

Politecnico di Torino

Corso di Laurea in Ingegneria del Cinema e dei Mezzi di Comunicazione
A.a. 2025/2026
Sessione di laurea Settembre 2026

Ottimizzazione e digitalizzazione della pipeline di produzione stop motion

Storyboarding procedurale in Blender e ambienti virtuali in Unreal Engine

Relatore:
Prof. Niccolò Gioia

Candidata:
Giorgia Di Mella

Sommario

La realizzazione di un contenuto stop motion richiede una pipeline produttiva profondamente artigianale, in cui set e pupazzi vengono manipolati fisicamente fotogramma per fotogramma: una rigidità intrinseca che rende costosa, in tempo e risorse, ogni modifica introdotta in corso di lavorazione. Con l'affermarsi della Virtual Production e del rendering in tempo reale, l'integrazione di strumenti digitali all'interno di processi tradizionalmente analogici sta assumendo un ruolo sempre più centrale, offrendo la possibilità di verificare in anticipo scelte di regia e ingombri di scena senza intervenire fisicamente sul set, pur mantenendo intatta la natura tattile del risultato finale. La presente tesi analizza e sperimenta una digitalizzazione parziale della pipeline stop motion, attraverso l'automazione procedurale della preproduzione in Blender e l'integrazione di un ambiente di Virtual Production in tempo reale in Unreal Engine 5.7, con l'obiettivo di individuare, da un lato, un flusso di lavoro unificato basato su strumenti open source per la fase di preproduzione capace di ridurre i tempi di lavorazione, e dall'altro un ambiente virtuale in tempo reale in grado di supportare concretamente la fase di ripresa sul set, senza compromettere la qualità artigianale tipica della tecnica.

Tabella contenuti

Elenco delle figure III

Elenco delle tabelle V

1 Introduzione 1
- 1.1 Il problema . . . 1
- 1.2 Obiettivi della tesi . . . 3
- 1.3 Metodo seguito . . . 3
- 1.4 Caso di studio . . . 4
- 1.5 Struttura della tesi . . . 5

2 Stato dell'arte 6
- 2.1 Il medium stop-motion e DragonFrame . . . 6
  - 2.1.1 La preproduzione nella pipeline tradizionale . . . 7
- 2.2 Tecnologie CG e pipeline: Blender e Unreal Engine . . . 8
- 2.3 Virtual Production at Model Scale, Single-DCC ed ecosistema software 9

3 Automazione della pre-produzione e preparazione degli asset in Blender 12
- 3.1 Architettura generale della pipeline di pre-produzione . . . 12
- 3.2 Ambiente Blender: workspace Storyboarding, Video Sequence Editor e Grease Pencil . . . 15
- 3.3 Il database di produzione: struttura e validazione della Shotlist . . . 18
- 3.4 Parsing nativo del formato .xlsx . . . 19
- 3.5 Generazione procedurale di Scene, Camera e Sequencer . . . 19
- 3.6 Library Override e instancing degli asset . . . 21
- 3.7 Setup ottico e layering 2D/3D con Grease Pencil . . . 24
- 3.8 Control rig parametrico e driver . . . 25
- 3.9 Preparazione degli asset 3D e esportazione con l'addon Blender For UnrealEngine (BFU) . . . 29
- 3.10 Export automatizzato dell'animatic e burn-in dei metadati . . . 30

3.11 Validazione dello script e casi di test 32

4 Virtual Production e Set Extension in Unreal Engine 5.7 35
4.1 Configurazione del progetto Unreal Engine 5.7 35
4.2 Ingegnerizzazione dello shader feltro 36
4.3 Sviluppo dell'ambiente virtuale 38
4.4 Architettura di comunicazione OSC tra DragonFrame e Unreal Engine 40
4.5 Ricezione ed elaborazione dei messaggi OSC nel Level Blueprint `Main` 43
4.5.1 Setup iniziale e debug 45
4.5.2 Smistamento degli indirizzi: `Switch on String` 46
4.5.3 Sincronizzazione del fotogramma 47
4.5.4 Cambio camera per shot 48
4.5.5 Pass di chroma key a doppia esposizione 49
4.5.6 Tracciamento degli assi motion-control 51
4.6 Validazione del sistema e sfide di sincronizzazione 52
4.7 Verifica spaziale ed ergonomica della Set Extension 58
4.7.1 Sperimentazione ergonomica e spaziale 58
4.7.2 Accessibilità dell'animatore 63

5 Analisi dei risultati e valutazione della pipeline 64
5.1 Metodologia di valutazione 64
5.2 Risultati funzionali verificati 65
5.3 Carico manuale eliminato dall'automazione 66
5.4 Valutazione della Set Extension ed ergonomia del set 67
5.5 Confronto tra pipeline tradizionale e digitalizzata 68
5.6 Criticità incontrate e limiti del lavoro 69

6 Conclusioni e sviluppi futuri 70
6.1 Sintesi del lavoro svolto 70
6.2 Contributi della tesi 71
6.3 Sviluppi futuri 72

Bibliografia 74

Elenco delle figure

3.1 Architettura della pipeline di pre-produzione: a sinistra le fasi eseguite dall'operatore, a destra gli artefatti che ciascuna di esse produce. Le istanze di `Location` e `Target` sono collegate per linking alla libreria asset condivisa (§3.6). 14
3.2 Il workspace *Storyboarding* di Blender: si hanno vista 3D, VSE e Dope Sheet tutte nella stessa schermata. 15
3.3 Il workspace *Video Editing* di Blender: gli shot dell'episodio sono accodati in sequenza nel Video Sequence Editor della scena `MASTER_RENDER`, come strip di tipo `SCENE`. 16
3.4 Schizzo di storyboard su Grease Pencil in vista camera. 25
3.5 Driver di apertura e chiusura delle palpebre: il pannello di configurazione e l'effetto sulla Shape Key corrispondente. 27
3.6 Driver di apertura della bocca del personaggio Calzino Puzzone, collegato alla Shape Key `OpenMouth`. 28
3.7 Render dell'animatic su Grease Pencil — Shot0080, ep106, Scena 1. 33
3.8 Render dell'animatic con pupazzi 3D — Shot0080, ep106, Scena 1. . 33

4.1 Il set "Area Case" in Unreal Engine 5.7. 36
4.2 Grafo del materiale `M_NewFeltro`. 37
4.3 Materiale `M_SimpleSkyDome`: gestione desaturazione/tinta per il colore emissivo. 39
4.4 Materiale `M_SimpleSkyDome`: catena dei tre nodi `RotateAboutAxis` con `Add`+`Normalize` tra un asse e il successivo. 40
4.5 Ciclo operativo della sincronizzazione tra DragonFrame e Unreal Engine con dettaglio degli output generati ad ogni ciclo: a sinistra gli eventi di acquisizione, al centro lo stato della scena virtuale, a destra i file e le catture generate (`Plate` EXR da Unreal Engine e scatti fisici da DragonFrame). 44
4.6 Setup iniziale e catena di debug nel Level Blueprint `Main`. 45
4.7 Nodo `Switch on String` con tutti gli indirizzi gestiti. 46

4.8 Ramo `/dragonframe/frame`: sincronizzazione del fotogramma con la Level Sequence. 47
4.9 Ramo `/dragonframe/shotname`: cambio camera per shot. 48
4.10 Ramo `/dragonframe/exposurename`: pass di chroma key. 49
4.11 Vista operatore in DragonFrame: le due file di fotogrammi X1 (sfondo normale) e X2 (chroma key), catturate in sequenza per ogni scatto. 50
4.12 Rami `/dragonframe/axis/*`: tracciamento assi motion-control. 51
4.13 Confronto prima/dopo la calibrazione del colore Emissive di `M_Chroma`: in (a) il verde alla massima intensità appare slavato per la sovraesposizione del sensore; in (b) il valore ridotto a 0,6 restituisce un verde pieno e correttamente esposto. 55
4.14 Plate Capture: allineamento ottico (FOV e Post Process). 57
4.15 Plate Capture: composizione del percorso e cattura del render. . . . 58
4.16 Allestimento fisico del test spaziale. 59
4.17 Confronto prima/dopo la sfocatura ottica: in (a) sono visibili la griglia dei pixel e le linee di giunzione dello schermo; in (b) la stessa inquadratura con la profondità di campo della fotocamera reale a eliminare l'artefatto. 62

Elenco delle tabelle

3.1 Struttura dei campi della shotlist. 18
3.2 Shotlist compilata per ep106, Scena 1, prima della sincronizzazione. Le focali sono espresse in millimetri. Sono omesse, per ragioni di spazio, la colonna `Tipologia` — che esprime in forma estesa la stessa informazione codificata dal campo `Camera` — la colonna `Location`, che per questa scena vale `AreaCase` su tutti gli shot, e la colonna `Note`, che raccoglie le indicazioni di regia e le battute. 32

5.1 Operazioni manuali sostituite dalla sincronizzazione automatica, per un singolo shot con una location e un personaggio. 66

Capitolo 1

Introduzione

Il capitolo individua il problema produttivo da cui il lavoro prende le mosse, dichiara gli obiettivi che si propone e il metodo seguito per raggiungerli, presenta il caso di studio su cui la sperimentazione è stata condotta e descrive l'organizzazione del documento.

1.1 Il problema

La spinta a digitalizzare porzioni della pipeline stop-motion nasce da una pressione economica strutturale del settore, non da una tendenza tecnologica contingente. I costi di produzione aumentano mentre i compensi corrisposti dai committenti restano sostanzialmente stabili: un disallineamento che pesa in modo particolare sulla stop-motion, tecnica particolarmente onerosa in termini di tempo. Il caso più autorevole a sostegno di questa interpretazione è quello di Aardman Animations. Lo studio britannico, pur restando fedele a un'impostazione produttiva artigianale costruita in oltre cinquant'anni di attività, ha avviato un'unità interna di ricerca e sviluppo dedicata proprio a esplorare come le tecnologie digitali possano essere integrate senza snaturare la pratica materiale che ne definisce l'identità [1, 2]. Le stime di mercato disponibili, di fonte commerciale e fortemente discordanti tra loro sul valore assoluto del comparto, concordano d'altra parte nell'indicarlo in crescita [3, 4, 5]: una domanda in espansione, a fronte di una tecnica che resta lenta e costosa per fotogramma, rende la ricerca di margini di efficienza un'esigenza concreta e non teorica.

Questa pressione non si distribuisce in modo uniforme lungo la pipeline, ma si concentra in due punti che presentano caratteristiche diverse e richiedono interventi distinti.

Il primo è la **pre-produzione**. Il percorso che porta dalla sceneggiatura all'animatic è consolidato e in larga parte invariato rispetto alla pratica dell'animazione

disegnata, e proprio per questo eredita un limite che nella stop-motion pesa più che altrove: lo storyboard è bidimensionale, mentre ciò che descrive dovrà essere costruito in scala. Verifiche come l'effettivo ingombro di un elemento di scenografia alla focale prevista, o la compatibilità tra la posizione immaginata per la camera e lo spazio realmente disponibile attorno al set, restano possibili — la scenografia viene dimensionata, le misure vengono prese — ma avvengono in un passaggio separato e successivo, incrociando il disegno con dati che stanno altrove. Disporre della stessa informazione *mentre* si compone l'inquadratura significa anticipare la verifica al momento in cui la decisione viene presa, quando correggerla costa ancora soltanto ricomporre lo shot.

Lo strumento che consentirebbe questa anticipazione esiste: comporre lo storyboard direttamente in un ambiente tridimensionale, dove gli ingombri sono quelli reali. Ma adottarlo ha un costo proprio, ed è qui che interviene la seconda inefficienza. Allestire una scena tridimensionale per ciascuno shot richiede di riportare a mano nello strumento i parametri già presenti nella shotlist — creare la scena, predisporre la camera e assegnarle la focale indicata, istanziare i personaggi e gli ambienti previsti, collocare il risultato nella sequenza dell'episodio — con una sequenza di operazioni che non presenta alcuna variazione concettuale da uno shot al successivo e che si ripete identica decine di volte per episodio. A scala seriale questo costo di allestimento annulla il vantaggio che lo storyboard tridimensionale porterebbe.

Il secondo è la **costruzione del set fisico**. In stop-motion ogni elemento visibile in campo deve esistere materialmente, e il costo cresce con l'estensione dell'ambiente da rappresentare. A differenza di una produzione con molte unità di ripresa in parallelo, dove il problema è la moltiplicazione dei set, in una produzione a unità singola il vincolo si sposta: non è il numero dei set a pesare, ma la necessità che l'unico set disponibile restituisca in immagine un ambiente più ampio di quanto sia stato effettivamente costruito.

Entrambi i punti hanno una caratteristica che li rende aggredibili senza compromettere la natura artigianale della tecnica: riguardano ciò che precede e ciò che circonda l'animazione, non l'animazione stessa. Il problema affrontato da questa tesi è quindi il seguente: *fino a che punto è possibile ridurre il costo della preproduzione e dell'allestimento scenografico di una serie stop-motion introducendo strumenti digitali che lascino inalterata la ripresa fotogramma per fotogramma del pupazzo reale?*

Che la domanda sia attuale lo conferma, per contrasto, il percorso inverso: produzioni interamente digitali investono tempo e risorse per *imitare* l'estetica stop-motion. È il caso della serie *Max & the Midknights* di Nickelodeon, la cui pipeline real-time in Unreal Engine — costruita per ottenere una resa "fatta a mano" pur senza impiegare un solo fotogramma di animazione fisica — è stata presentata in una Production Session a SIGGRAPH 2025 [6]. L'industria CG si

avvicina esteticamente alla stop-motion nello stesso momento in cui la stop-motion si avvicina operativamente alla CG: il punto di incontro tra le due tecniche costituisce un'area di interesse condivisa, e non una questione circoscritta al presente lavoro.

1.2 Obiettivi della tesi

Agli aspetti del problema individuati sopra corrispondono i due obiettivi di questa tesi, complementari e affrontati con strumenti diversi.

Il primo è **rendere praticabile lo storyboard tridimensionale in una produzione seriale**, eliminandone il costo di allestimento. Lo strumento sviluppato (Capitolo 3) genera in modo procedurale, a partire dalla sola shotlist compilata dalla regia, la struttura di scene, camere e asset su cui disegnare lo storyboard, e ne produce l'animatic completo di metadati. La trascrizione manuale dei parametri viene così sostituita da un'unica compilazione del documento di regia, e la verifica spaziale diventa disponibile nel momento stesso in cui l'inquadratura viene composta.

Il secondo è **estendere digitalmente il set fisico**, impiegando un motore di rendering in tempo reale per sostituire con uno sfondo virtuale la porzione di ambiente che non viene costruita materialmente (Capitolo 4). Le ragioni tecniche che hanno guidato la scelta di Blender e di Unreal Engine come piattaforme sono discusse nel Capitolo 2.

A questi due obiettivi se ne affianca uno di natura più specificamente sperimentale. La letteratura più recente sulla digitalizzazione della stop-motion, discussa nel Capitolo 2, segnala come possibile sviluppo il collegamento diretto tra il software di acquisizione e un motore di rendering in tempo reale, senza tuttavia documentarne una realizzazione tecnica. Realizzare tale collegamento costituisce il contributo sperimentale principale di questa tesi. Esso richiede più di un semplice sincronismo di fotogramma (*frame-lock*): ad ogni fotogramma acquisito sul set fisico la scena virtuale deve avanzare in sincronia e, se la camera fisica viene spostata dall'operatore, la camera virtuale deve seguirne la posizione.

1.3 Metodo seguito

Il lavoro è stato condotto come sviluppo iterativo di strumenti software, ma i due fronti sperimentali si trovano in condizioni diverse, che è opportuno dichiarare fin d'ora perché ne determinano il diverso grado di validazione.

Gli strumenti di pre-produzione sono stati sviluppati e messi a punto all'interno della lavorazione effettiva della serie: la shotlist impiegata, i documenti di regia e gli asset sono quelli reali di un episodio in preparazione, e gli script sono stati

corretti via via a fronte dei casi concreti incontrati. La verifica ha quindi la forma di un impiego in condizioni operative, non di una prova su dati costruiti *ad hoc*.

La sperimentazione di Virtual Production si colloca invece in una fase successiva e di natura esplorativa. La serie non è stata portata in produzione, e l'allestimento disponibile per i test era di conseguenza parziale: il pupazzo non era dotato di rig di animazione e parte della scenografia fisica non era stata realizzata. L'idea stessa di estendere digitalmente il set è maturata in parte come conseguenza del primo fronte, una volta constatato che gli asset modellati per la verifica spaziale in pre-produzione erano riutilizzabili a tale scopo. Quanto segue va quindi letto come verifica di fattibilità tecnica condotta su un allestimento rappresentativo ma incompleto, non come collaudo in condizioni di produzione.

Una scelta metodologica specifica ha riguardato l'integrazione tra i due software commerciali coinvolti, non progettati per comunicare tra loro. Il comportamento dell'interfaccia — formato dei dati trasmessi, temporizzazione degli eventi, corrispondenza tra i parametri dei due sistemi — non è stato assunto dalla documentazione, ma verificato empiricamente tramite registrazione su file e strumenti di test costruiti appositamente, indipendenti da entrambi i software. In più di un caso questa verifica ha smentito assunzioni inizialmente plausibili, e le soluzioni adottate ne sono derivate direttamente.

La valutazione conclusiva distingue di conseguenza due piani: la verifica funzionale dei meccanismi realizzati, condotta per via sperimentale, e la stima del vantaggio prodotto rispetto alla pratica tradizionale, per la quale non è stato applicato un protocollo di misurazione comparativa (Capitolo 5).

1.4 Caso di studio

L'intera sperimentazione è stata condotta assumendo come caso di studio la produzione di *Slippets*, una serie animata stop-motion rivolta a un pubblico prescolare (3–6 anni), scelta per verificare la pipeline proposta nel contesto di una produzione seriale reale piuttosto che di un esercizio isolato. La serie segue le relazioni tra coppie di piccole creature — le *slippets* del titolo, che si muovono e agiscono sempre in coppia — in opposizione alla figura solitaria del personaggio Calzino Puzzone, con un registro narrativo dichiaratamente tenero e surreale.

Gli esempi tecnici riportati nei capitoli seguenti — shotlist, personaggi, convenzioni di naming, specifiche cromatiche — fanno riferimento diretto al materiale di produzione reale di questa serie, in particolare alla Storyboard Bible, che fissa la nomenclatura e le convenzioni di regia, e allo *Slippets Asset Sheet Characters*, che documenta modelli e palette cromatiche dei personaggi e degli elementi di set.

1.5 Struttura della tesi

Il Capitolo 2 approfondisce lo stato dell'arte, posizionando il contributo di questa tesi rispetto alla letteratura e ai precedenti industriali più rilevanti, con particolare attenzione al progetto di ricerca condotto da Aardman Animations sulla Virtual Production applicata alla scala del modello.

Il Capitolo 3 descrive il primo fronte sperimentale: la pipeline di automazione della pre-produzione sviluppata in Blender, dalla lettura della shotlist alla generazione dell'animatic con burn-in dei metadati.

Il Capitolo 4 descrive il secondo: la realizzazione della Set Extension in Unreal Engine e il collegamento in tempo reale con DragonFrame, che ne costituisce il nucleo tecnico.

Il Capitolo 5 presenta la valutazione della pipeline proposta, distinguendo i risultati funzionali verificati dalle considerazioni di efficienza, e dichiara i limiti della valutazione condotta.

Il Capitolo 6 raccoglie le conclusioni, esplicita i contributi del lavoro e delinea le direzioni di sviluppo che restano aperte.

Capitolo 2

Stato dell'arte

Il capitolo inquadra il lavoro rispetto al contesto tecnico e alla letteratura di riferimento. Le prime due sezioni descrivono il medium stop-motion e gli strumenti software su cui la pipeline si appoggia; la terza esamina il progetto di ricerca che costituisce il precedente più diretto di questa tesi, individuando lo spazio in cui il lavoro si colloca.

2.1 Il medium stop-motion e DragonFrame

La stop-motion è una tecnica di animazione che genera l'illusione del movimento attraverso l'acquisizione sequenziale di singoli fotogrammi, tra i quali un operatore modifica manualmente la posizione di pupazzi, oggetti di scena o elementi dell'ambiente. A differenza dell'animazione digitale, in cui il movimento è generato interpolando valori numerici tra fotogrammi chiave, in stop-motion ogni fotogramma è il risultato di un intervento fisico irreversibile sul set.

La fase di ripresa, condotta un fotogramma alla volta, è quella in cui la tecnica si concentra e per la quale il software DragonFrame (nella versione 2026.07 impiegata in questa tesi) si è affermato negli ultimi due decenni come standard industriale. In un'unica interfaccia esso consolida il controllo diretto della fotocamera collegata e dei suoi parametri di esposizione, la sovrapposizione (*onion skinning*) tra il fotogramma corrente e quelli precedenti, la gestione di più esposizioni per fotogramma tramite il foglio di esposizione e la registrazione sincronizzata di note di scena. A queste funzioni si aggiunge un sistema di *Action Script*, che consente l'esecuzione automatica di codice esterno a ogni evento di scatto o spostamento [7]. La sua adozione diffusa rende DragonFrame un punto di riferimento imprescindibile per qualsiasi integrazione tra set fisico e strumenti digitali esterni, compreso il lavoro di sincronizzazione con un motore real-time affrontato nel Capitolo 4, che si appoggia precisamente al meccanismo di Action Script.

Al di là della scheda tecnica, la letteratura più recente inquadra DragonFrame non semplicemente come uno strumento di acquisizione, ma come un caso di *softwarizzazione* della pratica artigianale dello stop-frame: Livingstone [1] osserva come il software, pur assistendo il processo tattile dell'animazione a mano, costruisca una gestione dati capillare dell'intero flusso — dalla singola camera fino all'organizzazione dei file di produzione — al punto da rappresentare, nelle produzioni di scala industriale come quella di Aardman Animations, l'interfaccia primaria attraverso cui la produzione stop-frame viene effettivamente realizzata [1]. Questa centralità di DragonFrame nel flusso di lavoro reale è anche il motivo per cui, in questa tesi, il collegamento con Unreal Engine è stato realizzato passando per il suo meccanismo nativo di Action Script, anziché tentare un'integrazione che bypassasse il software di acquisizione.

