POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

Politecnico di Torino

Tesi di Laurea Magistrale

Sviluppo di un simulatore per due CubeSat 3U in fase di docking

Relatore

Prof. Andrea BOTTINO

Candidato

Marco CORTESE

Aprile 2022

Abstract

La creazione di simulatori 3D, con lo scopo di generare una rappresentazione affidabile di ciò che avviene all'interno di un sistema, è ormai fondamentale in molteplici ambiti tecnologici. Uno tra questi è la riproduzione di delicate manovre di satelliti in orbita, con l'obiettivo di avere una prima effettiva ricostruzione di ciò che accadrebbe a centinaia di chilometri dalla superficie terrestre. Negli ultimi anni, la tecnologia ha permesso di utilizzare satelliti sempre più piccoli per effettuare operazioni di ricerca in orbita. Tra queste tecnologie si riconoscono i cosiddetti CubeSat, nanosatelliti composti da diverse Units(U) o Unità, ciascuna di dimensioni $10\,\text{cm} \times 10\,\text{cm} \times 10\,\text{cm}$ e di peso massimo di $1.33\,\text{kg}$. Le dimensioni ridotte dei satelliti vanno chiaramente a creare una sfida per quanto riguarda la loro manovra. Un simulatore è diventato, di conseguenza, uno strumento indispensabile per la buona riuscita di una missione.

L'obiettivo di questa tesi è quello di creare un simulatore, sul Game Engine di Unity, che possa inserire in un ambiente realistico e affidabile, una visualizzazione in tempo reale di una missione di docking (attracco) tra due CubeSat. Uno, chiamato Chaser, attrezzato di una telecamera monoculare, cerca di avvicinarsi a un altro, chiamato Target, fino ad attaccarsi, servendosi di un sistema di Visual Based Navigation (VBN). L'algoritmo è gestito da un software esterno (Simulink) che calcola le manovre di posizionamento e orientamento del Chaser, che dipendono dalla posizione, dal punto di vista della telecamera, di alcuni Diodi ad Emissione di Luce (LED) posti sul Target. È necessario, quindi, ricreare fedelmente i modelli 3D dei due nanosatelliti, basandosi su reference, ponendo particolare attenzione sulle facce di docking di entrambi. In seguito a questo, una connessione bidirezionale, che sfrutta lo User Datagram Protocol (UDP), viene creata tra le due applicazioni. Vengono catturati i dati dei LED del Target da Unity, i quali sono mandati al secondo software per permettergli di calcolare la manovra successiva da far effettuare al Chaser. Tutto questo, ovviamente, dopo aver studiato e convertito tutti i dati nei corretti sistemi di riferimento, che differiscono tra le due applicazioni. Inoltre, viene creato un sistema di telecamere, al fine di visualizzare i due satelliti all'interno di una rappresentazione dei corpi celesti (Terra e Sole) con grandezze e distanze in scala. Insieme alla creazione di shader realistici per i corpi celesti, viene studiato un Post Processing che vada a garantire una miglior resa estetica della simulazione. Infine, viene fatto un confronto tra la simulazione effettuata con il contributo di Unity, e quella esclusivamente generata su Simulink, andando ad analizzare eventuali differenze tra le due versioni.

Keywords: Space, CubeSat, Docking, Simulator, 3D.

Ringraziamenti

Sono arrivato finalmente alla fine del percorso
e siccome le persone da ringraziare son tante,
non voglio dilungarmi troppo nel discorso.

Ringrazio chi mi ha aiutato a sentirmi un gigante,
perchè senza avrei fatto fatica per davvero;
a voi il mio ringraziamento più importante.

A tutti i miei più cari amici, un ringraziamento vero,
con chi ho bevuto insieme, con chi ho gioito, con chi ho pianto
dal profondo del cuore, un abbraccio sincero.

Grazie ai miei amici con cui ho videogiocato tanto,
lo svago insieme mi allietato il tempo.
Con voi, nessun secondo è stato rimpianto.

Grazie al mio amico Piero e al mio amico Doc al contempo,
la vostra amicizia è un bene veramente prezioso
che a tristezza e malinconia non lascia scampo.

A tutta la mia famiglia che compone un gruppo meraviglioso,
a voi dedico il mio abbraccio più amorevole.
Avervi come supporto è il mio pensiero più gioioso.

Ai miei cari nonni, che di amarli troppo son colpevole,
grazie del vostro infinito amore incondizionato.
A voi che avete reso la mia vita la migliore tra le favole.

A Tequila, che senza farlo apposta mi ha tanto aiutato.
Con la gioia di vivere e tanta semplicità,
durante i momenti di tristezza, un raggio di luce mi ha regalato.

A Fra e Alby, di cui invidio la bellissima complicità,
tra grigliate e giochi da tavola, ogni minuto è stato un gioiello.
Tutto questo per me sarà sempre un ricordo di felicità.

A Toto, che da tanto tempo considero un fratello.
Grazie per essere il migliore amico che una persona possa desiderare
e che tra pianti e risate, hai reso la mia vita qualcosa di bello.

A Gecca, che mi ha insegnato tanto, in primis ad amare.
Ti ringrazio infinitamente per ciò che fai per me ogni giorno;
insieme a te, sento che nessun ostacolo mi potrà mai fermare.

Grazie a voi, Mamma e Papà, che date alla mia vita un magnifico contorno,
dedico a voi questo lavoro, per l'amore che mi avete donato.
Per chi mi avete permesso di diventare, a voi il mio grazie eterno.

Al mio relatore, che alla fine del mio percorso mi ha accompagnato.
Grazie per la disponibilità e per la sua pazienza,
per avermi permesso di lavorare su un tema a cui sono tanto appassionato.

Infine, grazie a Gic, per la sua estrema competenza,
per essere stato, non solo un grande collega, ma anche un grande amico.
Grazie del tuo aiuto, perchè non ce l'avrei mai fatta senza.

Marco Cortese, Aprile 2022

Tabella dei Contenuti

Elenco delle tabelle

Elenco delle figure

Acronimi

ASCII American Standard Code for Information Interchange

CAD Computer-Aided Design

DOF Degrees Of Freedom

FOV Field Of View

LEO Low-Earth-Orbit

FBX Filmbox

RGB Red Green Blue

IP Internet Protocol

LED Light Emitting Diode

TCP Transmission Control Protocol

UDP User Datagram Protocol

VBN Vision Based Navigation

Capitolo 1

Introduzione

1.1 Motivazione

Lo sviluppo di simulatori per la visualizzazione e lo studio di particolari sistemi in fase di analisi, ha consentito alla tecnologia di andare ad anticipare eventuali comportamenti che avrebbero portato al fallimento di un esperimento e, nel peggiore dei casi, messo a repentaglio la vita di molte persone. In particolare la ricostruzione di un ambiente 3D permette di studiare possibili soluzioni che, in assenza di essa, sarebbero difficili da individuare. Questo approccio si può estendere a numerosi ambiti tecnologici, dall'elettronica alla meccanica, dalla robotica all'ingegneria edile, e, come in questo caso, all'ambito aerospaziale. L'evoluzione delle tecnologie ha portato ad un nuovo livello la tecnologia aerospaziale, permettendo all'uomo di portare in orbita dei satelliti sempre più potenti, precisi e soprattutto più piccoli, al fine di ridurne i costi. Questo comporta una maggiore facilità di assemblaggio e di manovra in orbita, ma anche una sfida, che in futuro porterà probabilmente un grande sviluppo di quella che è la ricerca di ciò che ancora non conosciamo del cosmo.

Esiste una suddivisione dei satelliti in base al peso, come mostrato in Figura 1.1, che ovviamente, con il diminuire delle dimensioni, porta a una maggiore complessità dei calcoli delle manovre da effettuare. Per cui, la creazione di un simulatore diventa un passo necessario per la buona riuscita della missione. L'obiettivo di questa tesi è proprio quello di facilitare lo studio di una missione che coinvolge due satelliti, con lo sviluppo di un simulatore.

1.2 Missione di Docking

Ci si pone come obiettivo primario quello di servirsi del *Game Engine* di *Unity*, per visualizzare una missione orbitale di Rendezvous & Docking di due nanosatelliti che

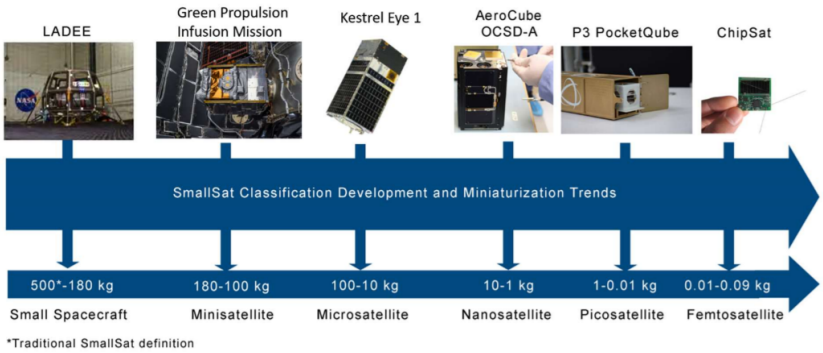


Figura 1.1: Suddivisione di satelliti in base al peso [1]

prendono il nome di CubeSat, il tutto all'interno di un sistema realistico e in scala. L'algoritmo che gestisce il controllo e le manovre dei satelliti è stato effettuato da Giacomo Ichino[2].

Questa tipologia di missione prevede che un satellite si avvicini ad un altro in orbita per attaccarsi, sulla base di informazioni di navigazione relativa. I due satelliti in questione prendono il nome di Chaser ("inseguitore") e Target ("obiettivo"). Tra i diversi metodi per determinare la posizione e la rotazione del Chaser relativamente al Target, si è utilizzata una Vision Based Navigation o VBN. Questa tecnica si basa sull'utilizzo di strumenti ottici (telecamere monoculari o stereo) che individuano dei marcatori passivi (adesivi o scritte) o attivi (LED). In questa simulazione verrà utilizzata una telecamera monoculare sul Chaser e dei marcatori attivi sul Target.

A livello teorico, le informazioni sul Target vengono carpite da un Image Processing delle immagini catturate dalla telecamera del Chaser. Questo processo è fondamentale per la VBN, e da questo ne dipende anche il peso computazionale dell'intera simulazione. Tra i vari algoritmi esistenti si riconosce il SIFT detector, o Scale-Invariant Feature Transform detector che, considerando diverse copie dell'immagine, sfocate e orientate in modi diversi, permette l'individuazione di caratteristiche invarianti rispetto a tali trasformazioni[3]. Ovviamente esistono altri algoritmi che possono individuare in modo molto più preciso forme e pattern all'interno di un'immagine, ma con una maggiore efficacia aumenta anche il costo computazionale. Diventano a quel punto effettivamente utilizzabili solo nel caso in cui le forme da individuare sono molto particolari.

Siccome tale simulazione è processata e analizzata nell'ambiente di Simulink[1], e i LED presenti su un'immagine possono essere considerati come un gruppo di pixel, gli stessi possono essere facilmente individuati con un algoritmo di analisi

[1]Software integrato in MATLAB® per modellazione, simulazione e analisi di sistemi dinamici

Blob presente all'interno del software. Questo tipo di analisi è computazionalmente più gestibile e, avendo un carico di complessità basso, ne permette una veloce esecuzione.

Nella simulazione di MATLAB ® non viene effettuato un vero e proprio Image Processing, bensì viene simulata la posizione dei LED sull'immagine. Indi per cui, sebbene nel lavoro di questa tesi verrà presentato nell'appendice come catturare immagini all'interno della simulazione, saranno effettivamente fornite e utilizzate le sole posizioni dei LED sull'immagine.

1.3 Stato dell'arte

Lo sviluppo di un simulatore, generalmente coinvolge un ambiente in cui elementi e modelli 3D interagiscono tra loro secondo precise regole, preferibilmente in tempo reale. Per farlo si può utilizzare un motore grafico o *Game Engine*. Uno tra questi è **Unity**, che permette la creazione di applicazioni 2D/3D, generalmente videogiochi, lavorando in un ambiente di programmazione prevalentemente in linguaggio C# (C Sharp).

Kerbal Space Program

Kerbal Space Program è un progetto sviluppato dalla software house Squad, che si serve di numerosi elementi per la creazione di un ambiente cosmico verosimile. Si tratta infatti di un videogioco ambientato nello spazio dove il giocatore, impersonando dei piccoli alieni che prendono il nome di Kerbals, deve occuparsi di costruire navicelle e razzi di diverso tipo partendo da specifiche componenti. Lo scopo è conquistare l'universo. Questa fase di progettazione comporta uno studio da parte del giocatore di come e dove posizionare gli elementi necessari alla sua navicella per volare (come ad esempio la quantità di carburante da portare con sè), considerando che una maggiore quantità ne comporta un maggior peso. Una volta riuscito ad abbandonare la superficie terrestre, il giocatore avrà modo di interfacciarsi con meccaniche orbitali accurate e un intero sistema solare da esplorare. A questo punto dovrà preoccuparsi di atterrare su altri pianeti e lune, costruire nuove basi, mandare in orbita stazioni spaziali e creare un vero e proprio Space Program.

Nonostante non sia una vera e propria simulazione della realtà, in questo gioco tutto (ad eccezione dei corpi celesti) simula la dinamica newtoniana: un razzo si sposterà in funzione delle forze in gioco e della posizione degli elementi sotto studio e le giunzioni che tengono insieme le componenti hanno resistenza limitata. Ciò può causare la rottura e la conseguente separazione delle componenti, portata dal posizionamento o un'eccessiva forza dei propulsori.

Nello sviluppo di questo gioco, viene affrontato un problema generato dal fatto che i corpi celesti, rispetto alla dimensione delle astronavi, sono infinitamente più

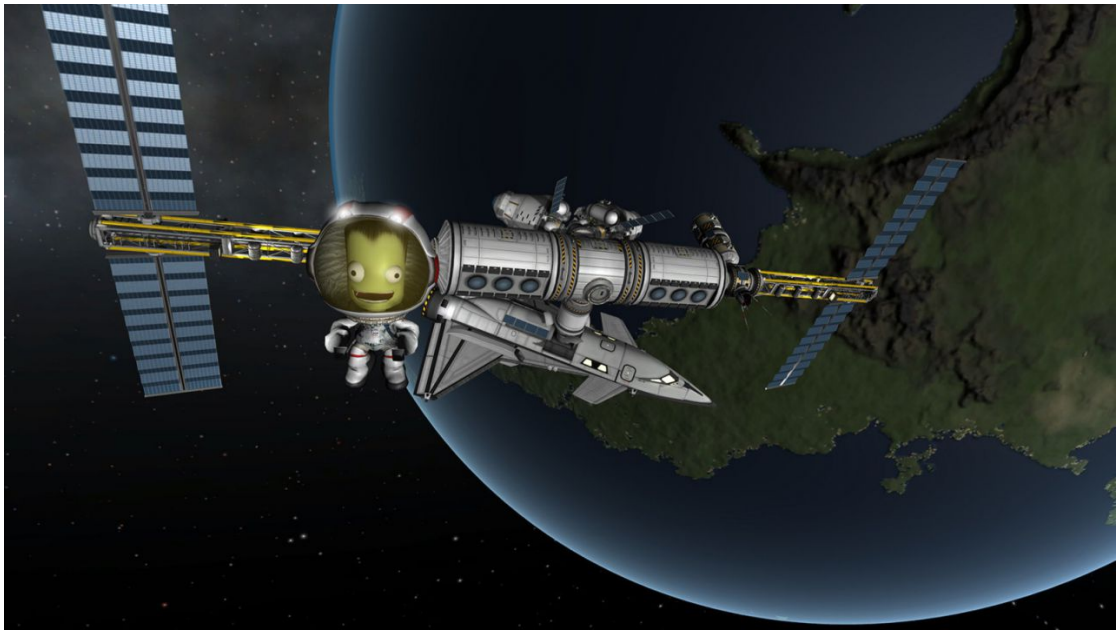


Figura 1.2: Kerbal Space Program

grandi. Infatti il mondo di Unity, per quanto vasto, non potrà mai contenere una differenza di ordini di grandezza così elevata. Questo tipo di ostacolo si va ad aggiungere al problema del **Camera Clipping**. Quando si utilizza una telecamera all'interno di una applicazione grafica, ci si deve ricordare che gli oggetti visualizzati saranno quelli all'interno del cosiddetto **Camera Frustum** (Figura 1.3), un tronco di piramide che corrisponde al volume di visualizzazione della telecamera stessa. I piani che delimitano il volume visualizzabile prendono il nome di Clipping Planes. Se un punto non ricade all'interno di questa zona, non potrà essere visualizzato. Ovviamente questi piani possono essere modificati entro un certo intervallo, che tuttavia non è sufficiente a contenere oggetti di ordini di grandezza troppo elevati.

La soluzione sfrutta la funzionalità dei **Camera Layers**, un metodo abbastanza comune che coinvolge l'utilizzo di diverse telecamere che renderizzano parti diverse dell'immagine. Ad esempio, una telecamera che renderizza oggetti vicini e una che renderizza oggetti molto lontani. Questa tecnica prende il nome di **Scale Space**.

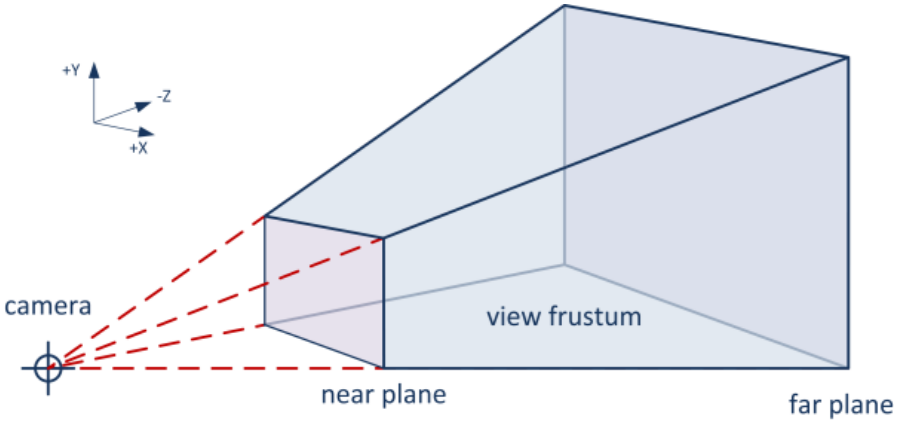


Figura 1.3: Camera Frustum

1.4 Struttura dell'elaborato

In questo lavoro, è stato realizzato il simulatore per due CubeSat in fase di docking all'interno del Game Engine di Unity. Le manovre che vengono effettuate dal Chaser sono calcolate sul software di Simulink, e dipendono da dei dati che vengono catturati all'interno di Unity. Questi satelliti sono posizionati in un ambiente che ha come obiettivo quello di inserire il sistema all'interno di una visualizzazione realistica di ciò che accadrebbe. Infatti vengono inseriti corpi celesti, quali Sole e Terra, a distanze e grandezze in scala.

