

Politecnico di Torino

Politecnico di Torino

Corso di Laurea Magistrale in Ingegneria Informatica

A.a. 2025/2026

Sessione di laurea Settembre 2026

Evoluzione e Sviluppo di una Piattaforma Web per Simulazioni Tattiche Computer-Generated Forces (CGF): dall'Analisi dei Requisiti a un'Architettura React ad Alte Prestazioni

Relatori:

Prof. Luca Ardito

Dott. Gianluca Gagliati

Candidato:

Simone Fenech

Ringraziamenti

Desidero ringraziare i miei genitori per il supporto emotivo ed economico che mi ha permesso di raggiungere i miei obiettivi: mia mamma per esserci stata durante i miei periodi più bui, anche se ha richiesto un gran sacrificio da parte sua, e mio papà per avermi aiutato a superare le difficoltà.

Grazie a mia nonna per la costante presenza anche a km di distanza.

Grazie a tutta la mia famiglia per ogni parola che mi ha accompagnato in questa parte della mia vita. E grazie a Nala per tutti i «miao» *di questi anni.*

Ringrazio il Relatore, il Prof. Luca Ardito, e il mio supervisore aziendale, Gianluca Gagliati, per l'opportunità che mi hanno concesso e per aver reso possibile questo progetto.

A tutti voi, grazie di cuore.

Indice

Elenco delle figure IX

1 Introduzione e Contesto 1

1.1 Contesto aziendale 1

1.2 Il dominio del Modeling & Simulation (M&S) 2

1.2.1 Il M&S nel settore della Difesa 2

1.2.2 Il paradigma LVC (Live, Virtual, Constructive) 2

1.3 Computer-Generated Forces (CGF) 4

1.3.1 Evoluzione storica dei sistemi CGF 5

1.3.2 Architettura e modellazione del comportamento 7

1.3.3 Livelli di astrazione: entità e modelli aggregati 7

1.3.4 Sfide tecnologiche e architetture Web-Based per i CGF . . . 8

1.4 Obiettivi del progetto e motivazioni della tesi 9

1.5 Metodologia di Lavoro 9

1.6 Struttura della Tesi 10

2 Background Tecnologico e Stato dell'Arte 12

2.1 Protocolli di Simulazione Distribuita 12

2.1.1 Distributed Interactive Simulation (DIS) 13

2.1.2 High Level Architecture (HLA) 15

2.1.3 Stato dell'Arte e Strategie di Interoperabilità Ibrida 16

2.2 Framework e Motori di simulazione 17
2.2.1 MIXR (Mixed Reality Simulation Platform) 17
2.2.2 JSBSim Flight Dynamics Model 21
2.2.3 Integrazione di JSBSim in MIXR 22
2.2.4 Modellazione Dichiarativa delle Entità tramite EDL 23
2.3 Architetture di Intelligenza Artificiale per CGF: Unified Behavior Framework (UBF) 24
2.3.1 Limiti degli approcci tradizionali e dei Behavior Trees 24
2.3.2 Principi di Progettazione dell'UBF 24
2.4 Tecnologie Web e Stack Frontend 27
2.4.1 Vite: Build Tool e Development Server 27
2.4.2 React: Libreria per la Costruzione di Interfacce Utente . . . 28
2.4.3 TypeScript: Sistema di Tipi Statici per JavaScript 28
2.4.4 CesiumJS: Rendering Geospaziale 3D su WebGL 29
2.4.5 Tailwind CSS: Stilizzazione 30
2.4.6 shadcn/ui: Componenti Interattivi 31
2.4.7 Protocollo WebSocket: Comunicazione Bidirezionale in Tempo Reale 32

3 Analisi dei Requisiti e Progettazione del Sistema 33
3.1 Ingegneria dei Requisiti e Modellazione del Sistema 33
3.1.1 Attori del Sistema 33
3.1.2 Requisiti Funzionali e Diagramma dei Package 34
3.1.3 Analisi dei Sottosistemi di Rilievo 36
3.1.4 Requisiti Non Funzionali e Metriche di Riferimento 40
3.2 Design dell'Esperienza Utente (UX) e Transizione Architetturale . . 41
3.2.1 I Limiti dell'Architettura di Partenza (AS-IS) 42
3.2.2 La Soluzione Progettuale (TO-BE) 43
3.3 High-Level Design (Progettazione Architetturale) 44

3.3.1 Disaccoppiamento tra Presentation Layer e Rendering Geospaziale . . . 44
3.3.2 Architettura a Strati del Client Web . . . 45
3.3.3 Modello Architetturale di Rete Ibrido (WebSocket + REST) 46
3.4 Pipeline di Configurazione e Orchestrazione degli Scenari . . . 46
3.4.1 Il formato dei file di scenario (.epp ed .edl) e Automazione della Build . . . 47
3.4.2 Orchestrazione Asincrona e Inizializzazione dello Scenario . . 47
3.4.3 Editing Dinamico e Pattern Pub/Sub (Creazione Waypoint) 49

4 Implementazione del Presentation e Spatial Layer 52
4.1 Architettura di Comunicazione nel Client React . . . 52
4.1.1 Gestione della Telemetria ad Alta Frequenza (WebSocket) . 52
4.1.2 Il Manager API e il Pattern Publisher/Subscriber per il Traffico Transazionale . . . 53
4.2 Definizione e Serializzazione dei Payload Dati (DTO) . . . 54
4.2.1 Contratti Dati per la Telemetria Continua . . . 54
4.2.2 DTO e Serializzazione per le Transazioni REST . . . 55
4.3 Resilienza di Rete e Gestione del Ciclo di Vita delle Entità . . . 56
4.3.1 Tolleranza ai Guasti e Riconnessione Automatica (Retry Logic) 57
4.3.2 Garbage Collection e Tracciamento Obsoleto (Stale Tracks) . 58
4.4 Sincronizzazione Temporale e Gestione dello Stato di Simulazione . 59
4.4.1 Orchestrazione dei Comandi di Trasporto (REST) . . . 59
4.4.2 Sincronizzazione Distribuita del Clock (WebSocket) . . . 59
4.5 Gestione del Carico Computazionale e Prevenzione dei Memory Leak 60
4.5.1 Bypass dello Stato Storico nel Virtual DOM . . . 60
4.5.2 Garbage Collection di Livello Applicativo (Entity Cleanup) . 61
4.5.3 Deallocazione delle Risorse GPU (WebGL) . . . 61

5 Sviluppo dell'Interfaccia Grafica (Front End) 62

5.1 Architettura Modulare React: Struttura dei componenti, gestione dello stato per dati dinamici ad alta frequenza 62

5.1.1 Transizione UI/UX e Componentizzazione (Tailwind CSS e Shadcn/UI) 62

5.1.2 Gestione Ibrida dello Stato (Redux e Context API) 63

5.1.3 Orchestrazione dei Menu Contestuali 64

5.1.4 Architettura Multi-Viewport (MultiViewManager) 64

5.2 Integrazione Mappe: Implementazione di Cesium per la visualizzazione delle mappe satellitari e dei dati vettoriali 65

5.2.1 Il Pattern Strutturale "MapAdapter" e la Strumentazione HUD 65

5.2.2 Simbologia Tattica e Vettori di Moto Adattivi 66

5.2.3 L'Editor Spaziale e l'Iniezione delle Entità (Point & Click Spawner) 67

5.3 Gestione delle Sfide Tecniche: Risoluzione di problemi legati al rendering e alle performance della GUI 68

5.3.1 Rendering On-Demand e Iniezione Diretta dei Dati (Bypass del Virtual DOM) 68

5.3.2 Memoization e Caching dell'Elaborazione Vettoriale (Simbologia APP-6) 69

5.3.3 Ottimizzazione del Motore di Filtraggio (Visibility Toggling) 70

6 Testing e Validazione 71

6.1 Specifica e Test Design (Matrice di Validazione) 71

6.1.1 Validazione del Ciclo di Vita e Sincronizzazione (Station) . . 71

6.1.2 Validazione Cinematica e Rispetto dei Vincoli Fisici (OPD) 72

6.1.3 Navigazione Logica e Gestione Dinamica del Tempo 73

6.1.4 Inseguimento Tattico e Risoluzione dei Conflitti Multipli . . 73

6.1.5 Logiche di Ingaggio e Controllo dei Sistemi d'Arma 74

6.1.6 Evoluzione Architetturale e Sviluppi Futuri 74

6.2 Automated Regression & Integration Testing (API Backend) 74
6.2.1 Architettura del framework e Dynamic Cleanup 75
6.2.2 Validazione del Ciclo di Vita e Limiti di Copertura 75
6.2.3 Validazione CRUD e coerenza topologica (Move, not Clone) 76
6.2.4 Anomalie di Routing e ciclo di vita delle API 76
6.3 Validazione Automatizzata e Strategie di Robustezza del Frontend . 77
6.3.1 Selezione del Framework e Requisiti Tecnologici 77
6.3.2 Superamento dei Limiti del Rendering WebGL 77
6.3.3 Architettura della Suite di Test E2E 78
6.3.4 Validazione Client-Side e Prevenzione degli Errori 78

7 Conclusioni e Sviluppi Futuri 80
7.1 Valutazione degli Obiettivi Raggiunti 80
7.1.1 Dall'Analisi dei Requisiti all'Architettura Realizzata 80
7.1.2 Robustezza e Validazione: Risultati Conseguiti e Limiti . . . 81
7.2 Applicabilità Industriale dell'Architettura Proposta 82
7.3 Sviluppi Futuri 83
7.3.1 Consolidamento del Testing Automatizzato 83
7.3.2 Estensioni Funzionali e Interoperabilità 83
7.3.3 Verso la Commercializzazione della Piattaforma 83
7.4 Considerazioni Conclusive 84

A Diagrammi degli altri Casi d'Uso 85

Bibliografia 89

Elenco delle figure

1.1 Operatori dell'Esercito Italiano strumentati con i sensori del Sistema Integrato per l'Addestramento Terrestre (SIAT) [3]. 3

1.2 Operatori dell'Esercito Israeliano nell'abitacolo interno di un velivolo da addestramento avanzato M-346 Master [4]. 3

1.3 Schermata di uno scenario di combattimento digitale generato durante una Command Post Exercise (CPX) presso il Joint Multinational Simulations Center [6]. 4

2.1 Simulatori indipendenti connessi tramite DIS [14]. 13

2.2 Struttura base dell'HLA [15] 15

2.3 Struttura delle simulazioni [17]. 20

2.4 MIXR (OpenEaagles) Simulation Framework [18]. 23

2.5 UML Diagram per UBF [19]. 25

2.6 Flusso di esecuzione a runtime [19]. 26

3.1 Diagramma UML dei Package. Illustra il perimetro dell'applicazione web e le interazioni di alto livello tra l'operatore, il motore C++ e i futuri nodi di rete DIS/HLA con i vari moduli funzionali. 35

3.2 Diagramma dei Casi d'Uso relativo al package "Gestione e ciclo di vita di un'entità". 37

3.3 Diagramma dei Casi d'Uso relativo al package "Logica Tattica e Scripting". 39

3.4 Rappresentazione dell'architettura a strati del client web e delle interfacce di comunicazione con il backend di simulazione. 45

3.5 Diagramma di Sequenza per l'orchestrazione e l'inizializzazione dello scenario. 48

3.6 Diagramma di Sequenza ibrido (REST + Pub/Sub) per l'aggiunta dinamica di un Waypoint. 50

4.1 Diagramma delle classi che illustra il boundary di serializzazione e il mapping dei DTO tra il Presentation Layer (TypeScript) e il Logic Layer (C++). 56

4.2 Diagramma della macchina a stati. 57

5.1 Pannelli in overlay sulla mappa WebGL. 63

5.2 Griglia Multi-View con 2 canvas aperti simultaneamente. 65

5.3 Mappa satellitare in light mode con bussola e scala in basso a sinistra. 66

5.4 Fase di spawn di un'entità. 68

5.5 Esempio di player attivo e player distrutto. 70

A.1 Diagramma dei Casi d'Uso relativo al package "Controllo del Tempo e della Simulazione". 86

A.2 Diagramma dei Casi d'Uso relativo al package "Interoperabilità, Rete e Osservabilità". 87

A.3 Diagramma dei Casi d'Uso relativo al package "Gestione dello scenario e dell'ambiente". 88

Capitolo 1

Introduzione e Contesto

1.1 Contesto aziendale

Il presente lavoro nasce nell'ambito di un tirocinio della durata di sei mesi, svolto presso TXT E-Tech, azienda facente parte del gruppo italiano TXT Group, costituita nel 1989 e specializzata nei comparti della difesa e dell'aerospazio. Più nel dettaglio, le attività si sono svolte all'interno del laboratorio di ricerca Futura, di recente istituzione, con sede a Torino. Inaugurata nel marzo 2026, questa struttura è il risultato di una collaborazione strategica avviata tra l'azienda e il Politecnico di Torino nell'ambito del progetto ISM4Italy, infrastruttura tecnologica per l'innovazione sostenuta dai finanziamenti del PNRR. Il Futura Innovation Lab indirizza le proprie risorse verso tre aree sperimentali caratterizzate da un elevato grado di complessità, fra cui si distingue il SimLab, orientato al paradigma del Modelling & Simulation-as-a-Service e all'integrazione dell'intelligenza artificiale ai fini dell'ottimizzazione delle prestazioni.

Questa tesi si colloca precisamente in questo perimetro di ricerca applicata. L'esigenza aziendale di rinnovare gli strumenti software esistenti e di ampliare le architetture proprietarie ha infatti costituito un caso di studio concreto per la progettazione e lo sviluppo di una nuova piattaforma web nel dominio delle simulazioni tattiche e delle Computer-Generated Forces (CGF).

1.2 Il dominio del Modeling & Simulation (M&S)

1.2.1 Il M&S nel settore della Difesa

Con il termine modello si intende la rappresentazione fisica, matematica o logica di un sistema, mentre la simulazione ne costituisce l'implementazione nel tempo, conferendo dinamicità all'oggetto di studio al fine di anticiparne e analizzarne il comportamento. Il Modeling & Simulation (M&S) consente di replicare virtualmente prodotti e processi all'interno di un ambiente appositamente predisposto e pienamente operativo. Diversi sono i fattori che motivano il ricorso sistematico a queste architetture: si osserva una riduzione dei costi, una drastica contrazione delle tempistiche di programmazione e un netto miglioramento della qualità del prodotto finale, mentre l'addestramento può svolgersi in condizioni di piena sicurezza operativa [1]. Nell'ambito della Difesa, il M&S si configura come strumento essenziale per la gestione dei sistemi d'arma dotati di elevata complessità tecnologica, con un impiego che si estende lungo l'intero spettro operativo, dalla fase concettuale fino all'addestramento pratico.

1.2.2 Il paradigma LVC (Live, Virtual, Constructive)

Secondo quanto stabilito dalle direttive NATO, le simulazioni impiegate in ambito militare vengono ripartite in tre categorie principali, complessivamente note come paradigma LVC [2], differenziate in base al ricorso a operatori e apparati reali oppure simulati:

1. **Live:** coinvolge persone reali operanti con sistemi altrettanto reali; trova impiego nell'addestramento fisico sul campo, ad esempio all'interno di poligoni o aree dedicate all'addestramento. In questo contesto, le truppe reali si confrontano avvalendosi di veicoli e armamenti effettivi, sui quali sono installati sensori laser innocui, capaci di riprodurre il comportamento balistico delle armi e di registrare, con riferimento geografico, i colpi andati a segno. L'insieme dei dati relativi agli scontri, alle posizioni e alle comunicazioni viene quindi raccolto da una Direzione di Esercitazione (EXCON), incaricata di valutare le prestazioni delle unità.



Figura 1.1: Operatori dell'Esercito Italiano strumentati con i sensori del Sistema Integrato per l'Addestramento Terrestre (SIAT) [3].

2. **Virtual:** coinvolge invece persone reali che operano per mezzo di equipaggiamenti simulati; è concepita per l'addestramento tecnico e procedurale di singoli individui o di interi equipaggi, all'interno di uno scenario generato al computer. L'esempio più ricorrente è rappresentato dai simulatori di volo o di guida, nei quali l'operatore fisico si interfaccia con una replica della strumentazione di bordo, mentre il software riproduce fedelmente sia la risposta del mezzo sia l'ambiente operativo circostante.



Figura 1.2: Operatori dell'Esercito Israeliano nell'abitacolo interno di un velivolo da addestramento avanzato M-346 Master [4].

3. **Constructive:** sono coinvolte persone simulate che operano con sistemi a loro volta simulati. In questo caso, truppe fisiche ed equipaggiamenti sul campo non esistono, ma le decisioni restano comunque affidate a operatori reali. Si tratta della modalità privilegiata per l'addestramento dei Posti Comando, nell'ambito della quale fazioni umane contrapposte impartiscono ordini effettivi alle rispettive unità virtuali; il software di simulazione elabora tali comandi, calcolando l'esito degli scontri con una tempistica prossima al tempo reale. L'intera esercitazione risulta supervisionata dall'EXCON, che agisce come "motore" logico dell'attività: oltre a raccogliere i dati destinati alla revisione critica conclusiva (After Action Review), tale organo gestisce le variabili di scenario introducendo enti superiori simulati (Higher Controller) e attori neutrali (White Cell), assicurando in tal modo che l'esercitazione si svolga entro un contesto tattico e politico verosimile [5].



Figura 1.3: Schermata di uno scenario di combattimento digitale generato durante una Command Post Exercise (CPX) presso il Joint Multinational Simulations Center [6].

1.3 Computer-Generated Forces (CGF)

I Computer-Generated Forces (CGF), noti anche, nella letteratura precedente, come Semi-Automated Forces (SAF), costituiscono simulazioni software in grado

di rappresentare forze militari (soldati, veicoli, velivoli e navi) all'interno di un ambiente virtuale. Equipaggiati con modelli di comportamento e algoritmi decisionali, questi sistemi mostrano un certo grado di autonomia, che consente loro di muoversi, reagire e assumere decisioni di natura tattica, pur necessitando generalmente, in fase di esecuzione, di un adeguato livello di supervisione e controllo da parte di un operatore umano.

L'impiego dei CGF in ambito militare persegue principalmente l'obiettivo di popolare i campi di battaglia "sintetici" (Synthetic Environments), riducendo così il numero di player umani richiesti e permettendo di condurre esercitazioni su larga scala con maggiore efficienza ed economicità [7].

1.3.1 Evoluzione storica dei sistemi CGF

Le origini del dominio dei CGF si collocano nei primi anni Ottanta, quando la DARPA avviò il programma SIMNET (SIMulator NETworking), concepito come dimostrazione di fattibilità della simulazione distribuita in tempo reale, finalizzata all'addestramento collettivo di equipaggi di carri armati e di veicoli da combattimento.

Nell'ambito di SIMNET, i simulatori delle singole piattaforme (tra cui l'M1 Abrams e l'M2 Bradley) venivano messi in rete per consentire esercitazioni di comando, controllo e manovra su un campo di battaglia virtuale condiviso: un approccio che anticipava concetti oggi ben noti nei sistemi di realtà virtuale e nei giochi multiplayer online. L'obiettivo perseguito era la riduzione dei costi e dei rischi connessi all'addestramento con equipaggiamento reale, senza tuttavia rinunciare a un elevato grado di realismo tattico.

L'esperienza operativa accumulata con SIMNET fece presto emergere l'esigenza di disporre di forze avversarie e di supporto più numerose e dinamiche rispetto a quelle ottenibili limitandosi ad aggiungere ulteriori simulatori con equipaggio umano.

Per rispondere a questa esigenza, la DARPA sviluppò una prima generazione di simulazioni constructive volta a fornire agli utenti di SIMNET un'opposing force (OPFOR) automatizzata: nacquero in questo modo le Semi-Automated Forces (SAF), entità software gestite da un singolo operatore, organizzabili in unità tattiche e utilizzabili in missioni elementari quali l'attacco e la difesa. Tali SAF di prima generazione risultavano tuttavia fortemente "hard-coded" e scarsamente flessibili: qualunque modifica dottrinale o miglioramento comportamentale richiedeva la ricompilazione integrale del software, con costi rilevanti in termini di tempo e risorse.

Il superamento di tali limiti portò allo sviluppo di ModSAF (Modular Semi-Automated Forces), riconosciuto come terza generazione di SAF e concepito per supportare la Distributed Interactive Simulation (DIS), allora emergente nel corso degli anni Novanta. ModSAF introdusse un'architettura modulare e aperta, che rendeva possibile a un singolo operatore creare e gestire un numero elevato di entità (veicoli terrestri e aerei, fanteria, missili, infrastrutture dinamiche) su un campo di battaglia virtuale, con un livello di realismo tale da rendere ardua, a un primo sguardo, la distinzione tra unità generate dal calcolatore e unità pilotate da equipaggi umani. Il progetto, finanziato congiuntamente da ARPA e STRICOM, si proponeva esplicitamente di offrire una piattaforma di ricerca alla comunità DIS e CGF, agevolando l'estensione del sistema a nuove tipologie di entità e di comportamenti.

Con la progressiva standardizzazione dei protocolli DIS e con il mutare delle esigenze operative, ModSAF cedette gradualmente il passo a OneSAF (One Semi-Automated Forces), programma dell'Esercito statunitense ideato per riunire in un'unica piattaforma le funzionalità dei vari sistemi SAF/CGF che lo avevano preceduto. OneSAF è concepito per supportare scenari articolati su più livelli (tattico e operativo) comprendendo forze con equipaggio, forze generate dal calcolatore e modelli aggregati; nel tempo, la sua evoluzione lo ha portato a supportare anche attività quali la pianificazione della missione e l'analisi di corso d'azione. In parallelo, altri sistemi CGF, tra cui JCATS (Joint Conflict and Tactical Simulation) e diverse applicazioni basate su DIS/HLA, sono stati adottati in ambiti interforze e multinazionali, estendendo il ruolo dei CGF ben oltre il semplice addestramento tattico.

Negli anni più recenti, l'evoluzione dei CGF risulta strettamente connessa allo sviluppo di nuove architetture e interfacce basate sull'intelligenza artificiale, con particolare attenzione alla modellazione del comportamento data-driven e all'adozione di tecniche di Machine Learning. OneSAF e i programmi che ne sono derivati sono stati arricchiti con moduli comportamentali più sofisticati e con interfacce che rendono possibile sperimentare algoritmi di apprendimento automatico; iniziative come il Synthetic Training Environment (STE) dell'Esercito statunitense mirano, dal canto loro, a integrare CGF "intelligenti" all'interno di ambienti LVC (Live-Virtual-Constructive) unificati. Su questa linea, numerosi studi accademici e industriali esplorano architetture di "next-generation CGF" fondate su principi di simulazione web-based, servizi distribuiti e modelli cognitivi avanzati, con l'obiettivo di sostenere l'intero spettro delle operazioni militari (addestramento, analisi, ricerca e sviluppo) in maniera scalabile e interoperabile [8].

1.3.2 Architettura e modellazione del comportamento

Tra gli aspetti coinvolti nello sviluppo di un CGF, la modellazione del comportamento (Behavior Modeling) rappresenta senza dubbio il più complesso. Tradizionalmente, lo sviluppo dei CGF costituisce un'attività onerosa sia dal punto di vista economico sia da quello temporale, nella quale l'esperienza militare e la conoscenza dottrinale devono essere interpretate e tradotte in codice manualmente dagli sviluppatori. Un simile approccio, per quanto efficace, tende a generare entità vincolate a regole fisse o euristiche, non sempre capaci di riprodurre la complessità e l'imprevedibilità tipiche del comportamento della controparte reale. Per far fronte a questa complessità, la ricerca si orienta verso paradigmi alternativi, tra cui:

- **Data-Driven Behavior Modeling (DDBM):** si basa su tecniche di Machine Learning (reti neurali, algoritmi genetici, Reinforcement Learning) per ricavare le regole comportamentali direttamente dai dati. Mediante l'Observational Learning, il CGF apprende imitando le azioni di un essere umano osservate in scenari reali o simulati; mediante l'Experiential Learning (per tentativi ed errori), affina invece il proprio comportamento sulla base di funzioni di ricompensa. Lo scopo è ottenere comportamenti più oggettivi, realistici e vicini a quelli umani [7].
- **Linguaggi di Scripting (es. Lua):** numerosi CGF commerciali di ultima generazione ricorrono a standard come il linguaggio di scripting Lua per la modellazione del comportamento umano. Si tratta di un approccio che mette a disposizione un insieme di strumenti compatto ma al tempo stesso estendibile, particolarmente adatto a consentire agli utenti finali di configurare e prototipare rapidamente comportamenti specifici, senza dover intervenire sul codice sorgente del motore di simulazione [9].

1.3.3 Livelli di astrazione: entità e modelli aggregati

A seconda della finalità della simulazione, i CGF possono essere progettati per operare secondo diversi livelli di fedeltà e di astrazione:

- **Livello Tattico (Entity-level):** il CGF rappresenta il singolo soldato, veicolo o team, con un'attenzione rivolta a scenari di breve durata caratterizzati da interazioni fisiche e spaziali dettagliate (movimento, linea di vista, ingaggio).
- **Livello Operativo/Strategico (Aggregate-level):** negli scenari di lungo periodo che coinvolgono migliaia di attori, la modellazione dei singoli individui diventa computazionalmente troppo onerosa per garantire il rispetto del tempo reale. Si ricorre pertanto a modelli aggregati, rappresentativi di interi gruppi o organizzazioni sociali (ad esempio forze armate, popolazione civile, fazioni ribelli). Questi CGF di tipo "macro" fondano le proprie decisioni su variabili di

stato interne (quali potenza bellica, economia, insoddisfazione) e su variabili relazionali, come il grado di ostilità o di amicizia nei confronti di altri gruppi, avvalendosi di approcci probabilistici, tra cui le Reti Bayesiane, per la selezione delle azioni [10].

1.3.4 Sfide tecnologiche e architetture Web-Based per i CGF

L'adozione di architetture distribuite e del paradigma MSaaS comporta un ripensamento radicale nella progettazione del software dei sistemi CGF. Mentre nelle generazioni precedenti il motore di simulazione (backend) e il visualizzatore tattico (frontend) costituivano moduli strettamente accoppiati, con condivisione della medesima memoria fisica, i sistemi attuali richiedono un disaccoppiamento rigoroso, sia logico sia fisico.

La separazione fra la logica di business (la simulazione e il comportamento delle Forze Computer-Generate) e la logica di presentazione (l'interfaccia utente interattiva) comporta sfide ingegneristiche tutt'altro che banali, in particolare quando il visualizzatore è realizzato attraverso tecnologie web (Web-Based GUI).

Fra le principali criticità tecniche che caratterizzano questa transizione architetturale si possono citare:

- **Gestione della larghezza di banda e latenza:** negli scenari tattici complessi, che coinvolgono migliaia di entità terrestri, aeree e navali, il motore di simulazione produce un flusso costante di dati di stato (posizione geospaziale, orientamento, velocità, stato dei sensori e dei sistemi d'arma). La trasmissione in rete, ad alta frequenza, di un simile volume di informazioni verso un client web necessita di protocolli di comunicazione fortemente ottimizzati, allo scopo di scongiurare colli di bottiglia e latenze in grado di compromettere il "tempo reale" della simulazione.

- **Gestione dello stato lato client:** un'interfaccia web moderna deve poter ricevere, decodificare e aggiornare il proprio stato interno (il DOM virtuale) decine di volte al secondo, senza per questo bloccare il main thread del browser. Ciò richiede il ricorso a librerie reattive e a pattern avanzati di state management, capaci di distinguere i dati statici (ad esempio menu e configurazioni) da quelli soggetti a una frequenza di mutazione molto elevata.

- **Rendering spaziale e cartografico ad alte prestazioni:** la situational awareness (consapevolezza situazionale) costituisce il fulcro operativo di un'interfaccia CGF. Il client è chiamato a renderizzare, in tempo reale, mappe

satellitari, modelli di elevazione del terreno (DEM) ed entità vettoriali tridimensionali. Demandare al browser web una simile mole di calcoli grafici richiede l'impiego di librerie basate su WebGL, come CesiumJS, in grado di sfruttare l'accelerazione hardware della GPU locale dell'operatore, preservando al tempo stesso un framerate fluido e stabile.

1.4 Obiettivi del progetto e motivazioni della tesi

Il presente lavoro di tesi si pone come obiettivo primario quello di guidare l'evoluzione di un software di simulazione, allo stato attuale ancora in una fase embrionale, trasformandolo in una piattaforma moderna e di livello industriale per le simulazioni di Forze Computer-Generate (CGF) in ambito militare. Tale esigenza di transizione trae forte ispirazione da soluzioni commerciali di riferimento quali VR-Forces, piattaforma multi-dominio avanzata capace di simulare scenari complessi che coinvolgono assetti terrestri, aerei, marittimi e spaziali. Il successo di sistemi come VR-Forces risiede in un'architettura fortemente modulare, nella quale il motore di simulazione costituisce, ad esempio, un modulo distinto rispetto all'interfaccia grafica, a garanzia di flessibilità e scalabilità [11].