Se una serie stop-motion condivide con il singolo cortometraggio le fasi fondamentali della lavorazione — sviluppo della storia, storyboard e animatic, costruzione di pupazzi e armature, progettazione e costruzione dei set, ripresa fotogramma per fotogramma e post-produzione — la letteratura di settore individua nella produzione seriale un regime produttivo distinto, con proprie prassi organizzative dettate dalla necessità di rispettare una cadenza di consegna periodica [8]. È in questo regime che la tensione economica richiamata in §1.1 si manifesta con maggiore durezza, poiché il ritmo di consegna non è negoziabile quanto in un progetto a uscita singola [1].

2.1.1 La preproduzione nella pipeline tradizionale

Tra le fasi di lavorazione della produzione seriale elencate sopra, quella su cui interviene la prima metà del lavoro presentato in questa tesi è la preproduzione: è utile ripercorrerne il flusso tradizionale per comprendere dove si collochino i margini di intervento.

Il percorso muove dalla sceneggiatura dell'episodio, sottoposta a *spoglio*: una lettura analitica che ne estrae, scena per scena, tutti gli elementi che la lavorazione dovrà predisporre — i personaggi, principali e secondari, le location, i dialoghi, gli effetti sonori e gli eventuali effetti visivi richiesti. Sulla base dello spoglio vengono prese le decisioni di regia, che confluiscono anzitutto nella *shotlist*: un documento tabellare che scompone ciascuna scena nelle sue inquadrature e ne fissa, shot per shot, i parametri operativi — identificativo, tipo di inquadratura, focale, movimento di camera, personaggi e ambienti coinvolti — corredati da note di regia e dalle battute di riferimento.

La shotlist costituisce la base su cui viene poi costruito lo *storyboard*: una sequenza di disegni, uno o più per shot, che traduce in immagine i parametri stabiliti definendo inquadratura e composizione, e che viene annotata con le indicazioni necessarie alla lavorazione — movimenti della camera e dei personaggi, focale, effetti

sonori ed effetti visivi previsti. Lo storyboard viene infine montato in un *animatic*: una versione temporizzata della sequenza con presenti animazioni basilari, in cui ciascun disegno resta a schermo per la durata effettiva dello shot corrispondente, accompagnato dagli effetti sonori, così da restituire il ritmo reale dell'episodio prima che la ripresa abbia inizio. È questo insieme di documenti a costituire il riferimento su cui si baseranno la costruzione dei set, la programmazione delle riprese e il lavoro degli animatori.

Questo flusso, consolidato e in larga parte invariato rispetto alla pratica tradizionale dell'animazione disegnata, presenta però due caratteristiche che nel contesto seriale diventano onerose. La prima riguarda la natura bidimensionale dello storyboard: un disegno fissa l'inquadratura, ma non consente di verificare le relazioni spaziali effettive tra camera, personaggi e scenografia. Domande come se un elemento di set rientri realmente in campo alla focale prevista, se due personaggi possano occupare contemporaneamente la posizione immaginata, o se la camera possa fisicamente essere collocata dove lo storyboard la presuppone, restano naturalmente verificabili — la scenografia viene dimensionata e le misure vengono rilevate — ma in un passaggio distinto dallo storyboard, che richiede di incrociare il disegno con dati raccolti altrove e collocato più avanti nel flusso, quando una modifica costa costruzione e non solo ridisegno. La seconda riguarda la trascrizione dei dati: i parametri raccolti nella shotlist devono essere riportati a mano, shot per shot, nello strumento in cui la sequenza viene effettivamente allestita, con un lavoro che si ripete identico a ogni riga del documento e che cresce linearmente con il numero di shot dell'episodio.

Sono questi due aspetti — la collocazione della verifica spaziale in un passaggio distinto e successivo allo storyboard, e la ripetitività della trasposizione dei dati — a costituire il bersaglio dell'automazione descritta nel Capitolo 3, che mantiene inalterata la sequenza logica delle fasi appena descritte e interviene invece sul modo in cui esse vengono eseguite.

2.2 Tecnologie CG e pipeline: Blender e Unreal Engine

Le due tecnologie abilitanti scelte per la parte sperimentale di questa tesi rispondono a esigenze complementari: l'automazione flessibile della preproduzione da un lato, la visualizzazione in tempo reale dall'altro.

Blender è un software di creazione di contenuti 3D distribuito con licenza GNU GPL, che espone un'estesa API Python (`bpy`), documentata ufficialmente dalla Blender Foundation, per manipolare programmaticamente scene, oggetti, materiali e il motore di rendering stesso [9]. È proprio questa possibilità di estensione via scripting, più che il solo risparmio sui costi di licenza, a rendere Blender

una piattaforma adatta alla costruzione degli strumenti di automazione della preproduzione descritti nel Capitolo 3. La letteratura ne documenta del resto l'impiego in produzioni professionali reali, come gli effetti visivi dell'ottava stagione di *The Walking Dead* e le cinematiche di *Mass Effect: Andromeda* realizzati dallo studio Goodbye Kansas, sottolineando come l'accessibilità economica del software si accompagni a un'architettura aperta all'estensione [10].

Unreal Engine 5.7 è un motore di rendering in tempo reale sviluppato da Epic Games, la cui adozione nell'ambito della Virtual Production cinematografica è cresciuta significativamente a partire dalla fine degli anni 2010 [11]. Le caratteristiche rilevanti ai fini di questa tesi includono il sistema di illuminazione globale dinamica Lumen e il sistema di upscaling temporale TSR, entrambi impiegati nel progetto [12]. A questi si aggiungono gli strumenti nativi di comunicazione in tempo reale (tra cui il plugin OSC) impiegati per la sincronizzazione con DragonFrame.

Al di fuori del contesto specifico della stop-motion, un caso di riferimento è la serie Disney+ *The Mandalorian*, che ha utilizzato il motore come base di StageCraft, la piattaforma di Virtual Production sviluppata da Industrial Light & Magic che combina pareti LED e rendering in tempo reale: in questa produzione, la luce emessa dalle pareti LED che proiettano l'ambiente virtuale illumina direttamente attori e oggetti di scena fisici. Una modifica all'illuminazione o all'ambiente digitale si riflette pertanto in modo immediato sulle condizioni di luce del set reale, preservando la continuità visiva ed eliminando gran parte del lavoro di relighting e compositing in post-produzione [11]. Un caso ulteriore, questa volta di animazione seriale, è *Love, Death + Robots* di Netflix, la cui produzione si è appoggiata a Unreal Engine per il rendering finale di alcuni episodi, accelerando la pipeline di animazione [11].

Un'ultima ragione, di natura non tecnica ma di continuità tra le due fasi del lavoro, ha concorso all'accostamento di questi due software: l'addon open source Blender For UnrealEngine (§3.9) automatizza il trasferimento degli asset dall'uno all'altro, rendendo riutilizzabili in Virtual Production i modelli già costruiti per la preproduzione anziché richiederne la ricostruzione nel secondo ambiente.

2.3 Virtual Production at Model Scale, Single-DCC ed ecosistema software

Il riferimento più diretto e recente per l'inquadramento teorico di questa tesi è il progetto *Virtual Production at Model Scale* (VPaMS), condotto da Aardman Animations nell'ambito del programma di ricerca MyWorld (finanziato UKRI) e documentato sia in sede accademica [1] sia in sede industriale, con una presentazione al SIGGRAPH 2025 a cura di Gray, Bolotova, Ling e Cresdee, dal titolo *Digital Meets Handmade: Can VP enhance stop-motion?* [13]. Il progetto nasce in risposta

diretta alla pressione descritta in §2.1, con l'obiettivo dichiarato di produrre più minuti di animazione a parità di tempo e di spazio disponibile in studio, integrando strumenti digitali — blockout virtuale, digital twin, Virtual Production — lungo l'intera pipeline, dalla story fino alla consegna finale.

La scala a cui il progetto opera rende concreta questa pressione: a piena capacità produttiva, lo studio può avere fino a cinquanta unità di produzione individuali attive simultaneamente, ciascuna costituita da un animatore e da un numero ridotto di collaboratori, con una resa incrementale di pochi secondi — o frazioni di secondo — di animazione finita al giorno per unità. Questa parallelizzazione, necessaria per sostenere i tempi di consegna di una produzione seriale, comporta la replicazione fisica di un numero corrispondente di set e allestimenti di ripresa, con un impatto diretto sui costi e sugli spazi richiesti dallo studio.

Il caso Aardman colloca questa pressione all'estremo opposto della scala produttiva rispetto al contesto di questa tesi: una produzione priva della possibilità di moltiplicare le unità di ripresa affronta lo stesso vincolo di fondo — il costo e il tempo richiesti dalla costruzione di ogni set fisico — partendo da una sola unità disponibile, per cui non è la parallelizzazione a pesare sui costi, ma la capacità di quell'unica unità di restituire in immagine un ambiente più ampio di quanto sia stato effettivamente costruito. È in questa versione del problema, non in quella di scala industriale, che si inserisce la proposta di questa tesi: la Set Extension digitale in Virtual Production (Capitolo 4) sostituisce con uno sfondo virtuale condiviso la porzione di scenografia che una produzione di piccole dimensioni non avrebbe altrimenti le risorse per costruire fisicamente.

Più in generale, l'affermarsi della Virtual Production come fenomeno di settore è oggetto di un filone di ricerca accademica a sé stante, di cui lo stesso Livingstone si avvale nel descrivere le "emergent orthodoxies" della disciplina. Sulla stessa rivista, Swords e Willment propongono un'agenda di ricerca sistematica sull'emergere della Virtual Production come pratica industriale, discutendone tanto i benefici dichiarati quanto le criticità meno esplorate, incluso l'impatto ambientale delle infrastrutture LED richieste [14].

Un argomento a sostegno della scelta di concentrare la preproduzione in un solo software (approccio *Single-DCC*) trova ampio riscontro in letteratura. Il problema dell'interoperabilità tra strumenti diversi è riconosciuto come una criticità storica del settore VFX e animazione [15]. Restare in un solo software durante l'intera fase di preproduzione elimina alla radice il problema, anziché mitigarlo con un formato di interscambio come Universal Scene Description (USD).

Per l'approccio specifico del Capitolo 3, un precedente produttivo diretto è *Ubisoft Shot Manager* [16], un addon open source che introduce un'entità "shot" nelle scene Blender, nato nel 2020 per la produzione della serie *Rabbids Invasion*.

Va infine segnalato che il lavoro di ricerca di Aardman discute esplicitamente la possibilità di un collegamento in tempo reale tra DragonFrame e un motore di

rendering real-time come Unreal Engine, pur senza proporre una soluzione tecnica pubblicata: è precisamente in questo spazio, la sincronizzazione puntuale tra scatto fisico e scena virtuale, che si colloca il contributo sperimentale del Capitolo 4 di questa tesi.

Il quadro delineato in questo capitolo individua quindi due margini di intervento distinti ma complementari: da un lato la verifica spaziale differita e il costo di allestimento di uno storyboard tridimensionale, dall'altro lo spazio tecnico lasciato aperto dalla letteratura sul collegamento tra acquisizione e rendering in tempo reale. I due capitoli seguenti descrivono gli strumenti sviluppati per intervenire su ciascuno di essi.

Capitolo 3

Automazione della pre-produzione e preparazione degli asset in Blender

Il capitolo descrive la progettazione e l'implementazione dello strumento software sviluppato per automatizzare la fase di pre-produzione della pipeline stop-motion, riducendo il tempo dedicato ad attività ripetitive e introducendo un livello di verifica spaziale — lo *storyboard tridimensionale* — non disponibile negli strumenti 2D tradizionali.

Il flusso di lavoro che precede la compilazione della shotlist segue le fasi tradizionali della pre-produzione: a partire dalla sceneggiatura dell'episodio viene condotto lo spoglio, da cui si ricavano le scelte di inquadratura per ciascuna scena, in base ai personaggi coinvolti, alle scenografie previste e alle indicazioni di regia definite nella Storyboard Bible di produzione. Il file `shotlist.xlsx` (§3.3) viene quindi compilato shot per shot sulla base di queste decisioni.

3.1 Architettura generale della pipeline di pre-produzione

Il sistema è organizzato in fasi sequenziali e cooperanti, ciascuna implementata come modulo Python autonomo eseguibile all'interno dell'ambiente Blender (`bpy`): la definizione del database di produzione in formato Excel; la generazione procedurale delle scene 3D a partire dai dati; la parametrizzazione del rig dei personaggi tramite

driver matematici; la preparazione tecnica degli asset per l'esportazione; il rendering automatizzato dei singoli shot e la composizione dell'animatic finale con overlay informativo.

L'automazione dell'intero processo nasce dalla volontà di evitare passaggi manuali ripetitivi e di ridurre il tempo di costruzione del file di produzione, mantenendo l'intero flusso di pre-produzione (script `Camera` per la generazione degli shot, script `MASTER` per il Sequencer, l'esportazione e il burn-in dei metadati sull'animatic) dentro un solo software.

Sequenza operativa. I due script vanno eseguiti in ordine, dall'editor di testo integrato all'interno del workspace `Scripting` di Blender. Il primo, `Camera`, legge la shotlist e genera per ciascuno shot la `Scene` corrispondente, la camera con la propria focale, l'oggetto Grease Pencil parentato ad essa e le istanze degli asset indicati (§3.5). Nello stesso passaggio lo script crea la scena contenitore `MASTER_RENDER` e vi accoda come strip del Video Sequence Editor tutte le `Scene` generate, rendendo l'episodio navigabile sulla timeline. Il secondo script, `MASTER`, non interviene sulla struttura della sequenza ma ne prepara il rendering: configura sulla scena `MASTER_RENDER` il percorso di output con versioning automatico e il burn-in dei metadati (§3.10), e per questo presuppone che la scena sia già stata creata dal primo script. Per produrre l'animatic è quindi sufficiente selezionare la scena `MASTER_RENDER` e avviare il rendering dell'animazione (Render → Render Animation): il processo percorre la timeline del Sequencer attraversando in sequenza tutti gli shot dell'episodio, e applica a ciascun fotogramma i metadati corrispondenti allo shot attivo in quel punto della timeline. La Figura 3.1 riassume la sequenza, affiancando a ciascuna fase l'artefatto che essa produce.

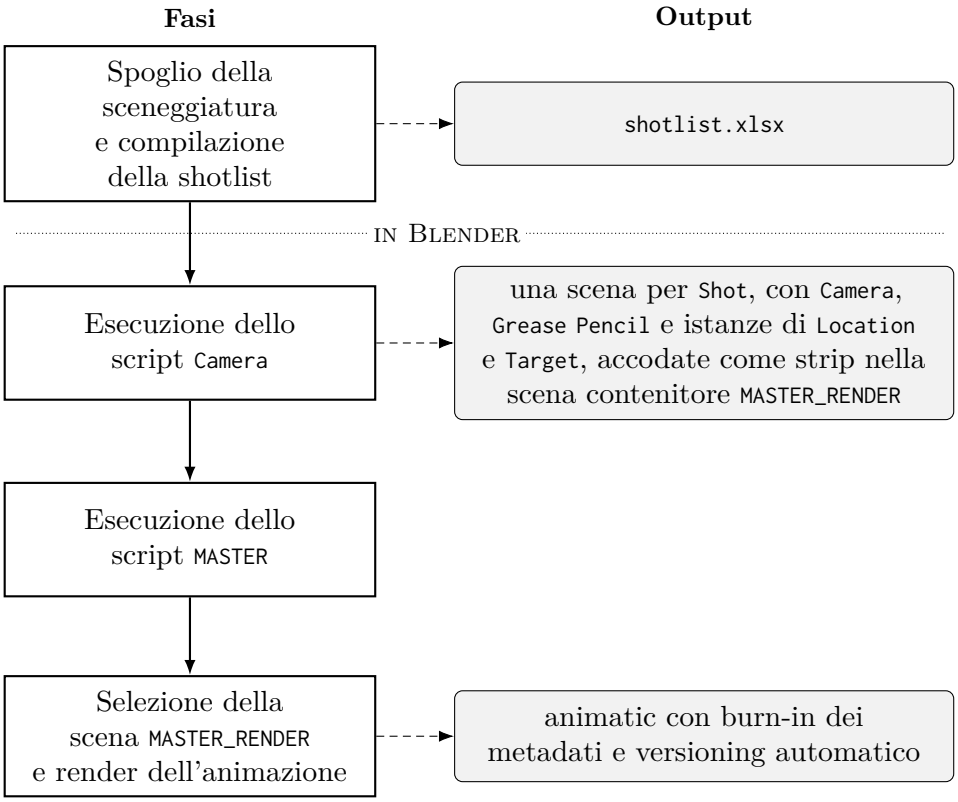


Figura 3.1: Architettura della pipeline di pre-produzione: a sinistra le fasi eseguite dall'operatore, a destra gli artefatti che ciascuna di esse produce. Le istanze di `Location` e `Target` sono collegate per linking alla libreria asset condivisa (§3.6).

3.2 Ambiente Blender: workspace Storyboarding, Video Sequence Editor e Grease Pencil

L'intera pipeline di pre-produzione è costruita sopra tre categorie native di Blender, organizzate all'interno del workspace *Storyboarding* (Figura 3.2) — una configurazione dell'interfaccia pensata specificamente per il lavoro di pre-visualizzazione.



Figura 3.2: Il workspace *Storyboarding* di Blender: si hanno vista 3D, VSE e Dope Sheet tutte nella stessa schermata.

Il workspace Storyboarding, introdotto nelle versioni recenti di Blender, combina in un unico flusso di lavoro non distruttivo il disegno 2D, il layout 3D e gli strumenti di montaggio video, con l'obiettivo dichiarato di colmare la distanza tra lo storyboard tradizionale bidimensionale e la pre-visualizzazione tridimensionale, supportando sia storyboard disegnati a mano sia *layout board* 3D (i cosiddetti "animatic 3D") [9]. L'organizzazione tipica prevede più `Scene` distinte, ciascuna corrispondente a uno shot con il proprio set di asset e la propria camera, oggetti Grease Pencil per gli schizzi rapidi, e un Video Sequence Editor che assembla le `Scene` in sequenza — la stessa struttura ripresa e automatizzata dalla pipeline descritta in questo capitolo. Un aspetto rilevante di questo workspace, sottolineato anche dalla documentazione ufficiale, è la natura interconnessa dei dati: una modifica alla posa di un personaggio, al posizionamento di una camera o a uno schizzo Grease Pencil si aggiorna automaticamente nel montaggio, senza richiedere una nuova esportazione o sincronizzazione manuale — un vantaggio sfruttato pienamente dagli script di automazione descritti nelle sezioni successive.

Il primo elemento strutturale è la corrispondenza adottata tra le entità native di Blender e quelle della produzione: ogni sequenza narrativa dell'episodio (ad esempio SQ011) è realizzata in un singolo file `.blend`, mentre ciascuno shot previsto dalla shotlist per quella sequenza è rappresentato da una `Scene` di Blender distinta all'interno dello stesso file. Questa scelta consente di sfruttare la gestione nativa delle `Scene` — ciascuna con la propria camera, il proprio set di oggetti visibili e i propri parametri di rendering — come unità elementare della pipeline, invece di dover simulare shot multipli con tecniche meno dirette (ad esempio marker o range di frame all'interno di un'unica scena). Questa corrispondenza è confermata direttamente dal codice della funzione `sync_master_sequencer` (§3.5), che crea una `Scene` per ciascun shot della shotlist.

Il secondo elemento è il *Video Sequence Editor* (VSE), il compositore/editor non lineare nativo di Blender, utilizzato in questa pipeline non per il montaggio di filmati esterni ma per assemblare le `Scene` stesse: ogni Scene-shot viene inserita come strip di tipo `SCENE` sulla timeline della scena master `MASTER_RENDER` (§3.5)(Figura 3.3), rendendo possibile scorrere, riordinare e visualizzare in sequenza tutti gli shot dell'episodio direttamente dentro Blender, senza esportazioni intermedie.

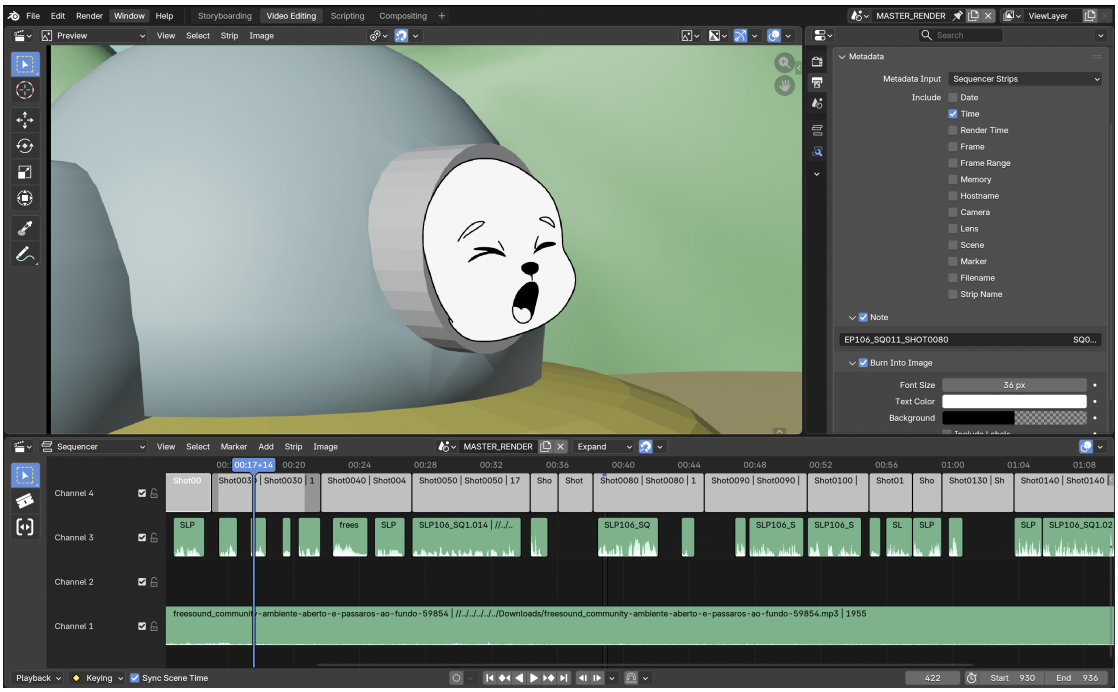


Figura 3.3: Il workspace *Video Editing* di Blender: gli shot dell'episodio sono accodati in sequenza nel Video Sequence Editor della scena `MASTER_RENDER`, come strip di tipo `SCENE`.

