UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame
e Realtà Virtuale

Ingegnerizzazione di una Pipeline Dati e Architettura Unity per la Visualizzazione di Umani Virtuali Parametrici (SMPL-H)

Candidato:
Luca Felcher

Relatore:
Dott. Lorenzo Stacchio

Correlatore:
Prof. Adriano Mancini
2025-2026

Anno Accademico 2025–2026

UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame
e Realtà Virtuale

Ingegnerizzazione di una Pipeline Dati e Architettura Unity per la Visualizzazione di Umani Virtuali Parametrici (SMPL-H)

Candidato:
Luca Felcher

Relatore:
Dott. Lorenzo Stacchio

Correlatore:
Prof. Adriano Mancini
2025-2026

Anno Accademico 2025–2026

Università Politecnica delle Marche
Facoltà di Ingegneria
Corso di Laurea in Ingegneria dell'Informazione per Videogame e Realtà Virtuale
Via Brecce Bianche – 60131 Ancona (AN), Italia

"All that is gold does not glitter, Not all those who wander are lost; The old that is strong does not wither, Deep roots are not reached by the frost. From the ashes a fire shall be woken, A light from the shadows shall spring; Renewed shall be blade that was broken, The crownless again shall be king." — J.R.R. Tolkien, The Fellowship of the Ring

Sommario

La sintesi e la generazione di animazioni 3D realistiche per avatar virtuali rappresentano alcune delle sfide più complesse nell'ambito della *Computer Graphics*. Recentemente, la ricerca si sta orientando verso l'uso dell'Intelligenza Artificiale Generativa Agentica (Agentic GenAI) per superare i limiti strutturali dei tradizionali modelli *end-to-end* [1, 2]. In questo nuovo paradigma di co-creazione Uomo-IA [3], un *Large Language Model* (LLM) funge da "regista digitale", scomponendo query testuali complesse in task sequenziali [4] e recuperando frammenti di movimento continuo da vasti database di Motion Capture spazialmente annotati.

Il presente lavoro di tesi si inserisce in questo contesto, affrontando la progettazione, lo sviluppo e l'ingegnerizzazione dell'intera infrastruttura software necessaria a supportare tale orchestrazione. Il problema principale affrontato consiste nel superare l'incompatibilità strutturale tra i grandi volumi di dati cinematici grezzi per modelli corporei parametrici, nello specifico lo standard SMPL-H [5], e l'ambiente di visualizzazione *real-time* Unity [6].

Il metodo proposto si è articolato su due fronti implementativi interdipendenti. Lato *back-end*, è stata sviluppata una *Data Pipeline* in Python, capace di estrarre, omogeneizzare e convertire asincronamente i tensori complessi provenienti dal benchmark AMASS [7], dal dataset linguistico HumanML3D [8] e dal database coreografico AIST++ [9, 10]. Tale modulo ha riscontrato complesse discrepanze geometriche, quali il riallineamento dei sistemi di coordinate tridimensionali (Z-up vs Y-up) mediante matrici di trasformazione spaziale [11] e il *padding* differenziale delle catene cinematiche tra il modello base SMPL [12], SMPL-H e SMPL+H di AMASS, che non sono state risolte completamente.

Lato *front-end*, l'architettura nativa del framework di visualizzazione *bmlSUP* [13] per il motore grafico Unity è stata riprogettata applicando il Design Pattern *Strategy*. È stato sviluppato un ecosistema C# reattivo e disaccoppiato in grado di deserializzare dinamicamente a runtime i file JSON prodotti, gestendo la regia e l'offset spaziale delle animazioni attraverso un sistema asincrono basato su *Coroutines* ed Eventi.

I risultati ottenuti dimostrano che la pipeline sviluppata riduce i tempi di processamento e i limiti di allocazione di alcuni degli applicativi originali. L'infrastruttura prodotta trasforma l'eseguibile Unity in un modulo di rendering totalmente passivo, scalabile e *fault-tolerant*, formalmente predisposto per ricevere file di manifesto compilati dall'Agente Logico esterno. Tale architettura costruisce di fatto il ponte ingegneristico essenziale per le future integrazioni di sistemi agentici in ambienti di simulazione distribuita.

Indice

1 Introduzione **1**

2 Stato dell'Arte e Lavori Correlati **5**
2.1 Modelli Parametrici del Corpo Umano 5
2.2 Dataset per la Cattura del Movimento e Annotazioni Semantiche . . 6
2.3 Sintesi del Movimento (Text-to-Motion) e Modelli Generativi 8
2.4 Motion Retrieval e Spazi Latenti Multimodali 9
2.5 Generazione di Sequenze Lunghe e Composizione Temporale 9
2.6 Motion Blending e Retargeting Cinematico 10
2.7 Sistemi Agentici e Co-Creazione Uomo-IA 10
2.7.1 Co-Creazione e Paradigma Human-in-the-Loop 13
2.8 Integrazione in Ambienti Real-Time e Limiti Architetturali 13

3 Ingegnerizzazione della Pipeline Dati e Conversione **14**
3.1 Obiettivi e Architettura Generale 14
3.2 Struttura degli Input e Formati Originali 16
3.2.1 Serializzazione NumPy in AMASS (`.npz`) 16
3.2.2 Serializzazione Pickle in AIST++ (`.pkl`) 17
3.3 Moduli di Conversione e Risoluzione Matematica 18
3.3.1 Conversione da Asse-Angolo a Quaternioni 18
3.3.2 Proposta per la Trasformazione dei Sistemi di Coordinate (Z-up $\rightarrow$ Y-up) 19
3.4 Ottimizzazione del Transfer Model e Padding Topologico 20
3.4.1 Tuning degli Iperparametri e Rilassamento delle Tolleranze . 20
3.5 Ottimizzazione Computazionale: Multiprocesso a Memoria Separata 21
3.5.1 Gestione dei Worker e Allocazione del Carico 21
3.6 Gestione delle Eccezioni e Tolleranza ai Guasti 22
3.7 Struttura del Formato JSON 22

4 Progettazione e Implementazione dell'Architettura in Unity **24**
4.1 Analisi e Limiti del Framework Originale bmlSUP 24
4.1.1 Vincoli del Thread Principale e Gestione della Memoria . . . 24
4.1.2 Limiti della Logica di Riproduzione Statica 24
4.2 Architettura del Sistema e Responsabilità delle Classi 25
4.2.1 Il Pattern Facade: SUPSequenceManager 26
4.2.2 Architettura Event-Driven per il Controllo del Flusso 26

4.3 Disaccoppiamento del Parser: Lo Strategy Pattern 26
4.4 Gestione dei Dati Tramite ScriptableObject 27
4.5 Livello di Configurazione e Parametrizzazione (Data-Driven) 27
4.5.1 Gestione Morfologica: Body Settings 28
4.5.2 Visualizzazione e Diagnostica: Display Settings 29
4.5.3 Regolazione Temporale: Playback Settings 29
4.6 Caricamento Esterno Asincrono: Da File a Memoria 30
4.6.1 I/O su Disco e Allocazione delle Stringhe 30
4.6.2 Deserializzazione Asincrona dei Tensori 30
4.7 Flusso di Esecuzione e Integrazione nell'Editor 31
4.8 Il Motore di Regia: SUPPlayer_Custom e Coroutines 32
4.8.1 Calcolo dell'Offset Spaziale e Sviluppi Futuri della Cinematica 33
4.8.2 Attesa Attiva e Ciclo di Cleanup 34

5 Risultati Sperimentali e Discussione 35
5.1 Ambiente di Test e Specifiche Hardware 35
5.2 Scalabilità della Pipeline Python: Analisi Multicore 35
5.3 Impatto dello Storage e Struttura Dati 36
5.3.1 Prestazioni del Motore di Rendering Unity 37
5.4 Ottimizzazione del Transfer Model AIST++ 37
5.5 Analisi Qualitativa e Limiti Topologici nel Dataset AIST++ 38
5.5.1 Ipotetiche Cause degli Artefatti Geometrici 38
5.5.2 Prospettive di Risoluzione 39

6 Conclusioni e Sviluppi Futuri 41
6.1 Sintesi dei Risultati 41
6.2 Limiti e sviluppi futuri 42

Ringraziamenti 45

Elenco delle figure

1.1 Rappresentazione concettuale del modello parametrico SMPL. L'immagine illustra l'influenza dei coefficienti di forma ($\boldsymbol{\beta}$) sulla morfologia della mesh a 6890 vertici e il controllo cinematico esercitato dai parametri di posa ($\boldsymbol{\theta}$) sulla struttura articolare a 24 giunti. Fonte: adattata da Loper et al. [12]. 1
1.2 Confronto tra i modelli parametrici SMPL e SMPL-H. A sinistra è mostrata la configurazione base di SMPL, mentre a destra è rappresentata l'estensione che integra il modello MANO per il controllo articolare delle mani. Fonte: adattata da Pavlakos et al. [14]. 2

2.1 Architettura concettuale di un modello di diffusione condizionato per il *Text-to-Motion*. Il diagramma illustra l'iniezione dei *text-embedding* estratti tramite modelli linguistici per condizionare il processo iterativo di rimozione del rumore (Denoising) sui tensori cinematici [1]. 9
2.2 Rappresentazione del ciclo operativo ReAct [4] applicato all'orchestrazione di *performance* virtuali. Il pattern evidenzia la delega del carico computazionale cinematico a database esterni (es. AMASS), limitando il compito dell'LLM alla pura pianificazione logica delle azioni. 12

3.1 Diagramma di flusso della pipeline di conversione per il dataset AMASS. Il processo evidenzia l'estrazione dai file `.npz`, la trasformazione matematica in Python e l'esportazione verso il *plugin bmlSUP Custom* in Unity. 15
3.2 Diagramma di flusso della variante della pipeline applicata ad AIST++. In questo caso, il flusso include uno step obbligatorio per il *padding* topologico da SMPL (24 giunti) a SMPL-H (52 giunti). . . . 16
3.3 Struttura tensoriale di un file `.npz` estratto dal dataset AMASS. Si noti la dimensione dell'array `poses` $[F, 156]$ che copre l'intera topologia di mani e corpo. 17
3.4 Albero strutturale di un file `.pkl` estratto da AIST++. Le dimensioni del tensore `smpl_poses` $(N, 24, 3)$ e l'assenza dei coefficienti $\boldsymbol{\beta}$ evidenziano l'incompatibilità diretta con il *player* basato su SMPL-H. 18
3.5 Albero strutturale dell'oggetto JSON generato dalla pipeline. Il formato include le primitive per i metadati e le liste annidate contenenti la cinematica. 23

4.1 Diagramma delle classi (UML) del sottosistema Unity implementato. Il grafo evidenzia il ruolo orchestratore di `SUPSequenceManager`, il modulo di estrazione a *runtime* `SUPExternalLoader` e il pattern *Strategy* adottato per il parser JSON. 25
4.2 Diagramma di Sequenza del sistema. Il flusso evidenzia la transizione dalla fase asincrona C# (Task) all'avvio delle Coroutines nel *Main Thread* di Unity per la gestione della regia temporale. 31
4.3 Schermata dell'Editor di Unity [6]. La finestra *Inspector* a destra mostra l'interfaccia personalizzata del `SUPSequenceManager`, impiegata per la definizione dei parametri di I/O e riproduzione. 32
4.4 Visualizzazione di *batch* multipli in Unity. Si nota l'applicazione della logica di *Offset Spaziale* attuale: ciascun modello è istanziato a una distanza orizzontale costante determinata dal parametro `OffSetSpacing`, evitando la sovrapposizione delle *mesh* poligonali. 34

5.1 Artefatto cinematico riscontrato durante l'integrazione di AIST++. La mesh mostra una torsione innaturale del tronco e un errato allineamento degli arti, sintomo di una discordanza nel sistema di riferimento locale del giunto radice. 38

Elenco delle tabelle

2.1 Confronto analitico dello stato dell'arte dei dataset per l'analisi cinematica del corpo umano virtuale. La tabella evidenzia le eterogeneità strutturali che la *Data Pipeline* sviluppata in questa tesi è stata ingegnerizzata per omogeneizzare. 7

5.1 Risultati del benchmark di parallelizzazione per la conversione JSON. Si nota il picco di efficienza massima a $p = 4$, seguito da un degrado prestazionale all'aumentare dei thread. 36

5.2 Prestazioni osservate durante la riproduzione di un numero crescente di animazioni concorrenti in Unity 2022.3. Il tempo medio per frame è stato calcolato a partire dal framerate medio. 37

Capitolo 1

Introduzione

La rappresentazione e l'animazione realistica di umani virtuali costituiscono un tema centrale nell'ambito della *Computer Graphics*, della simulazione interattiva e dello sviluppo di applicazioni basate su motori grafici real-time. In questi contesti, i modelli corporei parametrici consentono di descrivere in modo compatto sia la morfologia sia il movimento del corpo umano, rendendo possibile la riproduzione di sequenze cinematiche acquisite mediante sistemi di *Motion Capture*.

Tra le soluzioni maggiormente diffuse si colloca SMPL (*Skinned Multi-Person Linear Model*), un modello parametrico basato su una mesh a topologia fissa composta da 6890 vertici [12]. La configurazione del corpo è controllata mediante parametri di forma, indicati con $\boldsymbol{\beta}$, che descrivono le caratteristiche antropometriche del soggetto, e parametri di posa, indicati con $\boldsymbol{\theta}$, che definiscono le rotazioni dei 24 giunti della struttura scheletrica. Tale rappresentazione permette di separare la morfologia del soggetto dal movimento eseguito, fornendo una descrizione compatta e standardizzata dell'avatar.

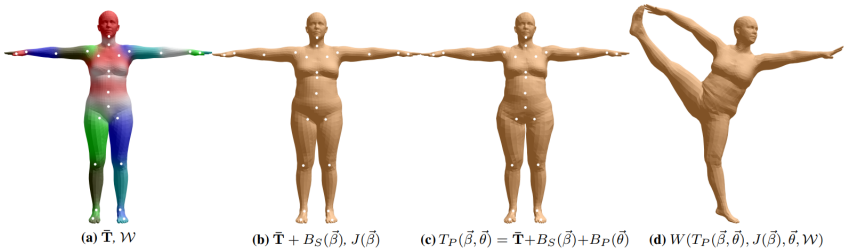

Figura 1.1: Rappresentazione concettuale del modello parametrico SMPL. L'immagine illustra l'influenza dei coefficienti di forma ($\boldsymbol{\beta}$) sulla morfologia della mesh a 6890 vertici e il controllo cinematico esercitato dai parametri di posa ($\boldsymbol{\theta}$) sulla struttura articolare a 24 giunti. Fonte: adattata da Loper et al. [12].

Il modello SMPL originale presenta tuttavia una rappresentazione limitata delle mani, le cui articolazioni non vengono descritte in modo indipendente. Per superare tale limite è stato introdotto MANO, un modello parametrico specializzato nella rappresentazione delle mani, successivamente integrato nel corpo SMPL per ottenere SMPL-H [5]. Quest'ultimo estende la struttura articolare fino a 52 giunti, includendo 30 articolazioni dedicate alle dita delle due mani. L'adozione di SMPL-H consente

quindi di rappresentare movimenti corporei e gestualità fini all'interno di una struttura parametrica unificata.

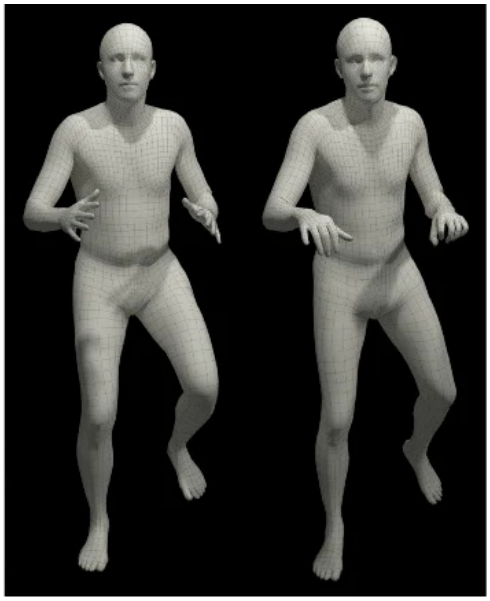

Figura 1.2: Confronto tra i modelli parametrici SMPL e SMPL-H. A sinistra è mostrata la configurazione base di SMPL, mentre a destra è rappresentata l'estensione che integra il modello MANO per il controllo articolare delle mani. Fonte: adattata da Pavlakos et al. [14].

La disponibilità di tali modelli ha favorito la creazione di grandi raccolte di dati cinematici. AMASS (*Archive of Motion Capture Assembled from Animated Body Shapes*) riunisce numerosi dataset di *Motion Capture* all'interno di una rappresentazione basata sui parametri di SMPL-H [7]. Le sequenze includono parametri di posa, traslazioni globali e coefficienti morfologici, rendendo AMASS una delle principali risorse per l'analisi e la generazione del movimento umano.

