ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Ingegneria e Scienze Informatiche

Simulazioni Fisiche in Mixed Reality: Progettazione e Realizzazione di un Laboratorio Scientifico Interattivo

Tesi di laurea in:
Computer Graphics

Relatore
Prof. Damiana Lazzaro

Candidato
Luigina Annamaria Mennuti

Sessione Unica
Anno Accademico 2024 - 2025

Sommario

Fenomeni solitamente confinati ai laboratori diventano esperienze accessibili nello spazio reale grazie a un sistema che unisce simulazione fisica, modellazione digitale e Mixed Reality. Modelli tridimensionali realizzati in Blender prendono forma in Unity e vengono collocati nell'ambiente fisico con precisione centimetrica tramite il Visual Positioning System di Niantic Lightship, trasformando superfici comuni in scenari scientifici dinamici.

L'analisi comparativa tra diversi metodi di integrazione, quali Eulero, Verlet e Runge–Kutta 4, mette in evidenza differenze significative in termini di stabilità ed errore energetico, evidenziando come RK4 risulti particolarmente preciso nelle condizioni adottate e come Verlet mantenga una buona coerenza energetica nei sistemi a lungo termine. Le simulazioni ottiche, basate sulla geometria e sulla legge di Snell, completano l'ambiente con una visualizzazione immediata dei fenomeni luminosi.

La combinazione tra fisica computazionale, modellazione 3D e realtà aumentata permette di osservare e manipolare concetti complessi in modo diretto e immersivo. L'esperienza risultante dimostra il potenziale della Mixed Reality come strumento intuitivo ed efficace per la comprensione e la comunicazione dei fenomeni scientifici.

Indice

Elenco delle figure

List of Listings

Introduzione

La diffusione di conoscenze scientifiche complesse richiede l'uso di strumenti innovativi, interattivi e immersivi, in grado di rendere accessibili anche fenomeni astratti o lontani dall'esperienza quotidiana. Le tecnologie di realtà mista esprimono un grande potenziale per superare questi limiti. La didattica tradizionale incontra grandi difficoltà quando ci si confronta con fenomeni troppo grandi, piccoli o lontani nel tempo e nello spazio per essere percepiti direttamente; in questi casi, diventa molto utile ricorrere a simulazioni virtuali che eliminano queste barriere.
In ambienti virtuali tridimensionali, è possibile imparare attraverso l'interazione con oggetti e simulazioni, agendo sulla realtà circostante e osservando come essa reagisca. In questo modo, l'apprendimento diventa esperienziale, avvicinandosi alle metodologie dell'Inquiry-Based Science Education (IBSE). Ad esempio, la realtà aumentata può consentire di visualizzare in modo pratico processi chimici o fisici complessi, rendendo tangibili concetti altrimenti astratti. Le tecnologie di Mixed Reality, inoltre, mostrano un grande potenziale per superare i limiti dei laboratori tradizionali; mentre i laboratori fisici devono affrontare vincoli di spazio, costi elevati e rischi legati a materiali pericolosi, un ambiente MR su dispositivi mobili consente di generare istantaneamente simulazioni sperimentali ovunque.

Lo scopo principale di questo progetto di tesi è realizzare un laboratorio scientifico interattivo in Mixed Reality che permetta agli studenti di esplorare in modo intuitivo i concetti fondamentali di fisica. È pensata per essere fruibile tramite l'uso di dispositivi mobili. Utilizzare uno smartphone come piattaforma di MR garantisce un'ampia accessibilità a basso costo, sia rispetto alle apparecchiature specializzate dei laboratori tradizionali, sia rispetto ai visori di ultima generazione.

Si intende facilitare la comprensione di leggi altrimenti solo teoriche:

- il fenomeno del Pendolo di Foucault, che illustra il moto della rotazione terrestre;
- un ambiente gravitazionale interattivo, soggetto a diverse accelerazioni e modelli di resistenza;
- la dinamica del Sistema Solare, per comprendere le distanze e le proporzioni astronomiche;
- i fenomeni di ottica con il prisma, quali rifrazione e dispersione della luce.

Si rende, inoltre, possibile interagire con modelli virtuali sovrapposti al mondo reale e manipolare parametri come massa, velocità o angoli, osservando in tempo reale gli effetti di queste variazioni.

La realizzazione tecnica di questi modelli è stata affidata al motore grafico Unity, integrato con l'Augmented Reality Developer Kit (ARDK) di Niantic Lightship, che estende le capacità AR fondamentali di Unity. Grazie al Visual Positioning System di Niantic, è possibile ancorare stabilmente i contenuti virtuali allo spazio fisico con precisione centimetrica, consentendo di creare simulazioni affidabili e ripetibili anche in ambienti aperti o poco strutturati. L'applicazione raccoglie le misure ambientali tramite la fotocamera e i sensori del dispositivo, costruisce una mappa tridimensionale della scena e colloca le simulazioni fisiche in modo persistente nello spazio reale.

L'aspetto di originalità del progetto risiede proprio nell'integrazione di queste simulazioni fisiche con una tecnologia MR avanzata su dispositivi mobili. Sebbene esistano lavori simili nel campo dell'educazione scientifica, il contributo di questo progetto innova sotto due profili:

- da un lato, la sinergia con Niantic Lightship permette un'accuratezza spaziale finora poco esplorata in ambito didattico
- dall'altro lato, l'intero sistema è progettato per funzionare su smartphone comuni, realizzando così un laboratorio scientifico interattivo portatile e a basso costo rispetto ai laboratori tradizionali.

In questo modo si apre la strada a nuovi scenari di didattica immersiva, dove l'apprendimento basato sulla scoperta e sulla manipolazione diretta diventa accessibile in contesti scolastici ordinari.

La tesi è organizzata nei seguenti capitoli:

- **Capitolo 1** - *Fondamenti di Mixed Reality*, introduce i principi teorici alla base della realtà mista, delineando le differenze con realtà virtuale e aumentata e analizzandone le potenzialità nel campo dell'apprendimento scientifico e della divulgazione interattiva;
- **Capitolo 2** – *Strumenti e pipeline di sviluppo*, illustra il flusso di lavoro adottato per la costruzione del progetto, descrivendo l'utilizzo combinato di Blender per la modellazione e ottimizzazione dei modelli 3D, Unity per la gestione della logica interattiva e Niantic Lightship ARDK per l'ancoraggio spaziale preciso degli oggetti digitali nel mondo reale;
- **Capitolo 3** - *Modelli matematici e simulazione fisica*, approfondisce i fondamenti teorici e numerici che guidano la rappresentazione dei fenomeni simulati. Vengono introdotte le equazioni che descrivono il Pendolo di Foucault, il GraviLab, il Sistema Solare e il Prisma Ottico, seguite dall'analisi dei principali metodi di integrazione — Eulero, Verlet e Runge–Kutta 4 — impiegati per ottenere una simulazione stabile e coerente nel tempo;
- **Capitolo 4** - *Implementazione e architettura del sistema*, descrive l'organizzazione del codice, la struttura modulare e i design pattern utilizzati per lo sviluppo in Unity, illustrando nel dettaglio l'implementazione dei quattro esperimenti principali. Ogni sezione mostra come i modelli fisici e matematici siano stati tradotti in logica interattiva, generando esperienze che coniugano rigore scientifico e immersività visiva.

Questo lavoro vuole contribuire alla divulgazione scientifica innovativa, sfruttando le potenzialità della Mixed Reality per rendere l'apprendimento della fisica un'esperienza coinvolgente ed efficace anche su dispositivi di largo consumo.

Capitolo 1

Fondamenti di Mixed Reality

La Mixed Reality (MR) è un'innovativa forma di interazione tra il mondo fisico e quello digitale. In MR, gli oggetti virtuali possono essere sovrapposti e ancorati all'ambiente reale, permettendo interazioni in tempo reale tra elementi digitali e fisici. Il concetto di Mixed Reality appare già negli anni '90 nel "continuum di virtualità" di *Milgram*, secondo cui la realtà fisica e virtuale sono due estremi di uno spettro continuo. In altre parole, la MR include qualsiasi esperienza immersiva che possa miscelare sapientemente la realtà aumentata e la realtà virtuale [1, 2].

- Realtà Aumentata (AR): arricchisce il mondo reale con elementi digitali sovrapposti all'ambiente fisico. L'utente vede principalmente il mondo reale, ma con informazioni virtuali aggiunte;
- Realtà Virtuale (VR): immerge completamente l'utente in un mondo virtuale, oscurando il mondo fisico e consentendo di interagire solo con un mondo digitale creato artificialmente;
- Mixed Reality (MR): combina gli elementi della AR e della VR. Consente la coesistenza e l'interazione in tempo reale di oggetti fisici e virtuali [3].

Secondo l'XR Association, associazione di settore AR/VR/MR, il 2024 ha visto una grande crescita nell'adozione di soluzioni immersive, con significativi progressi hardware e una sempre maggiore presenza in settori come salute, istruzione e intrattenimento. Vi sono tecnologie emergenti che includono l'integrazione di intelligenza artificiale, ad esempio, per il riconoscimento contestuale e la generazione

di contenuti in Mixed Reality in tempo reale, cloud computing a bassa latenza per lo streaming 3D e la standardizzazione del software. È necessario citare anche il concetto di metaverso e social MR, che sta spingendo sul lato collaborativo, con piattaforme virtuali e ambienti condivisi [4, 5].

Nell'istruzione STEM, le potenzialità della realtà mista emergono in modo evidente. Uno studio su studenti della scuola primaria ha mostrato come le applicazioni in MR aumentino la motivazione allo studio della scienza e favoriscano il lavoro collaborativo, grazie all'immersività e all'interazione intuitiva. Anche in contesti universitari, la MR è impiegata come piattaforma a basso costo per i laboratori di chimica e biologia. Ad esempio, il sistema HOVR Lab consente di eseguire un esperimento reale utilizzando strumenti fisici marcati all'interno di un visore [6]. In questo sistema, i sensori tracciano strumenti reali, fornendo un feedback tattile e visivo autentico. Questo approccio è risultato efficace e meno costoso di un laboratorio tradizionale, offrendo la reale esperienza di un laboratorio anche a distanza [7]. La Mixed Reality trova impiego anche nella formazione degli insegnanti. Un articolo di didattica della fisica ha sperimentato un simulatore di classe per preparare i docenti al dialogo pedagogico. I ricercatori hanno fatto esercitare gli assistenti alla didattica in un'aula virtuale, in cui questi ultimi potevano porre domande agli studenti avatar e ricevere un feedback sull'efficacia del loro insegnamento. I risultati hanno mostrato come l'uso intensivo del simulatore MR migliori le abilità di interrogazione degli studenti rispetto a una formazione tradizionale, evidenziando il potenziale di questa tecnologia nel training didattico [8].
In ambito scientifico, più in generale, esistono esempi che vanno dalla visualizzazione anatomica in 3D, come il progetto HoloAnatomy in medicina, alla simulazione di macchine complesse in ingegneria [9]. Il comune denominatore è sempre l'interazione immersiva: la Mixed Reality permette di manipolare modelli 3D di sistemi scientifici reali come se fossero davanti a noi, facilitando la comprensione di concetti astratti [5].

Capitolo 2

Strumenti e pipeline di sviluppo

L'exhibit scientifico è un sistema interattivo concepito per permettere agli utenti di esplorare fenomeni fisici complessi attraverso esperienze visive e manipolabili. Il progetto integra simulazioni di meccanica, astronomia e ottica, come il pendolo di Foucault, la caduta dei corpi, il Sistema Solare e la rifrazione della luce, offrendo un laboratorio digitale che unisce rigore scientifico e coinvolgimento esperienziale. Per la sua realizzazione, sono state integrate diverse tecnologie, ognuna delle quali con un ruolo specifico, dalla modellazione tridimensionale alla simulazione fisica e alla successiva implementazione in Mixed Reality. Il processo si è quindi articolato lungo tre assi principali:

- la modellazione 3D, realizzata in Blender
- la simulazione fisica e interattiva, gestita in Unity
- la realtà aumentata e l'ancoraggio spaziale, tramite Niantic Lightship

2.1 Blender - Modellazione e ottimizzazione dei modelli 3D

Come strumento principale per la creazione dei modelli 3D, è stato utilizzato Blender, software open-source per la creazione grafica che offre un ambiente unico in cui svolgere tutte le fasi della produzione. Questo permette di modellare, texturizzare, animare e preparare la scena per l'export senza dover trasferire i dati tra

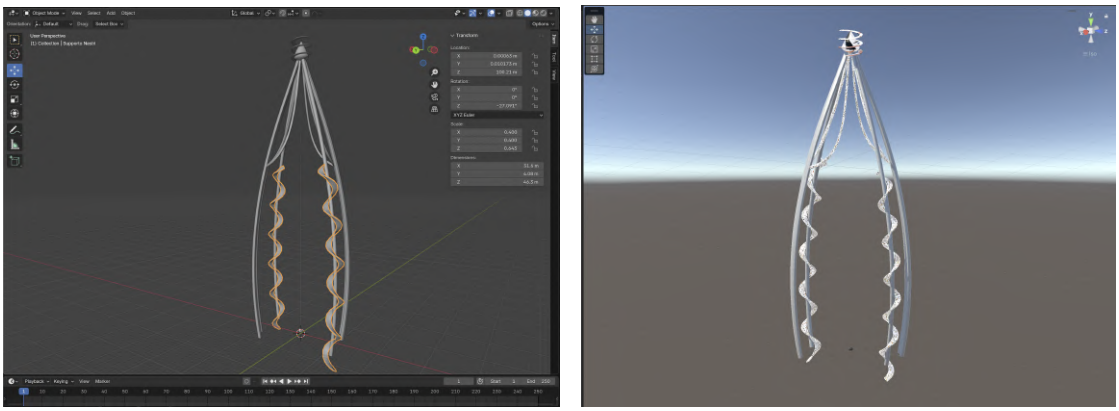
diverse applicazioni, evitando di introdurre problemi di compatibilità e garantendo coerenza nel risultato finale.

Pendolo

La struttura principale del pendolo di Foucault rappresenta l'elemento architettonico che sostiene il sistema oscillante. Si compone di quattro archi metallici curvilinei che convergono nella parte superiore, formando un vertice a cui è collegato il filo del pendolo. Ogni arco è stato generato a partire da una Bezier Curve, modellata manualmente per ottenere una forma proporzionata rispetto alla dimensione del pendolo. Le spirali che avvolgono gli archi sono state create con una combinazione di:

- Screw Modifier, per creare la forma elicoidale
- Curve Modifier che deforma la spirale lungo la traiettoria dell'arco corrispondente
- Solidify Modifier, per conferire spessore alla superficie del nastro

Infine, il Mirror Modifier ha permesso un disegno uniforme e simmetrico.

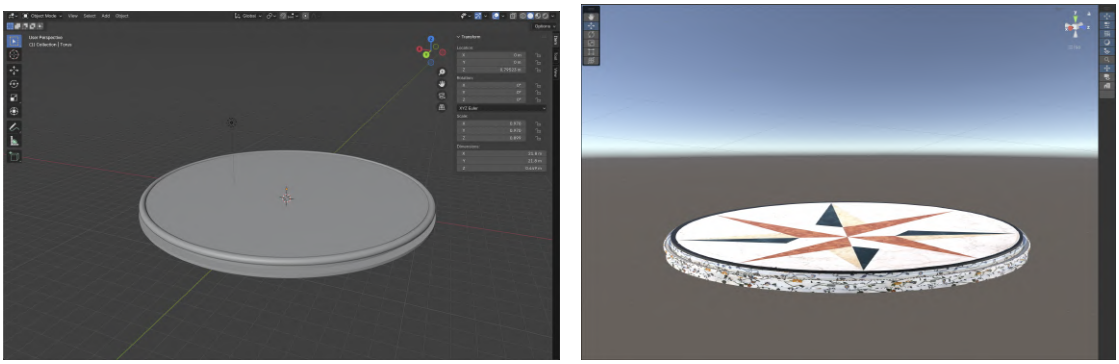


(a) Struttura in Solid Mode (b) Struttura con materiali

Figura 2.1: Struttura principale del Pendolo di Foucault

La base del pendolo di Foucault funge da supporto visivo associato al moto rotatorio terrestre. Ospita al suo interno il sistema di illuminazione che rappresenta la

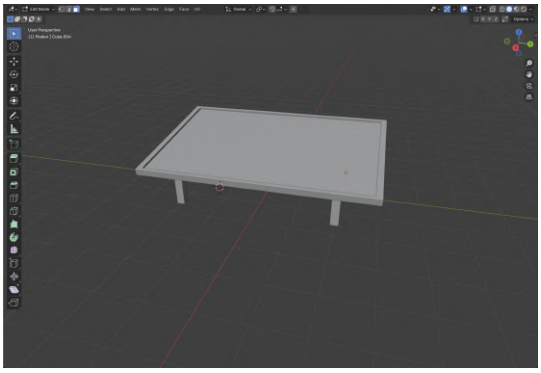
rotazione della Terra nel corso delle 24 ore.
Il disco principale costituisce la base di appoggio e ospita i led lungo il bordo. L'anello di led è stato generato attraverso un Array Modifier, che ne garantisce uniformità ed equidistanza, distribuendo le copie lungo una rotazione attorno a un Empty Object centrale. Una Torus Mesh rappresenta la teca di vetro protettiva per i led.



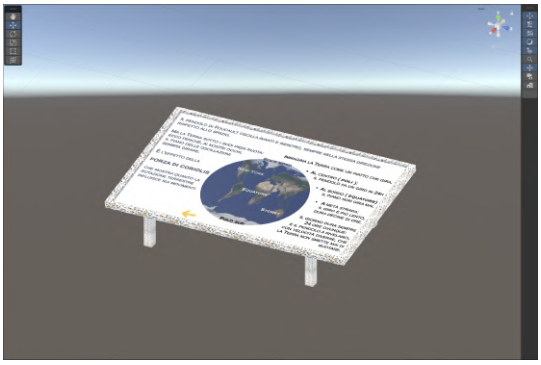
(a) Base in Solid Mode (b) Base con materiali

Figura 2.2: Base del Pendolo di Foucault

Accanto alla struttura rotante, è stato realizzato un tavolino informativo che completa la scena. La struttura è composta da un piano rettangolare, leggermente incassato in una cornice esterna, realizzato ridimensionando e rifinendo tramite Inset ed Extrude una Cube Mesh. Le quattro gambe sono modellate in prossimità dei vertici inferiori del tavolo.



(a) Desk informativo in Solid Mode

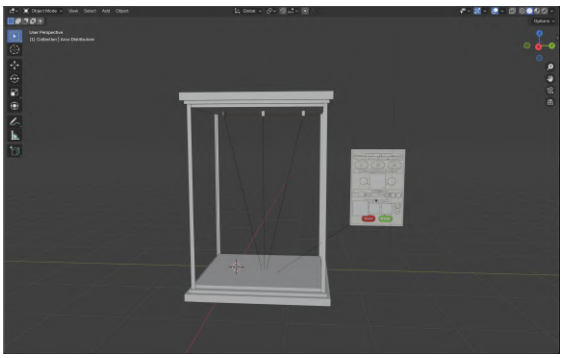


(b) Desk informativo con materiali

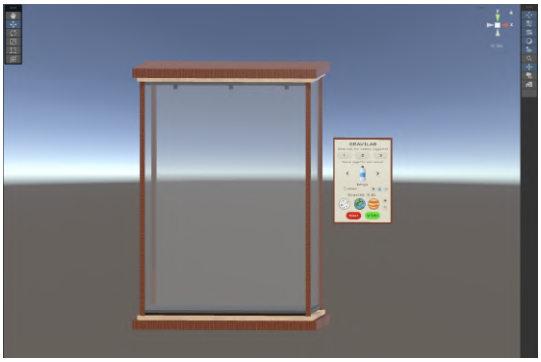
Figura 2.3: Desk informativo del Pendolo di Foucault

Le medesime tecniche di modellazione e ottimizzazione sono state successivamente applicate a tutti gli altri elementi dell'exhibit, realizzati con lo stesso approccio parametrico e integrati nella scena. Questi modelli, pur variando per funzione e complessità, condividono la stessa attenzione alla scala, alla pulizia delle mesh e alla compatibilità con il motore grafico. Tutte le scale sono state applicate in metri reali e i modelli esportati mantengono il pivot alla base della struttura per il corretto allineamento nella scena Unity in cui saranno importati.

GraviLab



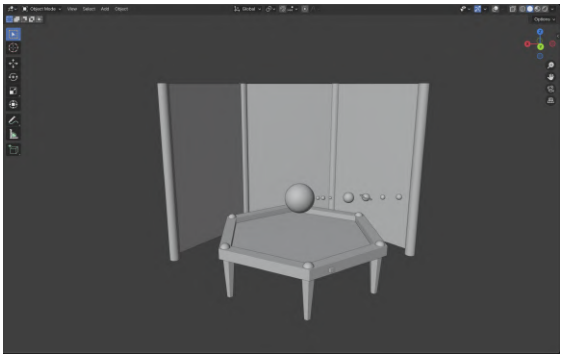
(a) Teca espositiva in Solid Mode



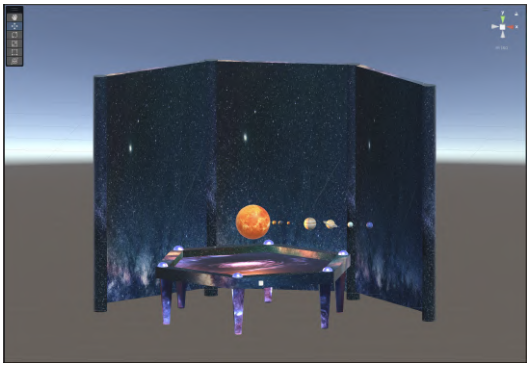
(b) Teca espositiva con materiali

Figura 2.4: Teca espositiva con lavagna interattiva

Sistema Solare



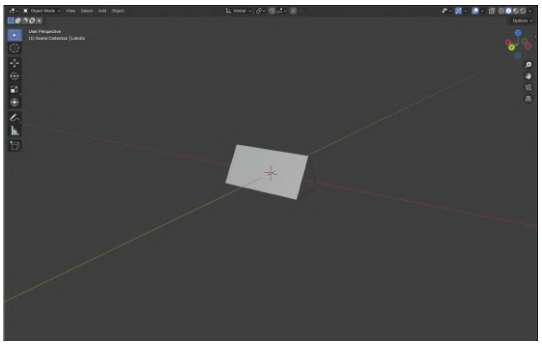
(a) Sistema solare in Solid Mode



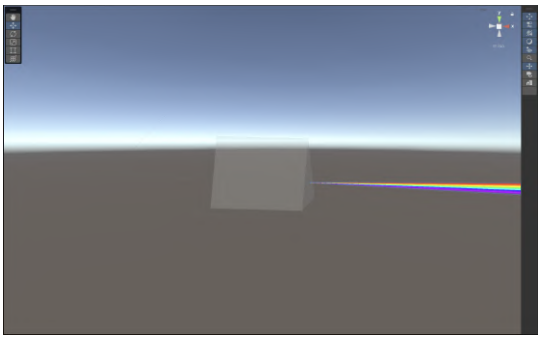
(b) Sistema solare con materiali

Figura 2.5: Sistema solare

Prisma Ottico



(a) Prisma ottico in Solid Mode



(b) Prisma ottico con materiali

Figura 2.6: Prisma ottico

Modellazione del personaggio

È stato introdotto un personaggio tridimensionale con la funzione di elemento guida e di connessione narrativa tra le diverse parti dell'esperienza. La parte più significativa del processo di modellazione ha riguardato la realizzazione della capigliatura, elemento che ha richiesto particolare attenzione sia dal punto di vista

estetico che tecnico. L'obiettivo era ottenere un risultato credibile e coerente con lo stile stilizzato del personaggio, pur mantenendo un numero di poligoni compatibile con la visualizzazione in tempo reale. La modellazione dell'acconciatura del personaggio si è svolta utilizzando il nuovo sistema di hair grooming basato sulle curve. Il sistema di Hair Curves consente di trattare i capelli come insiemi di curve speciali, anziché come particelle tradizionali, e Geometry Nodes per generare e modificare proceduralmente le chiome. Blender mette a disposizione strumenti di grooming utili per modellare i capelli curva per curva. È possibile pettinare o spostare le ciocche (tramite lo strumento *Comb*), tirare una ciocca (*Snake Hook*), far crescere e accorciare i capelli (*Grow / Shrink*), avvicinare o aggregare ciocche (*Pinch*), dare volume (*Puff*), lisciare (*Smooth*) e altri [10], permettendo un controllo diretto sulle forme e plasmando ogni dettaglio.