Il terzo elemento è il *Grease Pencil*, la categoria di oggetti di Blender pensata per il disegno 2D nello spazio 3D. A differenza di un livello di annotazione puramente bidimensionale, un oggetto Grease Pencil esiste nella scena con una propria posizione, rotazione e relazione di parentela con altri oggetti (nel caso specifico, con la camera dello shot, si veda §3.7): questo permette di tracciare schizzi di storyboard che restano ancorati al punto di vista della camera anche quando la scena viene esplorata o la camera stessa viene animata, unendo la rapidità dello schizzo 2D tradizionale alla persistenza spaziale di un oggetto 3D.

Infine, la gestione degli asset condivisi tra tutti gli shot si appoggia sull'*Asset Browser* nativo di Blender, configurato come un vero e proprio catalogo dei personaggi della serie. L'intera libreria condivisa risiede in un unico file Blender dedicato (`LibreriaAssetsDef.blend`), registrato come Asset Library nelle Preferenze di Blender con il nome `AssetsPuppets` (§3.6), e gli asset al suo interno sono organizzati tramite *Asset Catalogs* — le categorie native dell'Asset Browser, che permettono di raggruppare e sfogliare i personaggi per nome o tipologia direttamente dall'interfaccia, invece di navigare manualmente la struttura di cartelle sul disco. Il file raccoglie, organizzati per *Asset Catalogs*: i personaggi della serie, le scenografie (elementi di set, come descritti nel documento *Slippets Asset Sheet Characters*, §1.4), una libreria di frecce (elementi grafici 2D di supporto a storyboard/animatic), e una libreria di *bocche disegnate in 2D*, intercambiabili per comporre le diverse espressioni facciali dei personaggi, destinate a essere sovrapposte in compositing anziché modellate tridimensionalmente tramite Shape Key. Fa eccezione il personaggio Calzino Puzzone, che dispone di un controllo aggiuntivo di apertura bocca tramite Shape Key (§3.8), utilizzato in affiancamento alle bocche 2D per facilitare alcune espressioni facciali specifiche. Questa configurazione rende la libreria consultabile e trascinabile in scena anche al di fuori del flusso automatizzato, ad esempio per verifiche manuali o per l'aggiunta di nuovi asset da parte di altri collaboratori.

I cataloghi adottati — `Target`, `Location`, `Mouth` e `Arrows` — riprendono deliberatamente la nomenclatura delle colonne della shotlist (§3.3): il valore letto dal documento di produzione corrisponde così direttamente alla categoria in cui l'asset va cercato, senza necessità di una tabella di conversione tra i due sistemi di nomi. Le bocche 2D raccolte nel catalogo `Mouth` meritano una precisazione sul loro impiego, che è il medesimo in entrambi gli stadi della lavorazione. Nella produzione finale esse non corrispondono ad alcun elemento fisico costruito e non vengono quindi sostituite sul pupazzo secondo la tecnica della *replacement animation*: sono invece sovrapposte in compositing bidimensionale sui fotogrammi già ripresi. Nell'animatic tridimensionale descritto in questo capitolo si adotta lo stesso principio: l'espressione del volto è ottenuta dalle Shape Key pilotate dai driver (§3.8) per palpebre e narici, mentre la bocca viene aggiunta come elemento 2D sovrapposto al pupazzo virtuale. L'anteprima prodotta in pre-produzione risulta così coerente, nel meccanismo oltre che nell'aspetto, con la resa prevista per il prodotto finale.

Per alcuni personaggi lo *Slippets Asset Sheet Characters* (§1.4) non documenta tutte le viste necessarie alla modellazione tridimensionale. In questi casi sono state generate tramite intelligenza artificiale immagini di riferimento nelle viste frontale, laterale, dall'alto e dal basso, impiegate come guida visiva durante la modellazione in Blender — una pratica adottata per colmare le lacune di documentazione visiva dei personaggi non completamente coperti dal materiale di produzione originale.

3.3 Il database di produzione: struttura e validazione della Shotlist

Il flusso operativo nasce da una gestione strutturata dei dati di regia tramite un file `shotlist.xlsx`, organizzato su due fogli di lavoro: `shotlistep#numeroepisodio`, contenente l'elenco degli shot, e `Assets`, contenente l'anagrafica degli elementi validati per il progetto. Gli esempi che seguono in questo capitolo sono tratti dalla produzione reale della serie *Slippets*(§1.4).

Campo	Descrizione	Esempio
`Shot`	Identificativo univoco dello shot	`Shot0000`, `Shot0010`, `Shot0020`, ...
`Focale`	Parametro ottico, in millimetri	28, 50, 85, 100
`Camera`	Codifica dell'inquadratura	TOT1, FI1, PP1, DETT0, ...
`Tipologia`	Tipo di inquadratura, in forma estesa	Totale, Intera, Primo piano, ...
`Movimento`	Dinamica della camera	Statica, Pan, Tilt
`Target`	Elementi da istanziare nello shot	Koru, Clio, CalzinoPuzzone, ...
`Location`	Location da istanziare nello shot	AreaCase, Foresta, ...
`Note`	Indicazioni di regia e battute	(testo libero)

Tabella 3.1: Struttura dei campi della shotlist.

La convenzione di naming adottata in tutta la pipeline — identificativo di scena nella forma `ep<NNN>_sq<NNN>_sh<NNNN>`, con l'identificativo di shot a quattro cifre, completato con zero non significativi (`Shot0000`, `Shot0010`, ecc.) usato internamente come nome delle Scene Blender — deriva direttamente dalla sezione "Software/Nomenclatura" della Storyboard Bible di produzione, che specifica questa suddivisione come funzionale al software di gestione della produzione, insieme alla convenzione di naming per le versioni dell'animatic (`SLP_SB_<numeroepisodio>_animatic_v1`, `_rtk_v1`, `_v2`, ecc.), impiegata dallo script di esportazione (§3.10).

3.4 Parsing nativo del formato .xlsx

L'estrazione dei dati dal file Excel è affidata alla funzione `read_excel_no_lib`, che opera senza dipendenze da librerie Python esterne. Questa scelta non è motivata da una presunta semplicità di lettura del formato xlsx lato Python, bensì dal vincolo opposto: l'interprete Python integrato in Blender non include librerie di terze parti come `openpyxl` o `pandas`, per cui la lettura del file richiede un parser nativo che apra l'xlsx come archivio ZIP e navighi direttamente l'XML del foglio di calcolo (`sharedStrings.xml` e `sheet1.xml`), dereferenziando riga per riga gli indici delle stringhe condivise (attributo `t="s"`). Il formato xlsx resta comunque la scelta più adatta per la compilazione e la lettura del file da parte del team di produzione al di fuori di Blender — è su questo fronte, non su quello della lettura via Python, che il formato si è rivelato conveniente.

Listato 3.1: Parsing nativo del formato .xlsx

```
def read_excel_no_lib(file_path):
    with zipfile.ZipFile(file_path, 'r') as zip_ref:
        shared_strings = [...]          # da xl/sharedStrings.xml
        sheet_xml = ET.fromstring(zip_ref.read('xl/worksheets/sheet1.xml'))
        # ricostruzione riga per riga, dereferenziando gli indici
        # delle stringhe condivise (attributo t="s")
```

3.5 Generazione procedurale di Scene, Camera e Sequencer

La funzione orchestratrice `sync_master_sequencer` opera in due passaggi sequenziali sull'intero contenuto della shotlist, non riga per riga in isolamento.

Primo passaggio — creazione di scene, camere, Grease Pencil e istanze asset. Per ciascuna riga della shotlist:

- Viene creata (o recuperata, se già esistente) una `Scene` di Blender il cui nome corrisponde esattamente al valore della colonna `Shot` (es. `Shot0010`), troncato all'eventuale suffisso dopo un punto.
- Viene creata (o recuperata tramite corrispondenza sul nome base, tollerante ai suffissi numerici automatici di Blender come `.001`) una camera, il cui oggetto prende il nome dal valore della colonna `Camera` della shotlist (es. `TOT1`, `FI1`, `FI2`, `PP1`, `PP2`, `PP3`, `DETT0`, `DETT1`, `DETT2`) — una nomenclatura che codifica il tipo di inquadratura (Totale, Figura Intera, Primo Piano, Dettaglio), con un numero progressivo che distingue le diverse occorrenze dello stesso tipo di inquadratura nell'episodio. Il data-block della camera (distinto dall'oggetto,

per convenzione Blender) prende invece il nome `Data_<ShotID>`. La focale della camera viene impostata leggendo il valore della colonna `Focale` della shotlist (in millimetri, es. `28`, `50`, `85`, `100`) e applicandola al data-block tramite la funzione `apply_focal_length` (Listato 3.2); l'operazione viene ripetuta ad ogni sincronizzazione anche per le camere già esistenti, cosicché un aggiornamento del valore in shotlist si rifletta immediatamente sulla scena senza richiedere l'eliminazione manuale della camera.

- Viene istanziato un oggetto *Grease Pencil* condiviso (`GP_Storyboard`), parentato alla camera dello shot (§3.7).
- Per ciascun valore non vuoto nelle colonne `Location` e `Target` (che possono contenere più asset separati da virgola o punto e virgola) viene creata un'istanza dell'asset corrispondente tramite `link_collection_as_instance`, con nome `INST_<NomeAsset>_<ShotID>` (es. `INST_Koru_Shot0010`) — un Empty di tipo istanza-collezione, collegato alla collezione originale linkata dalla libreria asset (§3.6), e immediatamente convertito in Library Override tramite `make_override_library`.
- Lo shot viene infine accodato come strip di tipo `SCENE` nel Video Sequence Editor della scena `MASTER_RENDER`, con una durata di default di 100 frame, posizionato progressivamente in coda agli shot già presenti.

Listato 3.2: Applicazione della focale letta dalla shotlist al data-block della camera in Blender

```python
def apply_focal_length(cam_obj, row, shot_id, log_block):
    raw_focale = str(row.get('Focale', '')).strip()
    if not raw_focale:
        write_to_log(log_block, f"AVVISO: nessuna Focale per {shot_id}")
        return
    try:
        cam_obj.data.lens = float(raw_focale)
    except ValueError:
        write_to_log(log_block, f"AVVISO: Focale '{raw_focale}' non numerica")
```

Secondo passaggio — consolidamento globale degli Override e indipendenza delle animazioni. Una volta processate tutte le righe, lo script itera nuovamente su tutte le scene create per garantire che ogni istanza sia stata effettivamente convertita in Override (un secondo tentativo globale, necessario perché la conversione eseguita inline nel primo passaggio non risultava sempre affidabile shot per shot). Per ogni oggetto di tipo Armatura risultante da un Override viene quindi verificata la presenza di una *Action* dedicata e indipendente, con prefisso `ACT_`. L'Action viene normalmente creata già nel primo passaggio, al momento dell'istanziazione (Listato 3.3), con nome `ACT_<NomeAsset>_<ShotID>`; il secondo passaggio la crea soltanto per le armature che ne fossero ancora prive, ad esempio perché la conversione inline non era andata a buon fine. È questo il meccanismo tecnico che garantisce l'indipendenza delle animazioni tra shot diversi che condividono lo stesso personaggio, verificata nei test descritti in §3.11. Chiude il processo la disattivazione dell'illuminazione di scena sui layer Grease Pencil (`use_lights = False`, per una resa piatta e non influenzata dalle luci) e l'aggiornamento del frame finale della scena master in base alla somma delle durate di tutti gli shot.

Sfida tecnica. Le prime versioni dello script hanno presentato due criticità distinte, entrambe risolte nella versione finale. La prima riguardava la lettura del file xlsx: il tentativo iniziale di affidarsi a un parsing più semplice o a librerie non disponibili nell'ambiente Python di Blender ha reso necessario lo sviluppo del parser nativo `read_excel_no_lib` (§3.4). La seconda criticità riguardava l'affidabilità della conversione delle istanze in Library Override: eseguire `make_override_library` immediatamente dopo la creazione di ciascuna istanza, shot per shot all'interno dello stesso ciclo, non garantiva un esito consistente per tutte le istanze. La soluzione adottata è stata separare il processo nei due passaggi distinti descritti sopra, eliminando l'inconsistenza riscontrata nella prima versione dello script.

Nota sulla gestione della camera. In una fase iniziale di sviluppo era stata valutata l'aggiunta automatica di una constraint di tipo *Track To* (Look At) sulla camera di ciascuno shot, per farla puntare automaticamente al personaggio target indicato nella shotlist. La soluzione è stata successivamente scartata e non è presente nella versione finale dello script: si è rivelata scomoda in scene con interazioni tra più personaggi o con movimenti che avrebbero richiesto un ri-orientamento non desiderato della camera in corso di scena, complicando più che semplificando l'inquadratura rispetto al posizionamento manuale.

3.6 Library Override e instancing degli asset

La funzione `link_collection_as_instance` gestisce il collegamento degli asset alla scena-shot. Gli asset vengono collegati tramite link (`bpy.data.libraries.load(..., link=True)`) anziché tramite append (copia locale), per garantire la coerenza delle

modifiche: un aggiornamento a un asset nella libreria condivisa (una correzione al modello di un personaggio, una modifica al rig, un cambiamento a un elemento di scenografia) si propaga automaticamente a tutte le scene-shot in cui quell'asset è utilizzato, senza necessità di aggiornare manualmente ciascuna copia. È precisamente per preservare questo vantaggio, pur avendo bisogno di animazioni indipendenti per ciascuno shot, che il sistema si appoggia al meccanismo nativo di Library Override di Blender: l'asset resta collegato e sincronizzato con la libreria per tutto ciò che non è animazione, mentre l'animazione stessa (tramite le Action dedicate `ACT_<NomeAsset>_<ShotID>`, §3.5) resta locale e indipendente per ogni shot.

La collezione sorgente viene individuata con una priorità in due passi: per ciascun nome estratto dai campi `Location`/`Target`, lo script cerca anzitutto la collezione originale già linkata in memoria (filtrando esplicitamente `col.library is not None` per escludere eventuali override locali già creati); se non trovata, la carica da libreria, con fonte primaria la libreria Asset registrata nelle Preferenze di Blender con il nome `AssetsPuppets` — la stessa consultabile manualmente nell'Asset Browser (§3.2) — e un percorso assoluto cablato nel codice, mantenuto solo come ripiego di sicurezza, attivato (con relativo avviso in log) esclusivamente se quella libreria non risulta registrata sulla macchina in uso.

Ogni istanza viene poi convertita in un *Library Override* gerarchico tramite:

Listato 3.3: Creazione dell'istanza di un asset, conversione in Library Override e assegnazione di una Action dedicata

```
def link_collection_as_instance(target_scene, asset_name, shot_id):
    clean_asset_name = asset_name.strip()
    target_instance_name = f"INST_{clean_asset_name}_{shot_id}"
    [...]  # controllo anti-duplicato e ricerca della collezione linkata

    if target_col:
        obj = bpy.data.objects.new(target_instance_name, None)
        obj.instance_type = 'COLLECTION'
        obj.instance_collection = target_col
        target_scene.collection.objects.link(obj)

        [...]  # attivazione della scena di destinazione e selezione
        bpy.ops.object.make_override_library(
            collection=target_col.override_library.hierarchy_root
            if target_col.override_library else None)

        for scene_obj in target_scene.objects:
            if scene_obj.override_library:
                if scene_obj.type == 'ARMATURE':
                    if not scene_obj.animation_data:
                        scene_obj.animation_data_create()
                    if (not scene_obj.animation_data.action or
```

```
23                          not scene_obj.animation_data.action.name.startswith("
    ACT_")):
24                      new_act = bpy.data.actions.new(
25                          name=f"ACT_{clean_asset_name}_{shot_id}")
26                      scene_obj.animation_data.action = new_act
27          return obj
```

Questo crea, per ogni shot, una copia locale ed editabile dell'intera gerarchia del personaggio, mantenendo mesh e materiali condivisi con l'asset originale.

3.7 Setup ottico e layering 2D/3D con Grease Pencil

La funzione `setup_gpencil_for_drawing` configura, per ciascuna camera generata, un oggetto Grease Pencil (`GP_Storyboard`) imparentato alla camera e posizionato a distanza fissa dal piano immagine ($Z = -1.5\,\text{m}$ nello spazio locale della camera).

Listato 3.4: Setup dell'oggetto Grease Pencil parentato alla camera

```
def setup_gpencil_for_drawing(scene, camera_obj):
    gp_name_base = "GP_Storyboard"
    existing_gp = None
    for obj in scene.objects:
        if re.match(r"^GP_Storyboard(\.\d{3})?$", obj.name):
            existing_gp = obj
            break

    if existing_gp:
        if existing_gp.parent != camera_obj:
            existing_gp.parent = camera_obj
            existing_gp.location = (0, 0, -1.5)
        gp_obj = existing_gp
    else:
        gp_data = bpy.data.grease_pencils.new(gp_name_base)
        gp_obj = bpy.data.objects.new(gp_name_base, gp_data)
        scene.collection.objects.link(gp_obj)
        gp_obj.parent = camera_obj
        gp_obj.location = (0, 0, -1.5)
        gp_obj.show_in_front = True
        if not gp_data.layers:
            gp_data.layers.new("Lines", set_active=True)

    for layer in gp_obj.data.layers:
        layer.use_lights = False
    return gp_obj
```

La funzione è tollerante a un oggetto `GP_Storyboard` già esistente in scena (incluso il suffisso numerico automatico di Blender, es. `.001`, gestito tramite espressione regolare): se lo trova, si limita a correggerne il parent e la posizione anziché crearne uno nuovo, evitando duplicati a ogni successiva sincronizzazione. Il parametro `show_in_front` garantisce che lo schizzo resti visibile anche quando si trova geometricamente dietro ad altri oggetti della scena, mentre `use_lights = False` sui layer disattiva l'influenza dell'illuminazione di scena sul tratto disegnato, per una resa piatta coerente con la funzione di annotazione dello storyboard (Figura 3.4).



Figura 3.4: Schizzo di storyboard su Grease Pencil in vista camera.

3.8 Control rig parametrico e driver

Un *driver*, nella terminologia di Blender, è un meccanismo che permette di controllare il valore di una proprietà per mezzo di una funzione o di un'espressione matematica, anziché tramite fotogrammi chiave animati manualmente: il valore di una o più variabili di input (posizioni, rotazioni, o altre proprietà di scena) viene combinato secondo una funzione predefinita oppure un'espressione Python arbitraria (*Scripted Expression*), producendo un singolo valore che viene poi applicato alla proprietà controllata (la *proprietà guidata*) [9]. Questo meccanismo consente di costruire controlli indiretti — ad esempio, la posizione di un osso controller che determina, tramite un'espressione, l'apertura di un occhio o il valore di una Shape Key — invece di dover animare direttamente ogni singolo parametro del rig.

Il sistema di driver per l'apertura di palpebre, bocca e narici, applicato a ciascun personaggio della serie sulla base dei modelli documentati nello *Slippets Asset Sheet Characters* (§1.4), segue una logica comune: il vincolo tra 0 e 1 non è imposto da un `clamp()` nell'espressione del driver, ma dal *limite fisico di movimento* imposto sull'osso controller tramite un constraint `Limit Location`, mentre il driver stesso è un'espressione lineare non vincolata (*Scripted Expression*), calibrata in modo che il valore raggiunga esattamente 0 o 1 quando l'osso raggiunge l'estremità della propria corsa consentita.

Per ciascun controller (`ControllerLids` per le palpebre, `ControllerNarici` per le narici, entrambi ossa del rig `PinkRig.001`), il driver è collegato direttamente al valore (`Value`) della Shape Key corrispondente, con variabile di input `var` impostata sulla posizione X dell'osso (`X Location`, Local Space). La formula generale è:

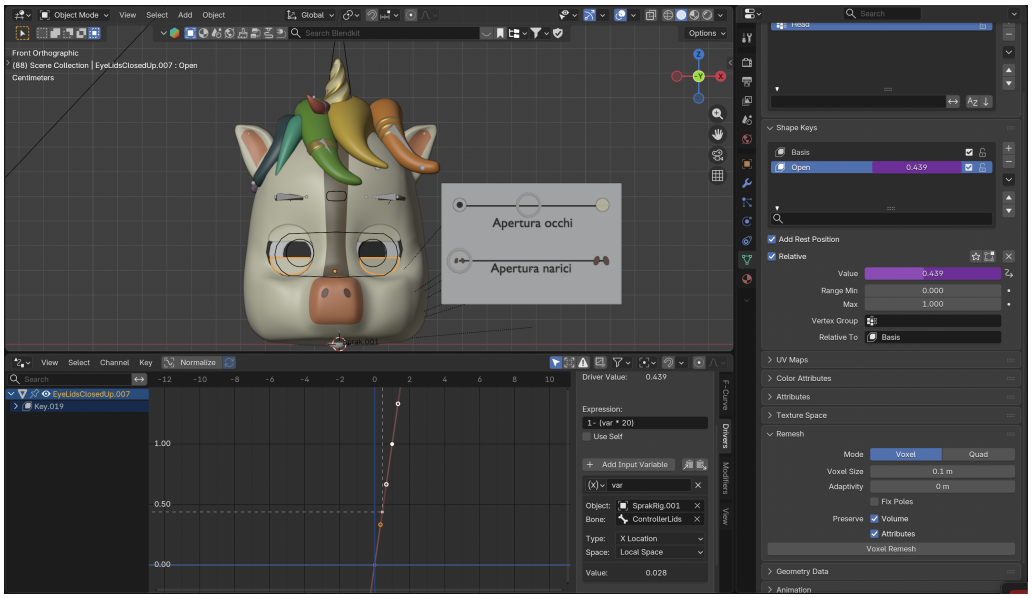
$$V = 1 - (\text{var} \times K) \tag{3.1}$$

dove K è il reciproco del range di movimento consentito all'osso su quell'asse: se l'osso può muoversi da 0 a un valore massimo R, si sceglie $K = 1/R$, cosicché a var $= 0$ (osso a riposo) il driver valga 1 (Shape Key attiva, es. occhio aperto), e a var $= R$ (osso a fine corsa) il driver valga esattamente 0 (Shape Key disattivata, es. occhio chiuso).