Figura 1.4: Visualizzazione del risultato finale

Questo lavoro è suddiviso su 6 capitoli: in ognuno di questi viene trattato un diverso aspetto tecnico che ha contribuito allo sviluppo di questo simulatore, valutandone i limiti e i vantaggi.

Nel **Capitolo 2** viene trattato l'approccio di modellazione dei due satelliti, utilizzando il software Blender. Viene posta particolare attenzione alle componenti che saranno utili ai fini della simulazione: il corpo del satellite e il meccanismo di docking. Quest'ultimo infatti è diverso tra il Chaser e il Target. Il primo dei due prevede un foro all'interno del quale viene posizionata una telecamera con parametri specifici, che si occuperà di catturare i dati necessari all'algoritmo per calcolare la manovra successiva. Questi dati vengono forniti dal pattern di LED posizionati sul meccanismo di docking del Target.

Nel **Capitolo 3** è presente la parte più sostanziale di questo lavoro. Viene approfondita la progettazione del simulatore all'interno di Unity. Tra i vari aspetti trattati è opportuno richiamare la costruzione dell'ambiente di simulazione secondo corretti e coerenti sistemi di riferimento. In seguito, vengono approfondite le dinamiche di processo dei due satelliti. Si definiscono contestualmente i cosiddetti Camera Layers, i quali permettono di collocare oggetti di grandi dimensioni (come corpi celesti) all'interno di un "mondo" limitato, come quello di Unity. Infine, viene trattata la creazione di un Post Processing, con lo scopo di dare una migliore resa estetica della simulazione.

Nel **Capitolo 4** viene configurata la connessione tra le due applicazioni, servendosi dell'UDP. Per ognuna delle due applicazioni viene spiegato come ricevere e come inviare i dati utili ai fini della simulazione. Inoltre, per ciascuno di questi dati, viene esaminato come è costruito, impacchettato, letto e infine utilizzato.

Nel **Capitolo 5** si effettua un confronto tra le due applicazioni, andando ad analizzare eventuali differenze sorte tra le due.

Nel **Capitolo 6**, infine, vengono discussi i limiti incontrati nella progettazione, e vengono proposti sviluppi futuri possibili.

NOTA: si ritiene necessaria una specificazione di carattere semantico. In tutta la trattazione si conviene di utilizzare il termine "rotazione" in relazione alla variazione di orientamento rispetto all'asse di riferimento. In realtà in ambito aerospaziale sarebbe più proprio far riferimento al termine "assetto" che però introdurrebbe qualche elemento di ambiguità interpretativa, nel caso dei software utilizzati.

Capitolo 2

Modellazione 3D dei CubeSat

2.1 Modello 3D di un CubeSat

In questo capitolo viene presentata la struttura di un CubeSat e di come la fase di modellazione è stata affrontata.

Un CubeSat è un nanosatellite composto da diverse unità o units, indicate con la lettera U. Ogni unità ha le dimensioni di 10 cm × 10 cm × 10 cm e massa di 1.33 kg. Ci possono essere diverse combinazioni di unità, a seconda delle necessità, come rappresentato nella Figura 2.1.

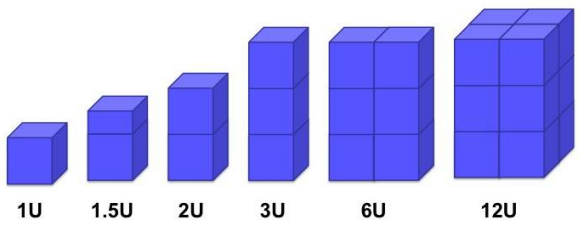


Figura 2.1: Famiglia di CubeSat composti da un numero diverso di Units

Nel nostro caso andremo a utilizzare due CubeSat 3U (Figura 2.2). Le loro dimensioni dunque saranno di 10 cm × 10 cm × 30 cm e massa di circa 4 kg ciascuno.

Oggi molti CubeSat 3U sono stati utilizzati in numerose missioni come GOMX-3, GOMX-4B, QARMAN, SIMBA, Picasso, e molte arriveranno nel prossimo futuro, come RadCube, Sunstorm, GomX-5[4].

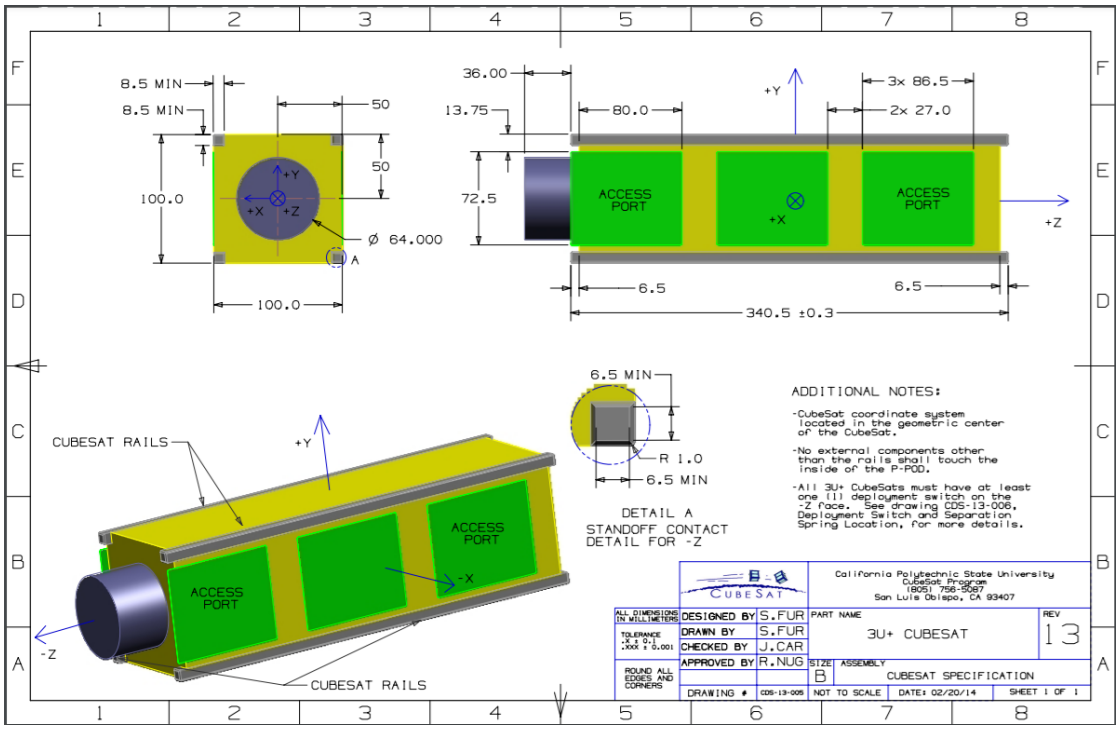


Figura 2.2: Modello 3D di CubeSat 3U di partenza

Nella missione che effettueremo ci sarà un Chaser che si avvicinerà a un Target, servendosi di un sistema di VBN, grazie a una telecamera presente su di esso, e dei LED posti sulla faccia di docking del Target. Entrambi hanno la stessa forma di base, quello che cambia è il contenuto dei satelliti e il sistema di docking. Ai fini di questa simulazione, il contenuto del satellite non è rilevante, siccome nessuna delle strumentazioni interne, fatta eccezione per la telecamera del Chaser(Figura 2.3), è stata utilizzata.

Figura 2.3: Basler ACE camera acA3800-10um[5]

2.2 Blender

Per la modellazione 3D dei CubeSat è stato utilizzato il software gratuito e open source Blender[6].

Blender Version
3.0.0

Tabella 2.1: Versione di Blender utilizzata

Il fatto che il nano satellite sia composto da moduli uguali attaccati tra loro (Unità), facilita l'approccio alla modellazione, utilizzando un modificatore Array (Figura 2.5). Questo modificatore viene utilizzato per la modellazione di oggetti che si ripetono in sequenza, andando ad evitare che uno stesso oggetto venga modellato più volte. È importante inserire manualmente le dimensioni, in modo tale che alla fine l'intero CubeSat rispetti le dimensioni corrette. L'intero modello 3D deve

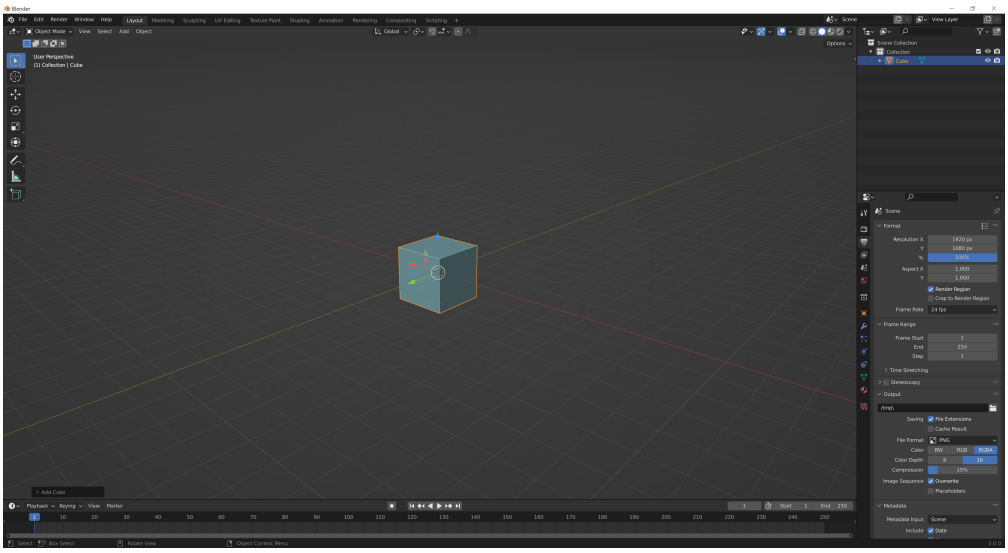
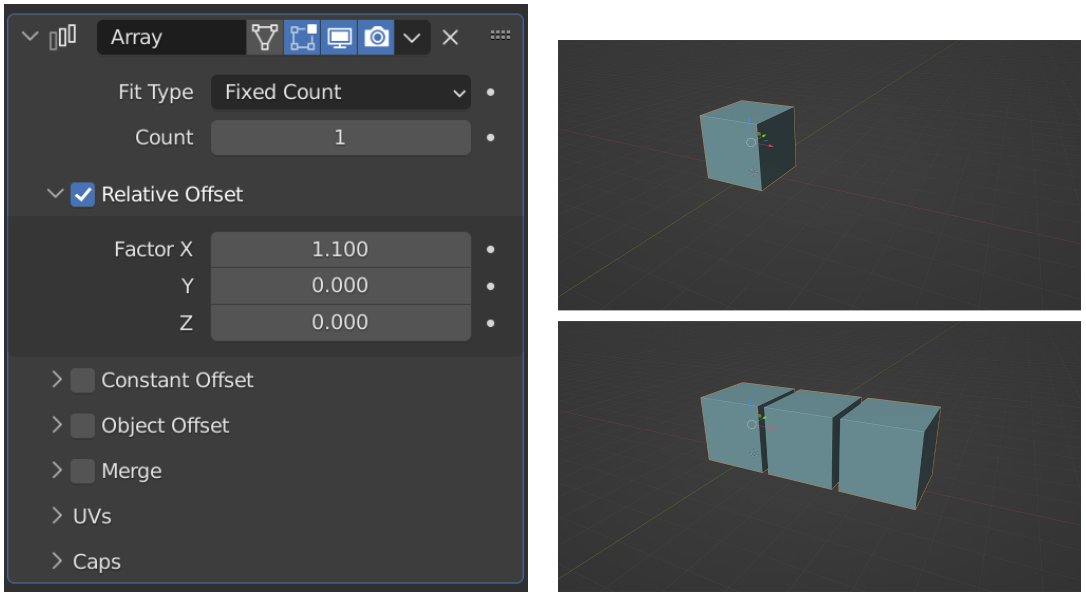


Figura 2.4: Interfaccia di Blender 3.0.0

essere posizionato in modo tale che il centro della faccia di docking sia nel centro del "mondo", perchè durante la fase di export del modello, verrà assunto come centro delle trasformazioni. Ad ogni parte del modello vengono assegnati degli Shader, che simulino un materiale metallico. Per ora il modello non include l'area di docking, che verrà trattata successivamente nel capitolo. Il modello esportato è in formato Filmbox (FBX), comodo da importare su Unity perchè assegna i materiali in modo automatico.



(a) Modificatore Array su Blender 3.0.0 **(b)** Funzionamento

Figura 2.5: Modificatore Array

Il motivo di modellare prima un corpo iniziale per entrambi i satelliti, è principalmente dovuta al fatto che i satelliti sono esternamente identici (se non per il sistema di docking).

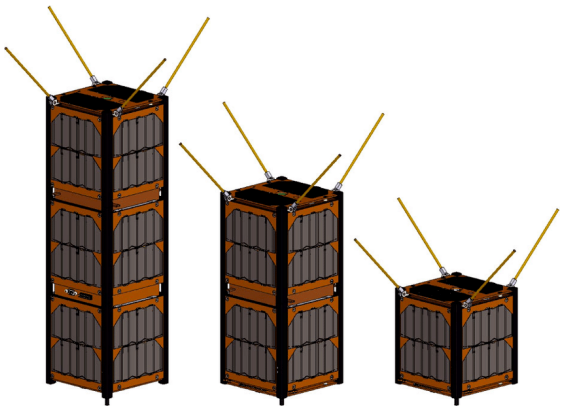


Figura 2.6: CAD isometrici di CubeSat di reference per modellazione

La modellazione del corpo si è basata su diverse references (Figura 2.6) riguardanti la progettazione e il design di CubeSat[7] ad eccezione delle antenne presenti nell'immagine (che per questa simulazione non erano consone, siccome in quella posizione avrebbero potuto causare una collisione tra i due satelliti).

2.2.1 Meccanismo di Docking

In questa sezione si esamina il meccanismo di attracco (o docking) di entrambi i satelliti. È importante sottolineare che la struttura che diversifica i due nanosatelliti, è proprio quella che si occupa del docking. Il Chaser ha sulla parte frontale della sua struttura una telecamera, che può essere integrata o meno al sistema di docking, come mostrato nella Figura 2.7. Il Target avrà a sua volta un pattern di LED, che verrà inquadrato dalla telecamera del Chaser per creare le immagini necessarie all'algoritmo di Image Processing. Questo pattern, quindi, è quello che fornisce, indirettamente, informazioni di orientamento e posizione del Target.

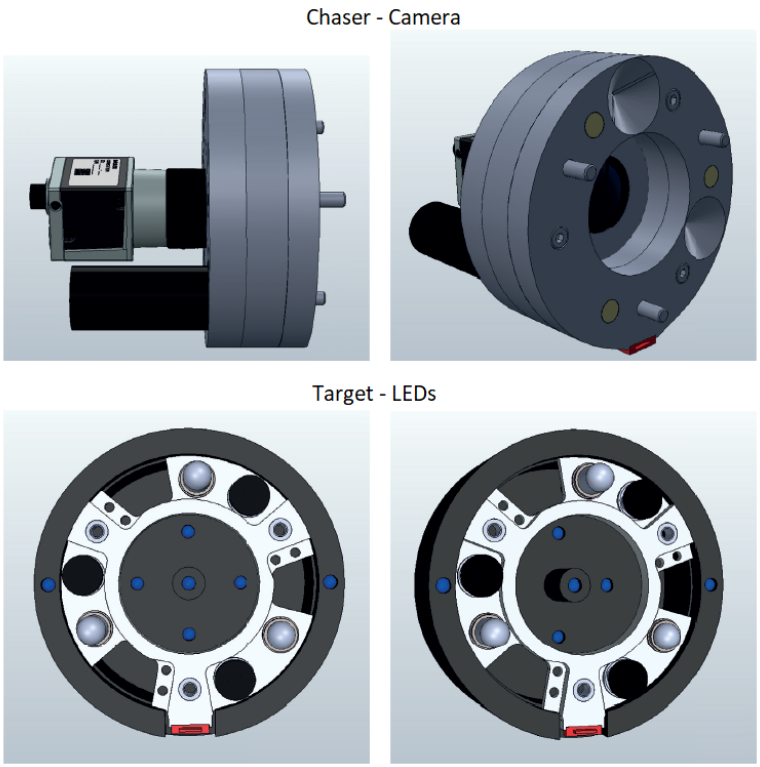
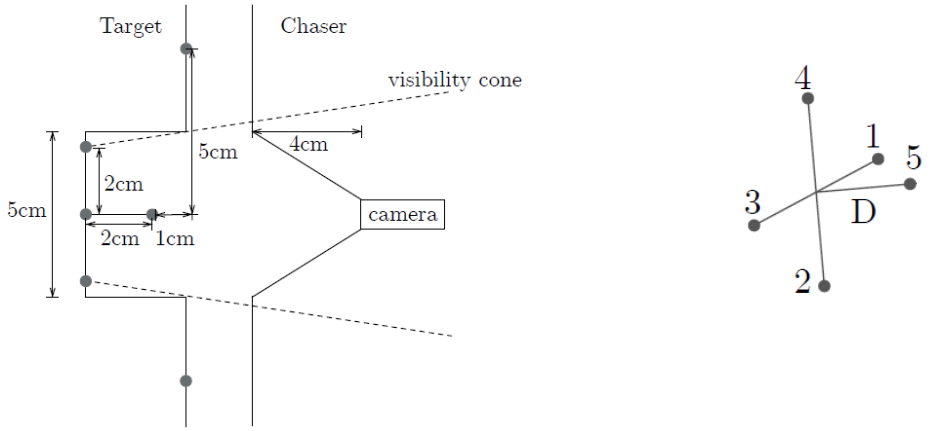


Figura 2.7: Reference dei due meccanismi di attracco dei CubeSat[5]

La simulazione parte da 5 m di distanza tra i due meccanismi e arriverà fino a un massimo di 0.05 m, questo perchè oltre questo valore entreranno in gioco forze che non sono trattate in questa tesi. In aggiunta a questo, il pattern di LED utilizzato non è universale: a seconda delle necessità si può creare una variante o più varianti che sfrutteranno comunque lo stesso principio di funzionamento[2].

2.2.2 Pattern di LED

Il Chaser ha la telecamera integrata al sistema di docking, leggermente rientrante, al fine di ridurre il rischio che i LED escano dal campo visivo, o FOV (Field of View).



(a) Vista dall'alto della disposizione dei LED

(b) Vista 3D e numerazione dei LED

Figura 2.8: Pattern[5]

Il pattern di LED del Target è costituito da una croce simmetrica, posta al centro della faccia di docking, composta da 4 LED più uno centrale (Figura 2.8). I primi 4 sono disposti sui vertici di un quadrato ruotato di 45° attorno alla normale della faccia del satellite. Il quinto LED è posizionato al centro del quadrato e sporgente rispetto ai primi 4 (Figura 2.8b). È importante specificare che i primi 4 LED sono rientranti nel Target(Figura 2.8a). La numerazione dei LED sarà necessaria per l'invio dei dati, oggetto di trattazione nel Capitolo 4.2.