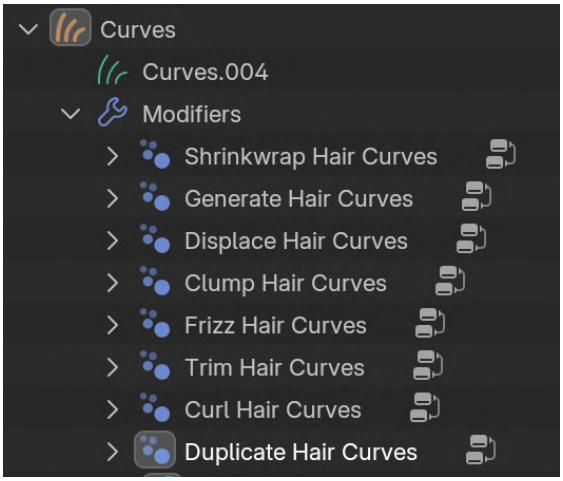


Figura 2.7: Stack dei modificatori applicati per la generazione dei capelli.

Ogni nodo della catena contribuisce a definire la forma, la densità e il comportamento delle ciocche. La sequenza logica con cui sono stati generati e trasformati i capelli del personaggio agisce in modo non distruttivo sulla geometria delle curve, introducendo un progressivo livello di dettaglio. L'ordine con cui vengono applicati è fondamentale: i modificatori di generazione e deformazione devono precedere quelli di rifinitura e duplicazione per garantire coerenza geometrica e stabilità del risultato. Descrivendo in ordine i modificatori usati per ottenere l'effetto finale:

- **Shrinkwrap Hair Curves**

Figura 2.8: Effetto del modificatore *Shrinkwrap*

Ha la funzione di creare un'ancora tra i capelli e la superficie della testa. Permette di gestire la relazione tra la mesh del cuoio capelluto e il punto di origine delle ciocche, assicurando che ogni capello "nasca" esattamente dalla base della testa. Senza questo modificatore, le radici potrebbero intersecarsi con la mesh sottostante, causando artefatti e deformazioni.

- **Generate Hair Curves**

Figura 2.9: Effetto del modificatore *Generate*

Gestisce la creazione delle ciocche che costituiscono il volume principale della capigliatura, controllando la densità, la distribuzione e la lunghezza. Il modificatore genera in modo procedurale migliaia di ciocche interpolate, creando una base densa e uniforme.

- **Displace Hair Curves**

Figura 2.10: Effetto del modificatore *Displace*

Introduce leggere variazioni casuali e microspostamenti per spezzare la regolarità prodotta dal nodo precedente. L'obiettivo è evitare un effetto artificiale e statico, aggiungendo irregolarità locali che rendono i capelli più naturali e credibili in fase di rendering.

- **Clump Hair Curves**

Figura 2.11: Effetto del modificatore *Clump*

Raggruppa più ciocche in ciuffi. Nei capelli reali, le fibre tendono a muoversi insieme formando gruppi coerenti e il *Clump* riproduce questo comportamento naturale. Ogni gruppo converge verso una "ciocca madre", generando variazioni di volume e coesione nella chioma.

- **Frizz Hair Curves**

Figura 2.12: Effetto del modificatore *Frizz*

Aggiunge ondulazioni e increspature casuali alle ciocche. Questo modificatore serve per creare l'effetto di capelli mossi o ricci e viene applicato dopo il *Clump* per mantenere la struttura dei ciuffi, introducendo però un leggero disordine interno.

- **Trim Hair Curves**

Figura 2.13: Effetto del modificatore *Trim*

Permette di regolare la lunghezza globale delle ciocche in modo uniforme. Essendo un modificatore non distruttivo, la lunghezza può essere modificata liberamente in fase di iterazione senza alterare le curve originali. È stato utilizzato per bilanciare la forma generale della capigliatura dopo le deformazioni introdotte dai nodi precedenti.

- **Curl Hair Curves**

Figura 2.14: Effetto del modificatore *Curl*

Definisce la curvatura e la torsione lungo le ciocche, controllando parametri come ampiezza, frequenza e direzione della spirale. Nel progetto, è stato impiegato per ottenere l'effetto riccio della chioma, con torsioni più compatte alla radice e più morbide verso le punte.

- **Duplicate Hair Curves**

Figura 2.15: Effetto del modificatore *Duplicate*

Duplica le ciocche generate in precedenza per aumentare la densità visiva della capigliatura. Le copie vengono leggermente spostate o ruotate per simulare variazioni direzionali e migliorare la copertura volumetrica. È fondamentale collocarlo alla fine della pila per evitare che i duplicati vengano deformati in modo incoerente dai modificatori precedenti.

L'ordine di applicazione è cruciale perchè i modificatori vengono calcolati sequenzialmente dall'alto verso il basso nella pila. I primi (*Shrinkwrap, Generate*) definiscono la base strutturale, i successivi (*Displace, Clump, Frizz*) modellano la variazione e la forma, e gli ultimi (*Trim, Curl, Duplicate*) rifiniscono la forma finale e il *Duplicate* ottimizza la resa visiva.

Una volta completata la fase di modellazione, il personaggio è stato rifinito attraverso l'applicazione di materiali e texture, gestite in parte tramite Geometry Nodes e in parte tramite il Texture Editor fornito da Blender.

Figura 2.16: Personaggio modellato

2.2 Unity – Ambiente di simulazione

Unity è un motore grafico multi-piattaforma ampiamente diffuso per lo sviluppo in tempo reale di contenuti interattivi, giochi e applicazioni AR/VR. Si basa su scripting in C# e offre un'ampia serie di tool integrati, quali motori grafici avanzati, come il rendering 3D e l'illuminazione fisicamente plausibile, e fisica, Nvidia PhysX, per simulazioni dinamiche. Unity è utilizzato da milioni di sviluppatori in tutto il mondo e viene definito "*la principale piattaforma di strumenti per la creazione di videogiochi, app ed esperienze in tempo reale*". L'ultima versione disponibile è Unity 6.2, rilasciata nel 2025, che prosegue il modello di aggiornamenti periodici con una release LTS e rilasci regolari di nuovi strumenti e miglioramenti.

- Unity Editor e Unity Hub: l'ambiente di sviluppo integrato dove si costruiscono scene 2D/3D complesse e si scrive codice C#.
  Unity Editor include pannelli per la gerarchia degli oggetti, l'ispezione delle proprietà, un marketplace di asset e tool visuali.
  Unity Hub gestisce installazioni multiple dell'editor e dei progetti;

- AR Foundation: framework cross-platform per applicazioni AR mobili. Consente di sviluppare un'unica volta e distribuire su diversi dispositivi

AR/VR. Integra funzionalità base come il tracciamento di piani, point cloud, occlusione ambientale e altre feature ARKit/ARCore;

- XR Interaction Toolkit: libreria Unity per interazioni in realtà estesa. Fornisce componenti pronti all'uso per input cross-platform, feedback aptico, raycasting, manipolazione di oggetti.

In ambito educativo e scientifico, Unity viene impiegato per visualizzazioni e simulazioni avanzate. Ad esempio, la rivista Journal of Computational Science Education descrive progetti universitari in cui gli studenti creano strumenti VR in Unity per visualizzare dati biologici complessi, ad esempio per il rendering volumetrico di immagini da microscopia confocale. Permette, inoltre, di realizzare esperienze multidisciplinari: un noto caso è HoloAnatomy (Cleveland Clinic, 2016), la prima app per HoloLens sviluppata con Unity per insegnare anatomia medica [11, 12].

2.3 Niantic Lightship – Ancoraggio spaziale

Niantic Lightship, ora parte della Niantic Spatial Platform, è una piattaforma AR specializzata in applicazioni location-based su scala globale[13]. Introdotta ufficialmente nel 2021, riunisce le tecnologie AR utilizzate da titoli come Pokémon GO e Ingress. Niantic definisce Lightship "*la prima e unica piattaforma AR scalabile*" a livello mondiale basata su una mappa spaziale 3D del mondo reale.
Il suo fulcro è l'ARDK (Augmented Reality Developer Kit) per Unity: un pacchetto SDK che si integra con Unity AR Foundation, estendendone i sistemi base con funzionalità avanzate di Niantic.
Le principali funzionalità offerte da Lightship ARDK includono:

- Depth, Occlusion e Meshing: mappatura in tempo reale delle superfici circostanti usando la fotocamera, creando mesh 3D persistenti anche su dispositivi senza LiDAR. Grazie a queste API, gli oggetti virtuali vengono correttamente occlusi dagli oggetti reali. Ad esempio, una palla virtuale che passa dietro una siepe rimane nascosta dietro di essa;

- Semantic Segmentation: riconoscimento istantaneo di fino a 20 categorie di elementi ambientali (terreno, cielo, edifici, acqua, vegetazione) mediante tecniche di visione artificiale. Questo consente al contenuto di realtà aumentata di reagire in modo realistico al contesto circostante;

- Object Detection: individuazione in tempo reale di oltre 200 classi di oggetti, per aumentare la consapevolezza spaziale dell'applicazione AR;

- Navigazione Dinamica: generazione di mesh di navigazione in AR, permettendo ad agenti virtuali di muoversi attorno agli utenti basandosi sulla geometria dell'ambiente circostante;

- VPS (Visual Positioning System): sistema cloud che fornisce a dispositivi mobili una localizzazione basata sulla mappa visiva del mondo. Grazie al VPS è possibile ancorare contenuti virtuali a posizioni geografiche specifiche con estrema precisione. Questo permette ai contenuti di rimanere stabili e persistenti anche tra sessioni diverse. Niantic ha riportato che il VPS offre appunto una "*precisione di livello centimetri*" nel posizionamento, caratteristica unica, al momento, nel panorama AR;

- WPS (World Pose System): algoritmo client-side di odometria visiva che affianca il GPS, aggiornando continuamente la posizione del dispositivo rispetto alla mappa globale. Il WPS migliora la stabilità e accuratezza della localizzazione anche in assenza di mappe VPS precedenti;

- Shared AR (AR Condiviso): supporto AR multiplayer nativo fino a 10 utenti che co-localizzano i dispositivi in uno stesso spazio fisico. Dopo la co-localizzazione, tramite VPS o QR code, tutti i partecipanti vedono in tempo reale gli stessi oggetti virtuali allineati nello spazio, consentendo esperienze collaborative.

Lightship è, quindi, specificamente progettato per applicazioni AR geolocalizzate. La piattaforma di Niantic si appoggia a una mappa mondiale di punti di interesse, o Wayspots, e location attivate. Al Lightship Summit del 2022, Niantic annunciò che erano già disponibili oltre 30.000 location pubbliche attivate per VPS, con forte densità nelle maggiori metropoli. Nel novembre 2024, Niantic ha rilanciato la sua

offerta come Niantic Spatial Platform, dichiarando di avere un milione di location VPS in produzione e di offrire ancora una volta ancoraggi persistenti ancora più precisi.

Lightship trova applicazioni anche in ambito educativo e divulgativo scientifico. Un esempio significativo è Wonderlab AR (Science Museum Group, UK): un'applicazione location-based che utilizza Niantic Lightship ARDK per trasformare elementi reali in manifestazioni visive. Con Wonderlab AR gli utenti sono incoraggiati a uscire ed esplorare il mondo circostante alla scoperta di "meraviglie" legate ai punti d'interesse locali. Questo tipo di applicazioni dimostra come Niantic Lightship possa estendere i contenuti di un laboratorio scientifico interattivo nello spazio reale di tutti i giorni, favorendo l'apprendimento. In sintesi, Lightship ARDK è attualmente l'unica piattaforma AR a consentire, grazie al suo Visual Positioning System globale, il posizionamento accurato di oggetti virtuali in coordinate reali con precisione centimetri aprendo nuove prospettive per esperienze scientifiche immersive e collaborative. [14, 15, 16]

L'integrazione dell'SDK è avvenuta direttamente all'interno di Unity, consentendo di sfruttare le funzionalità AR in modo nativo, attraverso componenti come ARSession, ARCamera e ARSemanticSegmentationManager.
Il primo passo ha riguardato la configurazione del VPS, in modo da inserire la rete di punti di riferimento riconoscibili visivamente tramite la fotocamera dello smartphone. Durante la scansione, il dispositivo confronta le feature visive catturate con quelle presenti nel database Niantic, ottenendo la posa esatta nello spazio tridimensionale. Sono stati scansionati, quindi, diversi punti di interesse e caricati nel database di Niantic

Figura 2.17: VPS del Campus di Cesena da Geospatial Browser

Una volta importati in Unity, quest'ultimo rileva le coordinate geografiche approssimative in cui la mesh di riferimento deve essere posizionata.

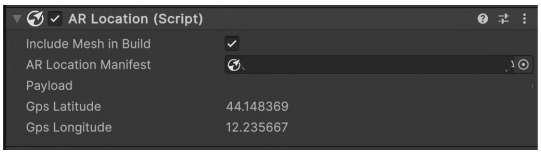


Figura 2.18: AR Location, esempio 1

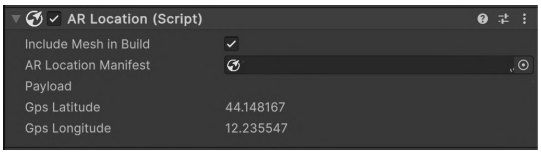


Figura 2.19: AR Location, esempio 2

Tra le varie configurazioni, l'occlusione semantica è sicuramente degna di nota. Questo sistema consente di integrare in modo plausibile oggetti digitali all'interno della scena reale. A differenza dell'occlusione basata esclusivamente sulla depth map, che si limita a stimare la distanza dei pixel della fotocamera, l'occlusione semantica di Lightship utilizza una Semantic Segmentation Map generata dal modello di visione artificiale dell'SDK. Questo modello è addestrato su un ampio dataset urbano e naturale e classifica ogni pixel dell'immagine in uno dei canali semantici disponibili [17].
La lista dei canali è configurabile direttamente dall'Inspector di Unity tramite il campo *Requested Suppression Channels*

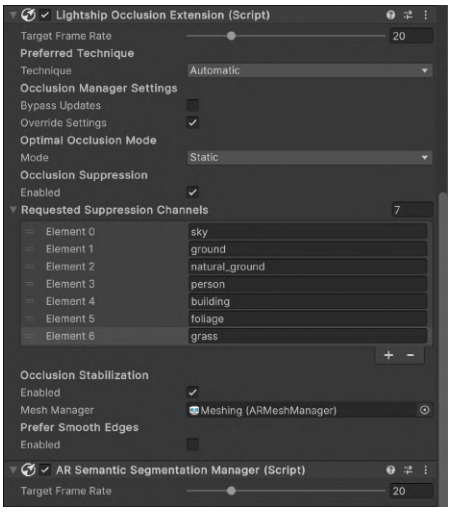


Figura 2.20: Configurazione della Semantic Segmentation

Questa tecnologia è particolarmente utile nelle esperienze MR interattive, dove l'utente si muove attorno agli oggetti o li osserva da prospettive differenti, evitando

discontinuità percettive dovute al riconoscimento instabile dei confini.

Durante le prime fasi di sviluppo, l'integrazione tra oggetti virtuali e la scena reale mostrava instabilità visiva, comunemente nota come *flickering*, dovuta alle variazioni di profondità e al tracking della fotocamera. Per risolvere il problema, è stato introdotto il componente ARMeshManager, fornito da Niantic Lightship in combinazione con AR Foundation. Questo modulo consente di effettuare una ricostruzione spaziale dell'ambiente circostante, generando una mesh tridimensionale che funge da riferimento geometrico stabile su cui vengono proiettati gli oggetti digitali, eliminando la discontinuità visiva e migliorando la coerenza spaziale tra il mondo reale e quello virtuale [18].

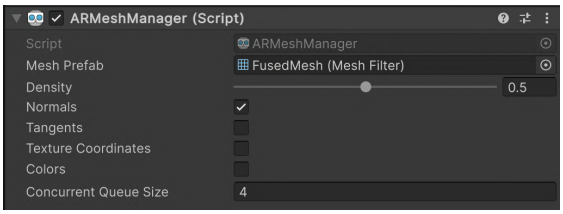


Figura 2.21: Componente ARMeshManager

Grazie a questa configurazione, gli oggetti virtuali mantengono una posizione stabile anche in presenza di variazioni del punto di vista o delle condizioni di luce.

Capitolo 3

Modelli matematici e simulazione fisica

Il capitolo illustra i fondamenti matematici che sostengono le simulazioni fisiche sviluppate nel progetto. Vengono esposte le principali equazioni differenziali che descrivono il moto dei corpi e le leggi ottiche che regolano la rifrazione. Vengono, inoltre, introdotti i metodi numerici impiegati per la risoluzione delle equazioni differenziali per l'approssimazione delle traiettorie, evidenziando il legame diretto tra la formulazione teorica e l'implementazione software.

3.1 Equazioni dei fenomeni simulati

3.1.1 Pendolo di Foucault

Il pendolo di Foucault prende il nome dal fisico francese Jean Bernard Lèon Foucault, che presentò al pubblico la famosa dimostrazione della rotazione terrestre. Nel 1851, sotto la cupola del Pantheon di Parigi, Foucault installò un pendolo costituito da una sfera in ottone, di $28kg$, appesa a un filo d'acciaio lungo $67m$ [19]. La sfera era dotata di una punta che tracciava il suo passaggio su uno strato di sabbia sul pavimento, rendendo visibile lo spostamento del piano di oscillazione a ogni periodo. Per la prima volta si offrì una prova visibile e diretta della rotazione terrestre al di fuori delle osservazioni astronomiche, dimostrando in modo suggestivo la rotazione della Terra. L'esperimento, inoltre, permise di dimostrare che,

misurando l'angolo di spostamento del piano di oscillazione, è possibile stimare la latitudine del luogo in cui si trova il pendolo. In particolare, alla latitudine λ il piano di oscillazione del pendolo subisce una precessione di $2\pi \sin \lambda$ radianti al giorno rispetto alla Terra. Ciò significa che:

- ai poli ($\lambda = 90°$) il piano compie un giro completo di 360° in circa 24 ore
- all'equatore ($\lambda = 0°$) non ruota affatto
- per le **latitudini intermedie**, l'angolo di rotazione in un giorno e il tempo per un giro completo assumono valori intermedi [20].

In generale, più ci si avvicina ai poli, maggiore è il valore di $\sin \lambda$ e quindi la precessione è più rapida; mentre avvicinandosi all'equatore, $\sin \lambda$ diminuisce fino a zero, allungando il periodo di rotazione all'infinito. La direzione di rotazione dipende dall'emisfero: verso destra (senso orario) nell'emisfero boreale e verso sinistra in quello australe [21]. Questo fenomeno è dovuto alla forza di Coriolis, un effetto inerziale che si manifesta nei sistemi di rotazione, come la Terra.
Dal punto di vista quantitativo, trascurando l'attrito dell'aria e altri effetti secondari, il pendolo di Foucault può essere modellato come un pendolo semplice di lunghezza L, soggetto alla gravità e alla rotazione terrestre. Per piccole oscillazioni attorno alla verticale, l'angolo θ rispetto alla direzione di equilibrio soddisfa, in un riferimento inerziale, l'equazione del pendolo semplice:

$$\ddot{\theta} + \frac{g}{L} \sin \theta = 0 \tag{3.1}$$

che per oscillazioni piccole (approssimando $\sin \theta \approx \theta$) diventa un moto armonico semplice di pulsazione $\omega_0 = \sqrt{\frac{g}{L}}$. In assenza di rotazione terrestre, quindi, il pendolo oscilla in un piano fisso con frequenza propria ω_0.

È necessario, però, considerare l'effetto della rotazione terrestre sul pendolo. Nel sistema di riferimento non inerziale, solidale con la Terra, compaiono forze fittizie; in particolare, la forza di Coriolis:

$$\vec{\mathbf{F}}_{Cor} = 2m\vec{\mathbf{v}} \times \vec{\mathbf{\Omega}} \tag{3.2}$$

dove m è la massa del pendolo, $\vec{\mathbf{v}}$ e la velocità della massa e $\vec{\mathbf{\Omega}}$ è la velocità angolare di rotazione terrestre [21]. Questa forza fittizia non compie lavoro, non altera l'energia cinetica, ma provoca una deviazione laterale del moto. La conseguenza è un accoppiamento tra i moti lungo le direzioni orizzontali. La forza di Coriolis, pur essendo apparente, cioè dovuta al cambio di riferimento, è indispensabile per descrivere il moto: non altera la velocità del pendolo, in modulo, ma ne cambia continuamente la direzione, producendo una rotazione nel piano di oscillazione. Matematicamente, l'equazione del moto del pendolo di Foucault risulta dalla combinazione dell'oscillazione armonica e della deviazione di Coriolis e, in forma vettoriale, si può scrivere come:

$$\ddot{\mathbf{r}} + \omega_0^2 \, \mathbf{r} = 2 \, \dot{\mathbf{r}} \times \mathbf{\Omega} \tag{3.3}$$

dove $\mathbf{r}$ è il vettore posizione della massa del pendolo rispetto al punto di sospensione, $\dot{\mathbf{r}}$ è la velocità della massa, $\ddot{\mathbf{r}}$ è l'accelerazione, $\omega_0 = \sqrt{\frac{g}{L}}$ è la frequenza naturale del pendolo e Ω è il vettore di rotazione terrestre, cioè la velocità angolare con cui la Terra ruota attorno al proprio asse. Scegliendo un sistema di coordinate con assi orizzontali (x, y) e approssimando le oscillazioni come piccole e quindi quasi orizzontali, le componenti di (3.3) danno luogo a due equazioni differenziali ordinarie accoppiate del secondo ordine:

$$\begin{cases} \ddot{x} + \omega_0^2 \, x + 2 \, \Omega \, \sin \lambda \, \dot{y} = 0, \\ \ddot{y} + \omega_0^2 \, y - 2 \, \Omega \, \sin \lambda \, \dot{x} = 0 \end{cases} \tag{3.4}$$

dove il termine contenente $\Omega \sin \lambda$ rappresenta l'effetto della forza di Coriolis sulle oscillazioni [21]. Questo sistema di equazioni non ammette, in generale, una soluzione analitica elementare, ma riproduce il comportamento osservato: il pendolo oscilla con periodo quasi pari a quello proprio $\frac{2\pi}{\omega_0}$ e il piano di oscillazione ruota lentamente con velocità angolare $\Omega \sin \lambda$ rispetto alla Terra.

Il pendolo di Foucault costituisce un esempio di dinamica in un riferimento non inerziale: la rotazione terrestre introduce termini aggiuntivi nelle equazioni del moto che spiegano la rotazione lenta del piano di oscillazione, confermando sperimentalmente la rotazione del nostro pianeta.