Nel caso delle palpebre, l'osso `ControllerLids` ha un range di movimento di 0.05 m, da cui $K = 1/0.05 = 20$ (Figura 3.5). Nel caso delle narici, l'osso `ControllerNarici` ha invece un range di 0.058 m: applicare lo stesso moltiplicatore 20 avrebbe prodotto un valore fuori scala all'estremità della corsa ($1 - (0.058 \times 20) = -0.16$, invece di 0); il moltiplicatore è stato quindi ricalcolato come $K = 1/0.058 \approx 17.241$. Questo evidenzia un aspetto metodologico rilevante: il moltiplicatore della formula *non è un valore arbitrario, né riutilizzabile tra controller diversi*, ma va ricalcolato ogni volta in funzione del range di movimento specifico impostato sul vincolo `Limit Location` di quel particolare osso.



(a) Valore del driver pari a 1

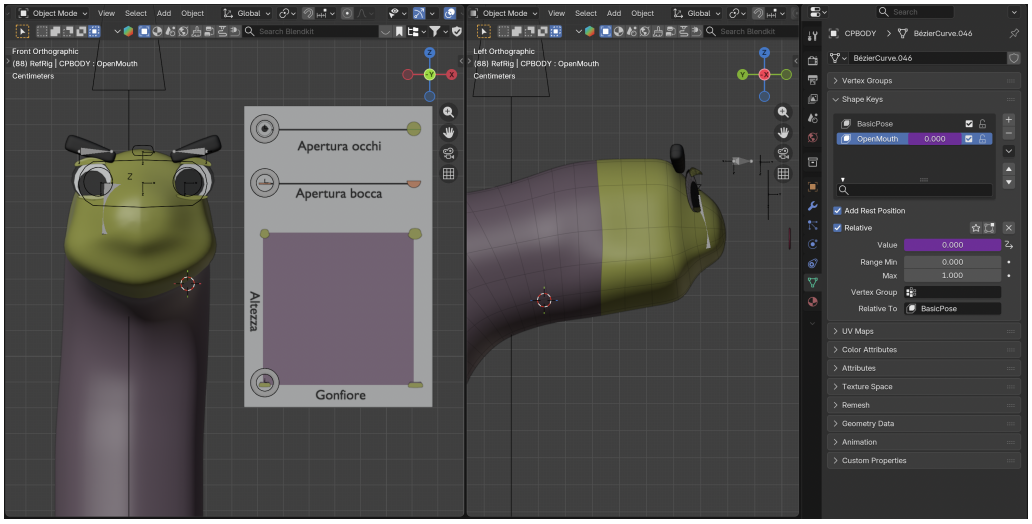


(b) Valore del driver minore di 1

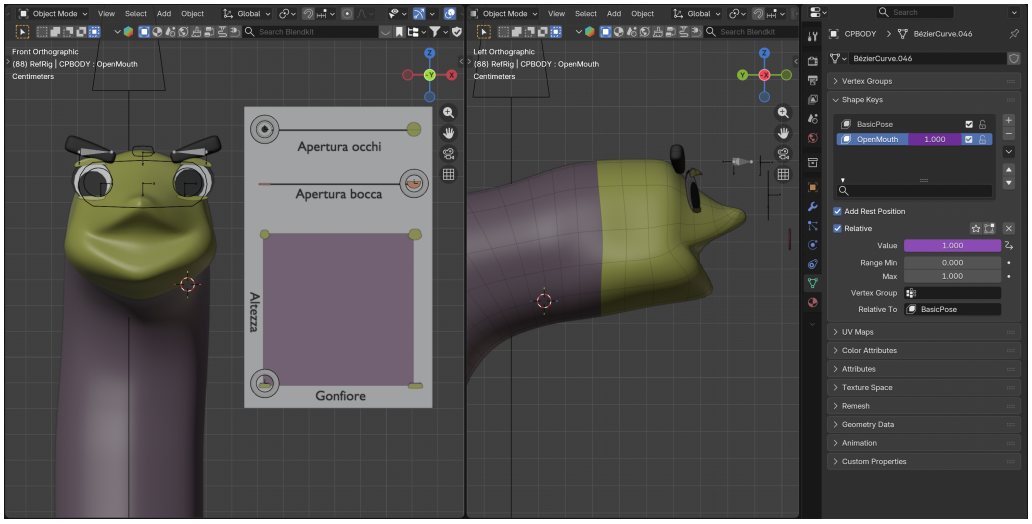
Figura 3.5: Driver di apertura e chiusura delle palpebre: il pannello di configurazione e l'effetto sulla Shape Key corrispondente.

Eccezione: apertura bocca di Calzino Puzzone. A differenza degli altri personaggi, per i quali le diverse espressioni della bocca sono ottenute sostituendo bocche disegnate in 2D (§3.2), Calzino Puzzone dispone di un controllo aggiuntivo

di apertura bocca tramite Shape Key (`OpenMouth`), impiegato in affiancamento alle bocche 2D per facilitare alcune espressioni facciali specifiche. Il driver è collegato alla posizione X dell'osso `ControllerBocca` del rig `CPRig` (Local Space), con formula var × 20 — una relazione diretta (non invertita, a differenza di palpebre e narici): muovendo il controller verso l'estremità della sua corsa, la bocca si apre progressivamente (Figura 3.6).



(a) Valore del driver pari a 0



(b) Valore del driver pari a 1

Figura 3.6: Driver di apertura della bocca del personaggio Calzino Puzzone, collegato alla Shape Key `OpenMouth`.

Occhi e sopracciglia, diversamente da palpebre, narici e dalla bocca di Calzino Puzzone, non sono controllati tramite driver su Shape Key: il loro movimento è ottenuto direttamente tramite la manipolazione delle ossa del rig, senza un sistema di espressioni matematiche dedicato.

Rig di deformazione corporea di Calzino Puzzone. Il personaggio è inoltre dotato di un sistema di rigonfiamento controllato da un controller planare (Empty) con due assi — X per la potenza del gonfiaggio, Y per l'altezza a cui il rigonfiamento si concentra lungo il corpo — che agisce su una catena di 8 Bendy Bones. Ciascun osso della catena riceve, sui parametri `Scale In X`, `Scale In Z`, `Scale Out X` e `Scale Out Z` (impostati identici tramite Copy/Paste Driver, per evitare una deformazione "a imbuto" tra ingresso e uscita dell'osso), un driver di tipo Scripted Expression nella forma:

$$S = 1 + \max\Big(0,\ (\mathrm{var} \times 8.6) \cdot \big(1 - |\mathrm{var}_{001} - Q| \times 40\big)\Big) \tag{3.2}$$

dove var è la posizione X del controller (potenza di gonfiaggio, costante di calibrazione 8.6 uguale su tutti gli ossi, tarata per un fattore di scala massimo di 1.5×), var_{001} è la posizione Y del controller (altezza), e Q è l'unico parametro che varia da osso a osso: la posizione assegnata a quello specifico osso lungo il range verticale del controller, distribuita in modo uniforme sugli 8 ossi della catena (0.0, 0.0083, 0.0166, 0.0249, 0.0331, 0.0414, 0.0497, 0.058). Il fattore 40 regola la nitidezza del falloff, ossia quanto il rigonfiamento resti localizzato attorno alla quota di ciascun osso anziché propagarsi lungo l'intera catena.

L'operatore $\max(0, \cdot)$ è stato introdotto per risolvere un problema riscontrato in una versione precedente della formula: senza questo vincolo, il termine di falloff poteva assumere valori negativi quando il controller non si trovava esattamente sulla quota di un dato osso, producendo un effetto di restringimento (de-scaling) non voluto invece di un valore neutro; il vincolo garantisce che il contributo di un osso "lontano" dalla posizione Y corrente del controller sia sempre pari a zero, lasciando il fattore di scala a 1.0 (neutro) anziché scendere sotto tale valore.

3.9 Preparazione degli asset 3D e esportazione con l'addon Blender For UnrealEngine (BFU)

Prima dell'esportazione verso Unreal Engine, gli asset richiedono una fase di preparazione tecnica: ottimizzazione della mappatura UV con attenzione alla *texel density*, e adozione di una nomenclatura standard (prefissi `SM_` per le Static Mesh, `M_` per i materiali, `T_` per le texture).

Il trasferimento degli asset così preparati verso Unreal Engine è automatizzato tramite l'addon open-source *Blender For UnrealEngine* (BFU), sviluppato da Xavier

Loux e distribuito pubblicamente su GitHub [17], compatibile con Blender dalla versione 2.8 in poi e con Unreal Engine 4 e 5 (gli script di importazione richiedono Unreal Engine 4.26 o superiore).

Il funzionamento dell'addon si articola in due fasi distinte, eseguite su due software diversi. Nella prima fase, lato Blender, l'addon esegue un *batch export* dell'intera scena — Static Mesh, Skeletal Mesh, animazioni, camere e dati di Sequencer (con prefisso `bfu_camera`) — organizzando automaticamente i file esportati (basati su un formato FBX modificato) in una struttura di cartelle coerente con le convenzioni di progetto di Unreal Engine. Contestualmente, l'addon genera anche uno o più script Python di importazione (tra cui `asset_import_script.py` e `run_unreal_import_script.py`), che non vengono mai eseguiti da Blender, ma sono scritti appositamente per essere eseguiti nell'ambiente Python interno di Unreal Engine.

Nella seconda fase, lato Unreal Engine, è questo script generato — non l'addon stesso, che non richiede alcun plugin installato lato Unreal — a occuparsi effettivamente dell'importazione: l'utente copia un comando pronto (fornito dall'addon tramite un pannello dedicato di Blender) e lo incolla nella console Python integrata di Unreal Engine, che lo esegue. Lo script legge i file esportati nella prima fase e ricostruisce gli asset nel Content Browser di Unreal, mantenendo nomenclatura, gerarchia di cartelle, e — nel caso delle camere — i parametri ottici (focale, aspect ratio, lens shift) e i tagli di montaggio già presenti nel Sequencer di Blender. È precisamente questo meccanismo a due fasi — esportazione via addon in Blender, importazione via script Python eseguito dentro Unreal Engine — a rendere possibile, con un intervento manuale limitato alla copia e all'esecuzione di un singolo comando, il trasferimento degli asset la cui osservazione è discussa in §5.5.

3.10 Export automatizzato dell'animatic e burn-in dei metadati

Lo script `MASTER` si appoggia al sistema nativo di *Stamp* di Blender (`render.use_stamp`, `use_stamp_note`, `use_stamp_time`), con sfondo semitrasparente (`stamp_background` = $(0.0, 0.0, 0.0, 0.25)$). La funzione `setup_render_path_and_metadata` viene registrata su tre handler differenti di Blender (`render_init`, `render_pre`, `frame_change_pre`), così da ricalcolare i metadati ad ogni cambio di fotogramma — necessario perché, scorrendo la scena master, lo shot e la camera attivi cambiano continuamente. Ad ogni fotogramma, la funzione individua la strip di tipo `SCENE` attiva nel Sequencer in base alla posizione corrente, ne ricava il nome dello shot e il nome della camera attiva nella scena corrispondente, e compone un testo in due colonne (identificativo shot a sinistra, sequenza e camera a destra) allineate tramite un calcolo di larghezza visiva approssimata carattere per carattere (funzione `get_visual_width`),

per compensare la resa a larghezza variabile del font di overlay di Blender. La stessa funzione gestisce anche il naming con versioning automatico del file di output (`SLP_SB_<numeroepisodio>_animatic_v<N>`, con incremento automatico di N se un file con lo stesso nome esiste già), secondo la convenzione di naming specificata nella Storyboard Bible di produzione (§3.3).

Listato 3.5: Calcolo dei metadati dello shot attivo e loro registrazione come testo di Stamp

```python
def setup_render_path_and_metadata(scene):
    if scene.name != MASTER_NAME:
        return
    [...]  # percorso di output con versioning automatico

    active_strip_name = "NO_STRIP"
    camera_name = "NO_CAM"
    current_f = scene.frame_current

    if scene.sequence_editor:
        # solo le strip di tipo SCENE: le strip audio vengono ignorate
        scene_strips = [s for s in scene.sequence_editor.strips
                        if s.type == 'SCENE']
        for s in scene_strips:
            if s.frame_final_start <= current_f < s.frame_final_end:
                active_strip_name = s.name.split('.')[0].upper()
                if s.scene and s.scene.camera:
                    camera_name = s.scene.camera.name.split('.')[0].upper()
                break

    left_text = f"{base_prefix}_{active_strip_name}"
    right_text = f"SQ{sq_num}_{camera_name}"
    current_width = get_visual_width(left_text) + get_visual_width(right_text)
    num_spaces = int(TARGET_WIDTH - current_width)
    if num_spaces < 1: num_spaces = 1
    padding = " " * num_spaces
    scene.render.stamp_note_text = f"{left_text}{padding}{right_text}"

bpy.app.handlers.render_init.clear()
bpy.app.handlers.render_init.append(setup_render_path_and_metadata)
[...]  # stessa registrazione su render_pre e frame_change_pre
```

La funzione individua la strip attiva confrontando il fotogramma corrente con l'intervallo di ciascuna strip di tipo `SCENE`, escludendo esplicitamente le altre strip presenti sulla stessa timeline. Tra queste vi sono le strip audio che portano nell'animatic gli effetti sonori previsti dallo storyboard (§2.1.1): esse vengono aggiunte manualmente sulla timeline di `MASTER_RENDER` e, grazie a questo filtro, non interferiscono con il calcolo dei metadati. La chiamata a `clear()` prima della

registrazione su ciascun handler evita invece che un'esecuzione ripetuta dello script registri la funzione più volte, moltiplicando il calcolo a ogni cambio di fotogramma.

3.11 Validazione dello script e casi di test

La pipeline è stata validata su un caso di produzione reale: la generazione dello storyboard e dell'animatic per la Scena 1 dell'episodio 106 (`ep106`) della serie *Slippets*. Il processo seguito ha ricalcato esattamente il flusso descritto nelle sezioni precedenti di questo capitolo: a partire dalla shotlist compilata per quella scena (§3.3), la sincronizzazione ha generato automaticamente gli shot corrispondenti (Scene, camere, istanze asset, §3.5); su questa base sono stati disegnati gli schizzi di storyboard tramite gli oggetti Grease Pencil parentati alle rispettive camere (§3.7, Figura 3.7); infine, per lo stesso set di shot, sono stati posizionati i pupazzi tridimensionali dei personaggi coinvolti, sfruttando la possibilità — motivata in apertura di questo capitolo e nel Capitolo 1 — di verificare direttamente in 3D gli ingombri e gli spazi di scena, un controllo che con lo storyboard bidimensionale richiederebbe un passaggio separato e successivo (Figura 3.8).

Shot	**Focale**	**Camera**	**Movimento**	**Target**
Shot0000	28	TOT1	Pan	None
Shot0010	28	TOT1	Statica	Koru
Shot0001	100	DETT0	Statica	Asset2D
Shot0020	50	FI1	Statica	Koru
Shot0030	50	FI2	Statica	Clio
Shot0040	100	DETT1	Statica	Asset2D
Shot0050	100	DETT2	Statica	Asset2D
Shot0060	85	PP1	Statica	Clio
Shot0070	100	DETT1	Statica	Asset2D
Shot0080	85	PP2	Statica	Koru
Shot0090	50	FI2	Pan	Clio
Shot0100	85	PP2	Statica	Koru
Shot0110	85	PP1	Statica	Clio
Shot0120	85	PP2	Statica	Koru
Shot0130	28	TOT1	Statica	Koru, Clio, CalzinoPuzzone
Shot0140	85	PP3	Statica	CalzinoPuzzone

Tabella 3.2: Shotlist compilata per ep106, Scena 1, prima della sincronizzazione. Le focali sono espresse in millimetri. Sono omesse, per ragioni di spazio, la colonna `Tipologia` — che esprime in forma estesa la stessa informazione codificata dal campo `Camera` — la colonna `Location`, che per questa scena vale `AreaCase` su tutti gli shot, e la colonna `Note`, che raccoglie le indicazioni di regia e le battute.



Figura 3.7: Render dell'animatic su Grease Pencil — Shot0080, ep106, Scena 1.



Figura 3.8: Render dell'animatic con pupazzi 3D — Shot0080, ep106, Scena 1.

Il caso ha comportato la simulazione dell'intera pipeline così come descritta nelle sezioni precedenti: ogni passaggio (sincronizzazione della shotlist, generazione di scene/camere/istanze, disegno dello storyboard, posizionamento dei pupazzi 3D) è stato eseguito nell'ordine reale previsto dal flusso, anziché verificato isolatamente passaggio per passaggio, per controllare la tenuta del sistema end-to-end. In particolare, tra gli esiti verificati vi è la corretta riuscita del burn-in dei metadati sull'animatic (§3.10): il testo sovrimpresso (identificativo shot, sequenza, camera) è stato controllato visivamente sul rendering finale per confermare che riportasse le informazioni corrette per ciascun fotogramma della Scena 1.

Questo caso ha permesso di verificare in particolare la corretta indipendenza delle animazioni tra shot diversi condividenti lo stesso personaggio (§3.6) e la coerenza della sequenza generata in `MASTER_RENDER` a fronte di aggiornamenti incrementali della shotlist.

Capitolo 4

Virtual Production e Set Extension in Unreal Engine 5.7

Il capitolo descrive la sperimentazione di Virtual Production condotta in Unreal Engine 5.7 per realizzare la Set Extension digitale del set fisico in miniatura, e il collegamento in tempo reale con DragonFrame che ne costituisce il nucleo tecnico.

4.1 Configurazione del progetto Unreal Engine 5.7

Il progetto di Virtual Production è stato sviluppato in Unreal Engine 5.7, su piattaforma macOS. La pipeline di rendering real-time si appoggia sui sistemi nativi del motore per l'illuminazione dinamica: Lumen per l'illuminazione globale e le riflessioni, e TSR (Temporal Super Resolution) per l'upscaling temporale dell'immagine. La convergenza di questi due sistemi — in particolare l'accumulo temporale di Lumen GI e la stabilizzazione di TSR/TAA su alcuni fotogrammi — richiede un intervallo di tempo non nullo prima che l'immagine catturata risulti pulita; questo vincolo, emerso durante i test di sincronizzazione con DragonFrame, è discusso nel dettaglio in §4.6 in relazione al Plate Capture.

L'architettura logica del progetto è centralizzata nel Level Blueprint del livello principale (`Main`), che funge sia da punto di ricezione della comunicazione OSC con DragonFrame (§4.4–§4.5) sia da orchestratore degli elementi di scena controllati in tempo reale: cambio camera per shot, avanzamento della Level Sequence, gestione del pass di chroma key. Questa scelta di concentrare la logica in un unico Blueprint

di livello, anziché distribuirla su più Actor Blueprint dedicati, è discussa più nel dettaglio in §4.5.



Figura 4.1: Il set "Area Case" in Unreal Engine 5.7.

4.2 Ingegnerizzazione dello shader feltro

Un primo tentativo di materiale feltro, precedente a quello poi adottato in via definitiva, è stato costruito come `M_MasterMaterial` a partire da un asset feltro nativo della libreria Epic Games, adottando lo *Shading Model Cloth* — pensato per la resa di tessuti — con un canale `Fuzz Color` dedicato alla micro-peluria lungo i bordi e un effetto di parallasse (*Parallax Occlusion Mapping*) per simulare un rilievo tridimensionale della trama. Questo approccio è stato abbandonato perché la resa visiva risultante non era convincente: l'aspetto ottenuto si avvicinava più a quello del velluto che a quello della lana infeltrita cercata per lo stile del diorama, portando ad adottare la soluzione più semplice descritta di seguito.

Il materiale destinato a simulare la resa visiva del feltro/lana sulle superfici del diorama virtuale (colline, nuvole, elementi vegetali) è `M_NewFeltro` (Figura 4.2), sviluppato dopo aver valutato un asset Fab (`M_C_Gray_Felt_3JeWme`) la cui resa visiva di base è risultata convincente come punto di partenza. Il materiale adotta lo *Shading Model Default Lit* del sistema di material classico di Unreal Engine, con un'architettura modulare assemblata tramite il nodo `MakeMaterialAttributes`, organizzata in quattro gruppi funzionali distinti nel grafo:

- **Color**: texture Albedo, con controlli parametrici di contrasto, saturazione e desaturazione, combinati con un colore di base (`Albedo Color`, `BaseBrightness`) tramite `Lerp`;
- **Specular**: texture Specular con controllo di contrasto dedicato;
- **Roughness**: texture Roughness, con range regolabile (`Roughness Min/Max`), contrasto e intensità, combinati tramite `Lerp` e `Saturate`;
- **Normals**: normal map appiattita tramite `FlattenNormal`, con parametro di intensità (`Normal Strength`).