Allo scopo di colmare tale divario e realizzare un simulation engine altrettanto all'avanguardia, il progetto pone l'accento sullo sviluppo di un'interfaccia web avanzata (GUI) realizzata con React. Il passaggio a una GUI di concezione moderna comporta, inoltre, la necessità di definire e implementare un protocollo di comunicazione ottimizzato, elemento imprescindibile per l'accesso e la gestione in tempo reale di dati cartografici e mappe satellitari.

Tale approccio, fondato su un'architettura frontend modulare, si propone di affrontare e risolvere le criticità tecniche che caratterizzano i moderni sistemi di simulazione. Il lavoro si concentra, in particolare, sull'ottimizzazione delle prestazioni nello scambio di dati in tempo reale, con l'obiettivo di garantire la scalabilità del sistema e di contenere al minimo sia la latenza sia l'utilizzo della larghezza di banda. L'obiettivo ultimo della tesi è la validazione del sistema realizzato, condotta attraverso test funzionali e di performance, a dimostrazione del fatto che il passaggio a questa nuova piattaforma web, reattiva e affidabile, è in grado di sostenere efficacemente sistemi di simulazione, visualizzazione e supporto alle decisioni di livello industriale.

1.5 Metodologia di Lavoro

Lo sviluppo del progetto si è basato su una metodologia di impostazione Agile, articolata in cicli iterativi (sprint) della durata di due settimane. Tale approccio ha

assicurato un elevato grado di flessibilità, unitamente a un costante allineamento rispetto ai requisiti in evoluzione, grazie anche a incontri di sincronizzazione settimanali che hanno coinvolto l'intero gruppo di lavoro.

Il contesto operativo aziendale si è caratterizzato per una marcata natura multidisciplinare, tratto essenziale nel dominio del Modeling & Simulation. Il team ha infatti riunito figure professionali diverse tra loro: sviluppatori dedicati alle architetture di backend, ma anche profili con formazione in ambito fisico, impiegati nella modellazione matematica e cinematica delle entità simulate. L'intero processo di sviluppo è stato accompagnato in modo costante da figure ingegneristiche senior e dalla supervisione tecnica aziendale, a garanzia del rispetto degli standard architetturali e qualitativi.

All'interno di questo ecosistema, il presente lavoro si è concentrato principalmente sulla progettazione e sull'implementazione della nuova interfaccia grafica e del relativo livello di comunicazione. La natura fortemente interconnessa del sistema ha tuttavia richiesto un coinvolgimento trasversale sull'intero ciclo di vita del software: nello specifico, l'attività ha previsto una fase iniziale di raccolta e analisi dei requisiti, con la conseguente stesura degli use case e degli scenari operativi. In una fase successiva, per garantire la corretta interoperabilità tra i moduli, le attività di sviluppo front-end sono state affiancate da interventi diretti sul codice di backend, tra cui la progettazione e la scrittura di test di integrazione lato server, volti a validare la stabilità del motore di simulazione.

1.6 Struttura della Tesi

Il presente elaborato si articola in sette capitoli, pensati per accompagnare il lettore dai fondamenti teorici del dominio della simulazione fino all'implementazione pratica della piattaforma web e alla validazione dei risultati conseguiti.

Il presente capitolo ha delineato il contesto aziendale entro cui si è svolto il tirocinio, introducendo i concetti fondamentali del Modeling & Simulation in ambito Difesa (con particolare attenzione al paradigma LVC e ai sistemi CGF) per poi definire gli obiettivi e la metodologia di lavoro adottata.

Il secondo capitolo si propone di approfondire il background tecnologico e lo stato dell'arte: verranno esaminati i protocolli standard di interoperabilità (DIS e HLA) e i framework e motori di simulazione di backend adottati, tra cui MIXR/OpenEaagles e JSBSim, oltre a offrire una panoramica degli strumenti selezionati per lo sviluppo frontend, quali React, TypeScript e la libreria CesiumJS per il rendering geospaziale.

Il terzo capitolo è dedicato all'analisi dei requisiti e alla progettazione del sistema: vengono approfondite le dinamiche di gestione degli scenari operativi, realizzata

tramite il parsing di specifici file di configurazione (.epp e .edl), e viene proposta un'analisi critica dell'architettura di partenza (AS-IS), passaggio indispensabile per la definizione della nuova architettura modulare (TO-BE).

Il quarto capitolo verte sul design dell'interfaccia di comunicazione: viene descritta l'ingegnerizzazione del protocollo di scambio dati tra il motore C++ e il client React, con un'analisi dei meccanismi adottati per ottimizzare il flusso di dati cartografici e di stato della simulazione in tempo reale, allo scopo di ridurre al minimo il carico computazionale.

Il quinto capitolo entra nel merito dello sviluppo dell'interfaccia grafica (GUI): vi si illustrano l'architettura modulare React per la gestione ad alta frequenza dello stato, l'integrazione cartografica realizzata tramite Cesium, nonché le soluzioni individuate per affrontare le sfide tecniche connesse alle prestazioni di rendering.

Il sesto capitolo presenta le metodologie di testing e validazione impiegate per assicurare la robustezza dell'intero sistema distribuito, descrivendo gli integration test lato backend e le procedure di validazione funzionale e di performance lato frontend, indispensabili per verificare la reattività e l'efficienza della piattaforma anche sotto carico.

Il settimo e ultimo capitolo, infine, propone una sintesi complessiva del lavoro svolto, valutando i risultati raggiunti alla luce degli obiettivi iniziali e l'applicabilità industriale dell'architettura proposta, oltre a discutere i possibili sviluppi futuri in vista di un impiego commerciale su larga scala del sistema.

Capitolo 2

Background Tecnologico e Stato dell'Arte

2.1 Protocolli di Simulazione Distribuita

L'aumento della complessità dei sistemi contemporanei, unito alla necessità di far cooperare risorse e operatori dislocati in luoghi diversi, ha determinato il rapido superamento dell'approccio monolitico alla simulazione. L'esecuzione di modelli ad alta fedeltà comporta infatti oneri computazionali che, sovente, un singolo calcolatore non è in grado di sostenere; è proprio per colmare tale lacuna che interviene la simulazione distribuita. Grazie a questo metodo, il carico di elaborazione viene ripartito su reti di calcolatori, rendendo possibile la costruzione di ambienti sintetici caratterizzati da una popolazione massiva di entità.

Di conseguenza, la simulazione non viene più intesa unicamente come strumento di addestramento fine a se stesso, bensì assume il ruolo di banco di prova imprescindibile per validare l'interoperabilità tra piattaforme, elaborare nuove dottrine tattiche e supportare i processi di decision-making. Affinché simulatori eterogenei realizzati da produttori differenti o appartenenti a generazioni tecnologiche distinte possano interagire all'interno di uno stesso scenario, si è reso indispensabile definire un insieme di regole condivise; in mancanza di protocolli comuni, infatti, non sarebbe possibile alcuno scambio coerente di dati fra i nodi della rete.

Nell'ambito della modellazione e simulazione (M&S), specialmente in relazione al settore militare e della difesa, i due standard di riferimento rimangono il DIS (Distributed Interactive Simulation) e l'HLA (High Level Architecture). Pur perseguendo il medesimo obiettivo, ossia mettere in comunicazione sistemi differenti,

i due protocolli poggiano su impostazioni architetturali diametralmente opposte, che si traducono in modalità di sviluppo e utilizzo profondamente diverse tra loro. [12]

2.1.1 Distributed Interactive Simulation (DIS)

Il protocollo Distributed Interactive Simulation (DIS), disciplinato dalla serie di standard IEEE 1278, ha costituito il primo strumento realmente efficace per la simulazione distribuita, nato con l'obiettivo di mettere in collegamento simulatori eterogenei e dare vita a mondi virtuali complessi. Si tratta di una tecnologia configurata come protocollo di rete, o wire protocol [13], concepita soprattutto per l'impiego in ambienti virtuali operanti in tempo reale.

Il meccanismo su cui si fonda prevede l'impiego di un insieme fisso e standardizzato di messaggi, le Protocol Data Units (PDU), scambiate continuamente allo scopo di aggiornare le informazioni di stato. A titolo esemplificativo, l'Entity State PDU (ESPDU) veicola i dati concernenti posizione, orientamento e velocità di una determinata entità. Lo standard stabilisce con precisione tanto il formato binario dei dati, definendone la sintassi, quanto il relativo significato semantico, specificando i sistemi di coordinate da adottare, le unità di misura e i valori enumerati [12]. Per quanto riguarda la trasmissione, i messaggi DIS viaggiano solitamente in rete sfruttando il protocollo di trasporto UDP in modalità broadcast [6].

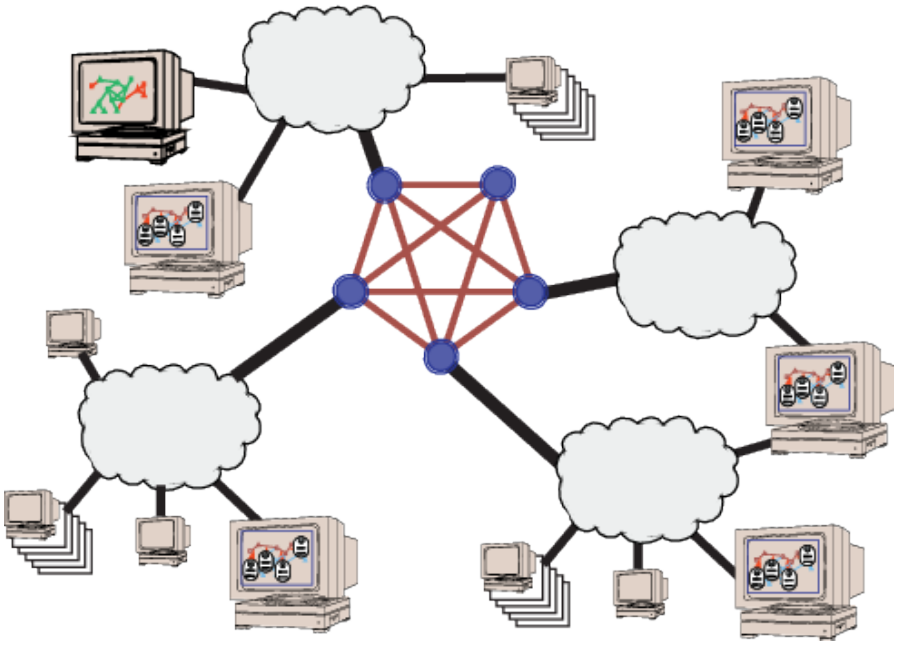


Figura 2.1: Simulatori indipendenti connessi tramite DIS [14].

Una simile impostazione strutturale assicura un elevato livello di interoperabilità a livello di rete: è sufficiente che un'applicazione sia in grado di leggere e interpretare i pacchetti conformi alla norma per poter prendere parte alla simulazione. Ciò rende possibile ricorrere a linguaggi di scripting moderni, quale Javascript, senza dover fare affidamento su librerie proprietarie. L'impianto complessivo mostra, tuttavia, una certa rigidità riconducibile alla propria origine: essendo stato pensato per riprodurre in tempo reale il comportamento di veicoli fisici, l'insieme dei messaggi si presta con difficoltà ad adattarsi a simulazioni di tipo logistico o costruttivo. Nelle versioni più recenti dello standard, fra cui l'IEEE 1278.1-200X, sono stati introdotti i record "Variable Parameter" nel tentativo di estendere i PDU già esistenti. Nonostante tale soluzione tecnica consenta di mantenere la retrocompatibilità dei sistemi, l'architettura di fondo continua a essere strettamente vincolata ai propri formati predefiniti.

2.1.2 High Level Architecture (HLA)

L'High Level Architecture (HLA) costituisce uno standard sviluppato su iniziativa del Dipartimento della Difesa statunitense, successivamente formalizzato a livello internazionale come IEEE 1516 e adottato dalla NATO per mezzo dello STANAG 4603. La sua ideazione risponde all'ambizione di realizzare un'architettura unica, in grado di trovare applicazione in una gamma di domini molto più estesa. Diversamente da quanto avviene con il protocollo DIS, l'HLA non impone alcuna regola sul formato dei dati veicolati in rete; l'attenzione si sposta piuttosto sulla standardizzazione dell'Application Programming Interface (API) e delle regole che governano l'interazione tra i software.

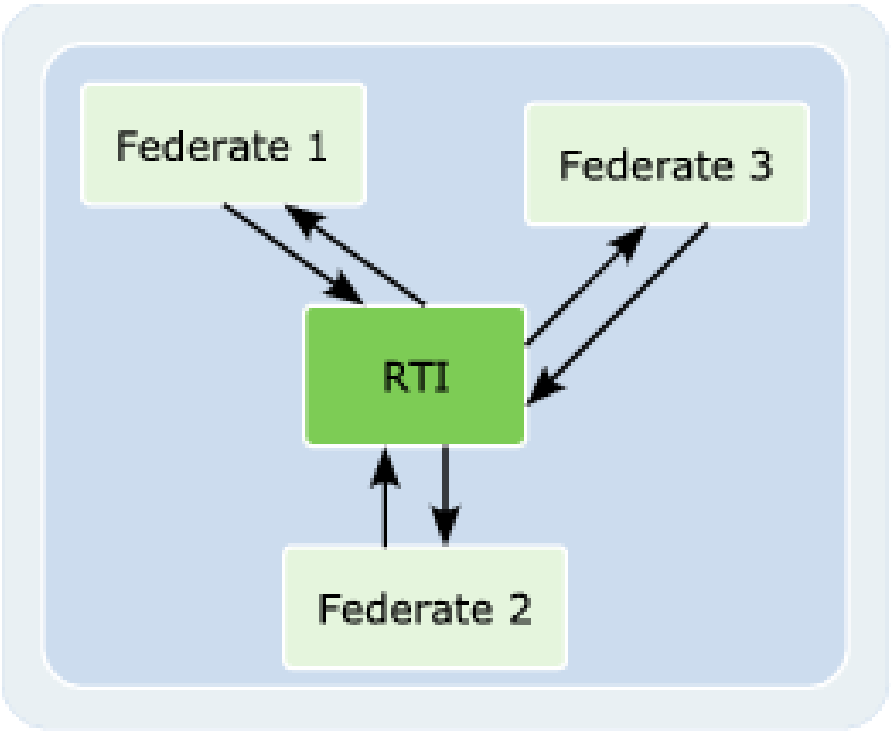


Figura 2.2: Struttura base dell'HLA [15]

L'architettura del sistema poggia su alcuni concetti fondamentali e prende le distanze da schemi rigidi e predefiniti.

La denominazione Federation (Federazione) designa la simulazione globale distribuita, composta da una molteplicità di entità dette Federates (Federati).

L'elemento centrale dell'architettura è rappresentato dalla Run-Time Infrastructure (RTI): tale middleware eroga i servizi di comunicazione e gestione indispensabili alle simulazioni partecipanti, dialogando con ciascun federato per mezzo dell'API

standardizzata. Le informazioni scambiate nel corso dell'esecuzione non risultano fissate a priori dalla norma, bensì vengono stabilite dagli sviluppatori stessi all'interno di un apposito file di configurazione, il Federation Object Model (FOM), tenuto a conformarsi a un Object Model Template (OMT).

Tale architettura consente lo svolgimento di simulazioni che non procedono necessariamente secondo il tempo reale, o wall-clock time; questa caratteristica rende l'HLA particolarmente adatto alla simulazione a eventi discreti nonché a esecuzioni accelerate o rallentate. Il protocollo introduce, inoltre, il Distributed Data Management (DDM), un sistema che filtra il traffico presente in rete instradando i dati unicamente verso i partecipanti effettivamente interessati alla loro ricezione, con conseguente riduzione della congestione complessiva dell'infrastruttura.

Ne deriva che la notevole flessibilità offerta dal FOM consente di rappresentare qualsiasi tipologia di scenario, andando ben oltre i confini delle applicazioni di natura strettamente tattica. Permane, tuttavia, un limite strutturale nell'impianto: la mancata standardizzazione del formato di rete, ovvero il wire format, fa sì che i software RTI realizzati da fornitori diversi, quali MÄK o Pitch, non siano in grado di comunicare direttamente tra loro [16].

2.1.3 Stato dell'Arte e Strategie di Interoperabilità Ibrida

Nel contesto attuale, assicurare una piena interoperabilità fra sistemi complessi richiede non di rado il ricorso a soluzioni ibride. Al fine di agevolare il passaggio delle simulazioni dal DIS all'HLA, è stato definito lo standard RPR-FOM (Real-time Platform Reference FOM), un modello che riproduce fedelmente la semantica, i sistemi di coordinate e i messaggi propri del DIS, riportandoli all'interno del paradigma HLA.

Sul piano operativo, mettere in comunicazione due simulazioni HLA basate su RTI di fornitori differenti si rivela spesso oneroso, oltre che tecnicamente complesso. Per superare tale difficoltà, lo stato dell'arte nell'integrazione di sistemi (system of systems) fa ricorso a gateway dedicati. Numerose federazioni impiegano il DIS come "lingua franca", ovvero come dorsale di comunicazione (backbone): i gateway convertono i dati dal formato RPR-FOM proprio dell'HLA in pacchetti DIS destinati al transito in rete, per poi ritradurli in HLA una volta giunti a destinazione, aggirando così le incompatibilità di rete esistenti fra le diverse infrastrutture RTI.

2.2 Framework e Motori di simulazione

2.2.1 MIXR (Mixed Reality Simulation Platform)

Detto in termini semplici, MIXR rappresenta un ambiente di sviluppo (framework) concepito per la realizzazione di simulatori. Analogamente a quanto fanno i "motori" grafici (si pensi a Unity o a Unreal Engine) che costituiscono la base per la creazione di videogiochi, MIXR mette a disposizione degli ingegneri del software un insieme di strumenti già predisposti per lo sviluppo di simulazioni aerospaziali e militari operanti in tempo reale.

In assenza di un framework di questa natura, la realizzazione di un simulatore di volo richiederebbe allo sviluppatore di implementare da zero la fisica, la gestione temporale, le interfacce sensoristiche e la comunicazione in rete. MIXR mette invece a disposizione un'infrastruttura fortemente ottimizzata, capace di occuparsi di tutti questi aspetti fondamentali, così da consentire allo sviluppatore di dedicarsi unicamente alla logica propria del proprio progetto. Il suo impiego spazia su un'ampia varietà di applicazioni: si va da semplici modelli matematici di natura ingegneristica, eseguibili su un singolo elaboratore, fino a giungere a simulatori addestrativi articolati, completi di cabina di pilotaggio e visori, interconnessi in rete con altri sistemi.

Cenni storici

Il progetto affonda le proprie radici nel settore della difesa: la sua origine risale infatti al laboratorio di ricerca dell'Aeronautica Militare statunitense (Air Force Research Laboratory), che lo sviluppò con l'acronimo EAAGLES (Extensible Architecture for the Analysis and Generation of Linked Environments and Simulations). In un momento successivo, il codice venne reso disponibile al pubblico sotto forma di progetto open-source, assumendo la denominazione di OpenEaagles.

Più recentemente, per dare conto della naturale evoluzione del software e della sua piena capacità di supportare le tecnologie contemporanee di realtà virtuale, aumentata e mista, il framework ha assunto ufficialmente il nome di MIXR (Mixed Reality Simulation Platform), pur conservando inalterata la propria solida architettura originaria.

Principi di Funzionamento e Architettura

MIXR si configura come un framework sviluppato in C++ secondo un paradigma orientato agli oggetti. Il flusso dei dati segue una logica di tipo "Push", per cui le informazioni vengono trasmesse in modo attivo verso i componenti che ne necessitano.

All'interno di MIXR, tale principio trova applicazione in due contesti distinti:

1. **Comunicazione tra Componenti:** un oggetto contenitore trasmette (push) i dati alle proprie liste di sottocomponenti.

2. **Elaborazione degli Input (Parser):** anche il sistema deputato alla lettura dei file di configurazione ricorre al medesimo meccanismo di push.

Il framework adotta il design pattern Model-View-Controller (MVC), suddividendo le componenti funzionali in appositi packages. Viene inoltre messa a disposizione un'interfaccia di rete di tipo astratto, grazie alla quale è possibile implementare protocolli personalizzati senza che ciò comporti ripercussioni sui modelli di sistema: ne sono un esempio i protocolli DIS e HLA.

MIXR consente di realizzare diverse tipologie di applicazioni simulative, eseguibili sia in modalità stand-alone sia in modalità distribuita; queste ultime sono in grado di interagire con altri sistemi e simulazioni per mezzo di DIS/HLA. L'applicazione può inoltre comprendere agenti software che riproducono la partecipazione umana (constructive), oppure prevedere l'interazione con un operatore umano reale (virtual).

Perché è stato scelto C++ per MIXR?

Il linguaggio C++ si distingue per la propria portabilità, potendo contare su compilatori disponibili su tutte le piattaforme; ciò consente agli sviluppatori di realizzare applicazioni basate su MIXR sui principali sistemi operativi esistenti. A ciò si aggiunge la flessibilità propria di C++, che consente di gestire la memoria senza dover ricorrere alla garbage collection.

Tra le classi che meritano particolare attenzione figura la classe Object. Diversamente da altri linguaggi orientati agli oggetti, C++ non prevede un System Object, ossia una classe base universale dalla quale derivino tutte le altre; il linguaggio, inoltre, non dispone nativamente di un meccanismo di garbage collection. Sebbene tali assenze possano apparire, a un primo sguardo, come dei limiti, esse si rivelano in realtà un vantaggio strategico nell'ambito delle applicazioni in tempo reale.

Il framework sfrutta appieno la natura di C++, lasciando allo sviluppatore la libertà di scegliere la strategia di gestione della memoria più adatta alle proprie esigenze:

1. Qualora il fattore tempo non risulti critico, lo sviluppatore può avvalersi degli smart pointers, delegando al programma la gestione automatica della creazione e della cancellazione degli oggetti;

2. nel caso invece di sistemi real-time, la creazione e la distruzione non controllate degli oggetti comporterebbero un dispendio eccessivo di tempo prezioso: è in questo contesto che entra in gioco la classe Object.

Quest'ultima mette infatti a disposizione un sistema interno di reference counting: anziché procedere continuamente alla distruzione e alla ricreazione degli oggetti, il sistema tiene traccia del numero di "utenti" che stanno facendo uso di una determinata porzione di memoria. In questo modo, gli sviluppatori possono esercitare un controllo manuale, altamente ottimizzato.

Un'ulteriore classe di rilievo è rappresentata da Component, ossia una classe di oggetti destinata a contenerne altri. All'interno di MIXR, Component svolge la funzione di contenitore per altre entità e definisce, al contempo, un sistema di messaggistica di base impiegato nell'intero framework. L'intera applicazione può dunque essere concepita come un ampio albero gerarchico di Componenti.

Come si costruisce un'applicazione in MIXR?

Nella prassi, uno sviluppatore che si avvale di MIXR tende a partire da classi già esistenti, per poi ampliarle con l'aggiunta di nuove funzionalità. Il passaggio successivo consiste nella stesura della funzione principale del programma, il main(). MIXR non mette a disposizione una funzione main() predefinita: il controllo resta pertanto interamente nelle mani dello sviluppatore, allo scopo di garantire la massima flessibilità.

Il codice del main segue, in linea generale, la struttura seguente:

1. **Lettura del file di input:** viene letto un file di testo che descrive la gerarchia degli oggetti e i relativi attributi; il framework dispone di un parser dedicato all'interpretazione di tale linguaggio di configurazione.
2. **Configurazione dei thread:**
   - qualora non sussistano vincoli temporali, è sufficiente un singolo thread.
   - in presenza di vincoli di tempo (real-time), viene invece creato un thread ad alta priorità, denominato thread time-critical.
3. **Avvio della simulazione:** i metodi updateTC (Time Critical) e updateData vengono richiamati ciclicamente.

Organizzazione di Alto Livello

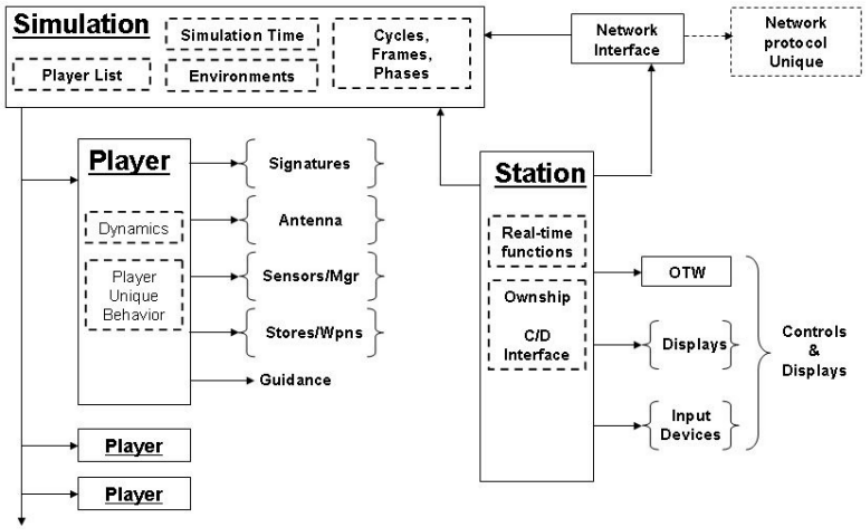


Figura 2.3: Struttura delle simulazioni [17].

Se si immagina l'applicazione come un albero di componenti, al vertice della struttura si colloca la classe Station, che rappresenta il nodo radice; ogni altro componente discende, in qualità di figlio, da Station.

Dal momento che MIXR si fonda sul pattern MVC, la classe Station svolge la funzione di elemento di collegamento: essa mette in comunicazione i Modelli (i dati e la logica della simulazione) con le Viste (i display grafici o visivi) e con i Controllori (gli input provenienti dall'utente o dall'hardware). Station si occupa inoltre della gestione dell'oggetto globale Simulation, il quale a sua volta tiene traccia dell'elenco dei partecipanti (le entità) e sovrintende allo scorrimento del tempo.

All'interno del framework, un Player costituisce una sottoclasse particolare di Component, comprensiva di logiche di dinamica e di comportamenti specifici, quali quelli propri di un veicolo aereo, terrestre o marittimo. A un medesimo Player possono essere associati sensori, antenne, sistemi d'arma o dispositivi di navigazione.

Da ultimo, MIXR definisce un'interfaccia di rete avanzata, funzionale all'interoperabilità: grazie all'adozione di protocolli militari standard, la simulazione è in grado di interagire con altri programmi dislocati in luoghi geograficamente distanti. Quando l'interfaccia di rete riceve dati provenienti da una simulazione esterna, provvede automaticamente a generare nuovi Player all'interno della lista locale; agli occhi del motore di calcolo del framework, tali player esterni (denominati networked

players) si comportano in maniera del tutto identica a quelli generati localmente. [18]

2.2.2 JSBSim Flight Dynamics Model

JSBSim costituisce un modello di dinamica del volo (FDM), realizzato prevalentemente in linguaggio C++. Il suo compito consiste nel fungere da motore matematico e fisico, occupandosi di calcolare e sommare le forze e i momenti generati sia dall'ambiente circostante (come, ad esempio, l'atmosfera) sia dai comandi di controllo, aggiornando conseguentemente lo stato del veicolo a intervalli di tempo discreti.

Da un punto di vista di alto livello, JSBSim può essere paragonato a una "scatola nera" (black box), interamente configurabile per mezzo di file di input in formato XML. Tali file contengono la descrizione delle caratteristiche aerodinamiche del veicolo, dei propulsori e dei sistemi di controllo. Una volta caricati, essi consentono a JSBSim di riprodurre il volo in tempo reale nell'ambito di un framework più ampio, fornendo in output le prestazioni e la dinamica del veicolo stesso.

Concetti Chiave

Ai fini del calcolo della fisica del volo e della determinazione della posizione degli elementi, JSBSim fa ricorso a diversi sistemi di riferimento (Frames of reference). Relativamente alle unità di misura, il software adotta nativamente il sistema imperiale anglosassone, pur consentendo l'impiego di unità alternative; risulta tuttavia essenziale indicarle sempre con precisione all'interno dei file di configurazione, così da evitare errori nei calcoli. Per l'apertura alare, ad esempio, si potrà indicare <wingspan unit="FT"> 35.8 </wingspan> qualora si intenda esprimerla in piedi, oppure <wingspan unit="M"> 10.91 </wingspan> qualora la si esprima in metri.

Nell'ambito delle simulazioni più complesse, la quantità di informazioni da gestire risulta particolarmente elevata. Per far fronte a tale problematica, JSBSim implementa il Property Manager, la cui caratteristica principale risiede nella capacità di generare nuove variabili in maniera dinamica durante l'esecuzione del programma (runtime).