3.1.2 Gravilab

La simulazione fisica realizzata all'interno del **GraviLab** è dedicata allo studio della caduta dei corpi in differenti condizioni ambientali, esplorando le principali leggi che descrivono la dinamica verticale di un corpo soggetto alla gravità terrestre. Vengono descritti tre modelli:

- la caduta libera nel vuoto
- la caduta con resistenza, in due sue varianti: lineare e quadratica

Caduta libera nel vuoto

La caduta libera nel vuoto, formalizzata da *Galileo Galilei* e successivamente da *Isaac Newton* tramite il secondo principio della dinamica, costituisce la base teorica su cui si sviluppano tutti i modelli successivi.
Nel vuoto, un corpo in caduta libera è soggetto alla sola forza di gravità. In assenza di resistenza dell'aria, tutti i corpi cadono verso il basso con la stessa accelerazione costante, indipendentemente dalla loro massa. Galileo dimostrò che lo spazio percorso da un corpo in caduta da fermo cresce in proporzione al quadrato del tempo di caduta. Applicando il secondo principio della dinamica, $\sum F = ma$, si ottiene l'equazione del moto:

$$m\frac{dv}{dt} = mg \tag{3.5}$$

la cui soluzione fornisce $v(t) = v_0 + gt$. In particolare, partendo da fermo, $v_0 = 0$:

$$v(t) = gt \tag{3.6}$$

L'incremento lineare di velocità comporta che lo spazio percorso cresca in modo quadratico; infatti, integrando nuovamente, si ottiene la legge oraria $y(t) = y_0 + v_0 t + \frac{1}{2}gt^2$.Assumendo $y_0 = 0$

$$y(t) = \tfrac{1}{2}gt^2 \tag{3.7}$$

in accordo con la relazione scoperta da Galileo [22, 23].

In assenza di forze resistive, la velocità aumenta linearmente nel tempo senza limiti e non esiste una velocità terminale finita poiché nulla si oppone alla gravità. Ovviamente, nella realtà, la resistenza dell'aria interviene a limitare l'accelerazione e a impedire che la velocità cresca indefinitamente. Quando si considera la resistenza del mezzo, la dinamica diventa più complessa. In generale, la forza di drag, o resistenza fluida, si oppone al moto di un oggetto in un fluido (gas o liquido) ed è funzione della velocità dell'oggetto rispetto al fluido [24]. I due modelli di drag, validi in distinti regimi di flusso sono:

- lineare: in regime di basse velocità e per oggetti molto piccoli in fluidi molto viscosi, la resistenza è proporzionale linearmente alla velocità;
- quadratico: per velocità e dimensioni ordinarie (ad esempio, un paracadutista o un'automobile) la resistenza è sperimentalmente proporzionale al quadrato della velocità.

Moto con resistenza lineare

A basse velocità, in fluidi altamente viscosi, la forza di resistenza è sperimentalmente proporzionale alla velocità relativa dell'oggetto rispetto al fluido. *Sir George G. Stokes* derivò per primo, nel 1851, l'espressione analitica di questa forza viscosa per una sfera di raggio r che si muove lentamente in un fluido di viscosità dinamica η. La **legge di Stokes** afferma che la forza di attrito F_s è:

$$F_s = 6\pi\,\eta\,r\,v \tag{3.8}$$

ed è linearmente proporzionale alla velocità v dell'oggetto [25].
Questo modello di resistenza lineare, detto attrito viscoso lineare, è valido quando il flusso attorno al corpo è regolare e privo di turbolenza, condizione realizzata, ad esempio, per piccole particelle in liquidi densi o gas a bassa velocità. In questi casi, gli effetti inerziali del fluido sono trascurabili, mentre dominano gli effetti viscosi lineari [26]. Esempi tipici di tale regime includono il moto di granelli di polline o goccioline in aria/acqua e le particelle di polvere che sedimentano lentamente. In tali situazioni, l'oggetto raggiunge rapidamente una velocità terminale costante e cade a velocità quasi uniforme [24]. In questo modello, per un oggetto generico,

si considera una costante di proporzionalità c tale che la forza di resistenza sia $F_d = -cv$. Applicando la seconda legge di Newton al moto verticale si ottiene:

$$m\frac{dv}{dt} = mg - cv \tag{3.9}$$

dove mg è la forza peso e cv è la forza di attrito. Questa è un'equazione differenziale lineare a coefficienti costanti. In condizioni iniziali $v_0 = 0$, la soluzione è nota e mostra un regime transitorio esponenziale verso una velocità limite. In particolare, la velocità tende asintoticamente al valore limite v_t tale che l'accelerazione si annulla ($\frac{dv}{dt} = 0$): imponendo $mg - cv_t = 0$ si ottiene

$$v_t = \frac{mg}{c} \tag{3.10}$$

La soluzione generale per $v(t)$ risulta:

$$v(t) = v_t\Big(1 - e^{-\frac{c}{m}t}\Big) \tag{3.11}$$

che descrive un incremento di velocità che parte da $v(0) = 0$ e si avvicina gradualmente a v_t senza mai superarlo. A differenza della caduta libera ideale, quindi, l'oggetto non accelera indefinitamente, ma converge verso una velocità limite determinata dal bilancio tra la forza di gravità e la forza viscosa lineare.

Dal punto di vista storico, il risultato di Stokes fu cruciale per spiegare, ad esempio, la lenta caduta delle gocce in liquidi viscosi e la sedimentazione di particelle fini in fluidi. È interessante notare che questo modello fallisce per oggetti di dimensioni maggiori o a velocità elevate, dove il flusso diventa turbolento. In quei casi, la dipendenza lineare non è più valida e bisogna adottare un modello quadratico di resistenza [24, 25].

Moto con resistenza quadratica

Per oggetti di dimensioni e velocità ordinarie che si muovono in fluidi meno viscosi, la resistenza del mezzo segue una legge quadratica rispetto alla velocità. Questo avviene perché, a velocità elevate, l'oggetto impartisce una quantità di moto al fluido circostante in modo proporzionale a v^2 e la forza necessaria risulta propor-

zionale a ρ (densità del fluido) e ad A (sezione trasversale) [27]. Questo porta all'equazione del drag aerodinamico:

$$F_D = \tfrac{1}{2} C_d \rho A v^2 \tag{3.12}$$

dove C_d è il coefficiente di resistenza, adimensionale, dipendente dalla forma dell'oggetto. La forza di drag è diretta in verso opposto al moto e cresce con il quadrato della velocità, diventando molto rilevante a velocità elevate [24].

Considerando un oggetto in caduta verticale nell'aria, la forza peso è mg si oppone alla resistenza quadratica (3.12). L'equazione di movimento diventa:

$$m \tfrac{dv}{dt} = mg - \tfrac{1}{2} C_d \rho A v^2 \tag{3.13}$$

Questa equazione differenziale non lineare ammette una soluzione analitica in termini di funzioni iperboliche. Analogamente al caso lineare, esiste una velocità terminale v_t raggiunta quando l'accelerazione si annulla: ponendo $\frac{dv}{dt} = 0$ si ottiene l'equilibrio $mg = \frac{1}{2} C_d \rho A v_t^2$, da cui si ricava:

$$v_t = \sqrt{\frac{2mg}{\rho C_d A}} \tag{3.14}$$

La soluzione dell'equazione differenziale completa, con velocità iniziale nulla, fornisce la dipendenza temporale:

$$v(t) = v_t \tanh\Big(\frac{g\,t}{v_t}\Big) \tag{3.15}$$

- per t piccoli, $\tanh\left(\frac{gt}{v_t}\right) \approx \frac{gt}{v_t}$ e, di conseguenza, $v(t) \approx gt$. Si deduce che l'accelerazione è circa g, come in caduta libera;

- per t grandi, $\tanh\left(\frac{gt}{v_t}\right) \to 1$ e la velocità si avvicina asintoticamente a v_t.

In altri termini, la resistenza dell'aria impedisce alla velocità di crescere indefinitamente: dopo un certo tempo, l'oggetto raggiunge una velocità quasi costante in cui la forza di gravità è bilanciata dalla forza di drag [27].

3.1.3 Sistema Solare

La simulazione scientifica che riguarda il sistema solare mostra gli effetti della gravitazione universale, ovvero dell'attrazione reciproca che si manifesta tra tutti i corpi dotati di massa. *Isaac Newton*, nel 1665, dimostrò che ogni corpo nell'universo attrae ogni altro corpo, e questa proprietà prende il nome di gravitazione. La legge di gravitazione universale si esprime

$$\overrightarrow{F_g} = G\,\frac{M_{\text{Sole}}\,m_{\text{Pianeta}}}{r^2}\,\hat{r} \tag{3.16}$$

dove M_{Sole} e $m_{pianeta}$ sono le masse dei due corpi, r è la distanza tra i loro centri, $\hat{r}$ è il versore che indica la direzione del vettore r e G è la costante di gravitazione universale [28].

Il modello, applicato in questa simulazione, prende il nome di Problema dei due corpi con massa dominante, in cui uno dei due corpi (in questo caso specifico il Sole) ha una massa molto maggiore rispetto a quella del pianeta, che viene considerato puntiforme, e la sua influenza gravitazionale sul moto del corpo più massiccio è trascurabile. In prima approssimazione, si considera, dunque, il corpo più pesante praticamente fisso nello spazio e si calcola l'accelerazione del corpo più leggero, dovuta esclusivamente alla forza gravitazionale che agisce su di esso. L'accelerazione gravitazionale $\overrightarrow{a_g}$ di una particella è dovuta esclusivamente alla forza gravitazionale che agisce su di essa. Quindi, in questo modello si calcola l'accelerazione del corpo più leggero verso quello più pesante, che viene considerato fisso nello spazio [29]. Per la seconda legge della dinamica, la forza risultante che agisce sul corpo è pari al prodotto della massa del corpo per la sua accelerazione

$$\overrightarrow{F} = m_{\text{pianeta}} \cdot \overrightarrow{a} \tag{3.17}$$

Isolando l'accelerazione $\overrightarrow{a}$, si ottiene:

$$\overrightarrow{a} = \frac{\overrightarrow{F}}{m_{\text{pianeta}}}$$

Sostituendo la forza gravitazionale $\overrightarrow{F_g}$ in questa formula:

$$\overrightarrow{a} = G\frac{M_{\text{Sole}} \cdot \cancel{m_{\text{pianeta}}}}{r^2}\hat{r} \cdot \frac{1}{\cancel{m_{\text{pianeta}}}}$$

$$\overrightarrow{a} = G\frac{M_{\text{Sole}}}{r^2}\hat{r} \tag{3.18}$$

Questa è l'accelerazione gravitazionale con cui il pianeta è attirato verso il Sole. Come è facile osservare, dipende unicamente dalla massa del corpo centrale e dalla distanza, risultando indipendente dalla massa del pianeta stesso [30]. Ciò è coerente con il fatto che, in un dato campo gravitazionale, tutti i corpi acquisiscono la stessa accelerazione, indipendentemente dalla loro massa, come già evidenziato negli esperimenti galileiani. In conclusione, nel modello semplificato adottato, il pianeta, corpo di massa minore, subisce un'alterazione a diretta verso il Sole, mentre il Sole rimane praticamente fermo perché la forza gravitazionale esercitata dal pianeta è trascurabile in confronto.

3.1.4 Prisma Ottico

Il modello fisico adottato per la simulazione del prisma è quello dell'ottica geometrica, che concettualizza la luce come un insieme di raggi che:

- si propaga in linea retta in mezzi omogenei, come il vuoto o l'aria
- subisce delle deviazioni quando incontra superfici con indici di rifrazione diversi.

Nel caso del prisma di vetro, questa semplificazione è valida perché la dimensione del prisma ($\approx$ centimetri) è di molti ordini di grandezza superiore alla lunghezza d'onda della luce visibile ($\approx 10^{-7}\,\text{m}$) rendendo trascurabili gli effetti ondulatori di diffrazione e interferenza [31]. Ogni materiale trasparente è caratterizzato da un indice di rifrazione η, definito come rapporto tra la velocità della luce nel vuoto c e la velocità della luce nel mezzo v:

$$\eta = \frac{c}{v} \tag{3.19}$$

da cui è intuitivo dedurre che a un maggior indice η corrisponde una minore velocità della luce nel mezzo. Per aria e vetro, i tipici valori dell'indice di rifrazione sono rispettivamente:

$$\eta_{\text{aria}} \approx 1.00029 \qquad n_{\text{vetro}} \approx 1.5168$$

[32] Quando un raggio luminoso passa da un mezzo con indice η_1 a un secondo con indice η_2, la direzione del raggio cambia secondo la legge di Snell:

$$\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2 \tag{3.20}$$

dove θ_1 è l'angolo di incidenza e θ_2 l'angolo di rifrazione, entrambi misurati rispetto alla normale alla superficie.

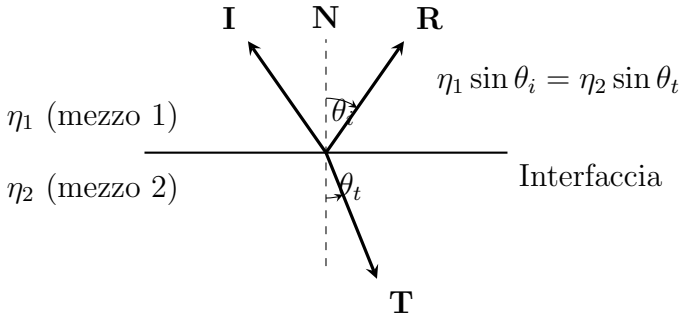


Figura 3.1: Rappresentazione dei vettori di incidenza, riflessione e rifrazione.

Questa legge può essere riformulata in forma vettoriale, derivata dalla geometria della rifrazione, permettendo di calcolare direttamente la direzione del raggio rifratto senza utilizzare gli angoli [33]. Siano:

- **I**: il vettore direzione incidente, normalizzato;
- **N**: la normale unitaria alla superficie, orientata verso il mezzo d'incidenza;
- η_1 e η_2: gli indici dei due mezzi.

Si definisce:

$$\cos\theta_i = -\mathbf{I} \cdot \mathbf{N} \qquad \eta = \frac{\eta_1}{\eta_2} \tag{3.21}$$

La direzione rifratta, scomposta nelle sue componenti

$$\mathbf{T} = \mathbf{T}_t + \mathbf{T}_n \tag{3.22}$$

si trova nel piano formato da $\mathbf{I}$ e da $\mathbf{N}$, per cui sarà una combinazione lineare di questi due vettori. Le condizioni fondamentali da soddisfare sono:

- **Snell**: $\eta_1 \sin\theta_i = \eta_2 \sin\theta_t \quad \Rightarrow \quad \frac{\eta_1}{\eta_2}\sin\theta_i = \sin\theta_t \quad \Rightarrow \quad \eta \sin\theta_i = \sin\theta_t$
- **Norma unitaria**: il vettore rifratto $\mathbf{T}$ deve avere lunghezza $1 \rightarrow \|\mathbf{T}\| = 1$

Inoltre, si impone che l'angolo tra $\mathbf{T}$ e $-\mathbf{N}$ sia proprio $\theta_\mathbf{t}$, cioè $\cos\theta_t = -\mathbf{T}\cdot\mathbf{N}$.

Componente tangenziale $\mathbf{T}_t$

Il vettore incidente può essere scomposto nelle sue componenti tangenziale e normale $\mathbf{I} = \mathbf{I}_t + \mathbf{I}_n$, dove:

$$\begin{aligned} \mathbf{I}_t &= \mathbf{I} - (\mathbf{I}\cdot\mathbf{N})\mathbf{N} & \mathbf{I}_n &= (\mathbf{I}\cdot\mathbf{N})\mathbf{N} \\ &= \mathbf{I} + \cos\theta_i\,\mathbf{N} & &= -\cos\theta_i\,\mathbf{N} \end{aligned}$$

Snell agisce sulla direzione tangenziale. La componente tangenziale del vettore d'onda è continua $\eta_1\mathbf{I}_t = \eta_2\mathbf{T}_t$, quindi, per le direzioni unitarie:

$$\mathbf{T}_t = \eta\mathbf{I}_t \tag{3.23}$$

Detto in altre parole: la direzione tangenziale della luce cambia solo di un fattore di scala dovuto al rapporto degli indici.

Componente normale $\mathbf{T}_n$

Nell'equazione (3.22) è possibile sostituire la componente tangenziale appena calcolata e la componente lungo la normale $\mathbf{T}_n = T_n\mathbf{N}$:

$$\mathbf{T} = \eta\mathbf{I}_t + T_n\mathbf{N} \tag{3.24}$$

e, successivamente, sostituendo il valore di $\mathbf{I}_t$

$$\mathbf{T} = \eta(\mathbf{I} + \cos\theta_i\mathbf{N}) + T_n\mathbf{N} = \eta\mathbf{I} + (\eta\cos\theta_i + T_n)\mathbf{N}$$

L'unica incognita ora è T_n. L'ampiezza non è immediata, ma è possibile ricavarla sapendo che $\mathbf{T}$ deve essere un vettore unitario, cioè $\|\mathbf{T}\| = 1$. Imponendo questa

condizione:

$$1 = \mathbf{T} \cdot \mathbf{T} = (\eta \mathbf{I} + (\eta \cos\theta_i + T_n)\mathbf{N}) \cdot (\eta \mathbf{I} + (\eta \cos\theta_i + T_n)\mathbf{N})$$

sviluppando i prodotti e sapendo che $\mathbf{I} \cdot \mathbf{I} = 1, \quad \mathbf{N} \cdot \mathbf{N} = 1, \quad \mathbf{I} \cdot \mathbf{N} = -\cos\theta_i$ e sostituendoli:

$$1 = \eta^2 + (\eta \cos\theta_i + T_n)^2 - 2\eta(\eta \cos\theta_i + T_n)\cos\theta_i$$

Calcolando il quadrato e dato che $1 - \cos^2\theta_i = \sin^2\theta_i$ si arriva alla forma finale:

$$T_n = \pm\sqrt{1 - \eta^2 \sin^2\theta_i}$$

Il segno corretto dipende dal verso della normale: il raggio rifratto entra nel secondo mezzo, quindi la componente $\mathbf{N}$ deve puntare verso l'interno. Si prende, perciò, il segno negativo rispetto alla direzione della normale uscente:

$$T_n = -\sqrt{1 - \eta^2 \sin^2\theta_i} \tag{3.25}$$

Sostituendo T_n in 3.24, si ottiene la direzione rifratta $\mathbf{T}$:

$$\mathbf{T} = \eta \mathbf{I} + \left(\eta(-\cos\theta_i) - \sqrt{1 - \eta^2(1 - \cos^2\theta_i)}\right)\mathbf{N}. \tag{3.26}$$

Il termine sotto radice,

$$k = 1 - \eta^2(1 - \cos^2\theta_i) \tag{3.27}$$

determina la condizione fisica della rifrazione: se $k < 0$, non esiste soluzione reale; la rifrazione è impossibile e si verifica la **riflessione totale interna** (*Total Internal Reflection*, TIR) [31, 32].

3.2 Metodi numerici di integrazione

Ogni simulazione fisica ha origine da una fase imprescindibile: la modellazione matematica che ha riguardato i singoli sistemi con lo scopo di ottenere un'evoluzione realistica del sistema nel tempo. L'implementazione di simulazioni fisiche in mixed reality richiede dei modelli dinamicamente plausibili e reattivi, ma contemporaneamente stabili nel tempo. Che si tratti della traiettoria del pendolo di Foucault o delle orbite in un sistema solare, il cuore della computazione è sempre lo stesso: integrare nel tempo equazioni differenziali ordinarie (ODE) che descrivono l'evoluzione dello stato del sistema, come posizione, velocità e angolo. Per questo motivo si ricorre a metodi numerici di integrazione che costruiscono una soluzione approssimata, passo dopo passo. In un ambiente interattivo e legato al frame rate, la scelta dell'integratore incide direttamente su tre aspetti chiave:

- **accuratezza** che determina quanto la simulazione si avvicini a quella reale a parità di passo temporale
- **stabilità** che stabilisce entro quali passi la soluzione non diverga e non introduca artefatti
- **costo computazionale** che misura quante valutazioni servono per ogni aggiornamento, condizionando il rendering e l'interazione.

Nel contesto di sistemi conservativi, oltre all'errore locale, diventa cruciale il comportamento a lungo termine: alcuni integratori preservano meglio quantità fisiche (come l'energia o il momento angolare), riducendo le derive numeriche che, accumulate nel tempo, compromettono la plausibilità della simulazione. Per questo motivo, lo stesso problema fisico viene esaminato con più integratori, scegliendo di volta in volta lo schema più adatto. I tre metodi impiegati sono: Eulero, Verlet, Runge-Kutta 4.

3.2.1 Eulero

Il metodo di Eulero è il più semplice schema di integrazione numerica per le equazioni differenziali ordinarie.

Eulero esplicito

Data un'ODE $\frac{dy}{dt} = f(y,t)$ e un passo temporale h, l'integratore calcola la nuova stima y_{n+1} con una singola valutazione della derivata, seguendo la pendenza corrente per far avanzare la soluzione di un passo h:

$$y_{n+1} = y_n + h\, f(y_n, t_n) \tag{3.28}$$

Questo metodo è di primo ordine: l'errore locale per passo è $O(h^2)$ e l'errore globale cresce linearmente con il passo ($O(h)$) [34, 35]. Questo significa che dimezzando h si dimezza approssimativamente l'errore globale [36]. In generale, Eulero richiede passi molto piccoli per evitare esplosioni numeriche; infatti, Eulero introduce errori sistematici che accumulano energia in eccesso, producendo traiettorie che si allontanano sempre più da quelle reali.

Eulero–Cromer (semi-esplicito)

Una variante dello schema di Eulero è il metodo *Eulero-Cromer*, o *semi-esplicito*, introdotto per migliorare la stabilità dei sistemi dinamici conservativi. L'idea di base consiste nell'aggiornare prima la velocità e poi la posizione, utilizzando la velocità aggiornata nel secondo passo di integrazione.

$$\begin{cases} v_{n+1} = v_n + h\, a(y_n, t_n) \\ y_{n+1} = y_n + h\, v_{n+1} \end{cases} \tag{3.29}$$

A differenza dello schema di Eulero esplicito, in cui la posizione viene aggiornata con la velocità al tempo t_n, qui la nuova posizione dipende dalla velocità calcolata al tempo t_{n+1}. Questa semplice modifica rende il metodo più stabile per sistemi oscillatori, dove la conservazione dell'energia è cruciale. Infatti, mentre Eulero esplicito tende ad accumulare energia artificiale nel sistema (causando traiettorie divergenti), Eulero-Cromer mantiene l'energia meccanica entro limiti oscillanti attorno al valore reale. Per questo motivo, è ampiamente utilizzato in fisica computazionale per la simulazione di moti planetari e sistemi con vincoli di energia. Dal punto di vista dell'analisi numerica, lo schema rimane di primo ordine: l'errore

locale per passo è $O(h^2)$ e quello globale $O(h)$, ma offre una stabilità qualitativa molto migliore rispetto a Eulero esplicito: l'ampiezza non cresce indefinitamente, ma resta confinata, rendendo il metodo stabile anche con passi più ampi rispetto a Eulero esplicito. [37, 38, 39]

3.2.2 Verlet

Il metodo di Verlet è un integratore numerico di secondo ordine impiegato per sistemi conservativi governati da equazioni del secondo ordine. È stato introdotto originariamente da *Loup Verlet*, nel 1967, per calcoli di dinamica atomica e si distingue per la sua eccellente stabilità a lungo termine [40, 41, 42]. A differenza di Eulero, che utilizza solo lo stato corrente per predire il futuro, Verlet è un metodo multi-passo: calcola la nuova posizione x_{n+1} combinando la posizione attuale x_n, la posizione precedente x_{n-1} e l'accelerazione $a(x_n)$ al tempo t_n. La formula di aggiornamento caratteristica è

$$x_{n+1} = 2x_n - x_{n-1} + a(x_n)\,\Delta t^2 \tag{3.30}$$

dove Δt rappresenta l'intervallo temporale tra due passi consecutivi di integrazione. Per avviare l'integrazione, è necessario un particolare passo iniziale, poiché il metodo utilizza i due passi precedenti e non ha immediatamente disponibile il primo passo. Tipicamente, si calcola prima x_1 con un metodo monostep (ad esempio, usando la velocità iniziale v_0: $x_1 = x_0 + v_0\,dt + \frac{1}{2}a(x_0)dt^2$) e poi si applica la formula di Verlet per $n \geq 1$ [43].