AIST++ rappresenta invece un dataset orientato alle coreografie e ai movimenti di danza [9, 10]. Le sequenze sono descritte mediante il modello SMPL standard a 24 giunti e non includono i coefficienti di forma necessari per ricostruire la morfologia originale dei soggetti. Questa differenza introduce una rilevante incompatibilità rispetto a un sistema di visualizzazione basato su SMPL-H, che richiede 52 rotazioni articolari per ciascun fotogramma.

Nonostante la disponibilità di dataset estesi e modelli parametrici standardizzati, l'integrazione di tali risorse in un motore grafico real-time presenta numerose difficoltà. I dati sono distribuiti mediante formati differenti, come `.npz` per AMASS e `.pkl` per AIST++, utilizzano strutture tensoriali eterogenee e possono adottare convenzioni spaziali non compatibili. Inoltre, i framework scientifici utilizzati per la loro elaborazione sono generalmente sviluppati in Python, mentre gli ambienti interattivi come Unity impiegano prevalentemente il linguaggio C# e strutture dati differenti.

Il caricamento diretto dei formati scientifici all'interno di Unity richiederebbe parser specifici e operazioni di trasformazione potenzialmente onerose durante l'esecuzione. Il framework bmlSUP [13], utilizzato come punto di partenza per la visualizzazione degli avatar SMPL, prevede inoltre un caricamento prevalentemente statico delle animazioni tramite oggetti `TextAsset` inclusi nel progetto. Tale approccio richiede che le sequenze siano importate prima della compilazione e limita la possibilità di aggiungere o modificare dati durante l'esecuzione dell'applicazione.

Il presente lavoro di tesi affronta queste problematiche attraverso la progettazione e l'implementazione di una pipeline software composta da due sottosistemi principali. Il primo è un modulo di elaborazione sviluppato in Python, incaricato di leggere i dati cinematici originali, convertirne le rappresentazioni rotazionali, uniformarne la struttura e produrre file JSON compatibili con il sistema di visualizzazione. La pipeline gestisce in particolare le sequenze provenienti da AMASS e sperimenta il trasferimento delle animazioni di AIST++ dalla topologia SMPL a quella SMPL-H. La conversione da asse-angolo a quaternioni viene effettuata nella fase di pre-elaborazione, riducendo il numero di trasformazioni richieste durante il runtime. Nel caso di AIST++, il processo comprende inoltre un tentativo di trasferimento del modello e l'inserimento dei dati articolari mancanti per le mani. La completa correzione delle convenzioni spaziali e degli artefatti cinematici di AIST++ non è tuttavia conclusa e costituisce uno dei principali limiti del lavoro. Per ridurre i tempi di elaborazione dei dataset, il convertitore Python è stato organizzato secondo un'architettura multiprocesso a memoria separata. I file vengono distribuiti tra più worker indipendenti, mentre un sistema di gestione delle eccezioni impedisce che la presenza di un file non valido interrompa l'elaborazione dell'intero archivio. L'output prodotto è costituito da file JSON contenenti metadati, parametri morfologici, traslazioni e quaternioni articolari per ciascun fotogramma.

Il secondo sottosistema è costituito da un'architettura personalizzata sviluppata in C# all'interno di Unity [6]. Il framework bmlSUP è stato esteso con l'obiettivo di eliminare la dipendenza dal caricamento statico tramite `TextAsset` e consentire la lettura di animazioni memorizzate esternamente al progetto. Sono stati introdotti componenti dedicati alla lettura del manifesto delle sequenze, alla gestione dei file JSON, alla deserializzazione dei dati e alla riproduzione ordinata delle animazioni. L'architettura applica pattern software quali *Facade* e *Strategy*, separando la sorgente dei dati dalla logica di parsing e dal sistema di visualizzazione. È stato inoltre sviluppato un player personalizzato capace di gestire gruppi di animazioni, riprodurre le clip in sequenza e applicare offset spaziali agli avatar, evitando la sovrapposizione delle mesh all'interno della scena.

L'obiettivo principale della tesi non consiste nella generazione diretta di nuove animazioni mediante un modello di Intelligenza Artificiale, ma nella realizzazione dell'infrastruttura dati e grafica necessaria per rendere le sequenze cinematiche accessibili, convertibili e visualizzabili dinamicamente. Il sistema sviluppato costituisce pertanto uno strato intermedio tra i dataset di *Motion Capture* e il motore Uni-

ty, riducendo l'accoppiamento tra i formati scientifici originali e il sottosistema di rendering.

Questa impostazione predispone inoltre l'architettura a future integrazioni con sistemi di pianificazione e orchestrazione agentica. In uno scenario successivo, un modello linguistico potrebbe interpretare una richiesta testuale, selezionare le animazioni pertinenti, ordinarle temporalmente e generare automaticamente il manifesto utilizzato dal client Unity. L'integrazione effettiva dell'agente LLM, la comunicazione tramite socket, il *motion blending*, la gestione dinamica delle coordinate spaziali e la correzione completa delle animazioni AIST++ restano tuttavia al di fuori del perimetro implementativo della presente tesi.

Il sistema è stato valutato attraverso analisi quantitative e qualitative. Dal lato Python sono stati misurati i tempi di conversione al variare del numero di worker, lo *speedup* ottenuto e l'incremento dello spazio di archiviazione derivante dalla serializzazione JSON. Dal lato Unity sono stati osservati il framerate medio, il tempo medio per frame, l'occupazione della memoria e il comportamento del sistema durante la riproduzione contemporanea di più avatar. L'analisi comprende inoltre gli artefatti riscontrati nel trasferimento delle sequenze AIST++, evidenziando i limiti legati alla differente topologia scheletrica e alle convenzioni dei sistemi di coordinate.

La tesi è organizzata come segue. Il Capitolo 2 presenta i modelli parametrici del corpo umano, i principali dataset cinematici e gli approcci esistenti alla sintesi, al recupero e alla composizione del movimento. Il Capitolo 3 descrive la pipeline Python, i formati di input, le trasformazioni applicate e le strategie di parallelizzazione. Il Capitolo 4 illustra l'architettura sviluppata in Unity, le classi implementate e il flusso di caricamento e riproduzione. Il Capitolo 5 riporta i risultati sperimentali e ne discute i principali limiti. Infine, il Capitolo 6 sintetizza i contributi del lavoro e presenta le possibili evoluzioni future.

Capitolo 2

Stato dell'Arte e Lavori Correlati

L'orchestrazione di umani virtuali attraverso paradigmi di Intelligenza Artificiale Agentica richiede la convergenza di molteplici domini di ricerca nell'ambito della *Computer Vision*, della *Computer Graphics* e dell'elaborazione del linguaggio naturale (NLP). Il presente capitolo analizza lo stato dell'arte relativo ai modelli parametrici per la rappresentazione corporea, ai database cinematici, alle tecniche di sintesi e recupero del movimento, e all'emergente campo della co-creazione Uomo-IA.

2.1 Modelli Parametrici del Corpo Umano

La rappresentazione digitale del corpo umano ha subito una radicale evoluzione passando da *mesh* poligonali statiche a modelli matematici guidati dai dati. Il modello SMPL (*Skinned Multi-Person Linear Model*) [12] rappresenta lo standard accademico e industriale per l'animazione 3D. Basato su un'architettura a 6890 vertici, SMPL supera i classici artefatti del *Linear Blend Skinning* (LBS), come il collasso dei giunti, introducendo *blendshape* correttive apprese statisticamente da migliaia di scansioni 3D di soggetti reali. Il modello scompone la variazione corporea in parametri di forma ($\boldsymbol{\beta}$) e parametri di posa ($\boldsymbol{\theta}$) espressi come rotazioni relative.

Nonostante l'efficacia sul corpo intero, il modello SMPL originale presenta limitazioni nella rappresentazione delle estremità. Per risolvere tale criticità, Romero et al. hanno introdotto il framework MANO [5], un modello parametrico specializzato per l'articolazione complessa delle mani, che a sua volta è stato fuso nel modello corpo originario dando origine a SMPL-H (SMPL + Hands). Tale estensione eleva i gradi di libertà del sistema, mappando 52 giunti articolari totali e consentendo la simulazione di gestualità fini.

La ricerca si è successivamente espansa verso modelli sempre più espressivi. SMPL-X [14] integra in un'unica topologia unificata il corpo, le mani e il volto (tramite il modello FLAME[15]), offrendo un controllo parametrico completo delle espressioni facciali. Parallelamente, approcci più recenti come STAR (*Sparse Trained Articulated Human Body Regressor*) [16] hanno ottimizzato l'architettura matematica di SMPL introducendo correzioni spazialmente limitate, riducendo il numero di parametri e migliorando la generalizzazione su corporature fuori distribuzione. Il presente

lavoro adotta SMPL-H come compromesso ottimale tra fedeltà articolare ed efficienza computazionale in ambito *real-time* [6].

Per comprendere i limiti affrontati nell'ingegnerizzazione della pipeline, è fondamentale analizzare la formulazione matematica alla base del modello SMPL [12]. Il framework calcola la posizione dei vertici della mesh $M(\boldsymbol{\beta}, \boldsymbol{\theta})$ applicando un operatore di Linear Blend Skinning (LBS) a una topologia di base deformata:

$$M(\boldsymbol{\beta}, \boldsymbol{\theta}) = \mathcal{W}\left(T_P(\boldsymbol{\beta}, \boldsymbol{\theta}), J(\boldsymbol{\beta}), \boldsymbol{\theta}, \mathbf{W}\right), \tag{2.1}$$

dove:

- $T_P(\boldsymbol{\beta}, \boldsymbol{\theta})$ rappresenta la mesh nella posa di riferimento, dopo l'applicazione delle blendshape dipendenti dalla forma e dalla posa;
- $J(\boldsymbol{\beta})$ rappresenta le coordinate dei giunti articolari, determinate in funzione dei parametri morfologici $\boldsymbol{\beta}$;
- $\boldsymbol{\theta}$ rappresenta il vettore dei parametri di posa, espresso mediante rotazioni locali in formato asse-angolo;
- $\mathbf{W}$ rappresenta la matrice dei pesi di skinning, che determina l'influenza di ciascun giunto sui vertici della mesh;
- $\mathcal{W}$ indica l'operatore di Linear Blend Skinning applicato al modello.

Nel modello SMPL standard, il vettore $\boldsymbol{\theta}$ contiene 72 parametri, corrispondenti a 24 giunti con tre componenti asse-angolo per ciascun giunto. L'estensione SMPL-H porta tale rappresentazione a 156 parametri, corrispondenti a 52 giunti.

2.2 Dataset per la Cattura del Movimento e Annotazioni Semantiche

L'addestramento di modelli generativi e di ricerca cinematica necessita di moli di dati massive e standardizzate. Storicamente, i dati di *Motion Capture* (MoCap) erano frammentati in formati proprietari e topologie disomogenee. Il rilascio di AMASS (*Archive of Motion Capture as Surface Shapes*) [7] ha uniformato oltre 15 dataset preesistenti, ottimizzando i marcatori ottici originali e convertendoli nei parametri posizionali di SMPL-H, generando un corpus di oltre 40 ore di movimento standardizzato.

Per abilitare l'interazione tra testo e animazione, la ricerca si è concentrata sull'annotazione semantica dei movimenti. Il dataset KIT Motion-Language [17] ha rappresentato uno dei primi sforzi per accoppiare descrizioni in linguaggio naturale a sequenze MoCap. Successivamente, BABEL [18] ha fornito annotazioni a livello di frame per l'intero archivio AMASS, catalogando azioni multiple e transizioni. Per

far fronte alle esigenze dell'IA generativa, sono stati sviluppati benchmark specifici come HumanAct12 [19] e, più recentemente, HumanML3D [8]. Quest'ultimo estrae feature cinematiche invarianti da AMASS e le associa a descrizioni testuali complesse composte da verbi, nomi e avverbi, diventando lo standard de facto per i task di *Text-to-Motion*.

Per i domini ad alta dinamicità, il dataset AIST++ [9] costituisce un punto di riferimento. Derivato dal database video multicamera AIST [10], mappa coreografie complesse sul modello SMPL base a 24 giunti. Sebbene privo di coefficienti di forma e di annotazioni semantiche lineari, l'elevata densità di frame e la complessità delle rotazioni articolari lo rendono essenziale per testare i limiti fisici dei motori di simulazione.

Al fine di sintetizzare le divergenze strutturali e le vocazioni applicative dei database analizzati, la Tabella 2.1 propone un confronto formale dei parametri cardine. Emerge chiaramente come il passaggio dall'addestramento algoritmico puro (AIST++) all'analisi semantica (HumanML3D) imponga variazioni profonde sia nella topologia scheletrica adottata, sia nella rappresentazione vettoriale delle rotazioni.

Parametro	**AMASS [7]**	**AIST++ [9]**	**HumanML3D [8]**
Dominio Applicativo	MoCap Generalista	Coreografia / Danza	Text-to-Motion
Formato Originale	`.npz` (Dizionari NumPy)	`.pkl` (Dizionari Pickle)	`.npy` (Tensor Array) + `.txt`
Topologia Scheletro	SMPL-H (52 Giunti)	SMPL Standard (24 Giunti)	SMPL Ristretto (22 Giunti)
Rappresentazione Rotazioni	Asse-Angolo (Axis-Angle)	Asse-Angolo (Tensore 3D)	Matrici continue 6D + Velocità
Parametri di Forma (β)	Presenti (16 coefficienti)	Assenti	Non applicabile
Sistema di Coordinate	Sinistroso (Y-up) / Destroso (Z-up) variabile	Destroso (Z-up)	Normalizzato all'origine

Tabella 2.1: Confronto analitico dello stato dell'arte dei dataset per l'analisi cinematica del corpo umano virtuale. La tabella evidenzia le eterogeneità strutturali che la *Data Pipeline* sviluppata in questa tesi è stata ingegnerizzata per omogeneizzare.

La proliferazione di molteplici database cinematici indipendenti ha generato un ecosistema frammentato, rendendo complessa l'interoperabilità dei dati. Oltre alle differenze nei formati di salvataggio, le discrepanze più critiche riguardano le convenzioni spaziali adottate in fase di cattura del movimento [10].

I sistemi di *Motion Capture* ottico e i framework di *Computer Vision* impiegano frequentemente sistemi di coordinate destrosi con l'asse Z orientato verso l'alto (*Z-up*), in cui il piano XY rappresenta il pavimento. Al contrario, la maggior parte dei software di simulazione 3D e motori grafici [6] adotta convenzioni sinistrose con l'asse Y come vettore verticale (*Y-up*). Questa divergenza richiede l'applicazione rigorosa

di trasformazioni matriciali e rotazioni di compensazione sulla radice della gerarchia scheletrica per evitare il ribaltamento degli avatar virtuali nell'ambiente target.

Un ulteriore livello di complessità è dato dalla risoluzione topologica. Database come AIST++ campionano il movimento limitandosi ai 24 giunti del modello SMPL base [12], tralasciando la cinematica delle estremità. L'esecuzione di tali animazioni su un *player* configurato per l'architettura a 52 giunti di SMPL-H genera eccezioni a *runtime* dovute alla mancanza dei vettori rotazionali attesi per le falangi. Risulta pertanto necessaria l'introduzione di procedure di *padding* dinamico, mediante l'inserimento di vettori di rotazione neutri per uniformare le impronte tensoriali prima della fase di riproduzione.

2.3 Sintesi del Movimento (Text-to-Motion) e Modelli Generativi

La generazione di animazioni tridimensionali a partire da istruzioni testuali (*Text-to-Motion*) ha registrato un cambio di paradigma con l'introduzione dei modelli di Diffusione. Mentre gli approcci precedenti basati su Reti Neurali Ricorrenti (RNN) tendevano a produrre movimenti ripetitivi o a collassare in pose medie su lunghe sequenze, le architetture probabilistiche contemporanee offrono alta varianza e realismo.

Il modello MDM (*Human Motion Diffusion Model*) [1] sintetizza direttamente i tensori geometrici applicando il processo di denoise iterativo sui parametri spaziali. Approcci alternativi si basano invece sulla quantizzazione del movimento: T2M-GPT [2] sfrutta architetture basate su Transformer per prevedere sequenze di token discreti, analogamente alla generazione di testo, traducendoli successivamente in animazioni continue tramite un *Vector Quantized Variational Autoencoder* (VQ-VAE). Per abbattere i costi computazionali, modelli come MLD (*Motion Latent Diffusion*) [20] spostano il processo di diffusione dallo spazio cinematico ad alta dimensionalità a uno spazio latente compresso, migliorando i tempi di inferenza mantenendo la coerenza fisica.

Il paradigma alla base dei moderni sistemi come MDM [1] si fonda sul processo di *Denoising Diffusion Probabilistic Models* (DDPM)[21]. Come schematizzato in Figura 2.1, il modello apprende a invertire un processo di corruzione markoviano. Partendo da un tensore composto interamente da rumore gaussiano puro, la rete neurale (tipicamente un'architettura U-Net o Transformer) rimuove iterativamente il rumore prevedendo la cinematica corretta al *time-step* precedente. Il controllo semantico viene garantito iniettando le *feature* estratte dal testo testuale (tramite modelli come CLIP[22]) all'interno dei layer di *Cross-Attention* della rete, condizionando matematicamente la direzione del *denoising* verso sequenze di movimento coerenti con l'input dell'utente.

