Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione
A.a. 2022/2023
Sessione di Laurea Dicembre 2023

Progettazione e sviluppo di un'applicazione in realtà aumentata a supporto dell'apprendimento della Fisica

Relatori:
Giovanni Malnati
Fabio Forno

Candidato:
Luca Pauri

Indice

Introduzione **1**

1 Utilizzo di realtà aumentata e virtuale **2**

1.1 Tipi di Realta: AR e VR 2

1.1.1 Realtà aumentata 2

1.1.2 Realtà virtuale 4

1.2 Utilizzo di AR e VR per scopi didattici e di formazione 6

1.3 Motivazioni dell'utilizzo di AR e VR nell'apprendimento della Fisica . . 9

2 Scelta della piattaforma **11**

2.1 Unity 11

2.1.1 Caratteristiche generali 11

2.1.2 Interfaccia 12

2.1.3 Concetti principali 14

2.2 Oculus Quest 2 15

2.2.1 Caratteristiche generali 15

2.2.2 Setup Unity per Oculus 17

3 Image tracking **20**

3.1 Image tracking con ARFoundation 20

3.1.1 Scelta delle immagini 20

3.1.2 Setup della scena 22

3.1.3 Implementazione del tracciamento 23

3.2 Image tracking con AprilTag 27

3.2.1 AprilTag 27

3.2.2 Setup della scena 29
3.2.3 Implementazione del tracciamento 30

4 Sensori **34**
4.1 Arduino Nano 33 BLE Sense Rev2 34
4.1.1 Specifiche della scheda 35
4.1.2 Arduino_BMI270_BMM150 36
4.1.3 Calibrazione dell'accelerometro 37
4.2 CQRobot VL53L1X 39
4.2.1 Specifiche del sensore 39
4.2.2 Arduino VL53L1X 41
4.3 Connessione tramite Bluetooth Low Energy 42
4.3.1 ArduinoBLE 43
4.3.2 Unity Android BLE Plugin 43

5 Costruzione dell'esperimento **47**
5.1 Moto rettilineo uniforme 47
5.2 Moto rettilineo uniformemente accelerato 49
5.3 Moto sul piano inclinato 51

6 Sensor fusion **53**
6.1 Algoritmi di sensor fusion 53
6.2 Filtro di Kalman 54
6.2.1 Descrizione dell'algoritmo 54
6.2.2 Modellazione degli attriti 57
6.2.3 Implementazione in Unity 63
6.3 CUSUM 67
6.3.1 Descrizione dell'algoritmo 67
6.3.2 Implementazione in Unity 68

7 Setup dell'applicazione **70**
7.1 Connessione del sensore e selezione dei parametri 71
7.1.1 Selezione tramite menù 71

7.1.2 Punto iniziale e di arrivo 75
7.2 Simulazione 77
7.3 Visualizzazione dei vettori 80

Conclusioni e sviluppi futuri **85**

Bibliografia **86**

Introduzione

Nel contesto dell'evoluzione digitale, l'apprendimento assistito da tecnologie immersive come la Realtà Aumentata (AR) e la Realtà Virtuale (VR) sta rivoluzionando il modo in cui acquisiamo e interiorizziamo le conoscenze. Questa tesi si concentra sul design e sullo sviluppo di un'applicazione per amplificare l'apprendimento della Fisica attraverso l'integrazione sinergica di AR e VR. L'apprendimento della Fisica è spesso affrontato con sfide legate alla comprensione concettuale e all'applicazione pratica dei principi scientifici. L'uso combinato di AR e VR offre un ambiente educativo immersivo che supera le barriere tradizionali, offrendo agli studenti un'esperienza di apprendimento coinvolgente e interattiva.

La fase iniziale di questo progetto ha visto il design dell'applicazione, durante la quale sono state prese decisioni cruciali per la scelta delle piattaforme software e hardware. L'attenzione è stata posta su Unity come ambiente di sviluppo e su Oculus Quest 2 come dispositivo target, garantendo un'esperienza di realtà virtuale senza fili e di alta qualità. Successivamente, sono state prese decisioni fondamentali per la fase di sperimentazione all'interno dell'applicazione. Concentrandosi sull'apprendimento della cinematica, è stato scelto di replicare l'esperimento della rotaia con cuscino d'aria, un classico nelle lezioni di Fisica. La simulazione di questo esperimento richiede la raccolta di dati precisi, pertanto sono state fatte scelte strategiche riguardanti i sensori da utilizzare e la costruzione dell'esperimento virtuale stesso.

La fase di sviluppo ha visto l'implementazione di un algoritmo sofisticato per la fusione dei dati provenienti da diversi sensori, ottenendo così i dati necessari per alimentare la simulazione. Ulteriori sforzi sono stati dedicati alla gestione degli input degli utenti per vari parametri dell'esperimento. La visualizzazione delle grandezze fisiche vettoriali, come la velocità e le diverse forze in gioco, è stata gestita per garantire una rappresentazione accurata e comprensibile. L'obiettivo finale è fornire agli studenti uno strumento che non solo renda l'apprendimento della Fisica più accessibile, ma lo trasformi in un'esperienza entusiasmante e coinvolgente.

Capitolo 1

Utilizzo di realtà aumentata e virtuale

L'obiettivo di questo capitolo è introdurre i concetti di realtà aumentata e realtà virtuale, e di discuterne gli utilizzi a fini didattici e di formazione. Si vogliono inoltre introdurre le ragioni che hanno spinto a volerne fare uso nell'ambito dell'apprendimento della Fisica.

1.1 Tipi di Realta: AR e VR

Il primo passo è quello di definire e differenziare i due diversi tipi di Realtà, si può infatti avere:

- Realtà aumentata (AR)
- Realtà virtuale (VR)

Si andrà dunque adesso a descrivere le caratteristiche di entrambe, evidenziandone le differenze. [1]

1.1.1 Realtà aumentata

La realtà aumentata (AR) è una tecnologia avanzata che sovrappone elementi virtuali al mondo fisico, creando un'esperienza immersiva e interattiva per gli utenti. L'AR si basa sull'uso di dispositivi come smartphone, occhiali intelligenti o visori AR dedicati, che utilizzano telecamere, sensori e software per rilevare l'ambiente circostante e aggiungere elementi digitali in tempo reale. Questi elementi possono includere oggetti 3D, testo, video, audio o informazioni interattive.

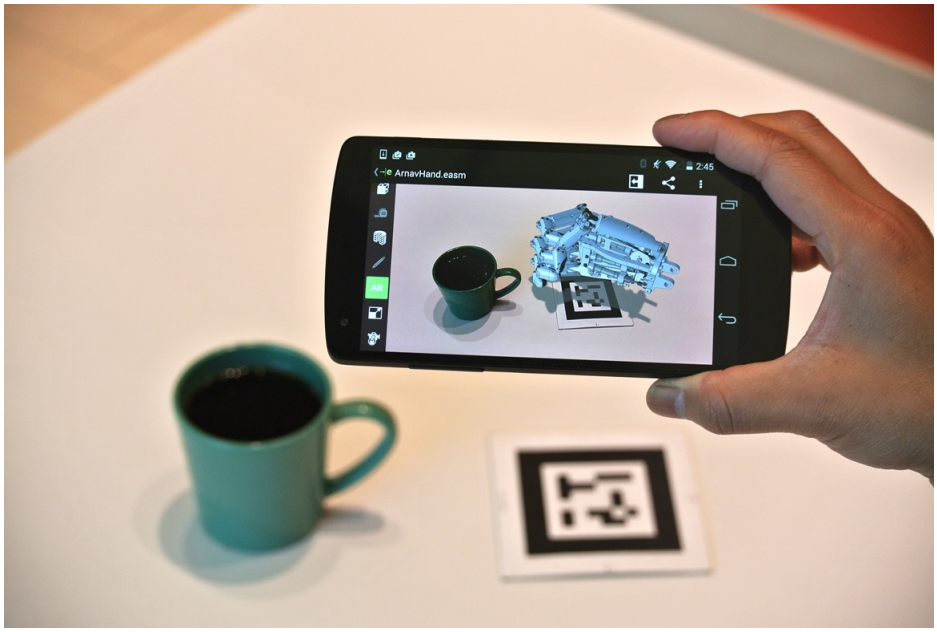


Figura 1.1: Esempio di AR.

Le caratteristiche principali dell'AR includono:

- **Tracking:** L'AR utilizza algoritmi di rilevamento e localizzazione per posizionare con precisione gli oggetti virtuali nell'ambiente reale. Questo assicura che gli oggetti virtuali appaiano stabili e coerenti, indipendentemente dai movimenti dell'utente.

- **Visualizzazione:** Gli oggetti virtuali possono essere visualizzati attraverso dispositivi come schermi di smartphone, visori AR o occhiali intelligenti. Alcuni visori AR forniscono un'esperienza hands-free proiettando gli oggetti direttamente nell'ambiente circostante.

- **Interazione:** Gli utenti possono interagire con gli oggetti AR attraverso gesti, comandi vocali o dispositivi di input, consentendo loro di manipolare, spostare o modificarli in tempo reale.

- **Applicazioni:** L'AR ha una vasta gamma di applicazioni. È utilizzata in giochi per la creazione di esperienze di gioco immersivo, nell'industria per l'assistenza in fase di progettazione e di manutenzione, nell'istruzione per la creazione di modelli didattici interattivi, nell'arte per la produzione di opere d'arte digitali e nella medicina per simulazioni chirurgiche e navigazione.

- **Evoluzione:** La tecnologia AR sta evolvendo costantemente. Alcuni sviluppi recenti includono il riconoscimento e la mappatura degli ambienti in tempo reale (SLAM), la realtà aumentata basata su cloud e la fusione con altre tecnologie come la realtà virtuale per creare esperienze miste.
- **Futuro:** L'AR sta guadagnando popolarità in settori come il commercio, il turismo, la formazione e l'assistenza sanitaria. Con il continuo avanzamento della tecnologia, ci si aspetta che l'AR diventi sempre più integrata nella vita quotidiana, trasformando la nostra interazione con il mondo circostante e aprendo nuove prospettive in termini di creatività, apprendimento e comunicazione.

In sintesi, la realtà aumentata è una tecnologia che promette di rivoluzionare il modo in cui interagiamo con il mondo fisico, offrendo esperienze sempre più ricche e sofisticate, e aprendo nuove opportunità in numerosi settori.

1.1.2 Realtà virtuale

La realtà virtuale (VR) è una tecnologia avanzata che crea ambienti e esperienze immersivi e tridimensionali attraverso dispositivi e simulazioni computerizzate. Questi ambienti virtuali possono essere esplorati da un utente come se fossero reali, dando vita a un'esperienza coinvolgente e interattiva.

Figura 1.2: Utente immerso in VR.

Tra le caratteristiche principali della realtà virtuale si possono annoverare:

- **Ambienti virtuali:** La realtà virtuale crea ambienti simulati digitalmente che possono variare da mondi fantastici a simulazioni realistiche di luoghi del mondo reale. Gli ambienti sono progettati per essere esplorati e vissuti attraverso l'uso di dispositivi di visualizzazione, come visori VR o occhiali intelligenti.

- **Dispositivi di visualizzazione:** Gli utenti indossano visori VR o occhiali intelligenti che coprono completamente il campo visivo. Questi dispositivi includono schermi per gli occhi, sensori di movimento e audio integrato per creare una percezione sensoriale completa.

- **Tracciamento del movimento:** Per garantire l'immersione, la VR monitora il movimento dell'utente in modo che gli ambienti virtuali rispondano ai movimenti della testa e del corpo. Questo tracking può essere basato su sensori esterni o sensori integrati nei dispositivi di visualizzazione.

- **Interazione:** Gli utenti possono interagire con l'ambiente virtuale tramite controller, gesti, comandi vocali o movimenti del corpo. Questa interazione può variare da semplici azioni, come puntare e sparare, a complessi puzzle da risolvere o simulazioni scientifiche.

- **Applicazioni:** La realtà virtuale trova applicazione in diversi settori, tra cui giochi, istruzione, medicina, addestramento, progettazione industriale, turismo, e altro ancora. Ad esempio, gli studenti possono esplorare antiche civiltà, i medici possono praticare procedure chirurgiche, e gli appassionati di viaggi possono visitare virtualmente luoghi esotici.

- **Simulazioni realistiche:** La VR può simulare situazioni complesse e realistiche, creando un'esperienza utente convincente. Questo è utile per l'addestramento militare, simulazioni di volo, simulazioni mediche e molto altro.

- **Sfide tecniche:** La realtà virtuale presenta sfide tecnologiche, tra cui la necessità di schermi ad alta risoluzione, latenza bassa, grafica avanzata e audio realistico per garantire un'esperienza fluida e convincente.

- **Evoluzione tecnologica:** La VR è in continua evoluzione, con nuove tecnologie e miglioramenti che stanno emergendo costantemente. Ciò include dispositivi wireless, esperienze sociali condivise in VR e integrazione con la realtà aumentata per creare esperienze miste.

In sintesi, la realtà virtuale è una tecnologia che trasforma la nostra percezione, consentendoci di esplorare mondi virtuali e interagire con essi in modi innovativi. È ampiamente utilizzata in settori diversi e continua a evolversi, aprendo nuove opportunità in termini di intrattenimento, formazione, simulazioni e molto altro.

1.2 Utilizzo di AR e VR per scopi didattici e di formazione

Realtà virtuale e realtà aumentata stanno trovando negli ultimi anni innumerevoli campi di applicazione. Tra questi è interessanti l'uso che se ne può fare nel campo della didattica e della formazione. Infatti queste tecnologie permettono di visualizzare cose non percepibili nell'immediato, ma anche di procedere all'addestramento in attività molto costose o particolarmente pericolose.

Partendo da quest'ultimo punto l'esempio più immediato che si può fare è quello dei simulatori di volo, in cui l'applicazione della realtà virtuale ha rivoluzionato l'addestramento dei piloti, offrendo un ambiente sicuro e controllato per acquisire competenze senza alcun rischio. I simulatori di volo in VR possono riprodurre una vasta gamma di scenari, tra cui condizioni meteorologiche avverse, guasti dei motori, e situazioni di emergenza. Ciò permette ai piloti di acquisire esperienza nella gestione di situazioni critiche in un ambiente virtuale prima di affrontarle nella realtà. L'utilizzo di simulatori di volo in VR inoltre riduce significativamente i costi associati all'addestramento tradizionale, come il carburante, la manutenzione degli aeromobili e la logistica. I piloti possono esercitarsi senza dover utilizzare effettivamente un aeroplano, consentendo un addestramento più efficiente ed economico.

Figura 1.3: Simulatore di volo.

Un altro campo in cui la formazione può beneficiare dell'utilizzo di queste tecnologie è quello medico, e a questo proposito si può portare come esempio HoloAnatomy [2], un'applicazione di realtà mista sviluppata per dispositivi di realtà aumentata come HoloLens.

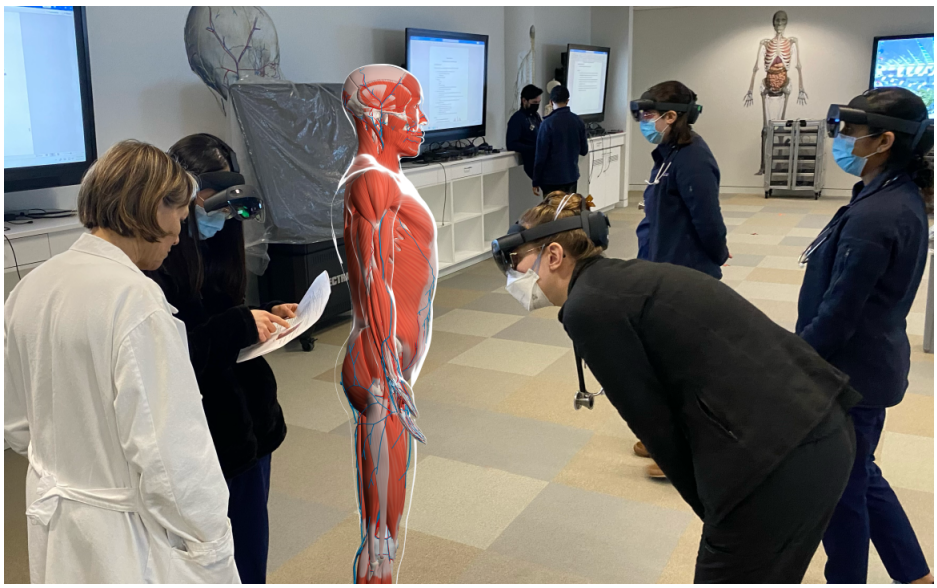


Figura 1.4: HoloAnatomy.

HoloAnatomy consente agli studenti di medicina e ai professionisti del settore di visualizzare il corpo umano in modo completamente tridimensionale e immersivo, consentendo un'ispezione dettagliata degli organi, dei sistemi e delle strutture anatomiche. Gli utenti possono manipolare virtualmente gli organi, ruotandoli, ingrandendoli o addirittura rimuovendoli per ottenere una comprensione più approfondita delle relazioni anatomiche e delle strutture interne. HoloAnatomy è particolarmente utile per l'addestramento virtuale a procedure mediche complesse. Gli studenti possono simulare interventi chirurgici o pratiche diagnostiche in un ambiente virtuale, riducendo così i rischi associati all'apprendimento pratico in situazioni reali e migliorando le loro abilità pratiche in modo sicuro e controllato.

Esplorando invece il campo della didattica si può trovare l'esempio di Electrostatic Playground [3], un'applicazione educativa basata in VR che offre un modo interattivo e coinvolgente per esplorare i principi dell'elettrostatica. Questa applicazione è progettata per offrire un ambiente virtuale dove gli utenti possono sperimentare e comprendere i concetti chiave dell'elettrostatica in modo pratico e visuale.

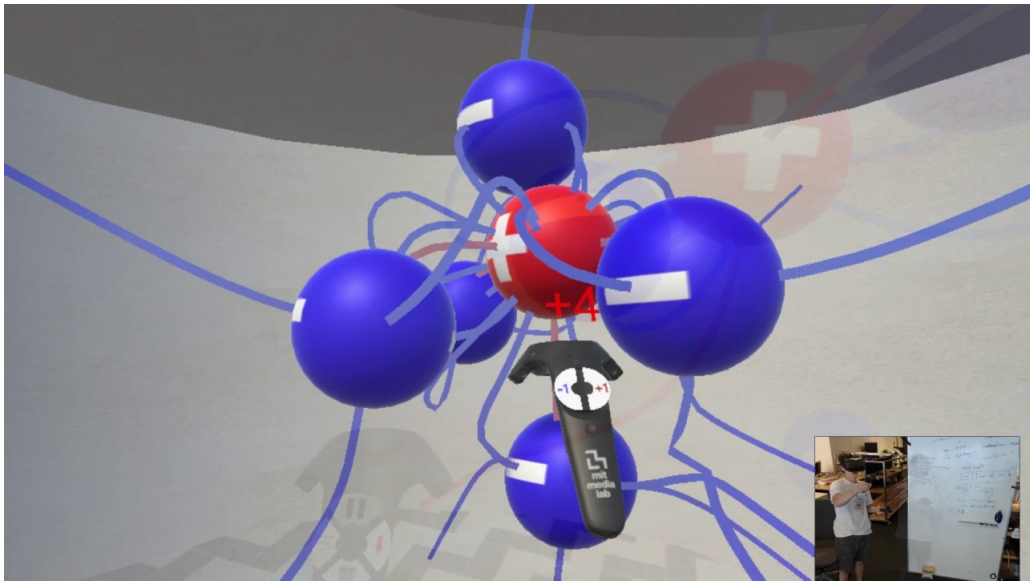


Figura 1.5: Electrostatic Playground.

Electrostatic Playground consente agli utenti di creare e manipolare cariche elettriche virtuali direttamente nel loro ambiente fisico, di osservare come queste interagiscono tra loro e sperimentare i principi fondamentali dell'elettrostatica. Gli utenti possono visualizzare le linee di forza elettrica generate dalle cariche, ottenendo un'illustrazione chiara dei modelli di campo elettrico. Questa caratteristica facilita la comprensione delle influenze delle cariche sulla regione circostante.

Un esempio di utilizzo della realtà aumentata nell'apprendimento si può trovare invece in un progetto sviluppato dalla harvard University [4] [5].

In questa applicazione la AR è utilizzata per l'insegnamento individuale da remoto sempre dell'elettromagnetismo. L'applicazione fornisce la possibilità di visualizzare i poli magnetici di una calamita e le linee di campo magnetico generate. Vi è inoltre la possibilità di visualizzare la sezione di una bobina e forza e direzione dei campi magnetici ed elettrici indotti quando la calamita e la bobina interagiscono. In questo modo, come nei casi precedenti, è possibile visualizzare qualcosa di invisibile nella realtà e di facilitare l'apprendimento di concetti complessi come possono essere quelli riguardanti l'induzione elettromagnetica.

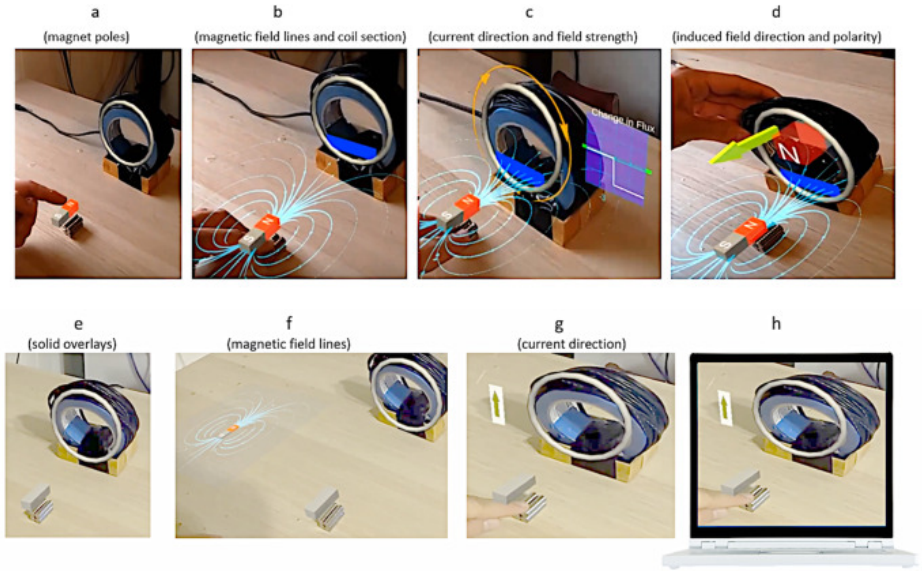


Figura 1.6: Illustrazione degli esperimenti.

1.3 Motivazioni dell'utilizzo di AR e VR nell'apprendimento della Fisica

La fusione della realtà aumentata e della realtà virtuale emerge come una straordinaria frontiera nell'ambito dell'insegnamento della Fisica, apportando innovazioni sostanziali alla modalità tradizionale di trasmissione dei concetti scientifici. Queste tecnologie offrono un'opportunità senza precedenti per rendere tangibili concetti astratti, consentendo agli studenti di immergersi in un ambiente educativo tridimensionale e interattivo. Attraverso l'utilizzo di dispositivi AR e VR, gli studenti possono visualizzare e manipolare vettori, esplorare campi elettrici e confrontare modelli ideali con la realtà, trasformando l'astrazione teorica in un'esperienza concreta.

L'aspetto tridimensionale delle simulazioni in VR permette agli studenti di esplorare lo spazio fisico in modi precedentemente inimmaginabili, avvicinandoli a fenomeni scientifici complessi in modo intuitivo. Le simulazioni realistiche, create attraverso la realtà virtuale, consentono agli studenti di condurre esperimenti virtuali che potrebbero essere difficili o pericolosi nella realtà, ampliando così le possibilità di apprendimento. Inoltre, l'interattività offerta da modelli 3D in AR può favorire una comprensione più profonda delle forze, dei movimenti e delle interazioni tra oggetti.

Un elemento cruciale dell'utilizzo di AR e VR nell'insegnamento della Fisica è la possibilità di esplorare la differenza tra modelli ideali e la realtà. Gli studenti possono

manipolare e osservare come variazioni nelle condizioni o nelle scale influenzino i risultati, contribuendo così a una comprensione più approfondita dei concetti fisici. Inoltre, l'esperienza immersiva della realtà virtuale può aumentare significativamente l'interesse e la motivazione degli studenti, fornendo un contesto coinvolgente per l'apprendimento.

Infine, la capacità di svolgere attività educative collaborative attraverso AR e VR offre una nuova dimensione all'apprendimento, permettendo agli studenti di esplorare insieme concetti complessi, condividendo osservazioni e costruendo conoscenze in modo condiviso. In questo contesto, la realtà aumentata e la realtà virtuale emergono non solo come strumenti educativi innovativi, ma come facilitatori di un approccio più inclusivo e collaborativo all'insegnamento della Fisica.

Capitolo 2

Scelta della piattaforma

In questo capitolo si andranno a discutere le scelte fatte per quanto riguarda piattaforme e dispositivi scelti per lo sviluppo dell'applicazione, indagando le loro caratteristiche principali così come i motivi che hanno portato a sceglierli.