I LED sono stati inseriti nell'ambiente di Unity, sotto forma di sfere con uno shader emittente. Lo scopo di ciò è una questione puramente funzionale, infatti il loro posizionamento in questo caso risulta più accurato e semplice.

Capitolo 3

Progetto di Unity3D

Nel seguente capitolo verrà presentata l'impostazione del progetto di Unity, a partire dai modelli all'interno del mondo, quali satelliti e corpi celesti, fino all'aspetto più tecnico strettamente legato alla simulazione. Infatti in questa sezione si parlerà di come sono state create le telecamere che si sono occupate di creare l'illusione della distanza di oggetti grandi. Infine, seguirà un breve cenno al Post Processing, di aspetto più estetico della simulazione.

Unity Version
2021.1.22f1

Tabella 3.1: Versione di Unity utilizzata

3.1 Sistemi di riferimento e assi

Ogni volta che si descrive la dinamica di un sistema, è necessario approfondire i sistemi di riferimento, o frame, utilizzati. In questo caso utilizzeremo lo stesso tipo di sistema presente nel progetto di partenza[2] e lo adatteremo successivamente al mondo di Unity.

- **Earth-Centred Inertial Frame:** $\mathcal{F}_I$ è definito come il sistema di riferimento inerziale della dinamica dei satelliti (Figura 3.1). Ha come origine il centro della Terra. In questa simulazione viene considerata perfettamente sferica. L'asse $\hat{Z}_I$ corrisponde all'asse di rotazione della Terra, uscente dal polo Nord. Invece, l'asse $\hat{X}_I$ giace sul piano dell'equatore, puntando verso il punto vernale. Infine, l'asse $\hat{Y}_I$ viene costruito in modo da definire un sistema destrorso[8]. Nonostante la Terra giri intorno al Sole, viene considerato un sistema di riferimento inerziale, poichè i CubeSat ruotano sulle cosiddette LEO, ovvero

orbite che hanno come centro la Terra stessa, e che hanno un'altitudine inferiore ai $2 \cdot 10^3$km.

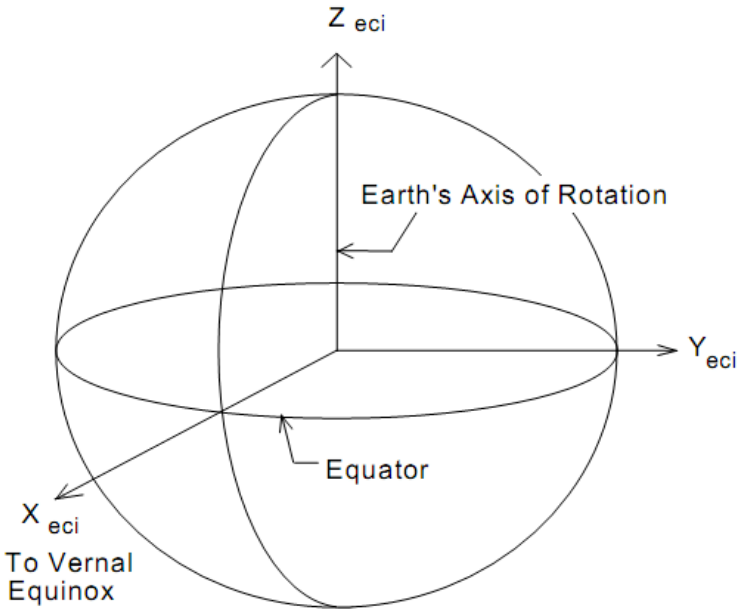


Figura 3.1: Earth-Centred Inertial reference frame [9]

- **Local-Vertical Local-Horizontal Frame:** $\mathcal{F}_O$ è definito come il sistema di riferimento orbitale (Figura 3.2). È centrato nel Centro di Massa del CubeSat Target. L'asse $\hat{z}_o$ è radiale, e punta costantemente verso il centro della Terra. L'asse $\hat{y}_o$ punta in direzione opposta al vettore momento angolare dell'orbita. Infine, l'asse $\hat{x}_o$ si ottiene con una costruzione destrorsa, e corrisponde al vettore velocità del satellite lungo l'orbita, nonostante non sia sicuro che combacino per tutto il tragitto[8].
- **Body Frame:** $\mathcal{F}_b$ è necessario, invece, a descrivere la dinamica degli angoli rispetto al sistema di riferimento dell'orbita, ad esempio se il CubeSat sta ruotando su se stesso (Figura 3.3). È centrato nel Centro di Massa del satellite[1].

[1]Dato il consumo di carburate del satellite, non si potrebbe considerare fisso il Centro di Massa. Attraverso, però, un'opportuna semplificazione, si può considerare questa variazione come disturbo, e quindi assumere il centro come fisso[2].

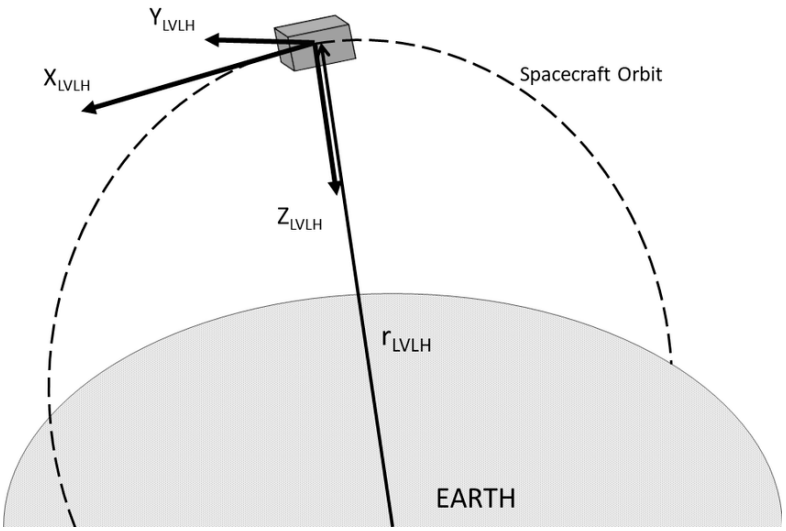


Figura 3.2: Local-Vertical Local-Horizontal reference frame [10]

- **Docking Frame:** $\mathcal{F}_d$ è posizionato in centro al meccanismo di attracco dei CubeSat, ma con delle accortezze per ciascuno. Infatti, nel caso del Chaser, $\hat{x}_{d_c}$ è uscente dalla faccia del satellite (Figura 3.3). $\hat{z}_{d_c}$ punta uscente dalla faccia sotto del satellite (scelto arbitrariamente). $\hat{y}_{d_c}$ completa il sistema destrorso. Al contrario, invece, il Target ha $\hat{x}_{d_t}$ entrante nel satellite, questo perchè dopo l'attracco, $\hat{x}_{d_c}$ e $\hat{x}_{d_t}$ possono sovrapporsi. Infine, $\hat{z}_{d_t}$ e $\hat{y}_{d_t}$ sono analoghi a quelli del Chaser.

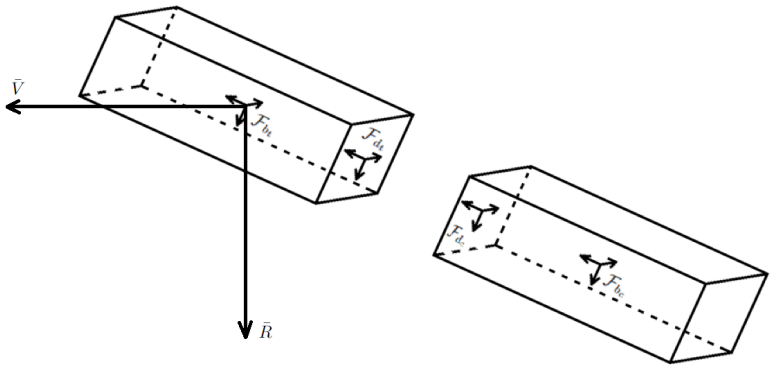


Figura 3.3: Body e docking frame di Target e Chaser nel Local-Vertical Local-Horizontal Frame

- **Navigation Frame:** $\mathcal{F}_n$ è un sistema di riferimento puramente definito ai fini del sistema di VBN, ed è differente per ciascuno dei due satelliti. Nel caso del Chaser, $\mathcal{F}_{n_c}$ è posizionato in corrispondenza del piano focale della telecamera citata precedentemente. Invece, per quanto riguarda il Target, $\mathcal{F}_{n_t}$ viene definito per semplificare il posizionamento dei LED fiduciari sulla faccia di attracco.

Conversione del sistema di riferimento in Unity

Il sistema di riferimento scelto per la simulazione non combacia con la configurazione degli assi di Unity. Occorre dunque effettuare una puntualizzazione su quello che verrà scritto successivamente riguardo alle triplette di coordinate trattate.

Assumendo una tripletta di coordinate (x, y, z) nel docking frame del Target $\mathcal{F}_{d_t}$, per ottenere una tripletta corrispondente su Unity, è necessario ricavare una matrice di passaggio.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \cdot M = \begin{bmatrix} x \\ -z \\ -y \end{bmatrix} \tag{3.1}$$

dove

$$M = \begin{bmatrix} M_{1,1} & M_{1,2} & M_{1,3} \\ M_{2,1} & M_{2,2} & M_{2,3} \\ M_{3,1} & M_{3,2} & M_{3,3} \end{bmatrix}$$

ne consegue che

$$\begin{bmatrix} M_{1,1} \cdot x + M_{1,2} \cdot y + M_{1,3} \cdot z \\ M_{2,1} \cdot x + M_{2,2} \cdot y + M_{2,3} \cdot z \\ M_{3,1} \cdot x + M_{3,2} \cdot y + M_{3,3} \cdot z \end{bmatrix} = \begin{bmatrix} x \\ -z \\ -y \end{bmatrix} \tag{3.2}$$

Andando a calcolare quindi i valori dell'equazione, si può concludere che la matrice di passaggio è

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{3.3}$$

Per quanto riguarda le rotazioni invece è necessario approfondire come viene elaborata una tripletta di angoli ϕ, θ, ψ.

$$R_1(\phi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\phi) & -\sin(\phi) \\ 0 & \sin(\phi) & \cos(\phi) \end{bmatrix} \quad \Rightarrow \quad \text{rotazione attorno asse } x \tag{3.4a}$$

$$R_2(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix} \quad \Rightarrow \quad \text{rotazione attorno asse } y \tag{3.4b}$$

$$R_3(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \Rightarrow \quad \text{rotazione attorno asse } z \tag{3.4c}$$

Le convenzioni utilizzate nelle due applicazioni sono differenti:

- **Mano destra e mano sinistra:** Unity utilizza, sia nelle rotazioni, che nel posizionamento degli assi, il metodo della mano sinistra. Questo significa che una rotazione positiva si effettua in senso orario attorno all'asse. Simulink, invece, usa il metodo della mano destra, per cui le rotazioni avvengono in senso antiorario attorno all'asse. Per risolvere questo problema si è opertato un confronto rispetto agli assi precedentemente ottenuti. Infatti, avendo determinato il nuovo sistema di assi, diventa semplice andare a riscrivere gli angoli nel nuovo sistema di riferimento.

  $$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \cdot M = \begin{bmatrix} \alpha \\ -\gamma \\ -\beta \end{bmatrix} \tag{3.5}$$

  A questo punto, siccome la rotazione va applicata in un sistema sinistrorso, è necessario aggiungere il segno negativo, altrimenti la rotazione sarebbe applicata destrorsamente. Ne ricaviamo quindi che per applicare la tripletta di angoli nel nuovo sistema di riferimento, devono essere convertiti come:

  $$-\begin{bmatrix} \alpha \\ -\gamma \\ -\beta \end{bmatrix} = \begin{bmatrix} -\alpha \\ -(-\gamma) \\ -(-\beta) \end{bmatrix} = \begin{bmatrix} -\alpha \\ \gamma \\ \beta \end{bmatrix} \tag{3.6}$$

- **Ordine di rotazione:** è fondamentale ricordare che se gli angoli sono applicati in ordine diverso, il risultato ottenuto è differente. Unity infatti applica le rotazioni nell'ordine *ZXY*, mentre le coordinate angolari ricevute da Simulink sono da considerare nell'ordine *XYZ*. Quindi a partire dalla nuova tripletta ottenuta nell'Equazione 3.6, si vedrà nel Capitolo 4.4 come applicare la funzione

Transform.Rotate presente in Unity, per leggere gli angoli in un ordine diverso, in questo caso *XYZ*.

È opportuno precisare che qualsiasi tripletta di posizione del Chaser verrà applicata nel sistema di riferimento $\mathcal{F}_{d_t}$, mentre le rotazioni nel sistema di riferimento $\mathcal{F}_{d_c}$.

3.2 Chaser

Il Chaser viene importato all'interno di Unity in modo separato. Infatti viene prima importato il corpo del CubeSat di base, e solo successivamente si importa la componente del docking all'interno della gerarchia dell'oggetto. Questo fa sì che corpo e meccanismo di docking si muovano insieme. La telecamera viene ovviamente inserita all'interno di Unity, nella posizione descritta precedentemente.

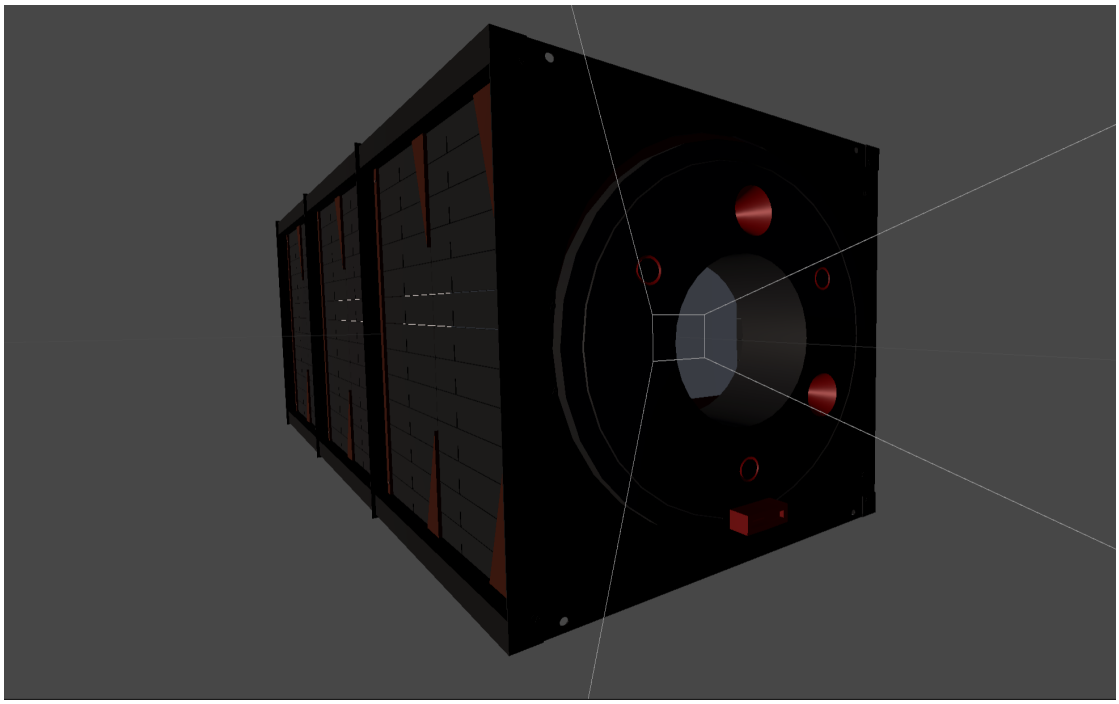


Figura 3.4: Chaser

Telecamera

Indifferentemente del Pattern che verrà scelto, i parametri della telecamera rimarranno invariati. La telecamera viene posta in centro alla faccia di docking del Chaser, rientrante di 4 cm. Siccome rappresenta la componente più importante di tutta la simulazione, è necessario assicurarsi che tutti i valori inseriti siano corretti. Trattandosi di una simulazione, è fondamentale non approssimare i parametri, per renderla il più verosimile possibile.

Su Unity sarà necessario servirsi della funzionalità della Physical Camera, poichè permette una personalizzazione più accurata della telecamera stessa, e dei valori inseribili dall'utente, garantendo più versatilità.

Tabella 3.2: Parametri della telecamera del Chaser[5]

Dimensioni sensore (H)	$6.44\,\text{mm} \times 4.62\,\text{mm}$
Risoluzione (res)	3856×2764
Lunghezza Focale (f)	$4\,\text{mm}$
Campo visivo (FOV)	60,012°
Dimensione pixel $(pixel_{dim})$	$1.67\,\mu\text{m} \times 1.67\,\mu\text{m}$

Partendo da questi valori, sarà possibile disinteressarsi all'inserimento manuale di alcuni di essi, poichè Unity si preoccupa di calcolare i parametri, nel seguente modo (da considerare coerentemente componenti verticali e orizzontali):

- **FOV:** si ricava inserendo i valori di dimensione del sensore (H) e la lunghezza focale (f)

$$FOV = 2 \times \arctan \frac{H}{2f} \tag{3.7}$$

- **Dimensione del pixel:** assumendo che un pixel sia quadrato, si ricava dalla dimensione del sensore (H) e dalla risoluzione (res), entrambi devono far riferimento o all'altezza o alla larghezza

$$pixel_{dim} = \frac{H}{res} \tag{3.8}$$

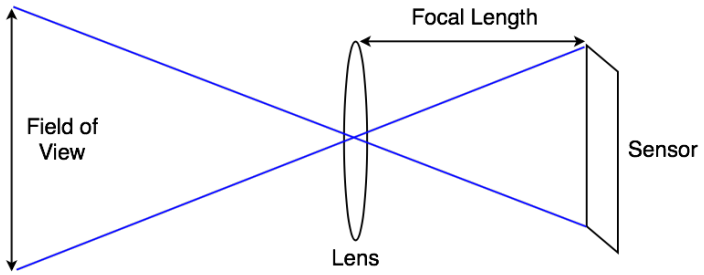


Figura 3.5: Relazione tra lunghezza focale, dimensione del sensore e FOV della telecamera

3.3 Target

Il Target viene importato all'interno di Unity nello stesso modo del Chaser. Ovvero, prima il corpo del satellite, e solo successivamente la componente del docking. I LED posizionati secondo il pattern vengono inseriti in seguito a questa procedura. Siccome la posizione del Chaser ricevuta da Simulink, lo si vedrà successivamente nel Capitolo 4.4, è relativa al docking frame del Target $\mathcal{F}_{d_t}$, sarà necessario far combaciare il centro del docking frame con il punto di coordinate $(0, 0, 0)$. Sempre facendo riferimento a questo sistema, si devono posizionare i LED, tenendo presente che $\mathcal{F}_{d_t}$ è complanare alla faccia di docking. È fondamentale rispettare la numerazione descritta precedentemente (Figura 2.8b).