Il metodo di Verlet è di secondo ordine globalmente, ovvero l'errore globale scala come $O(\Delta t^2)$. Ciò significa che dimezzando il passo Δt, l'errore globale si riduce di un fattore 4: è un notevole miglioramento rispetto al fattore ≈ 2 di Eulero, di ordine 1.

Un altro grande vantaggio del metodo di Verlet è la stabilità a lungo termine nei sistemi conservativi. Il metodo non introduce deriva di energia: l'energia totale di un sistema oscillante tende a rimanere quasi costante nel tempo, con piccole oscillazioni attorno al valore corretto, anziché crescere o decrescere senza controllo. Infatti, a differenza di Eulero Esplicito, che tipicamente aggiunge energia a ogni passo, facendo crescere l'ampiezza, Verlet non provoca né crescita né attenuazione

sistematica dell'ampiezza di oscillazione. Questa "quasi-conservazione" dell'energia fa sì che, nelle lunghe simulazioni, il metodo di Verlet riproduca comportamenti plausibili [44].

In sintesi, l'integratore di Verlet rappresenta un compromesso eccellente per simulazioni conservative interattive: accuratezza quadratica, stabilità a lungo termine e basso costo per passo.

3.2.3 Metodo di Runge-Kutta 4

L'ultimo integratore impiegato è il Runge-Kutta di quarto ordine, comunemente abbreviato in RK4. Questo metodo costruisce la soluzione avanzando in quattro sotto-fasi all'interno di ciascun intervallo Δt calcolando più stime della derivata della soluzione e combinandole.
Data un'ODE del primo ordine, RK4 effettua:

$$\begin{cases} k_1 = f(y_n,\, t_n) \\ k_2 = f\left(y_n + \dfrac{\Delta t}{2} k_1,\, t_n + \dfrac{\Delta t}{2}\right) \\ k_3 = f\left(y_n + \dfrac{\Delta t}{2} k_2,\, t_n + \dfrac{\Delta t}{2}\right) \\ k_4 = f(y_n + \Delta t\, k_3,\, t_n + \Delta t) \end{cases} \tag{3.31}$$

quindi combina queste stime incrementali in una media ponderata. Dopo aver calcolato le stime k_1, k_2, k_3, k_4, si aggiorna il valore di y:

$$y_{n+1} = y_n + \frac{\Delta t}{6}\Big(k_1 + 2k_2 + 2k_3 + k_4\Big) \tag{3.32}$$

In altri termini, RK4 campiona la derivata in 4 punti diversi dell'intervallo (t_n, t_{n+1}) e ottiene y_{n+1} sommando al valore corrente y_n una media di quattro pendenze, dando peso doppio alle stime centrali k_2, k_3 e peso singolo alle stime iniziali e finali. Questo schema di calcolo aumenta drasticamente l'accuratezza locale: RK4 è un metodo di ordine 4, il che significa che l'errore locale per ogni passo è dell'ordine di $O(\Delta t^5)$ e l'errore globale si riduce proporzionalmente a $O(\Delta t^4)$. In concreto, dimezzare il passo temporale riduce l'errore globale di un fattore approssimativa-

mente 16 (2^4). Questa elevata accuratezza consente, a parità di precisione richiesta, di utilizzare passi più grandi rispetto ai metodi di ordine inferiore [45, 46].

3.3 Applicazione dei modelli alle simulazioni sviluppate

3.3.1 Pendolo

Equazioni del moto. Il pendolo semplice, in assenza di attrito, soddisfa l'equazione differenziale del secondo ordine, non lineare:

$$\ddot{\theta}(t) = -\frac{g}{L}\sin\theta(t) \tag{3.33}$$

dove:

- $\theta(t)$ è l'angolo rispetto alla verticale
- g è l'accelerazione di gravità
- L è la lunghezza del filo.

Per integrarla numericamente, è utile riscriverla come un sistema di equazioni del primo ordine, introducendo la velocità angolare $\omega(t)$

$$\begin{cases} \theta & \text{angolo di deviazione} \\ \omega = \dot{\theta} & \text{velocità angolare} \end{cases} \tag{3.34}$$

In forma vettoriale:

$$y = \begin{bmatrix} \theta(t) \\ \omega(t) \end{bmatrix}, \qquad \dot{y} = \begin{bmatrix} \dot{\theta} \\ \dot{\omega} \end{bmatrix} = \begin{bmatrix} \omega \\ -\dfrac{g}{L}\sin\theta \end{bmatrix}$$

$$\begin{cases} \dot{\theta} = \omega \\ \dot{\omega} = -\dfrac{g}{L}\sin\theta \end{cases} \tag{3.35}$$

Schema di Eulero Il metodo di Eulero approssima con un passo dt

$$\begin{cases} \theta_{n+1} = \theta_n + \Delta t \cdot \omega_n \\ \omega_{n+1} = \omega_n + \Delta t \cdot \left(-\dfrac{g}{L}\sin\theta_n\right) \end{cases} \tag{3.36}$$

Schema di RK4 Le quattro stime intermedie (k_1, k_2, k_3, k_4):

$$\begin{aligned} k_1^\theta &= \omega_n, & k_1^\omega &= -\frac{g}{L}\sin(\theta_n) \\ k_2^\theta &= \omega_n + \frac{\Delta t}{2}\,k_1^\omega, & k_2^\omega &= -\frac{g}{L}\sin\Big(\theta_n + \frac{\Delta t}{2}\,k_1^\theta\Big) \\ k_3^\theta &= \omega_n + \frac{\Delta t}{2}\,k_2^\omega, & k_3^\omega &= -\frac{g}{L}\sin\Big(\theta_n + \frac{\Delta t}{2}\,k_2^\theta\Big) \\ k_4^\theta &= \omega_n + \Delta t\,k_3^\omega, & k_4^\omega &= -\frac{g}{L}\sin\Big(\theta_n + \Delta t\,k_3^\theta\Big) \end{aligned} \tag{3.37}$$

Con aggiornamento finale:

$$\begin{cases} \theta_{n+1} = \theta_n + \dfrac{\Delta t}{6}\Big(k_1^\theta + 2k_2^\theta + 2k_3^\theta + k_4^\theta\Big) \\ \omega_{n+1} = \omega_n + \dfrac{\Delta t}{6}\Big(k_1^\omega + 2k_2^\omega + 2k_3^\omega + k_4^\omega\Big) \end{cases} \tag{3.38}$$

Questo schema permette di stimare simultaneamente la posizione angolare e la velocità angolare con errore globale $O(\Delta t^4)$, garantendo un'evoluzione stabile e accurata anche con passi temporali più ampi rispetto a Eulero.

Schema RK4 con due componenti La stessa formulazione viene estesa a due componenti cartesiane del moto oscillatorio (x, y), includendo nel termine accelerativo la forza apparente di Coriolis. Il pendolo di Foucault rappresenta un caso particolare del pendolo semplice in cui, oltre alla forza peso, è necessario tenere conto delle forze apparenti dovute alla rotazione terrestre. In particolare, nel sistema di riferimento solidale con la Terra, l'accelerazione del punto materiale di massa m è influenzata dalla forza di Coriolis, proporzionale alla velocità della massa e alla velocità angolare terrestre $\mathbf{\Omega}$. Proiettando il moto sul piano orizzontale,

si ottiene il seguente sistema dinamico:

$$\begin{cases} \ddot{x} = -\omega_0^2\, x \;-\; 2\,\Omega\,\sin\varphi\,\dot{y} \\ \ddot{y} = -\omega_0^2\, y \;+\; 2\,\Omega\,\sin\varphi\,\dot{x} \end{cases} \tag{3.39}$$

dove:

- $\omega_0 = \sqrt{\dfrac{g}{L}}$ è la pulsazione naturale del pendolo
- Ω è la velocità angolare di rotazione terrestre ($\Omega \approx 7.292115 \times 10^{-5}\,\text{rad/s}$);
- φ è la latitudine del luogo
- (x, y) rappresentano le coordinate orizzontali della massa proiettate sul piano di oscillazione.

Il termine $2\Omega \sin\varphi$ rappresenta la componente della forza di Coriolis nel piano responsabile della precessione del piano di oscillazione del pendolo.
Introducendo le velocità $v_x = \dot{x}$ e $v_y = \dot{y}$, il sistema può essere riscritto in forma di primo ordine come:

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{v}_x = -\omega_0^2\, x - 2\,\Omega\,\sin\varphi\, v_y \\ \dot{v}_y = -\omega_0^2\, y + 2\,\Omega\,\sin\varphi\, v_x \end{cases} \tag{3.40}$$

In forma vettoriale, ponendo:

$$\mathbf{y} = \begin{bmatrix} x \\ y \\ v_x \\ v_y \end{bmatrix}, \qquad \dot{\mathbf{y}} = f(\mathbf{y}) = \begin{bmatrix} v_x \\ v_y \\ -\omega_0^2\, x - 2\Omega \sin\varphi\, v_y \\ -\omega_0^2\, y + 2\Omega \sin\varphi\, v_x \end{bmatrix} \tag{3.41}$$

Derivando poi le quattro stime intermedie necessarie per applicare RK4, per ogni variabile (x, y, v_x, v_y):

$$\begin{aligned}
k_1^x &= v_x, & k_1^{v_x} &= -\,\omega_0^2\, x \;-\; f_c\, v_y,\\
k_2^x &= v_x + \tfrac{\Delta t}{2}\, k_1^{v_x}, & k_2^{v_x} &= -\,\omega_0^2\Big(x + \tfrac{\Delta t}{2}\, k_1^x\Big) \;-\; f_c\Big(v_y + \tfrac{\Delta t}{2}\, k_1^{v_y}\Big),\\
k_3^x &= v_x + \tfrac{\Delta t}{2}\, k_2^{v_x}, & k_3^{v_x} &= -\,\omega_0^2\Big(x + \tfrac{\Delta t}{2}\, k_2^x\Big) \;-\; f_c\Big(v_y + \tfrac{\Delta t}{2}\, k_2^{v_y}\Big),\\
k_4^x &= v_x + \Delta t\, k_3^{v_x}, & k_4^{v_x} &= -\,\omega_0^2\Big(x + \Delta t\, k_3^x\Big) \;-\; f_c\Big(v_y + \Delta t\, k_3^{v_y}\Big).
\end{aligned} \tag{3.42}$$

$$\begin{aligned}
k_1^y &= v_y, & k_1^{v_y} &= -\,\omega_0^2\, y \;+\; f_c\, v_x,\\
k_2^y &= v_y + \tfrac{\Delta t}{2}\, k_1^{v_y}, & k_2^{v_y} &= -\,\omega_0^2\Big(y + \tfrac{\Delta t}{2}\, k_1^y\Big) \;+\; f_c\Big(v_x + \tfrac{\Delta t}{2}\, k_1^{v_x}\Big),\\
k_3^y &= v_y + \tfrac{\Delta t}{2}\, k_2^{v_y}, & k_3^{v_y} &= -\,\omega_0^2\Big(y + \tfrac{\Delta t}{2}\, k_2^y\Big) \;+\; f_c\Big(v_x + \tfrac{\Delta t}{2}\, k_2^{v_x}\Big),\\
k_4^y &= v_y + \Delta t\, k_3^{v_y}, & k_4^{v_y} &= -\,\omega_0^2\Big(y + \Delta t\, k_3^y\Big) \;+\; f_c\Big(v_x + \Delta t\, k_3^{v_x}\Big).
\end{aligned} \tag{3.43}$$

Valutando ad ogni passo l'effetto combinato della forza $-\omega_0^2(x, y)$ e del termine di Coriolis proporzionale alla velocità.

$$\begin{cases}
x_{n+1} = x_n + \dfrac{\Delta t}{6}\Big(k_1^x + 2k_2^x + 2k_3^x + k_4^x\Big)\\
y_{n+1} = y_n + \dfrac{\Delta t}{6}\Big(k_1^y + 2k_2^y + 2k_3^y + k_4^y\Big)\\
v_{x,n+1} = v_{x,n} + \dfrac{\Delta t}{6}\Big(k_1^{v_x} + 2k_2^{v_x} + 2k_3^{v_x} + k_4^{v_x}\Big)\\
v_{y,n+1} = v_{y,n} + \dfrac{\Delta t}{6}\Big(k_1^{v_y} + 2k_2^{v_y} + 2k_3^{v_y} + k_4^{v_y}\Big)
\end{cases} \tag{3.44}$$

Come nel caso del pendolo semplice, anche qui lo schema RK4 presenta un errore globale di ordine $O(\Delta t^4)$ e permette di integrare il moto in modo stabile anche con passi temporali relativamente ampi, conservando un'ottima precisione nella descrizione della precessione del piano di oscillazione.

3.3.2 Sistema solare

Nel modello adottato, si considera il Sole come corpo centrale fisso e di massa dominante M, mentre i pianeti si muovono unicamente sotto la sua attrazione gravitazionale. Le interazioni tra i pianeti sono trascurate e ogni orbita viene integrata in modo indipendente. Indicando con (x, y, z) le coordinate del pianeta rispetto al Sole, la forza gravitazionale è diretta verso l'origine e il modulo dell'accelerazione è inversamente proporzionale al quadrato della distanza:

$$a = \frac{GM}{r^2}, \qquad r = \sqrt{x^2 + y^2 + z^2}.$$

Per esprimere l'accelerazione in forma vettoriale (cioè diretta verso il centro) si moltiplica per il versore $\mathbf{r}/r = (x, y, z)/r$, ottenendo:

$$a_x = -\frac{GM\,x}{r^3}, \qquad a_y = -\frac{GM\,y}{r^3}, \qquad a_z = -\frac{GM\,z}{r^3}. \tag{3.45}$$

Il denominatore contiene quindi r^3 (anziché r^2) perché il termine $1/r^2$ deriva dall'intensità della forza, mentre la divisione per un ulteriore r serve a normalizzare la direzione del vettore (x, y, z). Per evitare instabilità numeriche quando il pianeta si trova molto vicino al Sole, è stato introdotto un piccolo termine di *softening* ε, che impedisce a r di annullarsi:

$$\begin{aligned} a_x &= -\frac{GM\,x}{\left(x^2 + y^2 + z^2 + \varepsilon^2\right)^{3/2}}, \\ a_y &= -\frac{GM\,y}{\left(x^2 + y^2 + z^2 + \varepsilon^2\right)^{3/2}}, \\ a_z &= -\frac{GM\,z}{\left(x^2 + y^2 + z^2 + \varepsilon^2\right)^{3/2}}. \end{aligned} \tag{3.46}$$

Definendo le velocità (v_x, v_y, v_z) si ottiene il sistema dinamico del primo ordine:

$$\begin{cases} \dot{x} = v_x \\ \dot{y} = v_y \\ \dot{z} = v_z \\ \dot{v}_x = a_x(x, y, z) \\ \dot{v}_y = a_y(x, y, z) \\ \dot{v}_z = a_z(x, y, z) \end{cases} \tag{3.47}$$

Schema di Eulero-Cromer Per la simulazione numerica, è stato impiegato il metodo di Eulero nella variante semi-esplicita, detta anche *Eulero-Cromer*, che aggiorna prima la velocità e poi la posizione. Con passo temporale Δt:

$$\begin{aligned} v_{x,n+1} &= v_{x,n} + a_x(x_n, y_n, z_n)\,\Delta t, \qquad & x_{n+1} &= x_n + v_{x,n+1}\,\Delta t, \\ v_{y,n+1} &= v_{y,n} + a_y(x_n, y_n, z_n)\,\Delta t, \qquad & y_{n+1} &= y_n + v_{y,n+1}\,\Delta t, \\ v_{z,n+1} &= v_{z,n} + a_z(x_n, y_n, z_n)\,\Delta t, \qquad & z_{n+1} &= z_n + v_{z,n+1}\,\Delta t. \end{aligned} \tag{3.48}$$

Questo schema, pur rimanendo esplicito, è più stabile dell'Eulero classico e conserva meglio l'energia orbitale nel tempo.

Schema di Runge–Kutta del quarto ordine (RK4). Nel caso della coordinata x:

$$\begin{aligned} k_1^x &= v_{x,n}, & k_1^{v_x} &= a_x(x_n, y_n, z_n), \\ k_2^x &= v_{x,n} + \tfrac{\Delta t}{2}\,k_1^{v_x}, & k_2^{v_x} &= a_x\Big(x_n + \tfrac{\Delta t}{2}\,k_1^x,\; y_n + \tfrac{\Delta t}{2}\,k_1^y,\; z_n + \tfrac{\Delta t}{2}\,k_1^z\Big), \\ k_3^x &= v_{x,n} + \tfrac{\Delta t}{2}\,k_2^{v_x}, & k_3^{v_x} &= a_x\Big(x_n + \tfrac{\Delta t}{2}\,k_2^x,\; y_n + \tfrac{\Delta t}{2}\,k_2^y,\; z_n + \tfrac{\Delta t}{2}\,k_2^z\Big), \\ k_4^x &= v_{x,n} + \Delta t\,k_3^{v_x}, & k_4^{v_x} &= a_x\Big(x_n + \Delta t\,k_3^x,\; y_n + \Delta t\,k_3^y,\; z_n + \Delta t\,k_3^z\Big). \end{aligned} \tag{3.49}$$

Le stesse relazioni valgono per le componenti y e z. L'aggiornamento finale è dato da:

$$\begin{aligned}
x_{n+1} &= x_n + \frac{\Delta t}{6}\big(k_1^x + 2k_2^x + 2k_3^x + k_4^x\big), \quad v_{x,n+1} = v_{x,n} + \frac{\Delta t}{6}\big(k_1^{v_x} + 2k_2^{v_x} + 2k_3^{v_x} + k_4^{v_x}\big),\\
y_{n+1} &= y_n + \frac{\Delta t}{6}\big(k_1^y + 2k_2^y + 2k_3^y + k_4^y\big), \quad v_{y,n+1} = v_{y,n} + \frac{\Delta t}{6}\big(k_1^{v_y} + 2k_2^{v_y} + 2k_3^{v_y} + k_4^{v_y}\big),\\
z_{n+1} &= z_n + \frac{\Delta t}{6}\big(k_1^z + 2k_2^z + 2k_3^z + k_4^z\big), \quad v_{z,n+1} = v_{z,n} + \frac{\Delta t}{6}\big(k_1^{v_z} + 2k_2^{v_z} + 2k_3^{v_z} + k_4^{v_z}\big).
\end{aligned} \tag{3.50}$$

Schema di Verlet Per i sistemi conservativi, è stato infine utilizzato il metodo di Verlet, che aggiorna le posizioni tramite la posizione precedente:

$$\begin{aligned}
x_{n+1} &= 2x_n - x_{n-1} + a_x(x_n, y_n, z_n)\,\Delta t^2, \qquad v_{x,n} \approx \frac{x_{n+1} - x_{n-1}}{2\,\Delta t},\\
y_{n+1} &= 2y_n - y_{n-1} + a_y(x_n, y_n, z_n)\,\Delta t^2, \qquad v_{y,n} \approx \frac{y_{n+1} - y_{n-1}}{2\,\Delta t},\\
z_{n+1} &= 2z_n - z_{n-1} + a_z(x_n, y_n, z_n)\,\Delta t^2, \qquad v_{z,n} \approx \frac{z_{n+1} - z_{n-1}}{2\,\Delta t}.
\end{aligned} \tag{3.51}$$

La posizione al passo precedente viene inizializzata come:

$$\begin{aligned}
x_{-1} &= x_0 - v_{x,0}\,\Delta t + \tfrac{1}{2}\,a_x(x_0, y_0, z_0)\,\Delta t^2,\\
y_{-1} &= y_0 - v_{y,0}\,\Delta t + \tfrac{1}{2}\,a_y(x_0, y_0, z_0)\,\Delta t^2,\\
z_{-1} &= z_0 - v_{z,0}\,\Delta t + \tfrac{1}{2}\,a_z(x_0, y_0, z_0)\,\Delta t^2.
\end{aligned} \tag{3.52}$$

Lo schema di Verlet, pur essendo di ordine $O(\Delta t^2)$, conserva molto bene l'energia meccanica del sistema ed è ideale per integrazioni di lunga durata.

Capitolo 4

Implementazione e architettura del sistema

Il progetto è stato concepito come una piattaforma per la simulazione di fenomeni fisici, con l'obiettivo di unire rigore scientifico, interattività e rappresentazione visiva. Oltre alla mera riproduzione di esperimenti reali, la piattaforma è stata progettata per consentire all'utente di **interagire attivamente con i sistemi simulati**, manipolando i parametri e osservando in tempo reale le conseguenze delle proprie azioni. Questa possibilità rappresenta un aspetto di forte valore educativo e divulgativo, poiché consente di esplorare fenomeni che, nel mondo reale, sarebbero difficilmente osservabili o completamente impossibili da replicare in condizioni controllate. Attraverso tali interazioni, non si è limitati a osservare un fenomeno fisico, ma se ne diventa parte attiva, esplorando relazioni di causa-effetto e approfondendo la comprensione dei principi che lo governano. L'intero ecosistema è stato pensato per favorire un approccio sperimentale e immersivo alla fisica, dove il confine tra teoria e pratica viene superato grazie alla mediazione della simulazione digitale e della realtà aumentata.

4.1 Pendolo di Foucault – Rotazione e sincronizzazione visiva

Il modello matematico del pendolo, descritto nel capitolo precedente, è espresso dal sistema:

$$\dot{\theta} = \omega, \qquad \dot{\omega} = -\frac{g}{L}\sin\theta,$$

Questo sistema di equazioni differenziali non lineari descrive un moto oscillatorio periodico, ma la sua integrazione numerica richiede attenzione: la qualità della simulazione dipende fortemente dalla gestione del passo temporale, dal metodo scelto e dalla capacità di preservare l'energia meccanica nel tempo. Nella simulazione del pendolo di Foucault in realtà aumentata, l'obiettivo non è soltanto riprodurre l'equazione del moto, ma farlo in modo da poter confrontare diversi integratori numerici, mantenendo coerenza visiva e interattiva con la fisica reale. Il sistema è suddiviso in tre blocchi principali:

1. Modello fisico puro: contenente le grandezze dinamiche e statiche del pendolo. È un modulo completamente indipendente da Unity, concepito come un modello matematico astratto.

2. Motore numerico: si occupa dell'integrazione temporale delle equazioni. Implementa i diversi metodi di integrazione e gestisce l'adattamento del passo tramite *sub-stepping*, mantenendo il controllo della stabilità numerica anche quando il frame rate varia.

3. Orchestrazione e resa grafica: è la parte visiva e di coordinamento, gestita da Unity. Traduce lo stato fisico (angolo e lunghezza) in posizioni spaziali, aggiornando il trasform della massa e la linea che rappresenta il filo.