2.1 Unity

Unity [6] è una potente e versatile piattaforma di sviluppo di videogiochi e applicazioni interattive. La sua caratteristica distintiva è la sua capacità di creare esperienze 2D e 3D immersive su una vasta gamma di piattaforme, tra cui computer desktop, console, dispositivi mobili, realtà virtuale (VR) e realtà aumentata (AR).

2.1.1 Caratteristiche generali

Le principali caratteristiche di questa piattaforma comprendono:

- **Motore di gioco:** Unity è noto per il suo robusto motore di gioco che consente agli sviluppatori di creare grafica di alta qualità, fisica realistica e animazioni fluide. Supporta il rendering 2D e 3D, offrendo un ambiente di sviluppo altamente flessibile.
- **Editor visuale:** Unity offre un editor visuale intuitivo che semplifica la creazione, la modifica e la gestione di scenari di gioco. Gli sviluppatori possono trascinare e rilasciare risorse, aggiungere script e impostare parametri in modo semplice.

- **Linguaggio di scripting:** Unity supporta diversi linguaggi di scripting, tra cui C#, JavaScript e Boo. C# è il linguaggio più comunemente utilizzato ed è noto per la sua sintassi pulita e la sua potente capacità di programmazione.

- **Asset Store:** Unity fornisce un vasto Asset Store che offre una varietà di risorse pronte all'uso, tra cui modelli 3D, effetti sonori, texture, script e plugin. Questo rende più facile per gli sviluppatori accedere a contenuti di alta qualità e accelerare il processo di sviluppo.

- **Simulazioni fisiche:** Unity include un motore fisico integrato che consente di simulare il comportamento realistico degli oggetti all'interno del gioco. Questo è fondamentale per creare interazioni coinvolgenti e realistiche.

- **Supporto multi-piattaforma:** Unity è noto per il suo supporto per una vasta gamma di piattaforme, tra cui Windows, macOS, iOS, Android, console di gioco, VR e AR. Ciò consente agli sviluppatori di creare un gioco una sola volta e distribuirlo su più piattaforme con relativa facilità.

- **Realtà virtuale e aumentata:** Unity è una scelta popolare per lo sviluppo di applicazioni e giochi in VR e AR. Supporta dispositivi come Oculus Rift, HTC Vive, HoloLens e molti altri, consentendo agli sviluppatori di creare esperienze immersive in queste tecnologie emergenti.

- **Comunità e supporto:** Unity ha una vasta comunità di sviluppatori e offre una serie di risorse, forum di supporto e documentazione. Ciò semplifica il processo di apprendimento e risoluzione dei problemi per chiunque inizi a sviluppare con Unity.

In sintesi, Unity è una piattaforma di sviluppo di videogiochi e applicazioni estremamente completa e accessibile. La sua flessibilità, il supporto multi-piattaforma e l'ampia comunità lo rendono uno strumento di prim'ordine per gli sviluppatori che desiderano creare esperienze interattive di alta qualità su una vasta gamma di dispositivi e piattaforme e hanno fatto sì che venisse scelto per lo sviluppo dell'applicazione.

2.1.2 Interfaccia

L'interfaccia di Unity è composta da vari pannelli, finestre e strumenti che consentono agli sviluppatori di creare, modificare e gestire il loro progetto:

- **Scene:** Il pannello *Scene* è l'area principale in cui gli sviluppatori compongono e modificano le scene del gioco, che possono essere tipicamente più di una. È qui che gli oggetti 3D, le luci, le camere e altri elementi vengono posizionati e organizzati all'interno dell'ambiente virtuale.

- **Game:** Il pannello *Game* consente di visualizzare l'anteprima del gioco o dell'applicazione in esecuzione. Mostra ciò che vedrà l'utente finale e consente agli sviluppatori di testare il gioco in tempo reale.

- **Hierarchy:** La finestra *Hierarchy* elenca tutti gli oggetti presenti nella scena, consentendo agli sviluppatori di organizzare la struttura dei loro progetti. Gli oggetti possono essere organizzati in gerarchie per gestire personaggi o oggetti complessi tramite relazioni padre-figlio.

- **Project:** Il pannello *Project* contiene tutti i file e le risorse presenti nel progetto, come modelli 3D, texture, animazioni e script. Gli sviluppatori possono organizzare e importare risorse, nonché trascinarle direttamente nella scena o nella gerarchia.

- **Inspector:** La finestra *Inspector* mostra le proprietà degli oggetti selezionati nella scena o nella gerarchia in modo che gli sviluppatori possano modificarne le proprietà, come posizione, rotazione, scala e componenti, direttamente da questa finestra.

- **Console:** La finestra *Console* visualizza i messaggi di debug, gli errori e le informazioni relative all'esecuzione del gioco o dell'applicazione. È utile per identificare e risolvere problemi durante lo sviluppo.

Oltre a questi pannelli principali, Unity offre una serie di finestre e strumenti specializzati per lo sviluppo, come i pannelli relativi alle animazioni, al sistema di particelle, all'illuminazione, alla fisica e molto altro. Gli sviluppatori possono personalizzare l'interfaccia utente in base alle proprie esigenze e preferenze.

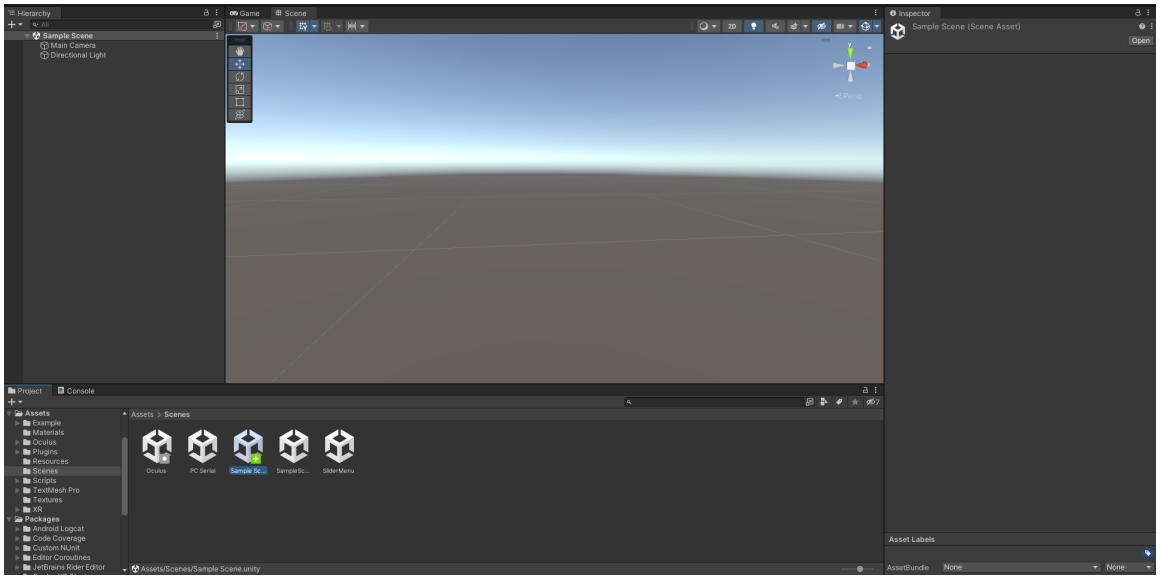


Figura 2.1: Interfaccia Unity di default.

2.1.3 Concetti principali

Oltre al già citato concetto di *Scena*, di cui si compone l'applicazione, è importante definirne altri fondamentali per la comprensione del funzionamento di Unity:

- **GameObject:** I *GameObject* sono gli elementi fondamentali in una scena di Unity. Rappresentano oggetti 3D o elementi interattivi all'interno dell'ambiente virtuale. Un *GameObject* può essere qualsiasi cosa, come un personaggio, un oggetto, una luce o una camera.

- **Component:** I *GameObject* sono costituiti da diversi *Component*, ovvero script o altri elementi specifici che definiscono il comportamento e le proprietà di un oggetto. Questo, ad esempio, potrebbe avere un *Component* di trasformazione per posizione, rotazione e scala, uno di rendering per definire l'aspetto visivo, o uno di animazione per controllarne le animazioni.

- **Resources:** Le *Resources* sono file o asset (come modelli 3D, texture, suoni, script, ecc.) che vengono utilizzati nel progetto. Possono essere caricati in runtime e utilizzati dinamicamente nelle scene.

- **Prefab:** Un *Prefab* è una versione prefabbricata di un *GameObject* che può essere riutilizzata in più scene o clonata in modo da mantenere coerenza e facilitare l'aggiornamento. Modificare un *Prefab* si riflette automaticamente su tutte le sue istanze.

2.2 Oculus Quest 2

Oculus Quest 2 rappresenta un punto di riferimento nell'ambito della realtà virtuale (VR). Si tratta di un visore VR autonomo prodotto da Oculus, una sussidiaria di Meta Platforms, Inc. (precedentemente nota come Facebook).

2.2.1 Caratteristiche generali

Questo dispositivo ha ottenuto una notevole popolarità grazie alle sue caratteristiche avanzate, tra cui possiamo annoverare [7]:

- **Display ad Alta Risoluzione:** Il visore è dotato di un display LCD con una risoluzione complessiva di 3664x1920 pixel, che si traduce in un'immagine estremamente nitida e dettagliata. Questa alta risoluzione è fondamentale per una rappresentazione visiva realistica in ambienti virtuali.
- **Frequenza di Aggiornamento di 90 Hz:** Il display opera a una frequenza di aggiornamento di 90 Hz, garantendo un'esperienza visiva fluida e riducendo la latenza visiva per un'immersione ottimale.
- **Processore Snapdragon XR2:** L'Oculus Quest 2 è alimentato dal processore Snapdragon XR2, che offre prestazioni di alto livello. Questo chip è progettato appositamente per dispositivi di realtà virtuale, consentendo l'esecuzione fluida di applicazioni e giochi VR complessi.
- **Archiviazione Variabile:** Il dispositivo è disponibile in diverse opzioni di archiviazione, tra cui modelli da 64 GB e 256 GB. Gli utenti possono scegliere la capacità di archiviazione in base alle proprie esigenze di spazio per giochi, applicazioni e contenuti.
- **Controller Oculus Touch:** I controller Oculus Touch offrono un'esperienza di tracciamento del movimento precisa e intuitiva. Questi controller consentono agli utenti di interagire con l'ambiente virtuale in modo naturale, grazie ai sensori di posizione e rotazione.
- **Tracking all'Interno-Out:** Il visore utilizza il tracking all'interno-out per monitorare il posizionamento e il movimento dell'utente. Questa tecnologia elimina la necessità di sensori esterni, rendendo l'esperienza più comoda e portatile.

- **Connettività Wireless:** L'Oculus Quest 2 può connettersi al Wi-Fi e offre la possibilità di eseguire applicazioni e giochi in modalità wireless. Questo elimina la necessità di cavi e offre una maggiore libertà di movimento.
- **Ecosistema Sociale VR:** Gli utenti possono connettersi con amici in ambienti virtuali, partecipare a eventi sociali virtuali e ospitare riunioni in VR. Questo aspetto sociale dell'esperienza VR apre nuove possibilità di comunicazione e collaborazione.
- **Peso Leggero e Design Ergonomico:** Il visore è leggero e presenta un design ergonomico che offre comfort durante l'uso prolungato. La fascia per la testa regolabile garantisce una vestibilità comoda.
- **Durata della Batteria:** L'autonomia della batteria varia in base all'uso, ma offre un'esperienza di gioco VR soddisfacente. La batteria è progettata per durare per diverse ore di utilizzo.

Figura 2.2: Oculus Quest 2.

Oculus Quest 2 ha rivoluzionato il mondo della VR rendendo l'accesso a questa tecnologia più accessibile e conveniente per una vasta gamma di utenti. La sua combinazione di prestazioni, facilità d'uso e accessibilità ha reso Oculus Quest 2 uno dei visori VR più popolari e influenti sul mercato.

2.2.2 Setup Unity per Oculus

Al fine di settare correttamente una scena in Unity destinata all'esecuzione su Oculus Quest 2 è necessario scaricare il package *Oculus Integration* [8]. Questo contiene al suo interno una serie di componenti fondamentali.

Il primo passaggio da effettuare è trascinare l'*OvrCameraRig* all'interno della scena. Questo componente sostituisce le normali camere di Unity, che non possono essere utilizzate per un'applicazione da eseguire su Oculus. In particolare, l'*OvrCameraRig* è un empty *GameObject* che agisce come contenitore di tutti i componenti principali collegati alla visione in realtà virtuale. In particolare si ha il componente *TrackingSpace* che rappresenta lo spazio di tracciamento. Tutti gli oggetti figli di *TrackingSpace* si muoveranno in base alle informazioni di tracciamento dei sensori del visore Oculus, mantenendo l'allineamento corretto con il mondo virtuale. Come GameObject figli di *TrackingSpace* si hanno una serie di ancoraggi:

- *CenterEyeAnchor*: rappresenta la posizione e l'orientamento dell'occhio centrale dell'utente. In una configurazione di realtà virtuale, questo punto corrisponde all'occhio dell'utente nel mezzo della testa. Questo GameObject ha associato un componente *Camera*.

- *LeftEyeAnchor* e *RightEyeAnchor*: rappresentano la posizione e l'orientamento degli occhi sinistro e destro dell'utente rispettivamente. Possono essere utilizzati per posizionare oggetti specificamente rispetto a ciascun occhio.

- *TrackerAnchor*: rappresenta la posizione e l'orientamento di eventuali tracker aggiuntivi collegati al sistema Oculus, come i sensori di tracciamento aggiuntivi o altri dispositivi di input.

- *LeftHandAnchor* e *RightHandAnchor*: rappresentano la posizione e l'orientamento delle mani sinistra e destra dell'utente, rispettivamente. Possono essere utilizzati per posizionare oggetti specificamente rispetto a ciascuna mano, come ad esempio per il rendering di modelli di mani virtuali o per l'interazione con oggetti virtuali. Al fine di gestire gli input provenienti dai controller, è necessario aggiungere come figli degli ancoraggi *LeftControllerAnchor* e *RightControllerAnchor* presenti in gerarchia i prefab *OvrControllerPrefab*. Questi prefabbricati sono modelli 3D dei controller Oculus Touch, e includono tutti gli elementi necessari per rappresentare in modo accurato i controller. Ogni *OvrControllerPrefab* include script incorporati che gestiscono l'input e l'interazione con gli oggetti virtuali.

Questi script sono configurati per funzionare con gli Oculus Touch, traducendo i movimenti e gli input fisici dei controller nelle azioni corrispondenti all'interno dell'ambiente virtuale.

Al fine di poter gestire questi input, e di poterli anche agganciare tramite script, è necessario settare nello script *OvrControllerHelper* il parametro *Controller* rispettivamente come L Touch e R Touch per i controller sinistro e destro.

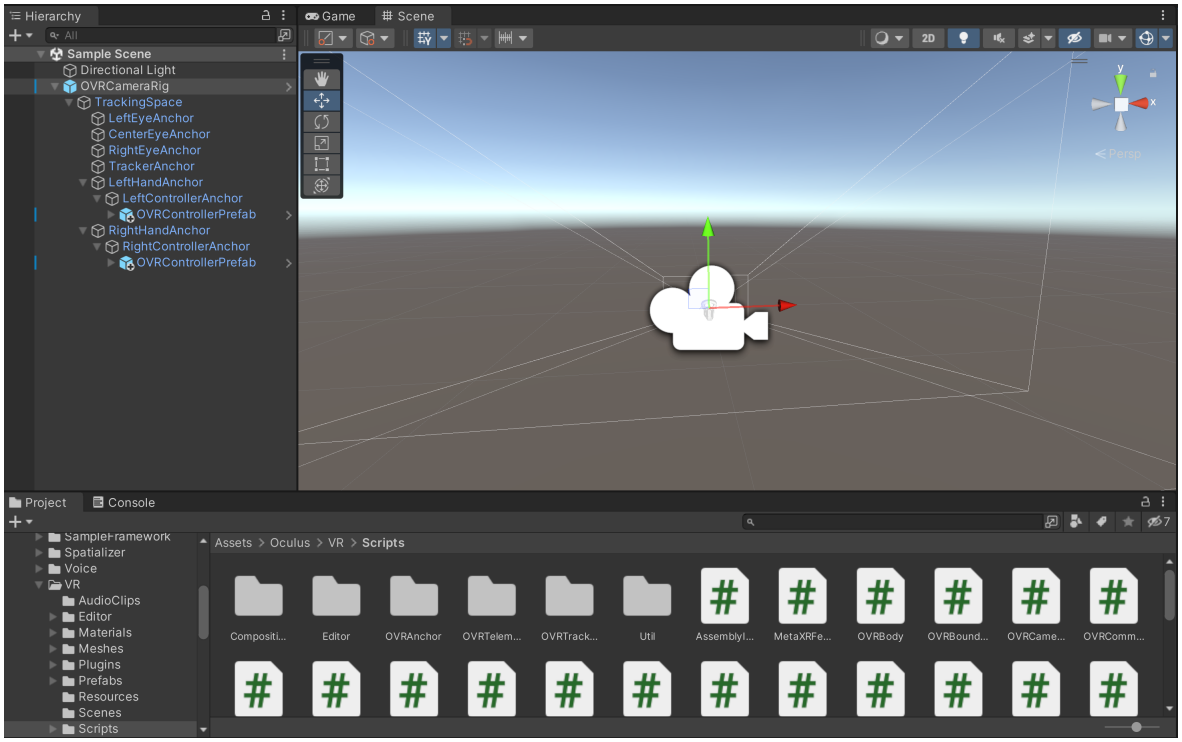


Figura 2.3: Setup OvrCameraRig.

Gli ultimi passaggi da effettuare sono quelli necessari ad abilitare la funzione Oculus Passthrough [9], una caratteristica presente nei visori della serie Oculus Quest che consente agli utenti di visualizzare il mondo reale attraverso le telecamere integrate nei visori. Ciò è particolarmente utile per situazioni in cui l'utente deve interagire con l'ambiente esterno senza rimuovere il visore. Ad esempio, quando si sta percorrendo una stanza o quando si sta cercando di trovare un controller o altri oggetti nel mondo reale.

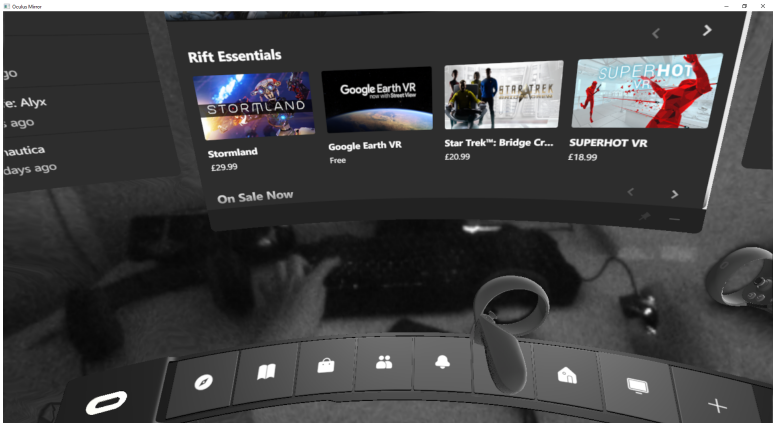


Figura 2.4: Oculus Passthrough.

L'oggetto *OvrCameraRig* ha associati di default tre script che contribuiscono al corretto funzionamento dell'oggetto e alla gestione dell'esperienza di realtà virtuale. Per abilitare il Passthrough è necessario agire sui parametri degli script, in particolare bisogna settare il parametro Passthrough Support come *Required* o *Supported*, inoltre va abilitata la checkbox Enable Passthrough. Entrambi questi parametri si trovano all'interno dello script *OvrManager*.

Infine, va aggiunto all'oggetto il componente *OvrPassthroughLayer*. Questo script contiene diversi parametri per gestire la visualizzazione del layer Passthrough, come il colore e l'ordine dei diversi layer. Al fine di poter visualizzare il mondo reale come sfondo, è necessario agire sul parametro Placement, che deve avere come valore *Underlay*.

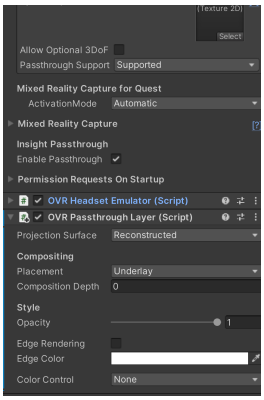


Figura 2.5: Parametri per abilitare Oculus Passthrough.

Capitolo 3

Image tracking

Il primo approccio che si è pensato di avere è stato quello di utilizzare l'image tracking per ottenere i dati necessari alle simulazioni fisiche, e in questo capitolo si andranno a discutere due diversi metodi, in particolare i meccanismi di tracking in Unity, ma anche diversi tipi di immagini target che possono essere sfruttate per il tracciamento.

3.1 Image tracking con ARFoundation

Un metodo molto comune di effettuare image tracking in Unity per sviluppo in Android è quello di sfruttare ARFoundation, un framework multipiattaforma disponibile in Unity tramite un omonimo package.

Il package ARFoundation [10] di Unity fornisce una serie di funzionalità differenti per costruire applicazioni in realtà aumentata, una di queste è l'image tracking. Tuttavia, ARFoundation non implementa funzionalità di per se ma fornisce solo un'interfaccia per lo sviluppo, è quindi necessario usufruire del plug-in specifico per ogni piattaforma. Nel caso di Android, ci si serve di ARCore XR Plugin [11]. Questo plugin consente agli sviluppatori di integrare le funzionalità di ARCore nei loro progetti Unity, permettendo loro di creare applicazioni di realtà aumentata per dispositivi Android.

3.1.1 Scelta delle immagini

Per ottenere un buon tracciamento con ARCore, è importante avere un'immagine chiara e distintiva che possa essere facilmente riconosciuta dall'algoritmo di riconoscimento. I suggerimenti contenuti nella documentazione di ARCore per la selezione dell'imma-

gine [12] forniscono delle indicazioni sulle specifiche che le immagini devono avere per ottenere un buon tracciamento:

- La risoluzione minima dell'immagine deve essere di 300x300 pixel, con l'utilizzo di immagini ad alta risoluzione che non migliora il tracciamento.
- Il formato dell'immagine deve essere PNG o JPEG.
- Siccome le informazioni di colore non vengono utilizzate per tracciare l'immagine, questa può essere anche in scala di grigi.
- Bisogna evitare di utilizzare immagini fortemente compresse, in quanto la compressione influisce sull'estrazione delle funzionalità.
- Non devono essere scelte immagini con un numero elevato di funzionalità geometriche o pochi elementi, così come quelle che contengono pattern ripetuti.

Immagine di esempio 1	Immagine di esempio 2
Punteggio: `0`	Punteggio: `100`
contiene elementi geometrici ripetitivi	una risoluzione sufficiente; contiene molte delle funzionalità uniche

Figura 3.1: Esempio di valutazione della qualità di due immagini per il tracciamento.

Al fine di valutare le scelte fatte l'SDK ARCore mette a disposizione lo strumento *arcoreimg*, che fornisce un punteggio di qualità all'immagine compreso tra 0 e 100, con 75 che rappresenta il punteggio minimo per ottenere un tracciamento di buona qualità. Lo strumento può essere utlizzato da linea di comando oppure tramite un'applicazione desktop [13].

La scelta delle immagini su cui testare il tracciamento è ricaduta su un QR Code, che era l'idea originale, e su un'immagine suggerita direttamente da ARCore, con entrambe che avevano ottenuto il punteggio di qualità massimo di 100.

Figura 3.2: Immagini scelte per testare il tracciamento.

3.1.2 Setup della scena

Per effettuare un setup di base della scena con ARFoundation è necessario rimuovere la camera di default di Unity e aggiungere i GameObject propri del package dedicati alla gestione della sessione AR, in particolare:

- *AR Session*: assume un ruolo fondamentale nella gestione del ciclo di vita delle funzionalità AR, inclusa l'inizializzazione, l'avvio, la pausa e la terminazione. Questa classe consente inoltre di configurare le opzioni di tracciamento, definire le impostazioni di illuminazione ambientale e gestire altri aspetti cruciali dell'esperienza di AR.

- *AR Session Origin*: rappresenta il punto di origine per la scena AR. Tutti gli oggetti AR dovrebbero essere figli di questo oggetto nella gerarchia degli oggetti Unity. *AR Session Origin* fornisce un riferimento globale per la posizione e l'orientamento degli oggetti AR nella scena, contribuendo all'allineamento virtuale tra il mondo fisico e virtuale. Questo contribuisce in modo significativo al tracciamento e al riconoscimento di superfici, facilitando l'ancoraggio degli oggetti virtuali nel mondo reale.

- *AR Camera*: inserita in gerarchia come figlia della *Ar Session Origin* rappresenta la camera fisica o virtuale che visualizza la scena AR. La *AR Camera* è responsabile del tracciamento del movimento della camera e fornisce dati essenziali sullo stato della sessione AR. Questa classe è fondamentale per garantire che la rappresentazione virtuale all'interno dell'applicazione rispecchi fedelmente il mondo reale.