Tale flessibilità si rivela indispensabile per il Flight Control System (FCS): i componenti del sistema di controllo, infatti, non risultano "scolpiti" all'interno del codice C++, bensì vengono definiti nei file XML. Nel corso della lettura del file (parsing), il Property Manager provvede a creare al volo una "proprietà" (una sorta di variabile globale organizzata secondo una struttura ad albero) destinata a memorizzare il risultato prodotto da ciascun componente.

In questo modo, l'output generato da un componente (si pensi, ad esempio, al comando dell'equilibratore) può essere trasmesso come input al componente successivo (come, ad esempio, un amplificatore di segnale), semplicemente richiamandone il nome della proprietà all'interno del file XML. Ciò rende possibile modificare in maniera sostanziale il comportamento del velivolo, oppure aggiungere nuovi interruttori e logiche di controllo, senza che sia mai necessario intervenire sul codice sorgente o procedere a una sua ricompilazione.

Scripting

Oltre al pilotaggio condotto in tempo reale, JSBSim consente altresì l'esecuzione di simulazioni automatizzate per mezzo di script, anch'essi redatti in formato XML.

Uno script si compone di una serie di eventi, ciascuno dei quali risulta associato a una determinata condizione (quale, ad esempio, il superamento di una certa quota oppure lo scadere di un intervallo di tempo prestabilito). Nel momento in cui la condizione si verifica, il sistema procede all'esecuzione delle azioni previste, come l'impostazione della manetta al valore massimo o l'attivazione dell'autopilota. Le transizioni dei valori possono avvenire in maniera istantanea oppure graduale, seguendo rampe o funzioni esponenziali opportunamente configurabili.

2.2.3 Integrazione di JSBSim in MIXR

Come già accennato in precedenza, MIXR fornisce l'architettura generale necessaria alla creazione di simulazioni robuste e scalabili, rendendo possibile la combinazione, all'interno del medesimo ambiente, di modelli caratterizzati da differenti livelli di fedeltà.

In tale ecosistema, JSBSim viene integrato allo scopo di svolgere la funzione di motore dinamico di un'entità aeronautica.

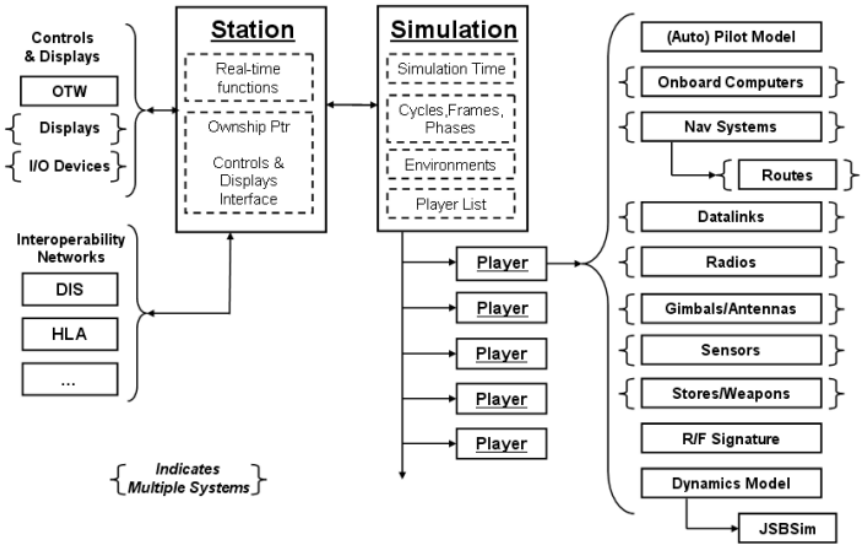


Figura 2.4: MIXR (OpenEaagles) Simulation Framework [18].

Osservando la figura d'insieme, è possibile notare come JSBSim risulti incapsulato all'interno di un oggetto Player (il quale, in MIXR, rappresenta un veicolo o comunque un partecipante alla simulazione).

Tale Player, ora dotato di un comportamento fisico realistico fornito da JSBSim, si colloca all'interno del mondo virtuale governato dalla Station, la quale mantiene il controllo sull'avanzamento temporale e svolge al contempo la funzione di tramite rispetto agli input dell'utente e al resto dell'architettura di rete.

2.2.4 Modellazione Dichiarativa delle Entità tramite EDL

All'interno di MIXR, le entità e le rispettive caratteristiche non vengono scritte in maniera rigida nel codice sorgente (hard-coded), bensì definite esternamente per mezzo di file di configurazione basati sul linguaggio EDL (Eaagles Definition Language). Tale linguaggio, per struttura assimilabile a JSON, consente di descrivere in modo dichiarativo un'entità e la totalità delle sue proprietà.

All'avvio dell'applicazione, il sistema si avvale di un parser interno per procedere alla lettura del file EDL, istruendo di conseguenza MIXR affinché provveda a istanziare dinamicamente il corrispondente oggetto C++. Tale oggetto racchiuderà al proprio interno tutti i modelli indispensabili al suo funzionamento, tra cui, a titolo esemplificativo, le istruzioni relative al calcolo dinamico, affidate a JSBSim.

Oltre a definire la fisica e l'aspetto delle entità, il file EDL viene altresì impiegato per istanziare il "cervello" dei veicoli autonomi, richiamando a tale scopo le classi appartenenti alla libreria UBF.

2.3 Architetture di Intelligenza Artificiale per CGF: Unified Behavior Framework (UBF)

2.3.1 Limiti degli approcci tradizionali e dei Behavior Trees

La finalità degli agenti autonomi (Computer-Generated Forces) all'interno dei sistemi di simulazione consiste nel riprodurre un'intelligenza tattica realistica, rendendo possibile la rappresentazione di scenari operativi complessi a fini di addestramento o di ricerca. Per far fronte a situazioni via via più articolate, sono disponibili diversi sistemi finalizzati all'implementazione dell'intelligenza artificiale, quali le macchine a stati finiti, le architetture Belief-Desire-Intention (BDI), i sistemi Soar e gli alberi comportamentali (Behavior Trees – BTs).

Gli approcci fondati sui BTs organizzano la selezione delle azioni ricorrendo a una gerarchia ad albero, costituita da nodi decisionali (selezione, sequenza, parallelo) che regolano il flusso di esecuzione indirizzandolo verso i nodi comportamentali, ossia le foglie. Una simile struttura assicura un'ottima modularità e contribuisce a contenere la complessità del codice, favorendo al contempo il riutilizzo dei sotto-alberi.

I BTs presentano, tuttavia, un limite di rilievo: risulta infatti complesso, anche qualora i comportamenti vengano eseguiti in parallelo, combinare tra loro gli output di più azioni che agiscono contemporaneamente sul medesimo motore dell'agente. Una simile combinazione risulterebbe invece particolarmente auspicabile, poiché consentirebbe l'emergere di comportamenti complessi a partire da blocchi elementari, riducendo ulteriormente la necessità di ricorrere alla scrittura di codice specifico.

L'UBF (Unified Behavior Framework) è stato concepito proprio per superare tale limite, rendendo possibile la fusione delle azioni senza per questo rinunciare ai vantaggi strutturali propri dei BTs.

2.3.2 Principi di Progettazione dell'UBF

Nello sviluppo tradizionale della robotica, così come in quello delle simulazioni, la logica decisionale (il comportamento), il controllore (ossia il sistema deputato a tradurre le decisioni in comandi) e l'hardware finiscono sovente per risultare strettamente intrecciati tra loro. Un simile accoppiamento stretto conferisce al codice un carattere "application-specific", rendendolo difficilmente riutilizzabile, e vincola inoltre l'agente a un unico modello decisionale.

L'UBF supera tale rigidità disaccoppiando in maniera netta la logica comportamentale dal controllore. Per conseguire tale grado di flessibilità, il framework si avvale di tre principi cardine propri della programmazione orientata agli oggetti:

1. **Incapsulamento:** nell'UBF, i comportamenti non impartiscono mai comandi diretti all'agente; al contrario, per mezzo di un metodo denominato genAction, si limitano a formulare una "raccomandazione" fondata sulla percezione dell'ambiente circostante. Tale raccomandazione risulta incapsulata all'interno di un oggetto Azione, che racchiude a sua volta dei "valori di voto" (priorità), indicativi dell'intensità con cui il comportamento suggerisce quella determinata azione.

2. **Composite Pattern:** i comportamenti risultano organizzati secondo una struttura gerarchica, per cui i comportamenti più complessi possono racchiudere al proprio interno comportamenti di natura più semplice. Grazie alla condivisione di una medesima interfaccia di base, lo sviluppatore è posto nelle condizioni di combinarli in modo uniforme, dando così vita a nuove forme di intelligenza stratificata.

3. **Strategy Pattern:** il controllore viene sollevato da qualsivoglia logica decisionale, riducendosi la sua unica responsabilità all'interrogazione del comportamento attivo, al fine di ottenere l'oggetto Azione e, in un secondo momento, provvedere alla sua esecuzione sull'agente.

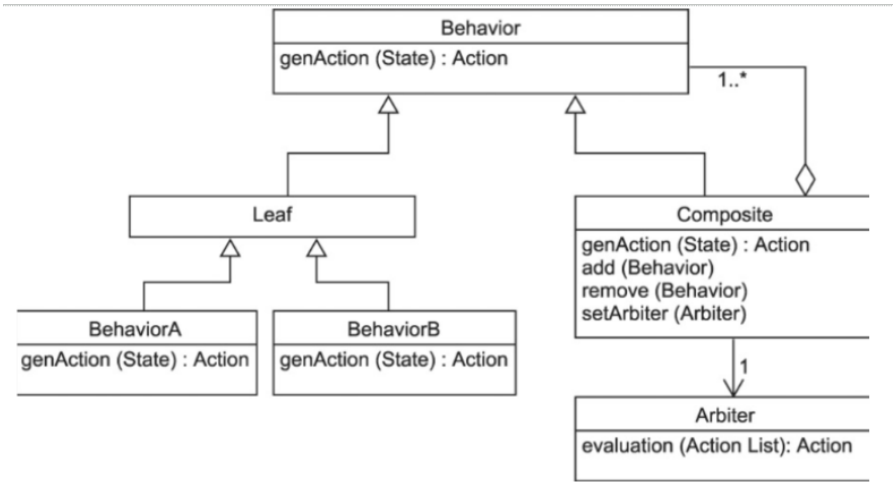


Figura 2.5: UML Diagram per UBF [19].

Esaminando l'architettura UML dell'UBF, è possibile individuare due implementazioni principali dell'interfaccia comportamentale astratta: i Leaf behaviors (comportamenti foglia) e i Composite behaviors (comportamenti compositi).

I Leaf behaviors racchiudono al proprio interno la logica algoritmica in senso proprio: il relativo metodo genAction procede all'analisi dello stato corrente, generando di conseguenza una raccomandazione. Grazie alla netta separazione tra le foglie

e i nodi compositi, la complessità matematica propria degli algoritmi decisionali rimane circoscritta esclusivamente all'interno dei Leaf behaviors.

I Composite behaviors, di converso, assolvono la funzione di contenitori per altri comportamenti "figli" (i quali possono, a loro volta, configurarsi come foglie oppure come ulteriori nodi compositi). Nel momento in cui il sistema necessita di una decisione, il nodo composito non procede al calcolo di un'azione propria, bensì dà avvio a una sorta di "sondaggio": richiama cioè il genAction di ciascuno dei propri figli, raccogliendo in tal modo un insieme composto da molteplici raccomandazioni differenti tra loro.

Poiché le regole architetturali del framework prevedono che al controllore venga restituita un'unica azione, il nodo composito rimette la risoluzione dell'eventuale conflitto a una componente interna denominata Arbiter. Quest'ultimo procede alla valutazione dell'intero insieme di azioni proposte dai figli, unificandole in un'unica raccomandazione conclusiva. Tale processo può articolarsi, in linea di massima, secondo due modalità principali:

- **Winner-Takes-All (WTA):** una competizione diretta, in esito alla quale viene selezionata unicamente l'azione proposta dal figlio caratterizzato dal valore di voto (priorità) più elevato;
- **Blending (Fusione):** una fusione di natura matematica tra i parametri delle diverse azioni proposte, dalla quale scaturisce un'azione di tipo ibrido (si pensi, a titolo esemplificativo, al calcolo di una media ponderata tra il comando volto a "evitare un ostacolo" e quello finalizzato a "raggiungere il bersaglio").

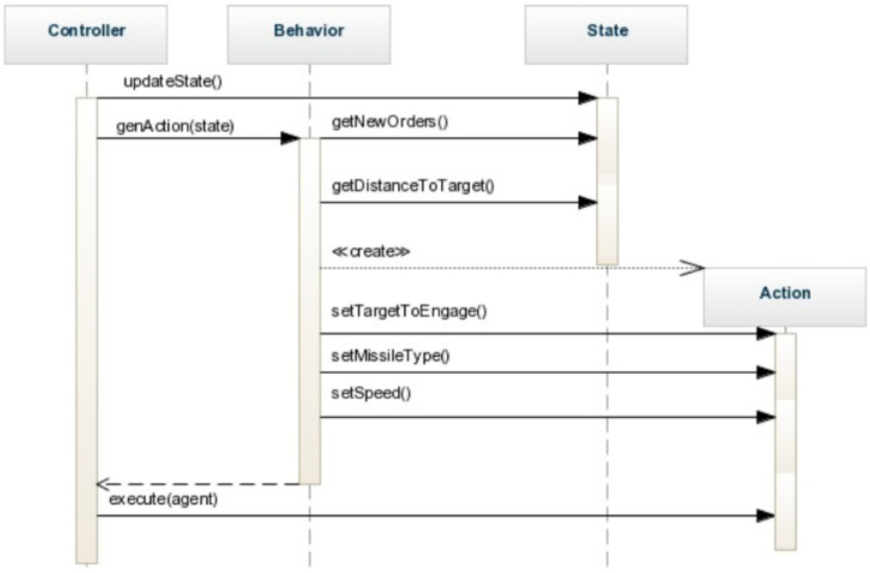


Figura 2.6: Flusso di esecuzione a runtime [19].

La sequenza cronologica delle chiamate eseguite a runtime nel simulatore segue un ciclo rigorosamente definito: in un primo momento, il Controllore provvede ad aggiornare la percezione del mondo circostante per mezzo di updateState(); successivamente, richiama genAction(state) sul comportamento radice. Quest'ultimo, una volta interrogato l'intero albero e risolti gli eventuali conflitti tramite l'Arbiter, restituisce un singolo oggetto Azione (si pensi, ad esempio, a setSpeed()). Il Controllore, infine, provvede all'esecuzione concreta di tale azione sull'agente per mezzo del comando execute(agent). [19]

2.4 Tecnologie Web e Stack Frontend

La realizzazione di un simulatore tattico fruibile tramite browser comporta il rispetto di vincoli tecnici particolarmente stringenti: un'interfaccia dotata di elevata reattività, un rendering 3D operante in tempo reale e una codebase complessa che risulti al contempo manutenibile. La scelta degli strumenti destinati al frontend costituisce dunque l'esito di un'analisi comparativa volta a soddisfare tali requisiti, privilegiando tecnologie mature, ampiamente diffuse nel settore e sostenute da un ecosistema attivo.

2.4.1 Vite: Build Tool e Development Server

Il processo di sviluppo moderno richiede la disponibilità di un build tool capace di occuparsi della trasformazione del codice sorgente, della risoluzione dei moduli, dell'ottimizzazione degli asset e del ricaricamento a caldo. Storicamente, strumenti quali Webpack hanno dominato questo ambito, sebbene a fronte di tempi di avvio e di ricompilazione destinati a dilatarsi progressivamente con l'aumentare delle dimensioni del progetto.

Vite [20], pubblicato nel 2020, rappresenta una rottura di natura architetturale rispetto ai bundler tradizionali. Nella fase di sviluppo, Vite si avvale dei moduli ES nativi, supportati dai browser di ultima generazione: anziché procedere alla costruzione di un unico bundle prima dell'avvio del server, esso provvede a servire i singoli moduli direttamente al browser su richiesta, rimettendo la risoluzione delle dipendenze al runtime del browser stesso. Un simile approccio consente di ridurre il tempo di avvio a freddo, portandolo da un ordine di grandezza dei secondi, tipico di Webpack, a poche decine di millisecondi, indipendentemente dalle dimensioni del progetto.

Analogamente ottimizzato risulta il meccanismo di Hot Module Replacement (HMR) proprio di Vite: le modifiche apportate al codice sorgente si propagano

istantaneamente nel browser, senza che sia necessario ricaricare l'intera applicazione, con conseguente preservazione dello stato dell'interfaccia nel corso dello sviluppo iterativo, caratteristica di particolare valore per un'applicazione dotata di uno stato geospaziale complesso, quale è appunto un simulatore tattico.

2.4.2 React: Libreria per la Costruzione di Interfacce Utente

Il livello dell'interfaccia grafica dell'applicazione poggia su React [21], la libreria JavaScript sviluppata e mantenuta da Meta (Facebook) a partire dal 2013, divenuta di fatto lo standard industriale per la realizzazione di interfacce utente web di natura complessa.

Il paradigma su cui si fonda React è rappresentato dalla programmazione dichiarativa basata su componenti: l'interfaccia viene descritta quale funzione dello stato dell'applicazione, mentre React si occupa di riconciliare in maniera efficiente il DOM reale con la propria rappresentazione virtuale (Virtual DOM) a ogni variazione di stato, effettuando aggiornamenti mirati esclusivamente sui nodi effettivamente modificati.

Nel contesto di un simulatore tattico, una simile architettura risulta particolarmente adeguata alla gestione di un'interfaccia articolata in numerosi pannelli eterogenei (l'elenco delle unità, il pannello di controllo della simulazione, il registro degli eventi, l'HUD geospaziale), ciascuno dei quali costituisce un sottoalbero di stato autonomo, sebbene coordinato con gli altri. Il ricorso ai React Hooks (useState, useEffect, useContext), unitamente all'impiego di librerie dedicate alla gestione dello stato globale, consente di modellare flussi di dati complessi mantenendo al contempo il codice modulare e agevolmente testabile.

Un ulteriore vantaggio, specifico per il contesto in esame, risiede nella netta separazione tra la logica di rendering e la logica applicativa: Cesium gestisce in maniera autonoma il proprio canvas WebGL e il relativo ciclo di rendering, mentre a React è demandato il solo governo dell'overlay HTML/CSS dell'interfaccia. La comunicazione tra i due livelli avviene per mezzo di riferimenti a oggetti Cesium, resi disponibili attraverso un contesto React o uno store condiviso, così da evitare conflitti tra i rispettivi cicli di aggiornamento.

2.4.3 TypeScript: Sistema di Tipi Statici per JavaScript

TypeScript [22], sviluppato e mantenuto da Microsoft, si configura come un superset tipizzato di JavaScript, che aggiunge al linguaggio un sistema di tipi statici di natura opzionale. Il codice scritto in TypeScript viene transpilato in JavaScript

puro prima della relativa esecuzione, risultando pertanto compatibile con qualsiasi ambiente JavaScript.

L'adozione di TypeScript in un progetto di tale natura trova giustificazione in ragioni di carattere strutturale. In un dominio applicativo come quello della simulazione tattica, il modello dei dati si presenta ricco e articolato: entità militari caratterizzate da attributi geospaziali, temporali e di stato; messaggi di aggiornamento provenienti da sistemi esterni; configurazioni scenariali di notevole complessità. La tipizzazione statica consente di formalizzare tali modelli sotto forma di interfacce e tipi, rendendo espliciti i contratti intercorrenti tra i moduli del sistema e prevenendo, già in fase di compilazione, intere categorie di errori runtime (quali l'accesso a proprietà inesistenti, l'invocazione di funzioni con argomenti errati o la composizione di tipi incompatibili) che, in JavaScript puro, emergerebbero soltanto nel corso dell'esecuzione.

A ciò si aggiunga come TypeScript contribuisca in maniera significativa a migliorare l'esperienza di sviluppo: il supporto all'autocompletamento (IntelliSense), alla navigazione del codice e al refactoring assistito, offerto dagli ambienti di sviluppo moderni (quale, ad esempio, VS), si fonda interamente sulle informazioni di tipo, rendendo in tal modo la manutenzione di una codebase di grandi dimensioni notevolmente più efficiente.

2.4.4 CesiumJS: Rendering Geospaziale 3D su WebGL

L'elemento tecnologicamente più caratterizzante dell'intera applicazione è rappresentato da CesiumJS [23], libreria JavaScript open source destinata alla visualizzazione e all'analisi di dati geospaziali tridimensionali all'interno del browser, sviluppata da Cesium e resa disponibile nel 2011.

CesiumJS mette a disposizione un globo terrestre tridimensionale di elevata accuratezza, fondato su un modello ellissoidale WGS84, corredato di supporto nativo per il terreno tridimensionale ad alta risoluzione (Cesium World Terrain), di layer di immagini satellitari (imagery layers) provenienti da fornitori standard (Bing Maps, Mapbox, OpenStreetMap, WMS/WMTS), nonché della capacità di renderizzare modelli tridimensionali in formato glTF/glb e tilesets 3D Tiles (standard OGC concepito per lo streaming di contenuti geospaziali 3D di dimensioni considerevoli). Il motore di rendering risulta interamente basato su WebGL: l'intera pipeline grafica, dalla gestione del frustum di visione fino alla proiezione cartografica, viene eseguita sulla GPU per mezzo dell'API WebGL messa a disposizione dal browser, senza che sia necessario ricorrere a plugin o a componenti native.

Nell'ambito di un simulatore tattico, CesiumJS assolve a funzioni di natura critica:

- il posizionamento geospaziale preciso delle entità simulate, con coordinate espresse secondo sistemi di riferimento standard (WGS84 geografico, ECEF cartesiano, ENU locale);
- la visualizzazione dinamica di traiettorie, aree di copertura, settori di tiro e ulteriori primitive tattiche, rappresentate come entity Cesium (polilinee, poligoni, ellissoidi, label);
- la gestione del tempo simulato per mezzo del Clock interno di Cesium, il quale consente di sincronizzare l'avanzamento della simulazione con l'animazione delle entità geospaziali;
- l'interrogazione spaziale, che rende possibile associare eventi di interazione da parte dell'utente (click, hover) a specifiche entità presenti sulla scena tridimensionale.

Rispetto ad alternative quali Google Maps API, CesiumJS si configura come l'unica soluzione capace di coniugare un globo tridimensionale completo, un supporto nativo al terreno 3D, la gestione del tempo simulato e un ecosistema pensato esplicitamente per applicazioni mission-critical negli ambiti della difesa, dell'aerospazio e dell'intelligence. La libreria risulta inoltre distribuita con licenza Apache 2.0, il che ne garantisce la libertà d'uso tanto in contesti commerciali quanto in ambito governativo.

2.4.5 Tailwind CSS: Stilizzazione

La stilizzazione dell'interfaccia grafica è demandata a Tailwind CSS [24], un framework CSS di tipo utility-first pubblicato nel 2017, che rappresenta un vero e proprio cambio di paradigma rispetto all'approccio tradizionale, fondato sui fogli di stile semantici.

Il modello su cui si fonda Tailwind è di natura atomica: anziché definire classi CSS semantiche (si pensi, ad esempio, a .card) che raggruppano al proprio interno molteplici regole di stile, Tailwind mette a disposizione un insieme esaustivo di classi a responsabilità singola, ciascuna delle quali risulta associata a una singola proprietà CSS (flex, px-4). Lo stile di un componente viene dunque composto direttamente all'interno del markup JSX, combinando tali classi atomiche tra loro, eliminando così la necessità di scrivere e mantenere file CSS separati.

Un simile approccio comporta vantaggi di natura strutturale in un contesto quale quello del simulatore tattico. In primo luogo, Tailwind integra un sistema di design configurabile: palette cromatiche, scale di spaziatura, tipografia e breakpoint vengono definiti in maniera centralizzata e propagati automaticamente a tutte le classi generate, garantendo coerenza visiva senza per questo imporre una disciplina

manuale nella gestione delle variabili. In secondo luogo, il compilatore JIT (Just-in-Time) di Tailwind genera, in fase di build, esclusivamente le classi effettivamente utilizzate all'interno del codice sorgente, producendo un foglio di stile finale di dimensioni contenute, spesso pari a pochi kilobyte, senza che siano necessarie ulteriori configurazioni di purging. In terzo luogo, la co-locazione di struttura e stile all'interno del componente React riduce il context-switching cognitivo nel corso dello sviluppo e rende ciascun componente autonomo e riutilizzabile, senza dipendenze esterne da classi globali.

Tailwind non fornisce componenti predefiniti, né tantomeno impone un'estetica specifica, distinguendosi in tal senso nettamente da framework quali Bootstrap: la responsabilità delle scelte visive rimane pertanto interamente in capo al team di sviluppo, caratteristica che si rivela essenziale per applicazioni dotate di requisiti di interfaccia non convenzionali, come è appunto il caso dei simulatori tattici.

2.4.6 shadcn/ui: Componenti Interattivi

La realizzazione di componenti UI interattivi di natura complessa (quali dialog, dropdown, tooltip, sheet laterali, combobox) richiede una gestione accurata dell'accessibilità, della navigazione da tastiera e del comportamento del focus: aspetti, questi, la cui corretta implementazione ex novo risulta particolarmente onerosa e soggetta a regressioni. Per rispondere a tale esigenza si è fatto ricorso a shadcn/ui [25], una raccolta di componenti React riutilizzabili, sviluppata da Shadcn e distribuita con licenza MIT.

L'elemento architetturale che maggiormente contraddistingue shadcn/ui risiede nel fatto che non si tratta di una libreria in senso tradizionale: i componenti, infatti, non vengono installati come dipendenza npm né importati da un pacchetto esterno, bensì copiati direttamente all'interno del codice sorgente del progetto per mezzo di una CLI dedicata. Un simile approccio, definito dagli stessi autori con l'espressione "you own the components", comporta che il codice di ciascun componente risulti pienamente visibile, modificabile e versionato unitamente al resto dell'applicazione, eliminando così il vincolo di aggiornamenti forzati e consentendo personalizzazioni arbitrarie senza la necessità di sovrascrivere gli stili per mezzo di specifiche CSS più elevate.

I componenti di shadcn/ui poggiano su due fondamenta di natura tecnica. Sotto il profilo dell'accessibilità, essi si basano su Radix UI [26], libreria di primitive headless (ossia prive di uno stile proprio) che implementa i pattern di interazione definiti dalle specifiche WAI-ARIA, garantendo così la compatibilità con gli screen reader e una navigazione da tastiera conforme agli standard internazionali in materia di accessibilità. Sotto il profilo visivo, i componenti risultano interamente stilizzati per mezzo di Tailwind CSS e si avvalgono delle CSS Custom Properties ai fini

della gestione del tema (colori, bordi, raggi), consentendo l'adattamento dell'intera palette dell'interfaccia mediante la semplice modifica di un unico file di variabili, soluzione particolarmente utile ai fini dell'implementazione di temi scuri (dark mode), tipici delle interfacce operative in ambito difesa.

2.4.7 Protocollo WebSocket: Comunicazione Bidirezionale in Tempo Reale

Un simulatore tattico si configura, per sua stessa natura, come un sistema distribuito e operante in tempo reale: lo stato della simulazione (si pensi, ad esempio, alle posizioni delle unità, agli eventi di ingaggio, agli aggiornamenti dello scenario) viene generato da un motore di simulazione collocato lato server e necessita di essere propagato, con latenza minima, al client web incaricato della sua visualizzazione. Il protocollo HTTP tradizionale, fondato su un modello request-response di tipo stateless, non risulta adeguato a un simile caso d'uso: il polling periodico, infatti, introduce una latenza artificiale oltre a un considerevole overhead di connessione, mentre tecniche quali il long polling si rivelano fragili e difficilmente scalabili.

La soluzione adottata è rappresentata dal protocollo WebSocket [27], standardizzato dall'IETF nel 2011 e supportato nativamente da tutti i browser di ultima generazione per mezzo dell'API WebSocket. Tale protocollo stabilisce un canale di comunicazione full-duplex e di natura persistente su un'unica connessione TCP: a seguito di un handshake iniziale condotto su HTTP/HTTPS, la connessione rimane aperta, e sia il client sia il server risultano in grado di inviare messaggi in qualsiasi momento, senza dover sostenere il costo relativo all'apertura di una nuova connessione a ogni scambio di dati. La latenza di consegna risulta pertanto limitata alla sola latenza di rete, la quale si attesta tipicamente nell'ordine dei millisecondi su reti locali.