Per cui, il modulo del pendolo è concepito come una catena di responsabilità ben definita:

- il modello mantiene lo stato fisico,
- l'integratore calcola i nuovi valori,
- il numerico gestisce i tempi e la stabilità,

- il controller orchestra e visualizza.

Alla base della simulazione si trova il modello fisico del pendolo, che rappresenta il cuore del sistema. In questo caso, la grandezza da integrare è composta da due gruppi di variabili concettualmente distinti:

- Parametri fisici, cioè le costanti che caratterizzano l'esperimento (massa, lunghezza del filo, accelerazione di gravità)
- Stato dinamico, ovvero le variabili che evolvono nel tempo (angolo e velocità angolare).

Dal punto di vista dell'architettura, queste due famiglie di informazioni hanno una natura diversa: i parametri sono immutabili durante la simulazione e possono essere condivisi da più istanze del modello, mentre lo stato è continuamente aggiornato dagli integratori e non può essere considerato stabile. Questo si riflette nell'implementazione attraverso due componenti fondamentali:

- `PendulumParameters` che rappresenta i parametri fisici, immutabili
- `PendulumState` che rappresenta lo stato dinamico, istantaneo

Ciascuna di queste classi espone una propria interfaccia, definendo i metodi e le proprietà per accedere ai rispettivi dati. La classe `Pendulum` rappresenta una Facade che, presentandosi esternamente come un oggetto unico, concentra in sé due concetti distinti ma correlati, fornendo sia i metodi per accedere ai parametri, sia quelli per interrogare e aggiornare lo stato del pendolo. Svolge, quindi, la funzione di facciata aggregante, implementando al suo interno entrambe le interfacce `IPendulumParameters` e `IPendulumState` e mascherando l'esistenza di due oggetti separati.

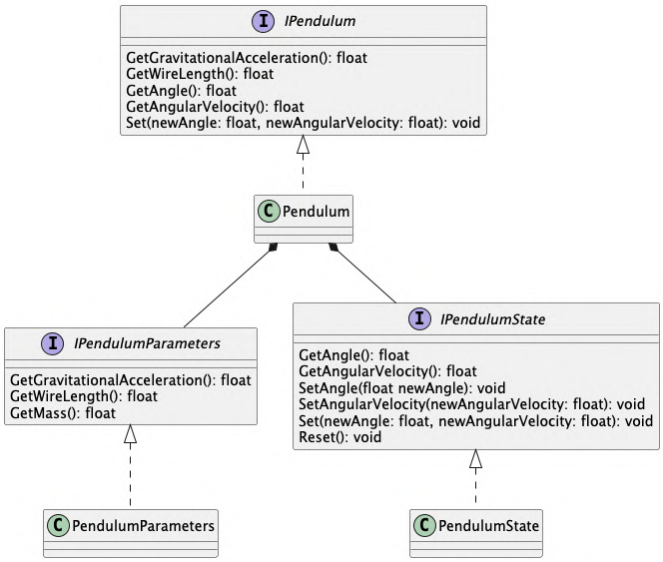


Figura 4.1: Diagramma UML del Facade nel Pendolo di Foucault

Dal punto di vista del chiamante, ottenere o modificare informazioni si riduce al solo dialogo con un solo oggetto tramite l'interfaccia `IPendulum`, creando l'isolamento del sottosistema. Si potrebbe sostituire l'intera implementazione di `PendulumParameters` e `PendulumState`, senza toccare il resto dell'applicazione, a patto di mantenere stabile l'interfaccia unificante. Questo incarna il principio del Facade di fornire debole dipendenza: il codice client dipende solo dall'interfaccia `IPendulum` e non dalle particolarità delle due classi.

La scelta dell'integratore influisce in modo determinante sulla precisione e sulla stabilità del risultato. Al fine di confrontare questi metodi in condizioni identiche, l'architettura è stata progettata per consentire la selezione e la sostituzione dell'integratore in modo dinamico, senza alterare il codice principale della simulazione. Questa esigenza ha portato all'adozione del pattern Strategy, permettendo di incapsulare algoritmi intercambiabili dietro a un'interfaccia comune. L'interfaccia comune `IPendulumIntegrator` dichiara un singolo metodo che rappresenta il contratto per l'avanzamento dello stato fisico di un singolo passo temporale. Ogni implementazione concreta fornisce la propria logica di aggiornamento coerente con le leggi del moto.

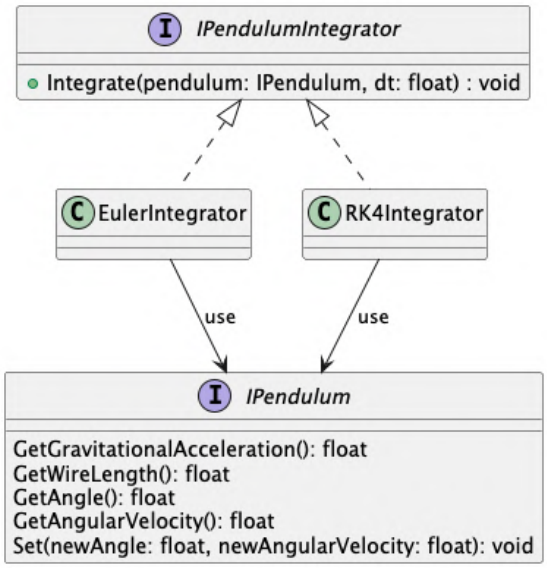


Figura 4.2: Diagramma UML dello Strategy nel Pendolo di Foucault

Le implementazioni concrete (`EulerIntegrator`, `RK4Integrator`) applicano gli aggiornamenti coerenti con le equazioni del modello, lasciando invariata l'orchestrazione. Questo realizza il pattern *Strategy*: il controller non dipende da implementazioni specifiche ma da un'astrazione stabile, permettendo sostituzione dell'integratore senza impatti sul resto del sistema. `EulerIntegrator` applica la formula più semplice dell'integratore esplicito:

Listing 4.1: Implementazione di Eulero

```
public void Integrate(IPendulum pendulum, float dt)
{
    float g = pendulum.GetGravitationalAcceleration();
    float L = pendulum.GetWireLength();

    float theta = pendulum.GetAngle();
    float omega = pendulum.GetAngularVelocity();

    float dTheta = omega;
    float dOmega = -(g / L) * Mathf.Sin(theta);

    pendulum.Set(theta + dTheta * dt,
                 omega + dOmega * dt);
}
```

e `RK4Integrator` divide l'intervallo in 4 parti:

Listing 4.2: Implentazione di RK4

```
public void Integrate(IPendulum pendulum, float dt)
{
    float g = pendulum.GetGravitationalAcceleration();
    float L = pendulum.GetWireLength();

    float theta0 = pendulum.GetAngle();
    float omega0 = pendulum.GetAngularVelocity();

    float k1Theta = omega0;
    float k1Omega = -(g / L) * Mathf.Sin(theta0);

    float k2Theta = omega0 + 0.5f * dt * k1Omega;
    float k2Omega = -(g / L) * Mathf.Sin(theta0 + 0.5f * dt * k1Theta);

    float k3Theta = omega0 + 0.5f * dt * k2Omega;
    float k3Omega = -(g / L) * Mathf.Sin(theta0 + 0.5f * dt * k2Theta);

    float k4Theta = omega0 + dt * k3Omega;
    float k4Omega = -(g / L) * Mathf.Sin(theta0 + dt * k3Theta);

    float theta = theta0 + (dt / 6f) * (k1Theta + 2f*k2Theta + 2f*k3Theta +
        k4Theta);
    float omega = omega0 + (dt / 6f) * (k1Omega + 2f*k2Omega + 2f*k3Omega +
        k4Omega);

    pendulum.Set(theta, omega);
}
```

Entrambi implementano la stessa interfaccia, differendo unicamente nel modo in cui calcolano i nuovi valori θ e ω. Quando la simulazione inizia, il controller sceglie l'integratore e il pattern Strategy garantisce la corretta delega.

Una volta definito il modello fisico del pendolo e la strategia di integrazione, rimane da risolvere un problema centrale: il controllo del passo temporale. Nei sistemi reali, l'evoluzione temporale è continua; in una simulazione digitale, invece, il tempo è campionato a intervalli discreti Δt. Se il passo scelto è troppo grande, la simulazione diventa instabile o imprecisa; se è troppo piccolo, diventa inefficiente. In un contesto di realtà aumentata, dove il frame rate può variare dinamicamente, questo problema si amplifica poiché Unity non garantisce un passo fisso tra due aggiornamenti consecutivi. Per isolare questa complessità, è stato introdotto un componente intermedio: la classe `PendulumNumerical`, che funge da wrapper, da *Adapter*, tra il modello fisico (`Pendulum`) e l'integratore

(`IPendulumIntegrator`), adattando il ciclo temporale del motore Unity al dominio fisico del pendolo. `PendulumNumerical` è un mediatore numerico: riceve da Unity il tempo trascorso nell'ultimo frame, lo normalizza in base alla scala fisica del sistema e decide quanti passi d'integrazione eseguire internamente; il suo compito è quello di coordinare il ritmo della simulazione.
Per garantire la stabilità numerica, è stato implementato un sistema di substepping, basato sulla frequenza naturale del pendolo, che adatta dinamicamente il passo di integrazione al tempo fisico del sistema.

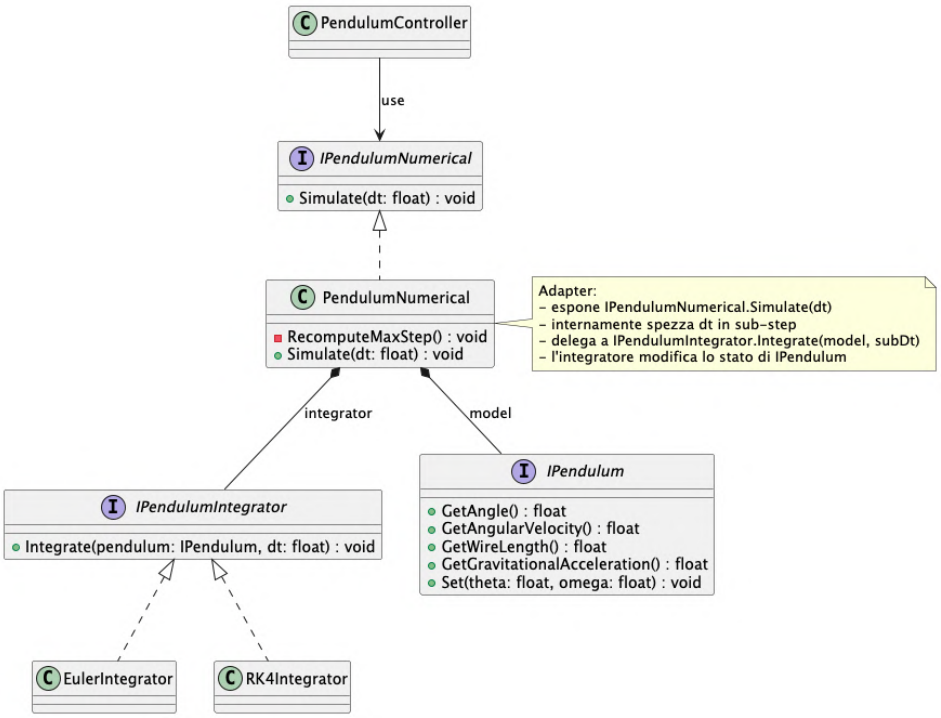


Figura 4.3: Diagramma UML dell'Adapter nel Pendolo di Foucault

Unity offre due cadenze logiche distinte per l'aggiornamento dell'esecuzione:

- **Update** è invocata una volta per ogni frame renderizzato, l'intervallo di tempo tra due chiamate è generalmente variabile e viene esposto dalla proprietà `Time.deltaTime` [47]
- **FixedUpdate**, avanza la simulazione con passo costante, definito da `Time.fixedDeltaTime`, con un valore predefinito di $0,02s$, indipendentemente dalla frequenza di rendering [48]. È importante sottolineare che la costanza

riguarda il tempo simulato, non quello reale: se il rendering rallenta, all'inizio del frame Unity può eseguire più chiamate consecutive a FixedUpdate per recuperare i passi fissi maturati dall'ultimo frame. Questo fenomeno prende il nome di meccanismo di *catch-up*.

Unity espone un limite superiore al passo variabile di Update, chiamato **Maximum Allowed Timestep**, con proprietà `Time.maximumDeltaTime`. Quando un frame risulta eccezionalmente lento, per esempio per operazioni AR, Unity clampa il valore di `Time.deltaTime`. Analogamente il medesimo meccanismo limita le FixedUpdate, quindi passi fissi di fisica, possono essere eseguiti all'interno di un singolo frame. Queste due azioni mitigano due problemi tipici dei salti temporali:

- **Teletrasferimento**, salto visibile dovuto a un `deltaTime` enorme, una transizione violenta e non sotto controllo
- **Spirale della morte**, accumulo di fisica che peggiora ulteriormente il frame, in quanto si innesca il circolo vizioso: *frame più lungo* $\Rightarrow$ *più FixedUpdate* $\Rightarrow$ *frame successivo ancora più lungo* $\Rightarrow \ldots$

Vincolare `deltaTime` e limitare il catch-up significa accettare che una frazione di tempo reale non venga simulata integralmente quando si verificano questi picchi estremi [49]. Questo modello temporale implica due conseguenze pratiche per un solver numerico personalizzato indipendente dal motore fisico PhysX:

1. se l'integrazione avviene in `Update`, il passo di integrazione dt varia in base al carico del rendering da un frame all'altro
2. se l'integrazione avviene in `FixedUpdate`, la simulazione avanza con un passo di gioco costante, ma il *catch-up* può concentrare più integrazioni nello stesso frame, mentre la telecamera renderizza con una frequenza diversa, causando disallineamenti percettivi con il rendering.

In entrambi i casi, la soluzione deve essere progettata per funzionare correttamente con passi effettivi irregolari. Da qui nasce l'esigenza di introdurre un **substepping** controllato dal modello fisico. Invece di imporre un passo numerico arbitrario, viene ancorato il passo massimo dell'integrazione alla scala temporale

intrinseca del sistema, cosicché l'accuratezza numerica segua la fisica del problema. Per il pendolo semplice, nel regime di piccoli angoli, la dinamica è caratterizzata dalla frequenza naturale:

$$\omega_0 = \sqrt{\frac{g}{L}} \qquad T_0 \approx \frac{2\pi}{\omega_0} = 2\pi\sqrt{\frac{L}{g}} \tag{4.1}$$

Fissato un numero di passi per periodo N_{pp} e definito un passo massimo per la sotto-integrazione

$$\Delta t_{\max} = \frac{T_0}{N_{pp}} = \frac{2\pi}{N_{pp}}\sqrt{\frac{L}{g}} \tag{4.2}$$

Listing 4.3: Implementazione di RecomputeMaxStep

```
private const int MAX_NUM_STEP_PER_PERIOD = 100;

private void RecomputeMaxStep()
{
    float L = model.GetWireLength();
    float g = model.GetGravitationalAcceleration();

    float T0 = 2f * Mathf.PI * Mathf.Sqrt(L / g);
    maxStep = T0 / MAX_NUM_STEP_PER_PERIOD;
}
```

Dato il passo di frame dt, si pone:

$$steps = \left\lceil \frac{dt}{\Delta t_{\max}} \right\rceil \qquad subDt = \frac{dt}{steps}$$

Se $dt \leq \Delta t_{max}$ viene eseguito un solo passo numerico; in caso contrario viene suddiviso dt in *steps* numero di sottopassi della durata *subDt*. In questo modo, la dimensione del passo rimane sempre compatibile con la scala w_0, riducendo l'errore di fase e contenendo il drift energetico degli integratori espliciti, anche in presenza di variazioni del frame time. Il sub-stepping guidato dal modello mantiene, quindi, automaticamente il passo di integrazione al di sotto di Δt_{max}, senza introdurre alcun overhead nel caso in cui il valore dt resti al di sotto della soglia. Inoltre, normalizzando il passo rispetto alla scala fisica del sistema, viene reso equo il confronto tra gli integratori: tutti i metodi operano sotto la medesima

soglia determinata dalla fisica e non dal frame rate o dall'hardware, in modo che le differenze misurate in termini di accuratezza (errore di fase, ampiezza, drift) e di costo computazionale siano realmente attribuibili al metodo numerico.

Listing 4.4: Implementazione dei sub-steps

```
public PendulumNumerical(IPendulum model, IPendulumIntegrator integrator)
{
    this.model = model;
    this.integrator = integrator;

    RecomputeMaxStep();
}

public void Simulate(float dt)
{
    if (dt <= 0f || float.IsNaN(dt) || float.IsInfinity(dt))
        return;

    int steps = Mathf.Max(1, Mathf.CeilToInt(dt / maxStep));
    float subDt = dt / steps;

    for (int i = 0; i < steps; i++)
        integrator.Integrate(model, subDt);
}
```

L'idea è che l'integratore resti "ignorante" del contesto Unity: non conosce i frame né la durata effettiva del rendering; lavora sempre su un passo coerente con la fisica del problema.

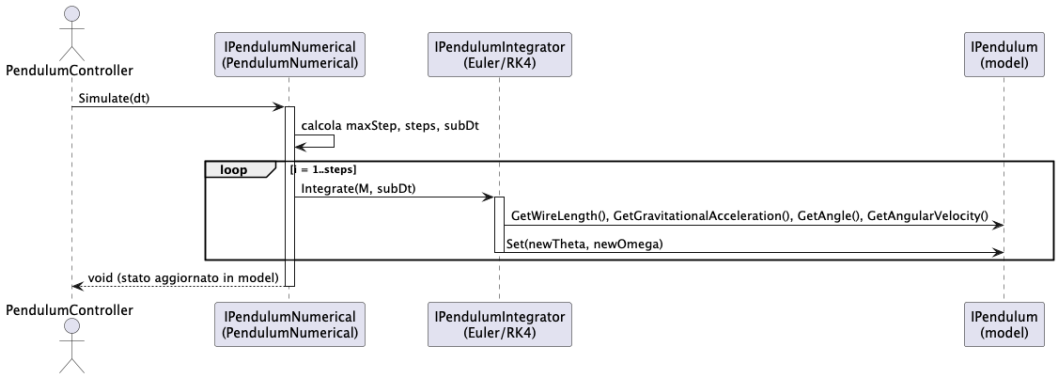


Figura 4.4: Flusso che illustra il funzionamento del modello fisico del pendolo

Questo diagramma mostra come, per ogni frame, il controller deleghi l'intero avanzamento a `PendulumNumerical`, che a sua volta può eseguire più sotto-passi fisici

all'interno dello stesso intervallo temporale.

L'orchestratore della scena è `PendulumController`: riceve il passo di tempo Δt, chiede al wrapper numerico di avanzare la dinamica del modello, poi aggiorna la resa grafica e registra l'energia totale per l'analisi del metodo.
All'avvio, il controller:

1. costruisce il modello tramite Factory

Listing 4.5: Implementazione di `PendulumFactory`

```
public sealed class PendulumFactory : IPendulumFactory
{
    private const float EARTH_GRAVITY = 9.80665f;
    private const float FOUCAULT_LENGTH_METERS = 67f;
    private const float FOUCAULT_MASS = 28f;

    private static IPendulum CreatePendulum(
        float mass,
        float wireLength,
        float gravity,
        float angle,
        float angularVelocity)
    {
        var parameters = new PendulumParameters(mass, wireLength, gravity);
        var state = new PendulumState(angle, angularVelocity);
        return new Pendulum(parameters, state);
    }

    public IPendulum CreateFoucault(
        float initialAngle,
        float initialAngularVelocity)
    {
        return CreatePendulum(
            mass: FOUCAULT_MASS,
            wireLength: FOUCAULT_LENGTH_METERS,
            gravity: EARTH_GRAVITY,
            angle: initialAngle,
            angularVelocity: initialAngularVelocity
        );
    }
}
```

2. sceglie l'integratore in base a una enum

Listing 4.6: Implementazione del sistema numerico

```
IPendulumFactory factory = new PendulumFactory();
model = factory.CreateFoucault(angleRad, INITIAL_ANGULAR_VELOCITY);

IPendulumIntegrator integrator = integratorType switch
{
    IntegratorType.Euler => new EulerIntegrator(),
    IntegratorType.RK4   => new RK4Integrator(),
    _                    => new RK4Integrator()
};

numerical = new PendulumNumerical(model, integrator);
```

3. avvolge il tutto nel wrapper numerico che gestisce il passo

Listing 4.7: Implementazione del ciclo di simulazione del pendolo

```
public void Step(float deltaTime)
{
    if (deltaTime <= 0f || float.IsNaN(deltaTime) || float.IsInfinity(
        deltaTime))
        return;

    numerical.Simulate(deltaTime);
    simulatedTime += deltaTime;

    UpdateMassTransformFromModel();
    UpdateLine();

    float energy = ComputeTotalEnergy(out float theta, out float omega);
    csvLogger?.Log(simulatedTime, theta, omega, energy);
}
```

Base Rotante

La base rotante rappresenta la componente del sistema che simula la rotazione terrestre e costituisce l'elemento di riferimento per l'intero esperimento del Pendolo di Foucault. Ha il compito di ruotare a velocità costante e di sincronizzare un anello di led disposti lungo la circonferenza, che vengono attivati progressivamente per evidenziare visivamente la rotazione. Questa soluzione consente di riprodurre l'effetto del moto relativo del piano di oscillazione rispetto alla Terra.

Dal punto di vista concettuale, incapsula due funzioni principali:

- **rotazione fisica** della base, definita da un periodo $T = 86.400s$, corrispondente a un giorno siderale
- **logica di illuminazione**, che determina quali led devono essere attivati in base all'angolo di rotazione corrente

Dal punto di vista implementativo, i componenti chiave sono:

- `ActivationLogic` che calcola quali LED devono accendersi man mano che la base ruota
- `LedLightingSystem` che coordina la logica di attivazione e rotazione con i LED effettivi
- `LightSwitcher` che accende e spegne fisicamente i LED nella scena
- `RotatingSupportManager`, l'unico script MonoBehaviour che collega il tutto, aggiornando la rotazione e i led ad ogni frame.

Mentre la base ruota, i led si accendono in sequenza per marcare il passare del tempo, azzerandosi a ogni rotazione completa.

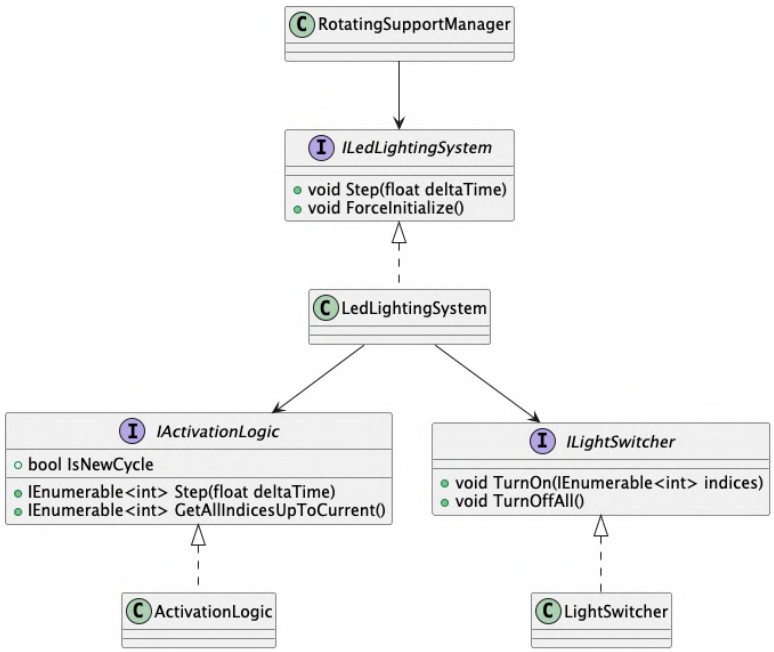


Figura 4.5: Diagramma UML della Base Rotante

La classe `ActivationLogic` calcola la velocità angolare a partire dal periodo di rotazione imposto:

$$\omega = \frac{2\pi}{T}$$

dove T rappresenta il periodo in secondi. Ad ogni aggiornamento, l'angolo di rotazione corrente θ viene incrementato di $\omega \cdot \Delta t$, mantenendo il valore nel range $[0, 2\pi)$. Il sistema traduce poi questo angolo in un indice led attivo secondo:

$$i = \left\lfloor \frac{\theta}{2\pi} \cdot N \right\rfloor$$

dove N è il numero totale di led. Quando l'indice corrente risulta minore di quello precedente, significa che è iniziato un nuovo ciclo di rotazione: la logica disattiva tutti i led e riparte da zero, assicurando la continuità visiva del movimento circolare. L'elenco di indici da accendere viene quindi passato alla classe `LightSwitcher`.