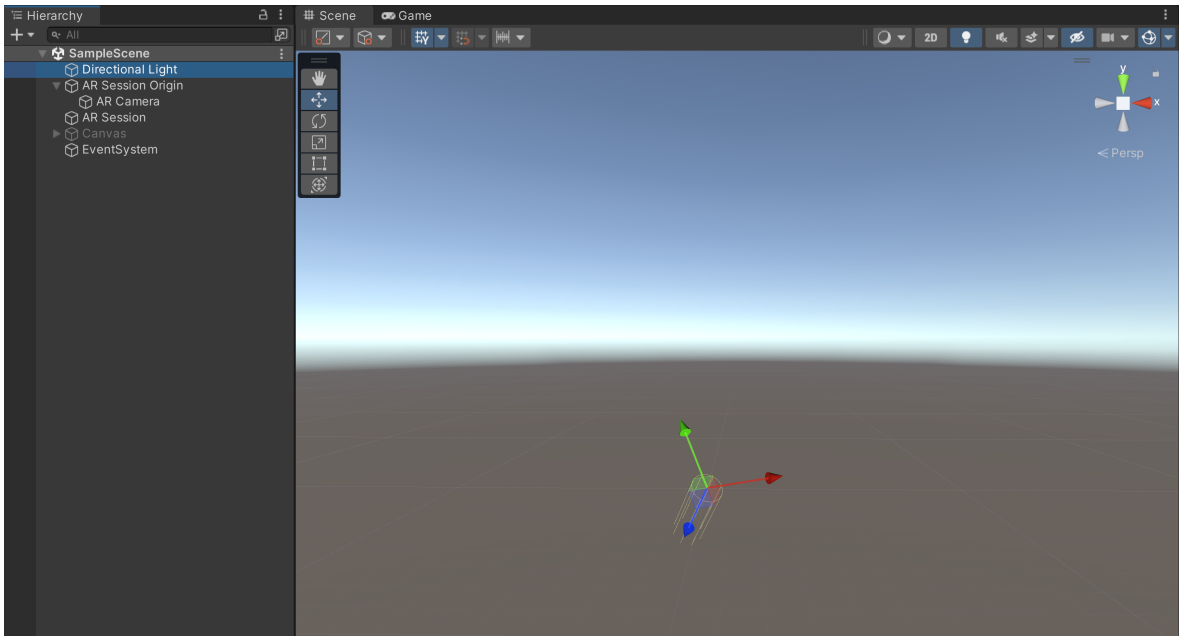


Figura 3.3: Setup della scena con ARFoundation.

3.1.3 Implementazione del tracciamento

Al fine di abilitare il tracciamento delle immagini è innanzitutto necessario creare una *Reference Image Library*. Questo componente consente di specificare un insieme di immagini predefinite che ARCore deve cercare e riconoscere in tempo reale, in questo caso le due immagini scelte in precedenza. Per ognuna immagine bisogna specificare un nome ed in maniera opzionale le dimensioni fisiche in metri, tramite gli opportuni parametri nell'Inspector.

Il passaggio successivo è quello di aggiungere all'oggetto *AR Session Origin* il componente *AR Tracked Image Manager*, uno script a cui è necessario fornire la *Reference Image Library*, il numero massimo di immagini tracciabili contemporaneamente (tramite il parametro Max Number Of Moving Images) e l'oggetto da instanziare sull'immagine target (tramite il parametro Tracked Image Prefab).

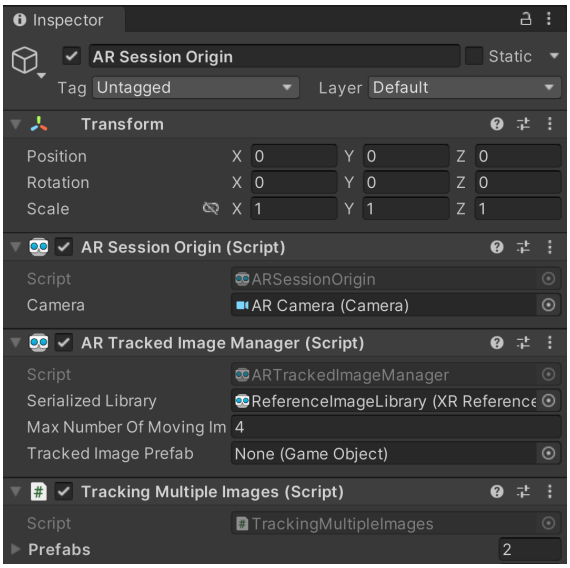


Figura 3.4: AR Session Origin.

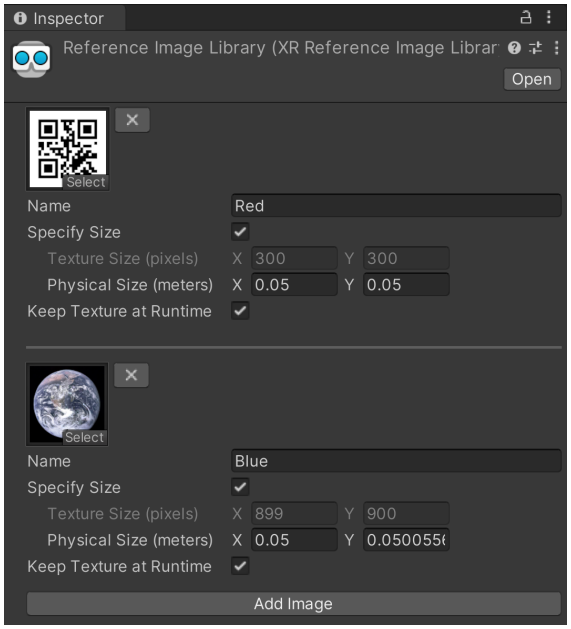


Figura 3.5: Reference Image Library.

Non volendo mostrare lo stesso oggetto su ogni immagine tracciata ma oggetti diversi su ogni immagine si deve creare uno script custom, che è stato denominato *TrackingMultipleImages*, a cui vanno forniti i due prefab da voler istanziare, che devono avere lo stesso nome assegnato nella *Reference Image Library* all'immagine su cui devono apparire. Anche questo script va aggiunto come componente di *AR Session Origin*.

Lo script, all'interno della funzione *Awake*, chiamata durante il caricamento della sua istanza, procede a istanziare tramite la funzione *Istantiate* i prefab forniti dall'utente, passando come parametri il GameObject, il vettore zero e la rotazione identità. Gli oggetti istanziati vengono poi disabilitati tramite la funzione *SetActive* passando come parametro *false* e aggiunti a un dizionario stringa-GameObject denominato *instanPrefabs* in cui la stringa è il nome dell'oggetto e il GameObject l'oggetto stesso. Inoltre si va a trovare il componente *ARTrackedImageManager* tramite la funzione *FindObjectOfType<ARTrackedImageManager>()*.

```
[SerializeField] private GameObject[] prefabs;
private Dictionary<string, GameObject> instanPrefabs = new Dictionary<string, GameObject>();
private ARTrackedImageManager trackedImageManager;

Messaggio Unity | 0 riferimenti
private void Awake()
{
    trackedImageManager = FindObjectOfType<ARTrackedImageManager>();

    foreach (GameObject go in prefabs)
    {
        GameObject newGo = Instantiate(go, Vector3.zero, Quaternion.identity);
        newGo.SetActive(false);
        instanPrefabs.Add(newGo.name, newGo);
    }

}
```

Figura 3.6: Funzione Awake e dichiarazione variabili.

Per poter rilevare i cambiamenti bisogna nella funzione *OnEnable* aggiungere all' *ARTrackedImageManager* una funzione *OnChanged* che venga chiamata quando avviene un cambiamento tramite il comando *trackedImageManager.trackedImagesChanged += OnChanged*. La funzione *OnChanged* riceve come parametro un *ARTrackedImagesChangedEventArgs* che contiene tutte le immagini aggiunte, aggiornate o rimosse. Per ognuna delle immagini che ricade nei primi due casi viene chiamata la funzione *HandleImage*, che riceve come parametro l'immagine stessa, mentre per le immagini rimosse si disattiva l'oggetto con il nome corrispondente, sempre tramite la funzione *SetActive(false)*.

Nella funzione *HandleImage* si vanno prima di tutto a ottenere il nome, la posizione e la rotazione dell'immagine, dopodichè si cerca nel dizionario *instanPrefabs* l'oggetto con il nome corrispondente e gli si assegnano la stessa posizione e rotazione dell'immagine. Infine, si va ad effettuare il controllo sullo stato dell'immagine, che se è *Tracking* indica come questa sia effettivamente tracciata. In tal caso l'oggetto può essere abilitato, altrimenti deve essere disabilitato.

```
Messaggio Unity | 0 riferimenti
private void OnEnable()
{
    trackedImageManager.trackedImagesChanged += OnChanged;
}

Messaggio Unity | 0 riferimenti
private void OnDisable()
{
    trackedImageManager.trackedImagesChanged -= OnChanged;
}

2 riferimenti
private void OnChanged(ARTrackedImagesChangedEventArgs eventArgs)
{
    foreach(ARTrackedImage image in eventArgs.added)
    {
        HandleImage(image);
    }

    foreach (ARTrackedImage image in eventArgs.updated)
    {
        HandleImage(image);
    }

    foreach (ARTrackedImage image in eventArgs.removed)
    {
        instanPrefabs[image.name].SetActive(false);
    }
}

2 riferimenti
private void HandleImage(ARTrackedImage image)
{
    string name = image.referenceImage.name;
    Vector3 position = image.transform.position;
    Quaternion rotation = image.transform.rotation;

    GameObject prefab = instanPrefabs[name];
    prefab.transform.position = position;
    prefab.transform.rotation = rotation;
    if (image.trackingState == UnityEngine.XR.ARSubsystems.TrackingState.Tracking)

        prefab.SetActive(true);
    else
        prefab.SetActive(false);
}
```

Figura 3.7: Funzioni OnChanged e HandleImage.

In questo modo è possibile tracciare le immagini e mostrare gli oggetti su di esse sovrapposti. Tuttavia il tracciamento con l'immagine in movimento non risulta abbastanza reattivo, l'oggetto infatti in caso di movimenti veloci scompare per poi riapparire dopo un tempo non compatibile con un tracciamento finalizzato allo sviluppo di un esperimento. Ciò ha spinto alla ricerca di nuove soluzioni che potessero fornire risultati maggiormente soddisfacenti.

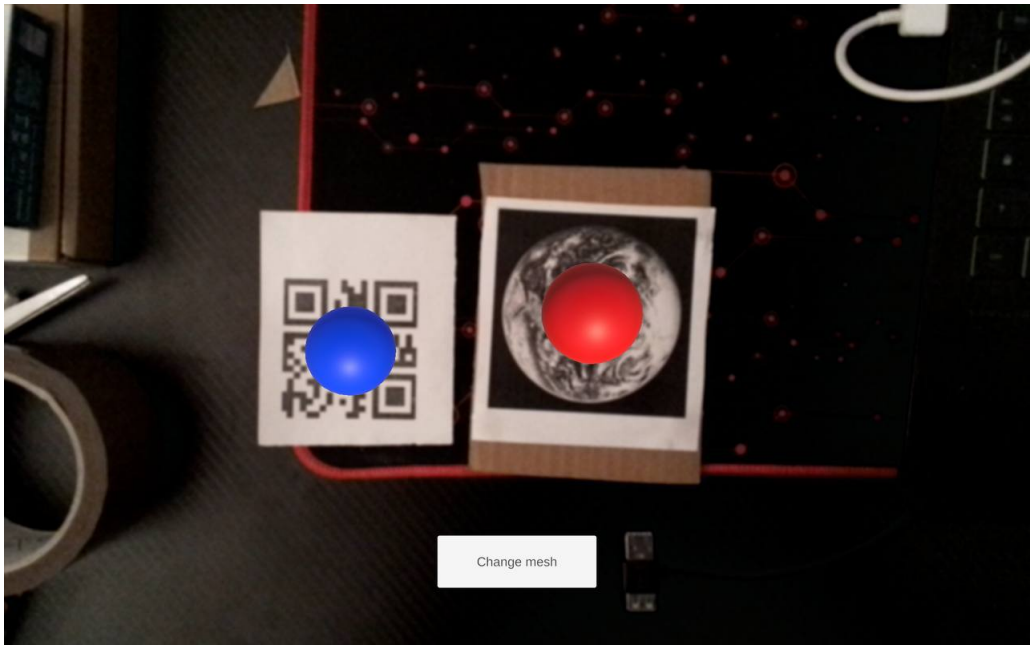


Figura 3.8: Risultato del tracciamento con ARFoundation.

3.2 Image tracking con AprilTag

Per via dei problemi discussi nel paragrafo precedente riguardo all'accuratezza del tracking di ARFoundation, si è cercata una soluzione alternativa. Si è quindi passati dall'utilizzo di immagini target generiche a quello di tag specifici ottimizzati per il tracciamento, e la scelta è ricaduta su AprilTag.

3.2.1 AprilTag

AprilTag [14] [15] è una tecnologia di riconoscimento visivo che ha rivoluzionato il campo della computer vision e dell'augmented reality (AR). Questo sistema, sviluppato dal laboratorio di robotica del Georgia Institute of Technology, offre un metodo altamente efficace per identificare e localizzare oggetti fisici tramite pattern visivi unici, noti come "tag" o "marcatori". La sua applicazione si estende a una vasta gamma di settori, dall'automazione industriale all'arte interattiva, dalla navigazione robotica all'educazione.

AprilTag si distingue per la sua capacità di riconoscere e seguire oggetti in tempo reale con estrema precisione e velocità. Questa tecnologia si basa su un design di marcatori ottimizzato, composto da una serie di quadrati neri e bianchi disposti in

uno schema specifico. Quando una fotocamera o un dispositivo di visione cattura un'immagine di un tag AprilTag, il software di riconoscimento è in grado di identificare il tag, determinarne la posizione e l'orientamento nell'ambiente e fornire un punto di riferimento per l'interazione digitale.

Questi marcatori sono stati progettati per essere rilevati e interpretati con estrema precisione da algoritmi di riconoscimento visivo, consentendo l'interazione tra il mondo fisico e il mondo virtuale. I componenti principali di un marcatore AprilTag sono:

- **Matrice di Quadrati Neri e Bianchi:** Il nucleo di un marcatore AprilTag è una matrice di quadrati neri e bianchi disposti in uno schema specifico. Questa matrice è l'elemento distintivo del marcatore ed è ciò che consente al sistema di identificarlo univocamente. I quadrati possono essere disposti in una griglia, di solito quadrata, e possono variare in dimensioni e densità a seconda del tipo di marcatore.

- **Bordo Bianco:** Spesso, un marcatore AprilTag presenta un bordo bianco intorno alla matrice di quadrati neri e bianchi. Questo bordo facilita la rilevazione del marcatore, poiché fornisce un punto di riferimento chiaro per il software di riconoscimento visivo.

- **Codice Binario:** All'interno della matrice di quadrati, sono inclusi una serie di quadrati neri che costituiscono un codice binario. Questo codice è una sequenza di 0 e 1 disposti in un modello specifico. Il codice binario è il vero identificatore del marcatore, poiché ogni sequenza di bit è associata a un numero univoco che rappresenta il marcatore.

- **Zona di Silenzio:** Tra il codice binario e il bordo del marcatore, c'è una zona di silenzio, generalmente costituita da quadrati bianchi. Questa zona aiuta il software a identificare i confini del marcatore e a evitare confusione con altri marcatori o elementi dell'ambiente.

- **Angoli di Rilevamento:** Gli angoli del marcatore AprilTag sono particolarmente importanti per determinarne la posizione e l'orientamento. Solitamente, questi angoli sono costituiti da quadrati neri e bianchi disposti in una configurazione specifica, che può variare a seconda del tipo di marcatore.

- **Centro del Marcatore:** Il centro del marcatore è il punto di riferimento da cui vengono calcolate le coordinate del marcatore nell'ambiente. È generalmente al centro della matrice di quadrati neri e bianchi.

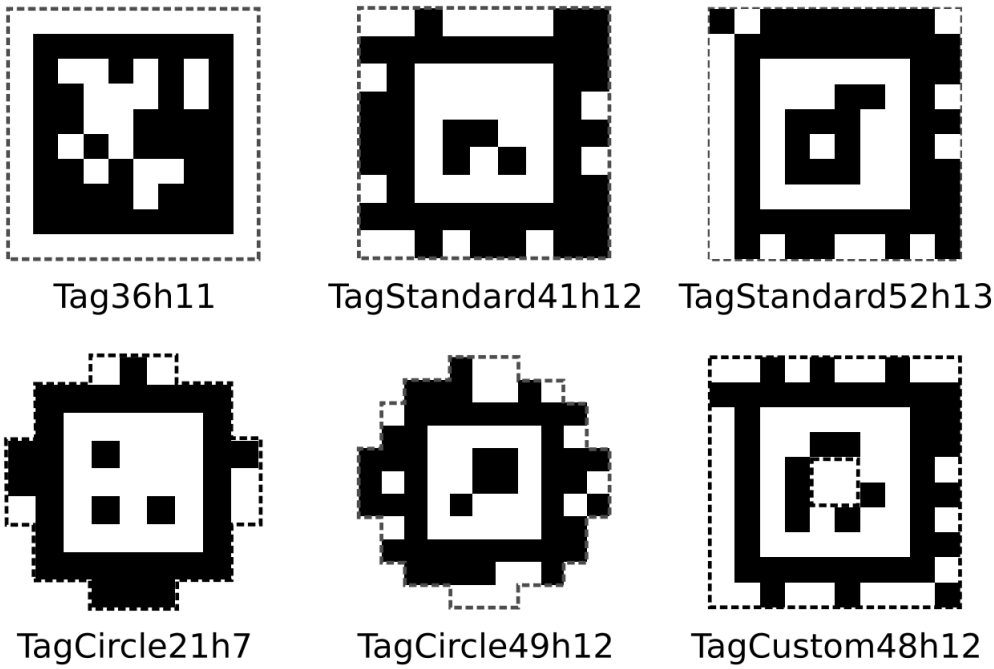


Figura 3.9: Diverse famiglie di AprilTag.

Esistono diverse famiglie di marcatori AprilTag e all'interno di esse ognuno possiede un suo id numerico unico identificativo. La scelta è ricaduta sulla *TagStandard41h12*, essendo quella standard raccomandata. Per testare il tracciamento è stato utilizzato il tag in Figura 3.9, identificato dall'id 0.

3.2.2 Setup della scena

Per localizzare l'immagine target nella scena si è deciso di sfruttare un package custom per Unity, AprilTag package for Unity, che implementa un detector di AprilTag [16]. In questo caso, non essendo il meccanismo di detection integrato in Unity, non è possibile sfruttare i componenti del setup precedente, dunque nella gerarchia della scena sarà necessario inserire:

- *Main Camera*: la camera principale di Unity, non essendo possibile utilizzare una AR Camera, in questo caso sarà questo oggetto a inquadrare la scena.
- *Webcam Input*: un empty GameObject a cui è stato aggiunto il componente *Image Source*, uno script fornito dal pacchetto che è necessario per ottenere l'immagine dalla camera del dispositivo su cui si andrà ad eseguire l'applicazione. Lo script

consente di settare alcuni parametri, in questo caso Source Type su *Webcam*, Resolution su *1920x1080* e Frame Rate su *60*.

- *Canvas*: un contenitore per diversi elementi di UI, il cui parametro Render Mode va predisposto a *Screen Space - Camera*, una modalità utilizzata quando si desidera che la UI sia resa in relazione a una specifica camera nella scena, in questo caso la Main Camera, che va quindi selezionata nel parametro Render Camera. L'oggetto figlio del *Canvas* è *Webcam*, che contiene un componente Raw Image a cui potrà essere impostata nel parametro Texture l'immagine ricavata dalla webcam in modo da poterla visualizzare a schermo.

- *Detection Test*: un oggetto vuoto in cui è stato aggiunto il componente DetectionTest, uno script che gestisce il tracciamento del tag e la visualizzazione dell'oggetto su di esso, e che dei cui dettagli si andrà a discutere di seguito.

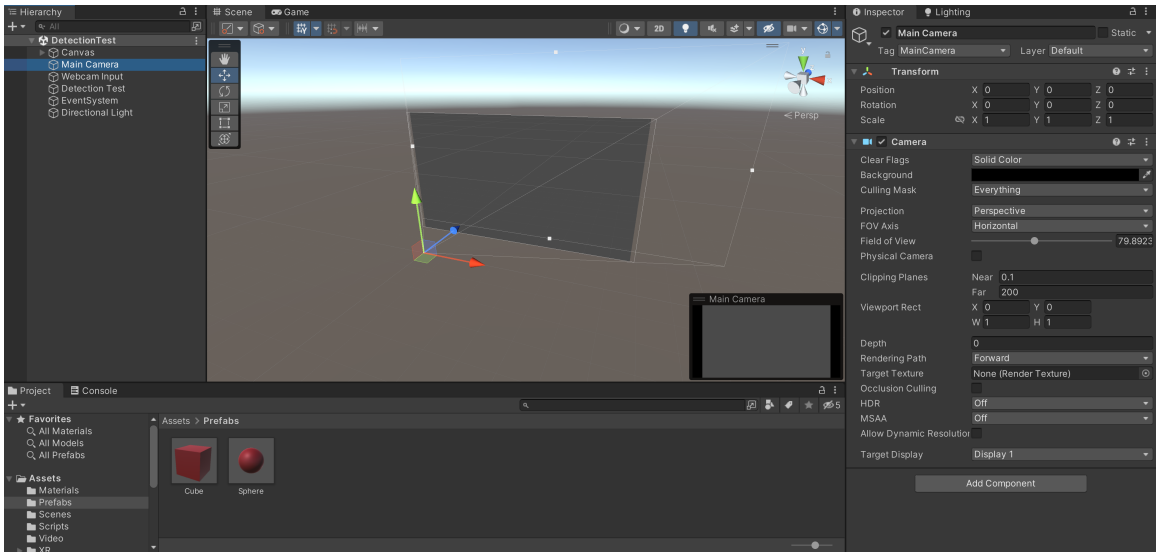


Figura 3.10: Setup della scena per tracciamento AprilTag.

3.2.3 Implementazione del tracciamento

Lo script *DetectionTest* contiene dei parametri da assegnare tramite l'interfaccia di Unity. In particolare:

- GameObject *go*: l'oggetto da mostrare sul tag tracciato, a cui è stato assegnato il prefab di una sfera rossa.

- ImageSource *_source*: il frame su cui effettuare la detection, a cui è stato assegnato il componente Image Source contenuto nell'oggetto *Webcam Input*.
- RawImage *_webcamPreview*, il componente sulla cui texture andare a proiettare il frame ricavato dalla webcam, a cui è stato assegnato il componenete Raw Image dell'oggetto *Webcam*.
- L'intero *_decimation* che, al suo aumentare, passato al detector riduce la risoluzione del frame velocizzando il tracciamento a scapito della qualità. Il valore di questo parametro è stato impostato su *4*.
- Il float *_tagSize* che rappresenta la dimensione fisica in metri del tag da tracciare, impostata su 0.05.

Nella funzione *Awake* si provvede a istanziare l'oggetto *go* con posizione nell'origine e rotazione nulla, tramite il comando *newgo = Instantiate(go, Vector3.zero, Quaternion.identity)*, per poi disabilitarlo. Nella funzione *Start* invece si inizializza il TagDetector *_detector* tramite il costruttore presente nel pacchetto AprilTag, a cui si passano come parametri le dimensioni del frame e il parametro *decimation*. Nella stessa funzione si procede ad assegnare al componente Texture di *_webcamPreview* il frame ricevuto da *_source*, in modo da proiettarlo sulla UI. Nella funzione *OnDestroy*, chiamata al momento della distruzione del componente, si procede a eliminare il detector tramite la funzione *_detector.Dispose()*.

```
[SerializeField] GameObject go;
[SerializeField] ImageSource _source;
[SerializeField] int _decimation;
[SerializeField] float _tagSize;
[SerializeField] UI.RawImage _webcamPreview;
private GameObject newgo;
private Texture texture;
private RenderTexture renderTexture;
AprilTag.TagDetector _detector;

void Awake()
{
    newgo = Instantiate(go, Vector3.zero, Quaternion.identity);
    newgo.SetActive(false);
}

void Start()
{
    var dims = _source.OutputResolution;
    _detector = new AprilTag.TagDetector(dims.x, dims.y, _decimation);
    _webcamPreview.texture = _source.Texture;
}

void OnDestroy()
{
    _detector.Dispose();
}
```

Figura 3.11: Dichiarazione variabili e funzioni Start, Awake e OnDispose.

Nella funzione *Update*, chiamata una volta per ogni frame, si procede con l'acquisizione della Texture da *_source* in modo da ottenere il frame e del *field of view* della Main Camera. Quest'ultimo viene convertito in radianti mentre il primo in un *ReadOnlySpan*, prima di poter essere entrambi passati insieme a *_tagSize* alla funzione *_detector.ProcessImage(image, fov, _tagSize)* che effettua la detection del tag nell'immagine.