Nel contesto dell'applicazione, il canale WebSocket costituisce il vettore primario per la ricezione degli aggiornamenti relativi allo stato della simulazione: il server provvede a trasmettere frame JSON contenenti le posizioni aggiornate delle entità, i rispettivi attributi tattici e gli eventi di simulazione; il client React, dal canto suo, gestisce la connessione all'interno di un hook dedicato, procede alla deserializzazione dei messaggi ricevuti e aggiorna lo store di stato globale, il quale, a sua volta, notifica a CesiumJS l'aggiornamento delle posizioni delle entità sulla scena tridimensionale. La gestione del ciclo di vita della connessione (apertura, riconnessione automatica in caso di interruzione, chiusura controllata) risulta incapsulata all'interno del medesimo hook, mantenendo così il codice dei componenti React disaccoppiato dai dettagli propri del protocollo di trasporto.

Capitolo 3

Analisi dei Requisiti e Progettazione del Sistema

3.1 Ingegneria dei Requisiti e Modellazione del Sistema

L'ingegnerizzazione di un'interfaccia web per un simulatore tattico CGF necessita di una fase di analisi rigorosa, volta a tradurre le logiche operative proprie del dominio militare in specifiche architetturali per il frontend. Il presente capitolo descrive il processo di raccolta dei requisiti e ne definisce la progettazione sistemica, delineando la transizione dall'architettura originaria (AS-IS) verso la nuova soluzione (TO-BE).

3.1.1 Attori del Sistema

Il primo passo nella modellazione funzionale della nuova architettura web è costituito dall'individuazione formale degli attori. Secondo quanto previsto dallo standard UML, un attore è un'entità esterna (umana, hardware o software) che interagisce direttamente con i confini del sistema considerato. Nel caso della piattaforma di simulazione sviluppata, l'interfaccia grafica (GUI) non agisce in modo isolato, bensì costituisce un nodo di orchestrazione tra l'utente e le articolate logiche di calcolo e di rete.

L'analisi del dominio e dei requisiti ha permesso di identificare tre attori principali:

- **Operatore della Simulazione (Primary Human Actor):** indicato genericamente come "User" nel documento di specifica, corrisponde al personale specializzato (analisti tattici, istruttori o direttori di esercitazione) incaricato

della gestione del teatro operativo. Tale figura interagisce con il frontend React per creare e organizzare gerarchicamente le entità, coordinare la logica tattica mediante l'assegnazione di Task e Plan e controllare il flusso temporale della simulazione (ad esempio pausa, accelerazione, ripristino tramite snapshot).

- **Motore di Simulazione CGF / Backend (Primary System Actor):** rappresenta il nucleo computazionale (realizzato in C++) responsabile del calcolo della fisica, della cinematica e dell'intelligenza artificiale (Computer-generated force) dell'ambiente LVC. Dal punto di vista dell'architettura web, il backend si configura come un attore di sistema con il quale viene mantenuto un canale di comunicazione bidirezionale continuo: il frontend gli trasmette comandi operazionali (quali l'aggiunta di waypoint o l'inizializzazione di uno scenario) e ne riceve in cambio il flusso di telemetria asincrona necessario ad aggiornare il canvas di Cesium e la strumentazione analitica.
- **Rete di Simulazione Distribuita DIS/HLA (Secondary System Actor):** corrisponde al network di simulatori esterni confederati. Pur essendo la comunicazione a basso livello demandata all'engine C++, l'architettura dell'interfaccia web è stata concepita in modo da prevedere i moduli necessari alla configurazione e all'aggancio a tale rete (ad esempio l'impostazione dei protocolli e degli identificatori Site/Application ID). Guardando a possibili sviluppi futuri, una volta attivata la sessione il network assumerà il ruolo di attore attivo, in grado di iniettare dinamicamente nuove entità ed eventi nello scenario e di modificare così, in tempo reale, lo stato visualizzato dal frontend.

L'individuazione formale di tali attori chiarisce con precisione il ruolo assunto dall'interfaccia web sviluppata: un client avanzato e reattivo, concepito per tradurre l'intento tattico dell'Operatore in comandi destinati al motore CGF e, al contempo, per renderizzare gli ingenti flussi di dati provenienti dalla simulazione distribuita.

La definizione delle specifiche di sistema ha condotto alla stesura di un documento di requisiti formale, elaborato allo scopo di mappare in modo esaustivo le funzionalità che l'applicativo deve mettere a disposizione degli operatori della simulazione.

3.1.2 Requisiti Funzionali e Diagramma dei Package

I requisiti funzionali individuano le operazioni dirette che l'utente è in grado di compiere all'interno dell'ambiente virtuale. Data la complessità del dominio, tali funzionalità sono state suddivise in cinque macro-aree strategiche:

- **Gestione dello scenario e dell'ambiente:** comprende le funzionalità dedicate alla creazione, al salvataggio e al caricamento degli scenari, inclusa la validazione strutturale precedente all'esecuzione e la modifica dinamica delle condizioni meteorologiche.

- **Ciclo di vita delle entità:** raggruppa gli strumenti destinati all'istanziazione, alla clonazione e alla rimozione delle entità, oltre che alla configurazione avanzata di sensori, armamenti e gerarchie tattiche (fazioni).
- **Logica tattica e scripting:** include i meccanismi per l'assegnazione di ordini complessi, articolati attraverso la creazione di Task, Set e Plan, che permettono di coordinare il comportamento delle entità in risposta a inneschi ambientali (Task reattivi).
- **Controllo della simulazione:** riguarda le funzionalità di gestione del tempo di esecuzione, quali l'avvio, la messa in pausa, l'accelerazione temporale e il ripristino (rewind) a stati globali precedenti mediante snapshot.
- **Interoperabilità e osservabilità:** comprende i moduli relativi all'accesso protetto (autenticazione), alla consultazione di dashboard analitiche e all'impostazione dei parametri di connessione ai network di simulazione distribuiti attraverso protocolli standard DIS o HLA.

Al fine di rappresentare visivamente l'architettura dei requisiti e di delineare con chiarezza i confini del sistema (System Boundary), è stato realizzato un diagramma UML dei Package (Figura 3.1). Tale approccio top-down consente di astrarre dalla complessità dei singoli casi d'uso, mettendo in evidenza in maniera immediata le modalità con cui gli attori precedentemente definiti interagiscono con i cinque moduli architetturali principali dell'interfaccia web.

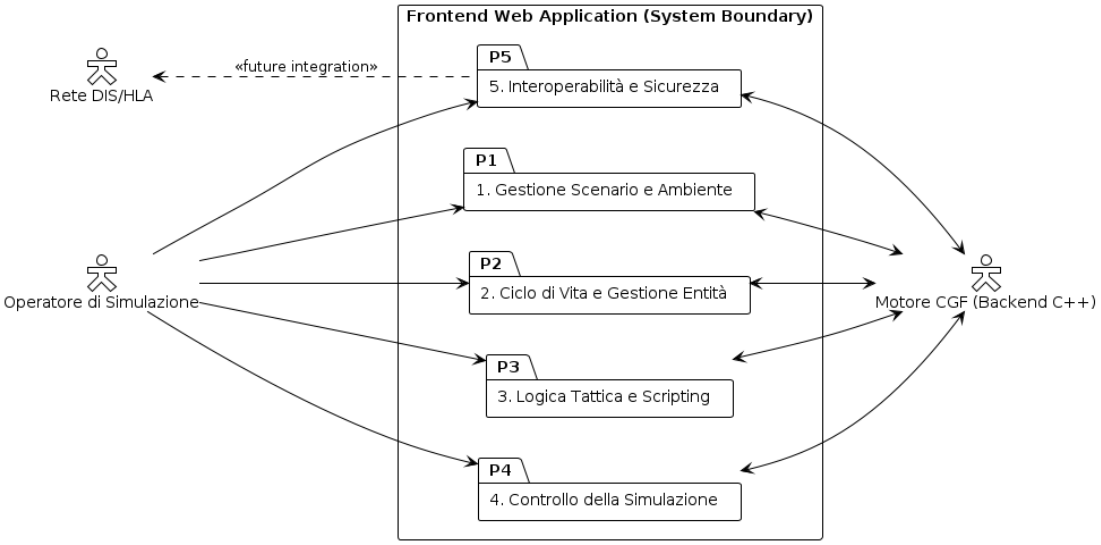


Figura 3.1: Diagramma UML dei Package. Illustra il perimetro dell'applicazione web e le interazioni di alto livello tra l'operatore, il motore C++ e i futuri nodi di rete DIS/HLA con i vari moduli funzionali.

Considerata l'ampiezza delle specifiche, nei paragrafi successivi si tralascerà la modellazione UML completa relativa a ciascun modulo (consultabile nella documentazione tecnica di progetto), per concentrarsi invece sui soli casi d'uso di maggior rilievo e complessità ai fini dello sviluppo dell'interfaccia React.

3.1.3 Analisi dei Sottosistemi di Rilievo

Come anticipato dal diagramma architetturale, il sistema si articola in molteplici moduli. Allo scopo di delineare le sfide ingegneristiche che hanno orientato lo sviluppo del frontend in React, si presenta di seguito l'analisi approfondita dei due sottosistemi caratterizzati dalla maggiore complessità interattiva. La modellazione UML completa dei restanti moduli (Controllo del Tempo, Rete e Gestione Scenari) è integralmente consultabile in Appendice A.

Modulo: Gestione e Ciclo di Vita dell'Entità Questo sottosistema disciplina la presenza e le proprietà delle Forze Computer-Generate (CGF) all'interno del teatro operativo. Dal punto di vista dell'interfaccia utente, esso costituisce uno dei moduli a maggiore densità informativa, in quanto richiede un'interazione continua tra i pannelli e la mappa.

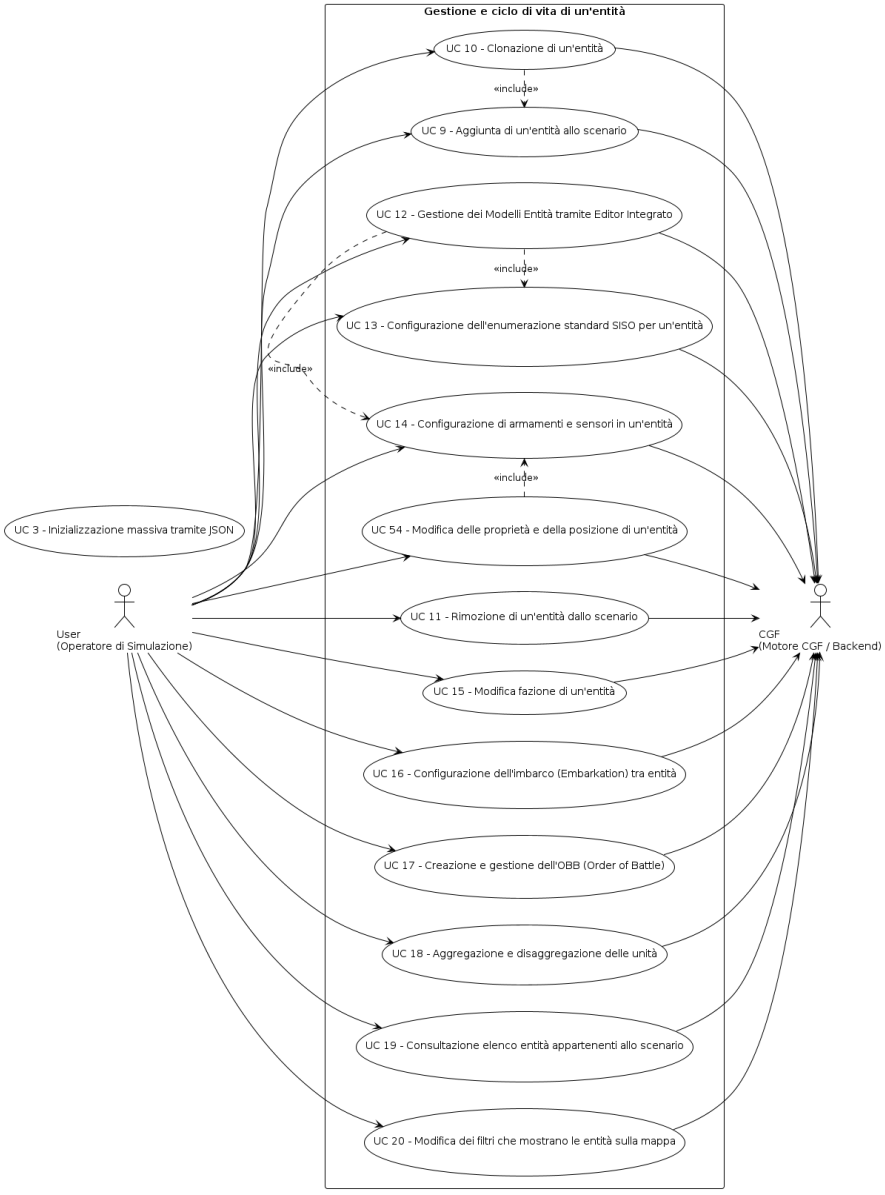


Figura 3.2: Diagramma dei Casi d'Uso relativo al package "Gestione e ciclo di vita di un'entità".

Come mostrato in Figura 3.2, l'Operatore dispone di un controllo granulare sulle unità. Le funzionalità di core business implementate nell'attuale iterazione del frontend riguardano principalmente la manipolazione diretta della simulazione: l'aggiunta (UC 9), la clonazione (UC 10) e la rimozione (UC 11) delle entità necessitano di un'interazione fluida con il canvas di CesiumJS ai fini della designazione spaziale. Allo stesso tempo, la modifica delle proprietà (UC 54) e la consultazione/filtraggio

dell'elenco entità (UC 19, UC 20) richiedono la realizzazione di form e liste reattive in grado di gestire il flusso di dati asincrono proveniente dal Motore CGF senza compromettere il thread di rendering visivo.

Modulo: Logica Tattica, Task e Scripting

Questo sottosistema è dedicato all'assegnazione degli ordini e alla pianificazione dei percorsi logici e spaziali.

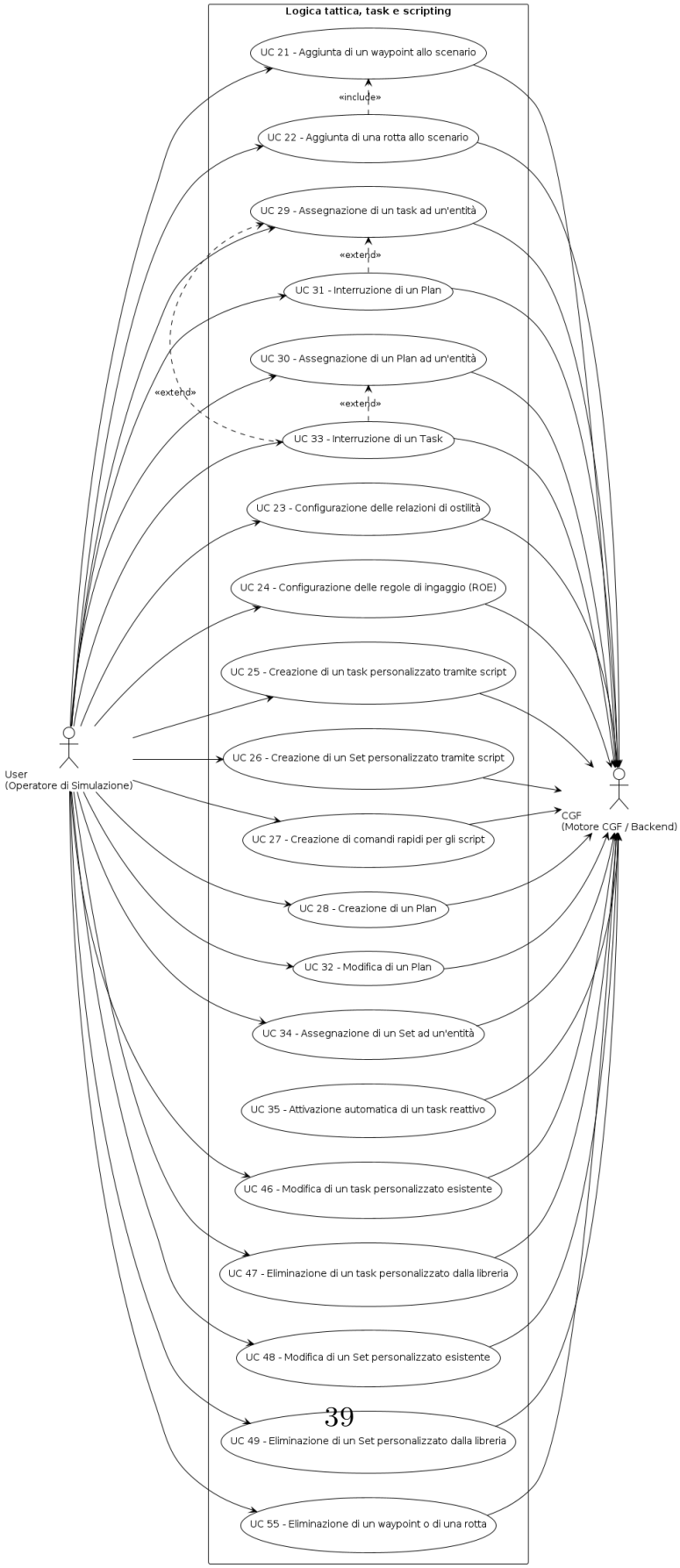

Logica tattica, task e scripting
UC 21 - Aggiunta di un waypoint allo scenario
«include»
UC 22 - Aggiunta di una rotta allo scenario
UC 29 - Assegnazione di un task ad un'entità
«extend»
UC 31 - Interruzione di un Plan
UC 30 - Assegnazione di un Plan ad un'entità
«extend»
«extend»
UC 33 - Interruzione di un Task
UC 23 - Configurazione delle relazioni di ostilità
UC 24 - Configurazione delle regole di ingaggio (ROE)
UC 25 - Creazione di un task personalizzato tramite script
UC 26 - Creazione di un Set personalizzato tramite script
UC 27 - Creazione di comandi rapidi per gli script
UC 28 - Creazione di un Plan
UC 32 - Modifica di un Plan
UC 34 - Assegnazione di un Set ad un'entità
UC 35 - Attivazione automatica di un task reattivo
UC 46 - Modifica di un task personalizzato esistente
UC 47 - Eliminazione di un task personalizzato dalla libreria
UC 48 - Modifica di un Set personalizzato esistente
UC 49 - Eliminazione di un Set personalizzato dalla libreria
UC 55 - Eliminazione di un waypoint o di una rotta
User
(Operatore di Simulazione)
CGF
(Motore CGF / Backend)

Figura 3.3: Diagramma dei Casi d'Uso relativo al package "Logica Tattica e Scripting".

La complessità di questo modulo (Figura 3.3) deriva dalla necessità di tradurre l'intento tattico dell'operatore in coordinate geografiche e direttive logiche interpretabili dal backend. Lo sviluppo si è concentrato principalmente sulla navigazione spaziale, consentendo all'operatore di aggiungere in modo interattivo Waypoint (UC 21) e Rotte (UC 22), oppure di rimuoverli (UC 55). La realizzazione di tali casi d'uso ha reso necessaria l'ingegnerizzazione di chiamate HTTP transazionali verso le API del backend, indispensabili per modificare lo stato della simulazione in memoria e riflettere istantaneamente i nuovi tracciati strategici sulla mappa.

3.1.4 Requisiti Non Funzionali e Metriche di Riferimento

La progettazione dell'architettura frontend non è stata orientata unicamente dalle esigenze operative dell'utente, ma è risultata fortemente condizionata dai requisiti non funzionali e prestazionali imposti a livello di sistema. Le specifiche tecniche di progetto, concordate con gli stakeholder aziendali, definiscono precise metriche di efficienza, affidabilità e latenza.

Dall'analisi dei vincoli progettuali emergono le seguenti direttive architetturali, che hanno inciso direttamente sulla modellazione dell'interfaccia in React e Cesium:

- **Frequenza di Aggiornamento e Latenza Visiva:** il livello di presentazione web è tenuto ad aggiornare la posizione e lo stato delle entità assicurando una fluidità minima di 30 FPS. Poiché il motore di backend distribuisce gli aggiornamenti di telemetria a intervalli predefiniti, spetta al frontend elaborare i pacchetti in arrivo entro la conclusione del frame corrente, così da minimizzare la latenza di visualizzazione. L'interfaccia, inoltre, deve implementare logiche di interpolazione spaziale delle coordinate tra un pacchetto dati e il successivo, così da evitare "scatti" visivi durante l'osservazione della mappa tattica.
- **Sincronizzazione Temporale e Flusso di Esecuzione:** il motore di calcolo (Runtime) adotta uno scheduler ad alta frequenza fissa, compresa indicativamente tra 50Hz e 100Hz, per l'aggiornamento dello stato delle entità. La GUI web deve poter agganciarsi a tale flusso continuo e, soprattutto, sostenere lo svincolo dal tempo reale: il sistema deve essere in grado di tollerare e renderizzare esecuzioni simulate accelerate (ad esempio 10x o 100x). Tale vincolo impone un'ottimizzazione critica della gestione dello stato in React, allo scopo di evitare colli di bottiglia nel main thread del browser.
- **Gestione del Rendering e Ottimizzazione del Carico:** dato l'elevato carico computazionale generato dalla coesistenza tra un'interfaccia composta da elementi complessi (menu, form e dashboard) e il contesto spaziale, il

motore geospaziale (CesiumJS) deve necessariamente implementare algoritmi di LOD (Level of Detail). Tale approccio garantisce che il caricamento della cartografia vettoriale e dei modelli avvenga senza penalizzare il framerate complessivo dell'applicativo.

- **Robustezza e Validazione dei Controlli:** l'interfaccia di controllo web non svolge il ruolo di terminale passivo, bensì costituisce il primo livello di barriera (gatekeeper) del sistema: è tenuta a impedire l'inserimento o la propagazione di valori fisicamente impossibili o fuori range (ad esempio massa negativa, oppure altitudini incompatibili con il dominio di un veicolo terrestre). Essa deve inoltre fornire all'operatore un riscontro visivo immediato sulle violazioni dei vincoli, ancora prima che la validazione venga demandata al backend.

Tali requisiti quantitativi mettono in luce la duplice sfida ingegneristica affrontata nel frontend: da un lato, comportarsi come un'applicazione gestionale finalizzata all'immissione sicura di dati e logiche; dall'altro, funzionare come un motore di rendering real-time in grado di ricevere centinaia di aggiornamenti telemetrici al secondo senza compromettere l'esperienza utente.

3.2 Design dell'Esperienza Utente (UX) e Transizione Architetturale

L'interfaccia di un simulatore tattico CGF costituisce un sistema critico dal punto di vista dell'interazione Uomo-Macchina (HCI - Human-Computer Interaction). L'operatore è sottoposto a un elevato carico cognitivo, dovendo monitorare contemporaneamente la topologia del terreno e un flusso continuo di dati telemetrici e logici. La progettazione dell'interfaccia deve pertanto mirare a un netto disaccoppiamento visivo tra il contesto spaziale (la mappa) e il livello decisionale (i pannelli operativi).

Al momento dell'avvio delle attività di tirocinio, il progetto versava in uno stato embrionale: lo stack tecnologico di base (React, TypeScript e Cesium) era già stato individuato per rispondere ai requisiti di rendering geospaziale, e alcune funzionalità visive elementari erano già state abbozzate. Tuttavia, per orientare l'evoluzione del software verso una piattaforma completa e strutturata, si è resa necessaria una rigorosa fase iniziale di Ingegneria dei Requisiti.

Tale fase di analisi ha condotto alla stesura di un documento formale di Specifica dei Casi d'Uso (Use Case Specification), fondamentale per mappare sia le funzionalità preesistenti sia, soprattutto, le numerose funzionalità logiche e operative ancora da implementare. Il documento elaborato comprende oltre 50 casi d'uso, coprendo l'intero ciclo di vita della simulazione.

L'analisi di tale documento ha rappresentato lo spartiacque del progetto: essa ha infatti messo in luce come l'ingente mole di interazioni richieste all'operatore non potesse essere sostenuta dall'approccio stilistico adottato in origine (basato su CSS puro). Per illustrare questa criticità, si riportano di seguito tre casi d'uso emblematici che hanno reso manifesta la necessità di riprogettare l'architettura dell'interfaccia grafica (GUI):

1. **Gestione dei Modelli Entità (Use Case 12):** la creazione e la modifica di un modello personalizzato richiedono un editor integrato in grado di esporre molteplici domini di parametri (cinematica, sensori, classificazione SISO). Tale caso d'uso comporta la realizzazione di form complessi dotati di validazione in tempo reale, requisito che rende il CSS puro e i componenti React non standardizzati estremamente onerosi da manutenere.
2. **Consultazione avanzata tramite dashboard (Use Case 42):** il sistema deve aggregare in tempo reale i dettagli operativi di un'entità e il relativo registro degli avvisi, consentendo il filtraggio dinamico per livello di criticità. L'implementazione di overlay informativi (dashboard, tabelle dati, pop-up) sovrapposti al globo WebGL richiede componenti UI altamente accessibili e strutturati.
3. **Pianificazione logica (Use Case 28 e 32):** la creazione e la modifica di un Plan (sia globale che individuale) richiedono all'operatore la costruzione di una gerarchia di blocchi logici (Task, Set, Condizioni). Dal punto di vista del frontend, ciò si traduce in interfacce complesse (quali menu contestuali, liste riordinabili e pannelli nidificati) la cui realizzazione necessita di librerie di componenti avanzate al fine di garantire un'esperienza utente fluida.

Questa disamina ha reso evidente come il software, per poter scalare e accogliere le funzionalità previste dal documento di requisiti, necessitasse di una transizione architetturale del livello di presentazione visiva.

3.2.1 I Limiti dell'Architettura di Partenza (AS-IS)

Al momento della presa in carico del progetto, il sistema disponeva di una base web nella quale l'interazione spaziale era affidata a CesiumJS, mentre l'infrastruttura dell'interfaccia utente (menu, overlay e pannelli) veniva gestita mediante l'impiego di codice CSS puro. La fase di analisi dei requisiti ha messo in luce come tale approccio non fosse in grado di scalare per supportare le nuove specifiche. In particolare, sono emersi tre limiti architetturali:

1. **Frizione nella realizzazione di interfacce complesse:** la gestione logica degli scenari (quale la configurazione dei Task o la consultazione della Dashboard delle entità) richiede form dinamici, menu nidificati e tabelle filtrabili.

La realizzazione di tali strutture da zero in CSS puro comportava una mole di codice difficilmente manutenibile.

2. **Assenza di incapsulamento (Scope Globale):** l'impiego di fogli di stile tradizionali esponeva l'applicazione al rischio di collisioni tra le classi al crescere del numero dei componenti React, rendendo pericoloso il refactoring in un ambiente in continua evoluzione.
3. **Mancanza di standardizzazione visiva:** l'assenza di un Design System centralizzato generava un'esperienza utente frammentata, con il rischio di incrementare i tempi di reazione dell'operatore nel corso delle sessioni di simulazione.

3.2.2 La Soluzione Progettuale (TO-BE)

Al fine di garantire una scalabilità a lungo termine e di soddisfare i requisiti funzionali emersi dalla modellazione dei casi d'uso, si è scelto di adottare un radicale cambio di paradigma nella progettazione del Presentation Layer. La nuova architettura TO-BE si fonda sull'introduzione di due standard di settore: Tailwind CSS e Shadcn/UI.

- **Il Paradigma Utility-First:** l'abbandono del CSS tradizionale a favore di framework utility-first come Tailwind consente di mappare direttamente lo stile all'interno del componente logico React. Ciò azzera i conflitti di naming e permette al contempo di alleggerire il bundle finale, ottimizzando i tempi di caricamento dell'interfaccia.
- **Componentizzazione Accessibile:** la necessità di fornire all'operatore un insieme di controlli sicuri e visivamente coerenti è stata risolta mediante l'adozione della filosofia dei componenti modulari componibili (headless UI), di cui Shadcn/UI costituisce lo stato dell'arte. In tal modo, elementi critici quali i menù di selezione (impiegati per l'assegnazione dei piani tattici) o i pannelli a scorrimento (destinati alla visualizzazione degli ordini di battaglia) sono stati standardizzati, garantendo accessibilità nativa e isolamento del codice.

3.3 High-Level Design (Progettazione Architetturale)

La transizione dai requisiti di dominio alla realizzazione del sistema richiede la definizione di un'architettura di alto livello (High-Level Design). L'obiettivo primario nella progettazione del client web è consistito nel gestire l'elevata complessità informativa e il carico computazionale tipico di una simulazione CGF, garantendo al tempo stesso un'elevata reattività dell'interfaccia utente.

Per soddisfare tali requisiti, l'architettura è stata strutturata secondo il paradigma della Separation of Concerns (Separazione delle Competenze), disaccoppiando nettamente il livello logico-presentazionale dal motore di rendering geospaziale.

3.3.1 Disaccoppiamento tra Presentation Layer e Rendering Geospaziale

La principale sfida architetturale nello sviluppo di una Graphical User Interface (GUI) per simulatori LVC su tecnologie web risiede nella convivenza forzata tra due paradigmi di rendering profondamente differenti:

- **Il Virtual DOM di React**, che gestisce lo stato dell'interfaccia e ottimizza gli aggiornamenti dell'albero dei componenti HTML.

