                  _Normals2.Clear();
56                  _UVs2.Clear();
57                  _Triangles2.Clear();
58
59                  subMeshTri.Clear();
60
61
62                  Vertices = mesh.vertices;
63                  Triangles = mesh.triangles;
64                  UV = mesh.uv;
65                  Normals = mesh.normals;
66
67                  segmentSize += segmentUnit;
68
69                  destroyNames.Add(name);
70
71                  Cut(segmentSize, mesh);
72              }
73              else
74              {
75                  foreach (string i in destroyNames)
76                  {
77                      GameObject go = GameObject.Find(i);
78                      Destroy(go.gameObject);
79                  }
80                  if (mesh.vertices.Length < 1)
81                  {
82                      Destroy(GameObject);
83                      destroyNames.Add(name);
84                  }
85                  else
86                  {
87                      succesfullNames.Add(name);
88                  }
```

```
89              }
90          }
91          else
92          {
93              ...
94          }
```

5.2.2 Remesh

Questo script ha l'obiettivo di modificare la geometria della mesh per mapparla lunga la circonferenza precedentemente descritta. Questo è definito da due funzioni principali:

- **GetPointAlongCurve()**: questa funzione, preso in input un vertice, restituisce il punto calcolato attraverso la funzione **PositionAlong()** precedentemente vista all'interno dello script Circle.
- **TransformVertex()**: qui avviene la modifica vera e propria, per ogni vertice si calcola una nuova posizione attraverso la funzione **GetPointAlongCurve()**, poi viene calcolato allo stesso modo un punto immediatamente successivo utilizzato per calcolare la rotazione di un piano immaginario che segue la traiettoria della circonferenza. Infine, al punto trovato lungo la circonferenza viene sommata la rotazione moltiplicata per il valore della x del punto nel piano:

```
1
2           float newX = (vertex.x * Circle.radius) / CutTest.sizeX;
3           Vector3 splinePoint = GetPointAlongCurve(vertex);
4
5
6           Vector3 nextPoint = new Vector3(vertex.x + 0.001f, vertex.
    y + 0.001f, vertex.z + 0.001f);
7           Vector3 futureSplinePoint = GetPointAlongCurve(nextPoint);
8
9           Vector3 forwardVector = (futureSplinePoint - splinePoint).
    normalized;
10          Quaternion imaginaryPlaneRotation = Quaternion.
    LookRotation(forwardVector, Vector3.up);
11          Vector3 pointWithinPlane = new Vector3(newX, vertex.y, 0f)
    ;
12
13
14          return splinePoint  /*+ new Vector3(vertex.x, 0f, vertex.z
    );//*/ + imaginaryPlaneRotation * pointWithinPlane;// +
    imaginaryPlaneRotation * pointWithinPlane;
```

```
15
16
```

5.2.3 TeleportMeshes

Infine, una volta effettuato il taglio della mesh e la trasformazione dei vertici, viene applicato lo script **TeleportMeshes()** sulla camera dell'utente. Questo script ha il compito di tenere traccia della posizione di questo lungo la mesh circolare e, nel momento in cui l'utente raggiunge una determinata posizione lungo la mesh, andrà a traslare i segmenti che la compongono in modo da aggiungere nuove porzioni e nascondere le precedenti già viste, mantenendo l'illusione:

```
1
2     private void OnTriggerEnter(Collider other)
3     {
4         int pos = System.Array.IndexOf(namesArray, other.name);
5
6         if((pos > next) && (pos + 4 < namesArray.Length))
7         {
8             Debug.Log("next"+pos);
9             previous = pos - 1;
10            next = pos;
11
12            if (next > previous)
13            {
14                if (previous >= 5)
15                {
16                    GameObject part1 = GameObject.Find(namesArray[
    previous - 5]);
17                    GameObject part2 = GameObject.Find(namesArray[next
    + 4]);
18
19                    part1.transform.Translate(0, 20, 0);
20                    part2.transform.Translate(0, -20, 0);
21                }
22            }
23        }
24
25        if((pos < next) && (pos >= 5))
26        {
27            Debug.Log("back" + pos);
28            previous = pos - 1;
29            next = pos;
30
```

```
31              previousBackward = pos + 1;
32              nextBackward = pos;
33
34              if (nextBackward != previousBackward)
35              {
36                  if (previousBackward + 4 < namesArray.Length)
37                  {
38                      GameObject part1 = GameObject.Find(namesArray[
    previousBackward + 4]);
39                      GameObject part2 = GameObject.Find(namesArray[
    nextBackward - 5]);
40
41                      part1.transform.Translate(0, 20, 0);
42                      part2.transform.Translate(0, -20, 0);
43                  }
44              }
45          }
46
47      }
```

5.3 Risultato

Il risultato è un'esperienza fluida.