A questo punto i tag rilevati nell'immagine possono essere recuperati nell'array *_detector._detectedTags*. Per ognuno si controlla se il suo ID è uguale a 0, ovvero quello del tag cercato, e in tal caso si procede con l'attivazione dell'oggetto e ad assegnargli posizione e rotazione del tag. In caso contrario invece si disabilita il GameObject, e lo stesso avviene se l'array di tag rilevati è vuoto, in questo modo si evita che l'oggetto rimanga attivo anche quando il marcatore non è più rilevato.

```
void Update()
{
    // Source image acquisition
    var image = _source.Texture.AsSpan();
    if (image.IsEmpty) return;

    // AprilTag detection
    var fov = Camera.main.fieldOfView * Mathf.Deg2Rad;
    _detector.ProcessImage(image, fov, _tagSize);

    // Detected tag visualization
    foreach (var tag in _detector._detectedTags)
    {
        if(tag.ID == 0)
        {
            newgo.SetActive(true);
            newgo.transform.position = tag.Position;
            newgo.transform.rotation = tag.Rotation;
        }
        else
        {
            newgo.SetActive(false);
        }
    }
    if (_detector._detectedTags.Count == 0)
        newgo.SetActive(false);
}
```

Figura 3.12: Funzione Update.

In questo caso il risultato del tracciamento risulta essere di qualità superiore e più reattivo ai movimenti rispetto a quello effettuato tramite ARFoundation. Tuttavia, l'impossibilità attuale di accedere direttamente al frame buffer di Oculus permette di implementare questa soluzione solo su altri dispositivi come tablet e cellulari, e ha spinto a cercare nuove soluzioni per effettuare il tracciamento sul visore.

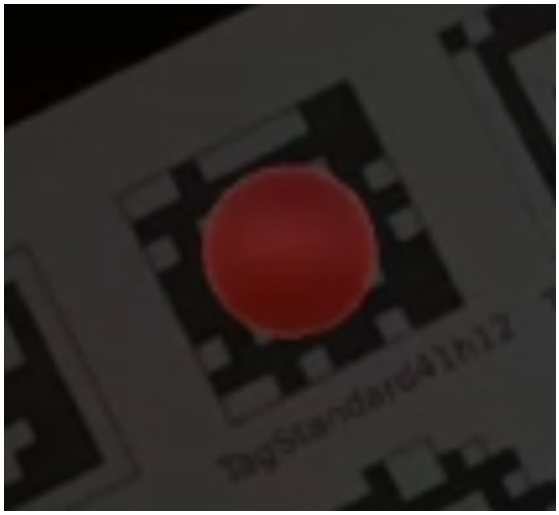


Figura 3.13: Tracciamento tramite AprilTag.

Capitolo 4

Sensori

L'impossibilità di utilizzare l'image tracking su Oculus ha spinto a cercare nuove soluzioni, in particolare diversi sensori attraverso cui tracciare il movimento. In questo capitolo si andranno a descrivere le caratteristiche dei sensori scelti per le misurazioni, oltre alle librerie ed il codice utilizzati per ricavare i dati. Si andrà inoltre a discutere della calibrazioni dei sensori stessi e infine del collegamento tra di essi e il visore per trasmettere i dati ricavati, effettuato tramite Bluetooth Low Energy.

4.1 Arduino Nano 33 BLE Sense Rev2

Il primo modulo scelto è Arduino Nano 33 BLE Sense Rev2 [17], si tratta di una scheda compatta e altamente versatile, che combina le funzionalità di un microcontrollore basato su ARM Cortex-M4 con connettività Bluetooth Low Energy (BLE) e una vasta gamma di sensori integrati. La sua versatilità lo rende un'opzione ideale per una vasta gamma di applicazioni, dall'elaborazione dati ambientali all'interazione con dispositivi mobili, nonché per progetti legati alla sicurezza e al monitoraggio sanitario.

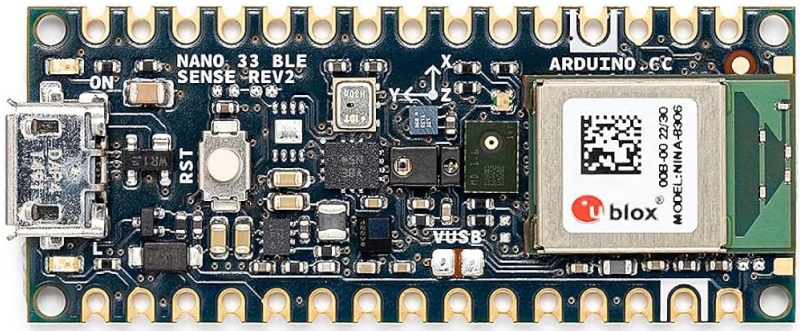


Figura 4.1: Arduino Nano 33 BLE Sense Rev2.

4.1.1 Specifiche della scheda

Sulla scheda è installato un processore Arm Cortex-M4F che lavora a 64 MHz e questa possiede 1 MB di memoria flash per il caricamento del programma e 256 KB di RAM per l'esecuzione delle applicazioni, supporta inoltre la connettività Bluetooth Low Energy 5.0, rendendola adatta a progetti IoT e comunicazione wireless.

Arduino Nano 33 BLE Sense Rev2 è inoltre dotato dei seguenti sensori: [18]

- **BMI 270**: composto da un accelerometro a 3 assi (con range ±2g/±4g/±8g/±16 g) e da un giroscopio a 3 assi (con range ±125dps/±250dps/±500dps/±1000dps/ ±2000dps) che permettono di misurare rispettivamente l'accelerazione lineare e angolare, fornendo informazioni sulla velocità di movimento e l'orientamento della scheda.

- **BMM150**: un sensore geomagnetico a 3 assi che consente la misurazione del campo magnetico terrestre. Questo sensore possiede una risoluzione di 0.3μT, un range di ±1300μT sugli assi x e y e di ±2500μT sull'asse z.

- **LPS22HB**: combina un sensore piezoresistivo di pressione atmosferica con un range di 260-1260 hPa e precisione a 24 bit, e un sensore di temperatura con precisione a 16 bit. Si interfaccia con il microcontrollore mediante un'interfaccia I2C.

- **HS3003**: combina un sensore di umidità con un range di 0-100% e un'accuratezza di ±1.5%RH, e un sensore di temperatura ambiente con un'accuratezza di ±0.1°C. Entrambi i sensori hanno un output fino a 14 bit.

- **APDS-9960**: combina rilevatori di gesti e di prossimità con sensori di luce ambientale e RGB.

- **MP34DT06JTR**: un microfono capacitivo omnidirezionale MEMS con AOP pari a 122.5 dbSPL, 64 dB di rapporto segnale-rumore e sensibilità di –26 dBFS ± 3 dB. Questo sensore può essere utilizzato per applicazioni di rilevamento del suono o per implementare funzionalità di riconoscimento vocale.

La scheda inoltre ha un'interfaccia USB per la programmazione e la comunicazione seriale con il computer. Questa può essere utilizzata anche per l'alimentazione, che può arrivare altrimenti attraverso i pin VIN o VUSB.

4.1.2 Arduino_BMI270_BMM150

Ai fini dello sviluppo dell'applicazione la necessità è quella di ottenere dati da cui poter ricavare la posizione della scheda, dunque i sensori da sfruttare sono il BMI270 e il BMM150. La libreria Arduino_BMI270_BMM150 [19] permette di leggere i valori dell'accelerometro, dal giroscopio e dal magnetometro delle rispettive IMU. Al fine di poter utilizzare questa libreria in uno sketch Arduino è necessario includere all'inizio il comando *#include "Arduino_BMI270_BMM150.h"*.

```
void setup() {
  if (!IMU.begin()) {
    //Serial.println("Failed to initialize IMU!");
    while (1);
  }
```

Figura 4.2: Inizializzazione dell'IMU.

Il passaggio successivo è quello di inizializzare l'IMU attraverso il metodo *begin()*. Questo metodo si occupa dell'inizializzazione e imposta i seguenti valori:

- *Range dell'accelerometro* a ±4 g con risoluzione 0.122 mg.
- *Range del giroscopio* a ±2000 dps con risoluzione 70 mdps.
- *Range del magnetometro* a ±1300 μT con risoluzione 0.3 μT.
- *Frequenza di output dell'accelerometro e del giroscopio* a 99.84 Hz.
- *Frequenza di output del magnetometro* a 10 Hz.

Il metodo va chiamato nella funzione *setup()* chiamata quando parte lo sketch all'interno di una condizione *if* con all'interno *while(1)*. In questo modo, ritornando il metodo 1 in caso di successo e 0 in caso di fallimento, si evita di far avanzare lo sketch se l'inizializzazione fallisce.

La libreria fornisce funzioni per leggere i valori da tutti e tre i sensori dell'IMU. In particolare, le funzioni *accelerationAvailable()*, *gyroscopeAvailable()* e *magneticFieldAvailable()* ritornano 1 se è disponibile un nuovo valore rispettivamente per accelerometro, giroscopio e magnetometro, mentre le funzioni *readAcceleration(x,y,z)*, *readGyroscope(x,y,z)*, *readMagneticField(x,y,z)* permettono di leggere i valori e salvarli nelle

tre variabili float passate come parametri. Ai fini dell'esperimento su cui è costruita l'applicazione, descritto nel Capitolo 5, è sufficiente ricavare l'accelerazione sull'asse x dell'accelerometro, trattandosi di moti rettilinei. Per fare ciò all'interno della funzione *loop()*, chiamata ripetutamente, è necessario inizializzare tre variabili float *ax*, *ay*, *az*, per poi chiamare la funzione per la lettura dell'accelerazione se questa è disponibile.

```
// Read accelerometer data
float ax, ay, az;
if(IMU.accelerationAvailable()){
  IMU.readAcceleration(ax, ay, az);
  // Apply calibration offsets
  ax -= acc_offsets[0];
  ay -= acc_offsets[1];
  az -= acc_offsets[2];
}
```

Figura 4.3: Lettura dei valori di accelerazione.

Nel caso invece in cui si voglia ottenere dati sulla posizione nello spazio 3D e sull'orientamento, come potrebbe essere in sviluppi futuri dell'applicazione, sarebbe necessario ottenere i dati da tutti e tre i sensori e procedere poi alla fusione di essi tramite algoritmi di sensor fusion, come quelli descritti nel Capitolo 6.

4.1.3 Calibrazione dell'accelerometro

Come si può vedere nella Figura 4.3 sono stati applicati degli offset di calibrazione ai valori di accelerazione per compensare l'errore medio del sensore. Questi valori sono stati ottenuti mediante un altro sketch dedicato alla calibrazione. Si deve in esso inizializzare le tre variabli necessarie alla lettura dei valori, così come un vettore di lunghezza 3 al fine di calcolare la somma di tali valori, inoltre deve essere presente una variabile *calCount* inizializzata a 0 per contare quanti valori sono stati sommati e una variabile *calibrationLength*, che indica quanti valori sommare prima di effettuare il calcolo dell'errore medio, impostata su 1024. Bisogna poi nella funzione *setup()* inizializzare l'IMU e la comunicazione seriale necessaria a stampare a monitor i valori di calibrazione.

```
#include <Arduino_BMI270_BMM150.h>

float ax, ay, az;

int calibrationLength = 1024;
int calCount = 0;
float sumAccel[3];

void setup() {

  Serial.begin(9600);

  Serial.println("Started");

  if (!IMU.begin()) {
    Serial.println("Failed to initialize IMU!");
    while (1);
  }

}
```

Figura 4.4: Inizializzazione e setup della calibrazione.

In seguito, nella funzione *loop()* è necessario leggere i valori di accelerazione e ad ogni lettura aggiornare la somma di ognuno dei tre così come il contatore *calCount*, che va incrementato. Quando la condizione $calCount >= calibrationLength$ è vera, e dunque sono stati letti 1024 valori di accelerazione, si puo procedere alla stampa dei valori dell'errore mediante il comando *Serial.print()* e al reset delle somme e del contatore.

```
void loop() {

  if (IMU.accelerationAvailable()) {
    IMU.readAcceleration(ax, ay, az);

    sumAccel[1] += ax;
    sumAccel[2] += ay;
    sumAccel[3] += (az-1);

    calCount += 1;
  }

  if (calCount >= calibrationLength) {
    Serial.println("Accelerometer Mean Error:");
    Serial.print("axE: ");
    Serial.print(sumAccel[1] / calibrationLength);
    Serial.print(" ayE: ");
    Serial.print(sumAccel[2] / calibrationLength);
    Serial.print(" azE: ");
    Serial.println(sumAccel[3] / calibrationLength);
    Serial.println();


    sumAccel[1] =0;
    sumAccel[2] =0;
    sumAccel[3] =0;

    calCount = 0;
  }
}
```

Figura 4.5: Loop di calibrazione.

Al fine di ottenere valori corretti bisogna eseguire lo sketch tenendo l'asse da calibrare parallelo al terreno, in modo che il valore atteso sia 0 e che quindi la media dei valori letti equivalga all'errore. Gli offset di calibrazione ricavati e inseriti nel vettore *acc_offsets* in Figura 4.3 sono 0.0143 per l'asse x, -0.0092 per l'asse y e -0.0016 per l'asse z.

4.2 CQRobot VL53L1X

Il secondo sensore che è stato aggiunto è il CQRobot VL53L1X, che rappresenta una componente chiave nell'ambito delle applicazioni di rilevamento a distanza. Questo dispositivo, basato sulla tecnologia Time-of-Flight (ToF), è stato sviluppato per misurare la distanza tra il sensore e un oggetto con precisione millimetrica, rendendolo ideale per una vasta gamma di applicazioni, dalla navigazione di droni al monitoraggio di presenze in ambienti industriali o di consumo. Il VL53L1X offre una soluzione altamente affidabile e precisa per il rilevamento della distanza, sfruttando impulsi laser infrarossi ad alta frequenza per misurare il tempo impiegato dalla luce a tornare dall'oggetto rilevato.

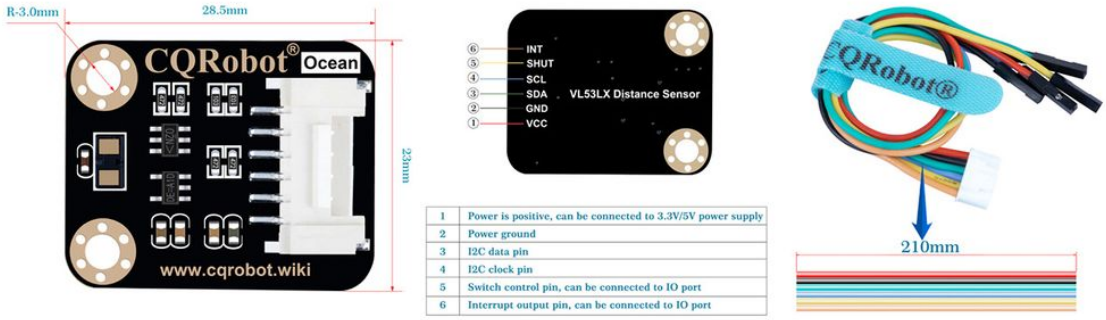


Figura 4.6: CQRobot VL53L1X.

4.2.1 Specifiche del sensore

Il sensore CQRobot VL53L1X [20] è un modulo Time-of-Flight (ToF) altamente avanzato progettato per misurare la distanza con precisione. Le sue specifiche includono:

- **Raggio operativo:** Il VL53L1X è in grado di misurare distanze da 40 millimetri fino a 4 metri, consentendo una notevole flessibilità nell'applicazione in una varietà di contesti.
- **Precisione:** Il sensore offre una precisione del ±5%, rendendolo ideale per applicazioni che richiedono rilevamenti dettagliati delle distanze.
- **Velocità di campionamento:** Il VL53L1X può effettuare misurazioni con una frequenza fino a 50 Hz in modalità Short Distance e fino a 30 Hz in modalità Medium e Long Distance, consentendo il rilevamento di oggetti in rapido movimento.
- **Interfaccia:** Il sensore è dotato di un'interfaccia I2C per una facile integrazione con microcontrollori e sistemi embedded.
- **Ampia gamma di applicazioni:** Grazie alla sua capacità di misurare distanze con precisione e velocità, il sensore VL53L1X è adatto per applicazioni come il monitoraggio della presenza, la navigazione di droni, il rilevamento di oggetti in movimento e il controllo di dispositivi autonomi.
- **Affidabilità:** Il sensore è progettato per funzionare in vari ambienti e condizioni di illuminazione, rendendolo affidabile in una vasta gamma di scenari.

Le specifiche avanzate del sensore CQRobot VL53L1X lo rendono una scelta eccellente per progetti come questo che richiedono misurazioni precise della distanza in tempo reale. Inoltre, il sensore può essere alimentato a 3,3 V o 5 V, e questo permette di connetterlo direttamente alla scheda Arduino, seguendo uno schema di connessioni illustrato nella Figura 4.7.

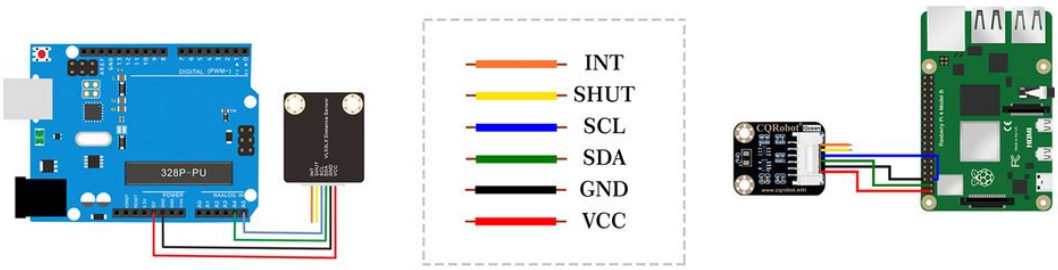


Figura 4.7: A sinistra, schema di collegamento del sensore con Arduino.

4.2.2 Arduino VL53L1X

Una volta connesso il sensore alla scheda arduino, si possono ottenere i dati tramite la libreria Arduino VL53L1X [21], dopo aver stabilito la connessione mediante I2C utilizzando la libreria Arduino Wire [22]. Entrambe le librerie vanno aggiunte allo sketch tramite i comandi *#include <VL53L1X.h>* e *#include <Wire.h>*.

Bisogna definire una variabile float *tof* per salvare il valore letto e una variabile VL53L1X *sensor* che rappresenta un'istanza del sensore. Poi nella funzione *setup()* si deve inizializzare la comunicazione tramite il metodo *Wire.begin()*, settare il clock con la rispettiva funzione a 400 kHz e inizializzare il sensore tramite il metodo *sensor.init()*, per poi andare a impostare i seguenti parametri:

- *Periodo di timeout* a 500 ms mediante la funzione *sensor.setTimeout(500)*.
- *Modalità di funzionamento* a Long Distance con la funzione *sensor.setDistanceMode(VL53L1X::Long)*.
- *Timing Budget*, ovvero il tempo allocato per la misura, a 33 ms tramite la funzione *sensor.setMeasurementTimingBudget(33000)*.

Infine, si deve far partire la lettura continua dei valori tramite il metodo *sensor.startContinuous(33)*, il cui parametro è il tempo che intercorre tra due misure, in questo caso 33 ms.

```
Wire.begin();
Wire.setClock(400000); // use 400 kHz I2C

sensor.setTimeout(500);
if (!sensor.init())
{
  //Serial.println("Failed to detect and initialize sensor!");
  while (1);
}
sensor.setDistanceMode(VL53L1X::Long);
sensor.setMeasurementTimingBudget(33000);
sensor.startContinuous(33);
```

Figura 4.8: Funzione setup.

Nella funzione *loop()* poi è sufficiente controllare che il dato sia pronto tramite il metodo *sensor.dataReady()* e in caso assegnare alla variabile tof il valore con il comando *sensor.read()*.

```
if(sensor.dataReady()){
  tof = sensor.read();
}
```

Figura 4.9: Funzione loop.

4.3 Connessione tramite Bluetooth Low Energy

Il problema successivo, una volta ottenuti i dati dai sensori, è quello di trasmetterli al visore ai fini della simulazione. La trasmissione avviene tramite Bluetooth Low Energy, di cui si andranno a descrivere alcuni concetti chiave:

- *Servizio*: un gruppo logico di caratteristiche correlate che svolgono un ruolo specifico. Ad esempio, un servizio potrebbe rappresentare la funzionalità di un sensore di temperatura o di un dispositivo di misurazione della frequenza cardiaca. Ogni servizio è identificato da un UUID (Universally Unique Identifier).

- *Caratteristica*: una singola entità di dati all'interno di un servizio. Ogni caratteristica ha un valore e può contenere informazioni come la lettura/scrittura di dati, notifiche e indici. Come i servizi, anche le caratteristiche sono identificate da un UUID

- *UUID*: un identificatore univoco utilizzato per distinguere servizi, caratteristiche e altri elementi in una comunicazione BLE. Gli UUID possono essere di due tipi: 16-bit o 128-bit. Gli UUID a 16 bit sono standardizzati e identificano servizi e caratteristiche BLE comuni, mentre gli UUID a 128 bit sono più flessibili e possono essere utilizzati per identificare elementi personalizzati.

- *Advertising*: un meccanismo attraverso il quale i dispositivi BLE trasmettono informazioni a dispositivi vicini per annunciare la loro presenza. Durante la fase di advertising, un dispositivo invia pacchetti di advertising contenenti informazioni come il proprio indirizzo, l'elenco dei servizi supportati e altri dati rilevanti. Questo consente ad altri dispositivi di individuare e connettersi al dispositivo pubblicizzante.

- *Subscribe*: il processo mediante il quale un dispositivo BLE può richiedere di essere notificato o indicato quando il valore di una caratteristica cambia. Quando un dispositivo è connesso e ha sottoscritto a una caratteristica, riceverà automaticamente aggiornamenti quando il valore della caratteristica cambia.

4.3.1 ArduinoBLE

Il primo passo è quello di trasmettere i dati ottenuti in precedenza dalla scheda, e ciò si può effettuare mediante la libreria ArduinoBLE [23]. Come per le librerie precedenti, bisogna includere anche questa mediante il comando *#include <ArduinoBLE.h>*.

A questo punto, è necessario definire un servizio e una caratteristica, di tipo stringa, in cui andare a scrivere i valori ricevuti dai sensori, e assegnare un UUID a 16 bit ad entrambi per identificarli. Questo avviene mediante i comandi *BLEService orientationService("181A")* e *BLEStringCharacteristic valueCharacteristic("2A5D", BLERead | BLENotify, 20)*, nel caso della caratteristica vanno anche specificati i parametri BLERead e BLENotify per abilitare la possibilità di leggere il valore e di essere notificati del suo cambiamento, e il massimo numero di caratteri della stringa, ovvero 20.

Nella funzione *setup()* è poi necessario iniziare la comunicazione BLE tramite il metodo *BLE.begin()*, per poi aggiungere la caratteristica *valueCharacteristic* al servizio *orientationService*, aggiungere il servizio e infine effettuare l'advertising.

```
if (!BLE.begin()) {
  //Serial.println("Failed to initialize BLE!");
  while (1);
}
orientationService.addCharacteristic(valueCharacteristic);
BLE.setAdvertisedService(orientationService);
BLE.addService(orientationService);
BLE.advertise();
```

Figura 4.10: Setup BLE.

Nella funzione *loop()* poi si può procedere con la scrittura del valore ogni volta che se ne ottiene uno, aggiungendo un flag iniziale per distinguere i valori dell'accelerometro, con flag settato ad A, da quelli del VL53L1X, con flag settato a T. Le funzioni per scrivere sono quindi rispettivamente *valueCharacteristic.writeValue("A," + String(ax))* e *valueCharacteristic.writeValue("T," + String(tof))*.

4.3.2 Unity Android BLE Plugin

Il passo finale è quello di ricevere i dati trasmessi dalla scheda sul visore. I dispositivi Android, come Oculus Quest 2, sono in grado di comunicare tramite BLE mediante delle librerie specifiche di Android. Al fine di poter utilizzare in Unity le funzionalità di tali librerie, la scelta si è orientata su Unity Android Bluetooth Low Energy [24], ovvero un plugin Unity che, sfruttando proprio le librerie Android, permette di connettersi ad

un dispositivo ed effettuare il subscribe ad una caratteristica, per poi ricevere notifiche al variare del valore di essa.

Il primo passaggio da effettuare è quello di aggiungere alla scena un oggetto vuoto a cui deve essere aggiunto come componente lo script *BleManager* e immediatamente sotto in gerarchia un altro oggetto vuoto con un componente *BleAdapter*. La comunicazione con il sistema Android BLE avviene tramite la classe *BleCommand*, i cui comandi possono essere messi in coda nel *BleManager* che li esegue uno per volta, mentre i cambiamenti vengono inviati verso Unity al *BleAdapter*.