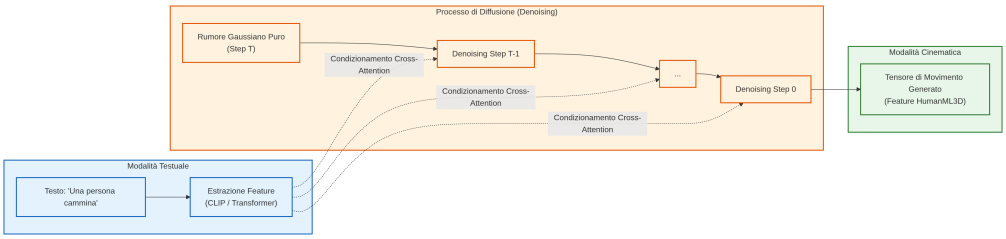

Figura 2.1: Architettura concettuale di un modello di diffusione condizionato per il *Text-to-Motion*. Il diagramma illustra l'iniezione dei *text-embedding* estratti tramite modelli linguistici per condizionare il processo iterativo di rimozione del rumore (Denoising) sui tensori cinematici [1].

2.4 Motion Retrieval e Spazi Latenti Multimodali

A differenza della sintesi pura, il *Motion Retrieval* consiste nella ricerca e nel recupero di sequenze preesistenti all'interno di un database in risposta a una query. Questo approccio garantisce la totale assenza di artefatti generativi (come la compenetrazione poligonale o il *foot skating*), basandosi su registrazioni fisicamente corrette.

Il successo di questi sistemi dipende dall'efficacia dell'allineamento tra la modalità testuale e quella cinematica. Studi come MotionCLIP [23] sfruttano le capacità zero-shot del modello CLIP (sviluppato originariamente per accoppiare immagini e testo) per proiettare i descrittori del movimento umano nel medesimo spazio latente del linguaggio naturale. Interrogando questo spazio condiviso, è possibile calcolare la similarità del coseno tra un costrutto testuale e un vettore di movimento, estraendo la clip che minimizza la distanza euclidea. L'infrastruttura ingegnerizzata nel presente lavoro di tesi funge da substrato di visualizzazione ottimizzato per simili architetture di *retrieval*.

2.5 Generazione di Sequenze Lunghe e Composizione Temporale

Sebbene i modelli attuali eccellano nella sintesi di azioni isolate della durata di pochi secondi, la concatenazione di molteplici task temporali pone problematiche di coerenza. La pianificazione di scene complesse (es. "cammina verso la sedia, siediti, alzati e saluta") sfugge spesso ai vincoli delle architetture *end-to-end*.

Il framework TEACH (*Temporal Action Compositions for Human Motions*) [24] affronta questa limitazione introducendo un modello compositivo in cui molteplici etichette testuali vengono distribuite su una linea temporale, vincolando l'attenzione della rete neurale a rispettare le pose intermedie. Tuttavia, la gestione di lunghi orizzonti temporali rimane computazionalmente gravosa, spingendo la ricerca verso architetture modulari in cui segmenti estratti o generati singolarmente vengono cuciti insieme in fase di *post-processing*.

2.6 Motion Blending e Retargeting Cinematico

L'orchestrazione sequenziale di clip isolate richiede la gestione matematica delle discontinuità al momento della transizione. La tecnica fondamentale per colmare tali salti (*in-betweening*) si basa sull'interpolazione lineare sferica (Slerp) per i quaternioni, teorizzata da Shoemake [25]. Interpolando le rotazioni dei giunti secondo archi di circonferenza a velocità angolare costante, si previene la deformazione geometrica tipica delle interpolazioni matriciali lineari.

Per animazioni complesse in cui la fisica del baricentro è determinante, interpolazioni matematiche pure risultano inadeguate. Recenti ricerche, come quelle proposte da Harvey et al. [26], implementano architetture di *Robust Motion In-betweening* basate su reti neurali in grado di prevedere le transizioni tenendo conto dei vincoli cinematici (es. posizione dei piedi a terra) e della dinamica del movimento, ricostruendo l'informazione mancante in modo biologicamente plausibile.

2.7 Sistemi Agentici e Co-Creazione Uomo-IA

L'integrazione di *Large Language Models* (LLM) come agenti autonomi ha recentemente rivoluzionato la pianificazione logica e l'interazione con l'ambiente digitale. Studi architetturali come ReAct [4] dimostrano come l'unione del ragionamento linguistico (Reasoning) e della capacità di innescare azioni su strumenti esterni (Acting) consenta agli LLM di superare ostacoli multistep.

Tale principio è stato applicato all'orchestrazione di reti neurali in framework come HuggingGPT [27], in cui un modello di linguaggio non esegue task specifici, ma pianifica quali modelli specialistici richiamare in base alla richiesta dell'utente. Nel contesto della simulazione comportamentale, Park et al. [28] hanno dimostrato l'impiego di agenti generativi capaci di conservare memoria e simulare azioni coerenti su lunghi orizzonti temporali all'interno di ambienti sandbox tridimensionali.

Nell'ambito specifico dell'animazione fisica, il presente lavoro si innesta sulle fondamenta poste da De Filippo e Milano [3], i quali hanno esplorato l'impiego di LLM per la scomposizione e l'orchestrazione di *performance* coreografiche in sistemi robotici discreti. L'architettura software progettata in questa tesi eleva tale concetto: abbandona l'uso di dizionari robotici limitati a favore del modello continuo e ad alta densità parametrica SMPL-H [5]. Svincolando la pipeline di caricamento asincrono [13] dall'elaborazione generativa [11], il sistema fornisce all'LLM un ambiente di *rendering* [6] reattivo e standardizzato, consolidando l'infrastruttura necessaria per una reale e autonoma co-creazione Uomo-IA.

Il funzionamento interno di tali orchestratori si basa su pattern architetturali complessi, tra cui il più noto è il framework ReAct (Reasoning and Acting) [4]. Come illustrato in Figura 2.2, un agente istanziato con tale logica non tenta di produrre un output definitivo in un singolo passaggio. Esso itera ciclicamente attraverso tre fasi: la formulazione di un pensiero analitico (*Thought*), l'esecuzione di una

chiamata a strumenti esterni come API o database (*Action*) e l'analisi dell'output ricevuto dallo strumento (*Observation*). Nel dominio dell'animazione *data-driven*, questa architettura demanda all'IA esclusivamente il carico semantico e logico della pianificazione temporale, demandando a motori esterni ottimizzati la complessa sintesi della fisica tridimensionale, prevenendo così le "allucinazioni" geometriche tipiche dei modelli generativi diretti.

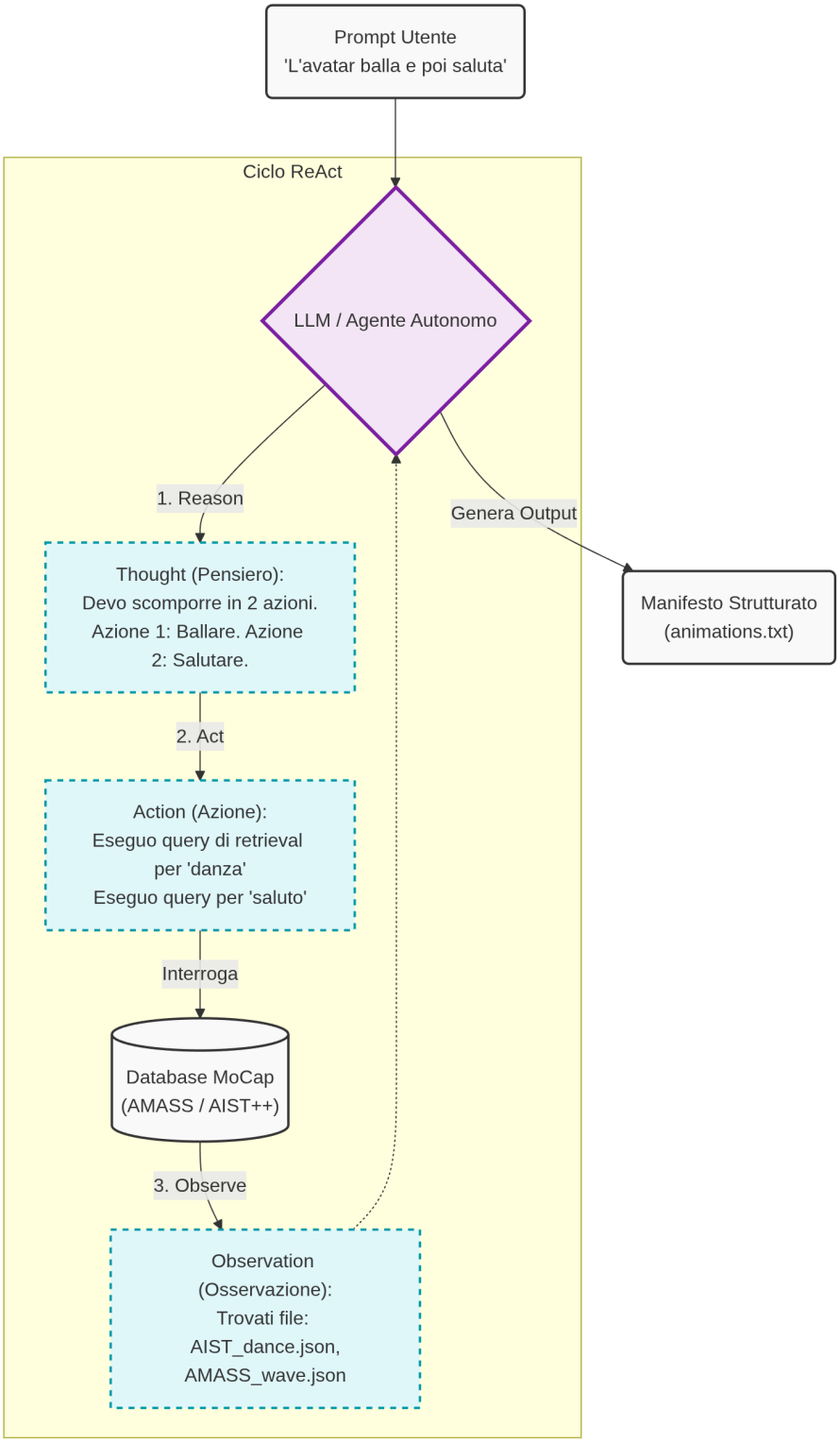

Figura 2.2: Rappresentazione del ciclo operativo ReAct [4] applicato all'orchestrazione di *performance* virtuali. Il pattern evidenzia la delega del carico computazionale cinematico a database esterni (es. AMASS), limitando il compito dell'LLM alla pura pianificazione logica delle azioni.

2.7.1 Co-Creazione e Paradigma Human-in-the-Loop

L'integrazione di agenti logici basati su LLM non mira alla completa automazione della sintesi 3D, bensì alla costituzione di un flusso di co-creazione *Human-in-the-Loop* (HITL). Nel flusso di lavoro tradizionale, un animatore tecnico è vincolato alla manipolazione manuale di *keyframe* e curve di interpolazione, o all'uso di complesse macchine a stati finiti per gestire le transizioni tra animazioni pre-registrate.

Come analizzato da De Filippo e Milano [3] nel contesto delle *performance* robotiche, l'adozione di un orchestratore agentico sposta il ruolo dell'operatore umano da esecutore a supervisore o "regista". L'utente fornisce direttive semantiche ad alto livello, mentre l'agente deduce le sequenze logiche, estrae i parametri da librerie come AMASS [7] e costruisce un piano di esecuzione. Questo piano viene successivamente validato dall'utente e inviato al motore grafico per la simulazione fisica. Tale separazione delle responsabilità (ragionamento semantico demandato all'LLM ed esecuzione geometrica demandata al *player* Unity) conferisce al sistema elevata scalabilità e modularità, permettendo l'aggiornamento indipendente dei singoli moduli software.

2.8 Integrazione in Ambienti Real-Time e Limiti Architetturali

La trasposizione di modelli parametrici complessi come SMPL-H [5] dai framework di ricerca in Python agli ambienti di simulazione interattiva pone notevoli sfide ingegneristiche. I motori grafici commerciali, come Unity [6], sono ottimizzati per l'elaborazione di animazioni pre-compilate tramite sistemi a stati finiti (es. *Mecanim*) e file proprietari strutturati (come il formato `.fbx`).

L'impiego di database MoCap di derivazione accademica introduce un'incompatibilità architetturale di base. I formati nativi come `.npz` (utilizzato da AMASS [7]) o `.pkl` (utilizzato da AIST++ [9]) sono dizionari serializzati ottimizzati per la lettura tramite librerie scientifiche e linguaggi non compilati. L'implementazione di parser diretti per tali file all'interno dell'ambiente di *runtime* C# comporta significativi colli di bottiglia computazionali. La deserializzazione di tensori multidimensionali binari nel thread principale del motore grafico provoca massicce allocazioni di memoria volatile. Tale processo innesca cicli aggressivi del *Garbage Collector*, risultando in fenomeni di *stuttering* visivo e cali drastici del *frame rate*, rendendo impossibile la visualizzazione fluida richiesta dai sistemi di orchestrazione generativa.

Inoltre, l'aggiornamento a *runtime* di una *mesh* a 6890 vertici richiede il ricalcolo continuo delle trasformazioni spaziali per ciascun giunto della gerarchia cinematica. Affinché un sistema di visualizzazione possa supportare la ricezione asincrona di animazioni da un agente esterno, è necessario disaccoppiare la fase di estrazione tensoriale da quella di renderizzazione, standardizzando i dati in formati intermedi leggeri e text-based strutturati gerarchicamente.

Capitolo 3

Ingegnerizzazione della Pipeline Dati e Conversione

Il presente capitolo documenta la progettazione e l'implementazione del modulo software *back-end* sviluppato in Python. L'obiettivo primario di questo modulo è fungere da strato intermedio (ETL - *Extract, Transform, Load*) tra i database MoCap accademici eterogenei e l'ambiente di *runtime* Unity. Vengono analizzati nel dettaglio i formati di input originali, i passaggi matematici per la correzione spaziale e le strategie computazionali adottate per l'elaborazione dei dati su larga scala.

3.1 Obiettivi e Architettura Generale

L'integrazione diretta dei file cinematici all'interno del motore grafico presenta limitazioni derivanti dalle procedure di deserializzazione. Come evidenziato nei capitoli precedenti, formati quali `.npz` e `.pkl` richiedono l'allocazione dinamica di tensori multidimensionali, operazione che in ambiente C# causa allocazioni in memoria e interruzioni del ciclo di rendering.

La soluzione adottata prevede il trasferimento del carico computazionale a monte. Lo script Python elabora i file sorgente, applica le correzioni geometriche necessarie per i modelli SMPL e SMPL-H [5], e produce in output file JSON strutturati. La Figura 3.1 illustra il diagramma di flusso logico della pipeline progettata per il trattamento dei dati provenienti da AMASS, mentre la variante adattata per la topologia di AIST++ è visibile in Figura 3.2.

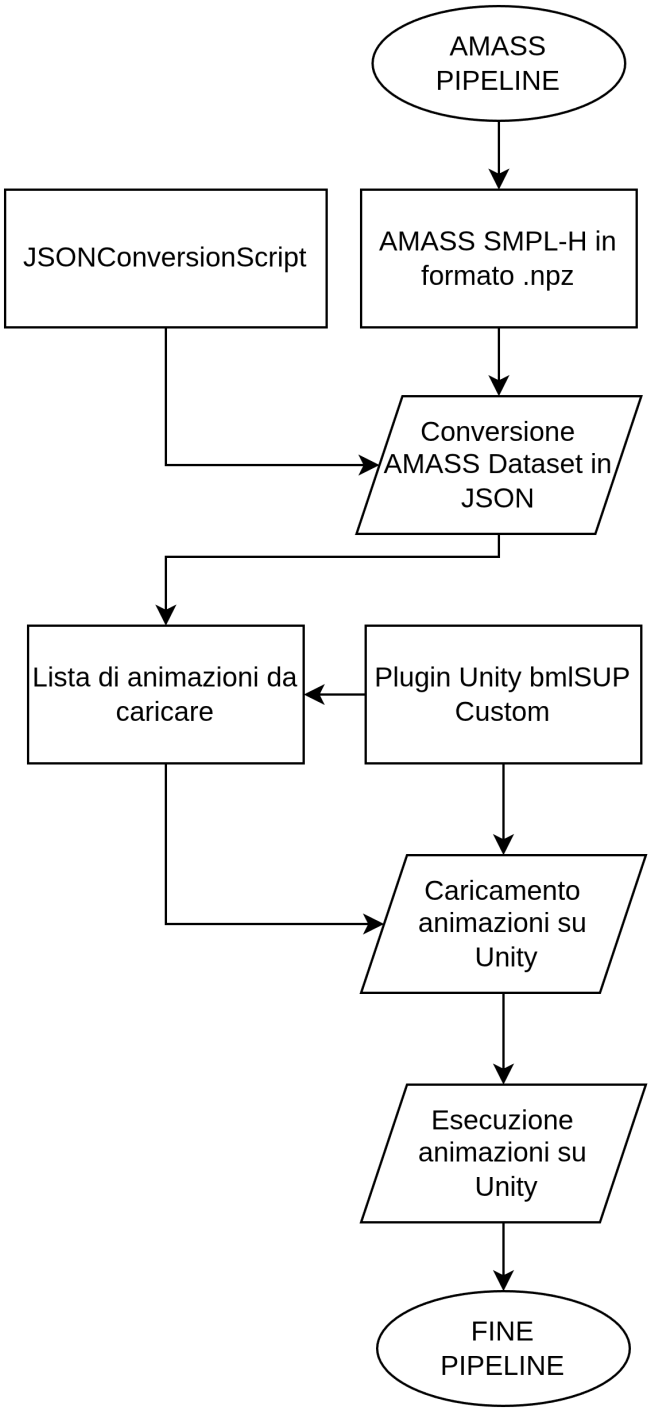

Figura 3.1: Diagramma di flusso della pipeline di conversione per il dataset AMASS. Il processo evidenzia l'estrazione dai file `.npz`, la trasformazione matematica in Python e l'esportazione verso il *plugin bmlSUP Custom* in Unity.