La mesh, indipendentemente dalle sue dimensioni, viene correttamente mappata in una traiettoria circolare lungo la quale l'utente si muove senza accorgersi della sostituzione di porzioni di circonferenza, permettendo quindi di percorrere spazi molto lunghi semplicemente muovendosi in cerchio, e quindi massimizzando lo spazio fisico disponibile.
Qui sotto, le immagini relative all'esecuzione dell'algoritmo:

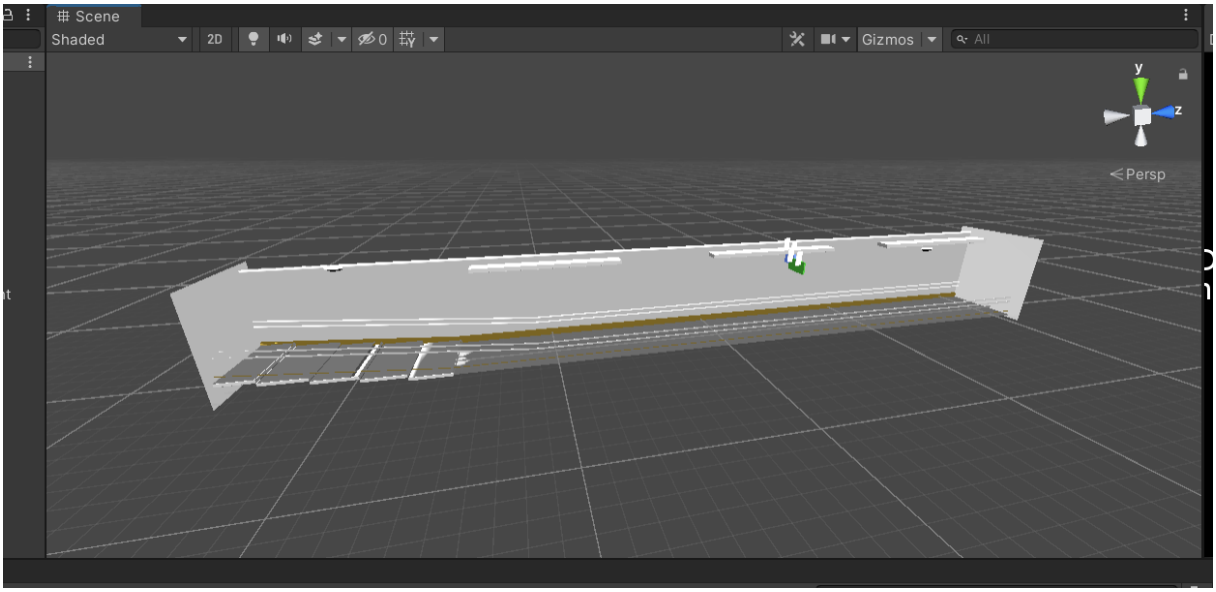