E' stato poi aggiunto un altro oggetto vuoto *Serial* che contiene il componente *Experiment1*, lo script a cui è affidata l'intera gestione dell'esperimento. All'interno di questo script vengono innanzitutto definite una serie di variabili dedicate alla gestione della comunicazione BLE, in particolare l'indirizzo della scheda Arduino *mac*, il tempo per cui effettuare la ricerca del dispositivo *_scanTime*, il contatore *_scanTimer*, i comandi *subscribeAcc* e *discover* rispettivamente per effettuare il subscribe e la ricerca, e i flag *_isScanning*, *connected* e *subscribed* che assumono valore *true* rispettivamente quando si sta effettuando la ricerca, quando si è connessi alla scheda Arduino e quando si è effettuato il subscribe alla caratteristica.

```
//BLE parameters
private string mac = "BB:74:A4:85:AB:4E";
private int _scanTime = 20000;
private float _scanTimer = 0f;
private bool _isScanning;
private bool connected;
private bool subscribed;
private SubscribeToCharacteristic subscribeAcc;
private DiscoverDevices discover = null;
```

Figura 4.11: Parametri BLE.

In seguito, all'interno della funzione *Awake* si vanno ad inizializzare i tre flag tutti a *false* e si imposta il testo all'interno di *connectText* a "Disconnected". L'oggetto *connectText* è parte dell'interfaccia utente che si vedrà nel Capitolo 7. All'interno invece della funzione *Update* se si sta effettuando la scansione si incrementa *_scanTimer* di una quantità *Time.deltaTime* pari all'intervallo dall'ultima chiamata della funzione, se questo contatore supera il massimo tempo di scansione viene azzerato e si imposta *_isScanning* a *false* e il testo a "Disconnected".

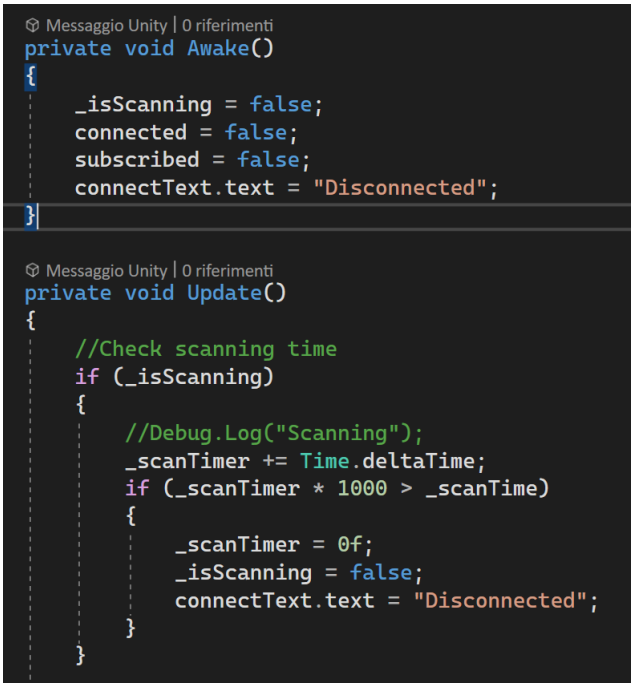


Figura 4.12: Funzioni start e update.

Per gestire la connessione ad Arduino si è creata una funzione *Connect* che può essere chiamata ogni volta che si vuole far partire una scansione. Qui *_isScanning* viene impostato a *true*, poi si fa in modo di fermare la scansione precedente se ve ne è una in corso tramite il comando *discover.End()*. A questo punto si crea un comando di tipo DiscoverDevices passando come parametri il tempo per cui effettuare la scansione e una callback *OnDeviceFound* da chiamare quando viene trovato un dispositivo, dopodichè questo comando viene messo in coda nel *BleManager* tramite la funzione *BleManager.Instance.QueueCommand(discover)*. Infine il testo viene impostato su "Connecting...".

All'interno della funzione *OnDeviceFound*, si controlla che il nome del dispositivo ricevuto da parametro sia uguale a *mac* e in caso si mette in coda un comando di tipo ConnectToDevice per connettersi, passando come parametri lo stesso *mac* e due callback, *OnConnected* da chiamare a connessione effettuata e *OnDisconnected* che viene chiamata quando si perde la connessione. Nella funzione *OnDisconnected* si settano a *false* i flag *connected* e *subscribed*, a "Disconnected" il testo e si rende cliccabile *connectButton*. Nella funzione *OnConnected* il flag *connected* assume valore *true* e *_isScanning false*, dopodichè si chiama la funzione *Subscribe*, si imposta il testo a "Connected!" e si disattiva il bottone per effettuare la connessione.

```
1 riferimento
private void OnDeviceFound(string name, string device)
{
    if (name == mac)
    {
        BleManager.Instance.QueueCommand(new ConnectToDevice(mac, OnConnected, OnDisconnected));
    }
}

1 riferimento
private void OnConnected(string uid)
{
    connected = true;
    _isScanning = false;
    Subscribe();
    connectText.text = "Connected!";
    connectButton.interactable = false;
}

1 riferimento
private void OnDisconnected(string uid)
{
    connected = false;
    subscribed = false;
    connectText.text = "Disconnected";
    connectButton.interactable = true;
}

0 riferimenti
public void Connect()
{
    _isScanning = true;
    if(discover != null)
    {
        discover.End();
    }
    discover = new DiscoverDevices(OnDeviceFound, _scanTime);
    BleManager.Instance.QueueCommand(discover);
    connectText.text = "Connecting...";
}
```

Figura 4.13: Funzioni OnDeviceFound, OnConnected, OnDisconnected e Connect.

Nella funzione *Subscribe* si effettua il subscribe alla caratteristica mettendo in coda il comando SubscribeToCharacteristic che prende come parametri il mac del dispositivo, gli UUID del servizio e della caratteristica e una callback che viene chiamata ogni volta che si riceve un dato. Questa chiama una funzione *OnDataReceived* in cui si effettua la gestione del dato come si vedrà nel Capitolo 6. Infine si setta il valore di *subscribed* a *true*.

```
1 riferimento
private void Subscribe()
{
    subscribeAcc = new SubscribeToCharacteristic(mac, "181a", "2a5d", (byte[] value) =>
    {
        OnDataReceived(value);
    });
    BleManager.Instance.QueueCommand(subscribeAcc);
    subscribed = true;
}
```

Figura 4.14: Funzione Subscribed.

Capitolo 5

Costruzione dell'esperimento

Una volta effettuate le scelte riguardanti la piattaforma e i dispositivi, oltre ai sensori da cui ricavare le misure, il focus si è spostato sulla costruzione dell'esperimento da cui partire per lo sviluppo.

Si è deciso di partire dallo studio del moto rettilineo di un carrello su di una rotaia provvista di cuscino d'aria, in diverse condizioni in modo da avere:

1. Moto rettilineo uniforme
2. Moto rettilineo uniformemente accelerato in piano
3. Moto rettilineo uniformemente accelerato su piano inclinato

Si andranno di seguito a descrivere le tre condizioni e le leggi che ne regolano il moto in modo da ottenere il modello.

5.1 Moto rettilineo uniforme

La prima modalità in cui si può far operare la rotaia è quella in cui essa si trova in posizione orizzontale e il carrello rivece una spinta iniziale tramite un elastico. In questo caso il carrello si muove di moto rettilineo uniforme.

Il moto rettilineo uniforme [25] è un tipo di movimento in cui un oggetto si sposta lungo una retta in modo che la sua velocità sia costante nel tempo. Questo significa che l'oggetto si muove di quantità uguali in intervalli di tempo uguali. Definiamo *s0* come la posizione dell'oggetto all'istante iniziale, t come l'intervallo di tempo trascorso

dall'istante iniziale a quello attuale, s come la posizione dell'oggetto all'istante t e vm, la velocità media, come il rapporto tra la variazione di posizione e il tempo trascorso:

$$vm = \frac{\Delta s}{\Delta t} \tag{5.1}$$

Andiamo poi a definire la velocità istantanea v, che è pari alla derivata dello spazio rispetto al tempo:

$$v = \frac{ds}{dt} \tag{5.2}$$

Si può poi risolvere questa formula in ds ricavando:

$$ds = v \cdot dt \tag{5.3}$$

A questo punto si può procedere a integrare a sinistra rispetto allo spazio tra $s0$ e s e a destra rispetto al tempo tra $t0$ e t:

$$\int_{s0}^{s} ds = \int_{t0}^{t} v \cdot dt \tag{5.4}$$

Si può poi risolvere facilmente il primo integrale, portare fuori la velocità v dal secondo, essendo questa costante, e considerare $t0$ uguale a 0:

$$s - s0 = v \cdot \int_{0}^{t} dt \tag{5.5}$$

Effettuando gli ultimi calcoli, si arriva a ricavare la formula che regola la posizione di un oggetto in moto rettilineo uniforme:

$$s = so + v \cdot t \tag{5.6}$$

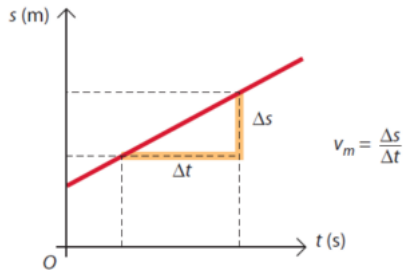


Figura 5.1: Rappresentazione grafica dell'evoluzione temporale del moto.

5.2 Moto rettilineo uniformemente accelerato

Il carrello, con la rotaia sempre in piano, può anche non avere alcuna spinta iniziale ma essere trainato da una seconda massa legata al carrello stesso tramite un filo passante per una carrucola. La Figura 5.2 rappresenta una schematizzazione dell'esperimento, in cui la massa 2 rappresenta il carrello, la massa 1 rappresenta la massa trainante, *Fa* la forza di attrito che nel caso ideale può essere trascurata, *Fp* la forza peso e *T* la tensione del filo. Quest'ultima, considerando il caso ideale di filo inestensibile e carrucola ideale si può considerare uguale da entrambi i lati.

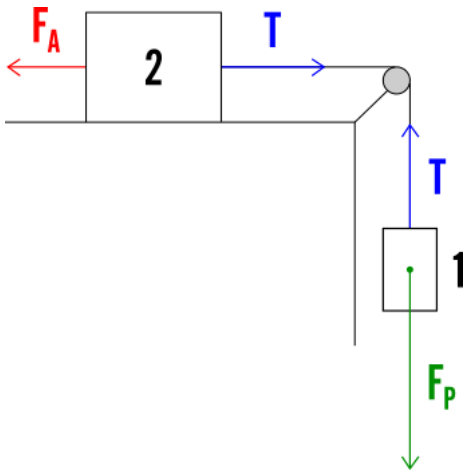


Figura 5.2: Schematizzazione dell'esperimento.

Per arrivare a ricavare l'accelerazione del carrello bisogna partire dalle seguenti equazioni che tengono conto delle risultanti delle forze applicate alle masse *m1* e *m2*:

$$T = m2 \cdot a \tag{5.7}$$

$$Fp - T = m1 \cdot a \tag{5.8}$$

Sostituendo T nella formula 5.8 si ottiene:

$$m1 \cdot g - m2 \cdot a = m1 \cdot a \tag{5.9}$$

A questo punto si possono separare i termini contenenti a per ottenere:

$$m1 \cdot g = (m1 + m2) \cdot a \tag{5.10}$$

E dunque l'accelerazione risulta essere pari a:

$$a = \frac{m1}{m1 + m2} \cdot g \tag{5.11}$$

Il carrello si muove con accelerazione costante, dunque di moto rettilineo uniformemente accelerato [26], questo significa che la velocità dell'oggetto cambia di quantità uguali in intervalli di tempo uguali. L'accelerazione a è pari alla derivata della velocità rispetto al tempo:

$$a = \frac{dv}{dt} \tag{5.12}$$

Anche in questo caso si può procedere a separare le variabili nel seguente modo:

$$dv = a \cdot dt \tag{5.13}$$

Integrando a sinistra rispetto alla velocita tra *v0* e v e a destra rispetto al tempo tra *t0* e t si ottiene:

$$\int_{v0}^{v} dv = \int_{t0}^{t} a \cdot dt \tag{5.14}$$

Si può risolvere facilmente l'integrale rispetto al tempo e portare fuori l'accelerazione dall'integrale temporale, essendo costante. Inoltre si può considerare *t0* pari a 0:

$$v - v0 = a \cdot \int_{0}^{t} dt \tag{5.15}$$

Si ottiene dunque la formula che regola la velocità come:

$$v = v0 + a \cdot t \tag{5.16}$$

Avendo ottenuto la formula della velocità, si puo utilizzare la formula 5.4 per ricavare la legge oraria del moto, dunque:

$$\int_{s0}^{s} ds = \int_{t0}^{t} (v0 + a \cdot t) \cdot dt \tag{5.17}$$

Si può quindi procedere alla risoluzione del primo integrale e alla separazione del secondo, essendo *v0* e a costanti poi queste possono essere portate fuori. Anche in

questo caso consideriamo *t0* pari a 0:

$$s = s0 + v0 \cdot \int_0^t dt + a \cdot \int_0^t t \cdot dt \tag{5.18}$$

Effettuando gli ultimi calcoli si ottiene dunque la legge oraria:

$$s = s0 + v0 \cdot t + \frac{1}{2} \cdot a \cdot t^2 \tag{5.19}$$

5.3 Moto sul piano inclinato

La terza condizione in cui si può trovare la rotaia è quella in cui è inclinata e il carrello viene lasciato scorrere liberamente su di essa. In questo caso si va a generare di nuovo un moto rettilineo uniformemente accelerato e si possono quindi utilizzare nuovamente le formule 5.16 e 5.19.

Ciò che bisogna fare è ottenere il valore dell'accelerazione in questa condizione di piano inclinato [27], considerando per ora sempre trascurabili gli attriti.

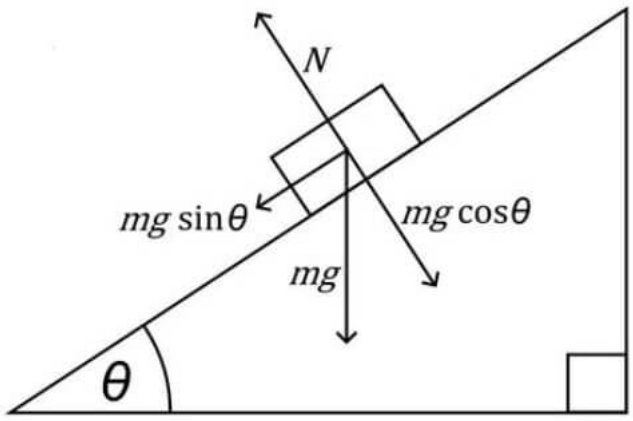


Figura 5.3: Schematizzazione del piano inclinato.

Partendo dalla schematizzazione dell'esperimento in Figura 5.3, si evidenzia come in questo caso sull'oggetto agiscano la forza peso verso il basso e una reazione vincolare N verso il basso. Scomponendo la forza peso nelle sue componenti perpendicolare e parallela al piano si ottengono le equazioni:

$$N = m \cdot g \cdot cos\vartheta \tag{5.20}$$

$$m \cdot g \cdot sin\vartheta = m \cdot a \tag{5.21}$$

A questo punto, risolvendo la seconda equazione si ottiene il valore di accelerazione anche in questo caso, dipendente esclusivamente dall'inclinazione del piano:

$$a = g \cdot sin\vartheta \tag{5.22}$$

Capitolo 6

Sensor fusion

Una volta collezionati i dati ricevuti dai sensori, la sfida successiva è stata quella di trovare un meccanismo per unire queste misure in modo da avere un rilevamento della posizione ad ogni istante che sia il più preciso possibile.

Un approccio tipico utilizzato in queste situazioni è quello di sfruttare un algoritmo di sensor fusion. In questo capitolo si andranno ad elencare i principali, e inoltre si deettaglieranno le caratteristiche e l'implementazione dell'algoritmo scelto tra questi, oltre ad elencare le ragioni alla base della scelta.

6.1 Algoritmi di sensor fusion

Tra i principali algoritmi di sensor fusion si possono annoverare:

- **Kalman Filter (Filtro di Kalman):** Il filtro di Kalman è un algoritmo di stima dello stato utilizzato per integrare dati da sensori al fine di ottenere una stima ottimizzata del vero stato di un sistema. È particolarmente efficace in scenari in cui i dati dei sensori possono essere soggetti a rumore o errori. L'algoritmo calcola due stime: la previsione dello stato futuro basata sullo stato attuale e l'aggiornamento dello stato basato sulla misurazione attuale. Questo filtro è ampiamente utilizzato in applicazioni come la localizzazione GPS, la navigazione inerziale e il tracciamento di oggetti.
- **Extended Kalman Filter (Filtro di Kalman Esteso):** L'Extended Kalman Filter (EKF) è una variante del filtro di Kalman utilizzata quando il sistema o il processo di misurazione non è lineare. L'EKF estende il concetto del filtro

di Kalman al caso in cui le equazioni di transizione e di osservazione non sono lineari. Questo lo rende adatto per applicazioni più complesse in cui i modelli non lineari devono essere considerati, come il tracciamento di oggetti in movimento in scenari complessi.

- **Complementary Filter (Filtro Complementare):** Il filtro complementare è un algoritmo di fusione sensoriale utilizzato per integrare dati da sensori inerziali, come accelerometri e giroscopi, con dati provenienti da altri tipi di sensori, come quelli di visione o GPS. Questo approccio consente di ottenere una stima accurata dell'orientamento e del movimento in scenari in cui i sensori inerziali potrebbero essere affetti da errori. È comunemente utilizzato in applicazioni di realtà virtuale, droni e veicoli autonomi.

- **Mahony Filter (Filtro di Mahony):** Il filtro di Mahony è una variante del filtro complementare, focalizzata sulla fusione dei dati provenienti dai giroscopi e dagli accelerometri per calcolare l'orientamento. Questo filtro è noto per la sua efficienza computazionale ed è spesso utilizzato in applicazioni portatili, come dispositivi indossabili, per il monitoraggio dell'orientamento dell'utente.

- **Madgwick Filter (Filtro di Madgwick):** Simile al filtro di Mahony, il filtro di Madgwick è progettato per l'integrazione dei dati provenienti da giroscopi e accelerometri al fine di calcolare l'orientamento. Una delle sue principali caratteristiche è l'efficienza computazionale, che lo rende adatto per dispositivi con risorse limitate. Viene spesso utilizzato in applicazioni di realtà virtuale, droni e dispositivi mobili per il monitoraggio dell'orientamento.

6.2 Filtro di Kalman

La scelta finale è ricaduta sul filtro di Kalman, per via della sua efficienza e soprattutto della sua efficacia in situazioni in cui i sensori sono affetti da rumore.

6.2.1 Descrizione dell'algoritmo

Il filtro di Kalman [28] è un metodo di stima che utilizza una serie di misure successive nel tempo per calcolare una stima più accurata di una quantità in modo efficiente. Questo filtro è particolarmente utile quando si affrontano problemi di stima in presenza di rumore o incertezza nei dati.

L'algoritmo consiste sempre in uno step di previsione in cui lo stato attuale del sistema viene derivato dal precedente e da uno o più step di correzione, attraverso i quali viene corretta la stima attraverso le misure ricavate dai sensori, che possono essere anche più di uno. Di seguito si andrà a descrivere l'algoritmo nella sua formulazione più generale, partendo dalle definizioni fondamentali:

- **Stato del sistema**: il sistema che si sta monitorando è descritto da un vettore di stato x. Questo vettore rappresenta le variabili del sistema che si vogliono stimare.
- **Matrice di covarianza dello stato (P)**: questa matrice rappresenta l'incertezza associata alle stime delle variabili di stato. All'inizio, P viene inizializzata con le conoscenze iniziali e viene aggiornata dinamicamente durante l'esecuzione dell'algoritmo.
- **Matrice di transizione di stato (A)**: Rappresenta la relazione tra lo stato al tempo k e quello al tempo $k+1$.
- **Matrice di covarianza del processo (Q)**: rappresenta il rumore del processo e può essere modificato per riflettere la dinamica del sistema.
- **Matrice di covarianza dell'errore di misura (R)**: rappresenta l'incertezza nelle misure effettuate dal sistema.
- **Matrice di misura (H)**: rappresenta la relazione tra lo stato e la misura. Specifica quali variabili di stato influenzano le misurazioni.
- **Innovazione (y)**: la differenza tra la misura reale e la previsione del sistema.

Lo step di predizione si compone dei due passaggi seguenti:

1. **Calcolo della previsione dello stato (xp)**: ovvero la stima lo stato futuro del sistema basandosi sullo stato attuale.

$$x_p = A \cdot x_{n-1} \tag{6.1}$$

2. **Calcolo della previsione della covarianza (Pp)**: ovvero l'aggiornamento della covarianza dello stato per riflettere l'incertezza nella previsione.

$$P_p = A \cdot P_{n-1} \cdot A^T + Q \tag{6.2}$$

Dopodichè, in ogni step di correzione (uno per ogni sensore utilizzato) si andranno ad effettuare i seguenti calcoli:

1. **Calcolo dell'innovazione (yn)**: in cui si va a integrare la misura ricavata dal sensore zn.

$$y_n = z_n - H \cdot x_p \tag{6.3}$$

2. **Calcolo del Kalman gain (Kn)**: la matrice Kn determina quanto dare peso alla misura reale rispetto alla previsione del sistema.

$$K_n = P_p \cdot H^T \cdot (H \cdot P_p \cdot H^T + R)^{-1} \tag{6.4}$$

3. **Correzione dello stato (xn)**: si effettua l'aggiornamento della stima dello stato basandosi sulla misura reale e sulla previsione del sistema.

$$x_n = x_p + K_n \cdot y_n \tag{6.5}$$

4. **Correzione della covarianza (Pn)**: si aggiorna la covarianza dello stato per riflettere la nuova conoscenza.

$$P_n = (I - K_n \cdot H) \cdot P_p \tag{6.6}$$

   Dove I è la matrice identità.

L'algoritmo del filtro di Kalman è efficace nella stima di variabili di stato in presenza di rumore nelle misure e dinamiche complesse del sistema. La chiave del successo del filtro di Kalman è la sua capacità di combinare in modo ottimale le informazioni provenienti dalle misure e dalla dinamica del sistema per ottenere una stima più accurata dello stato.

Nel caso in esame, si è optato per una versione semplificata dell'algoritmo per rendere i calcoli meno onerosi per il software. In particolare, trattandosi di un moto unidimensionale, è stato possibile sostituire i calcoli matriciali con semplici moltiplicazioni, come si vedrà nei paragrafi seguenti. Sono stati implementati due step di correzione, uno per ognuno dei due sensori descritti nel Capitolo 4, che correggono la stima effettuata tramite le formule previste dal modello descritto nel Capitolo 5. Tuttavia, al fine di rendere questo modello il più possibile aderente alla realtà, si è andati a modificarlo con un'aggiunta della componente di attrito.

6.2.2 Modellazione degli attriti

Al fine di ottenere una accurata modellazione degli attriti, il primo passaggio è stato quello di effettuare delle misurazioni sulla rotaia che si andrà ad utilizzare per l'esperimento con condizioni diverse.

Nel caso del moto uniformemente accelerato in piano, si sono misurati i tempi impiegati dal carrello a percorrere 0.3, 0.6 e 0.9 metri, con la massa di base pari a 0.22 Kg e massa trainante sempre pari a 0.025 Kg. Vi era poi una massa aggiuntiva posta sul carrello, che è stata posta a 0 Kg, 0.06 Kg, 0.1 Kg e 0.2 Kg.

0.3 m	0.6 m	0.9 m
0.865 s	1.198 s	1.437 s
0.872 s	1.228 s	1.514 s
0.875 s	1.207 s	1.446 s

Tabella 6.1: Massa aggiuntiva pari a 0 Kg

0.3 m	0.6 m	0.9 m
0.968 s	1.338 s	1.603 s
0.973 s	1.368 s	1.689 s
0.968 s	1.361 s	1.679 s

Tabella 6.2: Massa aggiuntiva pari a 0.06 Kg

0.3 m	0.6 m	0.9 m
1.034 s	1.428 s	1.710 s
1.017 s	1.411 s	1.693 s
1.026 s	1.445 s	1.783 s

Tabella 6.3: Massa aggiuntiva pari a 0.1 Kg

0.3 m	0.6 m	0.9 m
1.155 s	1.603 s	1.922 s
1.173 s	1.621 s	1.941 s
1.162 s	1.609 s	1.929 s

Tabella 6.4: Massa aggiuntiva pari a 0.2 Kg

Partendo da queste misure si è andati a calcolare l'accelerazione attraverso la formula 5.11 a cui è stato aggiunto l'attrito, tramite il coefficiente μ, ricavando la formula:

$$a = \frac{m1 - \mu \cdot m2}{m1 + m2} \cdot g \tag{6.7}$$

Dall'accelerazione si è poi passati a calcolare i tempi impiegati a percorrere le tre distanze con m2 variabile attraverso la formula:

$$t = \sqrt{\frac{s}{2 \cdot a}} \tag{6.8}$$

Si è giunti alla conclusione che, considerando un coefficiente di attrito μ pari a 0.005, si ottenevano dei risultati molto aderenti alle misure effettuate aggiungendo una massa zavorra mz al denominatore pari a 0.05, ottenendo la formula:

$$a = \frac{m1 - \mu \cdot m2}{m1 + m2 + mz} \cdot g \tag{6.9}$$

Nella Figura 6.1 si può osservare l'andamento dell'accelerazione al variare di m2.