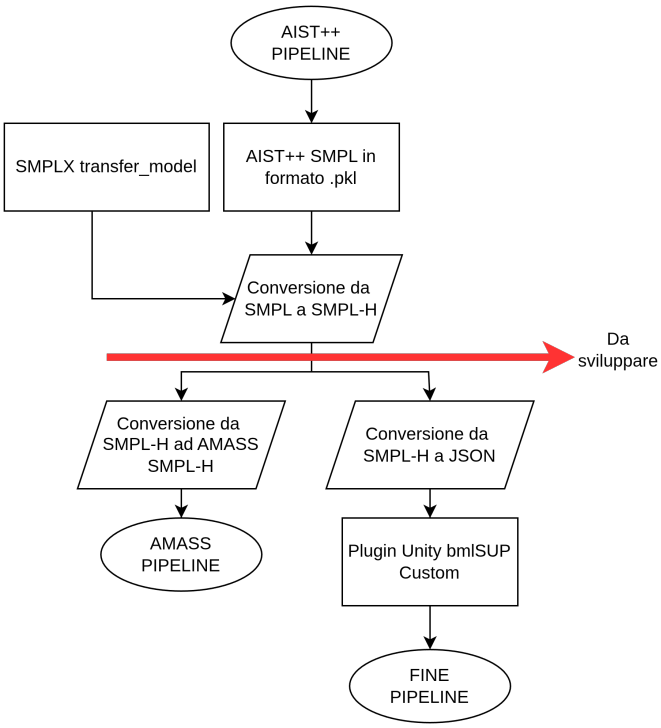

Figura 3.2: Diagramma di flusso della variante della pipeline applicata ad AIST++. In questo caso, il flusso include uno step obbligatorio per il *padding* topologico da SMPL (24 giunti) a SMPL-H (52 giunti).

3.2 Struttura degli Input e Formati Originali

L'implementazione delle routine di conversione richiede una dipendenza diretta dalla topologia dei dati originari. I due benchmark principali elaborati dalla pipeline presentano differenze nel metodo di serializzazione.

3.2.1 Serializzazione NumPy in AMASS (`.npz`)

Il dataset AMASS archivia le sequenze tramite dizionari compressi in formato `.npz`. L'ispezione delle chiavi allocate rileva un allineamento nativo al framework SMPL-H a 52 giunti (Figura 3.3). I parametri di posa (`poses`) sono codificati come un array bidimensionale di dimensione $[F, 156]$, dove F indica i fotogrammi e 156 rappresenta i gradi di libertà spaziali espressi tramite vettori asse-angolo (52×3). I coefficienti biometrici del soggetto (`betas`) sono presenti come vettore a 16 dimensioni, definendo la morfologia del modello *target*.

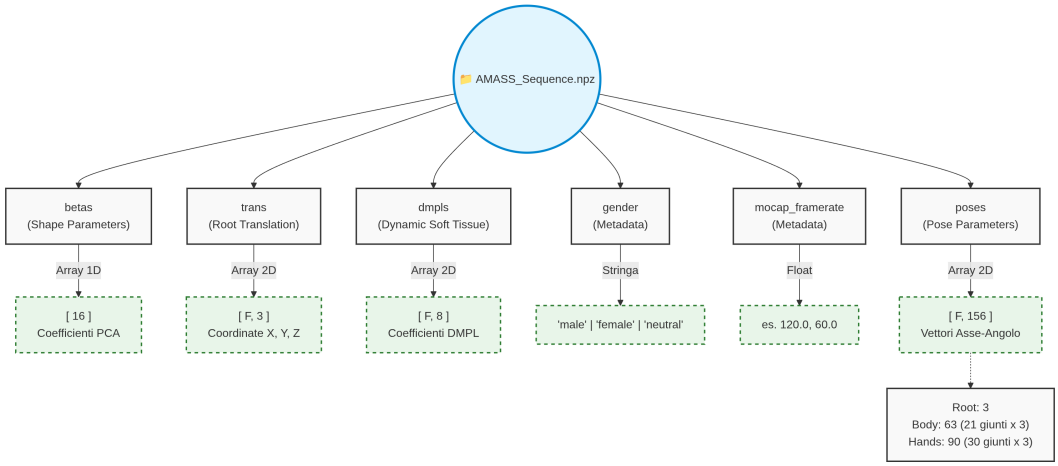

Figura 3.3: Struttura tensoriale di un file `.npz` estratto dal dataset AMASS. Si noti la dimensione dell'array `poses` $[F, 156]$ che copre l'intera topologia di mani e corpo.

3.2.2 Serializzazione Pickle in AIST++ (`.pkl`)

Il dataset coreografico AIST++ impiega la serializzazione binaria proprietaria Python (`.pkl`). L'analisi dei dump di memoria, illustrata in Figura 3.4, evidenzia differenze strutturali rispetto ad AMASS: il tensore tridimensionale `smpl_poses` di dimensione $(N, 24, 3)$ mappa il modello base SMPL, omettendo la catena cinematica delle estremità. Parallelamente, la chiave `smpl_trans` $(N, 3)$ memorizza le posizioni globali esprimendole in centimetri rispetto a un sistema di riferimento destroso, orientato con l'asse Z verticale (*Z-up*).

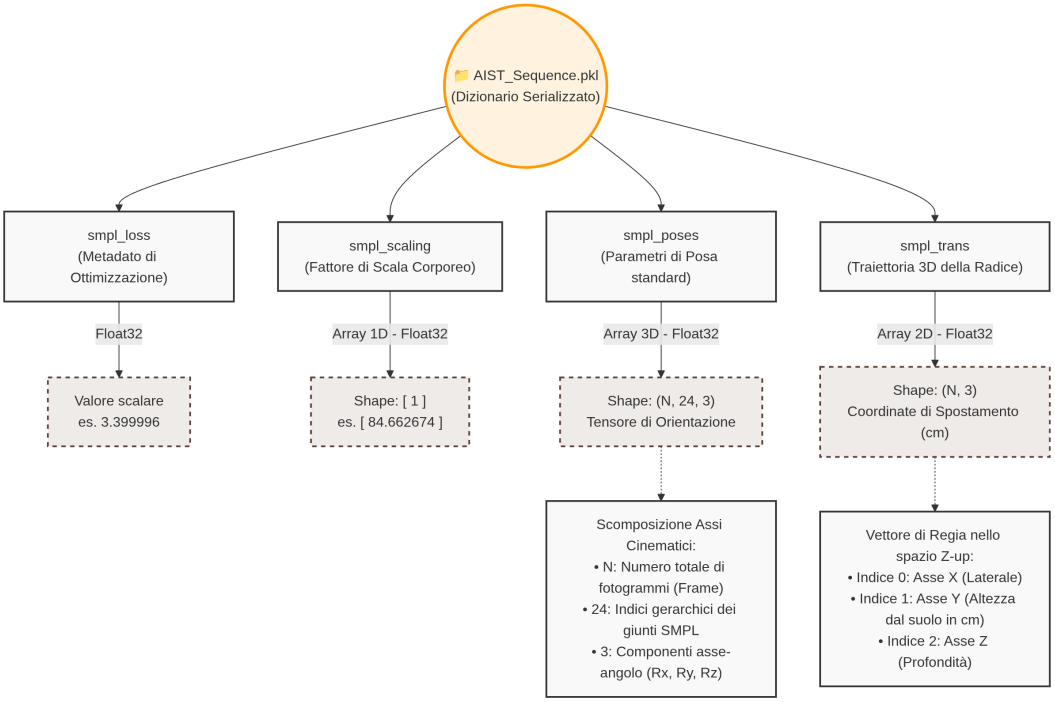

Figura 3.4: Albero strutturale di un file `.pkl` estratto da AIST++. Le dimensioni del tensore `smpl_poses` $(N, 24, 3)$ e l'assenza dei coefficienti $\boldsymbol{\beta}$ evidenziano l'incompatibilità diretta con il *player* basato su SMPL-H.

3.3 Moduli di Conversione e Risoluzione Matematica

La logica dell'applicativo Python opera tramite funzioni di trasformazione vettoriale gestite dalla libreria `scipy.spatial.transform` [11]. Gli scopi di questa fase sono la riduzione del calcolo a *runtime* tramite pre-conversione in quaternioni e l'omogeneizzazione dei sistemi di coordinate.

3.3.1 Conversione da Asse-Angolo a Quaternioni

I motori grafici valutano le rotazioni nello spazio tridimensionale mediante quaternioni, prevenendo singolarità geometriche (*Gimbal Lock*). L'estrazione delle pose dai file sorgente restituisce vettori asse-angolo $\vec{v} = \theta\hat{e}$, dove l'unità vettoriale $\hat{e}$ è l'asse di rotazione e il modulo θ è l'ampiezza dell'angolo in radianti. La pipeline elabora la matrice riconfigurandone la struttura prima della serializzazione:

Listing 3.1: Pseudocodice per la vettorializzazione e conversione in quaternioni delle rotazioni articolari locali, ottimizzata tramite NumPy e SciPy.

```
import numpy as np
from scipy.spatial.transform import Rotation as R

def convert_poses_to_quaternions(poses_array, num_joints):
    # Flattening temporaneo per raggruppare i 3 assi (x, y, z)
```

```
    # L'array [F, num_joints * 3] diventa [F * num_joints, 3]
    reshaped_poses = poses_array.reshape(-1, 3)

    # Inizializzazione dell'oggetto Rotation di SciPy
    # partendo dal formato rotvec (axis-angle)
    rotations = R.from_rotvec(reshaped_poses)

    # Conversione simultanea in quaternioni (x, y, z, w)
    quaternions = rotations.as_quat()

    # Ripristino della dimensionalita' temporale [F, num_joints,
   4]
    return quaternions.reshape(-1, num_joints, 4)
```

3.3.2 Proposta per la Trasformazione dei Sistemi di Coordinate (Z-up → Y-up)

La mancata correzione del sistema di coordinate produrrebbe il ribaltamento orizzontale degli avatar nel motore grafico. AIST++ modella l'ambiente in uno spazio destroso *Z-up*. Per adattarlo allo spazio sinistroso *Y-up* nativo di Unity [6], l'approccio architetturale delineato per i futuri sviluppi della pipeline prevede un intervento correttivo su due livelli geometrici.

Il primo livello interessa il vettore di traslazione globale (`trans`). La quota verticale Z originaria diviene la nuova quota Y, mentre l'asse della profondità Y originario viene negato e proiettato sull'asse Z. I valori grezzi richiedono inoltre una divisione scalare per 100. Tale divisione è motivata sia dalla documentazione originale di AIST++, sia da una verifica empirica sui tensori estratti, i quali confermano che le traslazioni originarie sono espresse in centimetri, rendendo necessaria la conversione per mantenere la compatibilità con il sistema metrico (in metri) standard internazionale di Unity. Un'ipotetica implementazione vettoriale è illustrata di seguito:

Listing 3.2: Pseudocodice di un approccio vettoriale proposto per la correzione spaziale e metrica della traslazione radice in AIST++.

```
def propose_fix_translation(trans_array):
    # trans_array e' un array di dimensione [F, 3] contenente [X,
   Y, Z]
    # L'unita' di misura originale e' in centimetri, Unity opera
   in metri
    fixed_trans = np.zeros_like(trans_array)

    fixed_trans[:, 0] = trans_array[:, 0] / 100.0   # Asse X:
   Laterale
    fixed_trans[:, 1] = trans_array[:, 2] / 100.0   # Asse Y:
   Diventa Z (Altezza)
```

```
    fixed_trans[:, 2] = -trans_array[:, 1] / 100.0  # Asse Z:
Diventa -Y (Profondita')

    return fixed_trans
```

Il secondo passaggio, da implementare in futuro a livello di *front-end*, riguarda l'orientazione della catena cinematica. Sfruttando il calcolo tramite *Forward Kinematics*, si dovrà generare una matrice di rotazione correttiva assoluta pari a $-90°$ sull'asse X, da moltiplicare a sinistra esclusivamente per il quaternione del giunto radice (pelvi, indice 0). L'ereditarietà delle trasformazioni topologiche applicherà conseguentemente la rotazione all'intero modello 3D, allineandolo al vettore visivo del motore grafico.

3.4 Ottimizzazione del Transfer Model e Padding Topologico

Le differenze strutturali tra i modelli SMPL a 24 giunti (AIST++) e SMPL-H a 52 giunti non si limitano a una semplice discrepanza negli array, ma richiedono un adattamento morfologico e topologico. Se l'esportazione verso il formato JSON si fermasse all'indice 23, il parser di destinazione in Unity riporterebbe eccezioni a *runtime* per l'assenza dei dati relativi alle falangi.

Per gestire questa transizione, il processo è stato demandato al framework **SMPL-X** [14]. L'algoritmo `transfer_model` non si limita a copiare banalmente le rotazioni, ma avvia un processo di ottimizzazione e *fitting* matematico. Sfruttando la funzione obiettivo del solutore, l'algoritmo ricalcola i parametri affinché la superficie della *mesh* target (SMPL-H) minimizzi la distanza geometrica rispetto alla *mesh* sorgente (SMPL) generata dalla posa corrente. Solo al termine di questo *transfer* algoritmico della postura corporea, la pipeline procede all'inserimento dei dati periferici mancanti. Nello specifico, viene effettuato il *padding* delle falangi iniettando quaternioni identità (`[0, 0, 0, 1]`) per le mani, e vengono inseriti i 16 parametri morfologici statici ($\boldsymbol{\beta}$) medi, necessari per l'attivazione del *Linear Blend Skinning* in Unity.

3.4.1 Tuning degli Iperparametri e Rilassamento delle Tolleranze

L'applicazione del `transfer_model` sull'intero dataset AIST++ ha evidenziato un limite computazionale. L'algoritmo sfrutta un solutore numerico *Trust Region Newton Conjugate Gradient* (`trust-ncg`), configurato originariamente per applicazioni su pose statiche. Le tolleranze predefinite prevedevano un limite di 100 iterazioni (`maxiters`) e una tolleranza del gradiente (`gtol`) pari a 10^{-6}.

Questo livello di precisione comporta un elevato costo computazionale per l'elaborazione iterativa di fotogrammi sequenziali, dove le variazioni angolari inter-frame sono minime. Per ridurre i tempi di calcolo, è stato condotto un *tuning* degli iperparametri nel file di configurazione, modificando i vincoli matematici:

Listing 3.3: Confronto delle configurazioni YAML per il solutore `trust-ncg` di SMPL-X [14]. A sinistra i parametri di *default*, a destra i parametri modificati per l'elaborazione *batch* in AIST++.

```
# Configurazione Originale SMPL-X
optim:
  type: 'trust-ncg'
  maxiters: 100
  gtol: 1e-06


summary_steps: 100
```

```
# Configurazione Modificata
optim:
  type: 'trust-ncg'
  maxiters: 6
  gtol: 1e-2
  ftol: 1e-3


summary_steps: 1000000
```

Come illustrato nel Codice 3.3, il numero massimo di iterazioni è stato ridotto a 6. La soglia `gtol` è stata impostata a 10^{-2} per limitare l'ottimizzazione sui micro-angoli, ed è stato introdotto il parametro `ftol` a 10^{-3}, configurando l'*early stopping* dell'algoritmo qualora il miglioramento della funzione obiettivo risulti inferiore allo 0.1%. Infine, l'incremento del parametro `summary_steps` a un milione ha diminuito il carico di scritture su disco.

Queste modifiche ai parametri, combinate con lo script di parallelizzazione descritto nel paragrafo successivo, hanno adattato l'algoritmo per l'elaborazione *batch* di sequenze multiple.