Figura 5.4: Foto prima dell'applicazione dell'algoritmo

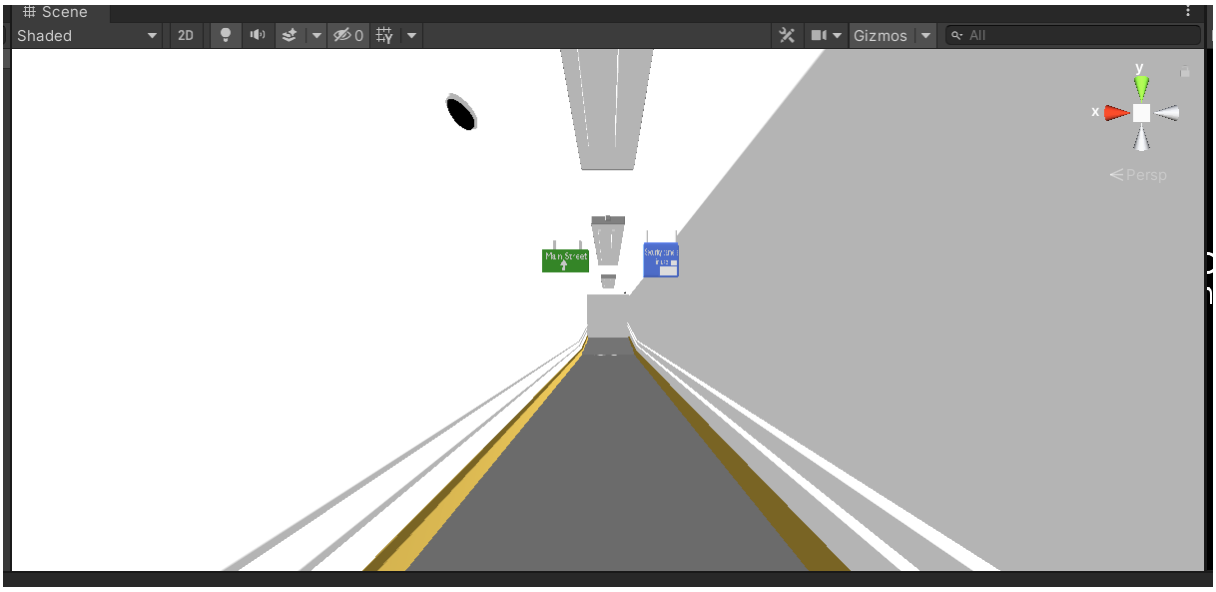


Figura 5.5: Prospettiva all'interno della mesh

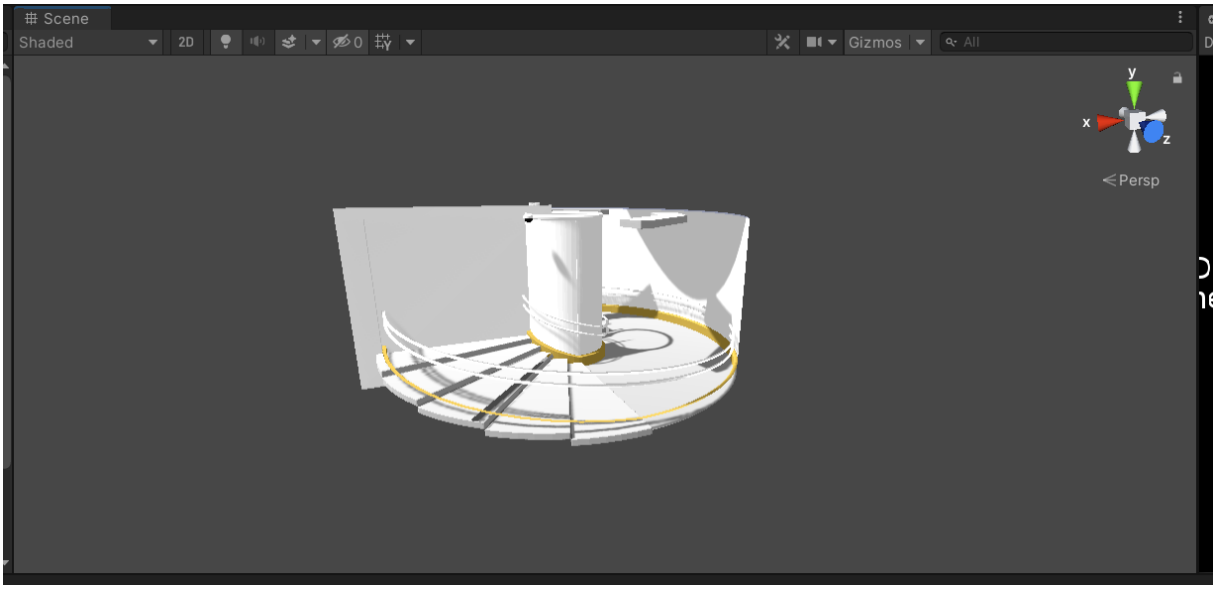


Figura 5.6: Foto dopo l'esecuzione dell'algoritmo

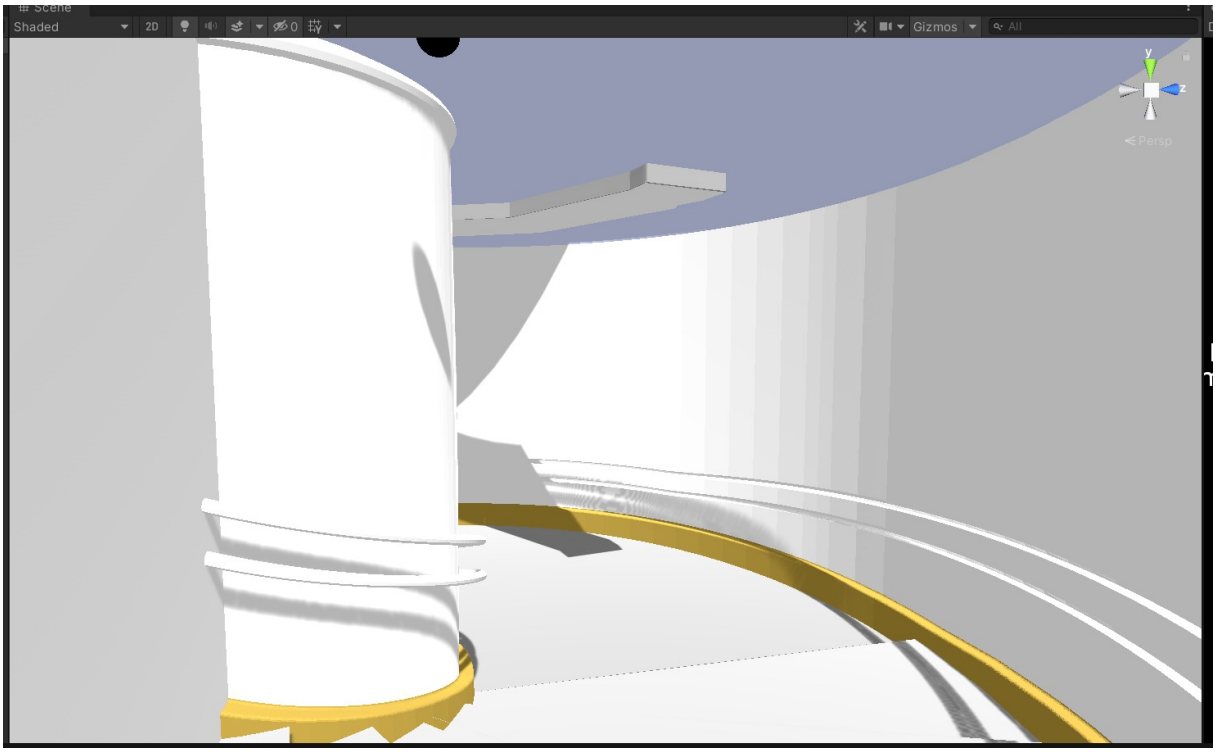


Figura 5.7: Prospettiva all'interno della mesh dopo l'esecuzione

Capitolo 6

Conclusione

Con lo sviluppo dei moderni visori, la realà virtuale sembra essere tornata al centro dell'attenzione del grande pubblico. Le sue applicazioni sono molteplici. Dal semplice mondo dell'intrattenimento raggiunge i campi più disparati come quello dell'istruzione o addirittura la sfera militare.

Inoltre, una delle maggiori risorse della VR è la componente sociale. Le persone possono entrare nel mondo virtuale e comunicare tra di loro attraverso l'utilizzo di un avatar, instaurando relazioni molto simili a quelle del mondo reale.