- **Il Canvas WebGL di CesiumJS**, che opera attraverso un ciclo di rendering continuo (game loop) per il calcolo spaziale.

Consentire a React di gestire direttamente le primitive WebGL di Cesium comporterebbe un degrado inaccettabile delle prestazioni (con conseguente perdita di framerate), dal momento che ogni mutazione di stato dell'interfaccia innescherebbe aggiornamenti ridondanti sul globo. Per ovviare a tale criticità, il sistema è stato progettato in modo che Cesium operi come un ecosistema incapsulato. Il frontend React comunica con il motore spaziale unicamente attraverso astrazioni dedicate (quali i MapAdapter), le quali ricevono i dati di telemetria e aggiornano le primitive vettoriali (rotte, waypoint, billboard delle entità) agendo direttamente sulle API imperative di Cesium, aggirando completamente il ciclo di re-rendering di React.

3.3.2 Architettura a Strati del Client Web

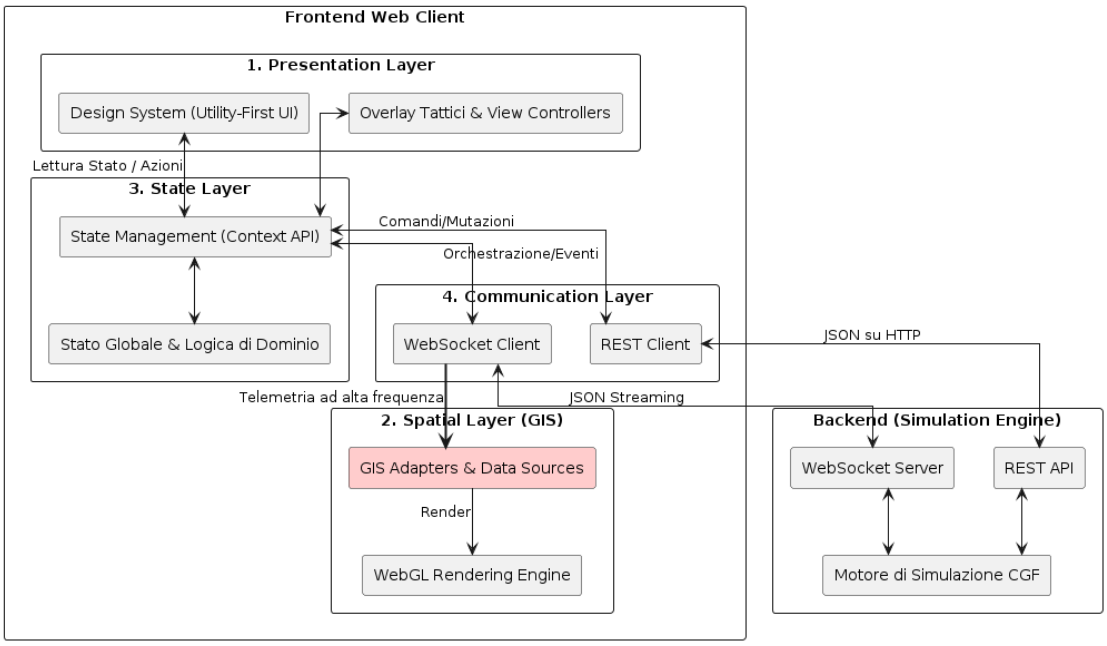


Figura 3.4: Rappresentazione dell'architettura a strati del client web e delle interfacce di comunicazione con il backend di simulazione.

Per assicurare manutenibilità e scalabilità, l'architettura interna dell'applicazione frontend è stata organizzata in quattro macro-livelli logici:

1. **Layer di Presentazione (UI/UX):** costituisce il livello più esterno, responsabile dell'interazione con l'operatore. Si avvale di Tailwind CSS per uno styling a basso impatto prestazionale e di Shadcn/UI per la realizzazione di componenti complessi, accessibili e standardizzati. Tale layer ospita gli overlay tattici, i form di controllo (come lo ScenariosController) e le dashboard informative, sovrapponendosi visivamente alla mappa senza interferire con esso.

2. **Layer Spaziale (GIS Engine):** gestito da CesiumJS, questo strato riceve in ingresso le coordinate spaziali dal layer di stato e si occupa esclusivamente della loro traduzione visiva. Gestisce la topologia, i modelli, l'estrapolazione dei vettori di movimento (velocità e rotta) e la simbologia tattica militare.

3. **Layer di Stato (State Management):** rappresenta il "cervello" del frontend. Sfrutta le funzionalità avanzate di React (quali le Context API e Custom Hook dedicati, ad esempio useCgfSession) per mantenere l'unica fonte di verità (Single Source of Truth) dello stato dell'applicativo. Tale livello memorizza

l'elenco degli scenari disponibili, le entità attualmente selezionate, i filtri di visibilità per l'ordine di battaglia e la condizione temporale della simulazione.

4. **Layer di Comunicazione (Network):** costituisce l'interfaccia verso il mondo esterno (il motore C++ di backend). Questo strato astrae la complessità della rete e garantisce la sincronizzazione bidirezionale dei dati tra client e server, instradando i flussi informativi verso il layer di stato oppure direttamente verso il layer spaziale.

3.3.3 Modello Architetturale di Rete Ibrido (WebSocket + REST)

Un elemento qualificante del design architetturale risiede nella scelta del protocollo di trasporto. La varietà e la natura dei dati scambiati tra il client web e il motore di simulazione C++ hanno reso inadeguata l'adozione di un unico protocollo, orientando la progettazione verso un'architettura ibrida.

- **Traffico ad Alta Frequenza (WebSocket):** il protocollo WebSocket (connessione full-duplex persistente) è stato riservato esclusivamente allo streaming continuo della telemetria. Il backend C++ elabora la cinematica delle entità e pubblica aggiornamenti di stato a frequenza fissa (ad esempio 20 Hz). Il layer di comunicazione del frontend riceve tali payload JSON (contenenti coordinate, heading, altitudine e velocità) e li inietta direttamente nel layer spaziale per aggiornare le entità, riducendo al minimo l'overhead di rete e assicurando una visualizzazione fluida del teatro operativo.
- **Traffico Transazionale (HTTP/REST):** per le operazioni di manipolazione dello stato logico o per l'inserimento di nuove direttive tattiche, l'architettura ricorre al paradigma HTTP/REST. Azioni discrete, deterministiche e puntuali, come la creazione manuale di un Waypoint da parte dell'operatore o l'invio di un Task specifico a un'entità, vengono trasmesse tramite richieste HTTP POST o PUT verso le API esposte dal backend. Tale approccio garantisce l'affidabilità della consegna (assente nei flussi di streaming continui) e consente di sfruttare i codici di stato HTTP per validare l'accettazione del comando prima di rifletterne il risultato nel layer di stato del frontend.

3.4 Pipeline di Configurazione e Orchestrazione degli Scenari

Affinché la simulazione tattica possa essere avviata, il sistema deve gestire il caricamento della configurazione iniziale del teatro operativo. Tale procedura

coinvolge l'intera architettura, dal livello di build dei file di configurazione fino all'interfaccia utente in React.

3.4.1 Il formato dei file di scenario (.epp ed .edl) e Automazione della Build

Nel dominio dei simulatori CGF trattati, la definizione di uno scenario si fonda su un approccio a due livelli. L'operatore o lo sviluppatore definisce la struttura dello scenario redigendo manualmente file con estensione .epp. Tali file sorgente, facilmente leggibili e modificabili, contengono le macro-definizioni e i riferimenti ai componenti della simulazione (come reti e modelli). Prima di poter essere eseguiti dal motore in C++, i file .epp devono essere pre-processati e compilati in file .edl. Il file .edl costituisce la configurazione risolta e pronta per l'iniezione nella memoria del motore di simulazione.

Per consentire all'interfaccia web di interrogare dinamicamente gli scenari disponibili, si è reso necessario un intervento strutturale sulla build chain del progetto. Lo script di compilazione (batch) responsabile della generazione dei file .edl è stato modificato e ottimizzato. Lo script automatizzato esplora le directory di configurazione e separa logicamente la compilazione degli scenari standard (impiegati dall'applicativo) da quelli destinati al testing. Tutti i file .epp validi (identificati dal prefisso scenario_) vengono compilati mediante preprocessore, e gli artefatti finali .edl vengono convogliati all'interno di una cartella input/ centralizzata. Tale standardizzazione dell'output costituisce un prerequisito fondamentale per consentire al backend di indicizzare dinamicamente le configurazioni disponibili per il client web.

3.4.2 Orchestrazione Asincrona e Inizializzazione dello Scenario

La responsabilità del client React non risiede nel parsing diretto dei file di configurazione, bensì nell'orchestrazione del caricamento e nella gestione dello stato asincrono mediante protocollo WebSocket. A tale scopo è stato implementato il componente modulare ScenariosController.

L'architettura progettata prevede una rigida separazione delle responsabilità tra il Presentation Layer (l'interfaccia React e il rendering di Cesium) e la logica di dominio (il motore C++ di backend). Per assicurare un'esperienza utente reattiva in un sistema intrinsecamente distribuito, è stato necessario modellare con attenzione il flusso dei dati asincroni.

Il diagramma di sequenza (Figura 3.5) illustra il processo di caricamento, il quale si affida interamente a una comunicazione asincrona bidirezionale:

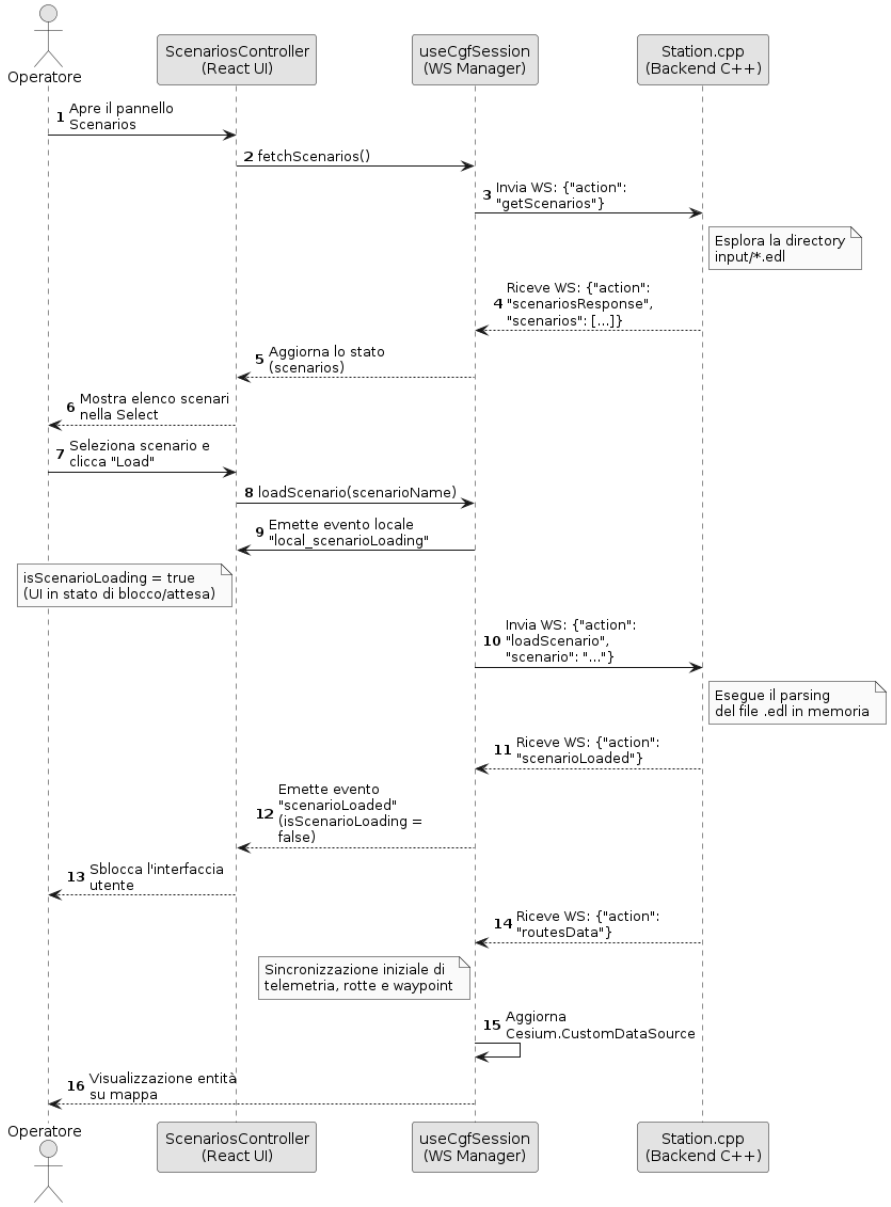


Figura 3.5: Diagramma di Sequenza per l'orchestrazione e l'inizializzazione dello scenario.

Il flusso si articola come segue:

1. **Indicizzazione:** all'apertura del pannello, il frontend invia un comando getScenarios attraverso un socket di controllo condiviso. Il backend in C++

intercetta la richiesta, esplora la directory input/ alla ricerca dei file .edl validi e restituisce un payload JSON contenente l'elenco degli scenari disponibili.

2. **Gestione dell'Interfaccia UI:** il componente ScenariosController riceve il payload e popola l'interfaccia, offrendo all'operatore funzionalità avanzate di ricerca testuale e selezione filtrata. L'adozione di componenti accessibili (quali Select e Input) assicura una User Experience fluida anche in presenza di un numero elevato di scenari.

3. **Trigger di Caricamento:** una volta selezionato lo scenario desiderato, il frontend emette l'evento loadScenario. Tale comando istruisce il backend C++ a eseguire il parsing effettivo del file .edl associato e a reinizializzare la logica di simulazione.

4. **Sincronizzazione dello Stato:** durante l'attesa, il frontend gestisce in modo reattivo lo stato di caricamento (isScenarioLoading), disabilitando le interazioni conflittuali. Al termine del caricamento, il backend notifica l'avvenuto successo mediante l'evento scenarioLoaded, attivando la sincronizzazione dei dati tattici (come rotte e telemetria delle entità) che verranno successivamente renderizzati su CesiumJS.

3.4.3 Editing Dinamico e Pattern Pub/Sub (Creazione Waypoint)

Sebbene la fase di inizializzazione si basi su file statici precompilati, l'architettura TO-BE è stata concepita per supportare la progressiva transizione verso un editing bidirezionale. La gestione di oggetti tattici, come l'aggiunta di Waypoint o rotte (Use Case 21), sta evolvendo dal puro data-entry nei file .epp verso una manipolazione dinamica tramite interfaccia web. Il diagramma di sequenza (Figura 3.6) illustra l'approccio architetturale ibrido adottato per tale casistica: l'aggiunta di riferimenti spaziali richiede garanzie transazionali che lo streaming WebSocket non offre nativamente. Si ricorre pertanto a una chiamata HTTP/REST abbinata a un pattern Publisher/Subscriber locale.

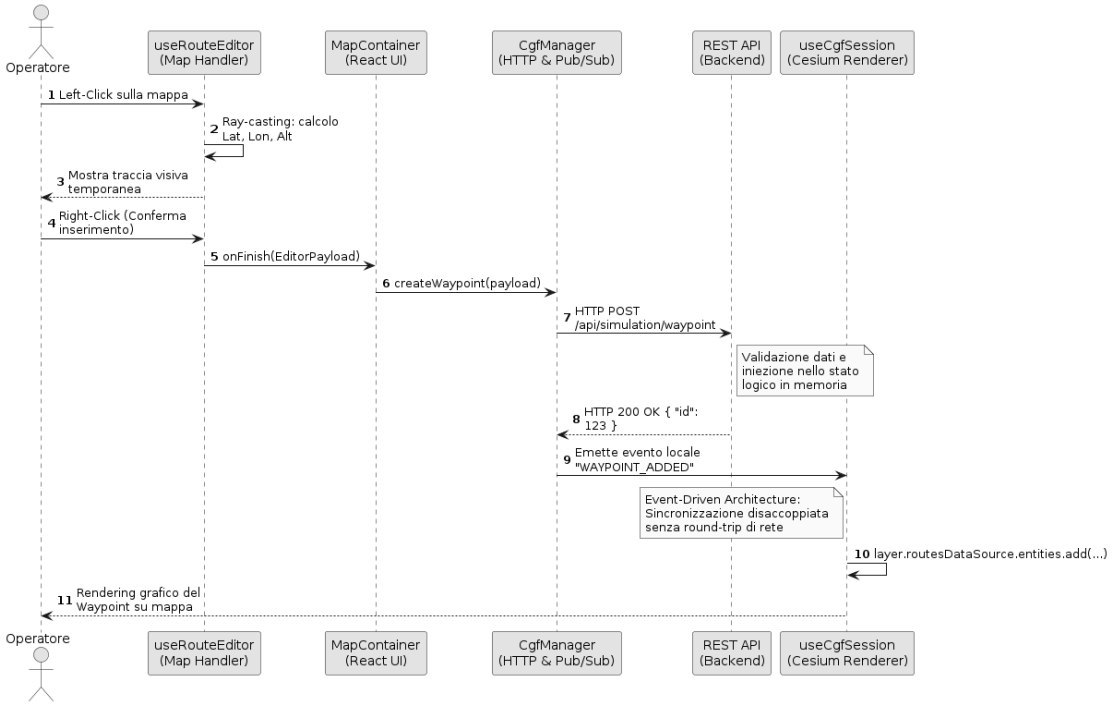


Figura 3.6: Diagramma di Sequenza ibrido (REST + Pub/Sub) per l'aggiunta dinamica di un Waypoint.

1. **Interazione Geospaziale:** l'utente accede alla modalità editing e definisce un punto sulla mappa. Il modulo useRouteEditor intercetta l'evento di click del mouse e decodifica le coordinate spaziali del terreno (Latitudine, Longitudine, Altitudine) selezionato mediante il ray-casting della telecamera di CesiumJS.

2. **Validazione e Chiamata Transazionale:** una volta confermato l'inserimento (tramite right-click), il componente MapContainer estrae il payload e invoca il CgfManager per istanziare l'oggetto. Il Manager effettua una richiesta HTTP POST (ad esempio /api/simulation/waypoint) verso il servizio REST del backend. Il ricorso al protocollo HTTP garantisce l'ottenimento di un acknowledgement sincrono, oltre alla gestione strutturata degli eventuali errori (ad esempio HTTP 400 Bad Request).

3. **Event Emitter Locale:** al ricevimento di un codice di successo (HTTP 200 OK), il backend inietta l'oggetto nella memoria della simulazione. A questo punto, allo scopo di evitare un ulteriore round-trip di rete (come l'attesa di un nuovo invio globale tramite WebSocket), il CgfManager notifica immediatamente il successo all'architettura frontend, emettendo un evento locale (WAYPOIN_ADDED) attraverso il proprio bus interno.

4. **Disaccoppiamento del Rendering:** il modulo useCgfSession, già sottoscritto agli eventi del Manager, intercetta l'evento WAYPOINT_ADDED. Avendo ottenuto la garanzia della persistenza in memoria, preleva le coordinate dal payload dell'evento e istruisce autonomamente la CustomDataSource di CesiumJS, materializzando visivamente l'entità sulla cartografia.

Questa topologia mista garantisce un'elevata responsività lato client senza rinunciare all'integrità transazionale richiesta per le operazioni di mutazione dello stato.

Capitolo 4

Implementazione del Presentation e Spatial Layer

4.1 Architettura di Comunicazione nel Client React

Come emerso dall'analisi condotta in fase di progettazione (Capitolo 3), il sistema necessita di una gestione ibrida del traffico di rete, articolata su due canali distinti: uno ad alta frequenza destinato alla telemetria continua (WebSocket) e uno di natura transazionale riservato ai comandi logici (REST). Recepire tale requisito all'interno dell'ecosistema React ha reso necessaria la progettazione di moduli appositi, con l'obiettivo di mantenere separata la logica di rete dal livello puramente presentazionale (UI).

4.1.1 Gestione della Telemetria ad Alta Frequenza (WebSocket)

La criticità principale che si pone nell'integrare flussi di dati ad alta frequenza (fino a 20 aggiornamenti al secondo) in un'applicazione React risiede nel meccanismo del Virtual DOM. Qualora le coordinate telemetriche in arrivo venissero memorizzate nello stato globale di React (per mezzo di useState o Redux), si innescherebbero cicli di re-rendering continui e massivi a carico dell'intera interfaccia, con conseguente saturazione, in tempi rapidi, del main thread del browser.

Per ovviare a tale criticità, il ciclo di vita della connessione WebSocket è stato incapsulato in un custom hook dedicato, denominato useCgfSession, il quale funge

da controllore asincrono e persegue due finalità principali:

1. **Ricezione e Instradamento Diretto:** l'hook conserva in memoria il riferimento al socket nativo del browser; non appena giunge un payload, il metodo handler (handleSocketMessage) provvede al parsing del JSON (estraendo ID, Latitudine, Longitudine, Heading, ecc.) e inoltra i dati direttamente al motore spaziale attraverso la funzione upsertTrack. In tal modo l'aggiornamento dei parametri fisici bypassa completamente il ciclo vitale di React, preservando così le prestazioni di rendering.
2. **Resilienza e Consistenza dei Dati:** dato che nel dominio LVC (Live, Virtual, Constructive) le entità possono andare incontro a disconnessioni, useCgfSession integra appositi algoritmi di robustezza. È stata implementata una logica di riconnessione automatica (scheduleReconnect), attivata in caso di caduta del collegamento. Il sistema, inoltre, ricorre a una procedura di garbage collection basata su criteri temporali (cleanupStaleTracks): qualora un'entità non riceva aggiornamenti telemetrici per un periodo superiore a 30 secondi (TRACK_STALE_MS), essa viene considerata obsoleta e rimossa in maniera sicura sia dalla mappa sia dalle strutture dati locali.

4.1.2 Il Manager API e il Pattern Publisher/Subscriber per il Traffico Transazionale

A differenza della telemetria, che costituisce un flusso di natura passiva, le interazioni dell'operatore (quali l'avvio della simulazione o la creazione di una rotta) danno origine a un traffico transazionale di tipo attivo. Al fine di evitare che la logica delle chiamate di rete risultasse frammentata e ripetuta all'interno dei diversi pannelli e menu della UI, è stata progettata un'unica classe di controllo, il CgfManager.

Tale classe è stata realizzata seguendo il Design Pattern Singleton (getInstance()), in modo da assicurare che l'intera applicazione condivida una singola istanza per la configurazione degli endpoint REST e per la gestione degli errori HTTP.

La sfida architetturale di maggior rilievo, in questo modulo, ha riguardato la sincronizzazione tra lo stato visivo e l'accettazione dei comandi da parte del backend in C++. Con l'obiettivo di eliminare round-trip di rete non necessari (vale a dire attendere che un waypoint appena creato dal client compaia nel successivo pacchetto WebSocket globale prima di poterlo renderizzare), all'interno del CgfManager è stato sviluppato un bus di eventi locale, basato sul pattern Publisher/Subscriber.

Il funzionamento si articola in tre fasi:

- Un componente React invoca il manager affinché esegua un'azione (ad esempio createWaypoint, tramite chiamata HTTP POST).

- Qualora il backend C++ risponda con esito positivo, restituendo l'ID della nuova entità, il Manager assume il ruolo di Publisher ed emette immediatamente un evento logico interno (ad esempio WAYPOINT_ADDED), contenente il payload del nuovo elemento.
- L'hook useCgfSession, fungendo da Subscriber, intercetta l'evento ed effettua il rendering grafico in maniera immediata (Data Binding ottimistico), ancor prima che il dato giunga attraverso il normale stream telemetrico.

Questa topologia mista consente un disaccoppiamento completo tra le azioni dell'utente sull'interfaccia e la logica di mutazione dei dati, preservando così un'elevata reattività (responsiveness) del sistema.

4.2 Definizione e Serializzazione dei Payload Dati (DTO)

L'interazione tra un ambiente di calcolo nativo e fortemente tipizzato (quale il backend in C++) e un'applicazione web (basata su TypeScript) comporta una criticità di natura strutturale: il trasporto delle informazioni avviene infatti tramite il formato testuale JSON, il quale risulta per sua natura privo di tipizzazione forte. Allo scopo di preservare l'integrità dello stato della simulazione e di prevenire errori a runtime nel corso della renderizzazione geospaziale, si è reso necessario progettare un sistema rigoroso di contratti dati, i cosiddetti Data Transfer Objects (DTO), condivisi logicamente tra i due layer dell'applicativo.

4.2.1 Contratti Dati per la Telemetria Continua

Il traffico dati di maggiore impatto per il sistema è quello generato dalla telemetria delle Forze Generate, veicolata tramite WebSocket. Per contenere l'overhead di rete e limitare la saturazione della banda, il backend C++ (per mezzo della routine di pubblicazione) compone un payload JSON compatto, in cui l'array delle entità viene incapsulato all'interno di una chiave di primo livello (denominata u), a cui vengono associati un timestamp Unix (chiave Timestamp) e il tempo simulato trascorso (chiave ElapsedSeconds).

Sul lato del Presentation Layer, spetta al frontend TypeScript accogliere tale struttura fluida e convertirla in un dato affidabile. A questo scopo sono state definite le interfacce CgfPayload e CgfUnitMessage. Considerato che, in un sistema distribuito, i pacchetti possono giungere incompleti o privi di alcuni campi (si pensi a un'entità che, in un determinato frame, non trasmette l'altitudine), l'interfaccia CgfUnitMessage dichiara tutti i campi come opzionali (tramite l'operatore "?").

La sanitizzazione e il casting sicuro dei dati sono affidati a routine di parsing dedicate (come parseUnitsFromMessage) e a funzioni di normalizzazione matematica, fra cui toFiniteNumber e normalizeHeadingDegrees. Grazie a questo approccio di tipo difensivo, in presenza di valori nulli, stringhe malformate o coordinate NaN (Not-a-Number), il client evita il verificarsi di eccezioni catastrofiche nel thread di rendering, inserendo valori di fallback di sicurezza (come 0.0 o Number.NaN) che permettono ai moduli grafici di scartare il pacchetto anomalo senza compromettere il ciclo di vita dell'applicativo.

4.2.2 DTO e Serializzazione per le Transazioni REST

Diversamente dalla telemetria, in grado di tollerare la perdita fisiologica di un pacchetto in virtù del suo aggiornamento continuo, i comandi transazionali impartiti dall'utente (REST) richiedono una validazione formale di tipo bidirezionale. La generazione di un tracciato tattico, così come quella di un singolo Waypoint, impone al client l'assemblaggio di payload estremamente accurati.

Per disciplinare tale flusso, il frontend definisce entità TypeScript apposite, quali EntityCreatePayload (destinata allo spawn di nuove unità) e RouteCreatePayload (relativa alle rotte). Nel caso specifico della creazione di una nuova rotta, l'operatore procede a tracciare i punti sull'interfaccia visiva; il modulo logico di editing (come descritto nell'architettura della UI) provvede a convertire le coordinate Cartesiane WebGL in coordinate geografiche assolute (Latitudine, Longitudine, Altitudine) e a costruire un array ibrido di nodi, denominato RouteNodePayload. Tale array può ospitare, senza distinzione, sia riferimenti a Waypoint già presenti nel database (per i quali viene inviato il solo identificatore id), sia coordinate pure (lat/lon), riferite a punti generati estemporaneamente.

Una volta giunto al backend, il payload non viene inserito direttamente nella memoria della simulazione. La procedura di serializzazione sicura prevede infatti che il controller HTTP intercetti la stringa JSON e ne affidi la ricostruzione a classi DTO omologhe scritte in C++ (quali RouteDto e WaypointDto), come rappresentato in Figura 4.1.