3.5 Ottimizzazione Computazionale: Multiprocesso a Memoria Separata

L'elaborazione di vasti database cinematici introduce tempi di latenza derivanti dalle operazioni di *Input/Output* sul disco (lettura file `.npz` e scrittura `.json`) e dal carico della CPU. Un approccio di esecuzione sequenziale comporta tempi di elaborazione prolungati.

Per ottimizzare l'utilizzo dell'hardware, l'architettura del convertitore Python è stata strutturata sfruttando il paradigma del parallelismo multiprocesso a memoria separata. Per superare le limitazioni del modulo `threading` in Python (legate al *Global Interpreter Lock* - GIL), è stata impiegata la libreria `multiprocessing` per generare processi indipendenti (*worker*) che operano in spazi di memoria isolati.

3.5.1 Gestione dei Worker e Allocazione del Carico

Il caricamento del database è gestito tramite un *Pool* di processi *worker*. L'algoritmo effettua una scansione (DFS - *Depth First Search*) della directory di *input*, isolando i percorsi validi. La lista risultante viene suddivisa tra i core logici disponibili.

Per evitare sovrascritture durante la serializzazione asincrona, ciascun *worker* istanzia le matrici di trasformazione spaziale `scipy` all'interno della propria memoria privata, evitando la condivisione di variabili. I file generati vengono scritti mantenendo

un albero delle directory speculare a quello originario. L'analisi quantitativa dei tempi di esecuzione è riportata nel Capitolo 5.

3.6 Gestione delle Eccezioni e Tolleranza ai Guasti

L'elaborazione di dati provenienti da dataset open-source comporta la possibile presenza di file malformati, *metadata* mancanti o differenze topologiche. Per evitare che un errore su un singolo file interrompa l'intero processo di elaborazione *batch*, il flusso di conversione è stato incapsulato in blocchi `try-except`:

- **`KeyError`:** Intercetta l'assenza di array strutturali nei dizionari NumPy o Pickle (es. assenza della chiave `poses`).
- **`ValueError`:** Rileva le discrepanze dimensionali durante le operazioni di *reshaping* matriciale o interpolazione spaziale.
- **`OSError / IOError`:** Gestisce permessi negati o errori di lettura sui file binari.

In presenza di un'eccezione, il file interessato viene escluso, l'evento viene registrato nel log di sistema e il processo prosegue con il file successivo della coda.

3.7 Struttura del Formato JSON

La fase finale dell'architettura ETL consiste nella generazione di un file JSON (*JavaScript Object Notation*), utilizzato come formato di interscambio tra le librerie Python e il parser C# del *front-end* Unity. L'adozione di un tracciato testuale consente la deserializzazione tramite le API del motore grafico (es. `JsonUtility`), riducendo i tempi di *overhead* legati alla lettura dei formati binari nativi.

La sintassi del documento generato, illustrata nella Figura 3.5, è organizzata per rispecchiare le strutture dati richieste dalle interfacce di visualizzazione.

Il dizionario principale contiene le costanti globali dell'animazione (nome, frequenza di campionamento e i 16 coefficienti morfologici β). I dati cinematici sono inseriti nell'array bidimensionale `Frames`. Ogni fotogramma mappa il vettore di traslazione della radice e un array lineare (`poses`) composto da 52 quaternioni. Questo formato sposta il carico computazionale delle trasformazioni matriciali dalla CPU a *runtime* alla fase di pre-elaborazione.

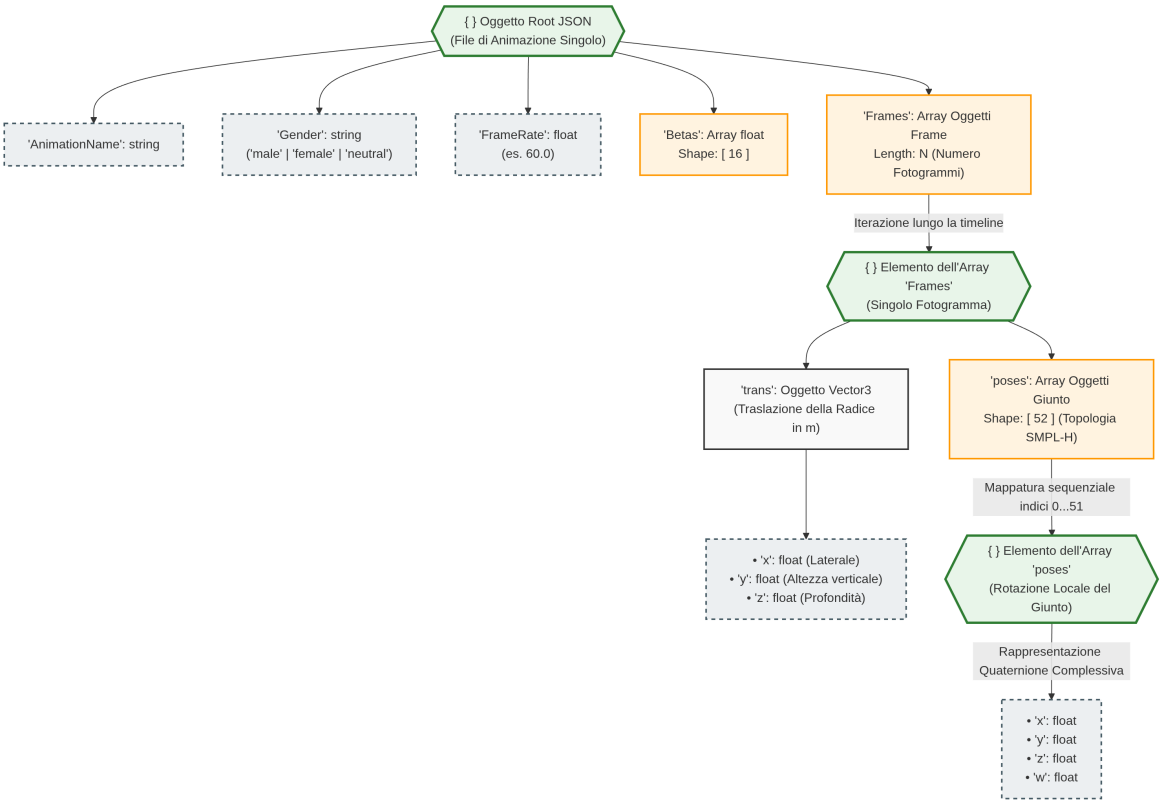

Figura 3.5: Albero strutturale dell'oggetto JSON generato dalla pipeline. Il formato include le primitive per i metadati e le liste annidate contenenti la cinematica.

Capitolo 4

Progettazione e Implementazione dell'Architettura in Unity

Il presente capitolo documenta lo sviluppo del sottosistema di *front-end* all'interno del motore grafico Unity [6]. L'obiettivo di questa fase ingegneristica è la costruzione di un client di visualizzazione modulare e disaccoppiato, in grado di interfacciarsi dinamicamente con i dati generati dalla pipeline Python, predisponendo l'ambiente alla futura orchestrazione agentica.

4.1 Analisi e Limiti del Framework Originale bmlSUP

Il punto di partenza per la renderizzazione tridimensionale è costituito da *bmlSUP* [13], un *player open-source* progettato per Unity. L'analisi del codice sorgente nativo ha evidenziato vincoli architetturali non compatibili con i requisiti di caricamento dinamico richiesti dall'Intelligenza Artificiale generativa.

4.1.1 Vincoli del Thread Principale e Gestione della Memoria

Il primo vincolo risiede nella gestione degli *asset* e delle operazioni di *Input/Output*. Il modulo di caricamento originale è strutturato per elaborare file in formato `.json` o `.h5`. Tuttavia, la lettura dei JSON è subordinata all'impiego della classe `TextAsset` di Unity. Tale approccio richiede che i file vengano importati manualmente nell'Editor e indicizzati staticamente all'interno dell'eseguibile durante la fase di *build*. Questo paradigma strutturale impedisce a un'applicazione in esecuzione (*runtime*) di rilevare, leggere e deserializzare file JSON generati dinamicamente sul disco rigido da un orchestratore esterno. Inoltre, l'assenza di un'architettura multithreading per le operazioni di I/O causa il blocco del *Main Thread* di Unity durante la deserializzazione di tensori complessi. In ambiente C#, l'allocazione massiva di array multidimensionali nel thread principale innesca frequenti chiamate al *Garbage Collector* (GC), le quali si traducono in latenze di caricamento e cadute del *frame rate* (fenomeno di *stuttering*).

4.1.2 Limiti della Logica di Riproduzione Statica

Il secondo vincolo riguarda la logica di riproduzione. Il componente originale implementa un metodo di riproduzione progettato per eseguire *clip* singole o gruppi

di animazioni istanziate simultaneamente all'origine degli assi vettoriali `(0,0,0)`. La riassegnazione dell'array dei personaggi attivi a ogni nuova chiamata interrompe le animazioni in corso, limitando l'accodamento sequenziale. Manca, di base, un gestore degli *offset* spaziali e un controllore del flusso temporale, elementi necessari per disporre azioni discrete su una linea temporale continua senza incorrere nella compenetrazione poligonale delle *mesh*.

4.2 Architettura del Sistema e Responsabilità delle Classi

Per superare le limitazioni riscontrate, l'infrastruttura C# è stata estesa implementando moduli guidati dai dati (*data-driven*). La Figura 4.1 illustra il diagramma UML delle classi sviluppate e le dipendenze con le librerie del framework originale.

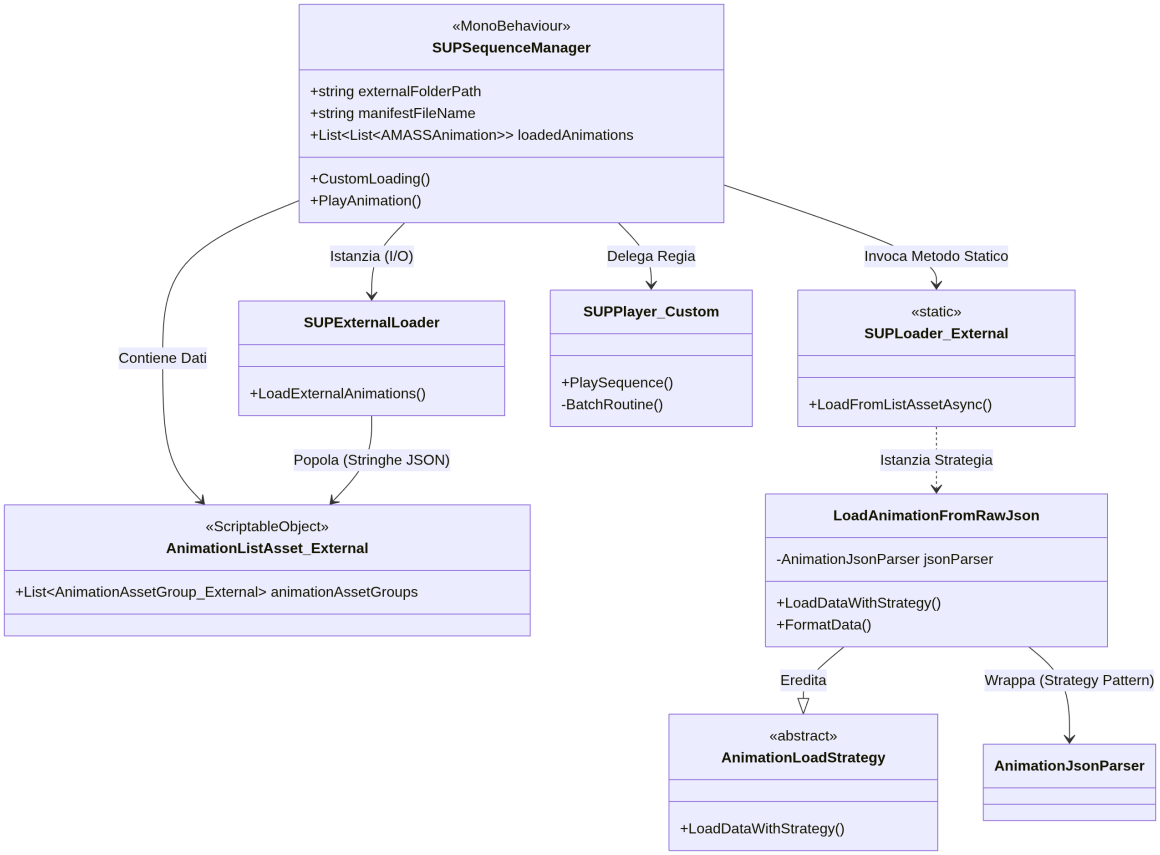

Figura 4.1: Diagramma delle classi (UML) del sottosistema Unity implementato. Il grafo evidenzia il ruolo orchestratore di `SUPSequenceManager`, il modulo di estrazione a *runtime* `SUPExternalLoader` e il pattern *Strategy* adottato per il parser JSON.

4.2.1 Il Pattern Facade: SUPSequenceManager

Il flusso logico è centralizzato dal componente `SUPSequenceManager`. Essendo l'unica classe del sottosistema ad ereditare da `MonoBehaviour`, essa possiede un ciclo di vita vincolato al motore grafico e funge da *entry point* per l'interfaccia utente (Inspector). Agendo secondo il *Design Pattern Facade*, la classe coordina tre responsabilità specifiche:

1. **Inizializzazione dello Stato:** Aggrega i parametri di simulazione definiti dall'utente, inclusi `PlaybackSettings` (configurazioni temporali), `BodySettings` (dati morfologici) e `DisplaySettings` (gestione dei materiali).
2. **Coordinamento dell'I/O:** Istanzia e invoca la classe `SUPExternalLoader` per l'interrogazione del file system ospite.
3. **Gestione della Regia:** Riceve l'array multidimensionale dei dati deserializzati e lo trasmette alla classe `SUPPlayer_Custom` per l'avvio della simulazione visiva.

4.2.2 Architettura Event-Driven per il Controllo del Flusso

Per garantire il disaccoppiamento tra il modulo di caricamento dati e il modulo di visualizzazione, la transizione di stato all'interno del `SUPSequenceManager` è gestita tramite un'architettura *Event-Driven*. L'infrastruttura espone un delegato statico `Action` (`OnLoadingFinished`) e un `UnityEvent` (`OnLoadingFinishedEvent`). L'invocazione di questi eventi avviene esclusivamente al termine di tutte le operazioni asincrone di calcolo, segnalando al motore grafico che i dati sono allocati in memoria e pronti per l'istanza dei *GameObject*.

4.3 Disaccoppiamento del Parser: Lo Strategy Pattern

Il mantenimento della retrocompatibilità con i solutori di cinematica diretta (Forward Kinematics) preesistenti ha richiesto l'implementazione del *Design Pattern Strategy*. L'obiettivo architetturale era permettere l'iniezione di stringhe JSON prelevate direttamente dalla memoria volatile, aggirando il vincolo di dipendenza dalla classe `TextAsset` senza alterare le firme dei metodi del *plugin* nativo.

È stata sviluppata la classe `LoadAnimationFromRawJson`, derivata dalla classe astratta base `AnimationLoadStrategy`. Questa implementazione agisce da *wrapper* per il parser `AnimationJsonParser`:

Listing 4.1: Implementazione del pattern Strategy per il disaccoppiamento della sorgente dati. La classe inietta le stringhe nel parser nativo, svincolando il sistema dalla classe `TextAsset` di Unity.

```
public class LoadAnimationFromRawJson : AnimationLoadStrategy {
    readonly AnimationJsonParser jsonParser;
```

```
    // Costruttore: riceve la stringa cruda (raw json) in memoria
    public LoadAnimationFromRawJson(string jsonString, Models
    models) : base(models) {
        jsonParser = new AnimationJsonParser(jsonString);
    }

    protected override async Task<AnimationData>
    LoadDataWithStrategy() {
        return await jsonParser.LoadDataAsync();
    }

    protected override bool IsMatchingModel(ModelDefinition model)
    {
        return jsonParser.IsMatchingModel(model);
    }

    protected override void FormatData() {
        jsonParser.FormatData();
    }
}
```

Attraverso l'uso del polimorfismo, il sistema di caricamento principale interroga il metodo `LoadDataWithStrategy()` senza necessità di identificare l'origine fisica del dato (disco rigido locale, rete o generazione procedurale).

4.4 Gestione dei Dati Tramite ScriptableObject

Per la memorizzazione temporanea delle stringhe lette dal disco e la loro organizzazione logica, è stata adottata la struttura dati `ScriptableObject` fornita da Unity. È stata implementata la classe `AnimationListAsset_External`, la quale contiene una lista di oggetti di tipo `AnimationAssetGroup_External`. Ciascun gruppo espone una lista di stringhe (`jsonEntries`) atta a contenere il testo crudo dei file JSON estratti.

L'impiego di uno `ScriptableObject` per il transito dei dati offre vantaggi legati alla serializzazione dell'Editor: permette l'ispezione visiva dei *payload* caricati a *runtime* direttamente dall'interfaccia grafica e previene la deallocazione accidentale delle stringhe da parte del *Garbage Collector* durante i passaggi di stato della scena.