La classe `LightSwitcher` si occupa dell'accensione e spegnimento dei led, cambiandone il materiale. Mantiene un elenco di oggetti `Renderer`, selezionati auto-

maticamente tramite uno script `Editor`. Le funzionalità principali includono:

- `TurnOn(IEnumerable<int> indices)`: imposta il materiale di quel renderer a `ledOn`, "accendendo" visivamente il led corrispondente
- `TurnOffAll()`: spegne tutti i led, iterando su ogni renderer nella lista e assegnandogli il materiale di off.

Listing 4.8: Implementazione di `LightSwitcher`

```
public class LightSwitcher : ILightSwitcher
{
    private readonly Material ledOff;
    private readonly Material ledOn;
    private readonly List<Renderer> leds;

    public LightSwitcher(List<Renderer> leds, Material on, Material off)
    {
        this.leds = leds;
        ledOn = on;
        ledOff = off;

        TurnOffAll();
    }

    public void TurnOn(IEnumerable<int> indices)
    {
        foreach (var i in indices)
            if (i >= 0 && i < leds.Count && leds[i] != null)
                leds[i].sharedMaterial = ledOn;
    }

    public void TurnOffAll()
    {
        for (int i = 0; i < leds.Count; i++)
            TurnOff(i);
    }

    private void TurnOff(int i)
    {
        if (i >= 0 && i < leds.Count && leds[i] != null)
            leds[i].sharedMaterial = ledOff;
    }
}
```

`LightSwitcher` isola i dettagli visivi dalla logica riguardante *quali* led accendere o spegnere. Il resto del sistema gli comunicherà solo gli indici dei led da accende-

re/spegnere, e `LightSwitcher` eseguirà il compito di aggiornare i materiali di quei led.

Listing 4.9: Implementazione di `LedLightingSystem`

```
public sealed class LedLightingSystem : ILedLightingSystem
{
    private readonly IActivationLogic logic;
    private readonly ILightSwitcher switcher;

    private LedLightingSystem(IActivationLogic logic, ILightSwitcher switcher)
    {
        this.logic = logic;
        this.switcher = switcher;
    }

    public static ILedLightingSystem Create(
        List<Renderer> leds,
        Material ledOn,
        Material ledOff,
        int ledCount,
        float rotationPeriodSeconds,
        float initialSimulatedTime,
        int indexOffset = 0)
    {
        var activation = new ActivationLogic(
            ledCount, rotationPeriodSeconds, initialSimulatedTime, indexOffset);

        var lightSwitcher = new LightSwitcher(leds, ledOn, ledOff);
        return new LedLightingSystem(activation, lightSwitcher);
    }

    public void Step(float deltaTime)
    {
        var indices = logic.Step(deltaTime);
        if (logic.IsNewCycle)
            switcher.TurnOffAll();
        switcher.TurnOn(indices);
    }

    public void ForceInitialize()
    {
        var indices = logic.GetAllIndicesUpToCurrent();
        switcher.TurnOffAll();
        switcher.TurnOn(indices);
    }
}
```

Se `logic.IsNewCycle` risulta true dopo lo step, significa che si è completata una

rotazione intera e, in tal caso, prima di accendere nuovi led, il sistema esegue un reset: spegne tutti i led esistenti, preparando l'inizio di un nuovo ciclo. Se invece non è iniziato un nuovo ciclo, il metodo si limita ad accendere i nuovi led senza spegnere quelli già accesi.

`RotatingSupportManager` è uno script MonoBehaviour che connette la logica al motore Unity. Viene calcolato il numero di secondi trascorsi dall'inizio della giornata, stimato a mezzanotte, fino all'ora in cui la simulazione ha inizio. Questo valore viene poi utilizzato come tempo simulato iniziale: allinea la simulazione con l'ora reale corrente. All'avvio, la base rotante e l'illuminazione led partono già nello stato corretto per rappresentare la porzione di giornata già trascorsa. Ad esempio, se sono le 6:00, la base partirà ruotata di 90° e la relativa frazione di led accesi. Il risultato è che, ad ogni frame, la base si sposta di un piccolo incremento angolare e, se necessario, il led successivo si accende. A mezzanotte, l'arco raggiungerà l'intero cerchio (tutti i led accesi); subito dopo, il sistema spegne tutti i led e riparte, con la base che continua a ruotare e i led che riprendono ad accendersi da capo. Questo crea un ciclo continuo giorno-notte, visualizzato dalla rotazione della base e dall'illuminazione progressiva dei led.

Listing 4.10: Implementazione di `RotatingSupportManager`

```
public class RotatingSupportManager : MonoBehaviour
{
    [SerializeField] private Transform baseTransform;

    public List<Renderer> leds = new();
    [SerializeField] private Material ledOnMaterial;
    [SerializeField] private Material ledOffMaterial;

    private float rotationPeriodSeconds = 86400f;
    private ILedLightingSystem ledSystem;
    private float simulatedTime;

    private void Awake()
    {
        if (baseTransform == null)
            baseTransform = transform;
    }

    private void Start()
    {
```

```
21          var initialRealTime = (float)(DateTime.Now - DateTime.Now.Date).
                TotalSeconds;
22          simulatedTime = initialRealTime;
23
24          ledSystem = LedLightingSystem.Create(
25              leds,
26              ledOnMaterial,
27              ledOffMaterial,
28              leds.Count,
29              rotationPeriodSeconds,
30              simulatedTime
31          );
32
33          ledSystem.ForceInitialize();
34          ApplyRotation();
35      }
36
37      private void Update()
38      {
39          Step(Time.deltaTime);
40      }
41
42      public void Step(float deltaTime)
43      {
44          var scaledDelta = SimulationTimeManager.Instance != null
45              ? SimulationTimeManager.Instance.GetDeltaTime()
46              : deltaTime;
47
48          simulatedTime += scaledDelta;
49
50          ApplyRotation();
51          ledSystem?.Step(scaledDelta);
52      }
53
54      private void ApplyRotation()
55      {
56          var period = Mathf.Max(0.0001f, rotationPeriodSeconds);
57          var degrees = (simulatedTime / period * 360f) % 360f;
58          if (degrees < 0f)
59              degrees += 360f;
60
61          baseTransform.localRotation = Quaternion.Euler(0f, degrees, 0f);
62      }
63  }
```

Desk Informativo – Traccia del Pendolo di Foucault

Il Desk Informativo rappresenta la parte interattiva: un tavolo digitale che consente di esplorare il comportamento del pendolo di Foucault in diversi punti della Terra. L'utente può selezionare una località sul globo virtuale e osservare in tempo reale come varia la rotazione del piano di oscillazione del pendolo in base alla latitudine scelta. Il sistema fornisce così un riscontro visivo immediato sull'effetto di Coriolis: ai poli la precessione è massima, mentre all'equatore il piano rimane fisso. Dal punto di vista implementativo, `DrawFoucault` integra le equazioni del moto nel piano e accumula i punti della triettoria. Il piccolo motore fisico interno alla classe (`StepRK4`) integra con Runge–Kutta d'ordine 4 le ODE accoppiate del pendolo soggetto al termine di Coriolis latitudinale.

Per rendere l'effetto percepibile in pochi minuti di simulazione, vengono introdotti due fattori di scala:

- **TIME_SCALE**: amplifica il Δt della simulazione.
- **PRECESSION_SPEEDUP**: moltiplica la componente di Coriolis per aumentare visivamente la velocità di precessione.

Il metodo `StepRK4` calcola la nuova posizione (x, y) e le velocità (v_x, v_y) del pendolo al passo successivo, includendo il termine di Coriolis opportunamente scalato. Il risultato viene poi convertito in un punto 3D e aggiunto alla lista di posizioni del `LineRenderer`, che disegna la traccia.

Listing 4.11: Implementazione dell'aggiornamento della traccia nel Desk

```
public void Update(float deltaTime)
{
    StepRK4(deltaTime);

    var p = new Vector3(x * DRAW_SCALE, LINE_Y_OFFSET, y * DRAW_SCALE);
    if (points.Count == 0 || Vector3.Distance(points[^1], p) >= minDistance)
    {
        points.Add(p);
        if (points.Count > maxPoints) points.RemoveAt(0);
        lr.positionCount = points.Count;
        lr.SetPositions(points.ToArray());
    }
}
```

4.2 GraviLab – Simulazione interattiva della gravità

Questa simulazione ha lo scopo di riprodurre il moto di caduta libera, permettendo di selezionare la gravità e il modello di resistenza del mezzo. Il sistema può essere suddiviso nei seguenti blocchi:

- Gestione della gravità: un modulo che mantiene il valore corrente dell'accelerazione e notifica eventuali cambiamenti
- Parametri degli oggetti: una struttura dati che raccoglie i dettagli di ogni oggetto fisico disponibile
- Modello fisico della caduta: l'astrazione del modello matematico che calcola l'evoluzione della velocità in caduta, con diverse implementazioni possibili
- Core di simulazione dell'oggetto: la logica che applica il modello fisico a un singolo oggetto in caduta
- Orchestrazione e simulazione: si occupa di creare gli oggetti in base alle selezioni dell'utente e garantisce la gestione di più oggetti in parallelo

Alla base del sistema `GravityManager` si occupa di mantenere il valore dell'accelerazione di gravità usata nella simulazione. Questa classe contiene una proprietà `Gravity` e metodi per modificarla, sia in modo incrementale (`Increase()`, `Decrease()`) sia impostandola direttamente a valori predefiniti (`SetEarth()`, `SetMoon()`, `SetJupiter()`) corrispondenti alla gravità terrestre, lunare, gioviana. Ogni volta che il valore viene cambiato, `GravityManager` invoca un evento `OnGravityChanged` notificando il nuovo valore a eventuali al listener. Questo meccanismo implementa il pattern Observer, permettendo ad altri componenti di aggiornare le proprie informazioni senza che `GravityManager` conosca i dettagli dell'UI.

Listing 4.12: Interfaccia `IGravityManager`

```
public interface IGravityManager
{
    float Gravity { get; }
    event Action<float> OnGravityChanged;
    void Increase();
    void Decrease();
    void SetEarth();
    void SetMoon();
    void SetJupiter();
    void Set(float value);
}
```

Listing 4.13: Implementazione di `GravityManager`

```
public class GravityManager : IGravityManager
{
    public float Gravity { get; private set; } = 9.81f;
    public event Action<float> OnGravityChanged;

    public void Increase() => Update(Gravity + 1f);
    public void Decrease() => Update(Gravity - 1f);
    public void SetEarth() => Update(9.81f);
    public void SetMoon() => Update(1.62f);
    public void SetJupiter() => Update(24.79f);
    public void Set(float value) => Update(value);

    private void Update(float value)
    {
        Gravity = value;
        OnGravityChanged?.Invoke(Gravity);
    }
}
```

Il manager espone solo operazioni di alto livello: i dettagli su come la UI presenti questi valori sono del tutto astratti dall'implementazione.

I parametri fisici e grafici degli oggetti sono centralizzati nella classe `PrefabInfo` e in una collezione `PrefabInfoList`. `PrefabInfo` è un tipo Serializable che funge da contenitore per i dati di un tipo di oggetto. Comprende:

- il riferimento al prefab 3D reale
- un materiale di preview, usato nell'anteprima sulla UI
- parametri visuali (offset rispetto al supporto, scala iniziale)

- parametri fisici, quali massa, area della sezione trasversale e coefficiente di drag aerodinamico

Organizzando questi valori in uno ScriptabeObject (`PrefabInfoList`), risulta facile aggiungere nuovi oggetti all'esperimento o modificare le proprietà senza intervenire sul codice.

I diversi modelli di resistenza del mezzo, sono stati implementati adottando il pattern Strategy, attaverso un'interfaccia comune `IPhysicsModel` con un metodo `ComputeVelocity` che calcola la nuova velocità verticale.

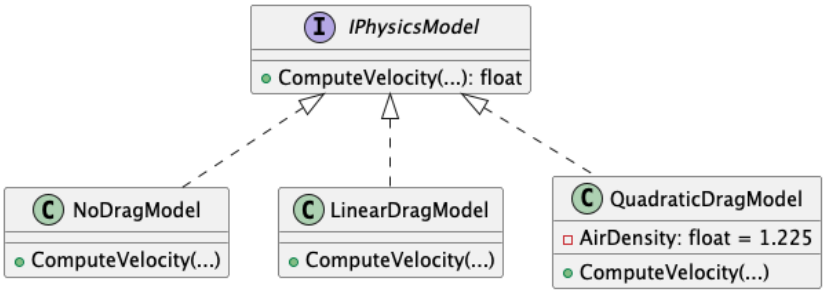


Figura 4.6: Diagramma UML dello Strategy nel GraviLab

Ogni algoritmo di integrazione della velocità è incapsulato in una diversa classe concreta che implementa `IPhysicsModel`, rendendo intercambiabili a runtime senza influenzare gli altri componenti.

- NoDragModel implementa la caduta libera nel vuoto, senza alcuna forza di resistenza

Listing 4.14: Implementazione di `NoDragModel`

```
public float ComputeVelocity(
    float currentVelocity,
    float mass,
    float gravity,
    float deltaTime,
    float areaSection,
    float coefficientDrag)
{
    return currentVelocity - gravity * deltaTime;
}
```

- LinearDragModel: implementa una resistenza del mezzo proporzionale alla velocità

Listing 4.15: Implementazione di `LinearDragModel`

```
public float ComputeVelocity(
    float currentVelocity,
    float mass,
    float gravity,
    float deltaTime,
    float areaSection,
    float coefficientDrag)
{
    float dragForce = coefficientDrag * currentVelocity;
    float acceleration = -gravity - (dragForce / mass);
    return currentVelocity + acceleration * deltaTime;
}
```

- QuadraticDragModel: implementa la resistenza aerodinamica quadratica

Listing 4.16: Implementazione di `QuadraticDragModel`

```
public float ComputeVelocity(
    float currentVelocity,
    float mass,
    float gravity,
    float deltaTime,
    float areaSection,
    float coefficientDrag)
{
    float velocityAbs = Mathf.Abs(currentVelocity);
    float dragForce = 0.5f * AirDensity * coefficientDrag * areaSection *
        velocityAbs * velocityAbs;
    float acceleration = -gravity - (dragForce / mass);
    return currentVelocity + acceleration * deltaTime;
}
```

La classe `FallingObject` funge da motore che integra i modelli fisici selezionabili per simulare la caduta dei corpi all'interno del GraviLab. Ogni oggetto istanziato eredita il comportamento del modello scelto e comunica con il `GravityManager` per l'aggiornamento in tempo reale della gravità.

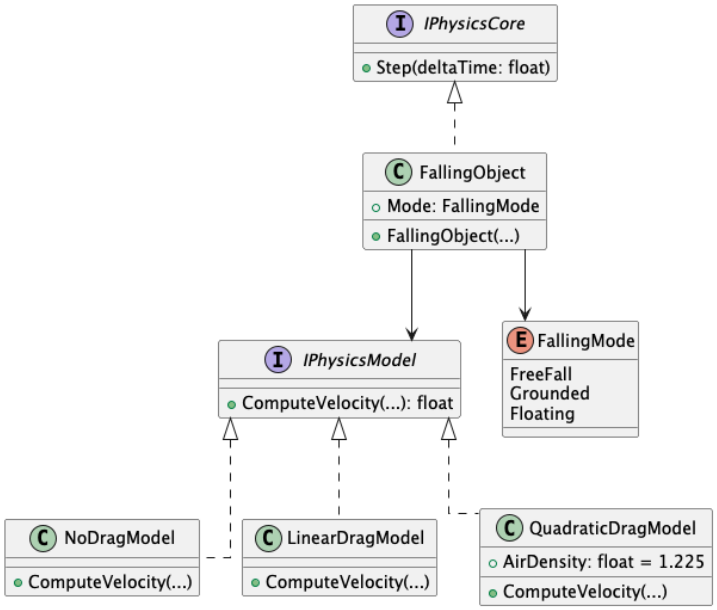


Figura 4.7: Diagramma UML relativo agli oggetti in caduta

Per inizializzare un nuovo oggetto in caduta è stato utilizzato il pattern Factory attraverso la classe `FallingObjectFactory` che accetta tutti i parametri necessari e si occupa internamente di istanziare il modello fisico corretto.

Listing 4.17: Implementazione di `FallingObjectFactory`

```
public class FallingObjectFactory : IPhysicsCoreFactory
{
    public enum PhysicsModelType { NoDrag, LinearDrag, QuadraticDrag }
    private PhysicsModelType currentModelType;

    public FallingObjectFactory(PhysicsModelType type) {
        currentModelType = type;
    }

    public IPhysicsCore Create(
        Transform transform,
        float mass,
        float gravity,
        Vector3 initialVelocity,
        float areaSection,
        float coefficientDrag,
        float groundLevelY)
    {
        IPhysicsModel model;
        switch (currentModelType) {
```

```
            case PhysicsModelType.NoDrag:
                model = new NoDragModel();
                break;
            case PhysicsModelType.LinearDrag:
                model = new LinearDragModel();
                break;
            default:
                model = new QuadraticDragModel();
                break;
        }

        return new FallingObject(
            transform,
            mass,
            gravity,
            initialVelocity,
            areaSection,
            coefficientDrag,
            model,
            groundLevelY);
    }
}
```

Il pattern Strategy e il Factory lavorano quindi in sinergia: il primo definisce come modulare l'algoritmo di integrazione, il secondo si occupa di selezionare e costruire l'istanza appropriata di tale algoritmo.

La classe `SimulationService`, che implementa l'interfaccia `ISimulationService`, mantiene una lista di tutti gli oggetti attualmente in scena (che implementano `IPhysicsCore`) e fornisce metodi per inserire un nuovo oggetto, aggiornare e resettare la simulazione.

Listing 4.18: Implementazione di `SimulationService`

```
public class SimulationService : ISimulationService
{
    private readonly List<IPhysicsCore> simulatables = new();

    public void RegisterSimulatable(IPhysicsCore physicsCore) {
        if (physicsCore != null)
            simulatables.Add(physicsCore);
    }

    public void UpdateSimulation(float deltaTime) {
        foreach (var core in simulatables)
            core.Step(deltaTime);
    }
```

```
14
15      public void ResetSimulation() {
16          simulatables.Clear();
17      }
18  }
```

Il sistema di controllo della lavagna interattiva adotta il Command Pattern per gestire le azioni associate ai pulsanti. Questo approccio separa in modo netto la gestione del click sul bottone e l'effetto prodotto nel sistema. Ogni bottone della lavagna è legato a un oggetto che implementa l'interfaccia `ICommand` che espone un unico metodo pubblico `Execute()`. Ogni comando concreto che eredita questa interfaccia, incapsula un'azione ben definita. I componenti principali sono:

- `ActionCommand`: implementa un comando generico che esegue una semplice Action passata per delega, utilizzabile ad esempio per l'aumento della gravità
- `SupportCommand`: cambia lo slot attivo sulla lavagna
- `OkCommand`: avvia la simulazione fisica, istanziando i modelli e applicando la gravità selezionata
- `ResetCommand`: resetta la scena, ripristinando prefab, slot attivo e stato visivo

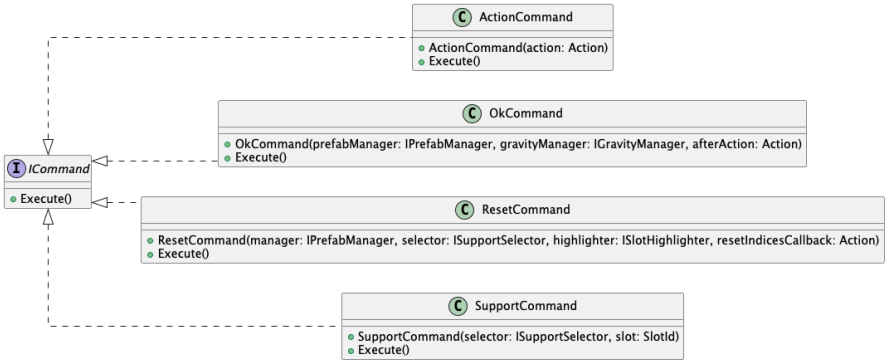


Figura 4.8: Diagramma UML del Command nel Gravilab

Nel controller centrale ogni pulsante è associato a un'istanza di `ICommand` tramite un dizionario che mappa il tipo di bottone all'azione corrispondente

Listing 4.19: Implementazione del dizionario dei bottoni

```
commands = new Dictionary<LIMButton.ButtonType, ICommand>
{
    { LIMButton.ButtonType.Support1, new SupportCommand(supportSelector, SlotId.
        Left) },
    { LIMButton.ButtonType.Support2, new SupportCommand(supportSelector, SlotId.
        Center) },
    { LIMButton.ButtonType.Support3, new SupportCommand(supportSelector, SlotId.
        Right) },

    { LIMButton.ButtonType.ArrowRight, new ActionCommand(ChangePrefabRight) },
    { LIMButton.ButtonType.ArrowLeft,  new ActionCommand(ChangePrefabLeft)  },

    { LIMButton.ButtonType.GravityPlus,    new ActionCommand(gravityManager.
        Increase) },
    { LIMButton.ButtonType.GravityMinus,   new ActionCommand(gravityManager.
        Decrease) },
    { LIMButton.ButtonType.GravityEarth,   new ActionCommand(gravityManager.
        SetEarth) },
    { LIMButton.ButtonType.GravityMoon,    new ActionCommand(gravityManager.
        SetMoon) },
    { LIMButton.ButtonType.GravityJupiter, new ActionCommand(gravityManager.
        SetJupiter) },

    { LIMButton.ButtonType.ModelVacuum, new ActionCommand(() =>
        modelSelector.SetModel(FallingObjectFactory.PhysicsModelType.NoDrag)) },

    { LIMButton.ButtonType.ModelLiquid, new ActionCommand(() =>
        modelSelector.SetModel(FallingObjectFactory.PhysicsModelType.LinearDrag))
            },

    { LIMButton.ButtonType.ModelAir, new ActionCommand(() =>
        modelSelector.SetModel(FallingObjectFactory.PhysicsModelType.QuadraticDrag
            )) },

    { LIMButton.ButtonType.OK, new ActionCommand(StartSimulation) },

    { LIMButton.ButtonType.Reset, new ResetCommand(
        prefabManager,
        supportSelector,
        supportHighlighter,
        ResetPrefabIndices
    )}
};
```

L'ultimo tassello è il controller della scena che che orchestra questi componenti. Al caricamento della scena, vengono creati o recuperati: un'istanza di `GravityManager`, un'istanza di `PhysicsModelSelector` e un `SimulationService`. Quando l'utente

seleziona un oggetto da far cadere, il controller:

1. Legge i parametri fisici dal corrispondente `PrefabInfo`
2. Istanzia il prefab grafico nella scena e ne ottiene il `Transform`.
3. Crea un Factory del tipo desiderato
4. Chiama un nuovo `FallingObject`
5. Registra questo core nel `SimulationService` con `RegisterSimulatable`.