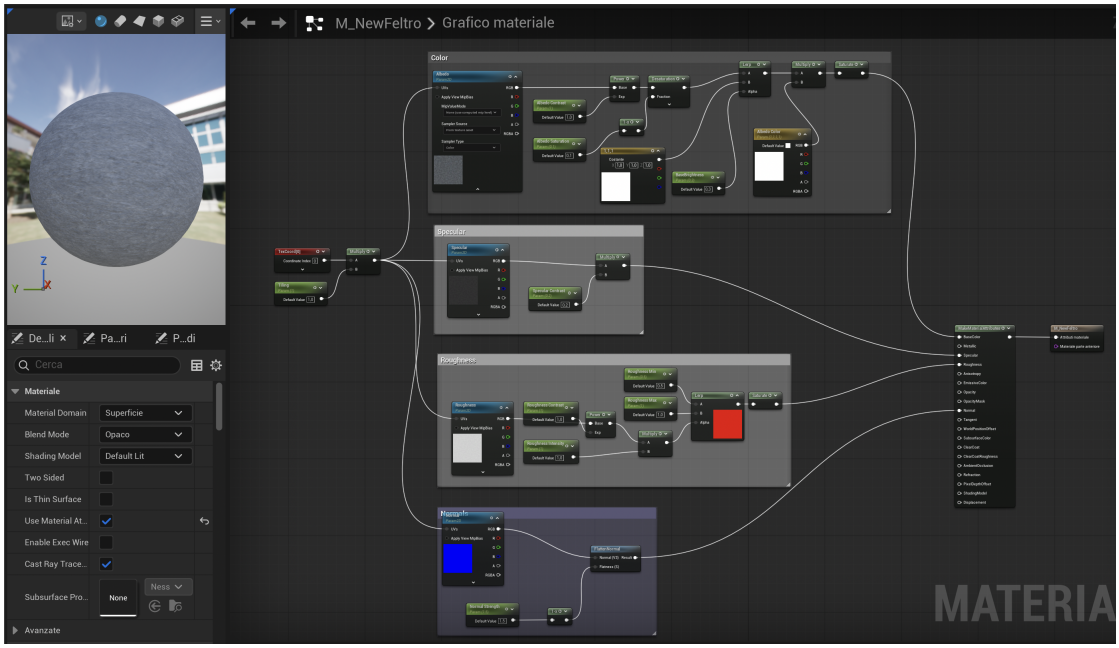


Figura 4.2: Grafo del materiale `M_NewFeltro`.

Un nodo `Tiling` (moltiplicatore applicato alle coordinate UV prima del campionamento delle texture) consente di controllare parametricamente la ripetizione della texture sulla superficie — funzionalità non disponibile nell'asset Fab di partenza nella forma richiesta dalla pipeline, e quindi aggiunta esplicitamente nella ricostruzione del materiale. Il valore di tiling è stato ridotto rispetto alla ripetizione di default dell'asset originale: un valore troppo alto, che produce un motivo di feltro a grana fine e densamente ripetuta, si è rivelato meno realistico proprio nel contesto d'uso specifico di questa pipeline — la texture non viene osservata direttamente a schermo, ma catturata da una fotocamera che riprende un monitor fisico (§4.7). In questa catena (texture → rendering su schermo → ripresa camera),

un dettaglio ad alta frequenza come una texture con tiling elevato tende a generare interferenze percettive simili a quelle già riscontrate con la griglia di pixel dello schermo, poi risolte con la sfocatura dell'immagine proiettata (§4.7): ridurre il tiling del materiale feltro ha quindi un ruolo complementare a quella soluzione, contribuendo a una resa complessivamente più pulita e meno soggetta ad artefatti nella stessa catena di acquisizione.

La resa cromatica del materiale è ottenuta partendo da una base di colore neutra (bianca) e sovrapponendo, per ciascun elemento di scena che richiede un colore specifico, il valore indicato nell'Asset Sheet di produzione — il documento di riferimento che specifica, per ciascun personaggio e per gli elementi di set, la palette cromatica dei materiali in "lana infeltrita" (ad esempio, per il personaggio Koru: corpo arancione chiaro, muso e interno orecchie bianco panna, fascia centrale muso arancione scuro) — garantendo così coerenza cromatica con le specifiche richieste. Il materiale `M_NewFeltro` funge da master; le istanze colorate per i singoli elementi di scena vengono derivate da esso secondo questa stessa logica.

4.3 Sviluppo dell'ambiente virtuale

Un primo tentativo di sfondo virtuale per l'estensione digitale del set fisico in miniatura era stato sviluppato a partire dall'asset skybox `QMS_Cartoon_Skybox_Free`; questo approccio è stato in seguito completamente sostituito da una soluzione costruita da zero, descritta di seguito, con lo stesso pattern di iterazione già visto per il materiale feltro (§4.2). In fase di integrazione con l'asset di libreria era comunque emerso che l'uso del materiale skybox come sostituto completo del sistema di cielo dinamico compromette la coerenza dell'illuminazione con Lumen; questo vincolo è stato mantenuto anche nella soluzione definitiva, che continua a tenere attivo il componente `SkyAtmosphere` nativo del motore per la gestione del cielo e della luce ambientale, limitando il materiale skybox custom alla sola resa delle nuvole, applicate con trasparenza sopra il cielo atmosferico nativo.

La soluzione definitiva è un materiale custom (`M_SimpleSkyDome`), costruito seguendo un tutorial di riferimento reperito online [18]. La texture del cielo utilizzata è stata reperita dal repository online `https://freestylized.com/all-skybox/`, selezionando l'immagine dallo stile più coerente con l'estetica del diorama; l'immagine è stata quindi sottoposta a upscaling $\times 2$ tramite gli strumenti Adobe e convertita dal formato PNG al formato HDR prima dell'importazione in Unreal Engine.

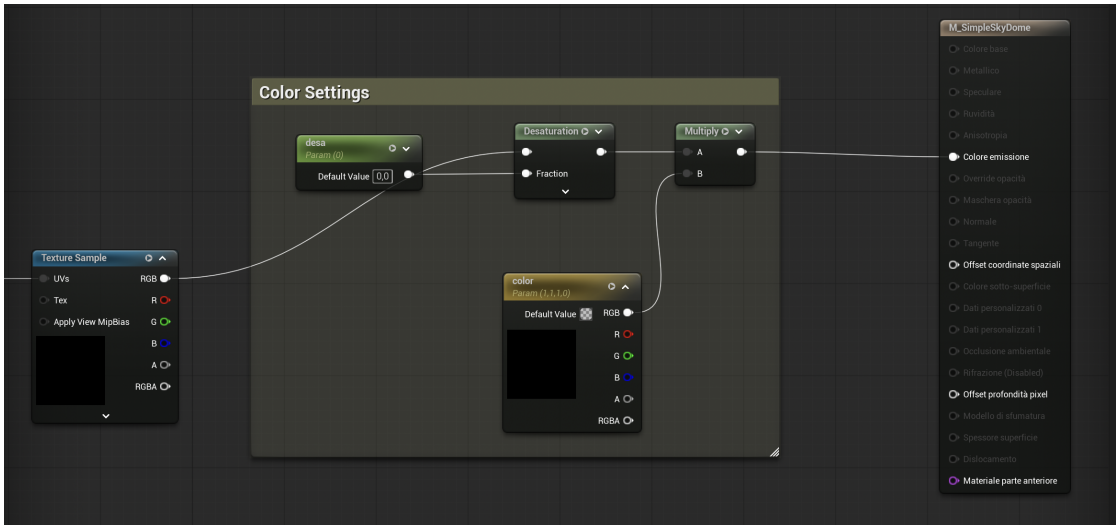


Figura 4.3: Materiale `M_SimpleSkyDome`: gestione desaturazione/tinta per il colore emissivo.

Il posizionamento e l'orientamento del cielo sono controllati tramite una catena di tre nodi `RotateAboutAxis` concatenati, uno per asse (Figura 4.4). Un aspetto tecnico rilevante emerso durante lo sviluppo è che l'output di ciascun nodo `RotateAboutAxis` rappresenta uno spostamento (*delta*) rispetto alla posizione in ingresso, e non la posizione finale assoluta: di conseguenza, ogni passaggio della catena richiede un nodo `Add` (che somma il delta alla posizione originale) seguito da una normalizzazione, prima di poter essere utilizzato come input per l'asse successivo o per il campionamento della texture.

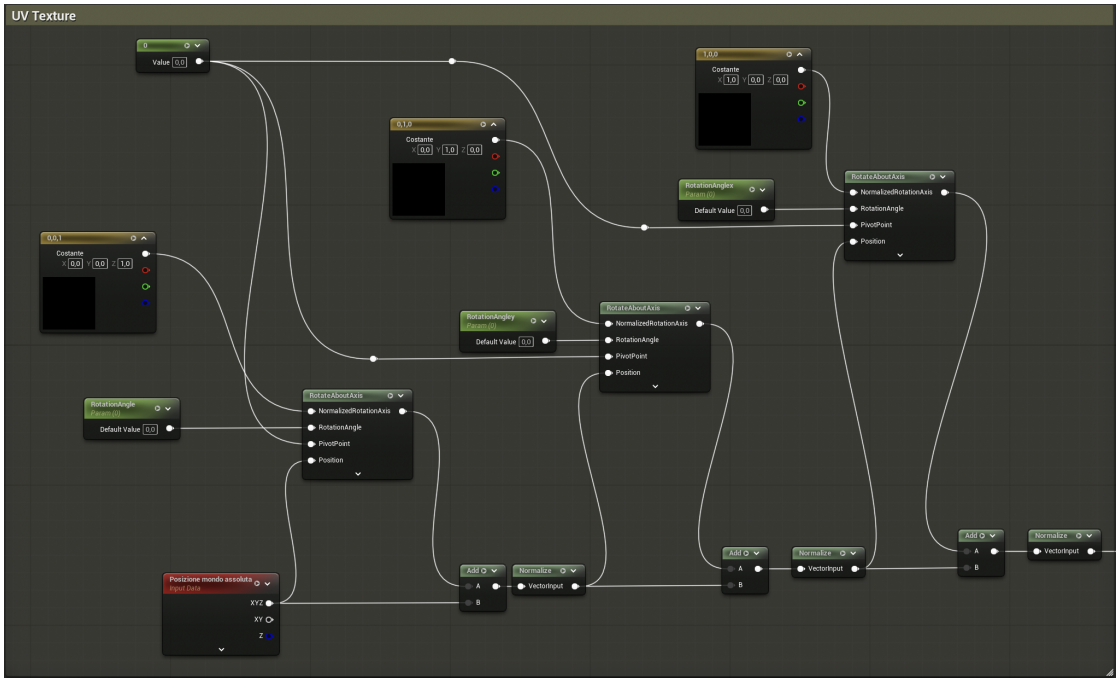


Figura 4.4: Materiale `M_SimpleSkyDome`: catena dei tre nodi `RotateAboutAxis` con `Add`+`Normalize` tra un asse e il successivo.

Per colmare la discontinuità visibile all'orizzonte — dove il pavimento a griglia nativo di Unreal Engine risulterebbe visibile oltre i limiti del diorama modellato — è stato valutato l'uso di un `Exponential Height Fog` con `Volumetric Fog` attivo, tecnica comune per dissolvere la linea dell'orizzonte in una leggera foschia. La prova ha però introdotto un artefatto visivo indesiderato: tra il cielo e l'orizzonte si generava un distacco netto, percepibile come una sottile linea bianca che separava in modo innaturale le due zone dell'inquadratura. La soluzione è stata quindi scartata, tenendo il pavimento a griglia semplicemente fuori dall'inquadratura delle camere di shot — un vincolo di composizione più che una soluzione tecnica generale.

4.4 Architettura di comunicazione OSC tra DragonFrame e Unreal Engine

Il collegamento in tempo reale tra il sistema di acquisizione DragonFrame e l'ambiente di Virtual Production in Unreal Engine è stato realizzato tramite il protocollo Open Sound Control (OSC), scelto per la sua nativa disponibilità sia in DragonFrame sia in Unreal Engine tramite il plugin OSC incluso nel motore. L'architettura adottata segue uno schema client-server: DragonFrame agisce da client, inviando

pacchetti UDP/OSC verso un server in ascolto lato Unreal Engine, attivato sulla porta 8000.

Il protocollo OSC è uno standard aperto nato in ambito audio professionale per lo scambio di dati in rete tra applicazioni, applicabile però a qualunque tipo di dato, non solo audio [12]. Il plugin OSC di Unreal Engine (attivabile da Edit → Plugins → Input Devices) espone questo protocollo come libreria Blueprint type-safe: un `OSC Server` (`UOSCServer`) funge da endpoint di ascolto per i messaggi indirizzati all'istanza del motore, con la possibilità di specificare quali indirizzi IP il server debba ascoltare; un `OSC Message` (`FOSCMessage`) è il pacchetto ricevuto, composto da un indirizzo — una struttura gerarchica nella forma `/Container1/Container2/Metodo`, analoga a un percorso di file — e da un payload di argomenti tipizzati (interi, float, stringhe) [12]. È su questa struttura di indirizzo e payload che si basa lo smistamento dei messaggi descritto in §4.5.

A differenza di un collegamento OSC nativo diretto, l'invio dei messaggi è mediato da uno script Python (`df_action.py`), configurato come *Action Script* nel pannello Preferenze → Avanzato di DragonFrame. Questa modalità fa sì che DragonFrame esegua lo script automaticamente ad ogni evento significativo del flusso di acquisizione (scatto, spostamento di posizione, salvataggio, cancellazione), passandogli come argomenti da riga di comando una tupla posizionale fissa: `Production, Scene, Take, Action, Frame, Exposure, ExpName, Filename`. L'ordine e il contenuto esatto di questi argomenti non sono stati assunti dalla documentazione ma verificati empiricamente, registrando in un file di log ogni invocazione dello script durante una sessione di test reale.

DragonFrame dispone anche di un proprio sistema di comunicazione OSC nativo (pannello Preferenze → Open Sound Control), configurabile tramite un modello di messaggio a template fisso — una stringa con placeholder (`[frame]`, `[scene]`, `[exposureName]`, ecc.) sostituiti automaticamente a ogni evento, senza possibilità di logica condizionale o di temporizzazione. La scelta di affidarsi invece a un Action Script eseguibile è motivata dalla necessità di una programmabilità che il template OSC nativo non offre: filtrare selettivamente solo l'evento `POSITION` tra tutti quelli generati da DragonFrame, comporre più messaggi OSC in sequenza a partire dagli stessi dati in ingresso, e soprattutto introdurre un ritardo temporale controllato (`time.sleep(3)`, §4.6) tra la ricezione dell'evento e la restituzione del controllo a DragonFrame — un'operazione priva di equivalente nel sistema a template fisso, che non ha alcun concetto di attesa.

Una volta adottato l'Action Script come meccanismo primario, ci si sarebbe aspettati che il campo "Evento Fotogramma di Posizione" del sistema a template fisso venisse svuotato, per evitare che i due meccanismi inviassero messaggi ridondanti verso lo stesso server in ascolto in Unreal Engine. In pratica, il campo (`/dragonframe/frame [frame]`) risulta ancora popolato nelle Preferenze, e un test di verifica — condotto mettendosi in ascolto sulla porta 8000 con uno script Python

indipendente sia da DragonFrame sia da Unreal Engine, così da osservare il traffico OSC grezzo senza alcuna mediazione — ha confermato che il template nativo genera effettivamente un secondo messaggio `/dragonframe/frame` per ogni scatto, con lo stesso numero di fotogramma e a pochi millisecondi di distanza da quello inviato da `df_action.py`. La duplicazione non compromette la sincronizzazione, poiché il valore ricevuto è identico e il secondo `Set Playback Position` in Unreal Engine (§4.5.3) è idempotente rispetto allo stesso numero di fotogramma; resta però una ridondanza di rete non necessaria, che si potrebbe eliminare svuotando quel campo.

Resta invece attivo, senza sovrapposizione di indirizzi con l'Action Script, un secondo canale nativo di DragonFrame: la funzione *Output posizioni motori* (stessa scheda Preferenze, sezione Controllo Movimento), che genera automaticamente un messaggio OSC all'indirizzo `/dragonframe/axis/<NOME_ASSE>` — con il nome dell'asse in maiuscolo, secondo la configurazione impostata nel pannello Arc Motion Control — ogni volta che il valore di un asse cambia. È questo secondo canale, e non l'Action Script, a generare i cinque indirizzi `/dragonframe/axis/PAN`, `TILT`, `X`, `Y`, `Z` gestiti dallo `Switch on String` e descritti in §4.5.6. Complessivamente, ogni indirizzo OSC ricevuto da Unreal Engine e gestito nel Level Blueprint proviene quindi da uno di due canali nativi di DragonFrame: l'Action Script `df_action.py` per fotogramma, nome dello shot ed esposizione (con la ridondanza appena descritta sul solo indirizzo frame); la funzione Output posizioni motori per gli assi di tracking.

Lo script filtra gli eventi in ingresso sul valore del campo `Action`: quando questo corrisponde a `"POSITION"` — l'evento generato da DragonFrame ad ogni spostamento su un nuovo fotogramma da catturare — vengono inviati, tramite la libreria `python-osc`, tre messaggi UDP verso `127.0.0.1:8000`:

Listato 4.1: Branch POSITION di df_action.py

```
if action == "POSITION":
    client.send_message("/dragonframe/frame", frame_number)
    client.send_message("/dragonframe/shotname", scene)
    client.send_message("/dragonframe/exposurename", exposure_name)

    if exposure_name == "X2":
        time.sleep(3)
```

La chiamata `time.sleep(3)` non è incondizionata sull'intero branch `POSITION`, ma subordinata al valore di `exposure_name`: viene eseguita solo quando l'esposizione in arrivo è `"X2"`, lasciando invece che DragonFrame proceda immediatamente allo scatto per l'esposizione `"X1"`, che non richiede alcun cambio di stato del Post Process. Questa scelta sfrutta il modello di esecuzione dell'Action Script: poiché DragonFrame invoca lo script come processo esterno e ne attende il completamento prima di procedere all'evento successivo, il ritardo introdotto in coda all'invio dei messaggi OSC — solo nel ramo X2 — blocca l'avanzamento di DragonFrame verso lo scatto reale (`SHOOT`) per l'intervallo di tempo necessario a Unreal Engine

per completare il cambio di stato del Post Process — il meccanismo tecnico alla base della soluzione discussa in dettaglio in §4.6. Il condizionamento su `exposure_name`, insieme alla scrittura di ogni invocazione dello script in un file di log (`/tmp/dragonframe_debug.log`, §4.6), è stato verificato ascoltando direttamente il traffico OSC in ingresso con uno script Python dedicato, indipendente sia da DragonFrame sia da Unreal Engine: durante uno scatto con doppia esposizione, il messaggio `/dragonframe/frame` per X1 e quello per X2 risultano infatti separati da meno di un secondo, mentre tra la fine del burst di X2 e l'inizio del fotogramma successivo intercorrono con precisione i 3 secondi della `sleep` — confermando che il ritardo si applica soltanto al ramo X2.

Il campo `Scene` veicola il nome completo dello shot secondo la convenzione adottata in tutta la pipeline (`Shot0000`, `Shot0010`, ecc., completato con zero non significativi fino a quattro cifre), condivisa con la shotlist Excel e con le Scene di Blender (§3.2). L'adozione di questa convenzione ha richiesto un intervento di configurazione non ovvio: il campo "Nome scena" di DragonFrame impone di default un limite di lunghezza (3 caratteri) insufficiente a contenere l'identificativo completo, ed è stato necessario innalzarlo esplicitamente dal pannello Preferenze → Nome per accogliere la convenzione a quattro cifre.

Il campo `ExpName` veicola invece il nome dell'esposizione corrente (`"X1"` o `"X2"`), reso disponibile da DragonFrame quando la camera è configurata con più di un'esposizione per fotogramma — funzionalità impostata dal pannello Camera Settings del workspace Cinematography, utilizzata in questa pipeline per implementare un pass dedicato di chroma key (§4.5).

4.5 Ricezione ed elaborazione dei messaggi OSC nel Level Blueprint `Main`

Prima di entrare nel dettaglio dei singoli gruppi funzionali del Blueprint, è utile ripercorrere la sequenza operativa end-to-end di una sessione di scatto, per collocare ciascun meccanismo nel flusso di lavoro reale in cui viene utilizzato. All'avvio di una sessione, la fotocamera fisica viene collegata al computer e la sua corretta lettura da parte di DragonFrame viene verificata; su Unreal Engine viene quindi avviata una finestra `New Editor Window (PIE)`, priva dell'interfaccia dell'editor e per questo proiettabile a schermo intero sul televisore che affianca il set fisico, mentre il viewport principale dell'editor resta sul computer per il controllo del sistema (§4.7.1). In base alla scena aperta in DragonFrame, il Blueprint attiva automaticamente la `CineCameraActor` corrispondente (§4.5.4); l'inquadratura può comunque essere corretta manualmente in tempo reale, spostando nel viewport la camera attiva, indipendentemente dal sistema di tracciamento assi. Ad ogni scatto effettuato in DragonFrame, la doppia esposizione configurata produce due catture

consecutive dello stesso fotogramma: la prima (`X1`) con lo sfondo virtuale normale, la seconda (`X2`) con il pass di chroma key verde attivo sul pupazzo (§4.5.5); entrambe vengono salvate automaticamente, tramite il meccanismo di Plate Capture (§4.6), in una sottocartella nominata secondo lo shot corrente, all'interno della directory di salvataggio del progetto Unreal Engine attivo. La Figura 4.5 riassume questo ciclo, affiancando a ciascun evento di acquisizione lo stato che esso produce nella scena virtuale.