4.5 Livello di Configurazione e Parametrizzazione (Data-Driven)

Al fine di garantire la massima flessibilità operativa e supportare un paradigma di sviluppo *Data-Driven*, l'architettura implementata isola i parametri di esecuzione

dalla logica algoritmica. Il sistema preserva e riutilizza le strutture di configurazione originarie del framework [13], le quali sono serializzate nel motore grafico [6] sotto forma di `ScriptableObject` (file con estensione `.asset`).

Questo strato di astrazione consente di alterare il comportamento visivo, morfologico e temporale della simulazione modificando direttamente i file di *settings* dall'Editor, senza necessitare la ricompilazione degli script C#. Tali configurazioni vengono fornite come dipendenze (iniettate tramite l'interfaccia dell'*Inspector*) al `SUPSequenceManager`, il quale le propaga a valle alle istanze della classe `SUP-Player_Custom`. Le configurazioni si suddividono in tre macro-categorie: *Body Settings*, *Display Settings* e *Playback Settings*.

4.5.1 Gestione Morfologica: Body Settings

Il modulo `BodySettings` governa il modo in cui il motore grafico processa i dati biometrici ($\boldsymbol{\beta}$) e l'aggiornamento topologico dei vertici durante l'esecuzione temporale. L'ispezione della struttura di serializzazione testuale nativa di Unity (Codice 4.2) rivela come i parametri booleani permettano un controllo granulare sul calcolo della cinematica.

Listing 4.2: Estratto del file di serializzazione `Custom Body Options.asset`. I flag booleani determinano quali componenti matematici (forma, posa, traslazione) debbano essere calcolati a *runtime*.

```
MonoBehaviour:
  m_Name: Custom Body Options
  ShowIndividualizedBody: 1
  UpdateBodyShapeLive: 1
  UpdatePosesLive: 1
  UpdatePoseBlendshapesLive: 1
  AllowPoseManipulation: 0
  UpdateTranslationLiveY: 1
  UpdateTranslationLiveXZ: 1
  GroundSnap: 0
```

L'abilitazione del parametro `UpdatePosesLive` autorizza il solutore a ricalcolare le matrici di rotazione per ogni singolo *frame*. Parallelamente, i flag `UpdateTranslationLiveY` e `UpdateTranslationLiveXZ` consentono al modello di traslare nello spazio tridimensionale. Per l'implementazione dell'orchestrazione agentica, il vincolo `GroundSnap` viene mantenuto disabilitato (`0`), in quanto forzare la proiezione verticale della *mesh* al suolo interferirebbe con animazioni complesse (es. salti o acrobazie provenienti dal dataset AIST++), le cui quote altimetriche sono già intrinsecamente gestite dalla *pipeline* di estrazione Python.

4.5.2 Visualizzazione e Diagnostica: Display Settings

Il modulo `DisplaySettings` controlla la componente di *rendering* e i *material* applicati alle istanze poligonali generate. La struttura del file serializzato (Codice 4.3) consente la sovrascrittura degli *shader* e l'attivazione di modalità di visualizzazione diagnostiche.

Listing 4.3: Estratto del file `Custom Display Settings.asset`. La configurazione permette di alternare il *rendering* visivo tra una *mesh* poligonale completa e rappresentazioni astratte basate su segmenti e nodi luce (*PointLights*).

```
MonoBehaviour:
  m_Name: Custom Display Settings
  DisplayBones: 1
  DisplayPointLights: 1
  DisplayMeshAs: 0
  MeshDisplaySettings:
    Opaque: {fileID: 2100000, guid: [...]}
    SemiTransparent: {fileID: 2100000, guid: [...]}
  BoneDisplaySettings:
    BoneWidth: 0.01
    DrawSidesDifferentColors: 1
  PointLightDisplaySettings:
    PointLightDisplaySize: 0.02
```

La disabilitazione della *mesh* primaria e la concomitante attivazione di `DisplayBones` e `DisplayPointLights` converte l'avatar in un diagramma cinematico a segmenti. Questa parametrizzazione risulta di fondamentale importanza durante le fasi di collaudo della *Data Pipeline*: permette infatti di verificare la corretta interpolazione dei quaternioni e il *padding* dei giunti periferici (mani) senza subire il calo di prestazioni derivante dall'aggiornamento dei 6890 vertici del modello SMPL.

4.5.3 Regolazione Temporale: Playback Settings

Il comportamento cronologico della sequenza è disciplinato dal file `PlaybackSettings`. Oltre a fornire le costanti matematiche per la separazione spaziale dei batch (`OffSetSpacing`, analizzato nella sezione dedicata all'orchestrazione), questo modulo definisce la resilienza temporale del sistema.

Listing 4.4: Estratto del file `Custom Playback Settings.asset`. Il parametro `FallbackFPS` fornisce un valore cronologico di sicurezza in caso di corruzione dei metadati sorgente.

```
MonoBehaviour:
  m_Name: Custom Playback Settings
  PlayBackwards: 0
  FallbackFPS: 60
```

```
Loop: 0
OffsetMultipleAnimations: 1
OffSetSpacing: {x: 2, y: 0, z: 0}
PlaybackImmediately: 1
```

Nel processo di conversione di dati eterogenei, è possibile riscontrare dizionari sprovvisti della chiave relativa alla frequenza di campionamento (FPS). L'infrastruttura implementata demanda a Unity la risoluzione di tali eccezioni: qualora il parser JSON non rilevi l'informazione di *framerate* nel *payload*, il gestore del tempo adotta automaticamente il valore di `FallbackFPS` (impostato di default a 60 Hz). Inoltre, per assicurare che il motore logico esegua correttamente la chiusura del blocco di iterazione asincrono (*Coroutine*) al termine di un'animazione, il parametro `Loop` deve essere imposto a falso (`0`), demandando esplicitamente la continuità dell'azione al manifesto strutturato fornito dall'agente logico.

4.6 Caricamento Esterno Asincrono: Da File a Memoria

L'operazione di estrazione e decodifica dei tensori a *runtime* è stata strutturata in due fasi asincrone distinte, demandate a *thread pool* di *background* attraverso l'impiego della libreria `System.Threading.Tasks`.

4.6.1 I/O su Disco e Allocazione delle Stringhe

La prima fase è controllata dalla classe `SUPExternalLoader`. Il metodo `LoadExternalAnimations()` legge in ingresso il file di manifesto (`animations.txt`), il quale definisce l'ordine di esecuzione fornendo i nomi dei file `.json` raggruppati per riga. La classe apre un canale di lettura verso la *directory* specificata utilizzando l'API di sistema `File.ReadAllText` e alloca le stringhe lette all'interno dell'istanza `AnimationListAsset_External`. Questo procedimento sostituisce il *database* interno di Unity, abilitando il caricamento di file processati esternamente.

4.6.2 Deserializzazione Asincrona dei Tensori

La seconda fase opera sulla deserializzazione algebrica. Attraverso i costrutti `async` e `await`, il metodo `LoadFromListAssetAsync` della classe `SUPLoader_External` itera sull'elenco delle stringhe precaricate. Per ogni iterazione, viene istanziata la strategia `LoadAnimationFromRawJson` e invocato il parser, il quale converte le stringhe in oggetti complessi di tipo `AMASSAnimation`. Al completamento di tutti i *Task* allocati, l'infrastruttura emette l'evento `Action<List<List<AMASSAnimation>>>` per notificare il gestore centrale. Il confinamento del carico di I/O in task asincroni mantiene il *Main Thread* in stato di disponibilità, riducendo le interruzioni del ciclo di rendering.

4.7 Flusso di Esecuzione e Integrazione nell'Editor

La sequenza delle operazioni è formalizzata nel diagramma di sequenza in Figura 4.2. Il diagramma illustra il ciclo temporale dall'innesco asincrono fino al trasferimento del controllo al *thread* principale di rendering per l'esecuzione visiva.

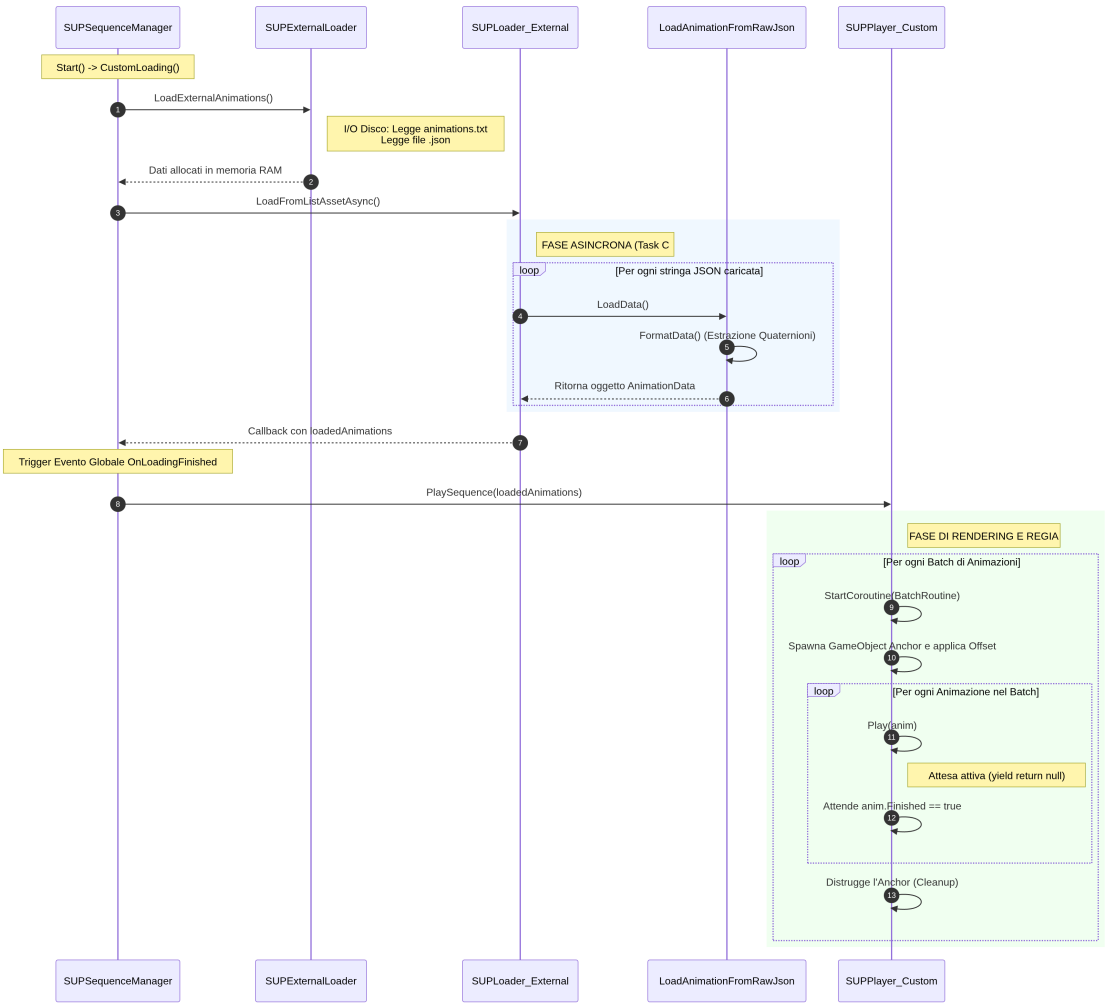

Figura 4.2: Diagramma di Sequenza del sistema. Il flusso evidenzia la transizione dalla fase asincrona C# (Task) all'avvio delle Coroutines nel *Main Thread* di Unity per la gestione della regia temporale.

Il modulo espone i parametri di configurazione nell'interfaccia grafica (*Inspector*) (Figura 4.3). Il `SUPSequenceManager` consente all'operatore di definire il percorso della cartella di origine (`External Folder Path`) e il nome del file di orchestrazione (`Manifest File Name`) prima dell'avvio della simulazione.

L'esecuzione automatica si innesca nel metodo `Start` tramite l'invocazione di `CustomLoading()`. A caricamento concluso, il delegato `OnLoadingFinished` instrada la matrice di animazioni alla classe `SUPPlayer_Custom`, delegandole la responsabilità dell'aggiornamento topologico a schermo.

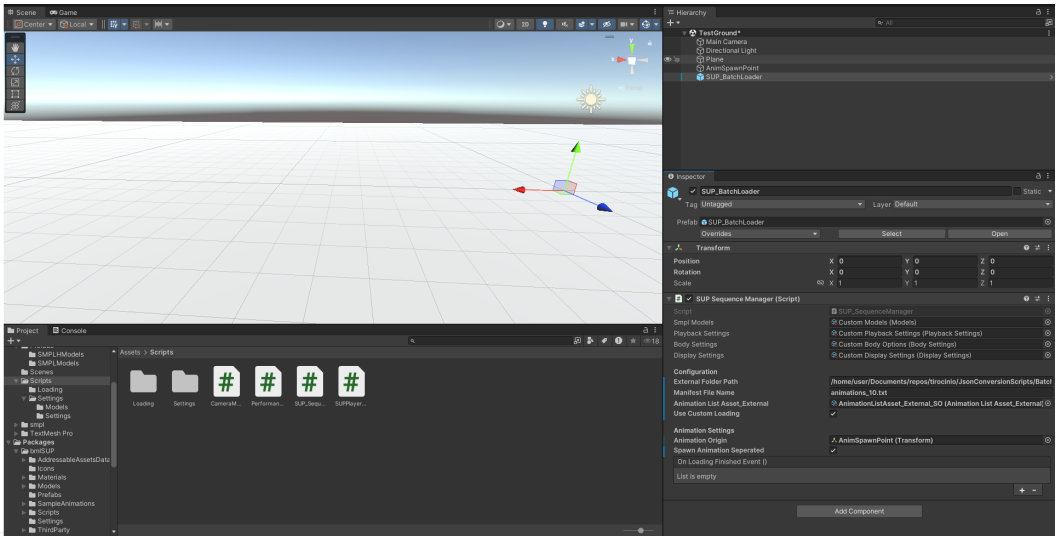

Figura 4.3: Schermata dell'Editor di Unity [6]. La finestra *Inspector* a destra mostra l'interfaccia personalizzata del `SUPSequenceManager`, impiegata per la definizione dei parametri di I/O e riproduzione.

4.8 Il Motore di Regia: SUPPlayer_Custom e Coroutines

La visualizzazione di animazioni multiple derivate dalla frammentazione di una query testuale ha richiesto lo sviluppo del modulo `SUPPlayer_Custom`. Per orchestrare sequenze temporali ordinate senza causare il blocco del motore grafico, la classe utilizza il paradigma della concorrenza cooperativa nativo di Unity, basato sull'impiego delle *Coroutines*.

Il metodo `PlaySequence` riceve la lista dei gruppi di animazione e, per ciascun gruppo (*batch*), istanzia una routine asincrona eseguita sul thread principale tramite la funzione `StartCoroutine(BatchRoutine)`.

Listing 4.5: Implementazione della Coroutine di regia in `SUPPlayer_Custom.cs`. Si osserva la definizione dell'offset spaziale, l'attesa attiva (*yield return null*) e la procedura finale di distruzione dell'oggetto.

```
private IEnumerator BatchRoutine(List<AMASSAnimation> batch, int
    batchIndex) {
     // Generazione dinamica dell'Anchor spaziale per il batch
    corrente
     GameObject anchorObj = new GameObject($"Batch_{batchIndex}
    _Anchor");
     Transform batchAnchor = anchorObj.transform;

     // Calcolo dell'offset per prevenire compenetrazioni visive
     Vector3 startPos = defaultOrigin != null ? defaultOrigin.
    position : Vector3.zero;
     batchAnchor.position = startPos + (defaultPlaybackSettings.
    OffSetSpacing * batchIndex);
```

```
    // Iterazione sulle animazioni appartenenti al batch
    foreach (AMASSAnimation anim in batch) {
        Play(anim, overrideOrigin: batchAnchor);

        // Attesa attiva: sospende l'iteratore finche' l'
    animazione non si conclude
        while (!anim.Finished) {
            yield return null;
        }
    }

    // Deallocazione dell'Anchor al completamento della sequenza
    Object.Destroy(anchorObj);
}
```

4.8.1 Calcolo dell'Offset Spaziale e Sviluppi Futuri della Cinematica

L'istanza di attori multipli nella medesima coordinata assoluta provocherebbe sovrapposizioni poligonali. Allo stato attuale dell'implementazione, per gestire la distribuzione ambientale, la routine `BatchRoutine` si limita a generare un oggetto vuoto (*Anchor*) per ogni gruppo e a disporre i modelli in esecuzione uno accanto all'altro. La traslazione di questo *Anchor* sull'asse orizzontale è determinata moltiplicando l'indice del *batch* in esecuzione per una costante configurabile (`OffSetSpacing`). L'applicazione di tale trasformazione geometrica produce l'effetto visivo documentato nella Figura 4.4.

Tuttavia, per elevare il grado di personalizzazione e controllo dell'orchestrazione agentica, l'architettura è stata strutturata per accogliere due fondamentali sviluppi futuri. In primo luogo, verrà estesa la sintassi del manifesto testuale (`animations.txt`) per consentire la dichiarazione esplicita dei parametri di *spawn* spaziale e rotazione (`Transform`). Questo permetterà all'IA di specificare le coordinate vettoriali di istanza per ciascuna animazione, allocando un parametro dedicato per riga direttamente all'interno della lista.