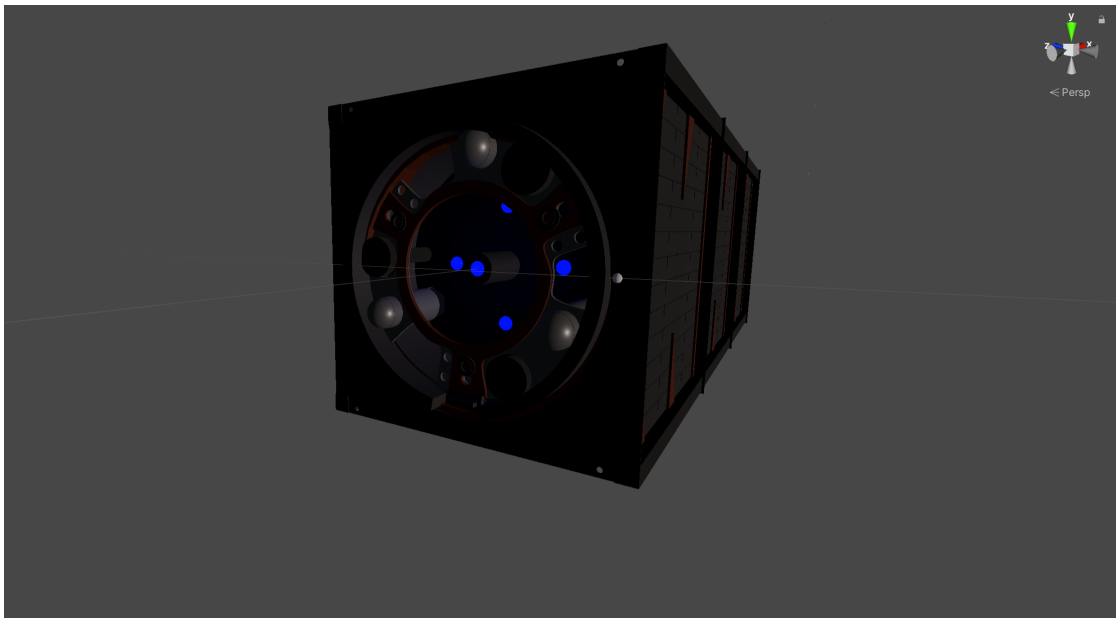


Figura 3.6: Target

Posizionamento dei LED

All'interno di Unity vengono inseriti i LED come sfere con un materiale che emette una luce. L'inserimento dei LED viene fatta relativamente al centro della faccia di docking. Tutti e cinque sono gestiti da un *empty* (parent) posizionato nelle coordinate globali $(0, -0.1, 0)$. Inserire questi oggetti all'interno di una gerarchia ne permette una migliore gestione e organizzazione, andando a semplificare quello che è anche il posizionamento di tali oggetti. Le coordinate espresse nella seguente tabella sono quindi locali rispetto al parent.

Tabella 3.3: Posizione dei LED del Target all'interno dell'*empty* (parent) in Unity

LED 1	(-0.2, -0.2, 0)
LED 2	(0, -0.2, 0.2)
LED 3	(0.2, -0.2, 0)
LED 4	(0, -0.2, -0.2)
LED 5	(0, 0, 0)

3.4 Camera Layers e illusione di grandi distanze

Dopo aver importato i modelli dei due satelliti, è necessario ricreare nel mondo di Unity le distanze in scala con la Terra, il Sole con le rispettive grandezze in scala.

È opportuno specificare che sono state create due modalità di visualizzazione, per permettere all'utente, a seconda delle esigenze, una migliore esplorazione dell'ambiente.

- **Vista libera:** l'utente è libero di muoversi con i comandi di direzionali standard, con aggiunta di controllo dell'altitudine, con i pulsanti sulla tastiera *Q* ed *E*.
- **Vista Chaser:** la visualizzazione è limitata a ciò che viene visto dalla telecamera del Chaser, senza alcuna libertà di movimento da parte dell'utente.

Mentre la telecamera del Chaser avrà dei valori ben specifici, come descritto precedentemente, la telecamera libera è funzionale solo a una visualizzazione esterna della simulazione, e quindi non ci sono particolari parametri da impostare.

Il principale problema che si presenta è che per quanto l'universo di Unity si estenda, non riuscirà mai ad effettuare una rappresentazione in scala di ordini di grandezza che partono dai metri (satellite) fino ad arrivare ai milioni di kilometri (distanza Terra-Sole), o per lo meno, una qualsiasi telecamera all'interno di Unity ha una visualizzazione limitata nel mondo, per cui è necessario studiare una via alternativa.

Per superare questo problema si utilizzerà un sistema di Layers o Livelli di telecamere implementato in Unity. Il ragionamento è quello di creare una telecamera, composta da più telecamere, che mostrino oggetti e distanze di ordini di grandezza diversi. Lo stesso ordine di grandezza verrà utilizzato per lo spostamento della telecamera, per creare l'illusione della distanza, sfruttando il principio di **parallasse di movimento**.

Questo principio è una componente fondamentale per la percezione dello spazio e della profondità dell'osservatore. È un effetto (monoculare) che si percepisce

in una situazione di movimento: rispetto a un punto di fissazione, infatti, gli oggetti più lontani sembrano muoversi più lentamente, mentre quelli più vicini si muovono più velocemente (Figura 3.7). Un esempio chiaro di questo fenomeno si può sperimentare guardando fuori dal finestrino quando si viaggia in macchina o in treno. Ciò che si vuole ottenere è questo effetto, partendo da oggetti infinitamente più piccoli di quelli reali e facendoli sembrare molto più grandi.

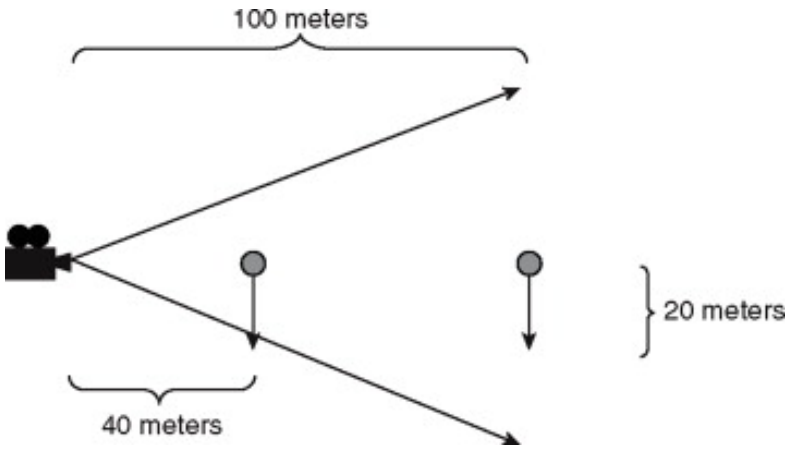


Figura 3.7: Parallasse di movimento[11]

Grazie ai Camera Layers, si può costruire un'immagine composta da più visualizzazioni (Cameras o telecamere). Ad ognuna di queste viene assegnato uno script che ne definirà l'ordine di grandezza. A un ordine di grandezza maggiore ne corrisponderà un movimento più lento, come definito prima dal principio di parallasse di movimento. Infine tutti i Layers vengono sovrapposti a partire dalla telecamera con ordine grandezza maggiore, fino ad arrivare alla telecamera con ordine di grandezza più basso, che nel nostro caso è quella che vede solo i due satelliti, come rappresentato nella Figura 3.8.

Tabella 3.4: Camera Scales

Scale Camera1	10^{6}
Scale Camera2	10^{3}
Scale Camera3	10^{-3}

In questo caso si utilizzano tre diverse telecamere, secondo il criterio presentato nella Tabella 3.4. Ogni ordine di grandezza è definito in riferimento a 1 km, ciò significa che per ogni movimento di un'unità nel mondo di Unity il valore

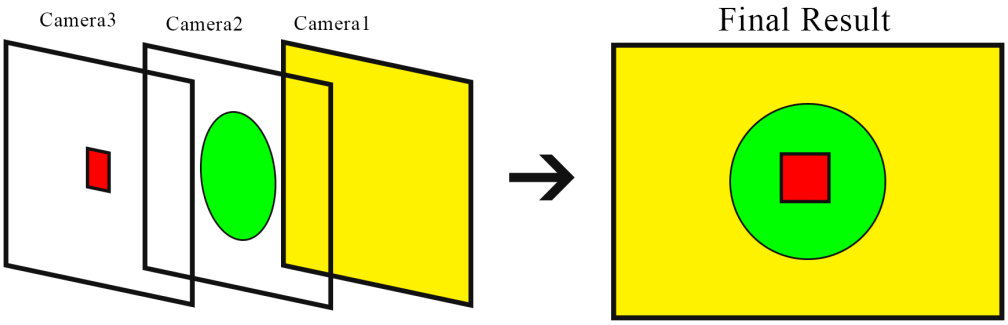


Figura 3.8: Camera Layers

corrispondente sarà di 1 km × $scale_N$. Per ogni telecamera quindi, un qualsiasi spostamento viene moltiplicato per $\frac{1}{scale_N}$.

Nel Listing 3.1 viene mostrato lo script inserito come componente di ognuna delle tre telecamere coinvolte, ovviamente con valori di scala diversi.

Listing 3.1: Camera scale manager script

```
1  public class CameraManager : MonoBehaviour
2  {
3      public float scale; //Fattore di scala
4      private float normalSpeed = 0.0005f;
5      float mouseSensitivity = 100.0f;
6      float clampAngle = 90.0f;
7      private float rotY = 0.0f;
8      private float rotX = 0.0f;
9
10     void Update()
11     {
12         //Gestione dell'input di mouse e tastiera per il movimento
13         float mouseX = Input.GetAxis("Mouse X");
14         float mouseY = -Input.GetAxis("Mouse Y");
15         float x = Input.GetAxis("Horizontal");
16         float z = Input.GetAxis("Vertical");
17
18         rotX += mouseX * mouseSensitivity * Time.deltaTime;
19         rotY += mouseY * mouseSensitivity * Time.deltaTime;
```

```
20
21          rotY = Mathf.Clamp(rotY, -clampAngle, clampAngle);
22
23          transform.localRotation = Quaternion.AngleAxis(rotX, Vector3.
    up);
24          transform.localRotation *= Quaternion.AngleAxis(-rotY,
    Vector3.left);
25
26          //Movimento moltiplicato per 1/scale
27          transform.position += transform.forward * normalSpeed * z *
    Time.deltaTime * (1 / scale);
28          transform.position += transform.right * normalSpeed * x *
    Time.deltaTime * (1 / scale);
29
30          //Comandi per controllo dell'altitudine della telecamera
31          if (Input.GetKey(KeyCode.Q))
32          {
33              transform.position += transform.up * normalSpeed * Time.
    deltaTime * (1 / scale);
34          }
35
36          if (Input.GetKey(KeyCode.E))
37          {
38              transform.position -= transform.up * normalSpeed * Time.
    deltaTime * (1 / scale);
39          }
40      }
41
42  }
```

3.5 Visualizzazione degli oggetti

Successivamente verrà presentata, per ciascuno dei corpi all'interno di Unity, la scelta della telecamera e quindi di ordine di grandezza, basandosi sulla Tabella 3.4. Inoltre sarà affrontata la discussione dei materiali utilizzati per i vari oggetti.

3.5.1 CubeSat

Posizionando i due CubeSat all'interno del mondo di Unity, è necessario, come per i corpi celesti, andare a impostare le loro corrette dimensioni, con la relativa visualizzazione da parte della telecamera corretta. La scelta della telecamera dipende dall'ordine di grandezza in cui si visualizzerà l'oggetto. È opportuno ricordare che le dimensioni dei due nanosatelliti sono $10\,\text{cm} \times 10\,\text{cm} \times 30\,\text{cm}$. La scelta più opportuna di ordine di grandezza, e quindi a quale telecamera fare riferimento per la visualizzazione, è chiaramente la $Camera3$. Quest'ultima infatti si occupa della visualizzazione degli oggetti di ordine di grandezza dei metri. Su Blender la scala è impostata secondo le dimensioni descritte in precedenza. È però importante puntualizzare che le dimensioni vengono normalizzate durante l'export. Quindi le proporzioni saranno corrette, anche se su Unity si vede la scala impostata come $(1, 1, 1)$.

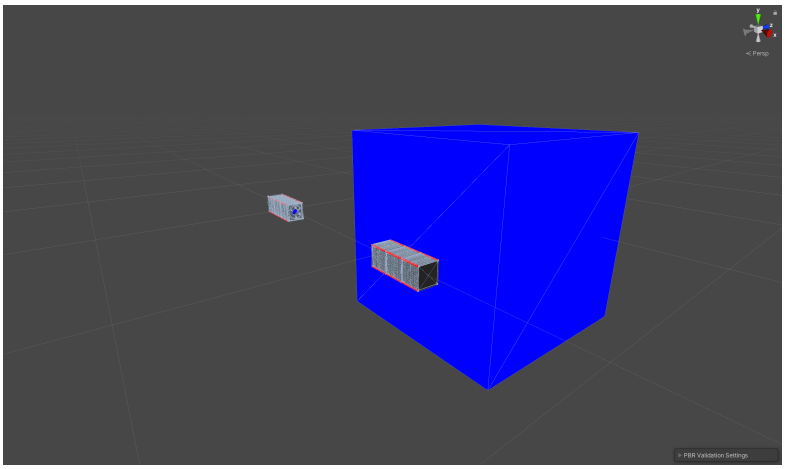


Figura 3.9: Oggetti CubeSat messi a confronto con un cubo unitario su Unity

Il primo satellite che viene posizionato è il Target, in modo tale che il centro della faccia di docking combaci con il punto $(0, 0, 0)$ del mondo di Unity, e il suo orientamento sia tale per cui l'asse $\hat{x}_{d_t}$ combaci con l'asse x del Game Engine, l'asse $\hat{y}_{d_t}$ punti nel verso opposto rispetto al centro della Terra e $\hat{z}_{d_t}$ sia uscente rispetto alla faccia bassa del nanosatellite. Il Chaser, invece, può essere posizionato in modo arbitrario, facendo attenzione che le facce di docking siano una fronte all'altra.

Infatti appena partirà la simulazione, posizione e orientamento del Chaser verranno forniti direttamente dal software esterno.

Materiali e Shader

In entrambi i satelliti, gli Shader sono stati impostati su Blender, ed esportati insieme al modello FBX. I componenti dei satelliti, ed in particolare dei CubeSat, sono principalmente in alluminio, questo perchè particolari proprietà fisiche (densità, dilatazione termica e resistenza radioattiva) e proprietà meccaniche (durezza e resistenza) devono essere necessariamente rispettate[12]. Per questo motivo è stato scelto un materiale che richiamasse l'alluminio, gestendo i valori di roughness (ruvidità) e metallic (metallico) presenti nel software. È opportuno ricordare che, in seguito all'import dell'oggetto su Unity (a seconda delle versioni), bisogna estrarre i materiali in una cartella all'interno del progetto, in modo tale che si possano modificare anche in un secondo momento. Nella Figura 3.17 si può osservare un render effettuato su Blender con il motore grafico di Cycles.

Figura 3.10: Materiali di un CubeSat

3.5.2 Terra

Per andare ad inserire la Terra in scala all'interno di Unity, è necessario prima definire da quale telecamera verrà effettuata la visualizzazione. Ovviamente i valori di scala scelti sono stati definiti proprio in relazione ai corpi che vengono visualizzati. Infatti la scala della Terra, come indicato nella Tabella 3.5, ha un raggio che è nell'ordine di grandezza delle migliaia di kilometri, ragion per cui verrà visualizzata dalla $Camera2$ (Tabella 3.4).

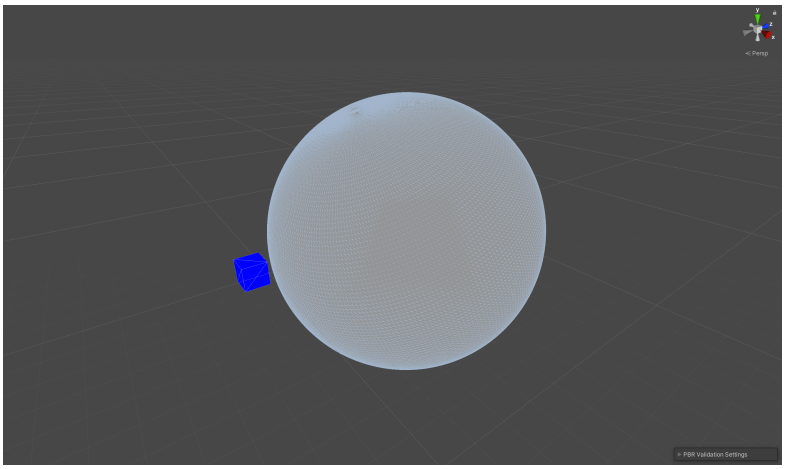


Figura 3.11: Oggetto Terra messo a confronto con un cubo unitario su Unity

Questo significa, che tutto ciò che concerne le dimensioni e la distanza della Terra dai satelliti, va messo in relazione con il fattore di scala della $Camera2$.

Tabella 3.5: Grandezze relative alla Terra[13]

Raggio della Terra	$r_E = 6378{,}137$ km
Altitudine dell'orbita	$z_O = 400$ km
Raggio dell'orbita	$r_O = z_O + r_E = 6778{,}137$ km

Quindi, se il fattore di scala è $scale_2 = 10^3$, la tripletta di coordinate che si occupa di definire la scala[2] dell'oggetto Terra in Unity sarà

$$\left(S_{E_x}, S_{E_y}, S_{E_z}\right) = \left(\frac{r_E \cdot 2}{scale_2}, \frac{r_E \cdot 2}{scale_2}, \frac{r_E \cdot 2}{scale_2}\right) \tag{3.9}$$

[2]Su Unity la scala di una sfera ne esprime il diametro, ecco perchè il raggio viene moltiplicato per 2.

mentre la tripletta che definisce la posizione della Terra nel mondo di Unity è

$$(x_E, y_E, z_E) = \left(0, 0, -\frac{r_E + z_O}{scale_2}\right) \tag{3.10}$$

La scelta di mettere il segno negativo per z_E è semplicemente per porre i CubeSat tra la Terra e il Sole.

Poligoni della sfera

Quando si tratta di scegliere quale tipo di primitiva utilizzare per rappresentare una sfera, si incontrano numerose casistiche con precise proprietà a seconda degli scopi. Su Blender, è possibile lavorare con due primitive.

- **UV Sphere:** è composta da due valori, che sono anelli e segmenti. Come costruzione, corrispondono al concetto di meridiani e paralleli utilizzati in geografia. Anche in questo caso la sfera è approssimata con multiple facce quadrangolari (ad eccezione dei poli, in cui le facce sono triangolari). L'utilizzo di facce quadrangolari rispetto a quelle triangolari permette una maggiore approssimazione di una sfera come mostrato in Figura 3.14 a fronte di una maggiore difficoltà di gestione da parte della scheda grafica. La sfera UV, inoltre, è simmetrica. Viene principalmente utilizzata per semplificare il processo di texturizzazione, ovvero di applicazione di una texture.