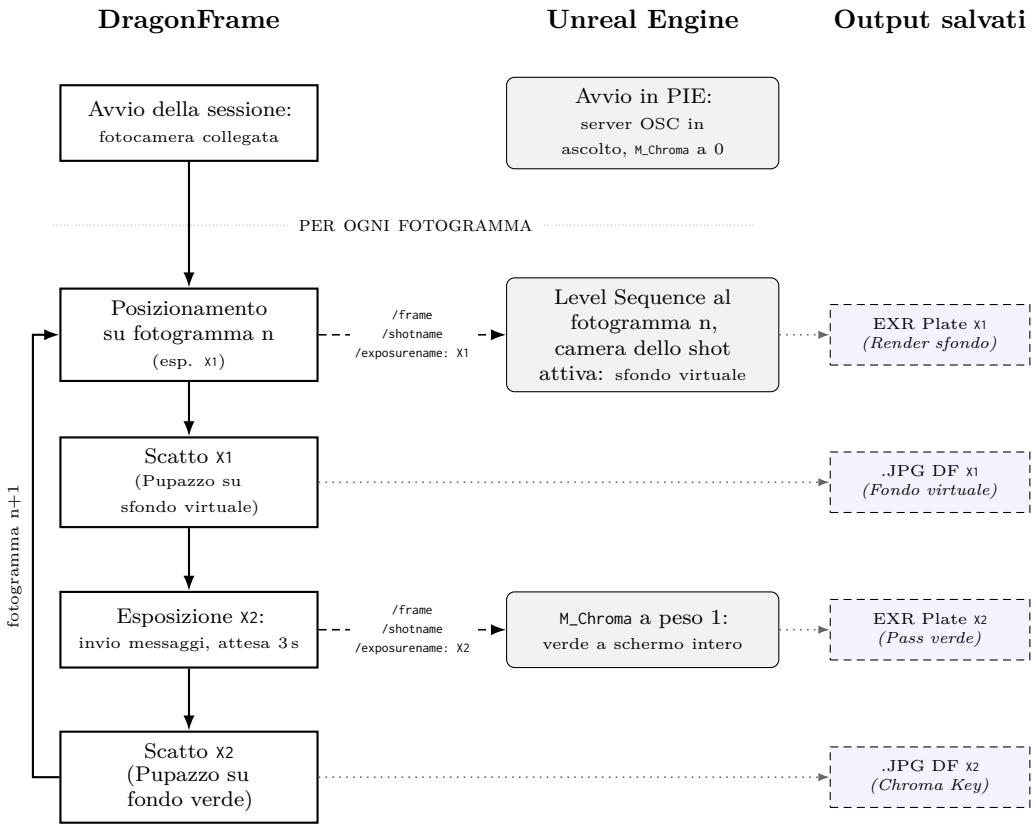


Figura 4.5: Ciclo operativo della sincronizzazione tra DragonFrame e Unreal Engine con dettaglio degli output generati ad ogni ciclo: a sinistra gli eventi di acquisizione, al centro lo stato della scena virtuale, a destra i file e le catture generate (`Plate` EXR da Unreal Engine e scatti fisici da DragonFrame).

4.5.1 Setup iniziale e debug

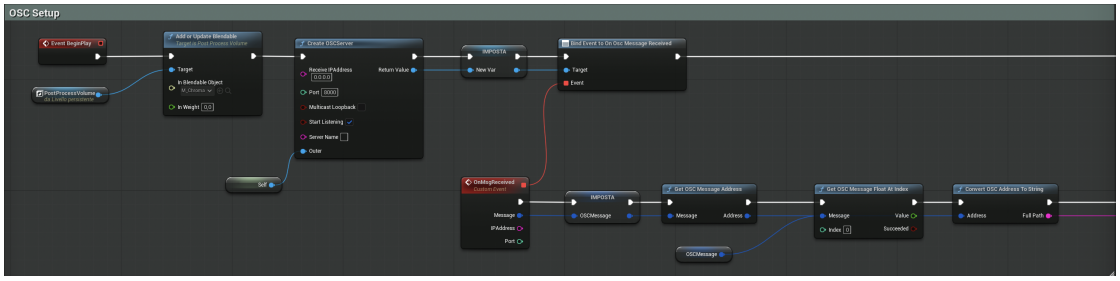


Figura 4.6: Setup iniziale e catena di debug nel Level Blueprint `Main`.

Questo primo gruppo (Figura 4.6) si attiva una sola volta, all'avvio della simulazione, e ha la funzione di aprire il canale di comunicazione con DragonFrame, di inizializzare lo stato del pass di chroma key e di predisporre uno strumento di verifica per il debug in fase di sviluppo.

All'evento `Event BeginPlay`, il Blueprint istanzia un server OSC tramite il nodo `Create OSC Server`: il nodo richiede un indirizzo IP di ricezione e una porta, e restituisce un riferimento al server a cui agganciare l'evento di ricezione. Un dettaglio configurativo rivelatosi critico durante il debug riguarda proprio l'indirizzo IP di ricezione: impostare `127.0.0.1` come *Receive IP* impedisce, su macOS, la corretta ricezione dei pacchetti, poiché il sistema operativo associa quell'indirizzo all'interfaccia di rete locale (LAN) anziché all'interfaccia di loopback; la diagnosi è stata condotta tramite il comando `lsof -i :8000`, e la soluzione consiste nell'impostare `0.0.0.0` come indirizzo di ricezione.

Nello stesso evento, prima che venga ricevuto qualsiasi messaggio, il materiale `M_Chroma` viene registrato nel `PostProcessVolume` tramite il nodo `Add Or Update Blendable` con peso iniziale pari a 0 (§4.6). Il materiale risulta così già presente nella pila di effetti ma privo di influenza sull'immagine, e la scena si apre con lo sfondo virtuale normale: la prima esposizione di ciascun fotogramma (`X1`) trova lo sfondo pronto senza richiedere alcun cambio di stato, e al ramo `/dragonframe/exposurename` (§4.5.5) resta il solo compito di aggiornarne il peso.

Al server viene agganciato, tramite `Bind Event to On Msg Received`, un evento personalizzato del Blueprint: ogni volta che arriva un messaggio, l'evento si attiva automaticamente, portando con sé il messaggio ricevuto (di tipo `FOSCMessage`, §4.4) come dato in uscita, che alimenta il gruppo di smistamento degli indirizzi (§4.5.2); durante la fase di sviluppo, lo stesso messaggio alimentava in parallelo anche la catena di debug descritta di seguito.

A supporto del debug, l'indirizzo estratto tramite `Get OSC Message Address` veniva convertito in stringa (`Convert OSC Address To String`) e concatenato al valore corrispondente tramite `Append`, per essere visualizzato a schermo e in log tramite

un nodo `Print String` esplicitamente marcato "Solo sviluppo" nell'editor. Questo meccanismo ha permesso di verificare in tempo reale, durante le sessioni di test, l'esatta corrispondenza tra i pacchetti inviati da DragonFrame e quelli ricevuti da Unreal Engine, complementando la validazione su file già descritta in §4.6; una volta confermato il corretto funzionamento del sistema, questa catena è stata rimossa dal Blueprint definitivo. Il debug del sistema si affida quindi agli strumenti nativi di Unreal Engine: i messaggi di un nodo `Print String` restano comunque visibili, senza alcuna modifica al livello di verbosità predefinito, nella finestra `Output Log` (Window → Output Log) sotto la categoria `LogBlueprintUserMessages`, mentre il ciclo di vita del server OSC (apertura e chiusura della connessione) è tracciato nella categoria nativa `LogOSC` del motore.

4.5.2 Smistamento degli indirizzi: `Switch on String`

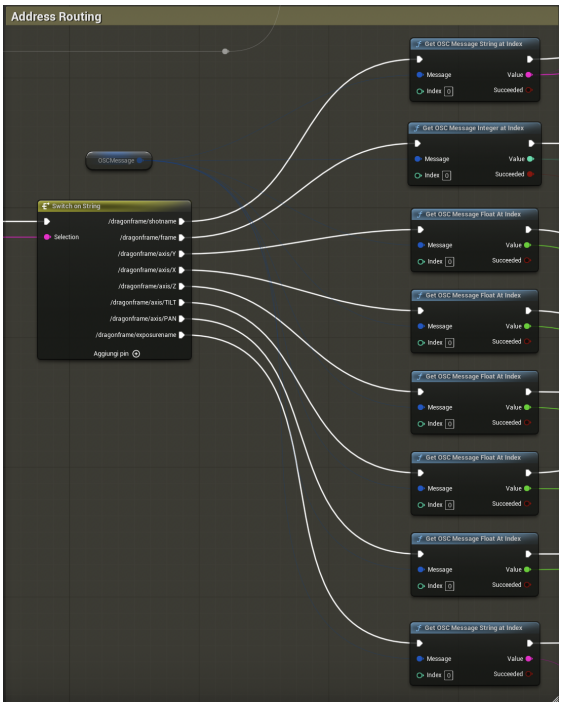


Figura 4.7: Nodo `Switch on String` con tutti gli indirizzi gestiti.

La funzione di questo gruppo (Figura 4.7) è instradare ogni messaggio in arrivo verso il gruppo funzionale competente, in base al suo indirizzo OSC. Il nodo `Switch on String` confronta una stringa in ingresso (qui, l'indirizzo del messaggio) con un elenco di valori possibili definiti dall'utente, ed esegue il ramo di esecuzione corrispondente al valore che coincide — l'equivalente visuale di un costrutto

`switch`/`case` nei linguaggi testuali. I rami gestiti sono quattro, ciascuno descritto in un gruppo funzionale a parte nel seguito della sezione: `/dragonframe/frame` (§4.5.3), `/dragonframe/shotname` (§4.5.4), `/dragonframe/exposurename` (§4.5.5) e il gruppo di indirizzi `/dragonframe/axis/*` (§4.5.6).

All'interno di ciascun ramo, il valore numerico o testuale associato al messaggio viene letto da due nodi ricorrenti in tutta la sezione: `Get OSC Message Float At Index` e `Get OSC Message String At Index`, che estraggono rispettivamente come numero decimale o come stringa l'argomento presente a una determinata posizione nel payload del messaggio OSC (§4.4); dove il valore è un numero intero (il fotogramma, §4.5.3), viene usato invece `Get OSC Message Integer At Index`.

4.5.3 Sincronizzazione del fotogramma

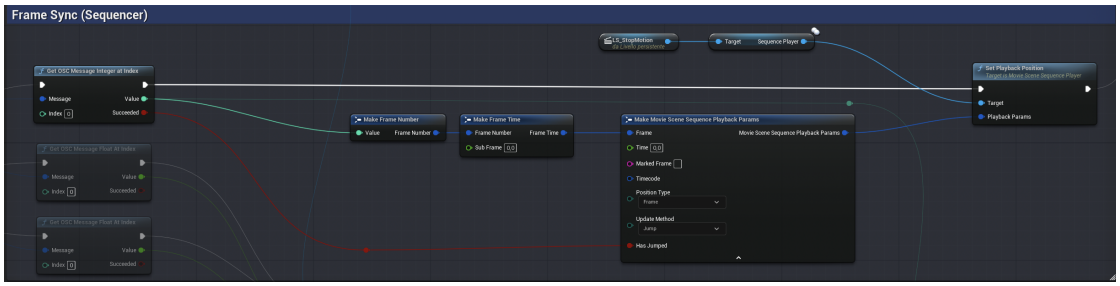


Figura 4.8: Ramo `/dragonframe/frame`: sincronizzazione del fotogramma con la Level Sequence.

Questo gruppo (Figura 4.8) ha la funzione di allineare la Level Sequence di Unreal Engine al fotogramma corrente di DragonFrame. Il valore numerico del fotogramma viene estratto e convertito in una struttura `Frame Time` (`Make Frame Number` → `Make Frame Time`), incapsulata in una struttura `Movie Scene Sequence Playback Params` con `Position Type` impostato su *Frame*, e infine applicata tramite `Set Playback Position` al `Sequence Player` della Level Sequence (`LS_StopMotion`) — funzione che sposta la posizione di riproduzione a un fotogramma specifico, con la possibilità di scegliere se il cambiamento debba essere interpolato nel tempo (`Play`) o avvenire istantaneamente (`Jump`). Per il test si è scelto di utilizzare la modalità *Jump* perché concettualmente coerente con l'obiettivo dichiarato in precedenza.

4.5.4 Cambio camera per shot

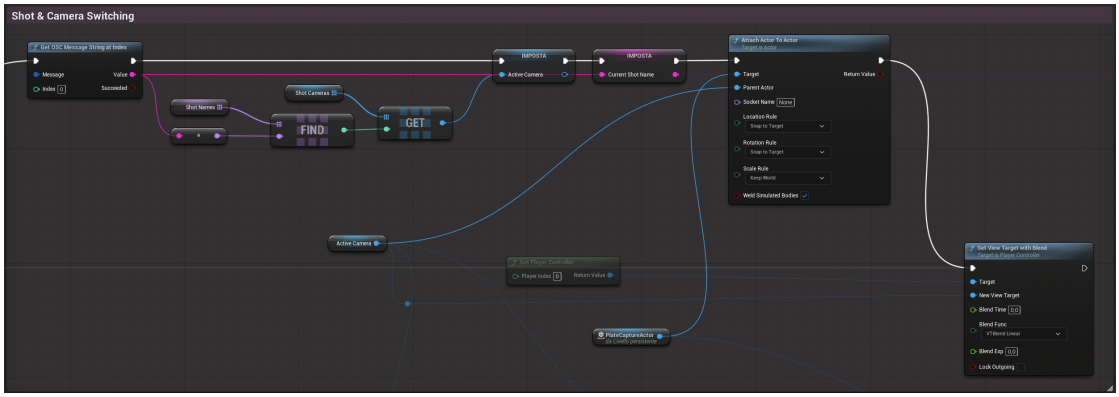


Figura 4.9: Ramo `/dragonframe/shotname`: cambio camera per shot.

Questo gruppo (Figura 4.9) ha la funzione di attivare la `CineCameraActor` corretta ad ogni cambio di shot. Il valore stringa dello shot viene estratto con `Get OSC Message String At Index` e confrontato, tramite `Find Index` — una funzione standard sugli array Blueprint che restituisce la posizione (indice) del primo elemento di un array che corrisponde al valore cercato, oppure -1 se non lo trova — con un array `ShotNames` popolato manualmente con gli identificativi di tutti gli shot del progetto; l'indice restituito viene quindi utilizzato per recuperare, dall'array parallelo `ShotCameras`, il riferimento alla `CineCameraActor` corrispondente, che viene impostata come camera attiva. L'uso di due array paralleli anziché di una struttura a mappa (`Map`) è una scelta deliberata, motivata dall'instabilità riscontrata nell'editor di Unreal Engine 5.7 nella gestione dei pin *wildcard* generici associati ai nodi Map durante l'editing del grafo.

Il cambio di shot comporta inoltre l'aggancio del `PlateCaptureActor` (§4.6) alla `CineCameraActor` appena attivata, tramite un nodo `Attach Actor To Actor` con `Location Rule` e `Rotation Rule` impostate su *Snap to Target* e `Scale Rule` su *Keep World*; il nome dello shot corrente viene inoltre salvato in una variabile dedicata (`CurrentShotName`), riutilizzata nella composizione del percorso di output del Plate Capture (§4.6).

4.5.5 Pass di chroma key a doppia esposizione

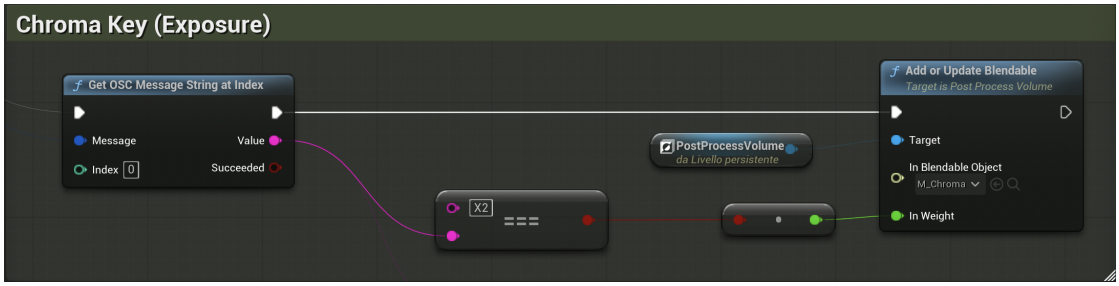


Figura 4.10: Ramo `/dragonframe/exposurename`: pass di chroma key.

Questo gruppo (Figura 4.10) ha la funzione di attivare il pass di chroma key esclusivamente sull'esposizione dedicata (`X2`), lasciando inalterata la resa normale della scena sull'esposizione principale (`X1`). Il valore stringa del nome esposizione viene confrontato con la costante `"X2"`; il risultato booleano del confronto pilota il nodo `Add Or Update Blendable`, che aggiunge — o aggiorna, se già presente — un materiale Post Process nella pila di effetti applicati da un `PostProcessVolume`, controllandone il peso di miscelazione (`In Weight`) rispetto alla resa normale della scena. Il meccanismo attuale, la calibrazione del colore Emissive e il problema di copertura che ha portato a questa soluzione sono descritti nel dettaglio in §4.6, insieme alla soluzione precedente (basata su `Set Visibility` su un piano `ChromaCard`) da cui questo meccanismo è derivato.



Figura 4.11: Vista operatore in DragonFrame: le due file di fotogrammi X1 (sfondo normale) e X2 (chroma key), catturate in sequenza per ogni scatto.

4.5.6 Tracciamento degli assi motion-control

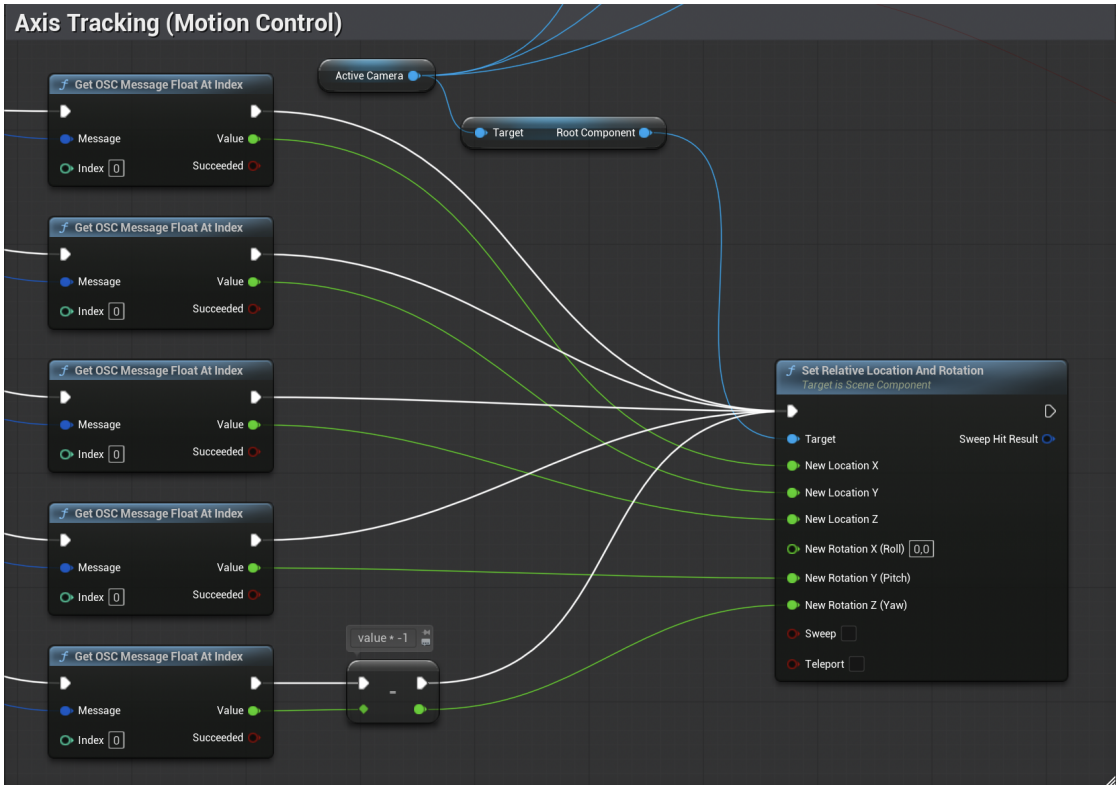


Figura 4.12: Rami `/dragonframe/axis/*`: tracciamento assi motion-control.

Questo gruppo ha la funzione di mantenere la telecamera virtuale sincronizzata in tempo reale con i movimenti del rig motion-control fisico (o, in assenza dell'hardware DMC-16/DMC+, con gli assi virtuali configurati in DragonFrame), così da preservare la coerenza di parallasse e prospettiva tra set fisico e set extension digitale descritta in §4.7. Cinque indirizzi dedicati (`/dragonframe/axis/PAN`, `TILT`, `X`, `Y`, `Z`) alimentano un nodo `Set Relative Location And Rotation`, che sposta rotazione e posizione della `CineCameraActor` attiva nella scena in base al valore ricevuto su ciascun asse.

Il collegamento tra ciascun indirizzo e il pin corrispondente non è una corrispondenza diretta uno-a-uno con la denominazione dell'asse, ed è stato verificato empiricamente inviando manualmente da terminale, tramite lo script Python di test già descritto in §4.6, valori di prova distinti su ciascun asse e osservando la risposta della camera nella Viewport di Unreal Engine con una shot camera attiva. Il ramo `PAN` passa per un nodo `Multiply` che inverte il segno del valore ricevuto prima di raggiungere `New Rotation Z (Yaw)`: con questa inversione, un valore positivo produce

una rotazione della camera verso sinistra e un valore negativo verso destra. I rami `X` e `Y` risultano invece scambiati rispetto alla convenzione di assi nativa di Unreal Engine: il valore ricevuto su `/dragonframe/axis/X` pilota `New Location Y` (movimento laterale — positivo verso sinistra, negativo verso destra), mentre il valore ricevuto su `/dragonframe/axis/Y` pilota `New Location X` (movimento in avanti/indietro rispetto all'inquadratura — positivo in avanti, negativo indietro). Questa corrispondenza incrociata è coerente con una correzione deliberata tra la convenzione di assi del rig (o degli assi virtuali configurati in DragonFrame) e quella nativa del mondo di Unreal Engine, dove è l'asse Y a governare lo spostamento laterale e l'asse X quello in profondità. I rami `TILT` e `Z`, verificati con lo stesso metodo, risultano invece collegati direttamente, senza alcun nodo di correzione intermedio: il valore ricevuto su `/dragonframe/axis/TILT` pilota `New Rotation Y (Pitch)` (positivo verso l'alto, negativo verso il basso) e quello su `/dragonframe/axis/Z` pilota direttamente `New Location Z` (positivo verso l'alto), in entrambi i casi coerentemente con la convenzione nativa di Unreal Engine, che non ha quindi richiesto alcuna correzione di segno o di asse per questi due canali.