Tuttavia, un sistema di realtà virtuale ideale richiede il massimo livello di immersività che un utente può raggiungere. Ciò avviene in presenza di varie condizioni: attraverso avatar realistici, attraverso la sperimentazione di un sense of embodiment molto forte e attraverso il corretto mapping tra il movimento dell'utente nel mondo reale e quello nel mondo virtuale.

Tuttavia, oltre al costo eccessivo, uno dei problemi sostanziali per la VR è la differenza tra lo spazio fisico e quello virtuale. Tale problema porta, infatti, l'utente a muoversi in uno spazio virtuale differente rispetto allo spazio fisico disponibile, che prende il nome di *matched zone*, soffrendo delle limitazioni che quest'ultimo porta.

Lo scopo di questo lavoro di tesi è la realizzazione di un algoritmo che vada a modificare un modello 3D da una forma rettilinea a una curvilinea, al fine di poter percorrere tale modello all'interno di uno spazio fisico ristretto. Nei mesi dedicati allo sviluppo di questo algoritmo, sono stati affrontati e trattati molteplici argomenti. Dalla modellazione 3D, dalle nozioni basilari di Computer Graphics, dallo studio del movimento nella VR fino alla geometria analitica nel piano e nello spazio.

L'algoritmo sviluppato è una forma *overt* di manipolazione dell'architettura virtuale, di conseguenza percettibile all'utente, tuttavia esegue anche delle manipolazioni di tipo *subtle*, andando a mostrare e nascondere segmenti della mesh mentre l'utente cammina in cerchio. In futuro, l'obiettivo sarà quello di condurre uno studio con più utenti per individuare la percentuale di sovrapposizione che un utente può sperimentare nella scena senza accorgersi di nulla, ossia quanto la mesh può sovrapporsi a sé stessa sen-

za che l'utente si accorga di aver percorso precedentemente quello specifico punto della circonferenza.

Un ulteriore studio aggiuntivo sarà condotto per studiare la percentuale di curvatura che può essere applicata al modello 3D senza che venga percepita dall'utente.

Bibliografia

[1] D. Reipur L. Embol N. C. Nilsson S. Serafin A. Junker, C. Hutters. Revisiting audiovisual rotation gains for redirected walking. *IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 2021.

[2] A. Zenner M. Speicher F. Daiber A. Simeone, N. C. Nilsson. The space bender: Supporting natural walking via overt manipulation of the virtual environment. *In Proceedings of the 2020 IEEE Conference on Virtual Reality and 3D User Interfaces (VR 2020)*, 2020.

[3] Ifigeneia Mavridou Adalberto L. Simeone and Wendy Powell. Altering user movement behaviour in virtual environments. *IEEE Transactions on Visualization and Computer Graphics 23, 4, 1312-1321.*

[4] Ronald T Azuma. A survey of augmented reality. *Presence: teleoperators & virtual environments*, 6(4):355–385, 1997.

[5] M. Lappe B. Bolte. Subliminal reorientation and repositioning in immersive virtual environments using saccadic suppression. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS*, 2015.

[6] Jen-Hao Cheng, Yi Chen, Ting-Yi Chang, Hsu-En Lin, Po-Yao Cosmos Wang, and Lung-Pan Cheng. Impossible staircase: Vertically real walking in an infinite virtual tower. In *2021 IEEE Virtual Reality and 3D User Interfaces (VR)*, pages 50–56. IEEE Computer Society, 2021.

[7] Robbe Cools and Adalberto L Simeone. Investigating the effect of distractor interactivity for redirected walking in virtual reality. In *Symposium on Spatial User Interaction*, pages 1–5, 2019.

[8] S. Finkelstein Z. Wartell D. Krum M. Bolas E. Suma, S. Clark. Leveraging change blindness for redirection in virtual environments. *IEEE Virtual Reality*, 2011.

[9] Samantha Finkelstein David M. Krum Mark Bolas Evan A. Suma, Zachry Lipps. Impossible spaces: Maximizing natural walking in virtual environments with self-overlapping architecture. In *IEEE Transactions on visualization and computer graphics*. IEEE, 2012.