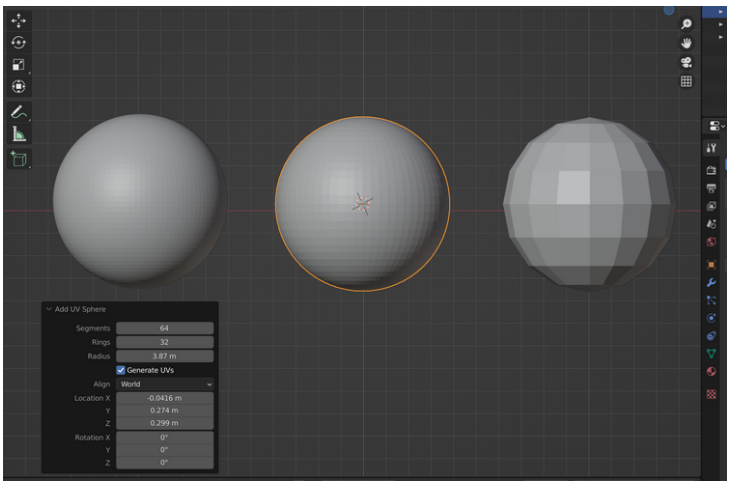


Figura 3.12: UV Sphere su Blender

- **Icosphere:** è un solido platonico[3] con 20 facce (che possono aumentare a seconda della risoluzione). Inoltre, matematicamente parlando, non possono essere incluse curve, per cui la sua rappresentazione è un'approssimazione di una sfera perfetta. Infine, ogni faccia dell'icosfera è triangolare. Un altro importante particolare è che non è una figura simmetrica. Viene principalmente utilizzata per semplificare (a seconda dei casi) la modellazione. Più complesso è il processo di texturizzazione.

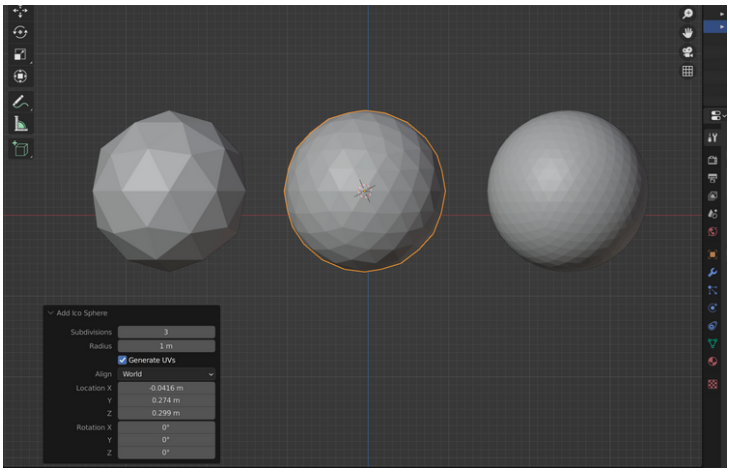


Figura 3.13: Icosphere su Blender

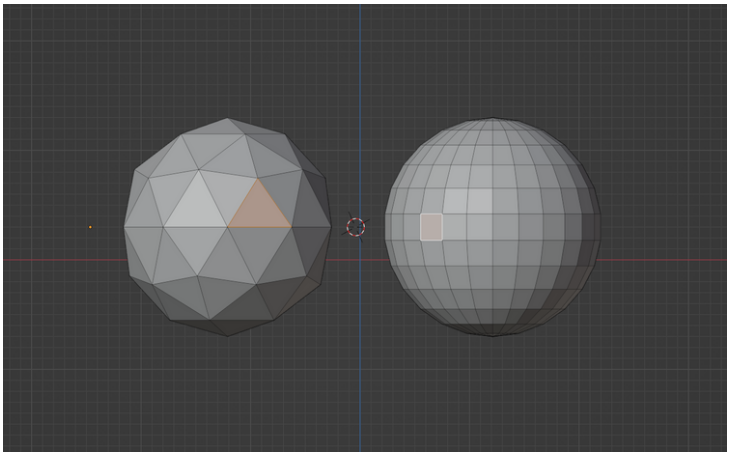


Figura 3.14: Confronto di UV Sphere e Icosphere su Blender

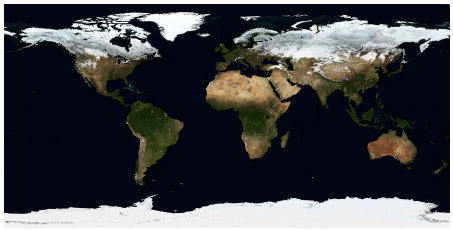
[3]Detto anche poliedro regolare, è un poliedro convesso le cui facce sono poligoni regolari tra loro congruenti e i cui angoloidi ai vertici sono tutti della stessa ampiezza.[14]

In questo caso, per la Terra è necessario applicare texture di vario tipo per il materiale. Di conseguenza, per quanto spiegato sopra, la miglior scelta è una UV Sphere.

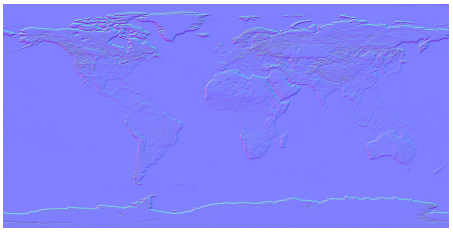
Materiale e Texture

Per creare un materiale che simuli in modo corretto la Terra vista dallo spazio, ci si serve di texture di diverso tipo. Essendo l'oggetto Terra di Unity molto piccolo e vicino alla telecamera, ci si serve di texture ad alta definizione. La tipologia di queste immagini varia a seconda della loro funzionalità. Il materiale che si andrà a creare usufruirà di tre texture diverse:

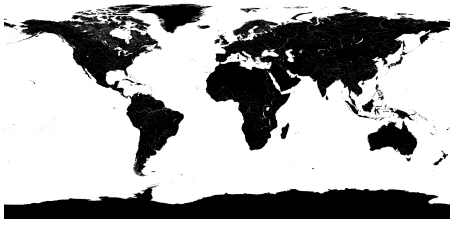
- **Diffuse:** questa texture si occupa di fornire informazioni riguardanti il colore dell'oggetto. L'intensità di colore che verrà visualizzata dipende dalla quantità di raggi luminosi colpiscono quel punto. (Figura 3.15a)
- **Normal:** questa texture si occupa di fornire informazioni che permettono di simulare una complessa geometria dell'oggetto senza andarla effettivamente a modificare. Le informazioni di colore $R(rosso), G(verde), B(blu)$ forniscono le coordinate di X, Y, Z della normale della rispettiva superficie. (Figura 3.15b)
- **Specular:** questa texture è in scala di grigi, ed è più semplice delle altre due. Definisce dove la luce dovrà riflettere (bianco) e dove invece non dovrà farlo (nero), con le corrispondenti sfumature. (Figura 3.15c)



(a) Diffuse Texture



(b) Normal Texture



(c) Specular Texture

Figura 3.15: Texture per la Terra

Un secondo materiale che viene utilizzato per la Terra è quello che rappresenta l'atmosfera. È stato aggiunto un oggetto Sfera che presenta un materiale trasparente con una tonalità azzurra.

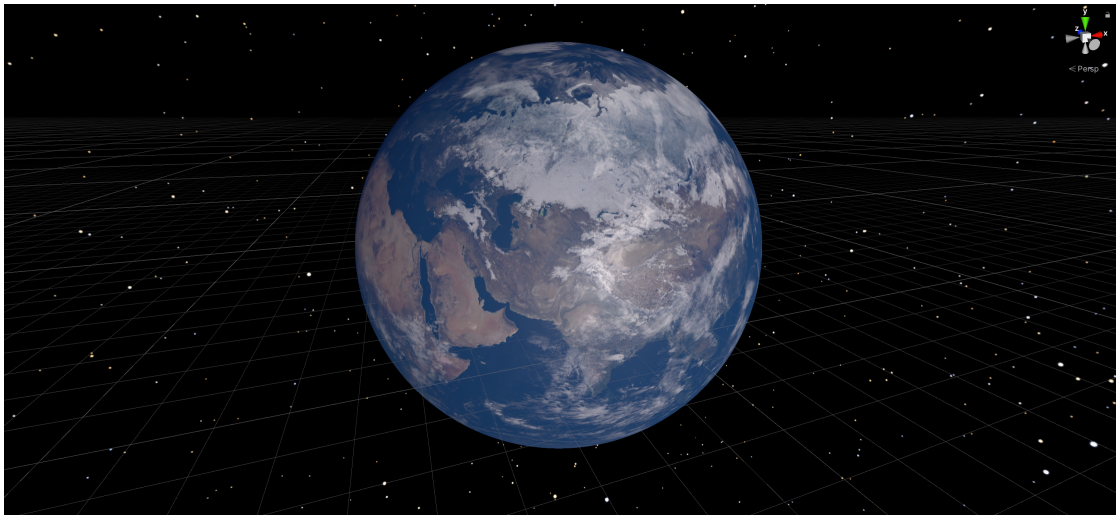


Figura 3.16: Terra

3.5.3 Sole

Per andare ad inserire il Sole in scala, all'interno di Unity, è necessario prima definire da quale telecamera verrà effettuata la visualizzazione. Come già effettuato per la Terra, ci si basa sulle dimensioni del Sole presentate nella Tabella 3.6. Entrambi i valori viaggiano sull'ordine dei grandezza dei milioni di kilometri, ragion per cui verrà visualizzato dalla $Camera1$ (Tabella 3.4).

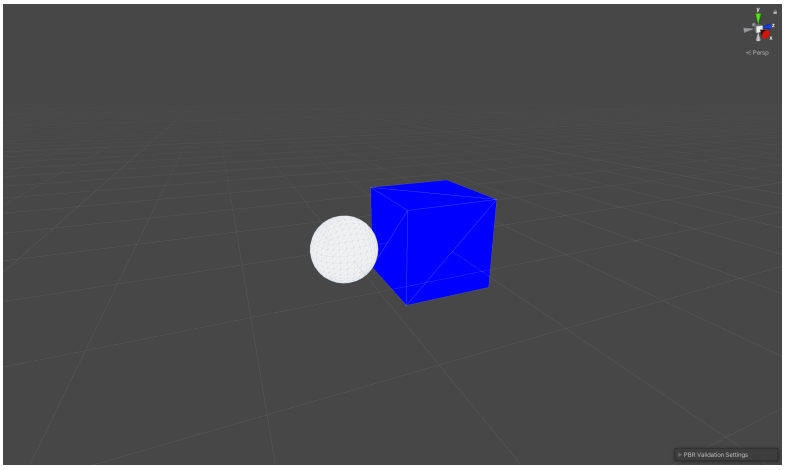


Figura 3.17: Oggetto Sole messo a confronto con un cubo unitario su Unity

Le dimensioni dell'oggetto Sole saranno quindi calcolate in relazione al fattore di scala della $Camera1$.

Tabella 3.6: Grandezze relative al Sole[15]

Raggio del Sole	$r_S = 0{,}6955 \cdot 10^6$ km
Distanza Sole-Terra	$d = 149{,}6 \cdot 10^6$ km

Il fattore di scala è quindi $scale_1 = 10^6$, per cui si effettua lo stesso procedimento di calcolo effettuato per la Terra. La tripletta di valori che determinano la scala dell'oggetto Sole su Unity si calcolerà con la formula

$$(S_x, S_y, S_z) = \left(\frac{r_S \cdot 2}{scale_1}, \frac{r_S \cdot 2}{scale_1}, \frac{r_S \cdot 2}{scale_1}\right) \tag{3.11}$$

mentre la tripletta che definisce la posizione del Sole nel mondo di Unity è

$$(x_S, y_S, z_S) = \left(0, 0, -\frac{d}{scale_1}\right) \tag{3.12}$$

La scelta di mettere il segno negativo per z_S è semplicemente per porre i CubeSat tra la Terra e il Sole.

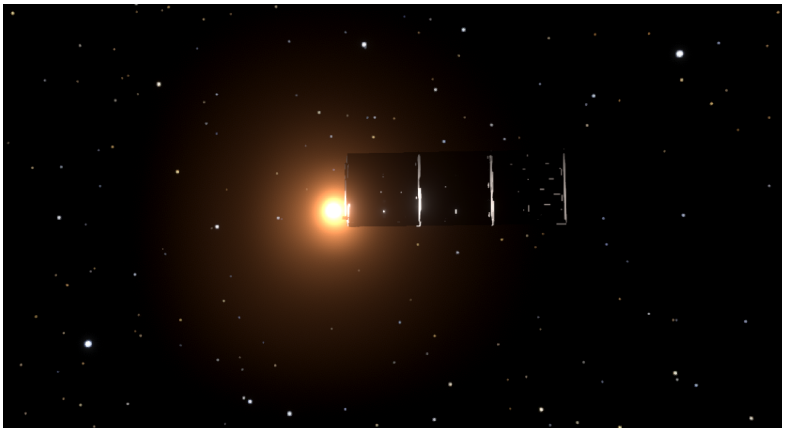
Materiale

Per il Sole non è stato scelto un vero e proprio materiale. Infatti, la scelta più ragionevole è stata quella di impostarlo come punto luce all'interno della scena. Siccome la distanza è piuttosto elevata, il raggio di illuminazione è stato definito sufficientemente grande, in modo che la Terra fosse illuminata, ed oltre un certo valore, il risultato non cambia. È possibile ricavare il colore di emissione del Sole attraverso la sua temperatura basandosi sul **Planckian locus**[16][17].

Tabella 3.7: Colore RGB a partire da temperatura in Kelvin

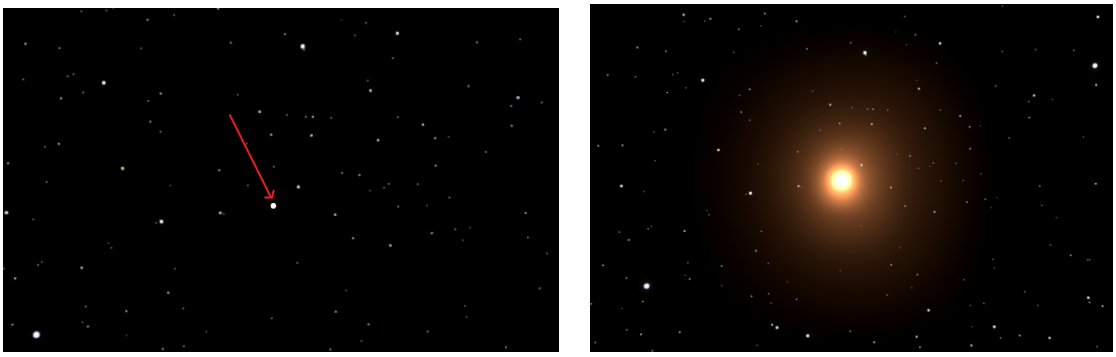
Temperatura espressa in gradi Kelvin	**Tripletta di coordinate RGB**
5800K ca.	$(255, 240, 233)$

Figura 3.18: Sole



3.6 Post Processing

Il Post Processing che è stato aggiunto ha come unico scopo fornire una migliore resa grafica alla simulazione. Il principale motivo risiede nel fatto di rendere realistico il Sole. Quest'ultimo infatti appare come una sfera bianca in lontananza, senza ricordare in alcun modo il Sole. Il profilo di Post Processing che permette di risolvere questo problema è il cosiddetto **bloom**. Questo effetto permette la simulazione di artefatti di immagine che forniscono realismo alle sorgenti luminose e/o riflettenti. Il funzionamento consiste nell'estendere la luminosità all'esterno dei bordi dell'oggetto stesso.

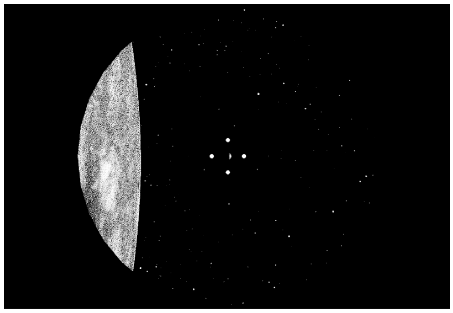


(a) Sole senza Bloom **(b)** Sole con Bloom

Figura 3.19: Bloom

Un secondo Post Processing è stato aggiunto alla vista della telecamera del Chaser. Lo scopo è dare un'idea di visualizzazione da parte del satellite. Per creare tale effetto si è abbassata la **saturazione** ed alterato il valore di **contrasto**. Successivamente si è inserito un **rumore bianco** all'immagine. Queste modifiche di visualizzazione non hanno in alcun modo coinvolto i valori strettamente coinvolti alla simulazione.

Figura 3.20: Vista del Chaser in seguito al Post Processing



Capitolo 4

Comunicazione UDP

4.1 UDP e TCP

Quando si deve creare una connessione tra due applicazioni si deve scegliere, a seconda delle necessità, tra UDP e TCP. La principale differenza che intercorre tra i due è l'assicurazione che il dato che parte dal trasmettitore arrivi al ricevitore, ovviamente a costo di velocità di trasmissione[18].

Infatti, il TCP è il più affidabile dei due, e quando deve instaurare una connessione tra due dispositivi, si preoccupa di verificarne la connessione, con un Three Way Handshake[1]. Tuttavia, questa continua richiesta di conferma da parte del ricevitore ne rallenta di molto la trasmissione di dati.

Per questo motivo è stata scelta per questo progetto una connessione UDP. La simulazione di satelliti in orbita intorno alla Terra deve essere gestita con il minor tempo di trasmissione possibile, a fronte di evitare invio di coordinate di posizione, che nel frattempo sono cambiate, e che quindi possano portare in collisione i due satelliti o di perdere di vista il Target.

Configurazione della connessione

Per creare una comunicazione bidirezionale tra le due applicazioni, è fondamentale permettere a ciascuna delle due parti di inviare e ricevere dei dati. Questi dati

[1]Metodo di creazione di una connessione local host/client, suddiviso in tre fasi:

1) Trasmettitore invia pacchetto SYN di sincronizzazione all'indirizzo del ricevitore.

2) Ricevitore invia pacchetto di ricezione SYN-ACK al trasmettitore.

3) Trasmettitore dopo aver ricevuto il pacchetto, invia il pacchetto finale e di conferma ACK al ricevitore.

saranno nel formato che è stato preventivamente scelto in modo opportuno, a seconda del tipo di dato che si vuole inviare. Ciascuna delle due applicazioni ha un metodo diverso di invio e ricezione. Nelle seguenti sezioni, si entra nel dettaglio di come è stata effettuata la comunicazione da entrambe le estremità. L'idea di fondo è che per inviare un qualsiasi tipo di pacchetto con UDP è necessario specificare un indirizzo IP, che va a specificare - in questo caso - a quale computer inviare il dato, seguito da una Port, che indica in quale particolare "porta" entrare.