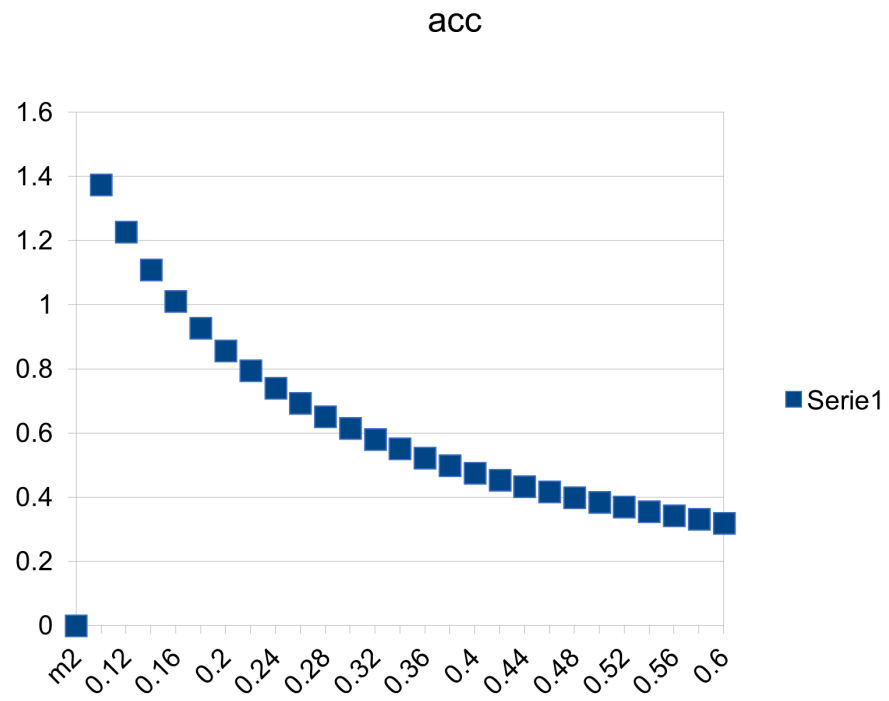


Figura 6.1: Plot dei valori di accelerazione al variare di m2.

Nella Figura 6.2 si possono osservare i valori di accelerazione e tempi, con evidenziati i casi in cui la massa m2 è pari a quella totale del carrello nei casi misurati. Si può notare come i calcoli rispecchino molto bene l'andamento delle misure.

m2	a	0.3	0.6	0.9
0.1	1.3734	0.660963	0.934743	1.144821
0.12	1.227507692	0.699139	0.988732	1.210944
0.14	1.10875814	0.735626	1.040333	1.274142
0.16	1.010221277	0.770668	1.089889	1.334836
0.18	0.927141176	0.804457	1.137674	1.39336
0.2	0.856145455	0.837147	1.183905	1.449982
0.22	0.794776271	0.868867	1.228763	1.504921
0.24	0.7412	0.899721	1.272398	1.558363
0.26	0.694020896	0.9298	1.314935	1.61046
0.28	0.652157746	0.959178	1.356483	1.661345
0.3	0.61476	0.987922	1.397133	1.711132
0.32	0.581149367	1.016089	1.436967	1.759918
0.34	0.550778313	1.043728	1.476054	1.807789
0.36	0.5232	1.070882	1.514456	1.854823
0.38	0.498046154	1.097592	1.552229	1.901085
0.4	0.475010526	1.123891	1.589421	1.946635
0.42	0.453836364	1.14981	1.626077	1.991529
0.44	0.434306796	1.175377	1.662235	2.035813
0.46	0.416237383	1.200619	1.697931	2.079532
0.48	0.39947027	1.225557	1.733199	2.122726
0.5	0.383869565	1.250212	1.768067	2.165431
0.52	0.369317647	1.274605	1.802564	2.207681
0.54	0.355712195	1.298752	1.836713	2.249505
0.56	0.34296378	1.32267	1.870538	2.290932
0.58	0.330993893	1.346374	1.90406	2.331988
0.6	0.319733333	1.369877	1.937299	2.372697

Figura 6.2: Misure di accelerazione e tempi.

Per quanto riguarda il moto sul piano inclinato, sono state effettuate delle misure con la stessa massa m2 del carrello e con le stesse masse aggiuntive, in condizioni di diverse inclinazioni.

Misure con ϑ pari a 4 gradi:

0.3 m	0.6 m	0.9 m
0.914 s	1.259 s	1.524 s
0.913 s	1.258 s	1.523 s
0.908 s	1.253 s	1.518 s

Tabella 6.5: Massa aggiuntiva 0 Kg.

0.3 m	0.6 m	0.9 m
0.903 s	1.248 s	1.512 s
0.899 s	1.244 s	1.509 s
0.904 s	1.249 s	1.513 s

Tabella 6.6: Massa aggiuntiva 0.06 Kg

0.3 m	0.6 m	0.9 m
0.900 s	1.248 s	1.512 s
0.898 s	1.244 s	1.509 s
0.097 s	1.242 s	1.513 s

Tabella 6.7: Massa aggiuntiva 0.1 Kg.

0.3 m	0.6 m	0.9 m
0.894 s	1.239 s	1.504 s
0.895 s	1.240 s	1.509 s
0.895 s	1.241 s	1.505 s

Tabella 6.8: Massa aggiuntiva 0.2 Kg

Misure con ϑ pari a 5.5 gradi:

0.3 m	0.6 m	0.9 m
0.816 s	1.141 s	1.387 s
0.815 s	1.139 s	1.389 s
0.817 s	1.141 s	1.388 s

Tabella 6.9: Massa aggiuntiva 0 Kg.

0.3 m	0.6 m	0.9 m
0.815 s	1.139 s	1.386 s
0.813 s	1.139 s	1.386 s
0.816 s	1.140 s	1.387 s

Tabella 6.10: Massa aggiuntiva 0.06 Kg

0.3 m	0.6 m	0.9 m
0.815 s	1.139 s	1.386 s
0.812 s	1.136 s	1.383 s
0.813 s	1.137 s	1.384 s

Tabella 6.11: Massa aggiuntiva 0.1 Kg.

0.3 m	0.6 m	0.9 m
0.812 s	1.135 s	1.383 s
0.811 s	1.134 s	1.381 s
0.810 s	1.133 s	1.381 s

Tabella 6.12: Massa aggiuntiva 0.2 Kg

Misure con ϑ pari a 7 gradi:

0.3 m	0.6 m	0.9 m
0.797 s	1.116 s	1.360 s
0.797 s	1.116 s	1.360 s
0.795 s	1.115 s	1.358 s

Tabella 6.13: Massa aggiuntiva 0 Kg.

0.3 m	0.6 m	0.9 m
0.797 s	1.117 s	1.360 s
0.796 s	1.115 s	1.358 s
0.796 s	1.116 s	1.359 s

Tabella 6.14: Massa aggiuntiva 0.06 Kg

0.3 m	0.6 m	0.9 m
0.797 s	1.117 s	1.360 s
0.796 s	1.115 s	1.358 s
0.796 s	1.116 s	1.359 s

Tabella 6.15: Massa aggiuntiva 0.1 Kg.

0.3 m	0.6 m	0.9 m
0.794 s	1.113 s	1.357 s
0.793 s	1.113 s	1.356 s
0.795 s	1.114 s	1.357 s

Tabella 6.16: Massa aggiuntiva 0.2 Kg

Misure con ϑ pari a 9 gradi:

0.3 m	0.6 m	0.9 m
0.798 s	1.103 s	1.336 s
0.795 s	1.101 s	1.336 s
0.795 s	1.101 s	1.336 s

Tabella 6.17: Massa aggiuntiva 0 Kg.

0.3 m	0.6 m	0.9 m
0.791 s	1.097 s	1.332 s
0.793 s	1.099 s	1.334 s
0.793 s	1.099 s	1.334 s

Tabella 6.18: Massa aggiuntiva 0.06 Kg

0.3 m	0.6 m	0.9 m
0.790 s	1.097 s	1.332 s
0.793 s	1.099 s	1.334 s
0.793 s	1.100 s	1.334 s

Tabella 6.19: Massa aggiuntiva 0.1 Kg.

0.3 m	0.6 m	0.9 m
0.792 s	1.097 s	1.332 s
0.793 s	1.097 s	1.335 s
0.793 s	1.100 s	1.336 s

Tabella 6.20: Massa aggiuntiva 0.2 Kg

Considerato l'andamento scaturito dalle misure, si è notato come queste vadano a raggiungere un'uniformità al salire dell'angolo di inclinazione, mentre l'effetto dell'attrito è maggiore ad angoli bassi. Per modellare questo comportamento, è stato fatto ricorso ad un sigmoide che moltiplica l'accelerazione di gravità ottenendo la seguente formula:

$$a = g \cdot \frac{1}{1 + 10^{-\vartheta}} \cdot k \tag{6.10}$$

In questa formula k è un parametro che modella l'effetto dell'attrito, e al cui salire diminuisce l'effetto di esso. Considerando un valore di k pari a 0.11 si ottiene un sigmoide con un andamento come in Figura 6.3, dove si può apprezzare come il valore salga per poi raggiungere un plateau come visto nelle misure. I valori dei tempi impiegati visibili in Figura 6.4 rispecchiano anch'essi l'andamento delle misure effettuate

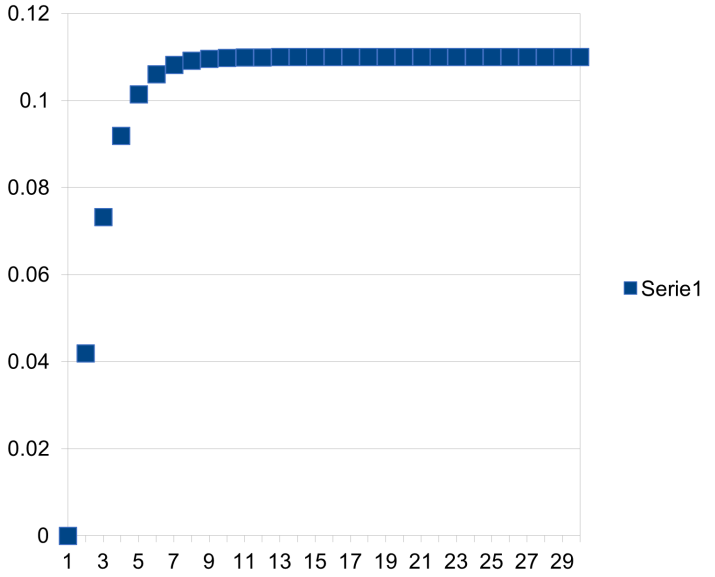


Figura 6.3: Plot dei valori del sigmoide.

alpha	rad	sin	sigmoide				
1	0.017453	0.007454	#NUM!	#NUM!	0.3	0.6	0.9
2	0.034907	0.024906	0.041967	0.4117	1.207217	1.707262	2.09096
3	0.05236	0.04235	0.073259	0.718671	0.913715	1.292188	1.5826
4	0.069813	0.059781	0.091865	0.901194	0.815956	1.153936	1.413277
5	0.087266	0.077194	0.101479	0.995513	0.77634	1.097911	1.344661
6	0.10472	0.094583	0.106091	1.040754	0.759279	1.073783	1.31511
7	0.122173	0.111944	0.108225	1.061691	0.751755	1.063143	1.302079
8	0.139626	0.12927	0.109197	1.071226	0.748402	1.0584	1.29627
9	0.15708	0.146558	0.109637	1.075541	0.746899	1.056275	1.293668
10	0.174533	0.1638	0.109836	1.07749	0.746224	1.05532	1.292497
11	0.191986	0.180993	0.109926	1.07837	0.745919	1.054889	1.29197
12	0.20944	0.19813	0.109966	1.078768	0.745781	1.054694	1.291731
13	0.226893	0.215207	0.109985	1.078949	0.745719	1.054606	1.291623
14	0.244346	0.232219	0.109993	1.079031	0.745691	1.054566	1.291574
15	0.261799	0.24916	0.109997	1.079068	0.745678	1.054547	1.291552
16	0.279253	0.266025	0.109999	1.079085	0.745672	1.054539	1.291541
17	0.296706	0.282809	0.109999	1.079093	0.745669	1.054535	1.291537
18	0.314159	0.299506	0.11	1.079097	0.745668	1.054534	1.291535
19	0.331613	0.316113	0.11	1.079099	0.745667	1.054533	1.291534
20	0.349066	0.332623	0.11	1.079099	0.745667	1.054532	1.291533
21	0.366519	0.349032	0.11	1.0791	0.745667	1.054532	1.291533
22	0.383972	0.365335	0.11	1.0791	0.745667	1.054532	1.291533
23	0.401426	0.381526	0.11	1.0791	0.745667	1.054532	1.291533
24	0.418879	0.397601	0.11	1.0791	0.745667	1.054532	1.291533
25	0.436332	0.413555	0.11	1.0791	0.745667	1.054532	1.291533
26	0.453786	0.429383	0.11	1.0791	0.745667	1.054532	1.291533
27	0.471239	0.44508	0.11	1.0791	0.745667	1.054532	1.291533
28	0.488692	0.460642	0.11	1.0791	0.745667	1.054532	1.291533
29	0.506145	0.476063	0.11	1.0791	0.745667	1.054532	1.291533
30	0.523599	0.49134	0.11	1.0791	0.745667	1.054532	1.291533

Figura 6.4: Valori ottenuti dai calcoli.

6.2.3 Implementazione in Unity

Nell'implementare l'algoritmo in Unity il primo passo è quello di definire alcune variabili fondamentali. Prima di tutto vi sono alcune variabili float, il cui valore viene impostato tramite menù con meccanismi che verranno descritti nel Capitolo 7, tra cui possiamo elencare *angle*, l'inclinazione del piano, *mass1*, la massa del carrello, *mass2*, la massa trainante, *mz*, la massa zavorra di correzione della formula dell'accelerazione, *mu*, il coefficiente di attrito, *k*, il coefficiente di attrito presente nel sigmoide in caso di piano inclinato, *simulation_initialVelocity*, la velocità iniziale in caso di moto rettilineo uniforme, P, la covarianza dello stato, Q, la covarianza del processo, R, la covarianza dell'errore di misura dell'accelerometro e R_TOF, la covarianza dell'errore di misura del lidar. Inoltre, vengono definite x e *velocity*, le variabili di stato posizione e velocità, *simulation_acceleration*, l'accelerazione ricavata dal modello, *simulation_velocity* e *simulation_distance*, variabili accessorie per contenere valori intermedi, *acceleration*, l'accelerazione misurata dall'accelerometro, *startSimulation*, un flag che segnala se la simulazione è iniziata, *firstTof* e *prev_tof*, necessarie per la gestione dei valori ricavati dal sensore lidar, *simulation_force*, *simulation_force_up* e *simulation_max_distance* che nel Capitolo 7 saranno necessarie alla visualizzazione dei vettori e all'individuazione della fine della simulazione.

```
//Simulation parameters
private float angle;
private float mass1;
private float mass2;
private float mz;
private float mu;
private float k;
private float simulation_initialVelocity;
private float simulation_acceleration;
private float simulation_force;
private float simulation_force_up;
private float simulation_velocity;
private float simulation_distance;
private float simulation_max_distance;
private bool startSimulation = false;
private float acceleration;
private float prev_tof;
private bool firstTof = true;

// Kalman filter parameters
private float x;         // State estimate (1D position)
private float velocity;         // State estimate (1D velocity)
private float P;         // Covariance of the estimate
private float Q;         // Process noise covariance
private float R;         // Measurement noise covariance
private float R_TOF;      //Tof measurement noise covariance
```

Figura 6.5: Parametri necessari alla simulazione.

All'interno della funzione *FixedUpdate*, chiamata ad ogni frame ad intervalli regolari, si controlla che il sensore sia connesso, che il subscribe sia effettuato e che il modello 3D sia istanziato (vedi Capitolo 7). In caso positivo, si controlla se viene premuto il pulsante A dei controller, da premere quando il sensore è in posizione per l'inizio della simulazione, e in se ciò avviene si inizializza *simulation_ acceleration* con il valore della formula 6.9 in caso di angolo pari a 0 gradi, o con il valore dato dalla formula 6.10 negli altri casi. Si procede inoltre a dare il valore iniziale di velocità, contenuto in *simulation_ initialVelocity*, alla variabile *simulation_ velocity* e a impostare a 0 sia le variabili di stato sia la variabile ausiliaria *simulation_ distance*.

```
if (connected && subscribed && instantiated)
{
    if (OVRInput.Get(OVRInput.Button.One))
    {
        //Set parameters for simulation
        if (angle == 0f)
        {
            simulation_acceleration = 9.81f * (mass2 - mu * mass1) / (mass1 + mass2 + mz);
        }
        else
        {
            simulation_acceleration = 1 / (1 + Mathf.Pow(10, -angle)) * k * 9.81f;
        }
        simulation_distance = 0f;
        simulation_velocity = simulation_initialVelocity;

        // Initialize Kalman filter parameters
        x = 0.0f;
        velocity = 0.00f;
```

Figura 6.6: Inizializzazione dei parametri.

Come vedremo in seguito, alla ricezione del valore dall'accelerometro, questo viene salvato nella variabile *acceleration* in g. A questo punto, sempre all'interno della funzione *FixedUpdate*, si controlla il valore di *startSimulation* e se questo è posto a *true* si procede con lo step di simulazione. Il primo step è quello di ricavare dall'accelerazione il valore di velocità e posizione tramite integrazione. Questa viene effettuata attraverso il metodo di Runge-Kutta [29], in particolare del quarto ordine. Questo metodo è una forma di approccio a passo fisso, il che significa che suddivide l'intervallo di integrazione in passi discreti e calcola la soluzione approssimata in ciascun passo. L'approssimazione viene effettuata calcolando quattro coefficienti in cui il primo è la variabile da integrare, il secondo e il terzo si calcolano sommandogli il coefficiente precedente diviso per due e moltiplicato per l'intervallo di tempo, in questo caso *timeSt* pari al tempo che trascorre tra due chiamate di *FixedUpdate*, e il quarto sommandogli solo il precedente moltiplicato per il delta temporale. I coefficienti vengono poi sommati con pesi differenti, moltiplicati per l'intervallo di tempo e divisi per 6. Questo

processo viene ripetuto due volte per ottenere il *dv*, che sommato a *velocity* dà il nuovo valore di velocità, e il *dx*, integrando la velocità.

Una volta così ottenute la velocità e la variazione di posizione stimate dal sensore, si procede con il calcolo delle variazioni di velocità e posizione stimate dal modello mediante le formule viste nel Capitolo 5 e a salvarle nelle variabili temporanee *simulation_dv* e *simulation_dx*. Si può poi proseguire con lo step di predizione del filtro di Kalman, in cui si somma a *simulation_distance* e *simulation_velocity*, che in questo momento contengono gli stessi valori dello stato, rispettivamente *simulation_dx* e *simulation_dv*. Si stima poi anche la covarianza sommando P e Q e si salva il risultato in *predictionCovariance*.

```
//Simulate
if (startSimulation)
{
    // Update velocityX based on accelerometer data
    float accelerationX = (acceleration + Mathf.Sin((angle * Mathf.PI) / 180)) * 9.81f;

    // Apply RK4 integration for velocityX
    float k1 = accelerationX;
    float k2 = accelerationX + k1 * timeSt * 0.5f;
    float k3 = accelerationX + k2 * timeSt * 0.5f;
    float k4 = accelerationX + k3 * timeSt;

    // Update velocityX using weighted averages
    float dv = (k1 + 2 * k2 + 2 * k3 + k4) * (timeSt / 6.0f);
    float v = velocity + dv;

    // Update position using a similar RK4 integration
    float l1 = v;  // Initial position change
    float l2 = v + 0.5f * l1 * timeSt; // Position change after half the time step
    float l3 = v + 0.5f * l2 * timeSt; // Position change again after half the time step
    float l4 = v + l3 * timeSt; // Position change at the end of the time step

    // Calculate dx using weighted averages
    float dx = (l1 + 2 * l2 + 2 * l3 + l4) * (timeSt / 6.0f);

    //Simulate
    float simulation_dv = simulation_acceleration * timeSt;
    float simulation_dx = (simulation_velocity * timeSt) + (simulation_acceleration * timeSt * timeSt / 2);

    // Kalman filter prediction step
    float prediction_x = simulation_distance + simulation_dx;  // Predicted position
    float prediction_v = simulation_velocity + simulation_dv;       //Predicted velocity
    float predictionCovariance = P + Q;  // Predicted covariance

    // Kalman filter update step
    float kalmanGain = predictionCovariance / (predictionCovariance + R);
    x = prediction_x + kalmanGain * ((simulation_distance + dx) - prediction_x);
    velocity = prediction_v + kalmanGain * (v - prediction_v);
    P = (1 - kalmanGain) * predictionCovariance;
    // Change in position
    float dx_kalman = x - simulation_distance;
    simulation_velocity = velocity;
    simulation_distance = x;
```

Figura 6.7: Simulazione tramite Kalman.

Il passo successivo è quello della correzione con i valori derivati dall'accelerometro. Si procede prima di tutto al calcolo del Kalman gain come *float kalmanGain = predictionCovariance / (predictionCovariance + R)* e poi all'aggiornamento delle variabili x e *velocity*. In entrambi gli aggiornamenti si calcola l'innovazione come valore ottenu-

to dalle misure (rispettivamente *simulation_ distance + dx* e *v*) a cui viene sottratto il valore della previsione. L'innovazione viene poi moltiplicata per il Kalman gain e sommata alla previsione. Poi, si provvede ad aggiornare la covarianza dello stato *P* come *(1 - kalmanGain) * predictionCovariance*. In ultima istanza, si salva il valore dello stato in *simulation_ distance* e *simulation_ velocity* dopo aver calcolato lo spostamento *dx_ kalman*, che verrà usato per aggiornare la posizione del carrello, come *x - simulation_ distance*, essendo *simulation_ distance* a questo punto ancora pari allo stato precedente.

All'interno della funzione *OnDataReceived*, vista nel Capitolo 4 e chiamata ad ogni ricezione di nuovi valori dal sensore, si procede con il parsing del valore in una stringa e alla divisione di essa nel punto in cui si trova la virgola, ottenendo un vettore che ha come primo elemento un carattere che indica il sensore da cui arriva il valore e come secondo elemento il valore stesso. Nel caso in cui il carattere sia pari ad "A" si procede con il parsing del valore dell'accelerometro e lo si assegna alla variabile *acceleration*, si chiama inoltre la funzione *CheckCusum* che, come si vedrà in seguito, individua l'inizio della simulazione. Se invece il carattere è pari a "T", si procede al parsing del valore del sensore lidar solo in caso di simulazione iniziata, dopodichè, nel caso in cui *firstTof* sia *true* come settato quando il sensore è in posizione, si provvede a impostarlo su *false* e si salva il valore ricevuto in *prev_ tof*. Se invece il valore non è il primo, si calcola la distanza percorsa dall'ultimo valore, ovvero l'innovazione, e si salva in *distance*, poi si provvede a correggere con un update del filtro il valore di posizione. Anche in questo caso quindi si aggiorna il Kalman gain, questa volta con la covarianza del lidar *R_ TOF*, poi si aggiorna lo stato *x*, la covarianza dello stato *P* , si calcola *dx_ kalman* come nel caso precedente e si salva il valore dello stato in *simulation_ distance*.

```
1 riferimento
private void OnDataReceived(byte[] value)
{
    string data = Encoding.UTF8.GetString(value);
    string[] stringData = data.Split(",");
    if (stringData[0].Equals("A"))
    {
        float acc = float.Parse(stringData[1], CultureInfo.InvariantCulture);
        //CUSUM
        CheckCusum(acc);
        acceleration = acc;
    }
    if (stringData[0].Equals("T") && startSimulation)
    {
        float tof = float.Parse(stringData[1], CultureInfo.InvariantCulture) / (1000f * Mathf.Cos((angle * Mathf.PI) / 180));
        if (firstTof)
        {
            prev_tof = tof;
            firstTof = false;
        }
        else
        {
            float distance = prev_tof - tof;
            // Kalman filter update step with the additional measurement
            float kalmanGain_additional = P / (P + R_TOF);
            x = x + kalmanGain_additional * (distance - x);
            P = (1 - kalmanGain_additional) * P;

            // Change in position
            float dx_kalman = x - simulation_distance;
            simulation_distance = x;
```

Figura 6.8: Funzione OnDataReceived.

6.3 CUSUM

Un altro algoritmo è stato poi necessario per risolvere un altro problema, ovvero quello dell'individuazione dell'istante di inizio della simulazione. Infatti, tramite il meccanismo di input già visto è possibile indicare quando il sensore è fermo e pronto all'inizio della simulazione, ma per individuare il momento in cui questo inizia a muoversi è stato necessario utilizzare l'algoritmo CUSUM.

6.3.1 Descrizione dell'algoritmo

L'algoritmo CUSUM [30], che sta per Cumulative Sum, è utilizzato per il monitoraggio di processi e il rilevamento di cambiamenti significativi o deviazioni da un normale comportamento. Questo algoritmo accumula (somma cumulativa) le differenze tra le osservazioni attuali e una media di riferimento. Quando la somma cumulativa supera una soglia predefinita, viene segnalato un cambiamento o una deviazione.