Il meccanismo di collegamento OSC tra DragonFrame e la trasformazione della camera è stato costruito adattando un tutorial di riferimento reperito online [19], che illustra lo stesso principio per un sistema di Virtual Production con LED wall basato su nDisplay e Switchboard — non disponibile su macOS (§4.6) e quindi sostituito, nell'implementazione qui descritta, con una semplice `CineCameraActor` pilotata direttamente. Il tutorial di riferimento gestisce un insieme più ampio di assi, specifico del rig motorizzato dell'autore (tra cui un piatto girevole); la struttura qui implementata è stata invece limitata ai cinque assi effettivamente rilevanti per l'allestimento fisico di questa tesi, con l'architettura a `Switch on String` comunque predisposta ad accogliere in futuro ulteriori assi motorizzati, qualora si rendesse disponibile hardware aggiuntivo (§6.3).

Questo canale è stato validato, come indicato in §4.6, tramite simulazione software indipendente dall'hardware di tracking motorizzato, con esito positivo su tutti e cinque gli assi (PAN, TILT, X, Y, Z).

4.6 Validazione del sistema e sfide di sincronizzazione

La messa a punto del collegamento OSC ha richiesto un percorso di debug empirico, condotto secondo una metodologia comune a entrambi i lati del sistema: prima di cablare la logica di ricezione in Unreal, ogni assunzione sul formato e sull'ordine degli argomenti trasmessi da DragonFrame è stata verificata registrando su file (`/tmp/dragonframe_debug.log`) il contenuto grezzo di `sys.argv` ad ogni invocazione dello script, disaccoppiando così la validazione del formato dei dati dalla logica di

consumo lato Unreal. Lo stesso approccio ha permesso di confermare, ad esempio, che un singolo scatto con doppia esposizione genera due eventi `POSITION` distinti per lo stesso numero di fotogramma — uno per `X1` e uno per `X2` — condizione necessaria al corretto funzionamento del pass di chroma key descritto in §4.5.

Un secondo livello di validazione, indipendente dalla presenza dell'hardware di scatto fisico, è stato condotto tramite uno script di simulazione (`simulate_capture.py`), in grado di generare via terminale la stessa sequenza di messaggi OSC prodotta da un ciclo di acquisizione reale (cambio shot, avanzamento di fotogramma, doppia esposizione X1/X2), permettendo di verificare la risposta del Blueprint `Main` — cambio camera automatico e toggle del pannello chroma — indipendentemente dal collegamento con DragonFrame.

Per il ramo degli assi motion-control (§4.5.6), lo stesso approccio si è reso necessario per un motivo diverso: il pannello Arc Motion Control di DragonFrame risultava non utilizzabile in assenza di un dispositivo di controllo movimento — reale o virtuale — associato nel relativo dialog di connessione, rendendo di fatto impossibile generare messaggi sugli indirizzi `/dragonframe/axis/*` tramite il canale nativo Output posizioni motori durante le sessioni di test estive. La validazione dei cinque assi (PAN, TILT, X, Y, Z, §4.5.6) è stata quindi condotta simulando direttamente da terminale i messaggi OSC corrispondenti, con due script complementari: `df_send_test.py`, per l'invio di singoli valori statici utile a isolare la risposta di un asse alla volta, e `test_axis_osc.py`, che genera invece una rampa continua di valori crescenti e decrescenti su PAN e TILT in sequenza (con incrementi di 1 grado ogni 0,15 secondi), più rappresentativa del movimento graduale di uno slider trascinato a mano.

Listato 4.2: Rampa continua PAN/TILT in test_axis_osc.py

```
pan_max = 30.0
tilt_max = 15.0
passo = 1.0
pausa = 0.15  # secondi tra un invio e l'altro

valore = 0.0
while valore <= pan_max:
    pan_val = valore
    tilt_val = valore * (tilt_max / pan_max)  # TILT proporzionale a PAN

    client.send_message("/dragonframe/axis/PAN", pan_val)
    client.send_message("/dragonframe/axis/TILT", tilt_val)

    time.sleep(pausa)
    valore += passo
```

La fase mostrata sopra fa crescere PAN e TILT insieme fino al valore massimo; una seconda fase simmetrica (omessa qui per brevità) riporta entrambi i valori a

zero con lo stesso incremento, così da simulare un movimento di andata e ritorno completo dello slider.

Un collegamento reale del canale Output posizioni motori, con un dispositivo di controllo movimento — reale o virtuale — effettivamente associato in DragonFrame, rimane parte degli sviluppi futuri (§6.3).

Il sistema, nel suo complesso, risulta validato end-to-end in modalità *Play in Viewport* (la modalità di Play in Editor utilizzata durante lo sviluppo e il debug): il cambio automatico di camera per shot e la sincronizzazione del pass chroma key con la doppia esposizione DragonFrame rispondono correttamente sia alla simulazione software sia a un'acquisizione di test condotta direttamente da DragonFrame con l'Action Script attivo.

Una limitazione emersa durante i test riguardava la copertura fisica del pannello `ChromaCard`: essendo un piano di estensione fissa posizionato staticamente nel mondo virtuale — scelta necessaria poiché il pannello rappresenta un oggetto fisico reale presente sul set, che non deve seguire i movimenti della camera — la sua proiezione nell'inquadratura non copriva l'intero fotogramma per tutte le configurazioni di shot, in particolare per inquadrature più larghe o distanti.

Il problema è stato risolto sostituendo il meccanismo di toggle basato sulla visibilità del pannello fisico con un materiale Post Process a schermo intero, che garantisce una copertura verde del 100% del fotogramma indipendentemente da inquadratura, distanza o angolazione dello shot. È stato creato un materiale dedicato (`M_Chroma`), con dominio impostato su *Post Process* e colore Emissive verde, applicato a un attore `PostProcessVolume` a estensione infinita presente nel livello persistente.

Il verde scelto per il colore Emissive è leggermente più scuro rispetto a un verde puro/brillante alla massima intensità (che sarebbe la scelta più immediata per un pass di chroma key): un verde troppo luminoso va incontro a clipping sul sensore della fotocamera reale, apparendo paradossalmente più slavato e meno saturo nello scatto, con conseguente difficoltà a sostituirlo in modo pulito con uno sfondo in post-produzione. Riducendo il valore del colore Emissive a 0,6 si ottiene invece un'esposizione corretta e un verde pieno, più facilmente distinguibile dagli altri colori della scena (Figura 4.13).



(a) Prima: verde alla massima intensità, sovraesposto



(b) Dopo: verde calibrato a (0; 0,6; 0), correttamente esposto

Figura 4.13: Confronto prima/dopo la calibrazione del colore Emissive di `M_Chroma`: in (a) il verde alla massima intensità appare slavato per la sovraesposizione del sensore; in (b) il valore ridotto a 0,6 restituisce un verde pieno e correttamente esposto.

Il materiale viene registrato nel volume all'evento `Event BeginPlay`, tramite il nodo `Add Or Update Blendable` con `M_Chroma` come `In Blendable Object` e peso iniziale nullo (§4.5.1). Lo stesso nodo, richiamato dal ramo `/dragonframe/exposurename` (§4.5.5), ne aggiorna poi il peso (`In Weight`) a ogni esposizione: quando il valore ricevuto corrisponde a `"X2"`, il peso viene impostato a 1.0, rendendo il materiale pienamente visibile in sovrimpressione sull'intera immagine; in caso contrario il peso torna a 0.0, ripristinando la normale resa della scena. Questo meccanismo è stato validato end-to-end e ha sostituito integralmente il pannello `ChromaCard`, rimosso dalla scena una volta verificato il corretto funzionamento della soluzione Post Process.

Durante gli esperimenti è emerso un problema di sincronizzazione temporale: senza un intervento correttivo, il toggle del materiale `M_Chroma` non aveva il tempo materiale di completarsi — cioè di sostituire visivamente la scena con il verde pieno dietro al personaggio — prima che DragonFrame effettuasse lo scatto reale dell'esposizione X2, producendo una cattura in cui il verde non era ancora (o non era del tutto) presente in campo. La soluzione adottata è stata l'introduzione di una chiamata `time.sleep(3)` in `df_action.py` (§4.4), che introduce un ritardo di 3 secondi tra l'invio dei messaggi OSC del cambio di stato e la restituzione del controllo a DragonFrame, dando così a Unreal Engine il tempo necessario per completare il cambio di stato del Post Process prima che DragonFrame proceda con lo scatto reale del fotogramma.

Plate Capture e convergenza di Lumen/TSR. Il ritardo di convergenza di Lumen e TSR anticipato in §4.1 riguarda in particolare il meccanismo di *Plate Capture*: un attore dedicato (`PlateCaptureActor`), agganciato alla camera attiva tramite `Set View Target with Blend` applicato al Player Controller, cattura il fotogramma corrente in un render target dedicato (`RT_Plate`, risoluzione 1920×1080, formato FloatRGBA) tramite un componente `Scene Capture Component 2D` e il relativo nodo `Capture Scene`. L'allineamento tra la vista catturata e quella della camera dello shot corrente non si limita al posizionamento dell'attore: il campo visivo e le impostazioni di Post Process della camera attiva, ottenuti tramite `Get Camera View` e successivamente scomposti con `Break Minimal View Info`, vengono estratti (`Field Of View` e `Post Process Settings`) e riapplicati esplicitamente allo `Scene Capture Component 2D`, garantendo che il render catturato rifletta fedelmente i parametri ottici impostati sulla `CineCameraActor` dello shot.

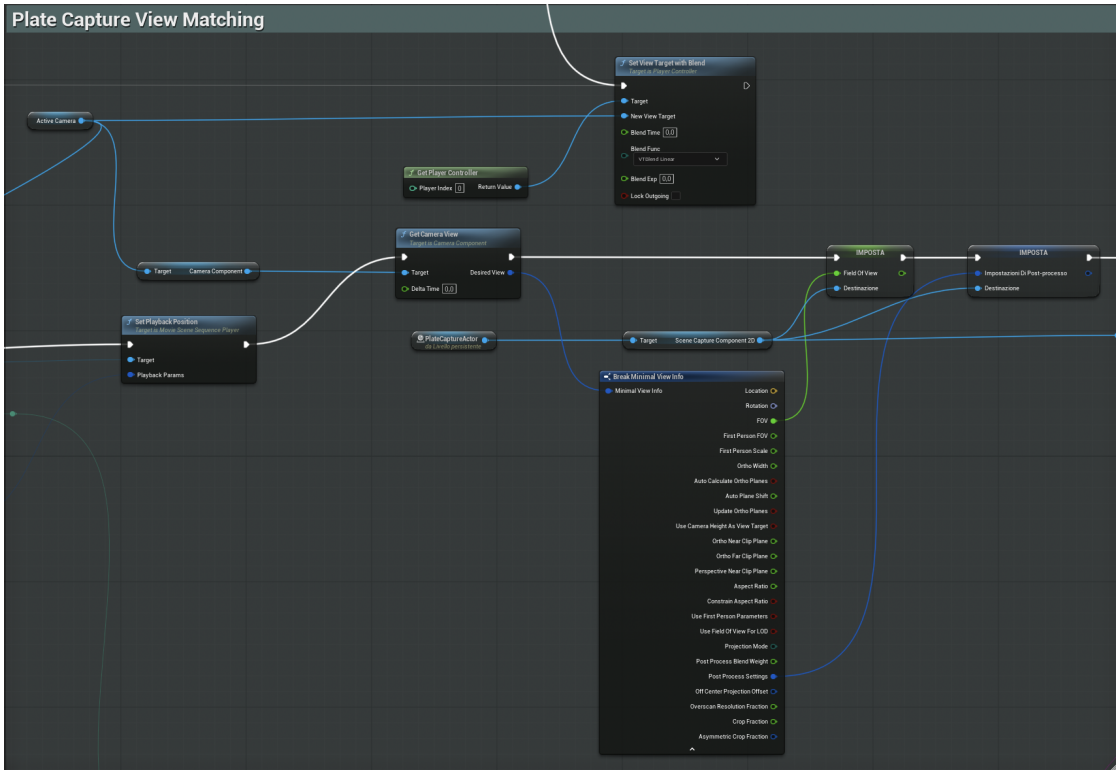


Figura 4.14: Plate Capture: allineamento ottico (FOV e Post Process).

Il render target viene poi scritto su disco tramite `Export Render Target`. Il percorso di destinazione è composto dinamicamente a partire dalla directory di salvataggio del progetto (`Get Project Saved Directory`), a cui viene concatenata tramite `Format Text` una sottocartella dedicata per shot (`{Dir}Plates/{Shot}/`), garantita a runtime da `Make Directory`; il nome del singolo file include sia il numero di fotogramma, con zero-padding a cinque cifre ottenuto tramite `Left Pad`, sia il nome dell'esposizione corrente (`{Frame}_{Exposure}.EXR`), così da preservare nella struttura dei file la stessa distinzione X1/X2 già utilizzata per il pass di chroma key (§4.5). È precisamente in questo punto della catena — tra l'avanzamento della Level Sequence (`Set Playback Position`, §4.5) e l'effettiva chiamata a `Capture Scene` — che il tempo di convergenza di Lumen GI e TSR deve essere rispettato: catturare il render target troppo presto dopo il cambio di fotogramma produce un'immagine ancora "rumorosa" (non convergente), non rappresentativa della resa finale dello sfondo virtuale.

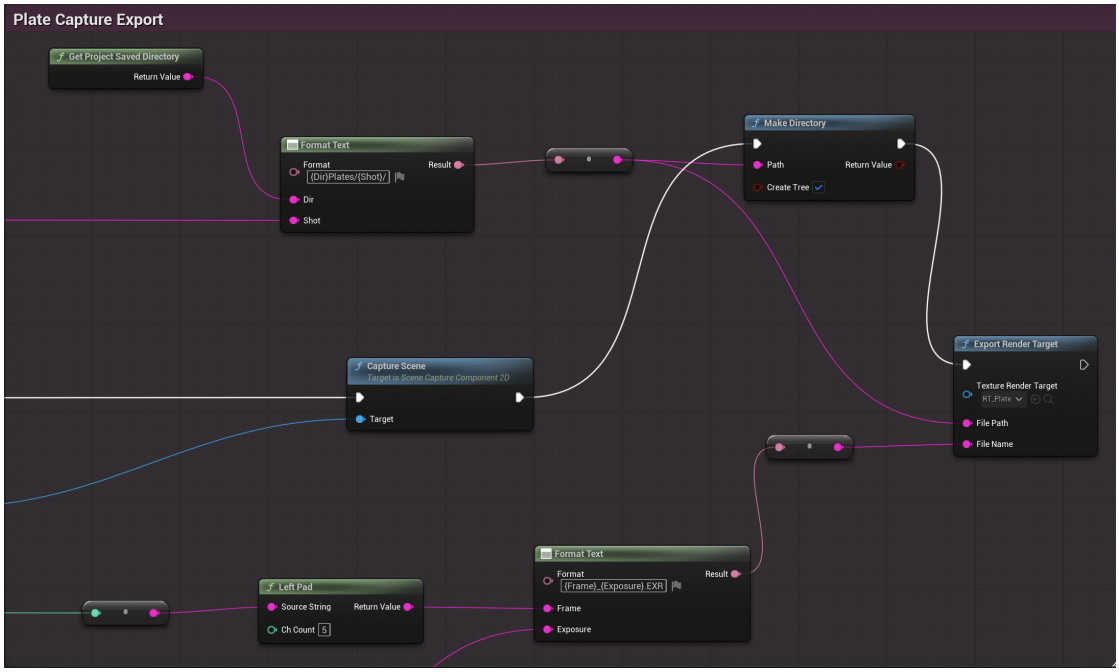


Figura 4.15: Plate Capture: composizione del percorso e cattura del render.

Il tracking degli assi Pan/Tilt è stato validato tramite simulazione software — uno script di test ha riprodotto la sequenza di messaggi OSC attesa dal movimento degli assi, confermando la corretta risposta del sistema lato Unreal — ma non è stato possibile estendere questa validazione al collegamento con l'hardware motorizzato reale (DMC-16/DMC+), poiché il modulo di controllo motion-control per stop motion non era disponibile in laboratorio al momento dei test. Resta infine da verificare, come sviluppo futuro (§6.3), il funzionamento end-to-end in presenza dell'effettivo hardware di tracking motorizzato degli assi, non ancora testato al momento della stesura di questa sezione.

4.7 Verifica spaziale ed ergonomica della Set Extension

4.7.1 Sperimentazione ergonomica e spaziale

Il test è stato condotto con il seguente allestimento fisico(Figura 4.16): lo sfondo virtuale, generato in tempo reale da Unreal Engine in una finestra *New Editor Window (PIE)* — priva dell'interfaccia dell'editor, e per questo proiettabile a schermo intero — è stato proiettato su uno schermo televisivo montato su un supporto mobile a rotelle, posizionato di fronte al set fisico in miniatura — un

pannello ricoperto di muschio ed erba sintetica, con elementi di scenografia pratici (rocce, piccola vegetazione) e il pupazzo del personaggio, appoggiato su due cavalletti da pittore in legno pieghevoli. Il viewport principale dell'editor, con tutti i controlli, restava sullo schermo del computer, così da mantenere il controllo del sistema senza interferire con quanto proiettato sul televisore; durante la sessione, l'inquadratura poteva inoltre essere corretta manualmente in tempo reale spostando direttamente nel viewport la `CineCameraActor` attiva in quel momento, indipendentemente dal sistema di tracciamento assi OSC (§4.5.6).



Figura 4.16: Allestimento fisico del test spaziale.

La distanza tra lo schermo e il set non è stata misurata con uno strumento di misura diretta, ma stimata per confronto con gli elementi di arredo presenti nell'allestimento, risultando dell'ordine di 0,5–1 m; il valore va quindi inteso come indicazione di scala, non come misura rilevata sul set. La ripresa è stata effettuata con una fotocamera Sony Alpha 7 III su treppiede, con un obiettivo manuale a innesto Leica R montato tramite adattatore (PHOLSY LR-E); la focale scelta è stata di 50 mm, non per una preferenza compositiva ma per un vincolo pratico

dell'attrezzatura disponibile in laboratorio al momento del test. Le impostazioni erano coerenti con una cattura fotogramma-per-fotogramma (ISO 200, tempi di posa dell'ordine di 1/25 di secondo, bilanciamento del bianco impostato per temperatura colore a 5600 K, lieve compensazione dell'esposizione negativa). L'illuminazione della scena combinava una fonte LED continua bicolore come luce principale (Nanlite Forza 72, impostata a circa il 28% di intensità e 6100 K di temperatura colore — coerente con la resa diurna dello sfondo virtuale) con modificatore octabox, e una barra LED lineare come luce di riempimento laterale; la luce emessa dallo schermo stesso, che proietta lo sfondo virtuale, contribuiva inoltre come ulteriore fonte di luce ambiente sulla scena. Le lanterne di carta sferiche visibili nelle foto dell'allestimento erano presenti come elemento di scena ma non erano accese durante questo test, e non hanno quindi contribuito all'illuminazione. Il bilanciamento del bianco impostato in camera (5600 K) non coincide esattamente con la temperatura colore della fonte LED principale (6100 K): la discrepanza andrebbe verificata in una sessione futura, poiché potrebbe introdurre una lieve dominante cromatica nella ripresa reale non presente nella resa dello sfondo virtuale.

Per riferimento rapido, l'attrezzatura e le impostazioni utilizzate per questo test sono riassunte di seguito:

- **Set fisico:** pannello ricoperto di muschio ed erba sintetica; elementi di scenografia pratici (rocce, piccola vegetazione); pupazzo del personaggio
- **Schermo di proiezione:** televisore Hisense 70 pollici, su supporto mobile a rotelle
- **Fotocamera:** Sony Alpha 7 III; obiettivo manuale a innesto Leica R, 50 mm, montato tramite adattatore PHOLSY LR-E; su treppiede
- **Impostazioni camera:** ISO 200; tempi di posa dell'ordine di 1/25 s; bilanciamento del bianco a 5600 K; lieve compensazione dell'esposizione negativa
- **Illuminazione principale:** Nanlite Forza 72 (LED continua bicolore), ∼28% di intensità, 6100 K, con modificatore octabox
- **Illuminazione di riempimento:** barra LED lineare (luce laterale); contributo aggiuntivo dalla luce emessa dallo schermo di proiezione stesso

Il test condotto ha avuto natura prevalentemente visiva, più che di misurazione ergonomica sistematica: non sono state raccolte misure comparative dello spazio operativo per le mani dell'animatore, ma è stata verificata la resa dello schermo di proiezione a distanza ravvicinata dal set fisico. Da questa verifica è emerso che, alla distanza operativa necessaria, la griglia dei pixel e le linee di giunzione dello schermo risultavano visibili a fotocamera — un artefatto che comprometteva la

continuità percettiva tra sfondo virtuale e set fisico. La soluzione adottata non agisce sul rendering virtuale, ma sulla ripresa fotografica reale: lato Unreal Engine, la camera virtuale è stata impostata senza alcuna limitazione di messa a fuoco, così che l'intero sfondo digitale venga renderizzato completamente nitido; la sfocatura (*defocus*) che elimina la visibilità della trama di pixel e delle linee dello schermo è ottenuta invece otticamente, sfruttando la profondità di campo ridotta della fotocamera reale, che mantiene a fuoco il pupazzo in primo piano lasciando lo schermo — più distante — naturalmente sfocato (Figura 4.17). Questa separazione di responsabilità evita che le due sfocature si sommino o entrino in conflitto, e lascia il controllo della profondità di campo interamente alla fotocamera fisica, coerentemente con una pipeline in cui è la ripresa reale a dettare i parametri ottici finali. La stessa considerazione — il dettaglio ad alta frequenza non legge bene nella catena texture-schermo-camera propria di questa pipeline — ha motivato anche la riduzione del valore di tiling del materiale feltro (§4.2), un intervento complementare alla sfocatura ottica piuttosto che una soluzione indipendente.