Per spiegare meglio questo concetto, prendiamo come esempio il caso utilizzato in questo lavoro: lavorando sullo stesso computer, è sufficiente un unico indirizzo IP, che nel nostro caso sarà 127.0.0.1. Questo indirizzo IP corrisponde al *localhost*, ovvero l'indirizzo standard che si riferisce al computer che si sta utilizzando. Dunque, se entrambe le applicazioni utilizzano lo stesso indirizzo, è proprio grazie alla Port che si distinguono le due porte di entrata dei dati. Entrambe le parti quindi invieranno i dati allo stesso indirizzo IP (che identifica il computer), ma a Port diverse. Infatti, la Port di Unity sarà 1234, mentre la Port di Simulink sarà 4321. Quindi se avessimo la possibilità di visualizzare un pacchetto in viaggio, che ha come destinazione 127.0.0.1 - 1234, sapremmo che sta viaggiando verso Unity. Viceversa, se la destinazione fosse 127.0.0.1 - 4321, il pacchetto è in direzione di Simulink. Essendo una comunicazione bidirezionale, i dati viaggeranno in entrambe le direzioni, contemporaneamente.

Sincronizzazione

Prima di creare una qualsiasi simulazione, è necessario sincronizzare l'invio dei dati da entrambe le parti. In questa sezione verranno brevemente citati i blocchi e la porzione di codice che ha permesso un corretto flusso di dati. Il principale problema nasce dalla natura del software di Simulink che, per come è impostato, sviluppa la simulazione nel modo più veloce possibile. Questa velocità ha molteplici fattori tecnici, che dipendono principalmente dalla capacità di calcolo del computer su cui si lavora. Senza un dovuto rallentamento, la simulazione viaggia molto più velocemente del tempo reale. Infatti, in una delle prime prove di simulazione, in un secondo di tempo reale la simulazione ne aveva già simulati dieci.

In generale, il corretto approccio era quello di inviare e ricevere dati alla stessa velocità. Unity esegue le sue operazioni nella funzione di Update una volta ogni frame. Come prima cosa, quindi, è stato necessario fissare il numero di frame al secondo, con la riga di codice mostrata nel Listing 4.1, inseribile in un qualsiasi script.

Listing 4.1: Fissare il frame rate

```
void Start()
{
    Application.targetFrameRate = 50;
}
```

Questa riga di codice fondamentalmente fissa il frame rate a 50 frame al secondo. La funzione di Update viene quindi eseguita ogni 0.02 s. È necessario, a questo punto, andare a fissare il medesimo valore di invio e ricezione dati anche su simulink. Per farlo è necessario entrare nelle impostazioni del blocco UDP, e impostare il sample time a 0.02 s (Figura 4.1).

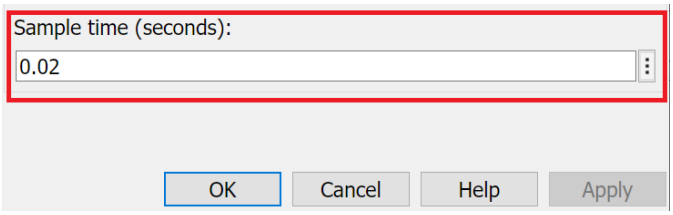


Figura 4.1: Simulink UDP sample time

4.2 UDP Send Unity Script

Come già citato in sezioni precedenti, Unity si occuperà di inviare a Simulink le coordinate dei LED del Target, dal punto di vista della telecamera del Chaser.

Listing 4.2: Namespaces utilizzati in UDPSend

```
using UnityEngine;
using System.Collections;
using System;
using System.Text;
using System.Net;
using System.Net.Sockets;
using System.Threading;
using System.Collections.Generic;
```

Appena la simulazione si avvia, è necessario definire dove mandare i dati. Qui vengono inseriti indirizzo IP e Port (Listing 4.3).

Listing 4.3: Definizione IP e Port in UDPSend

```
    public void init()
    {
        // Indirizzo IP e port
        IP = "127.0.0.1";
        port = 4321;

        // Send
        remoteEndPoint = new IPEndPoint(IPAddress.Parse(IP), port);
        client = new UdpClient();

    }
```

Ovviamente i dati vanno organizzati e impacchettati in modo utile ai fini della simulazione, in modo tale da ottimizzarne l'invio. Da parte di Unity, i valori da inviare sono le coordinate dei pixel dei 5 LED, per un totale di 10 valori. Trattandosi di coordinate di pixel, la tipologia più intuitiva per ciascuna coordinata è sicuramente l'utilizzo di numero interi (integer). In seguito a questo, tutti i valori vengono inseriti in un unico vettore. È opportuno specificare che le coordinate dei LED sono inserite nell'ordine stabilito nella Figura 2.8b. Inoltre, per una questione di semplicità, il vettore avrà come primi 5 valori le coordinate $pixel_x$ di ciascun pixel, mentre gli ultimi 5 saranno $pixel_y$:

$$(pixel_{x_1}, pixel_{x_2}, pixel_{x_3}, pixel_{x_4}, pixel_{x_5}, pixel_{y_1}, pixel_{y_2}, pixel_{y_3}, pixel_{y_4}, pixel_{y_5})$$

All'inizio di questo vettore viene inoltre inserito un numero che indica la lunghezza del vettore stesso. Inoltre, per poter inviare dati, è necessario convertirli prima in byte, e infine inviarli (Listing 4.4).

Listing 4.4: Invio del vettore di integer (coordinate dei pixel)

```

//Conversione da integer a byte
    public byte[] IntArrayToByteArray(int[] intArray)
    {
        int totalBytes = (intArray.Length * 4)+4;  //Gli integer
    occupano 4 byte, più l'aggiunta dell'integer iniziale che
    definisce la lunghezza del vettore
        byte[] serialized = new byte[totalBytes];  //Vettore di byte
    che restituiremo

        List<byte[]> listOfBytes = new List<byte[]>();  //Una lista
    temporanea di vettori di byte di integer convertiti
        foreach (int i in intArray)
        {
            byte[] converted = BitConverter.GetBytes(i);  //Converte
    un integer in un vettore di 4 byte
            listOfBytes.Add(converted);
```

```
13          }
14
15          //Assemblaggio del vettore di byte finale
16          int location = 0;
17
18          Array.Copy(BitConverter.GetBytes(intArray.Length), 0,
    serialized, location, 4);
19          location += 4;
20          foreach (byte[] ba in listOfBytes)
21          {
22              Array.Copy(ba, 0, serialized, location, 4);
23              location += 4;
24          }
25
26          return serialized;
27      }
28
29  //Funzione per l'invio del vettore all'indirizzo IP finale
30      private void sendIntArray(byte[] intArrayBytes)
31      {
32          client.Send(intArrayBytes, intArrayBytes.Length,
    remoteEndPoint);
33
34      }
35
36  //Invio definitivo del dato
37      private void Update()
38      {
39
40          sendIntArray(IntArrayToByteArray(leds.screenPos));
41          //leds.screenPos sarebbe il vettore contenente le 10
    coordinate di pixel dei 5 LED
42
43      }
```

Ogni LED è descritto da due coordinate, che corrispondono ai valori dei pixel su cui viene visualizzato tale LED. Infatti, le coordinate $(pixel_x, pixel_y)$ di ciascun LED, non sono altro che la proiezione del LED sull'immagine (considerata come una matrice di pixel) visualizzata dalla telecamera del Chaser. Per ottenere queste coordinate ci si serve della funzione della telecamera WorldToScreenPoint (Listing 4.5), che riceve in ingresso un Vector3 si coordinate spaziali dell'oggetto, e restituisce un Vector3, dove le prime 2 coordinate corrispondono a $(pixel_x, pixel_y)$ visti in precedenza, e il terzo valore rappresenta la distanza tra la telecamera e l'oggetto (in world units). Ai fini di questo lavoro non è necessario il terzo valore. Questo script si occuperà di preparare i dati da inviare a Simulink, e l'invio stesso di questi ultimi.

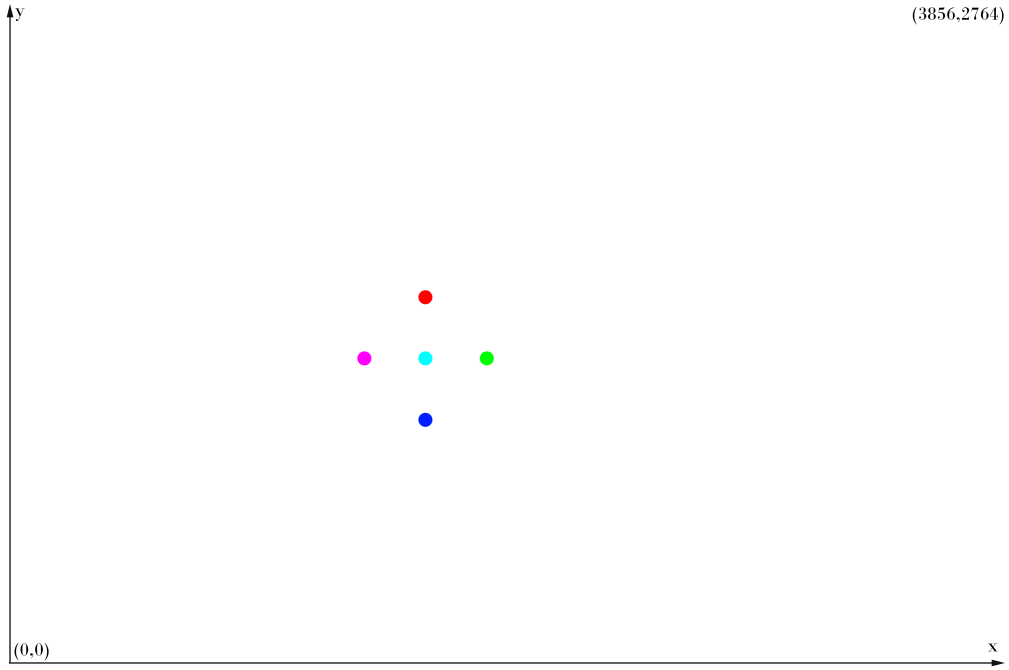
Listing 4.5: Funzione WorldToScreenPoint

```
1
2  using UnityEngine;
3  using System.Collections;
4
5  public class Esempio : MonoBehaviour
6  {
7      public Transform point;
8      private Camera cam;
9
10     void Start()
11     {
12         cam = GetComponent<Camera>();
13     }
14
15     void Update()
16     {
17         Vector3 screenPos = cam.WorldToScreenPoint(point.position);
18
19 //I valori screenPos.x e screenPos.y rappresentano le coordinate del
       pixel, mentre screenPos.z è la distanza a cui si trova, che ai
       fini del progetto non è rilevante
20
21         Debug.Log("point è posizionato nel pixel " + screenPos.x
22         + ", " + screenPos.y + " a una distanza di " + screenPos.z);
23     }
24 }
```

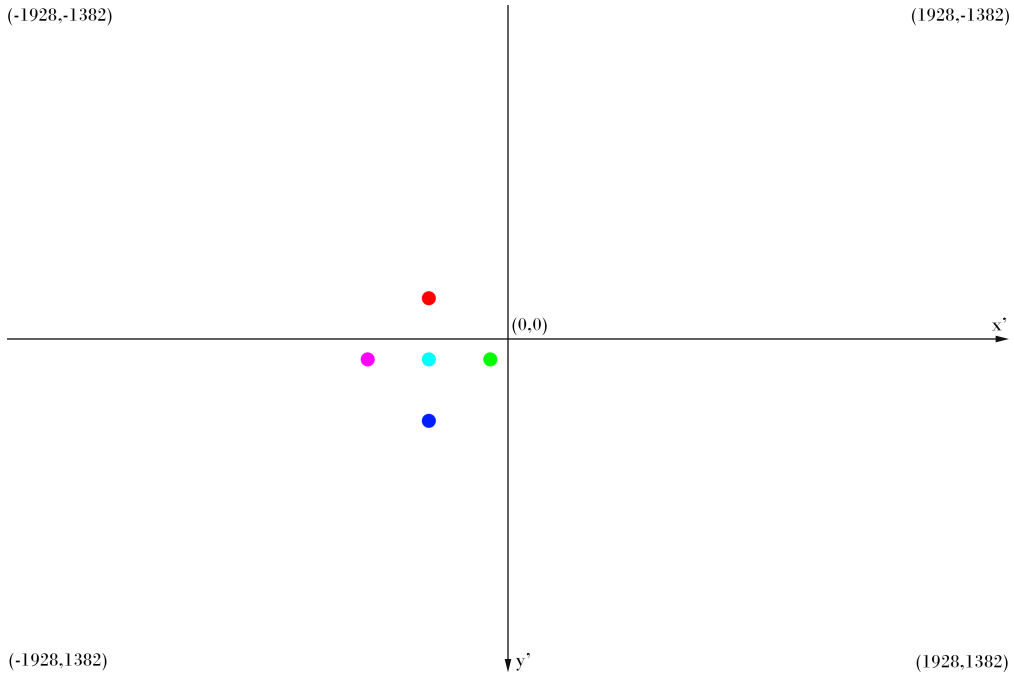
Una volta ottenuto questo valore, è necessario effettuare un cambio di sistema di riferimento per questo sistema di individuazione dei LED. La funzione *WorldToScreenPoint* individua i pixel su una matrice che considera il pixel in basso a sinistra come $(0, 0)$, fino ad arrivare a quello in alto a destra che ha coordinate $(resHeight, resWidth)$, che nel nostro caso corrispondono a $(3856, 2764)$ (Figura 4.2a). Ai fini della simulazione però, viene considerato il pixel $(0, 0)$ quello centrale. Inoltre, la numerazione del secondo valore dei pixel è opposta: i valori sono crescenti dall'alto verso il basso. Il primo valore invece è crescente da sinistra verso destra, per cui non è necessario modificarlo (Figura 4.2b).

$$\begin{cases} x' = x - \frac{resHeight}{2} = x - 1928 \\ y' = -(y - \frac{resWidth}{2}) = -(y - 1382) \end{cases} \tag{4.1}$$

Per scrivere quindi i pixel nel sistema di riferimento corretto è necessario effettuare una traslazione del sistema di riferimento, e successivamente applicare un segno negativo alla prima coordinata, come rappresentato nell'Equazione 4.1. Infine, questi valori vengono inseriti in un vettore di numeri interi (integer), e inviati secondo il Listing 4.4.



(a) Sistema di riferimento pixel su Unity



(b) Sistema di riferimento pixel su Simulink

Figura 4.2: Sistemi di riferimento dei pixel

4.3 UDP Send Simulink

Quando si tratta di spedire dati da Simulink a Unity, si deve prima di tutto andare ad identificare i dati che sarà necessario inviare. In questo caso, si tratta delle coordinate di posizione relative del Chaser rispetto al Target, e del suo orientamento. Queste coordinate sono entrambe espresse da triplette di valori, che per comodità verranno impacchettate nella stessa stringa da inviare. Successivamente si occuperà Unity di separare i singoli valori e applicarli nel modo corretto.

La prima cosa da fare con ciascuna tripletta è quella di separare i singoli valori, per convertirli in stringhe. La scelta di utilizzare delle stringhe è dovuta al fatto che l'implementazione su Simulink di un sistema di invio dati attraverso UDP risultava più semplice da incorporare al sistema. Ogni singolo valore, viene poi combinato nuovamente insieme agli altri, creando una stringa di 6 valori separati da "," (virgola) come mostrato in Figura 4.3. Le prime tre coordinate corrispondono alla posizione, mentre le ultime tre corrispondono agli angoli di orientamento.

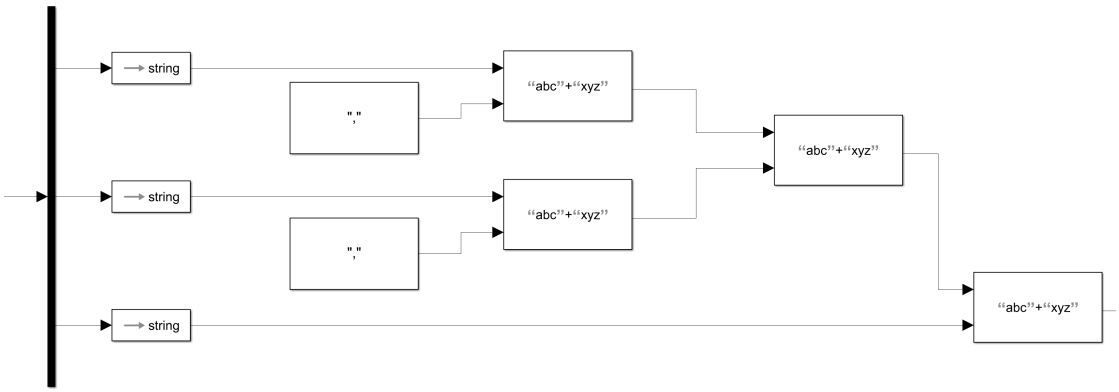


Figura 4.3: Composizione stringa di dati su Simulink

Una volta ultimata la costruzione della stringa, è necessario convertirla in codice ASCII, per permetterne l'impacchettamento in byte. Il blocco che si occupa dell'impacchettamento è Byte Pack, dove viene specificato il formato dei dati. Per come è strutturato il codice ASCII, si utilizzeranno uint8, ovvero interi senza segno su 8 byte. L'ultimo blocco, invece, è quello di invio. All'interno vengono inseriti indirizzo IP e Port, come spiegato nella sezione 4.1 (Figura 4.4).

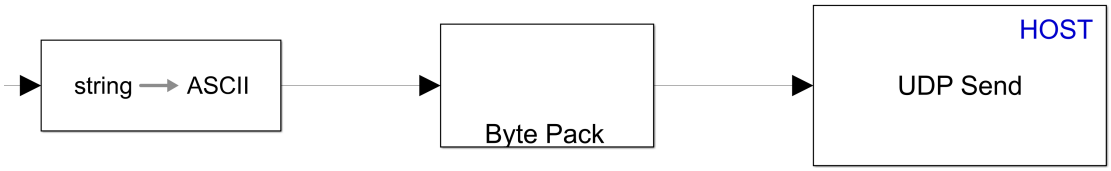


Figura 4.4: Invio di una stringa su Simulink

4.4 UDP Receive Unity Script

Quando si vuole mandare un dato a Unity, attraverso l'utilizzo dell'UDP, è fondamentale definire l'indirizzo IP e la Port a cui quella applicazione farà riferimento per ricevere i pacchetti. In questo caso, l'indirizzo rimane invariato, e la Port corrisponde a 1234, come stabilito preventivamente (Listing 4.6).