Da quel momento, durante l'evento `Update` di Unity, il controller ad ogni frame invoca `simulationService.UpdateSimulation(Time.deltaTime)`: questo aggiorna tutti gli oggetti in caduta. Se l'utente modifica la gravità globale durante l'esecuzione, `GravityManager.OnGravityChanged` viene catturato dal controller per, ad esempio, aggiornare un'etichetta di testo sulla UI e assicurarsi che i nuovi oggetti creati dopo abbiano il valore aggiornato. Allo stesso modo, per tutti gli altri elementi in scena.

4.3 Sistema Solare – Orbite dinamiche e scaling del Sole

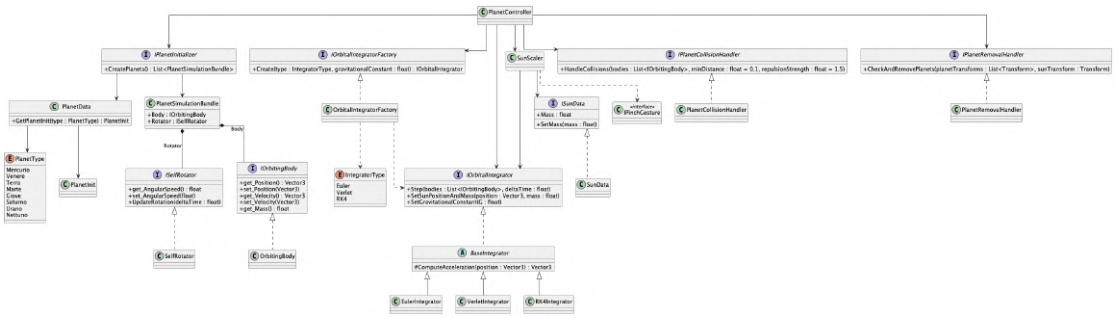


Figura 4.9: Diagramma UML del Sistema Solare

`PlanetController` è il coordinatore principale del sistema. Non implementa nè fisica, nè dati ma coordina i moduli specializzati. Dipende esclusivamente da interfacce, aderendo al principio di inversione delle dipendenze.

- `IOrbitalIntegrator` e `IOrbitalIntegratorFactory`, responsabili della fisica orbitale e della creazione dell'integratore numerico

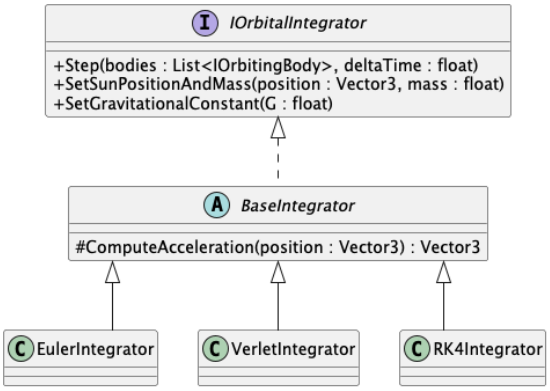


Figura 4.10: Diagramma UML dello Strategy nel Sistema Solare

L'implementazione del modello fisico del sistema solare si basa direttamente sulla formulazione della legge di gravitazione universale di Newton, discussa nel capitolo precedente. Come già evidenziato si adotta l'approssimazione a massa dominante: il Sole è considerato il corpo centrale fisso del sistema, mentre ciascun pianeta viene trattato come una particella puntiforme che orbita sotto l'influenza della gravità solare.

Listing 4.20: Implementazione di `ComputeAcceleration`

```
protected Vector3 ComputeAcceleration(Vector3 position)
{
    var direction = SunPosition - position;
    var distanceSqrt = direction.sqrMagnitude + 0.001f;
    return GravitationalConstant * SunMass / distanceSqrt * direction.
        normalized;
}
```

La struttura della classe `BaseIntegrator` garantisce un'interfaccia uniforme per tutti gli integratori numerici, isolando la logica fisica dal metodo di integrazione. In questo modo, la legge gravitazionale rappresenta il cuore fisico del sistema, mentre le diverse tecniche di integrazione agiscono soltanto sul modo in cui tale accelerazione viene integrata nel tempo.

- `ISunData` e `SunScaler`, che gestiscono la massa e la scala del Sole, aggiornandone in tempo reale l'effetto gravitazionale

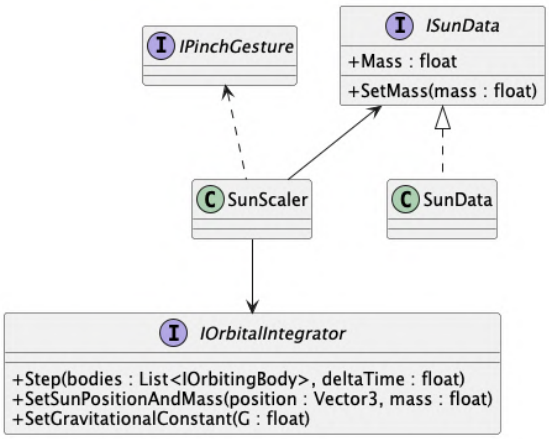


Figura 4.11: Diagramma UML relativo al Sole

Il Sole è rappresentato da diversi componenti:

  - `SunData`: incapsula la massa del Sole, che può cambiare se si scala la sua grandezza
  - `SunScaler`: permette di modificare dinamicamente la dimensione, con uno slider in Editor e con la gesture multi pinch su dispositivo

- `IPlanetInitializer`, incaricato di costruire le entità dei pianeti a partire dai dati statici definiti in `PlanetData`

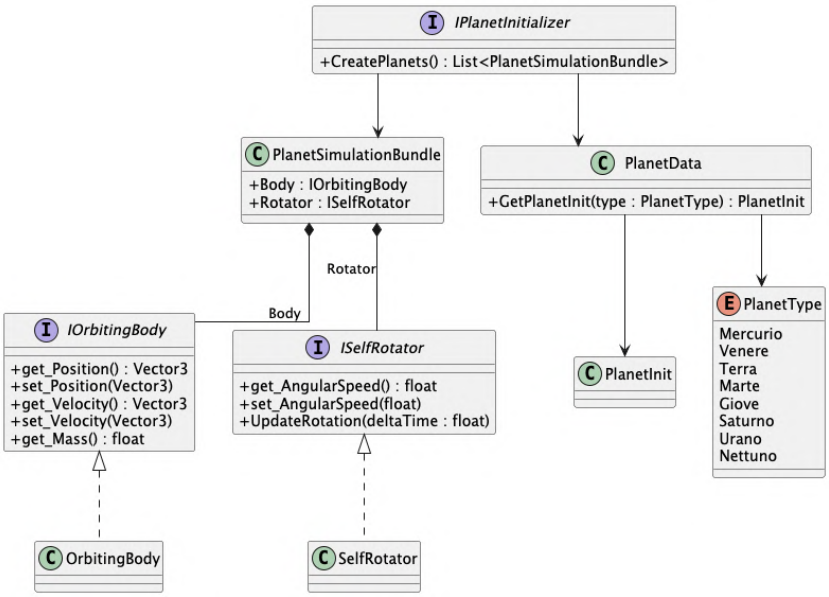


Figura 4.12: Diagramma UML relativo ai Pianeti

L'obiettivo di questo modulo è quello di fornire una rappresentazione coerente dei pianeti simulati, descrivendone le proprità fisiche fondamentali e offrendo un meccanismo per la loro inizializzazione. Il sistema utilizza una classe `PlanetData` che contiene dati fisici predefiniti per ciascun pianeta basati sull'enum `PlanetType`. Questi valori di inizializzazione sono incapsulati in strutture `PlanetInit`.

Listing 4.21: Implementazione dei parametri planetari

```
private static readonly Dictionary<PlanetType, PlanetInit> planetData = new
    ()
{
    { PlanetType.Mercury, new PlanetInit(0.055f, 0.3f, CalcRotationSpeed
        (58.6f), 14.0f) },
    { PlanetType.Venus,   new PlanetInit(0.815f, 0.45f, CalcRotationSpeed
        (243f, true), 6.8f) },
    { PlanetType.Earth,   new PlanetInit(1f, 0.6f, CalcRotationSpeed(1f), 0f
        ) },
    { PlanetType.Mars,    new PlanetInit(0.107f, 0.75f, CalcRotationSpeed
        (1.03f), 3.7f) },
    { PlanetType.Jupiter, new PlanetInit(57.8f, 1.0f, CalcRotationSpeed(0.41
        f), 2.6f) },
    { PlanetType.Saturn,  new PlanetInit(45.2f, 1.2f, CalcRotationSpeed(0.45
        f), 5.0f) },
    { PlanetType.Uranus,  new PlanetInit(24.5f, 1.4f, CalcRotationSpeed(0.72
        f, true), 1.5f) },
    { PlanetType.Neptune, new PlanetInit(27.1f, 1.6f, CalcRotationSpeed(0.67
        f), 3.6f) }
};
```

`PlanetInitializer` crea ciascun pianeta nel simulatore, calcolando la posizione iniziale, in base all'offset dal Sole in base al raggio e ne attribuisce una inclinazione.

- `IPlanetCollisionHandler` e `IPlanetRemovalHandler`, dedicati rispettivamente alla gestione delle collisioni tra pianeti e alla rimozione di quelli che si allontanano troppo dal Sole o impattano con il muro.

4.4 Prisma Ottico – Visualizzazione della dispersione luminosa

L'implementazione della simulazione del prisma si basa sul modello matematico descritto nel capitolo precedente, sfruttando Unity per la gestione dei raggi e delle collisioni. L'obiettivo è ricostruire il percorso del raggio luminoso all'interno del prisma, dalla superficie d'ingresso a quella di uscita, applicando la legge di Snell in forma vettoriale e gestendo i casi di riflessione interna.
La classe `PrismRefraction` è responsabile dei calcoli di rifrazione e della gestione delle superfici del prisma.

Il suo costruttore:

Listing 4.22: Implementazione del costruttore di `PrismRefraction`

```
public PrismRefraction(HashSet<Collider> faces,
        Dictionary<Collider, Collider> opposite,
        float nAir,
        float nGlass,
        float epsilon)
    {
        this.faces = faces ?? throw new ArgumentNullException(nameof(faces));
        this.oppositeOf = opposite ?? new Dictionary<Collider, Collider>();
        this.nAir = nAir;
        this.nGlass = nGlass;
        this.epsilon = epsilon;
    }
```

riceve l'insieme delle facce del prisma, una mappa delle facce opposte utile per gestire l'uscita del raggio, gli indici di rifrazione η_{aria} e η_{vetro} e un offset, il cui scopo è di evitare che raycast successivi colpiscano sempre la stessa faccia.
Il cuore del calcolo è il metodo `Refract`:

Listing 4.23: Implementazione del metodo `Refract`

```
static bool Refract(Vector3 I, Vector3 N, float n1, float n2, out Vector3 T)
{
    I = I.normalized;
    N = N.normalized;
    float cosTheta_i = Mathf.Clamp(Vector3.Dot(I, N), -1f, 1f);
    float eta1 = n1;
    float eta2 = n2;
    Vector3 n = N;
    if (cosTheta_i > 0f)
    {
        n = -N;
        (eta1, eta2) = (eta2, eta1);
        cosTheta_i = -cosTheta_i;
    }
    float eta = eta1 / eta2;
    float k = 1f - eta * eta * (1f - cosTheta_i * cosTheta_i);
    if (k < 0f)
    {
        T = default;
        return false;
    }
    T = (eta * I + (eta * (-cosTheta_i) - Mathf.Sqrt(k)) * n).normalized;
    return true;
}
```

che riceve la direzione incidente **I**, la normale alla superficie **N**, gli indici di rifrazione η_1 e η_2 e restituisce il vettore rifratto **T**. Per assicurare la validità in entrambe le direzioni (aria → vetro e vetro → aria) viene verificato il verso della normale: se il raggio incide dall'interno del prisma ($\cos\theta_i > 0$), la normale viene invertita e gli indici scambiati.

La funzione principale `CalculateTrace` implementa la logica per determinare il percorso del raggio luminoso, dalla faccia d'ingresso alla faccia d'uscita. Il suo comportamento segue le leggi fisiche descritte nel capitolo precedente, utilizzando le primitive fisiche di Unity (`Physics.Raycast` e `Physics.RaycastAll`) per individuare le intersezioni e applicando la legge di Snell in forma vettoriale. Il procedimento si divide in diverse fasi operative:

1. **Individuazione della faccia d'ingresso**

   La prima operazione consiste nel determinare il punto d'ingresso nel prisma, calcolando la prima intersezione con una delle superfici appartenenti a `faces`

   Listing 4.24: Implementazione della faccia d'ingresso nel prisma

```
Vector3 directionNormalized = direction.normalized;

RaycastHit enterHit;
bool hitSomething = Physics.Raycast(
    origin,
    directionNormalized,
    out enterHit,
    Mathf.Infinity,
    Physics.DefaultRaycastLayers,
    QueryTriggerInteraction.Ignore
);

if (!hitSomething || !faces.Contains(enterHit.collider))
{
    return false;
}

facesHit.Add(enterHit.collider);
enterPoint = enterHit.point;
```

   Il metodo `Physics.Raycast` restituisce in `enterHit` il primo punto della scena colpito dal fascio luminoso. Se il raggio non colpisce nulla oppure l'oggetto colpito non appartiene all'insieme delle facce del prisma, il metodo restituisce *false*. Il punto `enterHit.point` diventa così il punto di ingresso

e la sua normale (`enterHit.normal`) rappresenta la direzione normale alla superficie incidente.

2. **Rifrazione aria → vetro**
   Determinata la faccia d'ingresso, viene calcolata la direzione del raggio rifratto all'interno del prisma. Si applica la legge di Snell in forma vettoriale tramite la funzione `Refract`, che utilizza gli indici di rifrazione dell'aria e del vetro

Listing 4.25: Implementazione della prima rifrazione nel prisma

```
Vector3 directionInside;
bool refractedIn = Refract(directionNormalized, enterHit.normal, nAir,
    nGlass, out directionInside);

if (!refractedIn)
{
    return false;
}
```

Il valore di `refractedIn` ci indica se la rifrazione è fisicamente possibile: `Refract` verifica il discriminante sotto la radice e se questo è negativo, allora non esiste una soluzione reale, cioè si verifica la riflessione totale interna (TIR), e il metodo termina.

3. **Propagazione interna**
   Dopo essere entrato nel prisma, il raggio si propaga in linea retta fino a colpire la faccia di uscita. Per evitare che il nuovo `Raycast` colpisca nuovamente la faccia appena attraversata a causa di errori di precisione, il punto di partenza viene traslato leggermente nella direzione interna del raggio `directionInside` di un piccolo valore ϵ.

Listing 4.26: Implementazione della propagazione nel prisma

```
Vector3 startInside = enterHit.point + directionInside * epsilon;

Collider preferredExitFace = null;
if (oppositeOf.ContainsKey(enterHit.collider))
{
    preferredExitFace = oppositeOf[enterHit.collider];
}

```

```
RaycastHit[] internalHits = Physics.RaycastAll(
    startInside,
    directionInside,
    Mathf.Infinity,
    Physics.DefaultRaycastLayers,
    QueryTriggerInteraction.Ignore);

if (internalHits == null || internalHits.Length == 0)
{
    return false;
}

Array.Sort(internalHits, (a, b) => a.distance.CompareTo(b.distance));
```

Dopo aver eseguito il `RaycastAll` i risultati vengono ordinati per distanza in modo da valutare le collisioni più vicine prima delle collisioni più lontane.

4. **Rifrazione vetro → aria**
   L'ultimo passaggio consiste nel calcolare la direzione di uscita quando il raggio passa dal vetro all'aria. Si utilizza `Refract`, ma invertendo gli indici di rifrazione

Listing 4.27: Implementazione della seconda rifrazione nel prisma

```
Vector3 directionOutside;
bool refractedOut = Refract(directionInside, exitHit.normal, nGlass, nAir,
    out directionOutside);

if (!refractedOut)
{
    return false;
}

exitDirection = directionOutside;
```

Capitolo 5

Risultati e Valutazione

In questo capitolo vengono presentati i risultati finali dell'applicazione, che mostrano il funzionamento dei modelli sia nella simulazione completa in Unity, sia nella corrispondente visualizzazione in realtà aumentata. Oltre alle visualizzazioni, viene svolta una breve analisi dei metodi numerici utilizzati nei moduli e vengono fornite le conclusioni finali del progetto.

5.1 Risultati Visivi

Vengono presentati gli screenshot dei diversi moduli dell'applicazione. L'obiettivo è evidenziare come ciascun componente mantenga una resa coerente e funzionale nel passaggio dall'ambiente simulato allo spazio fisico reale. Le immagini permettono di apprezzare la stabilità dell'ancoraggio, la leggibilità della scena, la qualità visiva dei modelli e il comportamento complessivo dell'applicazione in contesti reali e in differenti condizioni di luce esterna.

Figura 5.1: Simulazione del pendolo sull'arco di un giorno.

Figura 5.2: Simulazione dettagliata del moto a breve termine.

Figura 5.3: Desk informativo interattivo

Figura 5.4: Modello 3D animato e integrato in Gravilab

Figura 5.5: Teca trasparente utilizzata per l'installazione AR.

Figura 5.6: Lavagna interattiva

Figura 5.7: Dettaglio oggetti caduti, esempio 1

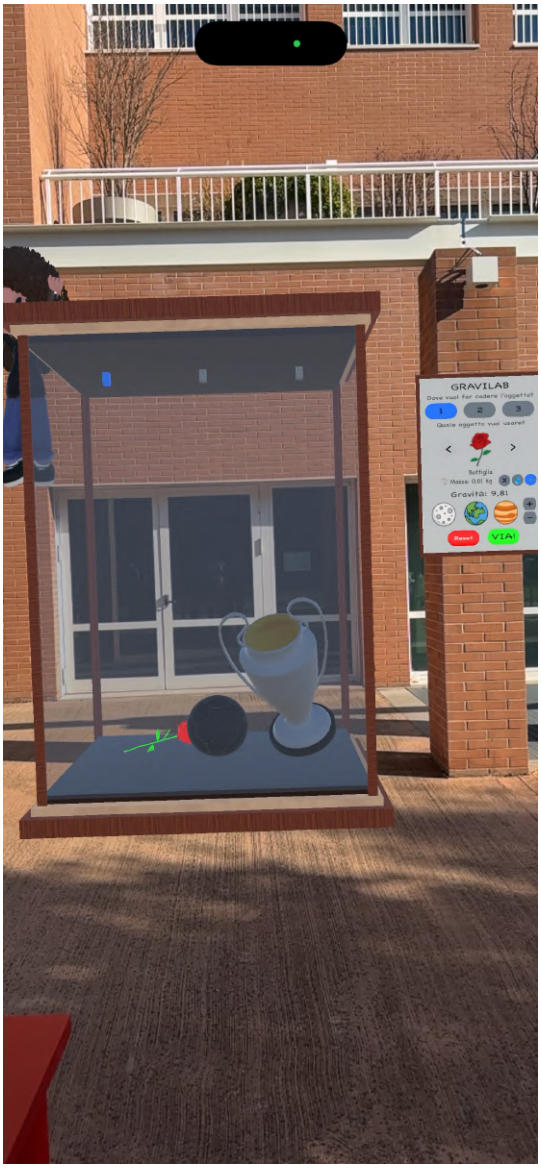


Figura 5.8: Dettaglio oggetti caduti, esempio 2

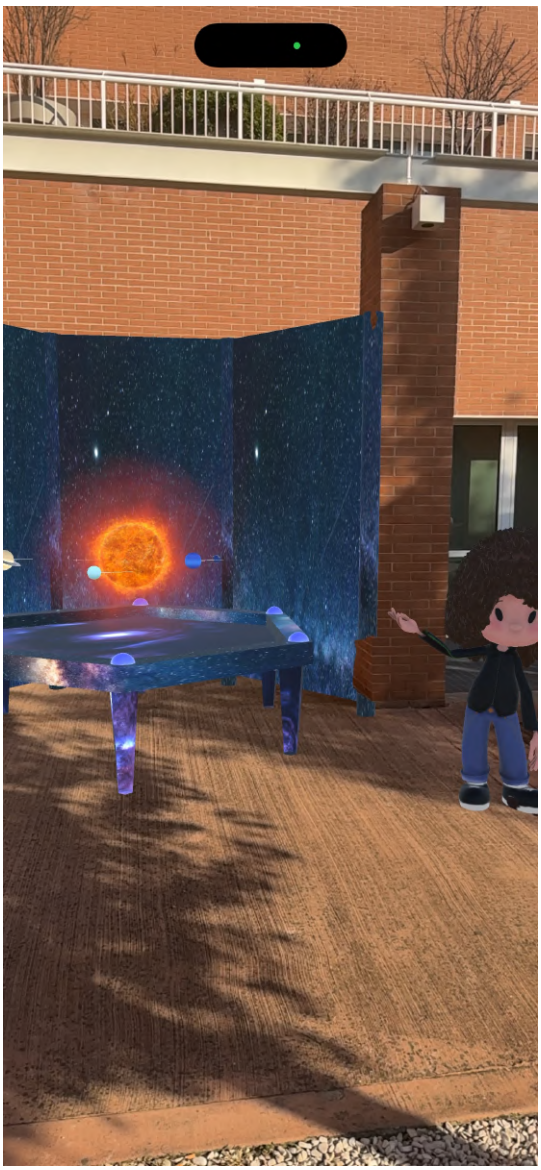


Figura 5.9: Exhibit del Sistema Solare

Figura 5.10: Modello 3D animato e integrato nel Sistema Solare

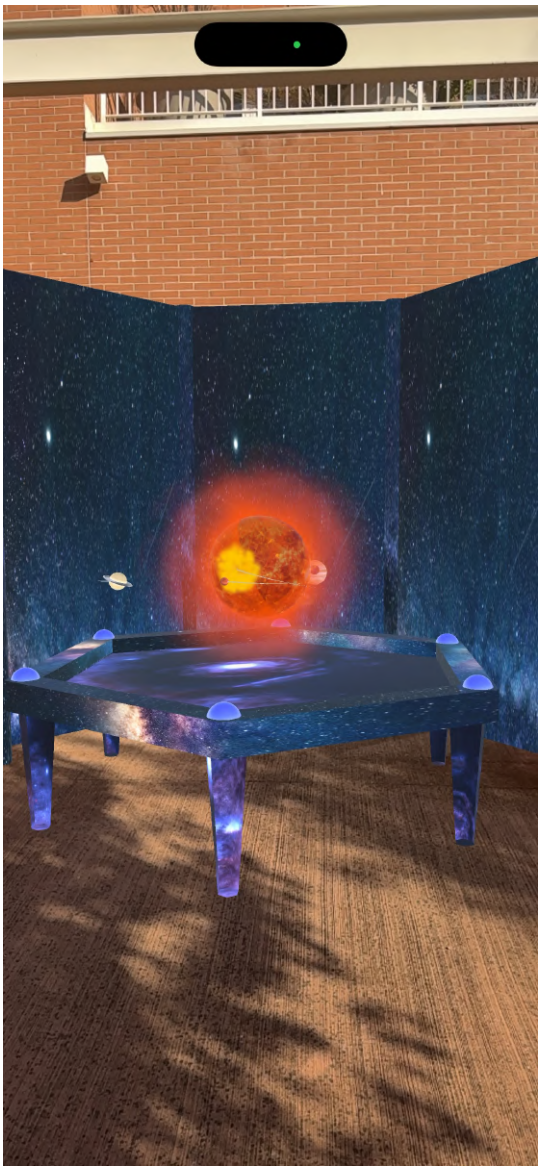


Figura 5.11: Collisione del pianeta con il Sole, dopo aver aumentato la dimensione di quest'ultimo