[10] T. Ropinski K. Hinrichs F. Steinicke, G. Bruder. Moving towards generally applicable redirected walking.

[11] C Martin Grewe, Tuo Liu, Christoph Kahl, Andrea Hildebrandt, and Stefan Zachow. Statistical learning of facial expressions improves realism of animated avatar faces. *Frontiers in Virtual Reality*, 2:11, 2021.

[12] Victoria Interrante, Brian Ries, and Lee Anderson. Seven league boots: A new metaphor for augmented locomotion through moderately large scale immersive virtual environments. In *2007 IEEE Symposium on 3D User interfaces*. IEEE, 2007.

[13] M. Bolas E. Suma K. Vasylevska, H. Kaufmann. Flexible spaces: Dynamic layout generation for infinite walking in virtual environments. *IEEE Symposium on 3D User Interfaces*, 2013.

[14] Daniel Christopher Lochner and James Edward Gain. Vr natural walking in impossible spaces. In *Motion, Interaction and Games*, pages 1–9. 2021.

[15] P. Maes M. Sra, S. Garrido-Jurrado. Oasis: Procedurally generated social virtual spaces from 3d scanned real spaces. *IEEE Virtual Reality*, 2017.

[16] P Milgram, H Takemura, A Utsumi, and F Kishino. Augmented reality: a class of displays on the reality-virtuality continuum. telemanipulator telepresence technol. 2351, 282–292 (1994), 2013.

[17] G. Bruder E. Hodgson S. Serafin M Whitton E. Suma F. Steinicke N. C. Nilsson, T.Peck. 15 years of research on redirected walking in immersive virtual environments. *IEEE Computer Graphics and Applications*, 2018.

[18] Niels Christian Nilsson, Stefania Serafin, Frank Steinicke, and Rolf Nordahl. Natural walking in virtual reality: A review. *Computers in Entertainment (CIE)*, 16(2):1–22, 2018.

[19] Tabitha C Peck, Henry Fuchs, and Mary C Whitton. Evaluation of reorientation techniques and distractors for walking in large virtual environments. *IEEE transactions on visualization and computer graphics*, 15(3):383–394, 2009.

[20] Adalberto L Simeone, Eduardo Velloso, and Hans Gellersen. Substitutional reality: Using the physical environment to design virtual reality experiences. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 3307–3316, 2015.

[21] Janujah Sivanandan, Eugene Liscio, and P Eng. Assessing structured light 3d scanning using artec eva for injury documentation during autopsy. *J Assoc Crime Scene Reconstr*, 21:5–14, 2017.

[22] Anthony Steed and Ralph Schroeder. Collaboration in immersive and non-immersive virtual environments. In *Immersed in Media*, pages 263–282. Springer, 2015.

[23] Frank Steinicke, Gerd Bruder, Klaus Hinrichs, Markus Lappe, Brian Ries, and Victoria Interrante. Transitional environments enhance distance perception in immersive virtual reality systems. In *Proceedings of the 6th Symposium on Applied Perception in Graphics and Visualization*, pages 19–26, 2009.

[24] Frank Steinicke, Gerd Bruder, Klaus Hinrichs, Anthony Steed, and Alexander L Gerlach. Does a gradual transition to the virtual world increase presence? In *2009 IEEE Virtual Reality Conference*, pages 203–210. IEEE, 2009.

[25] Betsy Williams, Gayathri Narasimham, Tim P McNamara, Thomas H Carr, John J Rieser, and Bobby Bodenheimer. Updating orientation in large virtual environments using scaled translational gain. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 21–28, 2006.

[26] Betsy Williams, Gayathri Narasimham, Bjoern Rump, Timothy P McNamara, Thomas H Carr, John Rieser, and Bobby Bodenheimer. Exploring large virtual environments with an hmd when physical space is limited. In *Proceedings of the 4th symposium on Applied perception in graphics and visualization*, pages 41–48, 2007.

[27] Nick Yee and Jeremy Bailenson. The proteus effect: The effect of transformed self-representation on behavior. *Human communication research*, 33(3):271–290, 2007.