Listing 4.6: Ricezione dei pacchetti, dopo aver determinato indirizzo IP e Port

```
void Start()
{
    udpServer = new UdpClient(1234);
    t = new Thread(() => {
        while (true)
        {
            this.receiveData();
        }
    });
    t.Start();
    t.IsBackground = true;
    remoteEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 1234);
}
```

Quando un dato viene ricevuto dall'applicazione, è un vettore di byte, che necessita di essere decodificato. Ovviamente, è risaputo in quale modo è stato codificato il dato, per cui è necessario effettuare questa procedura all'inverso. In questo caso il valore che viene ricevuto è una stringa con una codifica ASCII. Dopo la decodifica, la stringa ottenuta viene separata nelle varie componenti, separate da ","(virgola), esattamente come definito nell'invio del pacchetto. Le parti del messaggio sono 6 in totale, dove le prime 3 sono coordinate di posizione, e le ultime 3 sono angoli. A questo punto, è possibile andare a posizionare le due triplette di coordinate all'interno di due vettori distinti (Listing 4.7).

Listing 4.7: Ricezione dei dati in UDPReceive e separazione

```
private void receiveData()
{
    byte[] data = udpServer.Receive(ref remoteEP);
    if (data.Length > 0)
    {
        var str = System.Text.Encoding.ASCII.GetString(data);
        Debug.Log("Received Data: " + str);
        string[] messageParts = str.Split(',');

    // Position
    RelPosition[0] =
    float.Parse(messageParts[0], CultureInfo.InvariantCulture);
    RelPosition[1] =
    float.Parse(messageParts[1], CultureInfo.InvariantCulture);
    RelPosition[2] =
    float.Parse(messageParts[2], CultureInfo.InvariantCulture);

    // Orientation
    RelRotation[0] =
    float.Parse(messageParts[3], CultureInfo.InvariantCulture);
    RelRotation[1] =
    float.Parse(messageParts[4], CultureInfo.InvariantCulture);
    RelRotation[2] =
    float.Parse(messageParts[5], CultureInfo.InvariantCulture);

    }
}
```

Ottenuti i dati, ora, serve capire come andranno applicati nel modo corretto e nei corretti sistemi di riferimento. Come anticipato nella Sezione 3.1, è necessario porre una particolare attenzione su come vengono letti i dati di posizione e orientamento. Entrambe le trasformazioni si effettueranno ogni frame, per cui è necessario posizionarle all'interno della funzione Update.

La tripletta che si ottiene in seguito alla ricezione e separazione dei dati viene inserita in un vettore di 3 valori denominato RelPosition. Come visto nella Sezione 3.1, con il calcolo della matrice M (Equazione 3.3), è necessario convertire tali dati da un sistema destrorso con una particolare configurazione di assi (Simulink), a un sistema sinistrorso (Unity). Questo cambio di potrebbe eseguire in due modi. Il primo è attraverso il calcolo della matrice di conversione da un sistema all'altro, che è stata calcolata in precedenza. In questo caso però, siccome si tratta di un caso più semplice, è possibile eseguire un semplice cambio di segno e ordine alle coordinate. Infatti, ricevuta una tripletta (x, y, z), è sufficiente creare una nuova tripletta in forma $(x, -z, -y)$.

Listing 4.8: Posizione del Chaser in UDPReceive

```
transform.position = new Vector3
                    (RelPosition[0] / 10000,
                    -RelPosition[2] / 10000,
                    -RelPosition[1] / 10000);
```

NOTA: Si può notare che i valori di posizione, e successivamente anche quelli di rotazione, sono divisi per 10000. Il motivo di questa operazione è dovuto dal fatto che prima di inviare i dati da Simulink, i valori sono stati tutti moltiplicati per 10000, con il fine di evitare la scrittura del numero in forma scientifica, e rendere più veloce la lettura del dato.

Per quanto riguarda le rotazioni, invece, il problema è un po' più complesso. La tripletta che si riceve (in gradi), indica gli angoli che sono stati applicati al docking frame $\mathcal{F}_{d_c}$ del Chaser, nell'ordine *XYZ*. In particolare, una volta applicato un angolo, è necessario applicare quello successivo sul nuovo sistema di assi che si è creato in seguito alla rotazione precedente. In aggiunta a questo, ogni angolo è da considerarsi rispetto alla configurazione iniziale. Per cui se per due volte di fila si ottiene la stessa tripletta, significa che l'orientamento è rimasto invariato. Siccome Unity effettua le rotazioni in un ordine diverso, ovvero *ZXY*, è necessario trovare un modo per cambiare questa convenzione. La soluzione che è stata individuata è con l'utilizzo di tre funzioni *transform.Rotate* in successione, applicandole nell'ordine *XYZ*. È fondamentale effettuare tali trasformazioni nello spazio relativo all'oggetto *Space.Self* e non nello spazio globale. Ci si deve ricordare, inoltre, di applicare un reset dell'orientamento del satellite prima di effettuare le rotazioni, per il motivo spiegato sopra.

Listing 4.9: Orientamento del Chaser in UDPReceive

```
transform.localEulerAngles = new Vector3(0, 0, 0); //Reset

transform.Rotate(-(RelRotation[0]) / 10000,0,0,Space.Self);    //
    Rotazione su X
transform.Rotate(0, (RelRotation[2]) / 10000, 0, Space.Self); //
    Rotazione su Y
transform.Rotate(0, 0, (RelRotation[1]) / 10000, Space.Self); //
    Rotazione su Z
```

4.5 UDP Receive Simulink

Per ricevere i dati da Unity, è necessario inserire il blocco che riceve i pacchetti (UDP Receive). In seguito a questo, il vettore va scomposto nei vari valori. Siccome sappiamo che tali valori sono stati inviati nel formato integer, è sufficiente un blocco di **byte unpacking**. Questo blocco permette di definire come è costruito il pacchetto, dando modo al software di separare i valori nel modo corretto. In questo caso il nostro pacchetto di byte è composto da 11 valori integer definiti su 32 bit ciascuno (che corrispondono a 4 byte), esattamente come è stato definito nella sezione di invio dati da Unity. Il primo valore definisce il numero di valori totali, mentre gli altri 10 sono proprio le nostre coordinate (Figura 4.5).

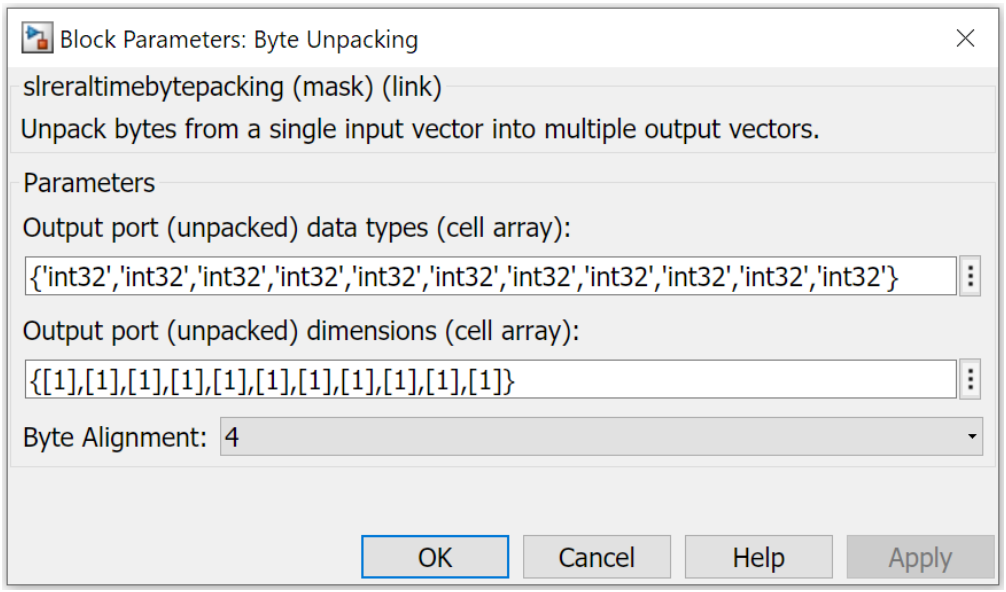


Figura 4.5: Blocco di Byte Unpack su Simulink

I dati che sono inviati da Unity sono le coordinate di pixel. In ognuna di questa coppia di coordinate viene individuato un LED del Target, per un totale di 5 LED. Le coordinate arrivano come espresso nel Capitolo 4.2 e vanno separate e reinserite in un vettore nel modo corretto. Infatti, l'algoritmo si aspetta le coordinate nell'ordine

$$(pixel_{x_1}, pixel_{y_1}, pixel_{x_2}, pixel_{y_2}, pixel_{x_3}, pixel_{y_3}, pixel_{x_4}, pixel_{y_4}, pixel_{x_5}, pixel_{y_5})$$

Si esegue quindi una scomposizione come rappresentato nella Figura 4.6.

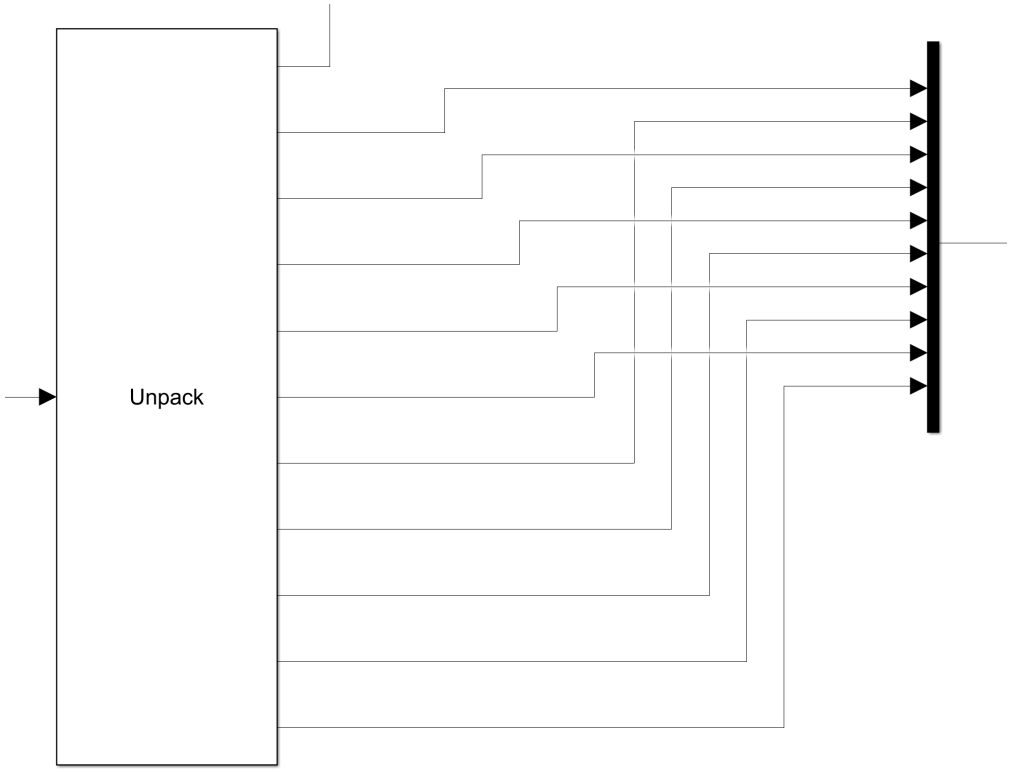


Figura 4.6: Composizione vettore di coordinate di pixel

Una volta ricevuti questi valori, è necessario moltiplicare ciascun valore per la dimensione di un pixel, valore definito nella Tabella 3.2. Infatti, ai fini della simulazione, la posizione di ogni LED sulla telecamera verrà espressa in metri.

Capitolo 5

Confronto delle simulazioni

In questo capitolo verranno presentate le varie differenze riscontrate tra le due simulazioni. In particolare la simulazione che si serve dei dati inviati da Unity, e la simulazione originale a partire da dati solamente interni a Simulink.

5.1 Condizioni Iniziali

Prima di addentrarsi nelle differenze, è importante puntualizzare che una possibile causa delle presenti divergenze può dipendere dalle condizioni iniziali. Le condizioni iniziali sono rappresentate nella Tabella 5.1.

Tabella 5.1: Valori Iniziali[2]

Rotazione	$\bar{\boldsymbol{\alpha}}^{d_c d_t} = \begin{bmatrix} 0° \\ 0° \\ 0° \end{bmatrix}$
Velocità angolare	$\bar{\boldsymbol{\omega}}_{d_t}^{d_c d_t} = \begin{bmatrix} 0° \\ 0° \\ 0° \end{bmatrix} \mathrm{s}^{-1}$
Posizione	$\bar{\boldsymbol{s}}_{d_t}^{d_c d_t} = \begin{bmatrix} -5 \\ 0 \\ 0 \end{bmatrix} \mathrm{m}$
Velocità lineare	$\bar{\dot{\boldsymbol{s}}}_{d_t}^{d_c d_t} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \mathrm{m\,s}^{-1}$

A questi valori iniziali bisogna aggiungere un errore di incertezza che il satellite può avere sulle sue condizioni iniziali, e viene generato casualmente secondo una distribuzione Gaussiana. Questa incertezza deriva dal passaggio da una fase precedente a quella trattata, che a livello teorico dovrebbe portare il CubeSat con precisione idealmente infinita nelle coordinate iniziali.

$$\Delta\boldsymbol{\alpha} \sim \mathcal{N}(0, H_\alpha) \tag{5.1a}$$
$$\Delta\boldsymbol{\omega} \sim \mathcal{N}(0, H_\omega) \tag{5.1b}$$
$$\Delta\boldsymbol{s} \sim \mathcal{N}(0, H_s) \tag{5.1c}$$
$$\Delta\dot{\boldsymbol{s}} \sim \mathcal{N}(0, H_{\dot{s}}) \tag{5.1d}$$

Le condizioni iniziali sono quindi generate come segue.

$$\boldsymbol{\alpha}_0 = \bar{\boldsymbol{\alpha}}^{d_c d_t} + \Delta\boldsymbol{\alpha} \tag{5.2a}$$
$$\boldsymbol{\omega}_0 = \bar{\boldsymbol{\omega}}_{d_t}^{d_c d_t} + \Delta\boldsymbol{\omega} \tag{5.2b}$$
$$\boldsymbol{s}_0 = \bar{\boldsymbol{s}}_{d_t}^{d_c d_t} + \Delta\boldsymbol{s} \tag{5.2c}$$
$$\dot{\boldsymbol{s}}_0 = \bar{\dot{\boldsymbol{s}}}_{d_t}^{d_c d_t} + \Delta\dot{\boldsymbol{s}} \tag{5.2d}$$

Le quattro deviazioni standard sono definite come

$$\sigma_{\boldsymbol{\alpha}} = 2^\circ \tag{5.3a}$$
$$\sigma_{\boldsymbol{\omega}} = 0.5^\circ\ \mathrm{s}^{-1} \tag{5.3b}$$
$$\sigma_{\boldsymbol{s}} = 0.1\ \mathrm{m} \tag{5.3c}$$
$$\sigma_{\dot{\boldsymbol{s}}} = 0.01\ \mathrm{m\,s}^{-1} \tag{5.3d}$$

Questo comporta che ciascuna componente dei vari valori tra cui rotazione, posizione, velocità angolare e velocità lineare del Chaser saranno casuali entro un determinato range.

5.2 Errore coordinate dei pixel

Al primo ciclo di simulazione avviene lo scambio di valori da entrambi i software nello stesso momento. Questo implica che i primi valori di LED inviati da Unity non sono quelli corrispondenti alla situazione corrispondente, siccome la stringa di valori inviati da Simulink non è ancora arrivata, e il Chaser non si è ancora posizionato.

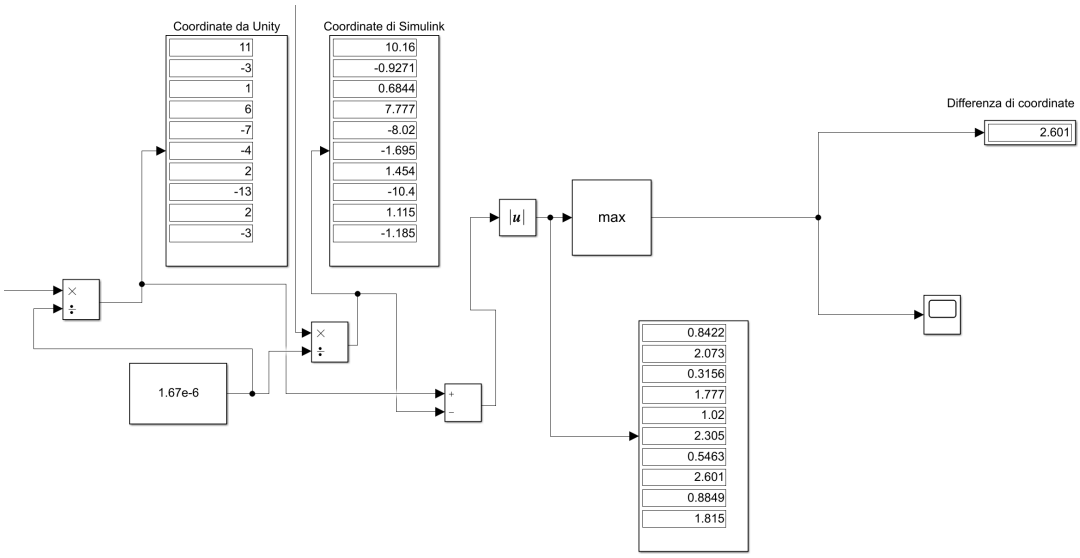


Figura 5.1: Confronto tra le coordinate e creazione del grafico

È opportuno ricordare che da Unity vengono inviate 10 coordinate che esprimono la posizione dei cinque LED sull'immagine. Ciascuna di queste coordinate è stata confrontata con la sua analoga generata dalla simulazione di Simulink, che avviene in contemporanea. I seguenti grafici rappresentano il valore massimo di differenza tra due valori di coordinate per tutto il corso della simulazione.

Il picco iniziale presenta quanto scritto prima, ovvero il mal posizionamento del Chaser al primo ciclo di simulazione. Dopo i primi 200 secondi di simulazione i valori iniziano a stabilizzarsi tenendosi sotto una differenza di 5 pixel.