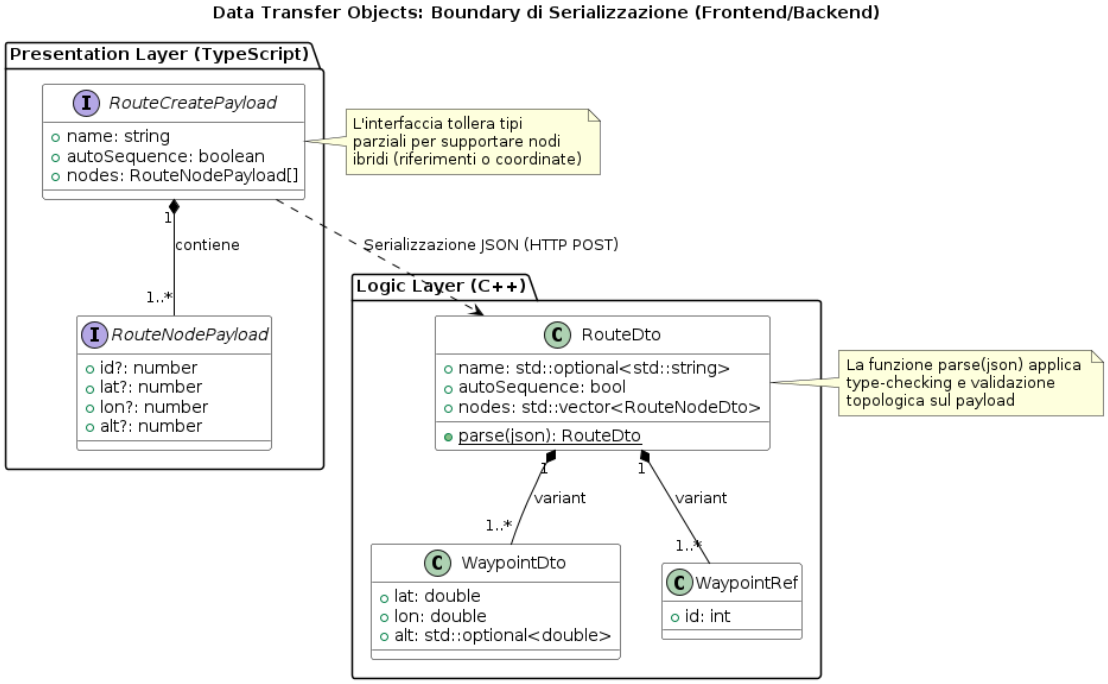


Figura 4.1: Diagramma delle classi che illustra il boundary di serializzazione e il mapping dei DTO tra il Presentation Layer (TypeScript) e il Logic Layer (C++).

Come emerge dal Low-Level Design del backend, tali classi effettuano un type-checking rigoroso (verificando la presenza delle chiavi indispensabili) e una validazione di dominio (verificando, ad esempio, che la latitudine ricada nell'intervallo compreso tra -90.0 e +90.0, oppure escludendo le rotte i cui nodi risultino coincidenti dal punto di vista topologico). Tale disaccoppiamento fa sì che il Presentation Layer operi come un raccoglitore intelligente, la cui struttura TypeScript risulta speculare rispetto al validatore C++ che la riceve, con conseguente riduzione delle vulnerabilità del sistema e delle anomalie di propagazione all'interno dell'ambiente di simulazione tattica.

4.3 Resilienza di Rete e Gestione del Ciclo di Vita delle Entità

Nell'ambito di un'architettura distribuita destinata alla simulazione LVC, l'affidabilità del trasporto dati non risulta garantibile in modo deterministico. Fluttuazioni della larghezza di banda, disconnessioni temporanee dei nodi HLA/DIS oppure arresti anomali del motore C++ possono infatti determinare l'interruzione del flusso telemetrico. Il Presentation Layer deve pertanto possedere requisiti di resilienza intrinseca, in grado di assicurare la tolleranza ai guasti (Fault Tolerance)

e l'integrità della rappresentazione tattica senza necessità di intervento da parte dell'operatore.

4.3.1 Tolleranza ai Guasti e Riconnessione Automatica (Retry Logic)

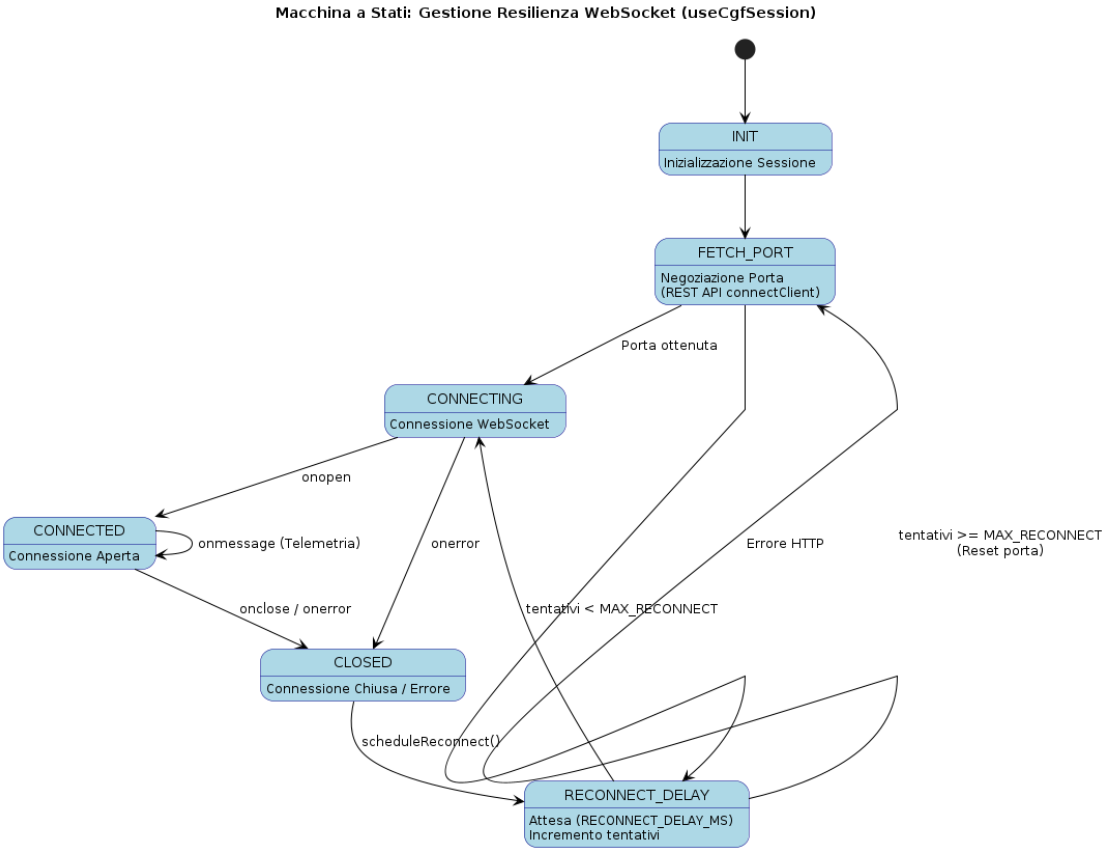


Figura 4.2: Diagramma della macchina a stati.

Il mantenimento del flusso WebSocket è regolato da una macchina a stati interna all'hook useCgfSession. Al verificarsi di un evento di chiusura inaspettata (onclose) o di un errore di rete (onerror), il client non procede all'invalidazione dell'intera sessione, bensì attiva una routine di riconnessione schedulata (Backoff), tramite la funzione scheduleReconnect.

Tale routine introduce un ritardo predefinito (RECONNECT_DELAY_MS, impostato a 2000 millisecondi) prima di tentare di ristabilire il socket, in modo da scongiurare l'innesco accidentale di attacchi di tipo Denial of Service (DoS) nei confronti del backend, qualora si verifichi un riavvio critico del server. Per

prevenire il verificarsi di loop infiniti su porte non più valide, l'algoritmo mantiene un contatore dei tentativi di riconnessione (reconnectAttemptsOnCurrentPort). Nel caso in cui il numero di fallimenti consecutivi superi la soglia limite (MAX_RECONNECT_ATTEMPTS_PER_PORT, configurata a 3), l'algoritmo procede al reset dei parametri di connessione e interroga nuovamente le API REST al fine di negoziare un nuovo endpoint valido, gestendo così in modo trasparente eventuali migrazioni di porta o riavvii del servizio backend.

4.3.2 Garbage Collection e Tracciamento Obsoleto (Stale Tracks)

Un problema analogo alla caduta della connessione globale è la problematica relativa alla perdita di telemetria di una singola entità. Qualora un veicolo cessi di trasmettere le proprie coordinate (ad esempio in seguito all'uscita dall'area di simulazione, oppure a causa di un fault temporaneo del simulatore federato ad esso associato), l'ultima posizione nota resterebbe congelata in memoria a tempo indeterminato, offrendo così all'operatore una percezione situazionale falsata.

Per mitigare tale rischio, il livello di rete implementa una strategia di Garbage Collection su base temporale, fondata su un dizionario in memoria (lastSeenByEntityId), che associa l'identificativo univoco di ciascuna entità al timestamp Unix relativo alla sua ultima ricezione telemetrica. A ogni ciclo di aggiornamento, la funzione cleanupStaleTracks effettua una scansione asincrona: qualora il delta temporale tra il timestamp corrente e l'ultimo aggiornamento registrato ecceda la soglia di obsolescenza (definita da TRACK_STALE_MS, pari a 30.000 millisecondi), l'entità viene contrassegnata come obsoleta e rimossa in modo imperativo sia dalle strutture dati locali sia dal motore geospaziale.

Per la gestione delle munizioni e dei missili (MajorType === "WEAPON") è stata introdotta una logica di ciclo di vita specifica. Diversamente dalle piattaforme terrestri o aeree, le armi presentano infatti un ciclo di vita transitorio, che si conclude con un impatto topologico (traducibile nel flag IsDead === true all'interno del payload telemetrico). Al fine di evitare che la mappa tattica venga saturata dalla persistenza visiva delle detonazioni, l'algoritmo registra le armi esplose in una struttura dati dedicata (deadWeaponsTimestamps). Decorsi 5 secondi dall'impatto, la funzione di cleanup elimina in via definitiva l'entità e ne inserisce l'identificativo in una black-list temporanea (removedWeapons), così da impedire che eventuali pacchetti telemetrici in ritardo (dovuti a lag di rete) possano dare luogo alla reistanziazione (Ghosting) di un elemento già distrutto e archiviato dalla simulazione.

4.4 Sincronizzazione Temporale e Gestione dello Stato di Simulazione

Nell'ambito di un'architettura di simulazione distribuita (LVC), la corretta gestione della linea temporale riveste un ruolo di primaria importanza. Il Presentation Layer non può infatti fare affidamento sul system clock locale della macchina host (il browser web), poiché l'evoluzione dello scenario è interamente governata dal motore computazionale C++, dotato della facoltà di mettere in pausa l'esecuzione, di riavvolgerla (Rewind) oppure di accelerarla mediante l'applicazione di specifici moltiplicatori di velocità. Per assicurare la coerenza a livello globale, il frontend deve pertanto implementare un meccanismo di sincronizzazione deterministico.

4.4.1 Orchestrazione dei Comandi di Trasporto (REST)

Il controllo del flusso temporale da parte dell'operatore avviene attraverso un insieme di comandi transazionali, comunemente noti come Transport Controls. Nel client web, tali interazioni sono demandate al modulo CgfManager, che astrae la complessità delle chiamate di rete per mezzo di metodi dedicati, quali executeTransportAction e setSimulationSpeed.

Questi metodi si occupano di tradurre l'intento dell'utente (Play, Pause, Restart) in chiamate HTTP POST e PUT indirizzate verso specifici endpoint REST esposti dal backend (ad esempio /api/simulation/start oppure /api/simulation/speed). Il ricorso al protocollo HTTP per i comandi di trasporto garantisce l'integrità della transazione e consente al client di ricevere un acknowledgement sincrono relativo all'effettivo cambio di stato del motore. Nel caso in cui il comando di avvio ("play") vada a buon fine, il frontend predispone appositi flag logici (come pendingPlayFocusRef), con lo scopo di istruire il motore di rendering affinché calcoli dinamicamente la Bounding Sphere delle entità e inquadri in modo fluido la zona di interesse topografica al primo aggiornamento telemetrico utile.

4.4.2 Sincronizzazione Distribuita del Clock (WebSocket)

La propagazione del tempo simulato a tutti i componenti dell'interfaccia (dal canvas WebGL fino alle dashboard grafiche) richiede un approccio architetturale caratterizzato da una latenza minima. Ogni pacchetto JSON telemetrico emesso dal server C++ contiene, oltre ai dati spaziali, una chiave fondamentale denominata ElapsedSeconds, la quale esprime il tempo simulato esatto trascorso dall'inizio dello scenario.

Ricevere e memorizzare tale valore a frequenze di 20 Hz all'interno del normale stato reattivo di React (ad esempio tramite useState) determinerebbe un collasso delle

prestazioni, dando origine a cicli di rendering superflui a carico dell'intero albero dei componenti a ogni "tick" dell'orologio. Per superare tale vincolo, all'interno di useCgfSession è stato implementato un pattern reattivo leggero (Observer), del tutto svincolato dal Virtual DOM.

Il sistema dichiara un set globale di callback (timeListeners), al quale i diversi componenti della UI possono iscriversi (Subscribe). Nel corso del parsing asincrono del payload WebSocket, il listener di rete intercetta il valore ElapsedSeconds e provvede a propagarlo istantaneamente, iterando sull'array delle callback (timeListeners.forEach(...)).

Questo paradigma di Event Emitter isolato assicura che l'orologio interno del frontend, così come le interpolazioni cinematiche sulla mappa, rispondano in modo deterministico ai dettami temporali del server, sostenendo in maniera fluida le accelerazioni temporali senza introdurre overhead computazionale nel ciclo di rendering dell'interfaccia utente.

4.5 Gestione del Carico Computazionale e Prevenzione dei Memory Leak

In un ambiente di simulazione Constructive, le sessioni operative possono protrarsi per diverse ore. La ricezione incessante di pacchetti telemetrici ad alta frequenza espone l'applicativo web a un rischio critico, ossia l'esaurimento della memoria allocata (Out of Memory) e il conseguente crash del browser dovuto a Memory Leaks (perdite di memoria). Sebbene JavaScript sia dotato di un Garbage Collector automatico, il mantenimento accidentale di referenze a oggetti ormai obsoleti impedisce al motore di liberare la RAM.

Per garantire la stabilità dell'applicativo su sessioni prolungate, l'architettura del frontend è stata progettata applicando precise strategie di gestione della memoria, agendo su tre livelli distinti.

4.5.1 Bypass dello Stato Storico nel Virtual DOM

Il primo intervento di ottimizzazione riguarda l'esclusione della telemetria dallo stato globale di React. Il ricorso a useState o Redux per la memorizzazione di un array composto da centinaia di entità in continuo mutamento non solo comporterebbe onerosi ricalcoli del DOM, ma indurrebbe React a conservare in memoria snapshot storici dello stato, necessari al processo di Reconciliation. Affidando la responsabilità del dato direttamente all'hook useCgfSession e mutando in modo imperativo la CustomDataSource di CesiumJS, il sistema fa sì che il

dato telemetrico precedente venga sovrascritto e reso immediatamente idoneo alla deallocazione da parte del Garbage Collector del browser.

4.5.2 Garbage Collection di Livello Applicativo (Entity Cleanup)

Il secondo livello di sicurezza agisce sulle strutture dati locali. Per il tracciamento delle entità, l'hook useCgfSession alloca in memoria alcune mappe, tra cui lastSeenByEntityId (deputata al tracciamento dei timestamp) e symbolStateByEntityId (deputata alla memorizzazione dei codici dei Milsymbol generati). Nel caso in cui un'entità venga distrutta o esca dallo scenario, tali mappe ne manterrebbero la referenza indefinitamente. L'implementazione della routine asincrona cleanupStaleTracks svolge, a tutti gli effetti, la funzione di un Garbage Collector di livello applicativo: individuate le entità le cui comunicazioni risultino interrotte da oltre 30 secondi (TRACK_STALE_MS), la routine provvede alla cancellazione esplicita (delete) delle relative chiavi da tutte le mappe locali. Questa netta recisione delle referenze fa sì che la memoria occupata dai metadati dell'entità venga restituita al sistema operativo.

4.5.3 Deallocazione delle Risorse GPU (WebGL)

Il terzo e ultimo strato di ottimizzazione riguarda la memoria video (VRAM) e le risorse di rendering. Quando la routine cleanupStaleTracks individua una traccia obsoleta oppure un'arma esplosa da oltre 5 secondi (deadWeaponsTimestamps), l'algoritmo non si limita alla rimozione dei dati logici, ma invoca esplicitamente il metodo removeById sulle istanze di CustomDataSource di CesiumJS, così da eliminare tanto l'entità principale quanto l'eventuale vettore di moto a essa associato. Tale chiamata imperativa riveste un'importanza cruciale: essa segnala al motore WebGL sottostante che i buffer dei vertici, le texture in base64 dei Milsymbol generate su Canvas e le primitive dei poligoni non necessitano più di essere calcolate nel game loop, autorizzando così CesiumJS a effettuare il flush (svuotamento) delle risorse grafiche allocate sulla GPU.

Questo approccio combinato assicura che il consumo di memoria (Memory Footprint) del client web rimanga stabile e deterministico, a prescindere dalla durata temporale dell'esercitazione simulata.

Capitolo 5

Sviluppo dell'Interfaccia Grafica (Front End)

5.1 Architettura Modulare React: Struttura dei componenti, gestione dello stato per dati dinamici ad alta frequenza

La progettazione dell'interfaccia utente (GUI) per un simulatore tattico comporta vincoli rigorosi tanto sul piano dell'usabilità quanto su quello delle prestazioni. All'interno di un ecosistema React, il modello reattivo prevede che ogni variazione dello stato globale determini un nuovo calcolo dell'albero dei componenti (Virtual DOM). L'applicazione non ponderata di tale meccanismo a nodi particolarmente onerosi dal punto di vista computazionale, quale il contenitore WebGL del motore geospaziale, produrrebbe ritardi non accettabili. Per superare questa criticità, la realizzazione del frontend ha fatto leva su una modularizzazione rigorosa e su una strategia ibrida di gestione dello stato (State Management).

5.1.1 Transizione UI/UX e Componentizzazione (Tailwind CSS e Shadcn/UI)

Come già evidenziato nell'analisi dei requisiti (Capitolo 3), il numero elevato di interazioni richieste all'operatore rendeva insostenibile il mantenimento di un'architettura stilistica fondata su CSS puro. Nella realizzazione della nuova GUI si è dunque scelto di adottare un paradigma Utility-First mediante Tailwind CSS, integrato con l'infrastruttura di componenti accessibili (Headless UI) messa a disposizione da Shadcn/UI.

Tale scelta progettuale ha consentito di sviluppare componenti articolati (quali dashboard, menu a tendina e pannelli modali), assicurando al contempo l'isolamento dello scope stilistico e prevenendo collisioni tra classi globali. Sul piano spaziale, l'interfaccia è stata concepita come un overlay assoluto sovrapposto al canvas di CesiumJS. Affinché la mappa sottostante restasse interattiva, si è ricorsi a una gestione puntuale della proprietà CSS pointer-events: i contenitori privi di contenuto risultano trasparenti al cursore, consentendo all'utente di continuare a manipolare la mappa, mentre i singoli componenti React (bottoni, menu, form) intercettano e isolano gli eventi del mouse, evitando che click accidentali interferiscano con la topologia del terreno.

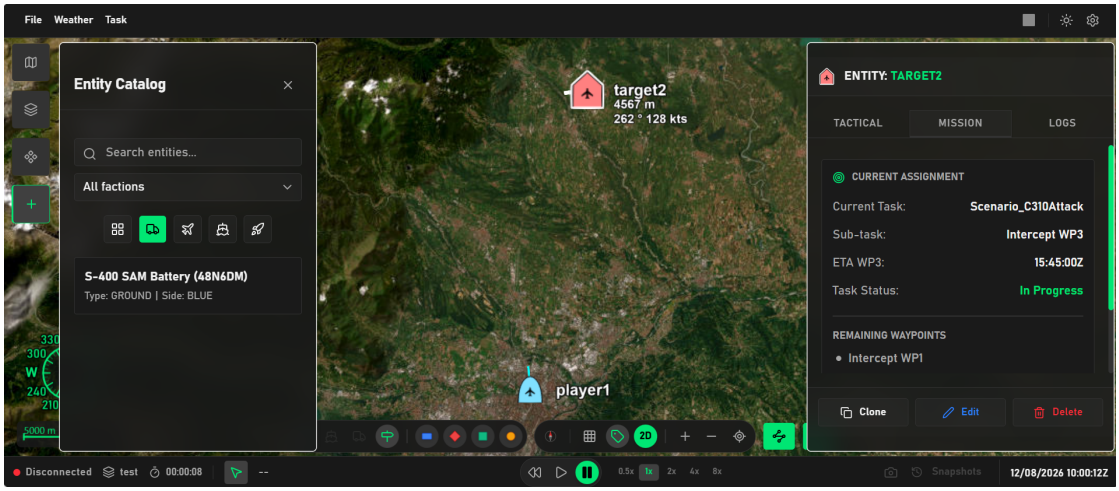


Figura 5.1: Pannelli in overlay sulla mappa WebGL.

5.1.2 Gestione Ibrida dello Stato (Redux e Context API)

Per assicurare l'isolamento dei cicli di vita tra i pannelli operativi e il motore di rendering, l'architettura suddivide lo stato dell'applicativo in due ambiti di competenza distinti:

- **Stato Globale e di Dominio (Redux):** le impostazioni strutturali e trasversali dell'applicativo sono demandate a uno store Redux centralizzato (RootState). Tale soluzione preserva la Single Source of Truth (singola fonte di verità) per i parametri ambientali, quali la sorgente luminosa (lightSource) o la visibilità degli indicatori astronomici, nonché per la condivisione delle entità correntemente selezionate (selectedEntities) fra i diversi visualizzatori.
- **Stato dell'Interfaccia Utente (Context API):** le interazioni di natura puramente presentazionale, quali l'attivazione dei pannelli operativi (ad esempio l'Editor di Rotte) o la scelta di uno strumento, vengono incapsulate

all'interno del modulo UiContext. Tale contesto, reso disponibile tramite l'hook useUi, governa variabili effimere (fra cui activePanel, selectedRoute e selectedCatalogEntity). Grazie a questa separazione logica, l'apertura o la chiusura di un menu laterale determina il ricalcolo del solo overlay HTML, lasciando inalterato il canvas di Cesium.

5.1.3 Orchestrazione dei Menu Contestuali

La convergenza fra questi due domini avviene all'interno del componente MapContainer. L'applicativo prevede che il click su elementi spaziali diversi attivi menù differenziati: attraverso un event listener intercettato a livello di telecamera, il sistema esamina la proprietà typeKey dell'oggetto selezionato e determina dinamicamente quali componenti React debbano essere montati. Qualora l'utente selezioni una singola unità tattica, viene montato il MapEntityRadialMenu (un menù posizionato in prossimità spaziale dell'entità); se invece la selezione riguarda uno strumento di misurazione, l'algoritmo attiva il MapRulerActionMenu oppure il pannello di intervisibilità (MapIntervisibilityMenu). Questa architettura state-driven garantisce che il DOM React allochi in memoria unicamente i nodi effettivamente necessari in quel preciso momento operativo.

5.1.4 Architettura Multi-Viewport (MultiViewManager)

Al fine di rispondere all'esigenza di un'analisi del teatro operativo condotta da prospettive multiple (ad esempio la visualizzazione contemporanea di una mappa 2D tattica e di una vista 3D locale), l'applicativo mette a disposizione un ambiente multi-viewport, gestito dalla classe MultiViewManager e renderizzato all'interno del componente MapPage.

La realizzazione modulare di tale componente ha comportato il superamento di due principali criticità tecniche connesse al layouting:

- **Istanziazione Dinamica della Griglia:** il sistema genera in modo dinamico un layout CSS capace di accogliere fino a quattro istanze indipendenti del motore geospaziale (limite definito dalla costante MAP_MAX_VIEWERS). Ciascuna istanza (cesiumContainer) dispone di un proprio ciclo di rendering, di telecamere dedicate e di una gestione autonoma dei filtri di visibilità.
- **Ottimizzazione del Ridimensionamento (Resize Handling):** il passaggio del layout da vista singola a griglia suddivisa comporta una variazione improvvisa delle dimensioni dei buffer grafici. Per prevenire il blocco del main thread e la comparsa di artefatti visivi indesiderati (tearing), il ricalcolo dimensionale dei canvas WebGL è stato delegato, a fini di ottimizzazione, alla Web API requestAnimationFrame.

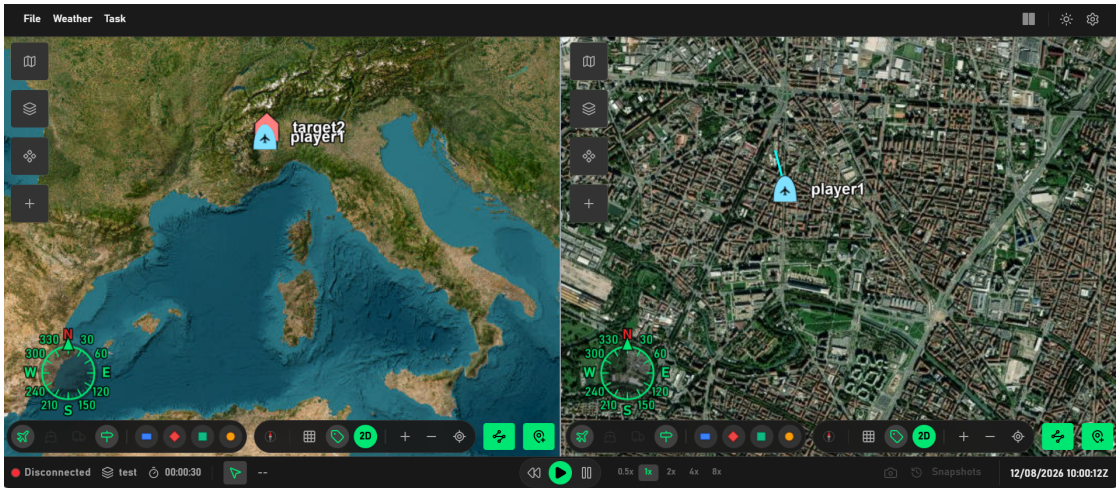


Figura 5.2: Griglia Multi-View con 2 canvas aperti simultaneamente.

5.2 Integrazione Mappe: Implementazione di Cesium per la visualizzazione delle mappe satellitari e dei dati vettoriali

Il motore geospaziale costituisce il nucleo visivo del simulatore tattico. L'integrazione di CesiumJS, tecnologia fondata su WebGL, nell'ecosistema React comporta una sfida ingegneristica di rilievo, riconducibile alla necessità di conciliare il paradigma dichiarativo proprio di React con il ciclo di rendering di natura imperativa che caratterizza CesiumJS.

5.2.1 Il Pattern Strutturale "MapAdapter" e la Strumentazione HUD

Per superare tale contrasto architetturale si è fatto ricorso al Pattern Adapter: il modulo MapAdapter svolge la funzione di wrapper di sicurezza attorno all'istanza nativa Cesium.Viewer. A nessun componente React è consentito intervenire direttamente sulla scena; ogni interazione di natura spaziale deve necessariamente transitare attraverso il contratto esposto dall'Adapter. Tale astrazione mette a disposizione API semplificate per la navigazione (come flyToEntities), racchiudendo al proprio interno la complessa trigonometria spaziale necessaria al calcolo delle Bounding Sphere delle entità, nonché all'estrapolazione delle altitudini del terreno topografico tramite l'interrogazione asincrona del modello digitale di elevazione (Cesium.sampleTerrain).

Tra gli elementi determinanti per la consapevolezza situazionale (Situational Awareness) figura la strumentazione di bordo in sovrimpressione (HUD), comprendente la rosa dei venti e la barra della scala metrica. Grazie all'isolamento assicurato dal MapAdapter, tali elementi sono stati concepiti in modo da risultare Context-Aware rispetto al tema applicativo. Mediante l'implementazione di un MutationObserver collegato al nodo radice del DOM (document.documentElement), il sistema è in grado di rilevare in tempo reale i cambi di tema (Light/Dark mode) impostati dall'utente. Non appena la variazione viene notificata, l'algoritmo impartisce simultaneamente a tutti gli HUD attivi nei diversi viewport l'istruzione di aggiornare le rispettive palette cromatiche, imponendo un nuovo calcolo grafico volto ad adattare senza ritardo il contrasto degli strumenti alle condizioni di luminosità dell'interfaccia.

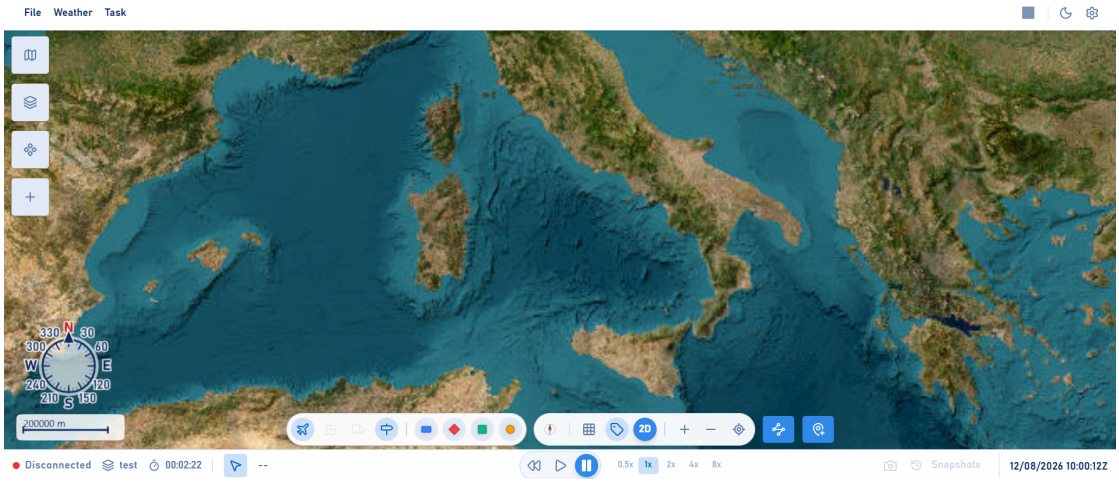


Figura 5.3: Mappa satellitare in light mode con bussola e scala in basso a sinistra.