In secondo luogo, si interverrà sul controllo della continuità spaziale. Attualmente, le animazioni successive vengono fatte partire sempre dal punto di origine del relativo *Anchor*. In una *release* futura, il modulo C# verrà implementato per estrarre il differenziale di posizione e la rotazione globale dell'ultimo fotogramma (*last frame*) dell'animazione precedente. Assegnando matematicamente queste coordinate di fine corsa come nuovo punto di origine per l'istanza dell'azione successiva, sarà possibile ottenere un'esecuzione coreografica altamente sequenziale, concatenata e fisicamente coerente nello spazio tridimensionale.

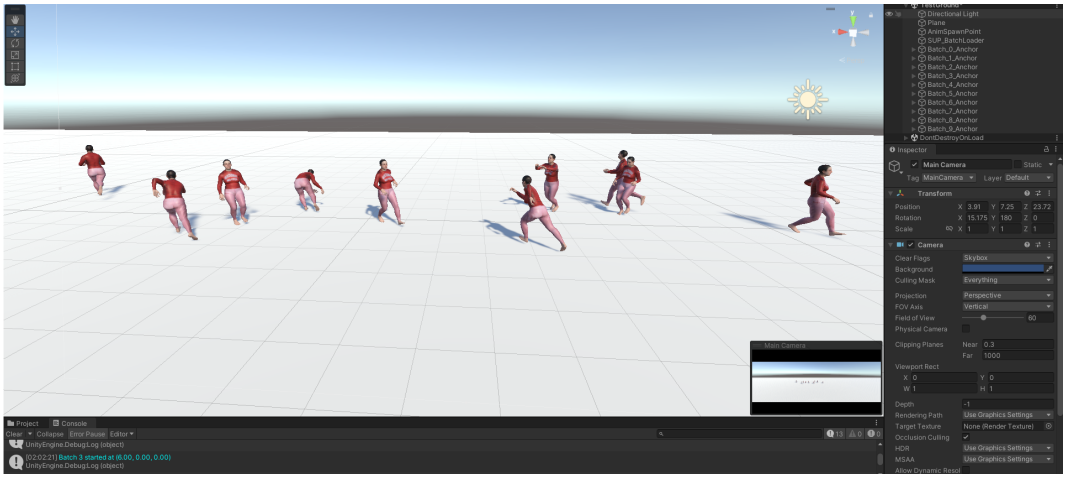

Figura 4.4: Visualizzazione di *batch* multipli in Unity. Si nota l'applicazione della logica di *Offset Spaziale* attuale: ciascun modello è istanziato a una distanza orizzontale costante determinata dal parametro `OffSetSpacing`, evitando la sovrapposizione delle *mesh* poligonali.

4.8.2 Attesa Attiva e Ciclo di Cleanup

L'esecuzione temporale in serie è regolata tramite l'istruzione `yield return null`. Inserita all'interno di un ciclo condizionale `while (!anim.Finished)`, tale istruzione sospende l'avanzamento logico della routine corrente e restituisce il controllo al motore grafico fino al *frame* successivo. Tale meccanismo di attesa attiva viene mantenuto per la durata dell'animazione; alla transizione dello stato `Finished` a `true`, l'iteratore riprende il ciclo elaborando la *clip* seguente.

Al termine dell'iterazione del *batch*, la funzione invoca il metodo di sistema `Object.Destroy(anchorObj)`. Questa operazione esegue il *cleanup* dell'albero gerarchico della scena, liberando i riferimenti e delegando al GC il recupero della memoria, un requisito per garantire la stabilità dell'applicativo in caso di esecuzioni prolungate nel tempo.

Capitolo 5

Risultati Sperimentali e Discussione

Il presente capitolo è dedicato alla valutazione quantitativa e qualitativa dell'architettura software implementata. L'analisi si divide in due sezioni principali: la misurazione delle prestazioni del *back-end* Python (con particolare attenzione allo *scaling* multicore e all'occupazione di memoria fisica) e il collaudo del *front-end* Unity in termini di fluidità di *rendering* e stabilità di allocazione. La sezione conclusiva discute i limiti qualitativi riscontrati nella topologia dei dataset, con particolare riferimento agli artefatti visivi in AIST++.

5.1 Ambiente di Test e Specifiche Hardware

Per garantire la riproducibilità dei risultati, tutti i collaudi prestazionali documentati in questo capitolo sono stati eseguiti sulla medesima postazione di calcolo. Il sistema ospite è basato sul sistema operativo Ubuntu 24.04.4 LTS, equipaggiato con un processore AMD Ryzen 9 7940HS (16 core logici) e 32 GiB di memoria RAM SO-DIMM a 5600 MHz. L'accelerazione grafica è delegata a una GPU NVIDIA GeForce RTX 4070 Laptop (AD106M) con 8 GiB di VRAM dedicata. Gli ambienti virtuali Python (*SMPLX* e *HumanML3D*) sono stati eseguiti su interprete Python 3.13.9, mentre il *client* di visualizzazione è stato compilato su Unity versione 2022.3.62f3.

5.2 Scalabilità della Pipeline Python: Analisi Multicore

Il primo test quantitativo valuta l'efficienza dello script di conversione parallela da `.npz` a `.json` (relativo al dataset AMASS). Per ottenere medie stabili e filtrare l'interferenza dei processi in *background* del sistema operativo, sono state eseguite 3 iterazioni di collaudo per ciascuna configurazione di *worker*.

Le metriche di riferimento per la valutazione dell'architettura parallela sono lo *Speedup* relativo (S_p) e l'Efficienza (E_p), calcolati secondo la Legge di Amdahl:

$$S_p = \frac{T_1}{T_p} \quad ; \quad E_p = \frac{S_p}{p} \times 100$$

dove T_1 è il tempo di esecuzione sequenziale, T_p è il tempo con p core attivi e p è il numero di *worker*.

Numero di Worker (p)	**Tempo Medio (s)**	**Speedup (S_p)**	**Efficienza (E_p)**
1 (Sequenziale)	507.44	1.00x	100.0%
2	255.20	1.99x	99.4%
4	133.07	3.81x	95.3%
8	148.79	3.41x	42.6%
12	157.36	3.22x	26.9%
15	208.25	2.44x	16.2%
16	198.95	2.55x	15.9%

Tabella 5.1: Risultati del benchmark di parallelizzazione per la conversione JSON. Si nota il picco di efficienza massima a $p = 4$, seguito da un degrado prestazionale all'aumentare dei thread.

Come illustrato nella Tabella 5.1, il passaggio da 1 a 4 core logici genera una scalabilità quasi lineare, riducendo il tempo da 507.44s a 133.07s con un'efficienza del 95.3%. Tuttavia, l'allocazione di 8 o più core innesca un regresso prestazionale (il tempo risale a 148.79s per 8 core e fino a 208.25s per 15 core). Questa anomalia documenta la transizione del sistema da un limite computazionale (*CPU-bound*) a un limite di *Input/Output* (*I/O-bound*). Le scritture asincrone simultanee di migliaia di file di testo saturano il controller del disco rigido e causano *overhead* nella gestione della memoria condivisa, dimostrando che il numero ottimale di *worker* per tale operazione sull'hardware di riferimento è 4. Le metriche di questo processo sono state esportate in formato testuale per analisi successive.

5.3 Impatto dello Storage e Struttura Dati

Il disaccoppiamento del *parsing* descritto nel Capitolo 4 comporta un onere quantificabile in termini di archiviazione persistente. L'abbandono dei dizionari binari compressi in favore della notazione JSON testuale genera un notevole incremento delle dimensioni su disco.

L'elaborazione di un lotto di test composto da 2.818 sequenze ha evidenziato che l'archivio originale compresso (`.tar.bz2`) occupava 1.3 GB, decodificati in 2.9 GB di file `.npz`. La serializzazione del medesimo lotto in formato `.json` ha generato un volume allocato di 20.5 GB. Questo incremento dimensionale (7.06× rispetto ai file `.npz` scompattati e 15.76× rispetto all'archivio originario) impone vincoli architetturali per le fasi future. Si stima che l'intero dataset AMASS, il cui peso originario si attesta sui 61.13 GB, richiederebbe circa 432.12 GB di spazio nella variante serializzata in JSON. L'accettazione di tale *trade-off* è giustificata unicamente dallo scarico computazionale in fase di *runtime* garantito alla CPU del motore grafico.

5.3.1 Prestazioni del Motore di Rendering Unity

Le prestazioni del sistema sono state analizzate mediante il Profiler di Unity durante la riproduzione a runtime, variando il numero di animazioni concorrenti. Per ciascuna configurazione sono stati rilevati il frame rate medio, il tempo medio per frame e l'occupazione della memoria di sistema. Il tempo medio per frame non rappresenta una misura indipendente di latenza, ma è stato ricavato dal framerate medio secondo la relazione:

$$t_{\text{frame}} = \frac{1000}{\text{FPS}}, \tag{5.1}$$

Dove t_{frame} è espresso in millisecondi.

Tabella 5.2: Prestazioni osservate durante la riproduzione di un numero crescente di animazioni concorrenti in Unity 2022.3. Il tempo medio per frame è stato calcolato a partire dal framerate medio.

Animazioni	**FPS medio**	**Tempo/frame (ms)**	**Memoria (MB)**
20	59,8	16,75	3664,45
10	54,4	18,42	2536,52
5	67,2	14,90	1994,95
2	69,7	14,38	1649,81

I risultati riportati nella Tabella 5.2 indicano che, nelle condizioni osservate, il sistema ha mantenuto un framerate medio compreso tra 54,4 e 69,7 FPS. L'aumento del numero di animazioni concorrenti determina, come atteso, una crescita dell'occupazione della memoria di sistema, che passa da circa 1,65 GB con due animazioni a circa 3,66 GB con venti animazioni.

Le misurazioni sono state raccolte su finestre di osservazione di durata non perfettamente uniforme. Di conseguenza, i valori devono essere interpretati come un'analisi prestazionale preliminare e non come un confronto sperimentale rigorosamente controllato tra le diverse configurazioni. In particolare, il valore inferiore osservato con dieci animazioni non consente di stabilire una relazione monotona tra numero di avatar e framerate.

Durante le finestre analizzate, il contatore del Garbage Collector riportato dal Profiler non ha mostrato valori superiori alla precisione di visualizzazione adottata. Tale risultato suggerisce una ridotta presenza di allocazioni gestite durante la fase di riproduzione, ma non dimostra la completa assenza di allocazioni o di interventi del Garbage Collector.

5.4 Ottimizzazione del Transfer Model AIST++

L'applicazione del `transfer_model` di SMPL-X per il ricalcolo della topologia di AIST++ da 24 a 52 giunti ha fatto emergere limitazioni dovute all'hardware video. Lo script `batch_transfer.py` è stato limitato a un massimo di 4 *worker*

paralleli. Questa restrizione è imposta dai requisiti di memoria VRAM del modello matematico basato su tensori PyTorch, che esaurisce rapidamente la capacità degli 8 GiB della RTX 4070 a disposizione.

La conversione sperimentale di 23 file `.pkl` ha prodotto un tempo cumulativo di elaborazione pari a 20 892,7 secondi, corrispondente a un tempo medio di 908,37 secondi per file. Tale valore rappresenta la somma dei tempi individuali delle operazioni e non il tempo *wall-clock* del batch parallelo.

Assumendo idealmente una distribuzione uniforme del carico tra quattro worker, il tempo teorico necessario per elaborare le 410 sequenze di AIST++ è stimato come:

$$T_{\text{stimato}} = \frac{908{,}37 \times 410}{4} = 93\,107{,}9 \text{ secondi}, \tag{5.2}$$

equivalenti a circa 25,86 ore. La stima rappresenta un limite teorico ottimistico, poiché non considera l'overhead di coordinamento, le differenze nella durata delle sequenze e i possibili tempi di inattività dei worker.

5.5 Analisi Qualitativa e Limiti Topologici nel Dataset AIST++

Parallelamente all'efficienza computazionale della *pipeline*, l'integrazione del dataset AIST++ ha rivelato criticità qualitative nella ricostruzione cinematica. Sebbene la procedura di *transfer* garantisca la conversione matematica formale da SMPL a SMPL-H, l'output visivo in Unity presenta artefatti in specifiche sequenze, come documentato nella Figura 5.1.

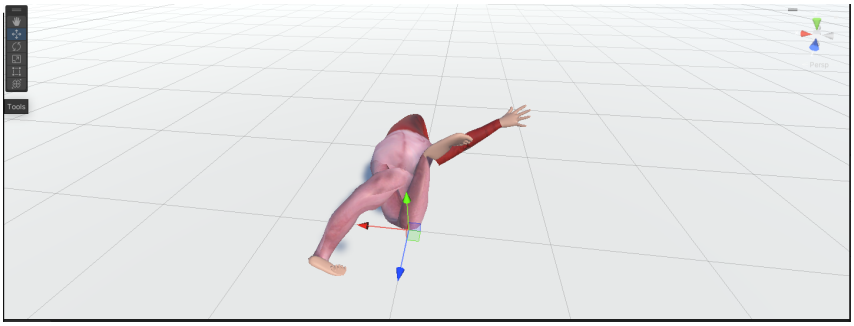

Figura 5.1: Artefatto cinematico riscontrato durante l'integrazione di AIST++. La mesh mostra una torsione innaturale del tronco e un errato allineamento degli arti, sintomo di una discordanza nel sistema di riferimento locale del giunto radice.

5.5.1 Ipotetiche Cause degli Artefatti Geometrici

La deformazione della *mesh* e la torsione degli arti osservate in queste sequenze non sembrano riconducibili a un singolo difetto algoritmico, bensì a una molteplicità

di possibili fattori concorrenti, la cui natura esatta richiederà studi e analisi future. Le potenziali cause di tali artefatti possono essere così elencate:

- **Mancanza dei Parametri Morfologici (β) Originali:** AIST++ omette nativamente i coefficienti β relativi alla corporatura degli attori originali. L'iniezione forzata di parametri neutri o predefiniti durante la fase di *transfer* potrebbe generare un calcolo distorto dei pesi di *Linear Blend Skinning* in prossimità delle articolazioni più complesse (es. pelvi e spalle).
- **Discrepanze di Formato tra SMPL-H e SMPL+H:** Sebbene la terminologia sia spesso usata in modo intercambiabile in letteratura, i dati originari di AMASS impiegano una variante specifica (*SMPL+H*), la quale potrebbe presentare lievi differenze topologiche o convenzioni rotazionali dissimili dal modello SMPL-H standard generato dalla libreria `transfer_model` di SMPL-X. Questa microscopica asimmetria topologica potrebbe accumulare errori algebrici nei quaternioni esportati.
- **Incongruenze nel Sistema di Riferimento:** I *dataset* nascono in spazi tridimensionali discordanti (AIST++ in *Z-up* destroso, Unity in *Y-up* sinistroso). Le trasformazioni correttive applicate alla traslazione radice potrebbero non essere sufficienti a invertire specularmente le matrici di rotazione locali di ciascun giunto, generando l'attorcigliamento visibile in Figura 5.1.
- **Discontinuità Topologica indotta dal Padding:** Il popolamento dei tensori delle falangi tramite quaternioni identità (`[0, 0, 0, 1]`), necessario per far combaciare AIST++ a una gerarchia a 52 giunti, crea blocchi cinematici rigidi alle estremità. La mancata propagazione del moto alle mani potrebbe costringere il solutore *Forward Kinematics* di Unity a compensare le rotazioni sui polsi in modo innaturale.
- **Interpolazione e Foot Sliding:** I dati di traslazione globale estratti dai *file* `.pkl` potrebbero non scalare linearmente nello spazio metrico del motore grafico. Questo genera discrepanze tra il baricentro calcolato e l'effettiva estensione poligonale delle gambe, esitando in collisioni errate e nell'assenza di ancoraggio al piano di calpestio (*foot sliding*).

5.5.2 Prospettive di Risoluzione

Le criticità evidenziate dimostrano che la conversione basata unicamente su *padding* geometrico non è sufficiente a garantire la fedeltà cinematica totale in presenza di formati nativamente discordanti. Il superamento di tali limiti richiederà interventi futuri focalizzati su:

1. L'implementazione di una routine di *Root Motion Correction* in C# che normalizzi il vettore di traslazione globale basandosi sull'effettiva estensione poligonale del modello.

2. L'integrazione di un solutore di *Inverse Kinematics* all'interno di `SUPPlayer_Custom` per ancorare dinamicamente gli arti inferiori al suolo, mitigando il fenomeno del *foot sliding*.

3. Una riconfigurazione delle matrici di rotazione per il *root joint* specifica per il formato AIST++, finalizzata a ricalibrare l'orientamento iniziale rispetto alla *bind-pose* del modello SMPL-H in Unity.