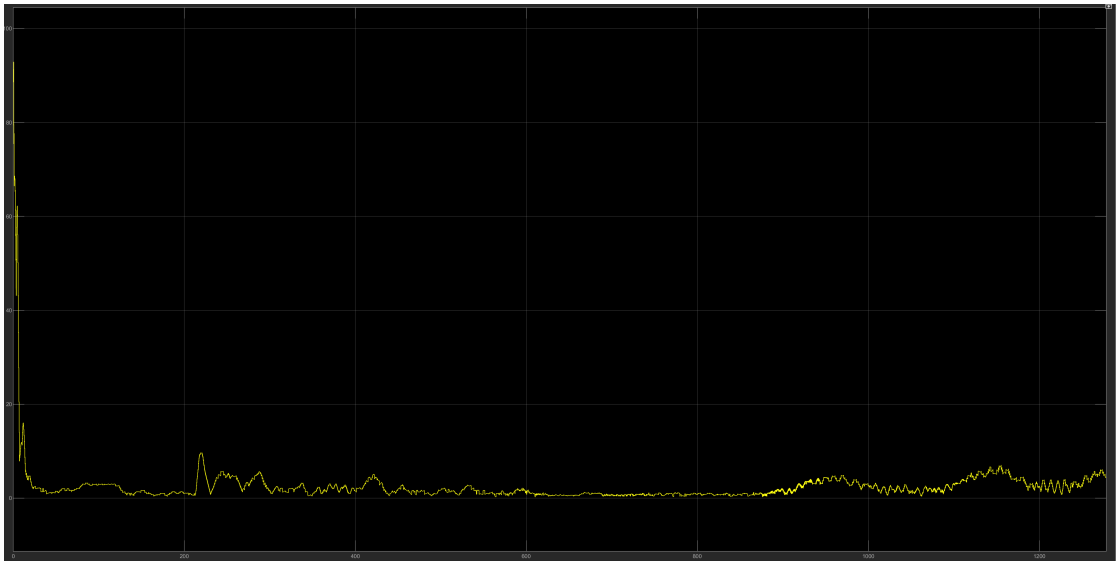


Figura 5.2: Errore massimo di coordinate nel corso della simulazione

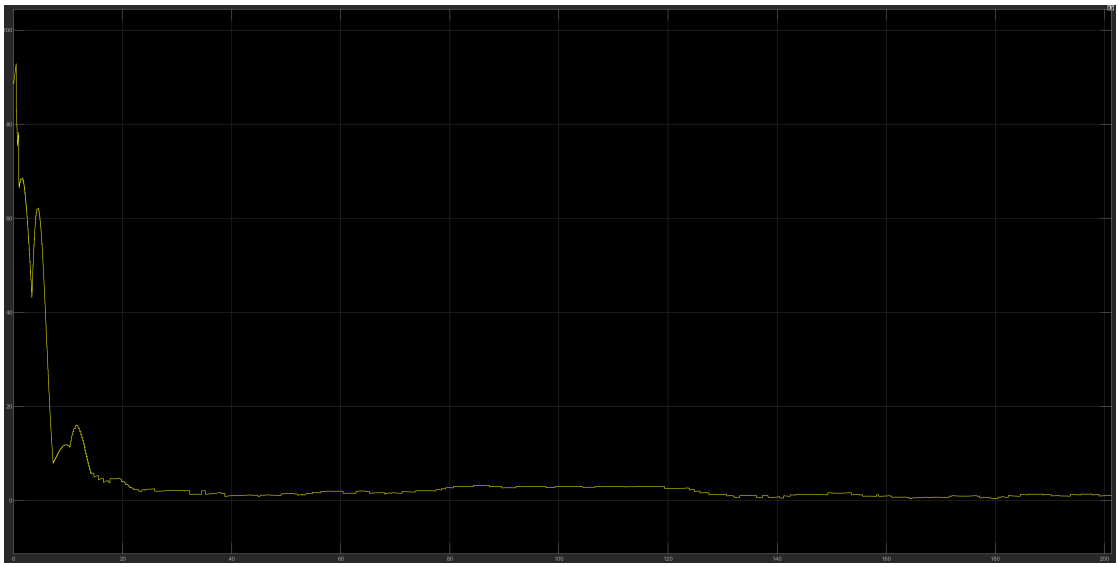


Figura 5.3: Errore massimo di coordinate per i primi 200 secondi

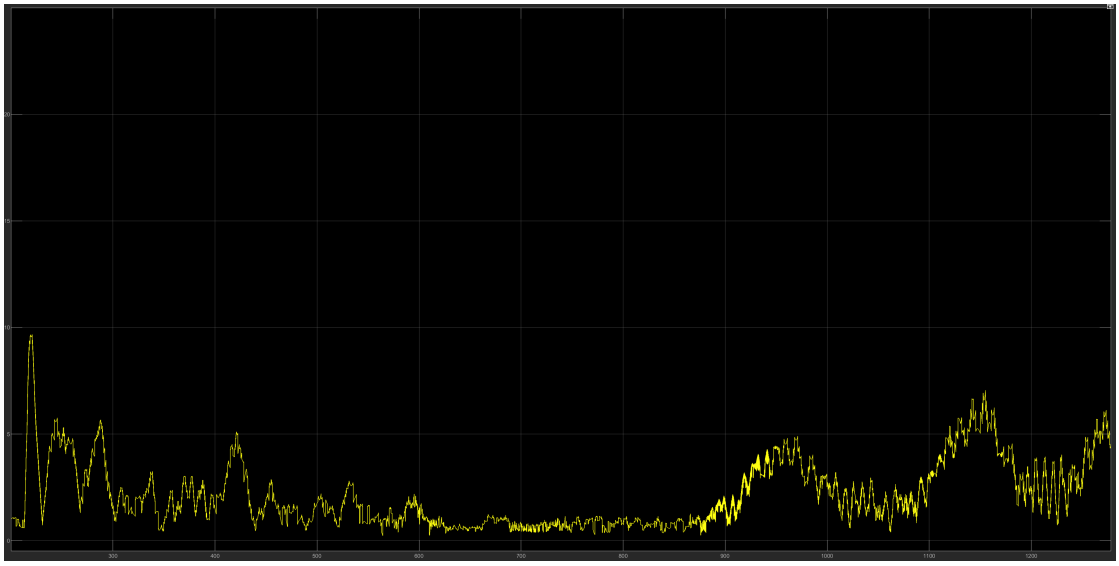


Figura 5.4: Errore massimo di coordinate oltre 200 secondi

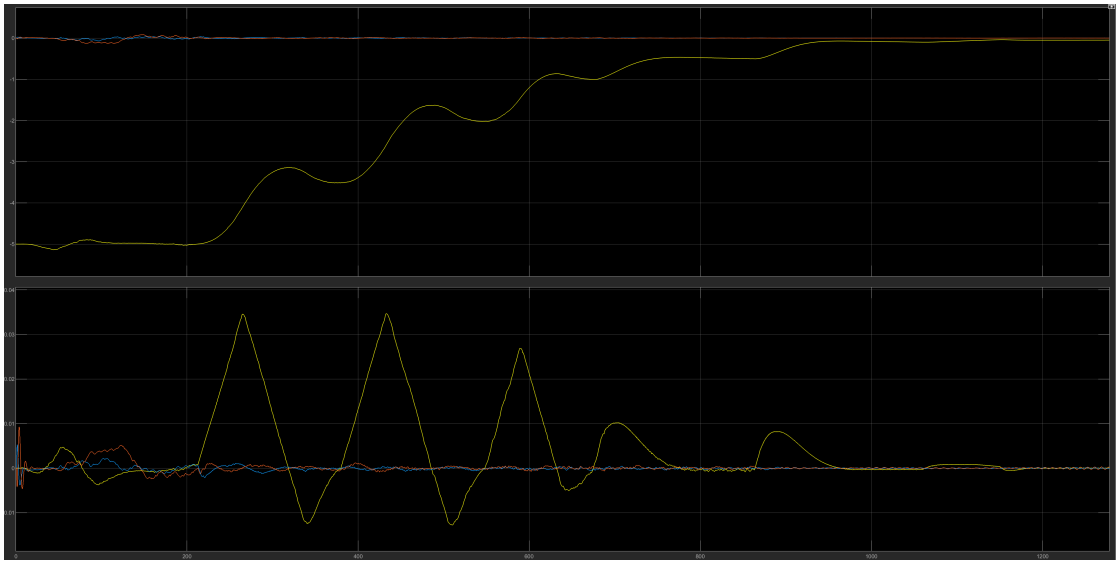


Figura 5.5: Grafici di posizione e velocità del Chaser

Confrontando i grafici presenti nella Figura 5.2 e quello in Figura 5.5, è possibile notare come il principale aumento dell'errore di pixel, fatta eccezione del primo che è già stato discusso, avviene sempre in corrispondenza di uno spostamento più veloce del Chaser. Con l'aumento della velocità infatti si nota come i picchi arrivino fino a 5 pixel di differenza. Un caso diverso invece, quello che avviene a circa 10 cm di distanza tra i due satelliti, infatti non sono più presenti dei picchi di velocità, ma la distanza ravvicinata tra la telecamera e i LED porta negli ultimi secondi una media dell'errore più alta. Nonostante questo, i picchi rimangono sempre intorno ai 5 pixel di differenza.

Considerando che la risoluzione della telecamera è 3856×2764, l'errore di 5 pixel di differenza è un valore che tra lo 0,12% e lo 0,18%. Si può quindi concludere che la simulazione è molto accurata nell'esecuzione.

Capitolo 6

Conclusioni e lavori futuri

L'obiettivo di questo lavoro è stato quello di sviluppare un simulatore che potesse ricevere, elaborare ed infine fornire i dati necessari all'algoritmo di Vision Based Navigation. Questo sistema permette a due satelliti di attaccarsi in orbita: uno dei due, che prende il nome di Chaser, che cerca di attaccarsi al secondo, chiamato Target, servendosi di una telecamera monoculare. Quest'ultima visualizzerà un pattern di LED posizionati sul secondo, ottenendo informazioni necessarie a calcolare la manovra successiva che va effettuata dal primo. È stato utilizzato il Game Engine di Unity, che ha permesso una buona versatilità progettuale e una buona resa grafica.

Come prima cosa è stata affrontata la modellazione 3D dei satelliti e dei corpi celesti. Basandosi su immagini ufficiali e modelli preesistenti si è potuto creare una ricostruzione verosimile dei CubeSat.

La fase successiva è stata quella di inserire all'interno del mondo di Unity i corpi celesti in modo che fossero visualizzati a distanze e grandezze in scala. Questo processo è stato possibile attraverso l'implementazione dei Camera Layers che, permettendo di renderizzare oggetti diversi all'interno della scena, e di essere visualizzati da telecamere diverse. Mettendo insieme questi livelli, si è generata un'immagine che simula la distanza degli oggetti cosmici. È quindi stata trattata la creazione di materiali che fornissero un effetto fotorealistico al risultato finale del simulatore, insieme a un Post Processing che ne migliorasse la resa grafica.

Il processo successivo è stato quello di mettere in comunicazione le due applicazioni, per permettere il trasferimento dei dati che compongono la principale componente del simulatore. È stata presentata la creazione di una connessione UDP per ciascuna delle due applicazioni, evidenziando quale tipologia di dati sarebbe stata trasferita. È stata poi studiata una corretta costruzione dei sistemi di riferimento per la lettura dei dati, per ciascuna delle due applicazioni.

Un confronto tra le due diverse simulazioni (quella che si serve di Unity, e

quella che avviene solo su Simulink) ha fatto emergere delle differenze probabilmente determinate dalla componente casuale delle condizioni iniziali, che implicano un periodo di assestamento necessario all'inizio della simulazione. In seguito a quest'ultimo, la simulazione raggiunge un massimo di 5 pixel di differenza nelle coordinate, con dei picchi in punti specifici. Ciò è confermato dal confronto del grafico della differenza di pixel con quelli di velocità e posizione del Chaser, si è potuta notare una correlazione tra l'aumento di "differenza" nel momento in cui il satellite accelerava per effettuare la sua manovra. Non appena iniziato il rallentamento, la differenza dei pixel tornava ad abbassarsi. Anche al di sotto dei 10 cm di distanza tra i due satelliti, il valore di differenza oscilla molto, ma comunque sempre entro i 5 pixel.

I passaggi più cruciali sono stati sicuramente quelli riguardanti il trasferimento dei dati attraverso una connessione UDP: all'inizio si sono dovuti superare dei problemi di interfaccia, che portavano a un fallimento della simulazione.

Sicuramente il mondo dei simulatori è ancora molto variegato, ma grazie a questa progettazione, sarebbe possibile aggiungere ulteriori elementi che arricchirebbero la simulazione per renderla più fruibile, versatile e realistica.

Prima tra tutte si segnala l'aggiunta di un sistema di propulsione del Chaser, basato sulle coppie di forze generate per la manovra. Questo aggiungerebbe realismo al simulatore, ma allo stesso tempo comportando un complesso sistema di forze all'interno del Game Engine, potrebbe portare a errori maggiori di quelli riscontrati in questo lavoro. Sarebbe in questo caso necessario effettuare uno studio approfondito dei propulsori utilizzati dai CubeSat per quanto riguarda la modellazione e il posizionamento nel modello finale.

Un'altra interessante miglioria sarebbe quella di permettere un'esplorazione del simulatore all'interno di un ambiente più grande, come ad esempio quello dell'intero sistema solare, andando a posizionare i diversi pianeti con i propri sistemi orbitali. Si potrebbe giungere a generare un modello simile, con l'utilizzo di forze gravitazionali che mettono in orbita i pianeti: sicuramente fornirebbe un'ottima occasione di utilizzo di altri tipi di simulazioni.

Infine, con lo scopo di aggiungere una maggiore accuratezza grafica al simulatore, si potrebbe effettuare uno studio di shader che simuli il comportamento della luce all'interno del sistema (ad esempio con l'entrata dei raggi luminosi all'interno dell'atmosfera, con gli effetti conseguenti).

Appendice A

Cattura di immagini all'interno di Unity

È stato detto nell'introduzione di questa tesi, che non è stato effettuato un vero e proprio Image Processing, ma viene analizzata la posizione dei LED all'interno dell'immagine. Durante questo lavoro di tesi è stato comunque sviluppato un metodo di cattura immagini all'interno del mondo di Unity. Il seguente script deve essere inserito come componente di una telecamera o comunque di un oggetto sottostante ad essa.

Listing A.1: Cattura dell'immagine di una telecamera all'interno di Unity

```
1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [RequireComponent(typeof(Camera))]
6  public class snapshotCamera : MonoBehaviour
7  {
8      Camera snapCam;
9      int resHeight = 2764;
10     int resWidth = 3856;
11
12     void Awake()
13     {
14         snapCam = GetComponent<Camera>();
15         if(snapCam.targetTexture == null) {
16             snapCam.targetTexture = new RenderTexture(resWidth,
       resHeight, 24);
17         }
18         else
19         {
20             resWidth = snapCam.targetTexture.width;
```

```
21              resHeight = snapCam.targetTexture.height;
22          }
23          snapCam.gameObject.SetActive(false);
24      }
25
26      public void CallTakeSnapshot() //Accensione della telecamera
27      {
28          snapCam.gameObject.SetActive(true);
29      }
30
31      void LateUpdate() //Verifica accensione telecamera
32      {
33          if (snapCam.gameObject.activeInHierarchy)
34          {
35              //Se attiva, scatto la foto
36              Texture2D snapshot = new Texture2D(resWidth, resHeight,
    TextureFormat.RGB24, false);
37              snapCam.Render();
38              RenderTexture.active = snapCam.targetTexture;
39              //Salva i pixel che vede la telecamera, determinando in
    che modo farlo
40              //In questo caso si parte dal primo in basso a sinistra
    fino a quello in alto a destra
41              snapshot.ReadPixels(new Rect(0, 0, resWidth, resHeight),
    0, 0);
42              byte[] bytes = snapshot.EncodeToPNG();
43              string fileName = SnapshotName();
44              System.IO.File.WriteAllBytes(fileName, bytes);
45              Debug.Log("Snapshot taken!\n");
46              snapCam.gameObject.SetActive(false);
47          }
48      }
49
50      string SnapshotName() //Nome dell'immagine
51      {
52          return string.Format("{0}/Snapshots/snap_{1}x{2}_{3}.png",//
    Percorso e nome autogenerato
53              Application.dataPath, //Percorso file di dove salvare l'
    immagine
54              resWidth, resHeight,
55              System.DateTime.Now.ToString("ddd-dd-MMM-yyyy_HH-mm-ss"))
    ;
56      }
57  }
```

Bibliografia

[1] Bruce Yost et al. *State-of-the-Art Small Spacecraft Technology*. NASA/TP—20210021263. Moffett Field, CA: Ames Research Center, ott. 2021.

[2] Giacomo Ichino. «Performance Evaluation of a Vision Based Navigation Algorithm for CubeSat Docking Missions». In: (dic. 2021), p. 114.

[3] D.G. Lowe. «Object recognition from local scale-invariant features». In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, pp. 1150–1157.

[4] *ESA - Technology CubeSats*. Accessed 15-DEC-2021. URL: https://www.esa.int/Enabling_Support/Space_Engineering_Technology/Technology_CubeSats.

[5] Camille Sébastien Pirat. «Guidance, Navigation and Control for Autonomous Rendezvous and Docking of Nano-Satellites». In: (2018), p. 402. DOI: 10.5075/epfl-thesis-8729. URL: http://infoscience.epfl.ch/record/261367.

[6] *Blender*. Accessed 15-DEC-2021. URL: https://www.blender.org/.

[7] Robert Bedington, Xueliang Bai, Edward Truong-Cao, Yue Tan, Kadir Durak, Aitor Villar Zafra, James Grieve, Daniel Oi e Alexander Ling. «Nanosatellite experiments to enable future space-based QKD mission». In: *EPJ Quantum Technology* 3 (dic. 2016), p. 12. DOI: 10.1140/epjqt/s40507-016-0051-7.

[8] W. Fehse. *Automated rendezvous and docking of spacecraft*. Cambridge University Press, 2003.

[9] Gabriel Popescu. «Pixel geolocation algorithm for satellite scanner data». In: giu. 2014.

[10] Nicoletta Bloise, Elisa Capello, Matteo Dentis e Elisabetta Punta. «Obstacle Avoidance with Potential Field Applied to a Rendezvous Maneuver». In: *Applied Sciences* 7 (ott. 2017), p. 1042.

[11] R. Brinkmann. *The Art and Science of Digital Compositing*. Morgan Kaufmann, 2008. Cap. 2.

[12] NASA. *Structures, Materials, and Mechanisms*. Accessed 02-MAR-2022. URL: https://www.nasa.gov/smallsat-institute/sst-soa/structures-materials-and-mechanisms.

[13] Oliver Montenbruck e Eberhard Gill. *Satellite Orbits: Models, Methods and Applications*. Springer Berlin Heidelberg, 2000.

[14] J.L. Heilbron. *Platonic solid*. A cura di Encyclopedia Britannica. Accessed 7 March 2022. URL: https://www.britannica.com/science/Platonic-solid.

[15] NASA. *Solar System Exploration: Planets: Sun: Facts & Figures*. Accessed 7 March 2022. URL: https://web.archive.org/web/20080102034758/http://solarsystem.nasa.gov/planets/profile.cfm?Object=Sun&Display=Facts&System=Metric.

[16] Kim et al. *Color Temperature Conversion System and Method Using the Same*. US patent 7024034. Accessed 02-MAR-2022. URL: https://patents.google.com/patent/US7024034.

[17] Wikipedia contributors. *Planckian locus — Wikipedia, The Free Encyclopedia*. [Online; accessed 8-March-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Planckian_locus&oldid=1066226328.

[18] Ahmed Elnaggar. «TCP Vs. UDP». In: (ott. 2015). DOI: 10.13140/RG.2.1.4244.1688.