5.2.2 Simbologia Tattica e Vettori di Moto Adattivi

La resa visiva dei dati vettoriali è tenuta a rispettare in maniera rigorosa gli standard militari NATO (MIL-STD-2525 / APP-6). Il sistema acquisisce i metadati grezzi associati alle singole entità (tra cui fazione e tipologia) e ne effettua la traduzione in marcatori grafici vettoriali. Oltre all'iconografia statica, il sistema provvede a proiettare sul terreno i vettori di movimento spaziale (Heading Vectors), offrendo così una percezione visiva immediata della cinematica delle Forze Generate.

Affinché i vettori si mantengano leggibili a qualunque livello di zoom, la geometria della polilinea viene calcolata in modo dinamico ricorrendo alla classe Cesium.CallbackProperty, costrutto che demanda alla GPU il ricalcolo a ogni singolo frame, interpolando la lunghezza del vettore visivo non soltanto sulla base della velocità reale (GroundSpeed), ma anche in funzione dell'altitudine corrente

della telecamera. Un apposito algoritmo di normalizzazione matematica evita che il vettore scompaia alle quote elevate o che, viceversa, distorca eccessivamente la visuale alle quote basse, assicurando in tal modo la leggibilità della direzione tattica indipendentemente dal livello di ingrandimento adottato per lo scenario.

5.2.3 L'Editor Spaziale e l'Iniezione delle Entità (Point & Click Spawner)

Il passaggio del frontend da strumento di monitoraggio passivo a piattaforma di comando attiva trova concreta espressione nell'Editor Spaziale e nel sistema di inserimento manuale delle entità. Qualora l'operatore selezioni, dal catalogo, una nuova unità da collocare nello scenario (si pensi, ad esempio, a una batteria missilistica terra-aria), l'interfaccia React transita verso una specifica macchina a stati: il cursore assume la forma di un mirino direzionale (crosshair), mentre la consueta navigazione pan/zoom della mappa viene temporaneamente inibita, così da prevenire eventuali errori in fase di inserimento.

L'acquisizione dell'intento dell'utente si realizza per mezzo di un Cesium.ScreenSpaceEventHandler, il quale intercetta il click sul piano bidimensionale dello schermo per proiettarlo nello spazi. Tale processo si articola in due distinte strategie algoritmiche:

1. **Ray-Casting Topologico:** nel caso in cui l'operatore clicchi su un'area libera della mappa, il sistema genera un raggio matematico (camera.getPickRay) proiettato dalla prospettiva dell'osservatore verso la mesh 3D del globo terrestre. L'elaborazione del punto esatto di intersezione produce una coordinata Cartesiana 3D assoluta, la quale viene immediatamente convertita in coordinate geografiche (Latitudine, Longitudine, Altitudine) e inserita nel payload JSON (EntityCreatePayload) destinato all'invio verso il backend REST.

2. **Drill-Picking Logico:** qualora, nel corso della creazione di tracciati e rotte, il raggio intersechi un'entità logica già esistente (ad esempio un Waypoint preesistente), l'algoritmo effettua un attraversamento profondo (drill-pick) dei layer grafici, ricavando l'identificativo esatto dell'oggetto agganciato e consentendo così all'interfaccia di costruire rotte tattiche pienamente interconnesse con il database esistente.

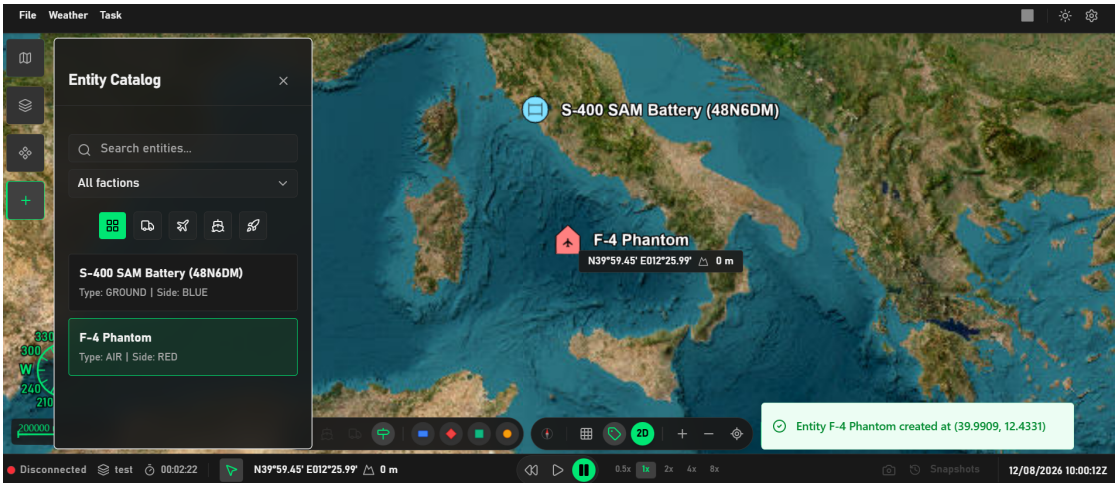


Figura 5.4: Fase di spawn di un'entità.

5.3 Gestione delle Sfide Tecniche: Risoluzione di problemi legati al rendering e alle performance della GUI

La realizzazione di un'interfaccia grafica per un simulatore LVC richiede il superamento di rigidi colli di bottiglia computazionali. Il browser web è dotato di un unico main thread, preposto tanto all'esecuzione della logica JavaScript quanto al calcolo dei layout geometrici (reflow/repaint). Qualora si verifichi la ricezione simultanea di decine di pacchetti telemetrici ad alta frequenza, tale thread rischia di saturarsi, generando fenomeni di stuttering (scatti) e compromettendo in maniera irrimediabile la reattività della GUI. Per tale motivo, l'architettura è stata sottoposta a un processo di ottimizzazione approfondito, riguardante sia il rendering sia la gestione della memoria video.

5.3.1 Rendering On-Demand e Iniezione Diretta dei Dati (Bypass del Virtual DOM)

La prima problematica di natura architetturale ha riguardato il meccanismo di aggiornamento della scena. Consentire al ciclo di rendering di React di governare, secondo un approccio dichiarativo, le strutture dati particolarmente onerose di CesiumJS (quali entità e dataset vettoriali) avrebbe comportato un degrado delle prestazioni non accettabile.

La soluzione adottata, implementata all'interno del metodo upsertTrack, si fonda sull'esecuzione di una logica di aggiornamento transazionale ad alte prestazioni

(Upsert: Update/Insert), che elude deliberatamente l'alberatura del DOM HTML. Nel momento in cui giunge un payload asincrono riferito a una determinata entità, la modifica delle coordinate o dell'orientamento spaziale si realizza accedendo alle primitive di CesiumJS (quali il billboard o la label) e iniettando i valori corrispondenti all'interno di un oggetto Cesium.PropertyBag. Tale mutazione imperativa dei nodi elimina integralmente, per React, il costo di ricalcolo altrimenti previsto.

Al fine di evitare, inoltre, un sovraccarico della GPU imputabile a un game loop continuo a 60 FPS anche nei momenti di stasi tattica, l'aggiornamento visivo è stato configurato in modalità on-demand. Una volta disattivato il rendering automatico, il modulo si limita a inviare al motore WebGL una richiesta esplicita di aggiornamento del frame (per mezzo dell'invocazione di requestRender), operazione che avviene esclusivamente al termine dell'iniezione dei nuovi delta telemetrici. Il client React mantiene in tal modo il controllo (ownership) della Business Logic, riducendo di fatto l'interfaccia a un semplice modulo esecutivo per l'I/O spaziale.

5.3.2 Memoization e Caching dell'Elaborazione Vettoriale (Simbologia APP-6)

Una seconda criticità sul piano prestazionale è emersa in fase di conversione dei metadati tattici nella simbologia militare standard: la generazione di tali simboli, affidata alla libreria dedicata, comporta infatti un costo computazionale tutt'altro che trascurabile, poiché presuppone la costruzione dinamica di alberature SVG di notevole complessità.

Per ottimizzare tale processo, la funzione incaricata della sintesi grafica (getSymbolImage) è stata dotata di una rigorosa logica di memoization (caching): il sistema costruisce una chiave univoca risultante dalla combinazione tra la stringa standard dell'unità (codice SISO) e i suoi stati visivi effimeri, quali la selezione (isSelected) o la distruzione (isDead).

Tra le casistiche critiche affrontate rientra proprio quella relativa allo stato di distruzione: al fine di differenziare le unità eliminate, il codice SVG generato in modo nativo viene "avvolto" (wrapped) all'interno di un ulteriore tag <svg> personalizzato, che vi sovrappone un indicatore rosso di kill (un cerchio barrato), calcolato matematicamente in riferimento al centro (Anchor) del simbolo originale. L'output così ottenuto, convertito in un Data URI (Base64), trova collocazione in un dizionario locale (symbolCacheRef). Grazie a questa ottimizzazione, un'operazione vettoriale altrimenti gravosa si traduce in un recupero testuale immediato, con complessità temporale O(1).

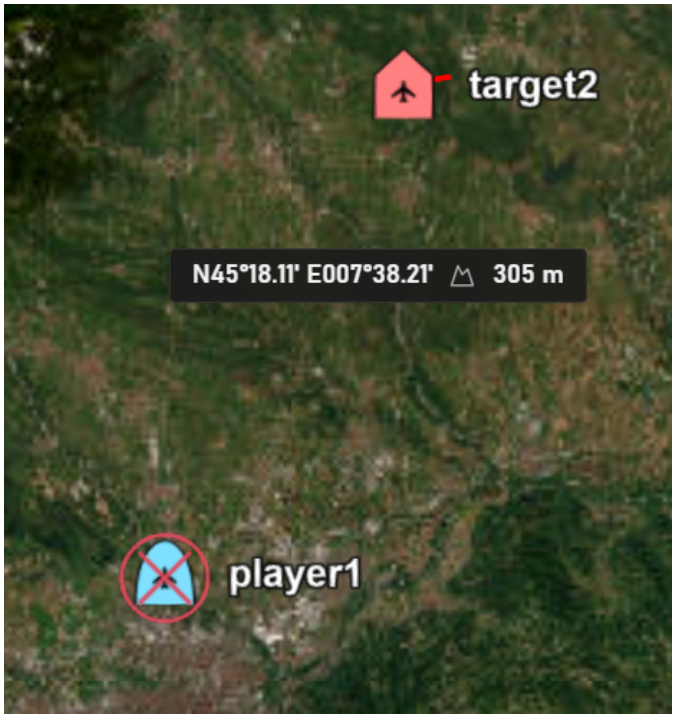


Figura 5.5: Esempio di player attivo e player distrutto.

5.3.3 Ottimizzazione del Motore di Filtraggio (Visibility Toggling)

L'ultima problematica affrontata riguarda la gestione della visibilità condizionale: l'operatore ha infatti a disposizione strumenti che consentono di filtrare le entità presenti sulla mappa, ad esempio occultando le forze nemiche o mostrando unicamente i vettori aerei.

Un approccio convenzionale avrebbe comportato la rimozione fisica (destroy) delle entità escluse dal layer della mappa, seguita da una loro re-istanziazione al momento della disattivazione del filtro, con la conseguenza di innescare cicli intensi di Garbage Collection.

Per assicurare una latenza nulla, la logica di controllo adottata agisce esclusivamente sulla proprietà show delle astrazioni Cesium.CustomDataSource e delle singole entità. L'attivazione o la disattivazione dei layer tattici si riduce così a una semplice operazione booleana, capace di occultare i poligoni dal canvas senza comprometterne la persistenza in memoria, azzerando il carico di allocazione a carico della CPU.

Capitolo 6

Testing e Validazione

6.1 Specifica e Test Design (Matrice di Validazione)

Garantire la coerenza degli stati e l'affidabilità comportamentale delle entità generate all'interno di un sistema di simulazione distribuito richiede necessariamente l'adozione di un approccio strutturato, vista la complessità intrinseca di tali architetture. A tale scopo è stata elaborata una Matrice di Validazione, articolata in circa sessanta casi di test formali. Va precisato che tale matrice non ambisce a un'esplorazione esaustiva dell'infinito spazio degli stati proprio del motore di simulazione, bensì è stata concepita come Baseline Framework, con l'obiettivo di isolare i pattern comportamentali critici (relativi a navigazione, vincoli fisici e regole d'ingaggio) così da delineare un perimetro solido e scalabile per i futuri cicli di automazione. Il documento assume dunque la funzione di specifica sia architetturale sia operativa, delimitando gli scenari critici mediante una struttura rigorosa articolata in precondizioni di avvio, componenti coinvolte, azioni scatenanti e risultati attesi. Nella progettazione della suite, l'attenzione non si è limitata agli scenari di normale esecuzione (happy path), ma si è rivolta soprattutto all'individuazione di limiti fisici, conflitti di stato e risposte del sistema a input anomali.

6.1.1 Validazione del Ciclo di Vita e Sincronizzazione (Station)

Il modulo "Station", cui spetta l'orchestrazione temporale della simulazione, ha costituito oggetto di una porzione sostanziale del test design. I relativi casi di test si basano sull'invio, tramite WebSocket, di payload JSON finalizzati a

comandare le transizioni di stato della variabile interna m_simState. È stata così definita la validazione dei passaggi tra SimState::PAUSE, stato assunto di default all'avvio, e SimState::PLAY, verificando il corretto aggiornamento dei metodi di elaborazione (updateTC(), publishPlayers()) e il conseguente avanzamento del tempo di simulazione.

Un'attenzione analitica particolare è stata dedicata alla transizione di REWIND, attivabile tramite il comando di restart ("action":"setState","status":"restart"). Secondo quanto previsto dalla specifica, il sistema deve ritornare alle condizioni iniziali, ripristinando le posizioni di partenza e sospendendo l'avanzamento temporale fino all'impartizione di un nuovo comando di play. L'analisi di tale scenario ha permesso di documentare criticità rilevanti legate al determinismo del motore: i test hanno infatti evidenziato la mancata distruzione delle entità istanziate dinamicamente (quali le armi in volo), nonché la mancata reinizializzazione degli identificatori univoci globali e dello stato interno delle rotte nel corso dell'esecuzione del comando. La matrice comprende, inoltre, test di resilienza volti a verificare che il server prosegua il proprio funzionamento senza andare incontro a crash anche a fronte della ricezione di JSON malformati o di comandi di esecuzione duplicati.

6.1.2 Validazione Cinematica e Rispetto dei Vincoli Fisici (OPD)

I test comportamentali sono stati progettati per assicurare che le manovre delle entità rispettino rigorosamente le limitazioni fisiche definite nei profili OPD (Operational Performance Data) di ciascun velivolo. Nel task "Fly Heading", ad esempio, la specifica impone che qualora venga richiesto un rateo di virata superiore alle capacità dell'aeromobile, il sistema debba limitare automaticamente il valore ponendolo uguale al massimo consentito dalle prestazioni reali.

La matrice differenzia in modo netto i comportamenti attesi in base alla classe dell'aeromobile. Per i velivoli ad ala fissa è stato specificato che task direzionali o di quota impartiti con il mezzo a terra debbano essere ignorati. Al contrario, per i velivoli ad ala rotante, i medesimi comandi ("Fly Heading" o "Move to Altitude") devono innescare corrette procedure di decollo verticale, raggiungimento di una quota di stabilizzazione e successiva esecuzione della manovra. Un caso limite documentato riguarda l'assegnazione di quota zero in volo per gli elicotteri, per il quale il risultato atteso prevede l'arresto della discesa a una quota di sicurezza, fissata a 33 metri, per prevenire collisioni col terreno.

6.1.3 Navigazione Logica e Gestione Dinamica del Tempo

Con riferimento alla validazione della navigazione autonoma ("Move Along Route"), i test garantiscono che il calcolo del percorso segua una sequenzialità logica rigorosa, dettata dal piano di volo, anziché affidarsi a una logica puramente geometrica. Qualora i waypoint siano disposti sulla mappa in modo non lineare, il velivolo deve portare a termine l'intera sequenza stabilita senza effettuare "tagli di rotta", anche nel caso in cui un waypoint successivo si trovi spazialmente più vicino rispetto a quello attualmente designato. È stato inoltre verificato il corretto funzionamento delle rotte cicliche, assicurando che, giunto al termine del percorso, il velivolo riprenda l'iterazione a partire dal primo vertice logico.

Tra gli aspetti critici coperti dalla specifica rientra la gestione temporale affidata al task "Time on Target (ToT)". I relativi casi di test verificano che, a fronte di un orario di arrivo prestabilito ampiamente nel futuro, l'entità riconosca l'anticipo ed entri spontaneamente in una procedura di holding (eseguendo virate in quota a una distanza minima di 3700 metri dal target) così da smaltire il tempo in eccesso prima di riprendere la rotta diretta. Qualora il ToT imposto risulti invece matematicamente irraggiungibile, il sistema deve intercettare il superamento delle tolleranze e indurre l'entità ad abbandonare autonomamente il task.

6.1.4 Inseguimento Tattico e Risoluzione dei Conflitti Multipli

Le logiche di intercettazione sono state validate per mezzo del task "Follow Entity", concepito per gestire l'inseguimento di bersagli in movimento nel rispetto di offset e distanze predefiniti. Secondo quanto stabilito dalla specifica, l'inseguitore è tenuto ad adattare la propria navigazione riproducendo i movimenti del bersaglio, pur rimanendo vincolato alla propria velocità massima qualora quest'ultimo risulti più rapido.

Sottoponendo tale componente a stress test è emersa una lacuna di natura architetturale nella gestione dei conflitti multipli: assegnando infatti il medesimo task, con parametri di offset spaziale identici, a due inseguitori distinti, si è osservato come il sistema elaborasse i comandi in maniera isolata, determinando la convergenza dei velivoli alleati e la conseguente collisione in volo. Dall'analisi di questa anomalia è scaturita la teorizzazione della necessità di un sistema centralizzato (Formation Manager), in grado di validare preventivamente gli slot spaziali già occupati, rifiutando configurazioni sovrapposte e segnalando l'errore all'operatore.

6.1.5 Logiche di Ingaggio e Controllo dei Sistemi d'Arma

L'ultima macro-area affrontata dal documento riguarda i criteri di affidabilità stabiliti per le routine di combattimento e difesa. Per quanto concerne le missioni di attacco (Air-to-Air e Air-to-Ground), i relativi casi di test coprono l'intero ciclo dell'ingaggio: dalla navigazione lungo le rotte di ingresso, all'acquisizione e al lock sul bersaglio, fino al rispetto della gittata massima delle armi e alla corretta interruzione dell'attacco qualora il bersaglio scompaia dai sensori radar.

Sono stati altresì previsti controlli logici stringenti, finalizzati a prevenire azioni non valide quali il lancio di armi in assenza di munizionamento adeguato, il tentativo di impiegare missili anti-aerei contro unità terrestri, oppure l'ingaggio di velivoli alleati. Per quanto riguarda infine i sistemi di difesa automatica, la matrice definisce le regole di risposta autonoma ("Automatic Air Defense" ed "Evade Missile"), in base alle quali le entità sono tenute a ignorare le minacce fuori portata e ad attivare manovre evasive combinate esclusivamente nel momento in cui un missile avversario superi la soglia critica di prossimità, fissata a 5000 metri.

6.1.6 Evoluzione Architetturale e Sviluppi Futuri

Nel periodo dedicato alla stesura e all'analisi di questi scenari, l'architettura del backend in C++ è stata interessata da refactoring frequenti e di ampia portata. Tale rapida evoluzione del motore ha reso prematura ul'implementazione del codice di test relativo alla GUI e alla Station, in quanto script fortemente accoppiati a logiche ancora in mutamento hanno prodotto test fragili e soggetti a falsi negativi. Si è pertanto scelto di adottare un approccio orientato alla Specifica Formale: il reale valore aggiunto di questa fase non risiede tanto nell'esecuzione del codice di test, quanto nell'aver modellato a monte le risposte del sistema, anticipando conflitti di stato e anomalie di sincronizzazione. Sebbene i test implementati nelle fasi più precoci dell'indagine siano stati in larga misura superati dalle successive revisioni architetturali, la Matrice di Validazione così prodotta conserva pienamente il proprio valore di artefatto strutturale, costituendo a tutti gli effetti il backlog tecnico ufficiale a cui fare riferimento per la futura integrazione di tali scenari all'interno di una pipeline aziendale di Continuous Integration strutturata.

6.2 Automated Regression & Integration Testing (API Backend)

La validazione dell'interfaccia di comunicazione tra i client e il motore di simulazione è stata condotta per mezzo di una suite di test black-box automatizzata, sviluppata in PowerShell (test_api.ps1) e finalizzata a esercitare l'intera copertura delle API

REST esposte dal server CGF. L'obiettivo perseguito in questa fase non si è esaurito nella verifica dell'assenza di crash di sistema ("didn't throw"), ma ha comportato altresì un controllo rigoroso sull'effettivo mutamento di stato delle risorse: a ogni chiamata di mutazione (POST, PUT, DELETE) segue infatti un'asserzione sul relativo codice di stato HTTP (ad esempio 200 OK, 400, 404, 409, 422), cui fa seguito un'operazione di GET volta a convalidare programmaticamente che il payload JSON rifletta la modifica attesa a livello architetturale.

6.2.1 Architettura del framework e Dynamic Cleanup

Al fine di garantire l'idempotenza e l'isolamento delle singole esecuzioni, l'architettura della suite di test prevede, nella fase conclusiva, l'integrazione di un pattern di Dynamic Cleanup. Anziché ricorrere a un semplice hard-reset del database, lo script interroga in modo programmatico gli endpoint di lista (/api/simulation/-routes, /api/simulation/waypoints), itera sulle risorse ancora presenti ed esegue, per ciascuna entità, chiamate DELETE mirate, verificando infine che il server risponda con il messaggio di specifica previsto per l'assenza di risorse nel pool. Tale approccio consente non solo di garantire la pulizia dello stato, ma anche di sottoporre a validazione esplicita le logiche di eliminazione a cascata.

6.2.2 Validazione del Ciclo di Vita e Limiti di Copertura

Una prima porzione della suite è riservata al ciclo di vita della simulazione, mettendo sotto stress gli endpoint /api/simulation/start, /pause, /reset e /speed. Sono state validate con esito positivo tanto le transizioni tra i vari stati operativi (ad esempio da PAUSE a PLAY), quanto la corretta generazione degli errori in presenza di conflitti di stato (come la ricezione di un HTTP 409 Conflict qualora si tenti di avviare una simulazione già in corso). Relativamente all'endpoint di manipolazione della velocità, la copertura è stata estesa fino a includere la Boundary Value Analysis, verificando il rifiuto (422 Unprocessable Entity) di valori negativi o pari esattamente a zero, pur ammettendo l'elaborazione di valori decimali estremi.

L'analisi di copertura ha altresì permesso di individuare e formalizzare un limite architetturale ben preciso del testing automatizzato di base: gli errori di sistema di livello 5xx (500 Internal Error, 503 Not Ready, 504 Timeout) restano infatti esclusi dalla copertura della suite. A tale proposito è stato documentato come l'innesco di simili anomalie richieda tecniche di environmental fault injection (quali la disconnessione forzata della Station esterna o la terminazione improvvisa dell'engine) non compatibili con i normali flussi di regressione.

6.2.3 Validazione CRUD e coerenza topologica (Move, not Clone)

La validazione degli endpoint dedicati alla gestione topologica (/api/simulation/-waypoint e /api/simulation/route) ha reso necessaria l'implementazione di controlli avanzati volti a garantire la coerenza dei dati, sia geografici sia strutturali. I test verificano la rigorosa applicazione della regola architetturale "Move, not clone", secondo la quale, nel momento in cui un waypoint viene assegnato a una rotta tramite API, l'entità deve essere rimossa dal pool generale ed esistere esclusivamente all'interno della rotta stessa. L'automazione certifica inoltre che eventuali tentativi di fare riferimento allo stesso ID in sequenza producano correttamente un errore 422.

Ulteriori test di robustezza condotti su tali endpoint assicurano che il sistema sia in grado di intercettare formati JSON malformati (restituendo, in tal caso, 400 Bad Request) così come input che eccedano i range geografici consentiti (ad esempio una latitudine superiore a 90 o inferiore a -90). Sono stati infine introdotti test di stress che iniettano payload contenenti matrici massive di nodi inline, allo scopo di verificarne la tenuta del parser lato backend.

6.2.4 Anomalie di Routing e ciclo di vita delle API

Grazie all'approccio esplorativo adottato dalla suite è stato possibile individuare un bug sistemico nel router HTTP C++ del backend. In particolare, nel corso dell'interrogazione degli endpoint GET /api/simulation/state e GET /api/simulation/routes, è emerso come l'aggiunta di query parameters inattesi (quali ?sort=asc&limit=5) determinasse il fallimento della chiamata con un 404 Not Found, anziché il previsto 200 OK. Tale anomalia è riconducibile al fatto che il backend esegue un match esatto della stringa URI comprensiva dei parametri, non riuscendo così a risolvere la rotta di base.

L'implementazione formale dei test ha infine orientato scelte ingegneristiche relative al ciclo di vita delle stesse API. La valutazione del comportamento di endpoint trasversali quali /api/simulation/waypoint/assign e /api/simulation/route/assign ha condotto il team a concludere che il design di tali API risultasse scorretto e, in quanto tale, meritevole di deprecazione. L'analisi di copertura documenta esplicitamente come, pur essendo stati mantenuti i test di base per evitare regressioni temporanee nel corso della transizione, tali endpoint siano formalmente destinati alla rimozione, a dimostrazione di come il collaudo funga da strumento attivo nella validazione dell'architettura software.

6.3 Validazione Automatizzata e Strategie di Robustezza del Frontend

La validazione del client si è configurata come una sfida metodologica di rilievo, riconducibile alla compresenza di un'interfaccia utente standard (basata sul DOM) e di un motore di rendering 3D fortemente specialistico (CesiumJS). Per assicurare la robustezza del sistema evitando al contempo di produrre test fragili e onerosi da mantenere, si è scelto di adottare una strategia di End-to-End (E2E) Testing incentrata sui flussi critici di controllo (Command & Control).

6.3.1 Selezione del Framework e Requisiti Tecnologici

La scelta dello strumento di automazione è ricaduta su Playwright, preferito a soluzioni quali Selenium o Cypress sulla base di tre motivazioni tecniche fondamentali:

- **Auto-waiting e Determinismo:** Playwright gestisce in maniera nativa l'asincronia tipica del frontend moderno (React), attendendo la stabilizzazione degli elementi prima di interagire con essi. Ciò consente di ridurre in maniera drastica i falsi negativi riconducibili a latenze di rete o ai tempi di caricamento del motore Cesium.
- **Isolamento e Cross-Browser:** la possibilità di eseguire i test in contesti isolati ha reso possibile validare l'applicazione in un ambiente pulito, prevenendo così interferenze tra sessioni di test successive.
- **Observability:** grazie al Trace Viewer è stato possibile analizzare ogni singola transizione di stato, aspetto fondamentale ai fini del debug di interazioni complesse tra la GUI e i flussi WebSocket.

6.3.2 Superamento dei Limiti del Rendering WebGL

Un aspetto critico della validazione ha riguardato specificamente il motore Cesium: poiché gli oggetti simulati (unità, tracce, aree) vengono renderizzati all'interno di un unico tag <canvas> per mezzo di WebGL, essi risultano invisibili agli strumenti di ispezione standard del browser.

Per aggirare tale ostacolo senza ricorrere a test fragili basati su coordinate fisse di pixel (sensibili, in quanto tali, alla risoluzione dello schermo), è stata implementata una strategia di validazione basata sul feedback asincrono: il test, anziché verificare "visivamente" la presenza dell'entità, intercetta i canali di comunicazione e i componenti di notifica (toast) che il frontend genera esclusivamente a seguito della conferma di avvenuta operazione da parte del motore di simulazione.

6.3.3 Architettura della Suite di Test E2E

La suite di test così implementata segue il pattern AAA (Arrange, Act, Assert) e risulta suddivisa in due macro-categorie funzionali:

1. **Gestione del Ciclo di Vita delle Entità (CRUD Operations)**

   Questi test hanno lo scopo di validare la capacità dell'operatore di interagire con il database tattico della simulazione.