L'algoritmo si compone delle seguenti fasi:

1. **Definizione del Modello:** definizione di una media di riferimento (μ) e una deviazione standard (σ) per il processo che si sta monitorando.

2. **Inizializzazione:** inizializzazione della somma cumulativa S a zero.

3. **Iterazione:** per ogni nuova osservazione x_i calcolare l'osservazione normalizzata Z, centrata sulla media e scalata per la deviazione standard:

$$Z_i = \frac{x_i - \mu}{\sigma} \tag{6.11}$$

   Poi aggiornare la somma cumulativa S utilizzando la seguente formula:

$$S_i = max(0, S_{i-1} + Z_i) \tag{6.12}$$

4. **Decisione:** verificare se la somma cumulativa S supera una certa soglia K, tipicamente pari a:

$$K = n \cdot \sigma \tag{6.13}$$

   La scelta del parametro n e di conseguenza di K è fondamentale per un buon funzionamento dell'algoritmo.

5. **Aggiornamento:** se viene rilevato un cambiamento, azzerare la somma cumulativa o regolarla in base a criteri specifici del problema.

6.3.2 Implementazione in Unity

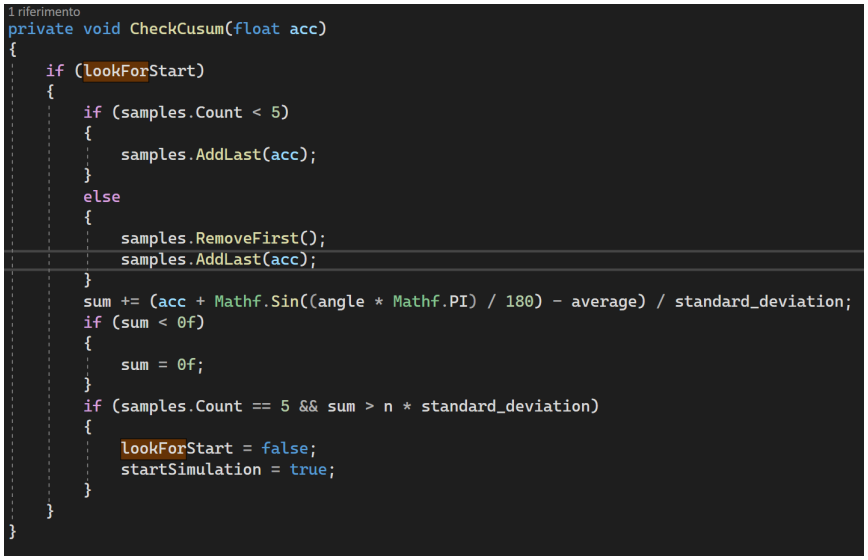
Si è scelto di utilizzare questo algoritmo monitorando i cinque campioni più recenti ricevuti dall'accelerometro, in modo da rilevare quando la somma dei sample normalizzati supera una certa soglia e il carrello inizia a muoversi.

Per implementare l'algoritmo in Unity, si parte dalla definizione di alcune variabili fondamentali, ovvero il booleano *lookForStart*, inizializzato a *false*, e che quando viene impostato su *true* innesca l'algoritmo CUSUM, i float *average* e *standard_ deviation* che rappresentano la media e la deviazione standard, l'intero n che rappresenta il parametro n dell'algoritmo, il float *sum* che rappresenta la somma S e infine una LinkedList di float *samples* utile a salvare i campioni ricevuti dall'accelerometro. Il parametro n è stato impostato a 5 e la scelta è avvenuta testando il funzionamento dell'algoritmo, mentre la media e la deviazione standard, rispettivamente 0 e 0.02, sono state calcolate da una serie di misure prese dall'accelerometro in condizioni statiche. Nel momento in cui l'utente preme il tasto A per segnalare che il carrello è fermo nella posizione di partenza, oltre alle operazioni descritte in precedenza, si procede anche a impostare *lookForStart* a *true* e *sum* a 0.

```
//CUSUM parameters
private bool lookForStart = false;
private float standard_deviation = 0.02f;
private float average = 0f;
private int n = 5;
private float sum;
private LinkedList<float> samples = new LinkedList<float>();
```

Figura 6.9: Parametri del CUSUM.

Nella funzione *CheckSum*, che viene chiamata ad ogni valore ricevuto dall'accelerometro e che riceve il suddetto valore tramite il parametro *acc*, si controlla il valore della variabile *lookForStart*. Se questo è pari a *true*, per prima cosa si aggiunge il campione alla lista *samples*, e se questa contiene già cinque campioni a rimuovere il meno recente. Dopodichè si aggiunge alla somma il valore normalizzato dell'accelerazione, ed eventualmente si azzera se il valore scende sotto lo 0. Infine, se i sample sono almeno cinque si controlla se la somma supera la soglia n * *standard_ deviation* e in caso affermativo si imposta *startSimulation* a *true*, dando il via alla simulazione, e *lookForStart* a *false*, terminando la ricerca del momento della partenza.



```
1 riferimento
private void CheckCusum(float acc)
{
    if (lookForStart)
    {
        if (samples.Count < 5)
        {
            samples.AddLast(acc);
        }
        else
        {
            samples.RemoveFirst();
            samples.AddLast(acc);
        }
        sum += (acc + Mathf.Sin((angle * Mathf.PI) / 180) - average) / standard_deviation;
        if (sum < 0f)
        {
            sum = 0f;
        }
        if (samples.Count == 5 && sum > n * standard_deviation)
        {
            lookForStart = false;
            startSimulation = true;
        }
    }
}
```

Figura 6.10: Funzione CheckSum.

Capitolo 7

Setup dell'applicazione

Lo scopo di questo capitolo è dettagliare lo sviluppo dell'applicazione dal punto di vista dell'implementazione dei meccanismi di input, dell'interfaccia utente e dei passaggi che questo deve effettuare. In particolare per quanto riguarda i menù di selezione dei parametri, la selezione dei punti di inizio e di arrivo, e la partenza e ripetizione della simulazione. Inoltre, si andrà a discutere dell'implementazione del meccanismo di visualizzazione dei vettori forza e velocità.

Poichè si parlerà di meccanismi di input, ci si riferirà alla mappatura dei pulsanti presenti sui controller di Oculus Quest2, che si può trovare alla Figura 7.1.

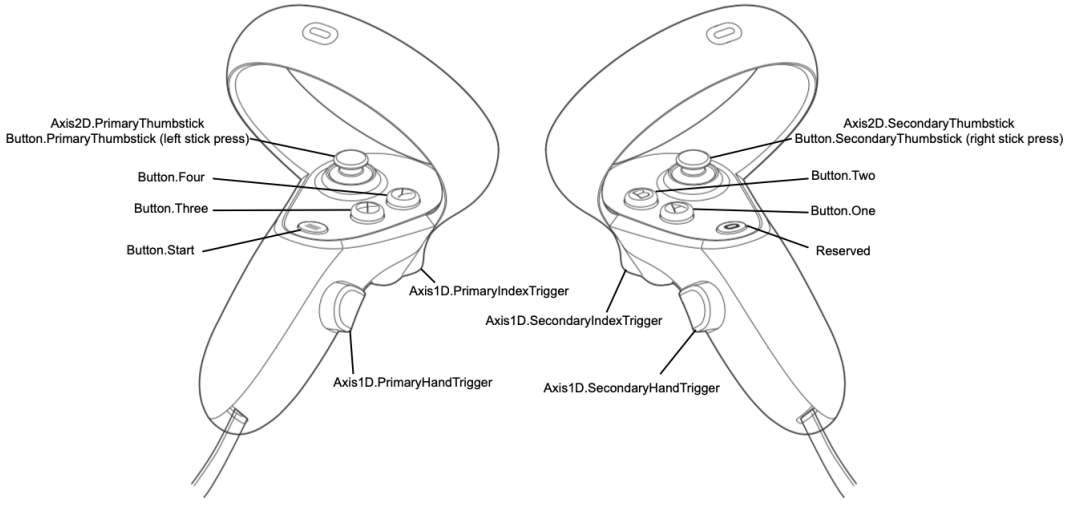


Figura 7.1: Mappatura dei controller.

7.1 Connessione del sensore e selezione dei parametri

All'apertura dell'applicazione, il primo passo che l'utente si trova a dover compiere è quello della connessione del sensore e la selezione dei parametri della simulazione tramite l'apposito menù.

7.1.1 Selezione tramite menù

Il menù prevede due diverse schermate tra cui l'utente può muoversi. Nella prima, che è abilitata di default all'apertura, è presente un pulsante CONNECT per connettersi al sensore e un box di testo che indica l'attuale stato della connessione. Un ulteriore pulsante SET PARAMETERS permette di passare alla schermata di selezione dei parametri, in cui è possibile scegliere i valori di tutti i parametri della simulazione, ognuno tramite un suo slider e frecce per incrementare o decrementare in maniera più precisa il valore. I parametri che si possono impostare sono l'angolo di inclinazione della rotaia *Angle* (tra 0 e 10 gradi), la velocità iniziale *V0* (tra 0 e 3 m/s), la massa trainante *Mass2* (tra 0 e 300 g), la massa del carrello *Mass1* (tra 1 e 500 g), i coefficienti di attrito K (tra 0 e 1) e MU (tra 0 e 1), la massa zavorra MZ (tra 0 e 300), e i parametri del filtro di Kalman Q (tra 0 e 1), P (tra 0 e 1), R (tra 0 e 10) e R_TOF (tra 0 e 10). Tutti gli input di selezione vengono effettuati tramite la pressione del pulsante IndexTrigger del controller destro.

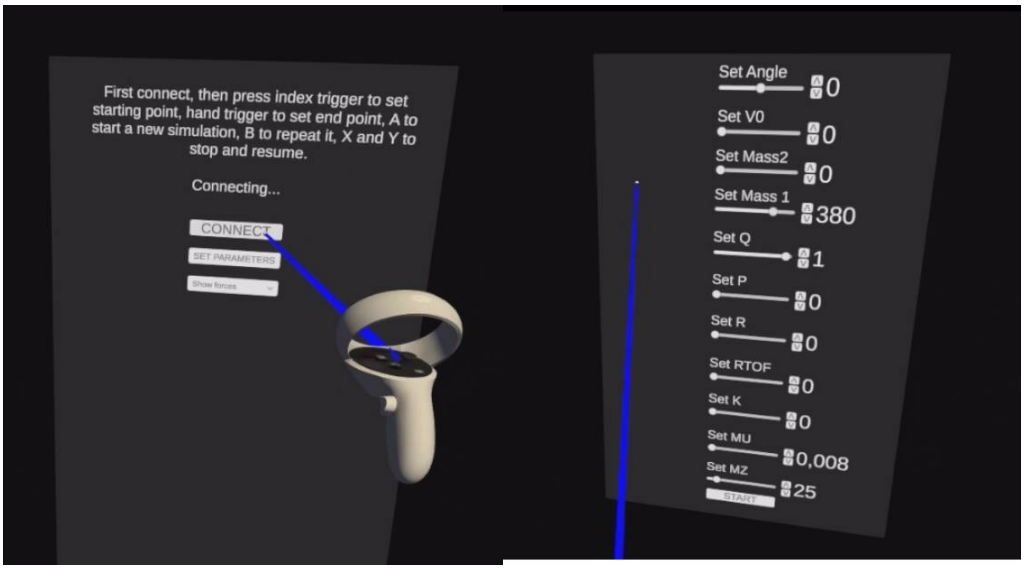


Figura 7.2: A sinistra menù di connessione, a destra menù parametri.

Il menù è implementato mediante un Canvas chiamato *MenuCanvas* che contiene due differenti pannelli.

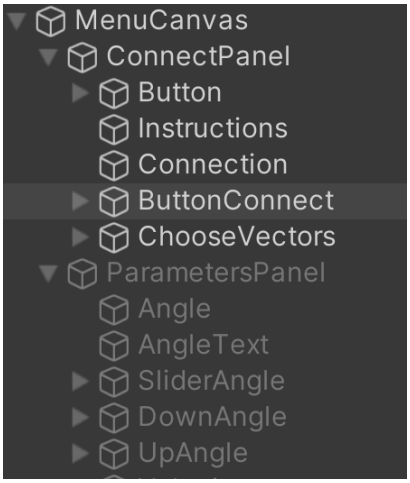


Figura 7.3: Gerarchia del MenuCanvas, per ParametersPanel solo l'esempio dell'angolo.

Il pannello *ConnectPanel* rappresenta il menù di connessione, e presenta un box di testo fisso *Instructions* con delle istruzioni, un box di testo *Connection* in cui si va a scrivere lo stato della connessione come visto nel Capitolo 4, un componente Button *ButtonConnect* che rappresenta il pulsante CONNECT, un componente Button *Button* che rappresenta il pulsante SET PARAMETERS e un componente Dropdown *ChooseVectors* che servirà per scegliere se mostrare i vettori forza o velocità. I componenti Button danno la possibilità di impostare nell'Inspector la funzione *OnClick* da chiamare nel momento in cui vengono premuti. A *ButtonConnect* è stata assegnata la funzione *Connect* vista nel Capitolo 4, mentre a *Button* è stata assegnata la funzione *OpenParameters* presente nello script *MenuController* assegnato a *MenuCanvas*. In questo script vengono definiti i GameObject *connectPanel* e *parametersPanel* corrispondenti ai due pannelli, nella funzione *OpenParameters* viene disattivato il primo e abilitato il secondo mediante i comandi *connectPanel.SetActive(false)* e *parametersPanel.SetActive(true)*. Il pannello *parametersPanel* invece presenta una serie di elementi che si ripetono per ognuno degli undici parametri, e che si andranno a spiegare facendo l'esempio del parametro *Angle*:

- Un box di testo, in questo caso chiamato *Angle*, in cui viene indicato il parametro a cui fa riferimento lo slider posto al di sotto di esso.

- Un secondo box di testo, in questo caso *AngleText*, posizionato alla destra dello slider, in cui viene scritto il valore del parametro. Al fine di scrivere il valore questo box di testo va passato allo script *MenuController*.

- Un componente Slider, in questo caso *SliderAngle*, a cui vanno impostati i valori minimo e massimo Min Value e Max Value tramite l'Inspector oltre a indicare tramite la checkbox Whole Numbers se si vogliono avere solo valori interi. Si passa inoltre una funzione *OnValueChanged* che viene chiamata al cambiare del valore dello slider, che in questo esempio è la funzione *OnAngleChange*, che riceve un parametro float *angle*. Il parametro viene impostato al testo di *angleText*, il box di testo già menzionato, dopo essere stato trasformato in stringa. Infine, questo viene salvato tramite il meccanismo dei *PlayerPrefs*, che permette di salvare semplici valori tra le sessioni nel registro locale. In questo modo sarà possibile accedere ai valori nello script che gestisce la simulazione.

```
0 riferimenti
public void OnAngleChange(float angle)
{
    angleText.text = angle.ToString();
    PlayerPrefs.SetFloat("Angle", angle);
}
```

Figura 7.4: Funzione OnAngleChange.

- Due bottoni rappresentanti le frecce per incrementare e decrementare il valore, in questo esempio *UpAngle* e *DownAngle*, che hanno come funzioni da chiamare al click rispettivamente *IncrementAngle* e *DecrementAngle*. In queste funzioni si estrae la variabile di interesse dai PlayerPrefs, ad esempio *float variable = PlayerPrefs.GetFloat("Angle")*, e si incrementa o decrementa dell'unità con la precisione richiesta dal valore, in questo caso 0.1 gradi. Dopodichè si salva il valore nel box di testo e di nuovo nei PlayerPrefs.

```
0 riferimenti
public void IncrementAngle()
{
    float variable = PlayerPrefs.GetFloat("Angle") + 0.1f;
    angleText.text = variable.ToString();
    PlayerPrefs.SetFloat("Angle", variable);
}

0 riferimenti
public void DecrementAngle()
{
    float variable = PlayerPrefs.GetFloat("Angle") - 0.1f;
    angleText.text = variable.ToString();
    PlayerPrefs.SetFloat("Angle", variable);
}
```

Figura 7.5: Funzioni IncrementAngle e DecrementAngle.

Infine, è presente un bottone *Button* che rappresenta il pulsante START dedicato al salvataggio dei parametri, che chiama ad ogni click due funzioni. La prima, chiamata *SetParameters*, nello script *MenuController* cambia la schermata con i comandi *connectPanel.SetActive(true)* e *parametersPanel.SetActive(false)*, la seconda, chiamata *UpdateParameters*, nello script dedicato alla gestione della simulazione assegna ad ogni variabile rappresentante il parametro il relativo valore presente nei PlayerPrefs.

```
1 riferimento
public void UpdateParameters()
{
    //Get parameters saved from menu
    angle = PlayerPrefs.GetFloat("Angle");
    mass1 = PlayerPrefs.GetFloat("Mass");
    mass2 = PlayerPrefs.GetFloat("Mass2");
    simulation_initialVelocity = PlayerPrefs.GetFloat("Velocity");
    Q = PlayerPrefs.GetFloat("Sigma");
    P = PlayerPrefs.GetFloat("P");
    R = PlayerPrefs.GetFloat("R");
    R_TOF = PlayerPrefs.GetFloat("RTOF");
    k = PlayerPrefs.GetFloat("K");
    mu = PlayerPrefs.GetFloat("MU");
    mz = PlayerPrefs.GetFloat("MZ");

}
```

Figura 7.6: Funzione UpdateParameters.

La possibilità di interagire con questa UI è data dall'aggiunta alla gerarchia della scena del GameObject *UIHelpers*. Questo contiene un GameObject *LaserPointer* che istanzia il raggio che parte dal controller e interseca la UI mediante il componente

LaserPointer, uno script definito dal package Oculus Integration, e un *Line Renderer* a cui assegnare il materiale del raggio, in questo caso di colore blu. Inoltre, vi è un EventSystem che ha come componente un altro script predefinito *OVRInputModule*, in cui si può indicare tramite il parametro Joy Pad Click Button il pulsante con cui interagire con la UI, in questo caso il Secondary Index Trigger.

7.1.2 Punto iniziale e di arrivo

Una volta scelti i parametri l'utente può procedere con l'input dei punti di inizio e fine della simulazione. Nella cartella Resources sono stati inseriti i prefab *carrello* e *sensor*, i modelli 3D della rotaia e del carrello. Il primo è modellato in modo da essere lungo esattamente un metro quando la sua scala è pari a 1, mentre il secondo ha l'origine impostata in modo da andarsi a posizionare all'inizio del binario se questo ha scala pari a 1. I due oggetti sono stati inseriti nello script nelle variabili *slider* e *sensor*. Inoltre sono state create le variabili *instanceSlider* e *instanceSensor* per contenere le istanze di quest'ultimi. Nelle variabili *sensorLength* e *initialOffset* sono stati salvati la lunghezza del carrello e del tratto aggiuntivo, quando la scala è pari a 1, presente all'inizio e alla fine della rotaia dove il carrello non può scorrere. Le variabili *initialVector*, *firstInput* e *instantiated* servono per gestire i meccanismi per istanziare i due oggetti, le ultime due vengono settate a *false* nella funzione *Start*.

```
//Game Objects
[SerializeField] GameObject slider;
[SerializeField] GameObject sensor;

//Game Objects instances
GameObject instanceSlider;
GameObject instanceSensor;

//Sensor parameters
private float sensorLength = 0.045f;
private float initialOffset = 0.03f;

//Set parameters to instantiate GameObject
private Vector3 initialVector;
private bool firstInput;
private bool instantiated;
```

Figura 7.7: Parametri per istanziare gli oggetti.

Nella funzione *Update*, se la connessione è effettuata, si verifica la condizione *OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger)*, ovvero se viene premuto l' In-

dex Trigger del controller sinistro. Se questa è vera, si procede a salvare in *initialVector* la posizione del controller sinistro e a settare a *true firstInput*. Inoltre se *instantiated* è pari a *true*, ovvero se gli oggetti sono già istanziati, questi vengono distrutti, si impostano a false tutti i parametri che indicano la possibilità di far partire o ripetere la simulazione e si eliminano i vettori e i valori di distanza di un'eventuale simulazione salvata. In questo modo si permette di indicare il punto di partenza.

```
if (OVRInput.Get(OVRInput.Button.PrimaryIndexTrigger) && connected && subscribed)
{
    if (instantiated)
    {
        Destroy(instanceSlider);
        Destroy(instanceSensor);
        instantiated = false;
        startSimulation = false;
        repeatSimulation = false;
        lookForStart = false;
        simulationAvailable = false;
        ClearVectors();
        distances = new List<float>();
    }
    initialVector = OVRInput.GetLocalControllerPosition(OVRInput.Controller.LTouch);
    firstInput = true;
}
```

Figura 7.8: Impostazione del punto di partenza.

Per impostare il punto di arrivo, si controlla la condizione *OVRInput.Get(OVRInput.Button.PrimaryHandTrigger)*, ovvero se viene premuto l'Hand Trigger del controller sinistro, se la connessione è effettuata e se *firstInput* è pari a *true*, ovvero se è stato impostato un punto di partenza. In caso affermativo, si procede ad estrarre la posizione del controller e a salvarla sostituendo la coordinata y con quella di *initialVector*, in modo da avere il punto di partenza a quell'altezza, essendo l'angolazione nota. Si calcola la direzione nelle altre due coordinate come differenza delle due posizioni e si salva in *direction*, per poi calcolare *simulation_ max_ distance* come lunghezza del carrello, quindi modulo della direzione diviso per il coseno dell'angolo di inclinazione. Si settano *instantiated* a *true* e *firstInput* a *false* e poi si istanziano i due oggetti tramite la funzione *InstantiateGameObject*, a cui vanno passati il prefab, il punto di origine, la forward, la rotazione, un eventuale offset rispetto all'origine e la scala. Entrambi hanno come punto di origine *initialVector*, come forward *direction* e come rotazione *angle*, l'offset è 0 per la rotaia e *initialOffset * (simulation_ max_ distance - 1)* per il carrello per compensare l'espansione o compressione del blocco iniziale della rotaia con la scala, e la scala stessa è 1 per il carrello e *simulation_ max_ distance* per la rotaia.

```
//Instantiate GameObjects
if (OVRInput.Get(OVRInput.Button.PrimaryHandTrigger) && connected && subscribed && firstInput)
{
    Vector3 v = OVRInput.GetLocalControllerPosition(OVRInput.Controller.LTouch);
    v.y = initialVector.y;
    Vector3 direction = v - initialVector;

    //Calculate max distance
    simulation_max_distance = direction.magnitude / Mathf.Cos(angle * Mathf.PI / 180);

    //Instantiate slider with correct scale and rotation
    instanceSlider = InstantiateGameObject(slider, initialVector, direction, angle, 0f, simulation_max_distance);

    //Instantiate sensor with correct scale and rotation
    instanceSensor = InstantiateGameObject(sensor, initialVector, direction, angle, initialOffset * (simulation_max_distance - 1), 1f);

    instantiated = true;
    firstInput = false;
```

Figura 7.9: Impostazione del punto di arrivo.

La funzione *InstantiateGameObject*, che prende i parametri già descritti, procede con l'istanziare l'oggetto nella posizione assegnata e con la rotazione identità, per poi impostare la sua forward e ruotarlo nelle sue coordinate locali dell'angolo assegnato sull'asse x. Poi imposta la sua scala su z e sullo stesso asse trasla l'oggetto dell'offset passato, per poi ritornare l'oggetto stesso.

```
14 riferimenti
private GameObject InstantiateGameObject(GameObject obj, Vector3 position,
    Vector3 forward, float angle, float offset, float scale)
{
    GameObject instance = Instantiate(obj, position, Quaternion.identity);
    instance.transform.forward = forward;
    instance.transform.Rotate(new Vector3(angle, 0f, 0f), Space.Self);
    instance.transform.localScale = new Vector3(1, 1, scale);
    instance.transform.Translate(new Vector3(0, 0, offset));
    return instance;
}
```

Figura 7.10: Funzione InstantiateGameObject.

7.2 Simulazione

A questo punto l'utente è pronto per iniziare la simulazione dell'esperimento. Una volta che il carrello è fermo nel suo punto di partenza premendo il tasto A si innesca l'algoritmo CUSUM che individua automaticamente l'istante in cui questo inizia a muoversi. Ciò avviene perchè, come visto nel Capitolo 6, si controlla la condizione *OVRInput.Get(OVRInput.Button.One)* e, oltre alle cose già viste, se verificata la condizione si imposta come vero *lookForStart* e *sum* a 0. Al fine di far avanzare il carrello dopo l'aggiornamento del filtro di Kalman, si verifica la condizione *simulation_ distance > 0 && simulation_ distance < simulation_ max_ distance - sensorLength - 2 * initialOffset **

simulation_max_distance, ovvero che la distanza totale percorsa sia maggiore di zero e minore della lunghezza totale del carrello meno gli ostacoli iniziale e finale. Se la condizione è falsa, si ferma la simulazione impostando *startSimulation* a *false*, altrimenti si chiama la funzione *UpdatePosition* passando *dx_kalman* come parametro. In questa funzione, si trasla sulla z locale il sensore della quantità passata, tramite il comando *instanceSensor.transform.Translate(new Vector3(0, 0, dx_kalman), Space.Self)*.