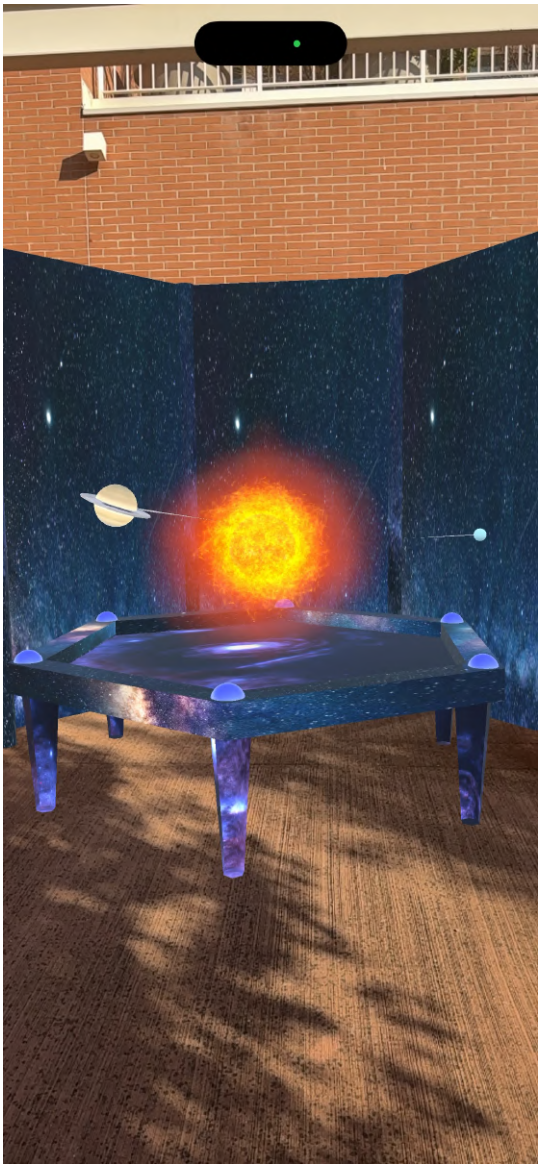


Figura 5.12: Deriva dei pianeti, dopo aver diminuito la dimensione del Sole

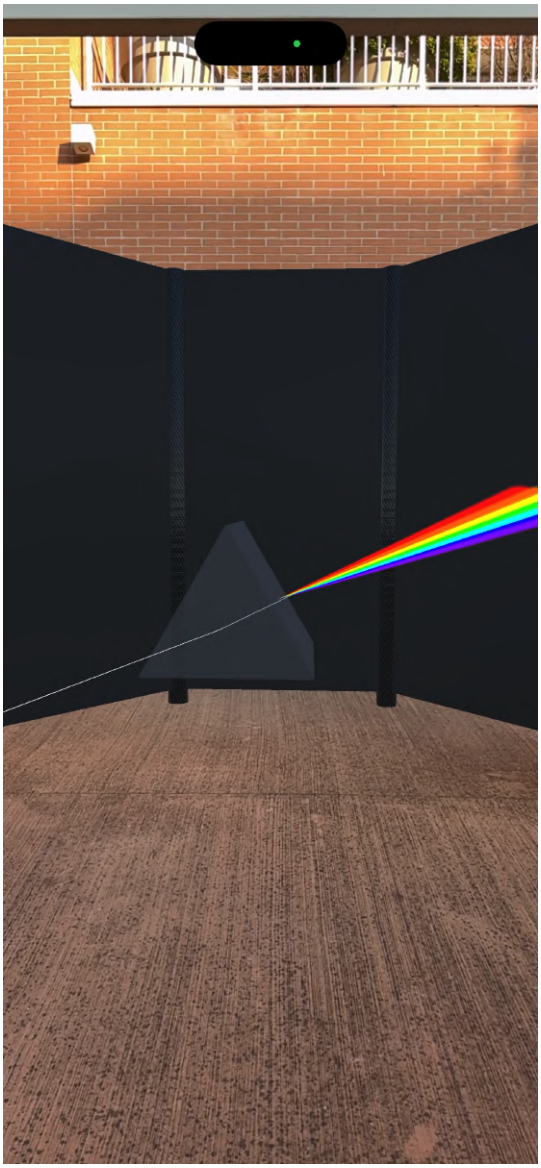


Figura 5.13: Simulazione della rifrazione interna nel prisma, esempio 1

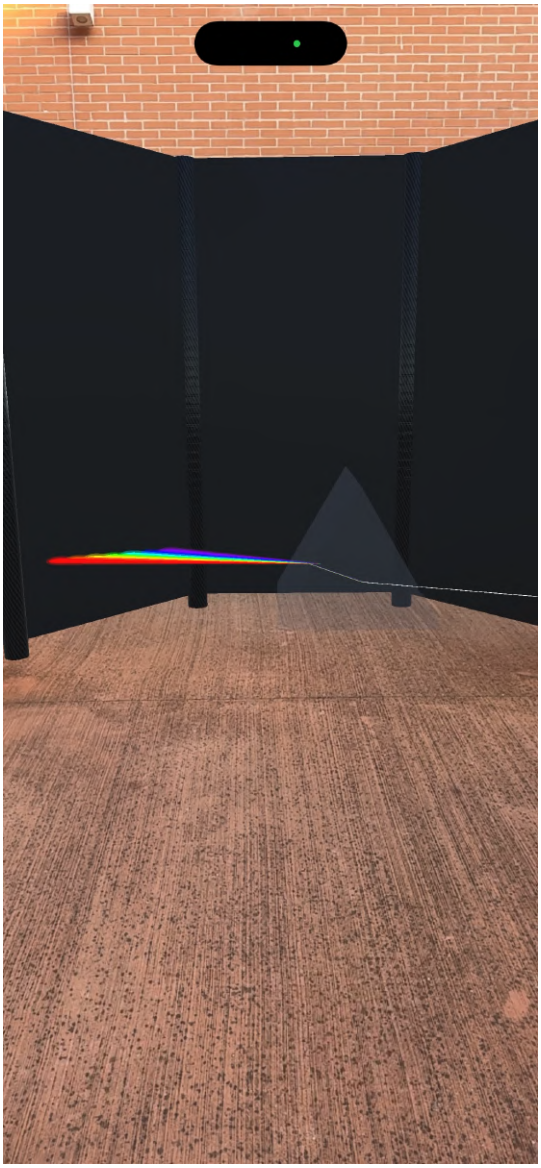


Figura 5.14: Simulazione della rifrazione interna nel prisma, esempio 2

Figura 5.15: Exhibit con personaggio animato

Figura 5.16: Dettaglio animazione 3D

5.2 Risultati Numerici

Analisi numerica del Pendolo di Foucalt

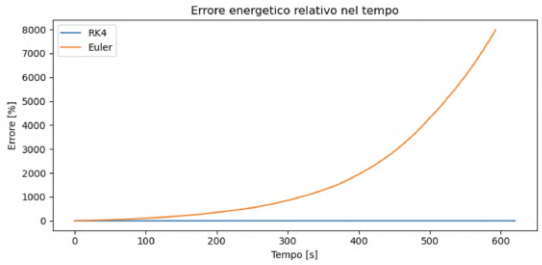
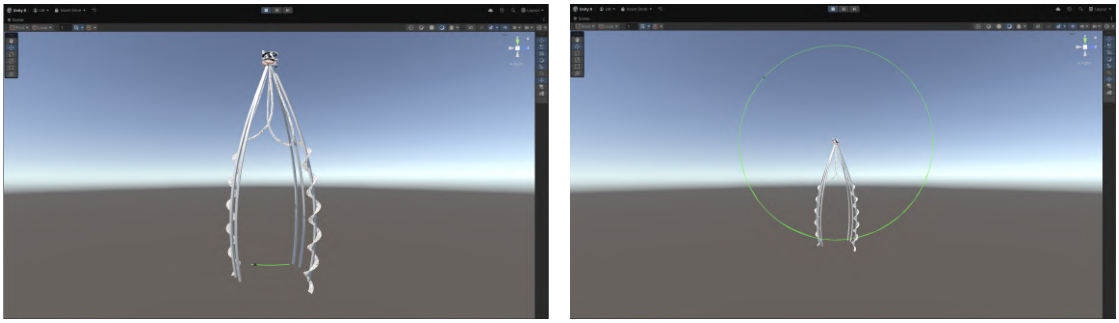


Figura 5.17: Andamento dell'energia del pendolo durante la simulazione numerica

Il grafico mostra l'evoluzione dell'errore energetico relativo del pendolo di Foucault nel tempo per i due integratori sperimentati: Eulero e Runge–Kutta 4. L'andamento evidenzia come il metodo di Eulero presenti un errore crescente e rapidamente divergente. Questa deriva rende l'integratore inadatto a descrivere un sistema oscillatorio che richiede stabilità numerica e conservazione dell'energia. Al contrario, l'integratore RK4 mantiene l'errore energetico praticamente costante e prossimo allo zero per tutta la durata della simulazione. Questo comportamento conferma l'elevata precisione locale del metodo e la sua capacità di preservare le caratteristiche dinamiche del sistema nel breve periodo.
Per visualizzare in modo intuitivo gli effetti numerici del passo temporale sull'integrazione del pendolo, è stata introdotta una *coda verde* che segue il movimento della massa e rappresenta la traiettoria recente percorsa. Si osserva immediatamente la differenza tra due integrazioni eseguite con lo stesso metodo ma con passo differente:

- con **passi temporali piccoli**, la traiettoria risulta regolare e la linea verde ricalca l'oscillazione, mantenendo costante l'ampiezza e la fase;
- con **passi temporali grandi**, la linea evidenzia chiaramente lo sfasamento progressivo e il drift di ampiezza dovuti all'errore numerico del metodo. La *coda* agisce come indicatore visivo dell'accuratezza dell'integrazione, mostrando come il pendolo sembri "guadagnare energia" a ogni oscillazione, compromettendo la coerenza della simulazione.



(a) Eulero (passo piccolo) (b) Eulero (passo grande)

Figura 5.18: Eulero con passi temporali differenti.

Analisi numerica del Sistema Solare

L'analisi con due differenti passi temporali dimostra come la stabilità energetica del sistema solare dipenda fortemente dall'integratore numerico adottato

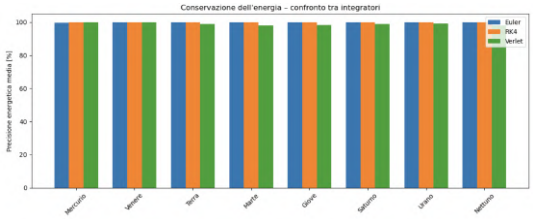


Figura 5.19: Simulazione con passo temporale basso.

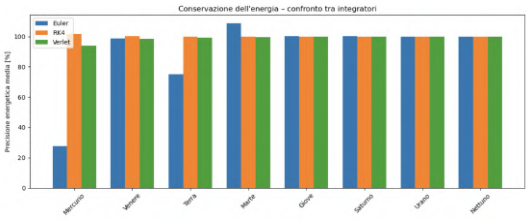


Figura 5.20: Simulazione con passo temporale alto.

Con un passo temporale ridotto, tutti gli integratori mostrano una conservazione dell'energia molto elevata. Eulero, pur essendo il metodo meno accurato, mantiene performance accettabili grazie alla piccolezza del Δt, mentre RK4 e Verlet si comportano in modo quasi indistinguibile. Questo comportamento riflette la natura dei metodi: un passo piccolo riduce l'errore locale per tutti gli integratori, mascherando le differenze strutturali tra metodi espliciti (Eulero), a precisione elevata (RK4) e simmetrici/conservativi (Verlet).

Con un passo temporale maggiore, emergono chiaramente le differenze tra gli integratori. Eulero diventa instabile e introduce errori energetici molto più elevati, fino a diverse decine di punti percentuali. RK4 e Verlet mantengono una buona precisione locale, mantenendo variazioni minime per tutti i pianeti.

5.3 Conclusioni

Il lavoro presentato in questa tesi dimostra come sia possibile integrare modelli fisici complessi, tecniche di simulazione numerica e tecnologie di Mixed Reality per costruire un sistema interattivo capace di rendere accessibili fenomeni difficili da osservare o riprodurre nel mondo reale.

La modellazione tridimensionale, realizzata in Blender e adattata per Unity, ha permesso di ottenere una resa visiva stabile nei diversi contesti di esecuzione. La pipeline di importazione e ottimizzazione dei modelli ha garantito una continuità estetica e funzionale tra la simulazione completa e la versione AR.

Dal punto di vista fisico e numerico, è stata implementata un'architettura modulare che supporta dinamiche differenti. Il confronto tra Eulero, RK4 e Verlet ha evidenziato come ogni metodo presenti vantaggi e limiti specifici:

- Eulero risulta inadatto a simulazioni prolungate;
- Verlet si conferma adatto a sistemi che richiedono stabilità energetica nel lungo periodo;
- RK4 mantiene un buon livello di precisione nelle condizioni adottate.

L'integrazione con la Mixed Reality rappresenta uno degli elementi centrali del progetto. L'utilizzo del Visual Positioning System di Lightship ha permesso un posizionamento stabile degli oggetti virtuali nello spazio fisico. Il confronto diretto tra simulazione ed esperienza AR mostra che i modelli mantengono consistenza visiva e comportamentale in entrambe le modalità.

L'applicazione costituisce quindi un primo esempio di laboratorio scientifico in MR, nel quale l'utente può osservare fenomeni fisici complessi, modificarne i parametri e valutarne gli effetti sia nel dominio numerico sia in quello visivo aumentato.

Il progetto può proseguire in molte direzioni:

- l'introduzione di ulteriori fenomeni fisici o moduli tematici;
- la realizzazione di simulazioni più estese nel tempo, anche multi-body reali;
- modalità di interazione avanzata o multiutente.

In conclusione, l'esperienza sviluppata mostra come la combinazione di simulazione numerica e Mixed Reality possa trasformare lo studio dei fenomeni fisici, offrendo un modo nuovo, visivo e coinvolgente di esplorare concetti complessi, e rappresentando un esempio concreto delle potenzialità delle tecnologie immersive applicate alla divulgazione scientifica.

Bibliografia

[1] Microsoft Corporation. *Mixed Reality — Discover Learn*. URL: https://learn.microsoft.com/it-it/windows/mixed-reality/discover/mixed-reality.

[2] Inc. Meta Platforms. *Discover Mixed Reality — Meta Horizon Developers*. URL: https://developers.meta.com/horizon/discover/mixed-reality-intro/.

[3] Noor E. Karishma Shaik. *Metaverse and Multiverse: The Real Sense of AR, VR, MR, XR and IR*. 2022. URL: https://yp.ieee.org/blog/2022/04/13/metaverse-and-multiverse-the-real-sense-of-ar-vr-mr-xr-and-ir/.

[4] XR Association. *"State of the Industry Report," with Highlights from 2024 and a Look Ahead to 2025*. 2025. URL: https://xra.org/xr-association-releases-state-of-the-industry-report-with-highlights-from-2024-and-a-look-ahead-to-2025/.

[5] Maya Georgieva et al. *XR in Higher Education: Adoption, Considerations, and Recommendations*. 2024. URL: https://er.educause.edu/articles/2024/1/xr-in-higher-education-adoption-considerations-and-recommendations.

[6] Hovr Labs. *Hovr Labs – Home*. URL: https://www.hovrlabs.com/.

[7] Doğuş Beyoğlu, Çiğdem Hürsen e Arda Nasiboğlu. "Use of mixed reality applications in teaching of science". In: *Education and Information Technologies* 25 (2020), pp. 4271–4286. DOI: 10.1007/s10639-020-10166-8. URL: https://link.springer.com/article/10.1007/s10639-020-10166-8.

[8] Constance M. Doty et al. “Impact of high-intensity training with a mixed-reality simulator on graduate teaching assistants’ use of questioning”. In: *Physical Review Physics Education Research* 19.2 (2023), p. 020101. DOI: 10.1103/PhysRevPhysEducRes.19.020101. URL: https://journals.aps.org/prper/abstract/10.1103/PhysRevPhysEducRes.19.020101.

[9] Case Western Reserve University. *HoloAnatomy® Software Suite*. URL: https://case.edu/holoanatomy/.

[10] Dalai Felinto. *The Future of Hair Grooming*. URL: https://code.blender.org/2022/07/the-future-of-hair-grooming/.

[11] Unity Technologies. *2024 Unity Gaming Report – Highlights Game Studios’ Continued Resilience*. 2024. URL: https://unity.com/news/2024-unity-gaming-report-highlights-resilience.

[12] Unity Technologies. *Unity 6 — Support and Release Information*. URL: https://unity.com/releases/unity-6/support.

[13] Inc. Niantic Spatial. *Niantic Spatial – Home*. URL: https://www.nianticspatial.com/.

[14] Inc. Niantic. *Niantic Opens Lightship Platform Globally*. 2021. URL: https://nianticlabs.com/news/lightshiplaunch?hl=en.

[15] Inc. Niantic. *SDK for Unity — Niantic Lightship ARDK*. URL: https://lightship.dev/docs/ardk/.

[16] PRELOADED. *Wonderlab AR – Science Museum Group Lightship*. URL: https://preloaded.com/work/wonderlab-ar/.

[17] Inc. Niantic. *Semantics — ARDK 3 Features*. URL: https://lightship.dev/docs/ardk/3.0/features/semantics/.

[18] Inc. Niantic. *Meshing — ARDK Features*. URL: https://lightship.dev/docs/ardk/features/meshing/.

[19] Comune di Padova. *Il Pendolo di Foucault — Palazzo della Ragione (Padova)*. URL: https://padovacultura.padovanet.it/sites/default/files/documenti-info-musei/il_pendolo_di_foucalut.pdf.

[20] Eduardo Moraes Diniz. "On the Foucault pendulum". In: *Revista Brasileira de Ensino de Física* (2023). URL: https://www.researchgate.net/publication/369566912_On_the_Foucault_pendulum.

[21] A. Reiser e J. Stiewe. *The Foucault Pendulum – a Simplified Trajectory Analysis for a Pendulum on a Turntable and an Outlook to a Pendulum on Earth*. URL: https://www.kip.uni-heidelberg.de/image/f/oeffwiss/pendel/Foucault.pdf.

[22] R. T. W. Arthur. *On the Mathematization of Free Fall: Galileo, Descartes, and a History of Misconstrual*. 2016. URL: https://manifold.umn.edu/read/untitled-7ca18210-217d-40f2-83fe-b0add1d84ede/section/bb88e7f4-7451-4b26-9056-a21f51f49091.

[23] R. Yáñez Valdez, P. A. Gómez Valdez e F. de Armas Rivero. "Horizontal projectile motion: comparing free fall and drag resistance". In: *Revista Mexicana de Física E* (2020). URL: https://revistamexicanadefisica.org/index.php/rmf-e/article/view/17_2_156.

[24] Lumen Learning. *Drag Forces — Physics – SUNY eText*. URL: https://courses.lumenlearning.com/suny-physics/chapter/5-2-drag-forces/.

[25] Inc. Encyclopædia Britannica. *Stokes's Law*. URL: https://www.britannica.com/science/Stokess-law.

[26] Peter Timmerman e Jacobus P. van der Weele. "On the rise and fall of a ball with linear or quadratic drag". In: *American Journal of Physics* (1999). URL: https://ris.utwente.nl/ws/portalfiles/portal/6689298/Timmerman99on.pdf.

[27] Department of Physics University of Texas at Austin. *Damped Harmonic Oscillation*. URL: https://farside.ph.utexas.edu/teaching/336k/Newton/node17.html.

[28] WeSchool. *Forza di Gravità: Formula ed Esercizio Svolto*. 2012. URL: https://library.weschool.com/lezione/forza-di-gravit%C3%A0-formula-ed-esercizio-svolto-7585.html.

[29] David Halliday, Robert Resnick e Jearl Walker. *Fondamenti di Fisica*. Settima edizione. Casa Editrice Ambrosiana, 2014. ISBN: 9788808190345.

[30] OpenStax. *Newton's Universal Law of Gravitation — College Physics 2e, Section 6.5*. URL: https://openstax.org/books/college-physics-2e/pages/6-5-newtons-universal-law-of-gravitation.

[31] Eugene Hecht. *Optics*. 5th edition. Pearson Education, 2017.

[32] Frank L. Pedrotti, Leno S. Pedrotti e Leno M. Pedrotti. *Fundamentals of Photonics: Basic Geometrical Optics*. SPIE Press, 2008.

[33] Justin Peatross e Michael Ware. *Physics of Light and Optics*. Brigham Young University Press, 2015.

[34] Massachusetts Institute of Technology. *10.001 Course Notes: Differential Equations — Node 3*. URL: https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node3.html.

[35] Massachusetts Institute of Technology. *Verlet Integration*. URL: https://fab.cba.mit.edu/classes/864.23/people/Erik/verlet-integration/.

[36] Jiri Lebl. *Notes on Diffy Qs: Numerical methods – Euler's method*. URL: https://www.jirka.org/diffyqs/html/numer_section.html.

[37] Richard L. Burden e J. Douglas Faires. *Numerical Analysis*. Cengage Learning, 2016.

[38] Randall J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. Society for Industrial e Applied Mathematics (SIAM), 2007.

[39] Nicholas J. Giordano e Hisao Nakanishi. *Computational Physics*. Pearson Addison-Wesley, 2006.

[40] Loup Verlet. "Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". In: *Physical Review* 159.1 (1967), pp. 98–103. DOI: 10.1103/PhysRev.159.98.

[41] Jesús M. Sanz-Serna e Mari-Paz Calvo. *Numerical Hamiltonian Problems*. Chapman & Hall, 1994.

[42] Ernst Hairer, Christian Lubich e Gerhard Wanner. *Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations*. 2nd. Springer, 2006.

[43] Massachusetts Institute of Technology. *Verlet Integration and Energy Conservation*. 2021. URL: https://ocw.mit.edu/courses/8-01sc-classical-mechanics-fall-2016/resources/lecture-31-energy-in-systems-with-variable-forces/.

[44] Stuart Brorson. *1.7: Symplectic integrators*. URL: https://math.libretexts.org/Bookshelves/Differential_Equations/Numerically_Solving_Ordinary_Differential_Equations_(Brorson)/01%3A_Chapters/1.07%3A_Symplectic_integrators.

[45] Massachusetts Institute of Technology. *Runge-Kutta Methods — Differential Equations Notes*. URL: https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html.

[46] Jirí Lebl. *Notes on Diffy Qs: Differential Equations for Engineers*. 2025. URL: https://www.jirka.org/diffyqs/html/numer_section.html.

[47] Unity Technologies. *Unity Documentation: Time.deltaTime*. URL: https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Time-deltaTime.html.

[48] Unity Technologies. *Unity Documentation: Time.fixedDeltaTime*. URL: https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Time-fixedDeltaTime.html.

[49] Unity Technologies. *Unity Documentation: Time.maximumDeltaTime*. URL: https://docs.unity3d.com/6000.2/Documentation/ScriptReference/Time-maximumDeltaTime.html.