   - **Creazione (Catalog-to-Map):** verifica che dalla selezione di un'entità specialistica (ad esempio una batteria S-400) e dal successivo posizionamento sulla mappa scaturisca la generazione di un'istanza corretta.
   - **Gestione e Manutenzione:** validazione dei flussi di editing delle proprietà e di eliminazione sicura (mediante guardia modale di conferma), a garanzia del fatto che ogni modifica venga persistita e comunicata all'utente.

2. **Resilienza di Stato e Idempotenza**

   Data la natura distribuita del sistema, risulta fondamentale che il frontend gestisca in modo corretto le race conditions. A tal fine sono stati sviluppati test specifici, volti a verificare il comportamento del sistema a fronte di comandi ridondanti (ad esempio doppi clic sui pulsanti "Play" o "Pause"). La validazione garantisce che il client sia in grado di intercettare i segnali di conflitto provenienti dal backend e di fornire un feedback di errore controllato, impedendo così la desincronizzazione tra interfaccia ed engine.

6.3.4 Validazione Client-Side e Prevenzione degli Errori

Accanto ai test E2E, una parte rilevante della validazione è stata riservata alla logica difensiva del client, tramite un filtraggio proattivo degli input finalizzato a garantirne l'integrità sia geografica sia operativa:

- **Sanificazione dei dati geografici:** implementazione di validatori dedicati alle coordinate di latitudine/longitudine e ai formati di input (ad esempio MGRS), così da bloccare alla fonte l'invio di dati potenzialmente in grado di generare eccezioni nel motore di simulazione.
- **Controllo dei range operativi:** i campi numerici (velocità, quota, orientamento) sono stati muniti di logiche di controllo basate sul dominio applicativo, tali da disabilitare le azioni di invio qualora i dati risultino non conformi.

In conclusione, la combinazione di una suite E2E mirata e di una solida logica di validazione degli input ha consentito di realizzare un client in grado di agire quale filtro di qualità, a tutela della stabilità dell'intera architettura distribuita.

Capitolo 7

Conclusioni e Sviluppi Futuri

Il presente capitolo trae le conclusioni del lavoro svolto, valutando il grado di raggiungimento degli obiettivi dichiarati in apertura della tesi e l'applicabilità industriale dell'architettura proposta, per poi delineare le direttrici di sviluppo future orientate a una piena commercializzazione della piattaforma.

7.1 Valutazione degli Obiettivi Raggiunti

L'obiettivo dichiarato in apertura di questo elaborato (Cap. 1) era guidare l'evoluzione della piattaforma CGF da uno stato embrionale (nel quale risultavano già individuati lo stack tecnologico di base e alcune funzionalità visive elementari) verso un'architettura web modulare e di livello industriale, ispirata a soluzioni commerciali consolidate come VR-Forces. Alla luce del percorso descritto nei capitoli precedenti, tale obiettivo può considerarsi sostanzialmente raggiunto, sebbene con un livello di maturità differenziato a seconda del sottosistema considerato.

7.1.1 Dall'Analisi dei Requisiti all'Architettura Realizzata

La fase di Requirements Engineering (Cap. 3) ha prodotto un documento di specifica di oltre cinquanta casi d'uso, distribuiti su cinque macro-aree funzionali. Di questo perimetro complessivo, il lavoro si è concentrato in particolare sui moduli a maggiore complessità interattiva (la gestione del ciclo di vita delle entità e la logica tattica di navigazione) portandoli a un livello di implementazione funzionante e verificabile, mentre altre aree della specifica (tra cui l'editor avanzato dei modelli

entità, la configurazione delle sessioni DIS/HLA e il sottosistema di autenticazione) sono rimaste, allo stato attuale, a livello di sola specifica, costituendo un perimetro di lavoro definito, ma non ancora attraversato in fase implementativa. Questa distinzione tra funzionalità specificate e funzionalità implementate non è un limite imprevisto, ma il risultato naturale di un processo di prioritizzazione condotto in un contesto di tempo e risorse vincolati, coerente con la metodologia Agile adottata.

Sul piano architetturale, il client web è stato consolidato ed esteso secondo la separazione in quattro livelli logici (Presentazione, Spaziale, Stato e Comunicazione) descritta nel Capitolo 3, con un intervento sostanziale sul layer di Comunicazione: l'insieme delle API REST per le operazioni transazionali è stato interamente sviluppato e la gestione della telemetria via WebSocket è stata ampliata per sostenere un flusso continuo fino a 20 aggiornamenti al secondo senza degrado delle prestazioni di rendering (Cap. 4). A questo si affianca la migrazione del Presentation Layer da CSS puro a Tailwind CSS e shadcn/ui (Cap. 3, Cap. 5), che ha eliminato il rischio di collisione tra classi di stile e reso possibile la realizzazione di componenti complessi (menu contestuali, ambiente multi-viewport, editor spaziale) altrimenti difficilmente sostenibili con l'approccio di partenza.

7.1.2 Robustezza e Validazione: Risultati Conseguiti e Limiti

La fase di testing e validazione (Cap. 6) ha prodotto risultati concreti ma disomogenei tra i diversi sottosistemi. Sul lato backend, la suite di regressione automatizzata ha permesso di individuare e documentare anomalie reali del sistema (tra cui la mancata pulizia delle entità dinamiche nel corso del comando di REWIND, un bug di routing legato alla presenza di query parameters non attesi, e la necessità di deprecare gli endpoint /waypoint/assign e /route/assign) dimostrando come il collaudo abbia svolto, in questo progetto, un ruolo attivo nella validazione e nella correzione dell'architettura, e non una semplice verifica a posteriori. Analogamente, la suite E2E sviluppata con Playwright ha validato con successo i flussi critici di Command & Control e introdotto una strategia originale per l'ispezione di entità renderizzate su canvas WebGL, altrimenti opache agli strumenti di test convenzionali.

La validazione del modulo Station e dei comportamenti cinematici delle entità (OPD) ha seguito un percorso più accidentato. Una prima implementazione di test automatizzati era stata effettivamente realizzata nelle fasi iniziali del lavoro; i successivi e frequenti refactoring subiti dal motore C++ nel corso del tirocinio ne hanno tuttavia progressivamente invalidato i presupposti, rendendone necessaria la rimozione anziché il mantenimento di una suite ormai disallineata rispetto al comportamento reale del sistema. Si è scelto pertanto di consolidare il lavoro già

condotto in una Specifica Formale, la Matrice di Validazione, che pur non eseguita in forma automatizzata conserva un valore consolidato: costituisce infatti una specifica comportamentale completa e già strutturata secondo un formato testabile, pronta per essere convertita in una nuova suite automatizzata non appena l'architettura di backend raggiungerà un grado di stabilità sufficiente. Va riconosciuto, in questo senso, che il presente lavoro raggiunge un livello di validazione solido e stabile sul perimetro dell'interfaccia web e delle sue API, mentre lascia in eredità, come debito tecnico esplicito e documentato, la ricostruzione della validazione automatizzata del motore di simulazione e della sua integrazione con la GUI.

7.2 Applicabilità Industriale dell'Architettura Proposta

L'architettura risultante dal lavoro di tesi presenta diverse caratteristiche compatibili con un'evoluzione verso un prodotto di livello commerciale. La separazione netta tra Presentation Layer, Spatial Layer e Communication Layer consente di intervenire su un sottosistema (ad esempio l'introduzione di un nuovo protocollo di rete, o la sostituzione del motore geospaziale) senza impatti diretti sugli altri livelli. Il sistema di Data Transfer Object speculari tra TypeScript e C++ (Cap. 4) definisce un confine di validazione esplicito e verificabile tra client e motore di simulazione, un requisito tipicamente richiesto in ambito difesa per la certificazione e l'audit dei sistemi software. La conformità della simbologia tattica agli standard NATO MIL-STD-2525/APP-6 (Cap. 5) rappresenta inoltre un prerequisito di interoperabilità già soddisfatto rispetto a potenziali contesti di impiego interforze o multinazionali.

Permangono, tuttavia, alcuni scostamenti significativi rispetto a un prodotto pronto per la commercializzazione, in linea con quanto messo in luce dalla fase di validazione. Manca, allo stato attuale, un livello di autenticazione e controllo degli accessi (specificato ma non implementato nella corrente iterazione), così come l'integrazione effettiva con reti di simulazione distribuita DIS/HLA, per ora prevista solo a livello architetturale come nodo di futura estensione. La gestione di scenari con entità multiple assegnate a task coordinati (rilevata come lacuna nel par. 6.1.4) richiede l'introduzione di un componente centralizzato di arbitraggio spaziale, oggi assente. Tali estensioni non intaccano, tuttavia, la validità dell'impianto architetturale complessivo, ma ne rappresentano un naturale prolungamento, reso possibile proprio dalla modularità introdotta nel corso del presente lavoro.

7.3 Sviluppi Futuri

7.3.1 Consolidamento del Testing Automatizzato

La priorità più immediata riguarda la conversione della Matrice di Validazione (par. 6.1) in una suite di test automatizzata, da condursi non appena l'architettura del motore C++ raggiungerà una cadenza di refactoring sufficientemente stabile. Tale suite dovrà essere integrata in una pipeline aziendale di Continuous Integration, così da intercettare tempestivamente le regressioni introdotte da futuri interventi sul backend. Parallelamente, la copertura della suite di regressione delle API REST dovrà essere estesa agli errori di sistema di livello 5xx, oggi esclusi per l'assenza di strumenti di environmental fault injection: l'introduzione di meccanismi controllati per simulare la disconnessione della Station o l'arresto improvviso del motore di simulazione permetterebbe di validare anche i percorsi di gestione degli errori più critici. Da ultimo, gli endpoint /api/simulation/waypoint/assign e /api/simulation/route/assign, individuati come architetturalmente scorretti nel corso della validazione (par. 6.2.4), dovranno essere ridisegnati e corredati di una nuova suite di test prima della loro messa in produzione definitiva.

7.3.2 Estensioni Funzionali e Interoperabilità

Il documento di specifica prodotto in fase di analisi dei requisiti (Cap. 3) individua un insieme di funzionalità non ancora implementate, che costituiscono il naturale successivo perimetro di sviluppo: l'editor avanzato dei modelli entità con classificazione SISO (UC 12), le dashboard analitiche per la consultazione degli avvisi operativi (UC 42) e il sottosistema di autenticazione e gestione degli accessi (UC 50-53). Di particolare rilievo strategico è l'integrazione con reti di simulazione distribuita DIS/HLA, per la quale l'architettura prevede già un punto di innesto (par. 3.1.2); tale sviluppo potrà avvalersi dell'approccio a gateway basato su RPR-FOM discusso nel Capitolo 2, così da garantire l'interoperabilità con federazioni HLA eterogenee senza rinunciare al DIS come dorsale di comunicazione. Sul piano della logica tattica, l'introduzione di un Formation Manager centralizzato (par. 6.1.4) risolverebbe le criticità di conflitto spaziale emerse nella gestione di task assegnati a più entità contemporaneamente.

7.3.3 Verso la Commercializzazione della Piattaforma

Un'evoluzione verso l'impiego commerciale della piattaforma richiede infine di affrontare aspetti che esulano dalla sola componente tecnica. La gestione di sessioni operative con più operatori simultanei, solo parzialmente affrontata dai test di idempotenza descritti nel par. 6.3.3, dovrà essere estesa a scenari di reale concorrenza multi-utente, con meccanismi espliciti di sincronizzazione e

risoluzione dei conflitti. Sarà inoltre necessario ampliare la copertura dei profili OPD oltre le classi di velivoli ad ala fissa e rotante attualmente coperte dalla Matrice di Validazione, per includere assetti navali e terrestri e avvicinare così la piattaforma allo spettro multi-dominio offerto da soluzioni di riferimento come VR-Forces. Su un piano più organizzativo, la trasformazione in prodotto commerciale imporrà la definizione di processi di documentazione, licensing e supporto tipici del software enterprise, elementi che esulano dal perimetro della presente tesi ma che ne rappresentano la naturale prosecuzione industriale.

7.4 Considerazioni Conclusive

Il percorso descritto in questo elaborato ha condotto un sistema di simulazione CGF da uno stato embrionale a un'architettura web strutturata, modulare e in larga parte validata, in linea con l'obiettivo dichiarato in apertura del lavoro. Il contributo principale non risiede in una singola componente isolata, quanto nella coerenza complessiva tra le scelte di analisi dei requisiti, progettazione architetturale, implementazione e validazione, che hanno permesso di trasformare un prototipo funzionante in una base tecnica solida e riutilizzabile. Gli sviluppi futuri delineati in questo capitolo, dal consolidamento del testing automatizzato all'integrazione con le reti di simulazione distribuita, fino agli aspetti organizzativi della commercializzazione, non rappresentano contraddizioni rispetto al lavoro svolto, bensì la sua più naturale prosecuzione: la dimostrazione, in altre parole, che l'architettura proposta è stata costruita per crescere.

Appendice A

Diagrammi degli altri Casi d'Uso

A completamento dell'analisi architetturale presentata nel Capitolo 3, questa appendice raccoglie i diagrammi dei casi d'uso relativi ai sottosistemi non dettagliati in precedenza. Nello specifico, vengono illustrati i modelli UML per i moduli responsabili del controllo temporale della simulazione, dell'interoperabilità di rete e della gestione ambientale dello scenario.

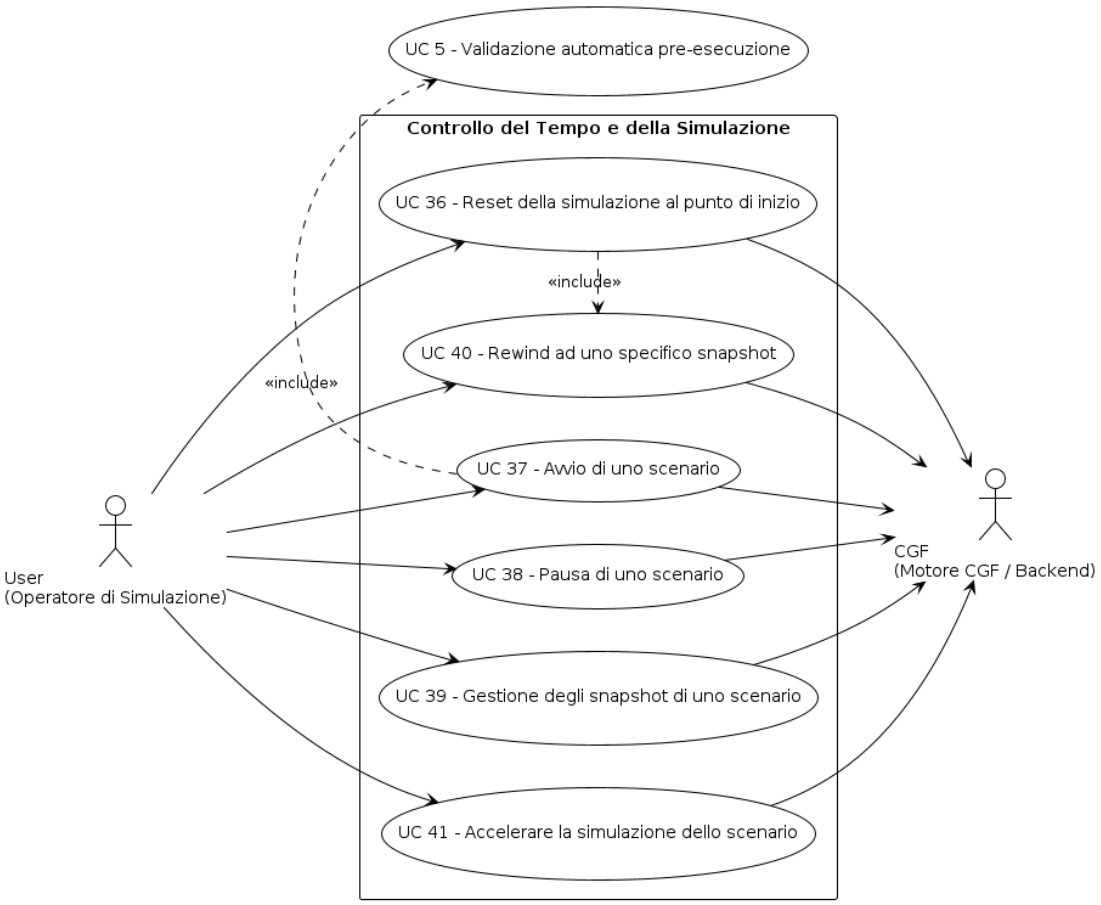


Figura A.1: Diagramma dei Casi d'Uso relativo al package "Controllo del Tempo e della Simulazione".

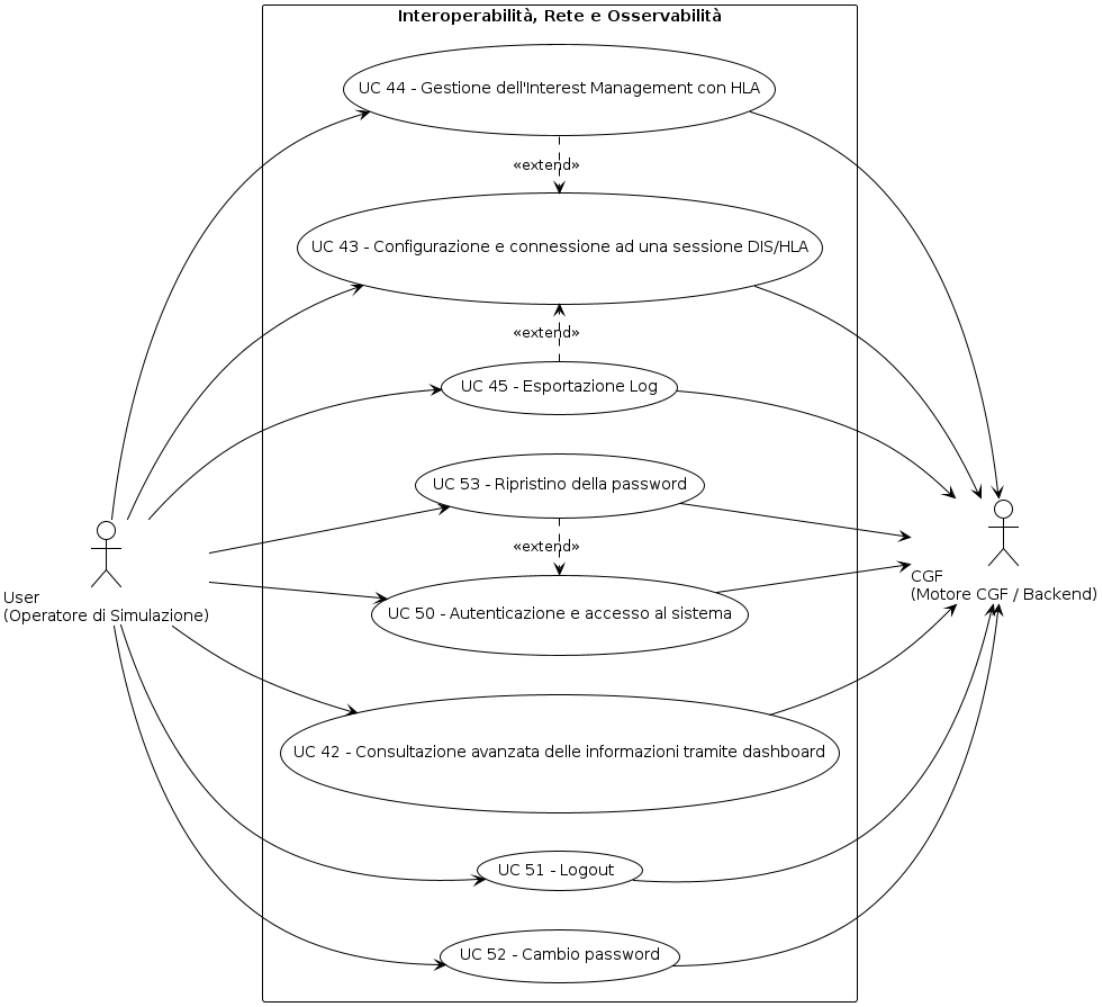


Figura A.2: Diagramma dei Casi d'Uso relativo al package "Interoperabilità, Rete e Osservabilità".

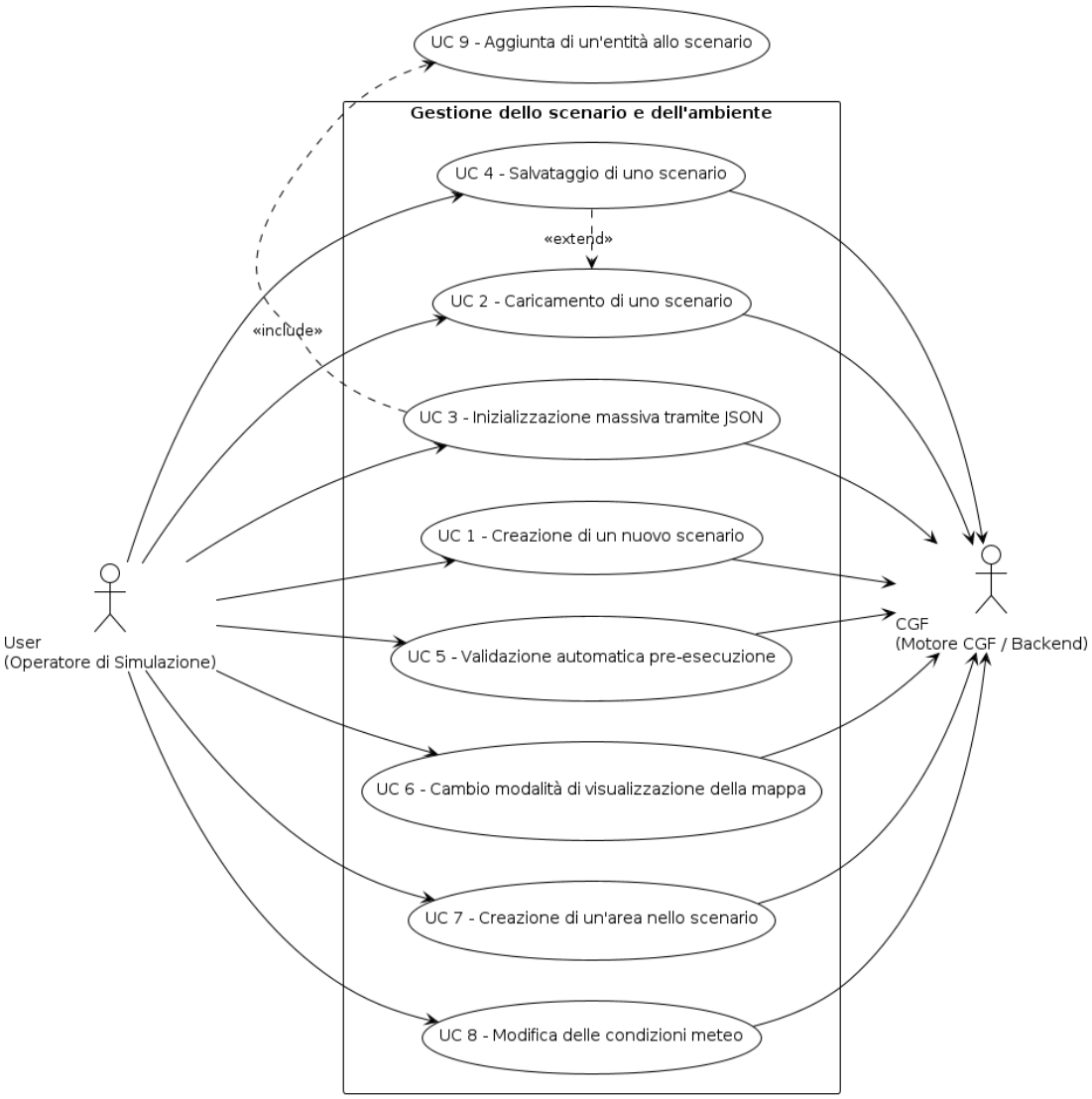


Figura A.3: Diagramma dei Casi d'Uso relativo al package "Gestione dello scenario e dell'ambiente".

Bibliografia

[1] *La crescente importanza del Modelling and Simulation*. URL: https://www.cesi-italia.org/it/articoli/la-crescente-importanza-del-modelling-and-simulation (cit. a p. 2).

[2] *Training Through Simulation in the Air Defence Artillery: Fire Control System Gun*Star Night Simulator*. URL: https://www.afahc.ro/ro/revista/Nr_2_2010/2.2010%20Cioca.pdf (cit. a p. 2).

[3] *La simulazione in ambito Forza Armata*. URL: https://www.esercito.difesa.it/comunicazione/la-simulazione-in-ambito-forza-armata/86286.html (cit. a p. 3).

[4] *Lavi Cockpit Immagine*. URL: https://commons.wikimedia.org/wiki/File:Lavi_Cockpit.jpg (cit. a p. 4).

[5] *Modeling and Simulation (M&S) Glossary*. URL: https://www.acqnotes.com/Attachments/DoD%20M&S%20Glossary%201%20Oct%2011.pdf (cit. a p. 4).

[6] *Schermata di scenario CPX*. URL: https://commons.wikimedia.org/wiki/File:A_screen_capture_depicts_a_combat_situation_in_a_digital_world_during_a_command_post_exercise_inside_the_Joint_Multinational_Simulations_Center_at_Joint_Multinational_Training_Command%27s_Camp_Aachen_in_120223-A-RT793-109.jpg (cit. a p. 4).

[7] *Data-driven behavior modeling for computer generated forces – a literature survey*. URL: https://www.ffi.no/en/publications-archive/data-driven-behavior-modeling-for-computer-generated-forces-a-literature-survey (cit. alle pp. 5, 7).

[8] *Investigating the application of web-based simulation principles within the architecture for a next-generation computer generated forces model*. URL: https://tinyurl.com/4rra6ckt (cit. a p. 6).

[9] *NATO Modelling and Simulation Standards Profile*. URL: https://www.mscoe.org/content/uploads/2017/12/AMSP-01C.pdf (cit. a p. 7).

[10] *The Modeling and Analysis of Computer Generated Forces Representing Groups and Organizations in Military Conflict Zones*. URL: https://www.foi.se/download/18.7fd35d7f166c56ebe0b1008b/1542623793977/The-modeling-and-analysis_FOI-S--4126--SE.pdf (cit. a p. 8).

[11] *The MAK ONE Multi-Domain Computer Generated Forces*. URL: https://www.mak.com/mak-one/apps/vr-forces (cit. a p. 9).

[12] *Department of Defense Modeling and Simulation Standards*. URL: https://github.com/open-dis/dis-tutorial/blob/master/DoDModelingAndSimulationStandards.md (cit. a p. 13).

[13] *The DIS Protocol - New and Improved*. URL: https://dl.acm.org/doi/pdf/10.5555/1639809.1639882 (cit. a p. 13).

[14] *Architecture of a Distributed Interactive Simulation Application*. URL: https://www.researchgate.net/figure/Architecture-of-a-Distributed-Interactive-Simulation-Application_fig1_3846540 (cit. a p. 13).

[15] *HLA Architecture*. URL: http://pnp-software.com/eodisp/documentation/developerManuals/hlaArchitecture.html (cit. a p. 15).

[16] *An introduction to developing federations with the High Level Architecture (HLA)*. URL: https://ieeexplore.ieee.org/document/8247820#full-text-header (cit. a p. 16).

[17] *MIXR IITSEC 2006 Paper*. URL: https://www.mixr.dev/assets/pages/docs/iitsec-2006-paper.pdf (cit. a p. 20).

[18] *JSBSim Reference Manual*. URL: https://jsbsim.sourceforge.net/JSBSimReferenceManual.pdf (cit. alle pp. 21, 23).

[19] *Unified behavior framework for discrete event simulation systems*. URL: https://apps.dtic.mil/sti/pdfs/ADA615242.pdf (cit. alle pp. 25, 27).

[20] *Vite - getting started*. URL: https://vite.dev/guide/ (cit. a p. 27).

[21] *React (software)*. URL: https://en.wikipedia.org/wiki/React_(software) (cit. a p. 28).

[22] *TypeScript*. URL: https://en.wikipedia.org/wiki/TypeScript (cit. a p. 28).

[23] *CesiumJS*. URL: https://cesium.com/platform/cesiumjs/ (cit. a p. 29).

[24] *Tailwind CSS*. URL: https://en.wikipedia.org/wiki/Tailwind_CSS (cit. a p. 30).

[25] *shadcn/ui*. URL: https://ui.shadcn.com (cit. a p. 31).

[26] *Radix UI*. URL: https://www.radix-ui.com (cit. a p. 31).

[27] *The WebSocket Protocol*. URL: https://datatracker.ietf.org/doc/html/rfc6455 (cit. a p. 32).