Vi è inoltre la possibilità di rivedere la simulazione una volta finita premento il tasto B, per poi poter metterla in pausa premendo X e riprenderla premendo Y. Per implementare ciò sono necessarie alcune variabili, in particolare *repeatSimulation* indica se la ripetizione è in corso, *simulationAvailable* se vi è una simualazione da rivedere e *stopRepeat* se la ripetizione è in pausa. Questi tre booleani sono inizialmente tutti falsi, vi sono poi tre liste *distances*, *velocities* e *accelerations* per salvare ad ogni step di simulazione in *FixedUpdate* i valori rispettivamente di spostamento, velocità e accelerazione rilevata dall'accelerometro. Infine il parametro intero *index* serve a riposizionare correttamente il carrello.

```
//Repeat simulation parameters
private bool repeatSimulation = false;
private bool simulationAvailable = false;
private bool stopRepeat = false;
private List<float> distances;
private List<float> velocities;
private List<float> accelerations;
private int index;
```

Figura 7.11: Parametri per la ripetizione.

Le tre liste vengono inizializzate nel momento in cui l'utente preme il tasto A per indicare che il sensore è pronto, e a tutte e tre viene aggiunto uno zero come primo valore. In questo momento si imposta anche a falso *simulationAvailable*, perchè un'eventuale simulazione precedente viene cancellata. Nel momento in cui viene chiamata la funzione *UpdatePosition* per far avanzare il carrello, vengono salvati nelle rispettive liste gli stati di posizione e velocità e l'accelerazione misurata, tramite i comandi *distances.Add(simulation_distance)*, *velocities.Add(simulation_velocity)* e *accelerations.Add(accelerationX)*, se invece la simulazione è giunta al termine si imposta *simulationAvailable* a *true* per indicare che vi è una simulazione da poter rivedere e *index* a *distances.Count - 1*, per indicare che la distanza di cui traslare il carrello per riposizionarlo è l'ultima nella lista delle distanze. Nella funzione *FixedUpdate* si verifica che il tasto B sia stato premuto e che ci sia una simulazione disponibile, in tal caso si

imposta a vero *repeatSimulation* e a falso *simulationAvailable*, poi si riposiziona il sensore traslandolo all'indietro sul suo asse z della distanza presente all'indice *index* della lista delle distanze, poi si resetta *index* a 1. Per stoppare la simulazione si controlla se viene premuto il tasto X mentre la ripetizione è in corso e in caso si imposta *stopRepeat* a *true*, mentre lo stesso parametro viene impostato a *false* se viene premuto il tasto Y mentre la variabile stessa è vera.

```
//Start repeating simulation
if (OVRInput.Get(OVRInput.Button.Two) && simulationAvailable)
{
    repeatSimulation = true;
    simulationAvailable = false;
    //Reset sensor position
    instanceSensor.transform.Translate(new Vector3(0, 0, -distances[index]), Space.Self);
    index = 1;
}

//Pause repeating simulation
if(OVRInput.Get(OVRInput.Button.Three) && repeatSimulation)
{
    stopRepeat = true;
}

//Resume repeating simulation
if (OVRInput.Get(OVRInput.Button.Four) && stopRepeat)
{
    stopRepeat = false;
}
```

Figura 7.12: Innesco ripetizione, stop e ripartenza della simulazione.

Sempre nella *FixedUpdate* si verifica se allo stesso tempo è vero *repeatSimulation* e falso *stopRepeat* e in caso si effettua lo step di ripetizione, ovvero viene chiamata la funzione *UpdatePosition* con parametro *distances[index] - distances[index - 1]*, ovvero la differenza tra i due ultimi valori di posizione, poi si incrementa l'indice se questo non raggiunge *distances.Count - 1*, altrimenti si rende la simulazione disponibile e si ferma la ripetizione giunta al termine. In questo modo, avendo l'indice sempre sulla distanza corrente, si può riposizionare correttamente il sensore anche se si fa partire una nuova simulazione mentre la ripetizione corrente non è arrivata al termine. Infatti, lo stesso riposizionamento si effettua anche quando l'utente preme il tasto A.

```
//Repeat
if (repeatSimulation && !stopRepeat)
{
    UpdatePosition(distances[index] - distances[index - 1]);
    UpdateVectors(accelerations[index] * mass1 / 1000, velocities[index]);
    if(index >= distances.Count - 1)
    {
        repeatSimulation = false;
        simulationAvailable = true;
    }
    else
    {
        index++;
    }
}
```

Figura 7.13: Step di ripetizione.

7.3 Visualizzazione dei vettori

L'utente a questo punto vedrà l'esperimento evolversi con i vettori e i rispettivi valori. I vettori sono GameObject imparentati al carrello e fissati in punti specifici, la loro scala viene ad ogni update aggiornata in modo da essere proporzionale al modulo della grandezza che rappresentano. Tramite il menù a tendina presente nella prima schermata è possibile scegliere se visualizzare i vettori forza o quelli velocità.

Nella cartella Resources sono presenti due prefab per la linea del vettore e due per la freccia, i quali contengono un box di testo per scrivere il valore del modulo. Sono stati necessari due vettori perchè i prefab *vector* e *arrow* presentano un tag Force, mentre i prefab *vector* e *arrow* presentano un tag Velocity, in questo modo possono essere identificati e distinti i vettori forza e velocità. In questa prima versione si è voluto rappresentare il vettore velocità, la forza trainante calcolata tramite il modello, la componente perpendicolare alla rotaia della forza peso, la reazione vincolare, la forza derivante dall'accelerazione misurata e infine la differenza tra quest'ultima e la forza trainante calcolata. Per ognuna è stato posizionato un oggetto vuoto nel prefab del carrello sul punto in cui deve partire il vettore, e sono state create sei variabili per contenerli. Lo stesso numero di variabili è stato allocato per le istanze delle linee e delle frecce dei vettori. Infine, la variabile *showForce* inizializzata a *true* indica se mostrare i vettori della forza o della velocità.

```
//Empties for vector positioning
private GameObject startVectorAM;
private GameObject startVectorAS;
private GameObject startVectorAT;
private GameObject startVectorFUp;
private GameObject startVectorFDown;
private GameObject startVectorVel;

//Vector prefab
[SerializeField] GameObject vectorLine;
[SerializeField] GameObject vectorCap;
[SerializeField] GameObject vectorLineV;
[SerializeField] GameObject vectorCapV;

//Vector line instances
private GameObject instanceVAM;
private GameObject instanceVAS;
private GameObject instanceVAT;
private GameObject instanceVFUp;
private GameObject instanceVFDown;
private GameObject instanceVVel;

//Vector cap instances
private GameObject instanceCAM;
private GameObject instanceCAS;
private GameObject instanceCAT;
private GameObject instanceCFUp;
private GameObject instanceCFDown;
private GameObject instanceCVel;

//Bool to show vectors
private bool showForce = true;
```

Figura 7.14: Variabili per mostrare i vettori.

Nel momento in cui la rotaia e il carrello vengono istanziati, si vanno a cercare gli oggetti vuoti che indicano la posizione in cui istanziare i vettori tramite il loro nome con la funzione *GameObject.Find()*. Dopodichè, per ognuno dei vettori scelti per la visualizzazione, si istanzia il prefab della linea corrispondente al tipo di vettore (se forza o velocità) nella posizione corrispondente alla sua ancora e con angolazione, offset e scala pari a 0. La forward invece è pari a quella del carrello per la velocità e le due forze trainanti, l'opposto di essa per la differenza tra le due, il vettore *up* per la reazione vincolare e l'opposto per la componente verticale della forza peso. Ognuna di queste linee va imparentata all'oggetto vuoto corrispondente tramite il metodo *SetParent*. Con gli stessi parametri delle linee vanno istanziate le frecce, eccetto per la scala che deve essere pari a 1, poi anch'esse vanno imparentate nello stesso modo delle linee. Infine, va estratto il box di testo che esse contengono tramite la funzione *GetChild(0)* e questo va ruotato in base al tipo di vettore per essere visualizzato di fronte all'utente. Al fine di visualizzare solo i vettori che l'utente sceglie, si deve chiamare la funzione *ChangeVectors* con parametro 0 se si vogliono visualizzare le forze ($showForce = true$) e con parametro 1 per visualizzare la velocità ($showForce = false$).

```
//Find vectors anchors
startVectorAM = GameObject.Find("VectorRendererAM");
startVectorAS = GameObject.Find("VectorRendererAS");
startVectorAT = GameObject.Find("VectorRendererAT");
startVectorFUp = GameObject.Find("VectorRendererFUp");
startVectorFDown = GameObject.Find("VectorRendererFDown");
startVectorVel = GameObject.Find("VectorRendererVel");
//Instantiate vectors
instanceVAM = InstantiateGameObject(vectorLine, startVectorAM.transform.position, instanceSensor.transform.forward, 0f, 0f, 0f);
instanceVAM.transform.SetParent(startVectorAM.transform);
instanceVAS = InstantiateGameObject(vectorLine, startVectorAS.transform.position, instanceSensor.transform.forward, 0f, 0f, 0f);
instanceVAS.transform.SetParent(startVectorAS.transform);
instanceVAT = InstantiateGameObject(vectorLine, startVectorAT.transform.position, -instanceSensor.transform.forward, 0f, 0f, 0f);
instanceVAT.transform.SetParent(startVectorAT.transform);
instanceVFUp = InstantiateGameObject(vectorLine, startVectorFUp.transform.position, instanceSensor.transform.up, 0f, 0f, 0f);
instanceVFUp.transform.SetParent(startVectorFUp.transform);
instanceVFDown = InstantiateGameObject(vectorLine, startVectorFDown.transform.position, -instanceSensor.transform.up, 0f, 0f, 0f);
instanceVFDown.transform.SetParent(startVectorFDown.transform);
instanceVVel = InstantiateGameObject(vectorLineV, startVectorVel.transform.position, instanceSensor.transform.forward, 0f, 0f, 0f);
instanceVVel.transform.SetParent(startVectorVel.transform);
//Instantiate caps
instanceCAM = InstantiateGameObject(vectorCap, startVectorAM.transform.position, instanceSensor.transform.forward, 0f, 0f, 1f);
instanceCAM.transform.SetParent(startVectorAM.transform);
instanceCAM.transform.GetChild(0).Rotate(new Vector3(0, 180, 0), Space.Self);
instanceCAS = InstantiateGameObject(vectorCap, startVectorAS.transform.position, instanceSensor.transform.forward, 0f, 0f, 1f);
instanceCAS.transform.SetParent(startVectorAS.transform);
instanceCAS.transform.GetChild(0).Rotate(new Vector3(0, 180, 0), Space.Self);
instanceCAT = InstantiateGameObject(vectorCap, startVectorAT.transform.position, -instanceSensor.transform.forward, 0f, 0f, 1f);
instanceCAT.transform.SetParent(startVectorAT.transform);
instanceCFUp = InstantiateGameObject(vectorCap, startVectorFUp.transform.position, instanceSensor.transform.up, 0f, 0f, 1f);
instanceCFUp.transform.SetParent(startVectorFUp.transform);
instanceCFUp.transform.GetChild(0).Rotate(new Vector3(90, -90, 0), Space.Self);
instanceCFDown = InstantiateGameObject(vectorCap, startVectorFDown.transform.position, -instanceSensor.transform.up, 0f, 0f, 1f);
instanceCFDown.transform.SetParent(startVectorFDown.transform);
instanceCFDown.transform.GetChild(0).Rotate(new Vector3(-90, -90, 0), Space.Self);
instanceCVel = InstantiateGameObject(vectorCapV, startVectorVel.transform.position, instanceSensor.transform.forward, 0f, 0f, 1f);
instanceCVel.transform.SetParent(startVectorVel.transform);
instanceCVel.transform.GetChild(0).Rotate(new Vector3(0, 180, 0), Space.Self);
//Show only selected
if (showForce)
{
    ChangeVectors(0);
}
else
{
    ChangeVectors(1);
}
```

Figura 7.15: Inizializzazione dei vettori.

Al fine di visualizzare correttamente i vettori, al termine di ogni step di simulazione o di ripetizione è necessario chiamare la funzione *UpdateVectors*, passando come parametri l'accelerazione misuarata moltiplicata per la massa e divisa per 1000 (poichè la massa è salvata in grammi) e la velocità. In questa funzione si va ad impostare il valore della forza da rappresentare diviso per 20 (in modo da normalizzare il vettore e non visualizzarlo eccessivamente lungo) come scala sull'asse z delle linee e come posizione locale sull'asse z delle frecce. Poi i valori vengono impostati come stringa nel box di testo corrispondente di ogni freccia. Il valore di *simulation_force* e *simulation_force_up*, ovvero della forza trainante calcolata dal modello e della reazione vincolare (pari alla componente perpendicolare della forza peso), essendo costante, viene calcolato all'inizio della simulazione rispettivamente come *simulation_acceleration * mass1 /1000* e *mass1 * 9.81f * Mathf.Cos((angle * Mathf.PI) / 180) / 1000*.

```
2 riferimenti
private void UpdateVectors(float forceM, float velocity)
{
    float AM = forceM / 20;
    float AS = simulation_force / 20;
    float FUp = simulation_force_up / 20;
    float Vel = velocity / 20;
    instanceVAM.transform.localScale = new Vector3(1, 1, AM);
    instanceVAS.transform.localScale = new Vector3(1, 1, AS);
    instanceVAT.transform.localScale = new Vector3(1, 1, AS - AM);
    instanceVFUp.transform.localScale = new Vector3(1, 1, FUp);
    instanceVFDown.transform.localScale = new Vector3(1, 1, FUp);
    instanceVVel.transform.localScale = new Vector3(1, 1, Vel);
    instanceCAM.transform.localPosition = new Vector3(0, 0, AM);
    instanceCAS.transform.localPosition = new Vector3(0, 0, AS);
    instanceCAT.transform.localPosition = new Vector3(0, 0, -(AS - AM));
    instanceCFUp.transform.localPosition = new Vector3(0, FUp, 0);
    instanceCFDown.transform.localPosition = new Vector3(0, -FUp, 0);
    instanceCVel.transform.localPosition = new Vector3(0, 0, Vel);
    instanceCAM.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round(forceM * 100f) / 100f).ToString();
    instanceCAS.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round(simulation_force * 100f) / 100f).ToString();
    instanceCAT.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round((simulation_force - forceM) * 100f) / 100f).ToString();
    instanceCFUp.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round((simulation_force_up) * 100f) / 100f).ToString();
    instanceCFDown.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round((-simulation_force_up) * 100f) / 100f).ToString();
    instanceCVel.GetComponentInChildren<TMPro.TextMeshPro>().text =
        (Mathf.Round((velocity) * 100f) / 100f).ToString();
}
```

Figura 7.16: Funzione UpdateVectors.

Nella Figura 7.17 si può osservare il modo in cui viene mostrato l'esperimento, con rotaia, carrello e vettori (in questo caso le forze).

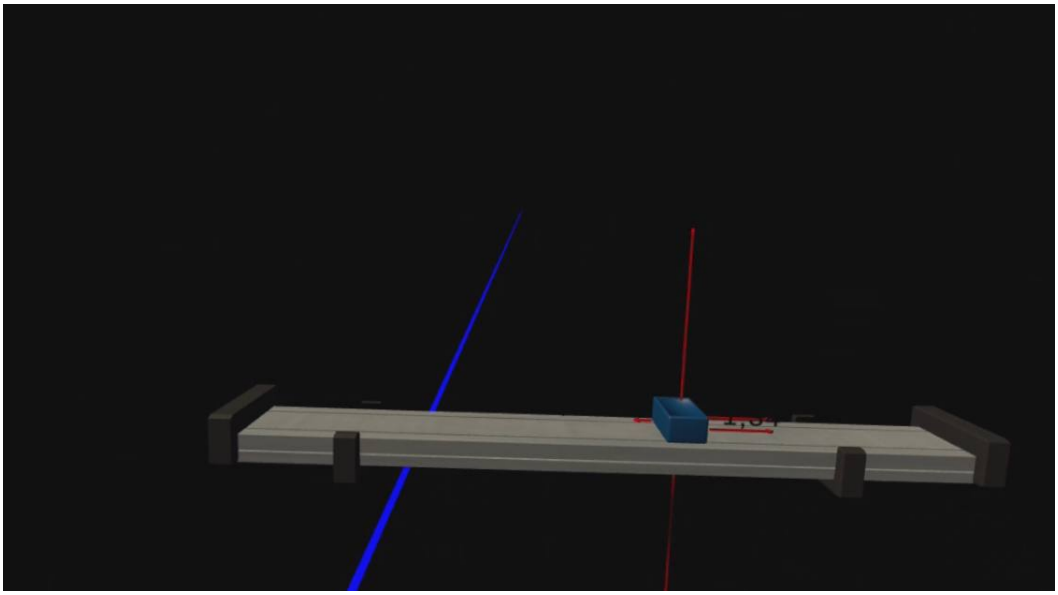


Figura 7.17: Visualizzazione dell'esperimento.

La funzione *ChangeVectors*, chiamata al variare del valore del menù a tendina, procede con il settare il valore di *showForce* a *true* se il parametro in ingresso è 0, o a *false* se il parametro è 1. Dopodiche, si cercano tramite la funzione *GameObject.FindGameObjectsWithTag()* i componenti taggati Velocity e quelli taggati Force, e si procede a cercare il loro componente MeshRenderer tramite la funzione *GetComponentsInChildren<MeshRenderer>()*. Se si vogliono visualizzare i vettori forza, basterà disattivare questo componente nei vettori velocità e attivarlo nei vettori forza, e viceversa se si vogliono mostrare i vettori velocità.

```
public void ChangeVectors(int value)
{
    if (value == 0)
    {
        showForce = true;
        GameObject[] gameObjects = GameObject.FindGameObjectsWithTag("Velocity");
        foreach (GameObject target in gameObjects)
        {
            MeshRenderer[] m = target.GetComponentsInChildren<MeshRenderer>();
            foreach (MeshRenderer mesh in m)
            {
                mesh.enabled = false;
            }
        }
        GameObject[] gameObjects1 = GameObject.FindGameObjectsWithTag("Force");
        foreach (GameObject target in gameObjects1)
        {
            MeshRenderer[] m = target.GetComponentsInChildren<MeshRenderer>();
            foreach (MeshRenderer mesh in m)
            {
                mesh.enabled = true;
            }
        }
    }
    else
    {
        showForce = false;
        GameObject[] gameObjects = GameObject.FindGameObjectsWithTag("Velocity");
        foreach (GameObject target in gameObjects)
        {
            MeshRenderer[] m = target.GetComponentsInChildren<MeshRenderer>();
            foreach (MeshRenderer mesh in m)
            {
                mesh.enabled = true;
            }
        }
        GameObject[] gameObjects1 = GameObject.FindGameObjectsWithTag("Force");
        foreach (GameObject target in gameObjects1)
        {
            MeshRenderer[] m = target.GetComponentsInChildren<MeshRenderer>();
            foreach (MeshRenderer mesh in m)
            {
                mesh.enabled = false;
            }
        }
    }
}
```

Figura 7.18: Funzione ChangeVectors.

Conclusioni e sviluppi futuri

In conclusione, lo sviluppo affrontato in questa tesi ha raggiunto una fase cruciale, culminando in un'applicazione completamente funzionante in grado di riprodurre l'esperimento selezionato. Questo fornisce agli studenti l'opportunità di comprendere in modo più approfondito i concetti coinvolti nell'esperimento. L'applicazione offre la possibilità di gestire i parametri dell'esperimento e visualizzare le grandezze fisiche coinvolte in modo interattivo. Questa caratteristica non solo rende l'apprendimento più coinvolgente ma anche più immediato, consentendo agli studenti di esplorare la teoria e la pratica in modo integrato. In particolare, la capacità di confrontare le grandezze attese con quelle misurate arricchisce la comprensione, mettendo in luce le differenze tra la teoria e la realtà sperimentale.

Prospettando il futuro sviluppo dell'applicazione, è evidente che esistono molteplici direzioni. Una fase di testing più lunga e accurata è essenziale per individuare miglioramenti potenziali e risolvere eventuali bug residui. Inoltre, una modellazione più dettagliata degli attriti può contribuire a rendere il modello dell'esperimento ancora più preciso. Tra le possibilità di sviluppo futuro, vi è l'espansione dell'applicazione su diversi dispositivi, come tablet, cellulari o altri visori VR che permettano la cattura del frame buffer, consentendo un accesso più ampio al suo utilizzo. Questa diversificazione potrebbe incorporare il già sviluppato image tracking e integrarne i dati nel filtro di Kalman, migliorando ulteriormente la precisione della simulazione. Infine, l'evoluzione futura potrebbe concentrarsi su esperimenti diversi, estendendo l'applicazione a nuovi scenari educativi. Ad esempio, lo studio del moto di corsa di una persona potrebbe rappresentare un nuovo ambito di esplorazione, utilizzando gli stessi strumenti per approfondire concetti differenti.

Bibliografia

[1] M. D. Mori. *Realtà virtuale, aumentata o mista?* https://www.focus.it/tecnologia/digital-life/realta-virtuale-aumentata-o-mista.

[2] Case Western Reserve University. *HoloAnatomy.* https://case.edu/holoanatomy.

[3] Mit Media Lab. *Electrostatic Playground: A multi-user virtual reality physics learning experience.* https://www.media.mit.edu/projects/vr-physics-lab/overview.

[4] Harvard University LIT. *AR for Physics 1-1 Tutoring.* https://lit.gse.harvard.edu/ar-physics-1-1-tutoring.

[5] I. Radu, X. Huang, G. Kestin, Z. Shah e B. Schneider. *How Augmented Reality Transforms Learning and Inquiry Patterns in 1-1 Remote Tutoring.* https://par.nsf.gov/servlets/purl/10365843.

[6] HTML.it. *Guida Unity3D.* https://www.html.it/guide/guida-unity-3d.

[7] Meta. *Oculus Quest 2.* https://www.meta.com/it/en/quest/products/quest-2/tech-specs.

[8] Meta. *Oculus Integration.* https://developer.oculus.com/reference/unity/v57.

[9] Meta. *Get started with Passthrough.* https://developer.oculus.com/documentation/unity/unity-passthrough-gs.

[10] Unity. *ARFoundation Manual.* https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.1/manual/index.html.

[11] Unity. *Google ARCore XR Plug-in Manual.* https://docs.unity3d.com/Packages/com.unity.xr.arcore@5.1/manual/index.html.

[12] Google. *ARCore Documentation.* https://developers.google.com/ar/develop/augmented-images?hl=it.

[13] JacksiroKe. *arcoreimg*. https://github.com/JacksiroKe/arcoreimg.

[14] E. Olson. *AprilTag: A robust and flexible visual fiducial system*. https://april.eecs.umich.edu/media/pdfs/olson2011tags.pdf.

[15] J. Wang e E. Olson. *AprilTag 2: Efficient and robust fiducial detection*. https://april.eecs.umich.edu/media/pdfs/wang2016iros.pdf.

[16] keijiro. *jp.keijiro.apriltag - AprilTag package for Unity*. https://github.com/keijiro/jp.keijiro.apriltag.

[17] Arduino. *Nano 33 BLE Sense Rev2*. https://docs.arduino.cc/hardware/nano-33-ble-sense-rev2.

[18] Arduino. *Nano 33 BLE Sense Rev2 Datasheet*. https://docs.arduino.cc/resources/datasheets/ABX00069-datasheet.pdf.

[19] Arduino. *Arduino_BMI270_BMM150*. https://www.arduino.cc/reference/en/libraries/arduino_bmi270_bmm150.

[20] CQRobot. *VL53L1X Distance Ranging Sensor*. http://www.cqrobot.wiki/index.php/VL53L1X_Distance_Ranging_Sensor_SKU:_CQRWX00744US.

[21] Polou. *VL53L1X*. https://www.arduino.cc/reference/en/libraries/vl53l1x.

[22] Arduino. *Wire*. https://www.arduino.cc/reference/en/language/functions/communication/wire.

[23] Arduino. *ArduinoBLE*. https://www.arduino.cc/reference/en/libraries/arduinoble.

[24] Velorexe. *Unity Android Bluetooth Low Energy*. https://github.com/Velorexe/Unity-Android-Bluetooth-Low-Energy.

[25] P. Mazzoldi, M. Nigro e C. Voci. *Fisica Volume I*. EdiSES, 2001, pp. 9–10.

[26] P. Mazzoldi, M. Nigro e C. Voci. *Fisica Volume I*. EdiSES, 2001, pp. 12–16.

[27] P. Mazzoldi, M. Nigro e C. Voci. *Fisica Volume I*. EdiSES, 2001, pp. 53–56.

[28] Greg Czerniak. *Kalman Guide*. https://greg.czerniak.info/guides/kalman1.

[29] MIT. *Runge-Kutta Methods*. https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html.

[30] Sarem Seitz. *A probabilistic version of CUSUM*. https://www.sarem-seitz.com/posts/probabilistic-cusum-for-change-point-detection.html#a-probabilistic-version-of-cusum.