In sintesi, i risultati confermano che la *Data Pipeline* sviluppata assolve efficacemente all'omogeneizzazione del formato e al caricamento asincrono massivo, fornendo un'architettura pronta per il paradigma agentico. La coerenza qualitativa finale rimane tuttavia subordinata all'introduzione di layer di calibrazione geometrica dedicati per ciascuna topologia sorgente.

Capitolo 6

Conclusioni e Sviluppi Futuri

L'evoluzione dei modelli di Intelligenza Artificiale generativa e l'emergere di architetture agentiche hanno aperto nuove prospettive nel campo della sintesi procedurale dell'animazione tridimensionale. Tuttavia, la transizione dalla teoria dei modelli multimodali alla realizzazione di *performance* virtuali interattive è storicamente ostacolata da una profonda frammentazione dei dati e da limiti intrinseci ai motori grafici tradizionali.

Il presente lavoro di tesi si è posto l'obiettivo di colmare questa lacuna architetturale, progettando e implementando un'infrastruttura *software* end-to-end capace di agire come strato intermedio tra i vasti database accademici di *Motion Capture* e l'ambiente di simulazione *real-time* Unity. L'analisi condotta e le soluzioni ingegnerizzate hanno dimostrato la fattibilità di un disaccoppiamento tra la logica di orchestrazione semantica e il motore di *rendering* geometrico.

6.1 Sintesi dei Risultati

Il superamento delle criticità di integrazione ha richiesto un intervento su due fronti computazionali distinti, esitando nello sviluppo di una *pipeline* ETL in Python e in una riscrittura profonda del *client* di visualizzazione in C#.

Standardizzazione e Scalabilità nel Back-End Dal lato dell'elaborazione dati (*back-end*), il lavoro documenta la riuscita standardizzazione di formati eterogenei e complessi quali AMASS e AIST++. La costruzione della Data Pipeline ha automatizzato la conversione delle rappresentazioni rotazionali da asse-angolo a quaternioni e una prima normalizzazione delle traslazioni globali. La pipeline ha quindi implementato la conversione preliminare delle traslazioni, mentre la trasformazione completa delle rotazioni articolari e la gestione della diversa convenzione dei sistemi di coordinate rimangono obiettivi per i futuri sviluppi. Inoltre, per far fronte all'ingente mole di dati richiesta per l'addestramento e l'uso di modelli generativi, il convertitore è stato ingegnerizzato sfruttando il paradigma del *Multiprocessing* a memoria condivisa. Come evidenziato dai *benchmark*, questa ottimizzazione ha permesso di scalare l'elaborazione su architetture *multicore*, abbattendo i tempi di esecuzione e gestendo autonomamente la *fault tolerance* rispetto a *file* sorgenti

malformati. Il risultato netto di questa fase è stato l'estrazione di un formato JSON standardizzato, leggibile e strutturato per l'ingestione asincrona.

Disaccoppiamento e Asincronia nel Front-End Dal lato della simulazione (*front-end*), l'architettura originale del *plugin* di renderizzazione è stata profondamente riprogettata per conformarsi ai requisiti di fluidità richiesti da un orchestratore *AI-driven*. L'implementazione del *Design Pattern Strategy* ha permesso di svincolare il motore grafico dalla lettura statica dei *database* locali. Il passaggio cruciale è stato l'introduzione di un modello *Event-Driven* supportato da *thread pool* di *background* (`Task` asincroni) per la decodifica dei tensori JSON. I collaudi prestazionali hanno validato empiricamente questa scelta: il confinamento del carico di allocazione ha soppresso le interruzioni legate al *Garbage Collector* (costantemente a 0 MB), garantendo al motore Unity un *framerate* stabile attorno ai 60 FPS anche in presenza di decine di animazioni processate in memoria. Infine, l'uso delle *Coroutines* ha dotato il sistema di un motore di regia temporale e spaziale autonomo, capace di distribuire attori virtuali su coordinate flessibili evitando compenetrazioni e collisioni *mesh*.

Abilitazione del Paradigma Agentico Oltre alle metriche di prestazione, il contributo primario di questa architettura risiede nella sua predisposizione al paradigma *Human-in-the-Loop* e ai sistemi agentici. Convertendo la cinematica in manifesti testuali componibili, l'infrastruttura fornisce a un potenziale modello linguistico (LLM) un'interfaccia di comunicazione strutturata. Benché l'integrazione attiva di un Agente LLM esuli dagli scopi implementativi di questa fase del progetto e costituisca uno sviluppo futuro, il *client* Unity è ora formalmente predisposto per agire come *Consumer* passivo, capace di tradurre un lotto di stringhe generate algoritmicamente in movimento fisico.

6.2 Limiti e sviluppi futuri

Sebbene i risultati ottenuti confermino la validità dell'architettura proposta e la sua capacità di gestire in modo scalabile l'elaborazione e la riproduzione di sequenze di movimento, il sistema presenta alcuni limiti che dovranno essere affrontati per consentirne l'impiego in scenari distribuiti, interattivi e maggiormente complessi.

Un primo limite riguarda il formato utilizzato per la rappresentazione e il trasferimento dei dati. La serializzazione in JSON garantisce leggibilità, interoperabilità e facilità di integrazione tra la pipeline Python e il client Unity, ma comporta un incremento significativo dello spazio occupato rispetto ai file sorgente compressi, stimato sperimentalmente nell'ordine di circa 15×. Nell'implementazione attuale, tale aspetto non costituisce una criticità sostanziale, poiché le sequenze vengono caricate dal disco locale. Tuttavia, in una configurazione distribuita, nella quale il modulo di orchestrazione o generazione risieda su un'infrastruttura remota, la

dimensione dei *payload* testuali potrebbe incidere sulla latenza e sull'occupazione della banda di rete. Un possibile sviluppo consiste pertanto nell'introduzione di formati di serializzazione binaria, quali *Protocol Buffers* o *MessagePack*, in grado di ridurre la quantità di dati trasmessi mantenendo una rappresentazione strutturata e indipendente dall'implementazione. I dati potrebbero quindi essere trasferiti mediante protocolli TCP o UDP e deserializzati direttamente in memoria dal modulo C#.

Un secondo elemento riguarda la normalizzazione cinematica delle animazioni provenienti da sorgenti eterogenee. La pipeline realizzata implementa la conversione preliminare delle traslazioni globali e delle rappresentazioni rotazionali, ma la completa gestione delle differenti convenzioni dei sistemi di coordinate e degli orientamenti articolari richiede ulteriori interventi. In particolare, l'analisi del dataset AIST++ ha evidenziato come la sola trasformazione delle traslazioni non sia sufficiente a garantire il corretto allineamento spaziale delle animazioni all'interno di Unity. Sarà quindi necessario introdurre procedure più complete di conversione dei riferimenti, verifica della gerarchia scheletrica e adattamento delle rotazioni locali.

Ulteriori miglioramenti riguardano la qualità e la continuità della riproduzione. L'integrazione di tecniche di *motion blending* consentirebbe di attenuare le discontinuità tra sequenze consecutive, evitando transizioni visivamente brusche. Analogamente, l'impiego di un solutore di *Inverse Kinematics* permetterebbe di correggere artefatti quali il *foot sliding*, migliorando il contatto dei piedi con il terreno e la coerenza complessiva del movimento. Queste estensioni risultano particolarmente rilevanti per applicazioni nelle quali più animazioni vengono concatenate dinamicamente o adattate a differenti caratteristiche degli avatar e dell'ambiente virtuale.

Un ulteriore limite dell'architettura corrente è rappresentato dalla necessità di definire l'intero insieme delle animazioni prima dell'avvio della scena. Il client Unity opera infatti su un manifesto testuale precompilato e non consente, allo stato attuale, di introdurre nuovi comandi o modificare la sequenza durante l'esecuzione. Tale modalità risulta adeguata per la validazione della pipeline e per la riproduzione di sequenze predeterminate, ma limita l'adozione del sistema in contesti interattivi e agentici.

Per superare questa limitazione, il componente `SUPExternalLoader` potrebbe essere esteso mediante un meccanismo di *streaming* procedurale basato su canali di comunicazione asincroni. L'utilizzo di *socket* o di servizi di messaggistica permetterebbe al client di ricevere, validare e decodificare progressivamente nuovi pacchetti durante l'esecuzione. In questo modo, un agente esterno potrebbe aggiornare dinamicamente la selezione delle animazioni, i parametri di riproduzione, la temporizzazione delle sequenze e la disposizione spaziale degli avatar, senza richiedere il riavvio della scena.

Questa estensione rappresenterebbe il passaggio dall'attuale architettura orientata alla riproduzione di sequenze predefinite a un sistema interattivo nel quale la regia virtuale possa essere modificata in tempo reale. L'infrastruttura potrebbe così supportare scenari *Human-in-the-Loop*, nei quali l'utente interviene mediante

comandi testuali, e scenari completamente agentici, nei quali un modello esterno seleziona e coordina autonomamente le animazioni sulla base del contesto, dello stato dell'ambiente o degli obiettivi della simulazione.

Nel complesso, gli sviluppi individuati non richiedono una revisione dei principi architetturali introdotti nel presente lavoro, ma ne costituiscono una naturale estensione. Il disaccoppiamento tra preparazione dei dati, orchestrazione semantica e visualizzazione *real-time* fornisce infatti una base modulare sulla quale integrare formati di comunicazione più efficienti, tecniche avanzate di correzione cinematica e modalità di controllo dinamico. Il lavoro svolto rappresenta pertanto un'infrastruttura preliminare ma funzionale per la futura realizzazione di sistemi di animazione tridimensionale adattivi, distribuiti e controllati da agenti intelligenti.

Ringraziamenti

Desidero ringraziare innanzitutto il mio relatore, il Dott. Lorenzo Stacchio, e il mio correlatore, il Prof. Adriano Mancini ed il Prof. Claudi Sirocchi, per la preziosa guida, la costante disponibilità e il supporto tecnico fornitomi durante l'intero periodo del tirocinio e della stesura di questo elaborato. I loro consigli e direzioni sono stati fondamentali per affrontare le sfide di questo progetto.

Inoltre voglio ringraziare tutti quelli che, nel bene o nel male, mi sono stati utili alla mia crescita e al mio sviluppo personale, quindi la mia famiglia, i miei amici, i miei colleghi di lavoro e universitari e tutto il resto degli abitanti di questo pianeta.

Dichiarazione sull'utilizzo di strumenti di Intelligenza Artificiale
Nel rispetto delle linee guida sull'integrità accademica, si dichiara che durante la stesura del presente elaborato è stato impiegato il modello di Intelligenza Artificiale generativa Gemini (Google). L'utilizzo di tale strumento è stato rigorosamente limitato alla revisione stilistica, al perfezionamento sintattico e al miglioramento della fluidità del testo in lingua italiana. L'ideazione della struttura, la progettazione del codice, l'analisi critica dei risultati, l'elaborazione dei contenuti e l'intera responsabilità intellettuale del lavoro rimangono esclusivamente e integralmente a carico dell'autore.

Jesi, Giugno 2026

Luca Felcher

Bibliografia

[1] G. Tevet, S. Raab, B. Gordon, Y. Shafir, D. Cohen-Or, and A. H. Bermano, "Human motion diffusion model," in *The Eleventh International Conference on Learning Representations*, 2023. [Online]. Available: https://openreview.net/forum?id=SJ1kSyO2jwu

[2] J. Zhang, Y. Zhang, X. Cun, Y. Zhang, H. Zhao, H. Lu, X. Shen, and Y. Shan, "Generating human motion from textual descriptions with discrete representations," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2023, pp. 14 730–14 740.

[3] A. De Filippo and M. Milano, "Large language models for human–ai co-creation of robotic dance performances," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, K. Larson, Ed. International Joint Conferences on Artificial Intelligence Organization, Aug. 2024, pp. 7627–7635.

[4] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.

[5] J. Romero, D. Tzionas, and M. J. Black, "Embodied hands: Modeling and capturing hands and bodies together," *ACM Transactions on Graphics*, vol. 36, no. 6, pp. 245:1–245:17, 2017.

[6] Unity Technologies, "Unity game engine," https://unity.com/, 2022, game engine and development platform.

[7] N. Mahmood, N. Ghorbani, N. F. Troje, G. Pons-Moll, and M. J. Black, "AMASS: Archive of motion capture as surface shapes," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, Oct. 2019, pp. 5442–5451.

[8] C. Guo, S. Zou, X. Zuo, S. Wang, W. Ji, X. Li, and L. Cheng, "Generating diverse and natural 3D human motions from text," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Jun. 2022, pp. 5152–5161.

[9] R. Li, S. Yang, D. A. Ross, and A. Kanazawa, "AI choreographer: Music conditioned 3D dance generation with AIST++," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, Oct. 2021, p. 13381–13392.

[10] S. Tsuchida, S. Fukayama, M. Hamasaki, and M. Goto, "Aist dance video database: Multi-genre, multi-dancer, and multi-camera database for dance information processing," in *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019*, Delft, Netherlands, Nov. 2019, pp. 501–510.

[11] P. Virtanen, R. Gommers, and T. E. e. a. Oliphant, "SciPy 1.0: Fundamental algorithms for scientific computing in python," *Nature Methods*, vol. 17, no. 3, pp. 261–272, 2020.

[12] M. Loper, N. Mahmood, J. Romero, G. Pons-Moll, and M. J. Black, "SMPL: A skinned multi-person linear model," *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, vol. 34, no. 6, pp. 248:1–248:16, Oct. 2015.

[13] A. O. Bebko, A. Thaler, and N. F. Troje, "bmlsup – a smpl unity player," in *2021 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, 2021, pp. 573–574.

[14] G. Pavlakos, V. Choutas, N. Ghorbani, T. Bolkart, A. A. A. Osman, D. Tzionas, and M. J. Black, "Expressive body capture: 3d hands, face, and body from a single image," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[15] T. Li, T. Bolkart, M. J. Black, H. Li, and J. Romero, "Learning a model of facial shape and expression from 4d scans," vol. 36, no. 6, Nov. 2017. [Online]. Available: https://doi.org/10.1145/3130800.3130813

[16] A. A. A. Osman, T. Bolkart, and M. J. Black, *STAR: Sparse Trained Articulated Human Body Regressor*. Cham: Springer International Publishing, 2020, pp. 598–613.

[17] M. Plappert, C. Mandery, and T. Asfour, "The kit motion-language dataset," *Big Data*, vol. 4, no. 4, p. 236–252, Dec. 2016. [Online]. Available: http://dx.doi.org/10.1089/big.2016.0028

[18] A. R. Punnakkal, A. Chandrasekaran, N. Athanasiou, A. Quiros-Ramirez, and M. J. Black, "Babel: Bodies, action and behavior with english labels," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2021, pp. 722–731.

[19] C. Guo, X. Zuo, S. Wang, S. Zou, Q. Sun, A. Deng, M. Gong, and L. Cheng, "Action2motion: Conditioned generation of 3d human motions," in *Proceedings of the 28th ACM International Conference on Multimedia*, 2020, pp. 2021–2029.

[20] X. Chen, B. Jiang, W. Liu, Z. Huang, B. Fu, T. Chen, and G. Yu, "Executing your commands via motion diffusion in latent space," pp. 18 000–18 010, June 2023.

[21] J. Ho, A. Jain, and P. Abbeel, "Denoising diffusion probabilistic models," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 6840–6851. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/4c5bcfec8584af0d967f1ab10179ca4b-Paper.pdf

[22] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever, "Learning transferable visual models from natural language supervision," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 8748–8763. [Online]. Available: https://proceedings.mlr.press/v139/radford21a.html

[23] G. Tevet, B. Gordon, A. Hertz, A. H. Bermano, and D. Cohen-Or, "Motionclip: Exposing human motion generation to clip space," in *Computer Vision – ECCV 2022*, S. Avidan, G. Brostow, M. Cissé, G. M. Farinella, and T. Hassner, Eds. Cham: Springer Nature Switzerland, 2022, pp. 358–374.

[24] N. Athanasiou, M. Petrovich, M. J. Black, and G. Varol, "Teach: Temporal action composition for 3d humans," in *2022 International Conference on 3D Vision (3DV)*, 2022, pp. 414–423.

[25] K. Shoemake, "Animating rotation with quaternion curves," *SIGGRAPH Comput. Graph.*, vol. 19, no. 3, p. 245–254, Jul. 1985. [Online]. Available: https://doi.org/10.1145/325165.325242

[26] F. G. Harvey, M. Yurick, D. Nowrouzezahrai, and C. Pal, "Robust motion in-betweening," *ACM Trans. Graph.*, vol. 39, no. 4, Aug. 2020. [Online]. Available: https://doi.org/10.1145/3386569.3392480

[27] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face," in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36. Curran Associates, Inc., 2023, pp. 38 154–38 180. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/77c33e6a367922d003ff102ffb92b658-Paper-Conference.pdf

[28] J. S. Park, J. O'Brien, C. J. Cai, M. R. Morris, P. Liang, and M. S. Bernstein, "Generative agents: Interactive simulacra of human behavior," New York, NY, USA, 2023. [Online]. Available: https://doi.org/10.1145/3586183.3606763