(a) Prima: pixel e linee dello schermo visibili



(b) Dopo: sfocatura ottica applicata

Figura 4.17: Confronto prima/dopo la sfocatura ottica: in (a) sono visibili la griglia dei pixel e le linee di giunzione dello schermo; in (b) la stessa inquadratura con la profondità di campo della fotocamera reale a eliminare l'artefatto.

Resta da eseguire, come sviluppo successivo, una verifica più sistematica dello spazio operativo per le mani dell'animatore, con misure comparative a diverse distanze schermo-set.

4.7.2 Accessibilità dell'animatore

Una verifica sistematica dell'ingombro fisico e dello spazio operativo necessario per l'accesso manuale al pupazzo — con gli strumenti di animazione tipici (pinzette, chiavi per i rig) — in presenza dello schermo di proiezione non è stata condotta in questa fase del lavoro: il pupazzo utilizzato per il test non era ancora dotato di un rig di animazione, rendendo non applicabile in questa fase una verifica che presuppone proprio la manipolazione degli strumenti tipici di rigging. Il punto resta quindi un obiettivo di sviluppo futuro, discusso in §6.3, condizionato al completamento della fase di rigging del personaggio.

Due osservazioni qualitative, emerse tuttavia nel corso del test descritto in §4.7.1 (condotto per altri scopi), forniscono un'indicazione preliminare non sistematica: l'assenza di un rig di tracking motorizzato collegato al pupazzo — dovuta all'indisponibilità dell'hardware DMC-16/DMC+ in laboratorio, §4.6 — ha lasciato libero lo spazio attorno al soggetto; la distanza tra schermo e set, stimata dell'ordine di 0,5–1 m e scelta per mantenere lo sfondo sfocato e il fuoco della camera concentrato sul pupazzo (§4.7.1), rende inoltre plausibile che l'animatore non incontrerebbe particolari difficoltà con lo spazio circostante anche in condizioni operative reali. Queste condizioni favorevoli andranno riverificate in un test dedicato, una volta disponibile un pupazzo rigged, specialmente in presenza di un rig di tracking motorizzato reale, il cui ingombro fisico — assente in questo test — tornerebbe rilevante.

Capitolo 5

Analisi dei risultati e valutazione della pipeline

Il capitolo valuta la pipeline sviluppata nei Capitoli 3 e 4, distinguendo i risultati funzionali effettivamente verificati dalle valutazioni di efficienza che, per la natura esplorativa del lavoro, restano di carattere indicativo.

5.1 Metodologia di valutazione

La valutazione condotta in questa tesi opera su due piani distinti, che è opportuno tenere separati per non attribuire ai risultati una portata che non hanno.

Il primo piano è quello della *verifica funzionale*: stabilire se i meccanismi progettati facciano effettivamente ciò per cui sono stati costruiti. Su questo piano la verifica è stata condotta tramite test diretti ed è conclusiva nei limiti dello scenario provato — validazione end-to-end in modalità *Play in Viewport*, acquisizioni di test con DragonFrame, esecuzione dello script di simulazione `simulate_capture.py` (§4.6), e verifica su un caso di produzione reale per la pipeline di pre-produzione (§3.11). Come precisato in §1.3, il grado di validazione non è però uniforme tra i due fronti: gli strumenti di pre-produzione sono stati impiegati nella lavorazione effettiva di un episodio, mentre la sperimentazione di Virtual Production si è svolta su un allestimento parziale, privo del rig di animazione del pupazzo e di parte della scenografia fisica. I risultati di questo piano sono raccolti in §5.2.

Il secondo piano è quello della *valutazione di efficienza*: stabilire *quanto* la pipeline digitalizzata convenga rispetto a quella tradizionale, in tempo o in costo. Su questo piano non è stato applicato un protocollo di misurazione comparativa controllata: non sono state cronometrate esecuzioni manuali equivalenti da contrapporre a quelle automatizzate, né ripetute misure su più casi.

Le considerazioni di §5.3 e §5.5 si fondano quindi su due basi diverse e dichiarate: da un lato un'analisi *strutturale* del lavoro eliminato — ossia il conteggio delle operazioni manuali che la pipeline automatizzata sostituisce, ricavabile dalla logica stessa degli script documentati nel Capitolo 3 e indipendente da una misura sul campo — dall'altro singole osservazioni temporali puntuali emerse durante lo sviluppo, che hanno valore indicativo e non statistico.

L'obiettivo primario del lavoro è stato infatti verificare la fattibilità tecnica dei due assi sperimentali, non quantificarne il rendimento comparato; una raccolta dati sistematica sul secondo piano resta indicata come sviluppo futuro (§6.3).

5.2 Risultati funzionali verificati

Sul piano della verifica funzionale, la pipeline ha prodotto risultati verificati su entrambi gli assi sperimentali.

Indipendenza delle animazioni tra shot. Il meccanismo che assegna a ciascuna armatura derivata da Library Override una Action dedicata e distinta (`ACT_<NomeAsset>_<ShotID>`, §3.5) è stato verificato esplicitamente nel caso di test (§3.11): shot diversi che condividono lo stesso personaggio mantengono animazioni separate, pur restando collegati alla medesima libreria asset condivisa. Questo è il requisito che rende praticabile l'uso del linking anziché della copia locale: senza di esso, il vantaggio di propagazione automatica degli aggiornamenti dalla libreria (§3.6) sarebbe stato inutilizzabile in una pipeline seriale, dove lo stesso personaggio ricorre in decine di shot.

Sincronizzazione DragonFrame–Unreal Engine. Il collegamento OSC è stato validato end-to-end (§4.6): a ogni scatto, il cambio automatico di camera in base allo shot aperto in DragonFrame, l'avanzamento della Level Sequence al fotogramma corrente e l'attivazione del pass di chroma key sulla sola esposizione dedicata rispondono correttamente, sia sotto simulazione software sia sotto acquisizione reale con l'Action Script attivo. È il risultato che risponde direttamente allo spazio di ricerca individuato dalla letteratura di riferimento (§2.3).

Robustezza del pass di chroma key. La soluzione finale adottata per il chroma key — un materiale Post Process applicato a un volume a estensione infinita — presenta un vantaggio non incrementale ma categorico rispetto alla soluzione inizialmente tentata: mentre il pannello `ChromaCard`, essendo un piano di estensione finita posizionato staticamente nel mondo virtuale, falliva la copertura del fotogramma per le inquadrature più ampie o distanti, il materiale Post Process garantisce per costruzione una copertura totale indipendentemente da inquadratura, distanza e angolazione (§4.6). Il miglioramento non è quindi una questione di grado misurabile, ma l'eliminazione di un'intera classe di casi di fallimento.

Tracciamento degli assi motion-control. I cinque assi gestiti (PAN, TILT, X, Y, Z) rispondono correttamente ai messaggi OSC ricevuti, con la mappatura verificata empiricamente asse per asse (§4.5.6). Questo risultato va però qualificato: la verifica è stata condotta simulando i messaggi da terminale, non generandoli dal movimento di un rig motorizzato reale, indisponibile in laboratorio. La logica software risulta quindi validata, mentre l'anello finale della catena — la generazione dei pacchetti a partire dal movimento fisico — resta da verificare (§6.3).

5.3 Carico manuale eliminato dall'automazione

Il beneficio dell'automazione descritta nel Capitolo 3 non è stato misurato in ore risparmiate; è però possibile quantificarne la portata su una base diversa e indipendente da una misura sul campo: il conteggio delle operazioni manuali che la sincronizzazione della shotlist sostituisce. Si tratta di un'analisi strutturale, ricavata dalla sequenza di operazioni che la funzione `sync_master_sequencer` esegue per ciascuna riga della shotlist (§3.5), non di una stima empirica.

La Tabella 5.1 elenca le operazioni richieste per predisporre manualmente un singolo shot che coinvolga una location e un personaggio — la configurazione più semplice ricorrente nella shotlist reale.

Tabella 5.1: Operazioni manuali sostituite dalla sincronizzazione automatica, per un singolo shot con una location e un personaggio.

#	**Operazione manuale sostituita**
1	Creare la Scene e rinominarla secondo l'identificativo di shot
2	Creare l'oggetto camera e rinominarlo secondo la colonna `Camera`
3	Rinominare il data-block della camera (`Data_<ShotID>`)
4	Impostare la camera come camera attiva della scena
5	Impostare la focale secondo la colonna `Focale`
6	Creare l'oggetto Grease Pencil
7	Parentarlo alla camera e posizionarlo sul piano immagine
8	Linkare e istanziare la collezione della location
9	Convertire l'istanza della location in Library Override
10	Linkare e istanziare la collezione del personaggio
11	Convertire l'istanza del personaggio in Library Override
12	Creare l'Action dedicata per l'armatura del personaggio
13	Accodare la strip `SCENE` nel Sequencer di `MASTER_RENDER`

Tredici operazioni per shot nella configurazione minima, a cui se ne aggiungono tre per ogni personaggio ulteriore presente in scena (istanziazione della collezione, conversione in Library Override e creazione della rispettiva Action) e, una sola

volta per l'intera scena, la disattivazione dell'illuminazione sui layer Grease Pencil e l'aggiornamento del frame finale della scena master.

Due proprietà di questo conteggio sono rilevanti per la valutazione. La prima è che il carico cresce *linearmente* con il numero di shot: è un costo che nella pipeline tradizionale si ripete identico a ogni riga della shotlist, ed è precisamente la classe di lavoro su cui l'automazione produce un beneficio proporzionale alla dimensione della produzione — un punto non marginale nel contesto seriale discusso in §2.1: la sola Scena 1 dell'episodio impiegato come caso di test conta sedici shot (Tabella 3.2).

La seconda è che l'automazione non elimina soltanto tempo, ma una categoria di errore: operazioni come la coerenza tra il nome della Scene e l'identificativo di shot, il valore corretto della focale per ciascuna camera, o la conversione di *ogni* istanza in Library Override sono soggette, se eseguite a mano su decine di shot, a omissioni e incoerenze che si manifestano tardi e che risultano costose da diagnosticare. La sincronizzazione automatica rende queste proprietà garantite per costruzione anziché affidate alla disciplina dell'operatore — un beneficio che non si esprime in ore risparmiate ma in errori non commessi, e che il conteggio della Tabella 5.1 da solo non cattura.

5.4 Valutazione della Set Extension ed ergonomia del set

Al di là del suo ruolo primario come sfondo digitale integrato in tempo reale con la ripresa fisica (Capitolo 4), l'ambiente virtuale sviluppato in Unreal Engine si presta anche a un uso in fase di valutazione preventiva: prima di procedere alla costruzione fisica di un elemento di scenografia, è possibile visualizzarlo nel contesto dell'inquadratura reale all'interno dell'ambiente virtuale, e valutare se quell'elemento rientri effettivamente in campo o a fuoco in un numero di shot sufficiente a giustificarne la realizzazione fisica, oppure se possa restare esclusivamente parte dello sfondo digitale esteso. Questa possibilità — resa concreta dal fatto che gli asset del set sono già modellati in Blender per le necessità della pre-produzione (Capitolo 3) e trasferibili in Unreal con un costo marginale ridotto grazie all'esportazione automatizzata tramite l'addon BFU (§3.9) — introduce nella pipeline digitalizzata una fase di verifica spaziale preventiva assente nel flusso di lavoro tradizionale, dove una simile valutazione emergerebbe solo a costruzione fisica già avvenuta.

Sul piano della resa visiva, il test spaziale condotto (§4.7.1) ha prodotto un esito positivo su una questione che costituiva un rischio concreto per l'intero approccio: alla distanza operativa necessaria, la griglia di pixel dello schermo di proiezione risultava visibile a fotocamera, compromettendo la continuità percettiva tra set fisico e sfondo virtuale. La soluzione individuata — lasciare il rendering virtuale completamente nitido e demandare interamente la sfocatura alla profondità di

campo della fotocamera reale — risolve l'artefatto senza richiedere compromessi sulla qualità dello sfondo e senza introdurre elaborazione aggiuntiva lato motore. È un risultato rilevante perché la visibilità della trama dello schermo a distanza ravvicinata è una condizione intrinseca alla scala ridotta del set stop-motion, non un difetto contingente dell'allestimento utilizzato: una pipeline di Set Extension a scala di modello deve necessariamente affrontarla.

Resta invece non verificata in forma sistematica la questione più propriamente ergonomica — quanto spazio operativo abbia effettivamente l'animatore per accedere al pupazzo con gli strumenti di scena — per le ragioni discusse in §4.7.2 e riprese in §5.6.

5.5 Confronto tra pipeline tradizionale e digitalizzata

Come anticipato in §5.1, un confronto quantitativo rigoroso tra le due pipeline — in termini di tempo risparmiato, costi evitati o numero di iterazioni ridotte — esula dal perimetro di questo lavoro (§5.6).

Un'osservazione emersa nel corso dello sviluppo merita comunque di essere riportata, in quanto conseguenza diretta delle scelte architetturali discusse nei Capitoli 3 e 4 piuttosto che dato isolato: il set fisico, essendo già stato modellato in Blender per le necessità dello storyboard tridimensionale e della pre-produzione (Capitolo 3), ha richiesto un pomeriggio di lavoro — comprensivo della preparazione tecnica degli asset (ottimizzazione della mappatura UV, §3.9) e del trasferimento tramite l'addon BFU (§3.9) — per essere reso disponibile come ambiente di Set Extension funzionante in Unreal Engine.

Il valore di questo dato non sta nella cifra assoluta, ma in ciò che essa verifica sul piano architetturale: l'approccio Single-DCC adottato in pre-produzione (§2.3) poggia sull'assunzione che gli asset costruiti per uno scopo siano riutilizzabili per uno scopo successivo e diverso senza ricostruzione. Il trasferimento osservato conferma che questa assunzione regge nel caso concreto: gli stessi modelli prodotti per la verifica spaziale in Blender sono risultati impiegabili come Set Extension in tempo reale con un adattamento tecnico limitato a poche ore, anziché richiedere una nuova modellazione nel secondo software. Un confronto diretto e misurato con l'alternativa — modellare lo stesso ambiente direttamente in Unreal Engine — non è stato condotto, e resta un limite dichiarato di questa valutazione (§5.6).

Le due forme di ottimizzazione documentate in questo capitolo, pur intervenendo su fasi distinte del flusso, rispondono allo stesso principio: sostituire lavoro ripetuto con lavoro riutilizzato. Nella pre-produzione ciò avviene sostituendo le tredici operazioni per shot della Tabella 5.1 con un'unica compilazione della shotlist; nel

passaggio verso la Virtual Production, riutilizzando gli asset già modellati anziché ricostruirli nel secondo software.

5.6 Criticità incontrate e limiti del lavoro

Questa sezione raccoglie in forma sintetica le criticità tecniche incontrate nello sviluppo, già narrate in dettaglio nei rispettivi capitoli, insieme ai limiti del sistema e della valutazione non ancora risolti al momento della stesura di questa tesi.

Criticità risolte in Blender (Capitolo 3). La lettura del file `shotlist.xlsx` ha richiesto un parser nativo (`read_excel_no_lib`, §3.4) in assenza di librerie esterne nell'ambiente Python di Blender. La conversione delle istanze asset in Library Override non risultava affidabile se eseguita inline shot per shot: la soluzione adottata separa il processo in due passaggi distinti, uno di creazione e uno di consolidamento globale (§3.5).

Criticità risolte in Unreal Engine (Capitolo 4). Il server OSC non riceveva correttamente i pacchetti con indirizzo di ricezione `127.0.0.1` su macOS, risolto impostando `0.0.0.0` (§4.5). Il pannello fisico virtuale `ChromaCard` non copriva l'intero fotogramma per inquadrature ampie, risolto sostituendolo con un materiale Post Process a schermo intero (§4.6). Il cambio di stato del Post Process non aveva il tempo di completarsi prima dello scatto reale dell'esposizione X2, risolto introducendo un ritardo controllato (`time.sleep(3)`) nell'Action Script (§4.6). Il primo materiale feltro sviluppato restituiva una resa più vicina al velluto che alla lana infeltrita, risolto ricostruendo il materiale (`M_NewFeltro`) con un'architettura modulare e base cromatica neutra (§4.2).

Limiti del sistema. Il tracciamento degli assi motion-control resta validato solo in simulazione software, non con l'hardware motorizzato reale, per indisponibilità del modulo in laboratorio (§4.6). La verifica dell'accessibilità del pupazzo con gli strumenti di rigging non è stata condotta, essendo condizionata al completamento della fase di rigging del personaggio (§4.7.2).

Limiti della valutazione. Il confronto tra pipeline tradizionale e digitalizzata non poggia su una raccolta dati sistematica (§5.5); l'analisi del carico manuale eliminato (§5.3) è strutturale e non temporale, e non stabilisce quindi quanto tempo le operazioni sostituite richiederebbero effettivamente; la verifica ergonomica dello spazio operativo per l'animatore è rimasta di natura prevalentemente visiva (§4.7.1). La pipeline, inoltre, è stata validata su un singolo caso di produzione (§3.11). Questi limiti sono ripresi, in quanto direzioni di lavoro, nel Capitolo 6.

Nel complesso, la valutazione conferma la fattibilità tecnica dei due assi sperimentali e circoscrive con precisione ciò che resta da dimostrare: non il funzionamento dei meccanismi, verificato, ma la misura del vantaggio che essi producono rispetto alla pratica tradizionale.

Capitolo 6

Conclusioni e sviluppi futuri

Il capitolo riprende gli obiettivi dichiarati in apertura per verificare in che misura siano stati raggiunti, esplicita i contributi del lavoro svolto e delinea le direzioni di sviluppo che restano aperte.

6.1 Sintesi del lavoro svolto

Il capitolo 1 ha posto la domanda di fondo di questa tesi: fino a che punto sia possibile ridurre il costo della pre-produzione e dell'allestimento scenografico di una serie stop-motion con strumenti digitali che lascino inalterata la ripresa fotogramma per fotogramma del pupazzo reale (§1.1). Da essa discendevano due obiettivi e un obiettivo di natura sperimentale (§1.2), che vengono qui ripresi nell'ordine.

Il primo obiettivo era rendere praticabile lo storyboard tridimensionale in una produzione seriale, eliminandone il costo di allestimento. La pipeline sviluppata nel capitolo 3 lo raggiunge: a partire dalla sola shotlist genera la struttura di scene, camere e asset su cui disegnare lo storyboard, sostituendo per ciascuno shot le operazioni manuali enumerate in §5.3, e rende disponibile la verifica spaziale nel momento stesso in cui l'inquadratura viene composta. Il requisito critico per l'impiego seriale — l'indipendenza delle animazioni tra shot che condividono lo stesso personaggio — è stato verificato nell'ambito della lavorazione effettiva di un episodio (§5.2).

Il secondo obiettivo era estendere digitalmente il set fisico. La Set Extension realizzata nel capitolo 4 ha superato la principale criticità di resa emersa — la visibilità della trama dello schermo alla distanza operativa — con una soluzione che non richiede compromessi sulla qualità dello sfondo (§5.4). A differenza del primo, questo risultato è stato ottenuto in una verifica di fattibilità su un allestimento parziale, privo del rig di animazione del pupazzo e di parte della scenografia (§1.3):

ha quindi il valore di una dimostrazione di principio, più che di un collaudo in condizioni di produzione.

L'obiettivo sperimentale, il collegamento in tempo reale tra DragonFrame e Unreal Engine, è stato raggiunto per fotogramma, shot ed esposizione, e validato end-to-end dallo scatto all'aggiornamento della scena virtuale (§5.2). Resta non verificato con hardware reale, per ragioni di disponibilità e non di progettazione, l'anello del tracciamento motorizzato degli assi.

Rispetto alla domanda iniziale, la risposta che il lavoro è in grado di dare ha dunque una portata precisa. Il vincolo di partenza è rispettato per costruzione: la digitalizzazione interviene sulla pre-produzione e sullo sfondo di scena, mentre l'animazione resta interamente fisica. La tesi dimostra che i meccanismi proposti funzionano e che sostituiscono un carico di lavoro manuale identificabile e proporzionale alla dimensione della produzione; non stabilisce invece di quanto essi riducano il costo complessivo, poiché la valutazione non comprende un confronto misurato con la pratica tradizionale (§5.6). Il "fino a che punto" della domanda trova così una risposta qualitativa e strutturale, mentre la sua misura quantitativa resta una delle direzioni di sviluppo indicate in §6.3.

6.2 Contributi della tesi

Il lavoro produce due contributi di natura distinta.

Contributo tecnico. La letteratura di riferimento (§2.3) individua esplicitamente la possibilità di un collegamento in tempo reale tra DragonFrame e un motore di rendering, senza tuttavia proporne una soluzione tecnica pubblicata [1, 13]. Il capitolo 4 fornisce una risposta concreta e documentata in questo spazio: un'architettura di sincronizzazione basata sul meccanismo nativo di Action Script di DragonFrame e sul protocollo OSC, che copre fotogramma, shot ed esposizione, insieme al pass di chroma key a doppia esposizione che ne sfrutta la struttura. La documentazione delle scelte implementative e dei vincoli incontrati — dal binding del server OSC su macOS alla temporizzazione del cambio di stato del Post Process — è parte del contributo quanto l'architettura stessa, poiché sono precisamente questi dettagli a mancare nei riferimenti disponibili.

Contributo metodologico. Il metodo dichiarato in §1.3 — verificare empiricamente il comportamento dell'interfaccia tra i due software anziché dedurlo dalla documentazione — si è rivelato decisivo per una parte consistente delle criticità incontrate: l'ordine esatto degli argomenti passati dall'Action Script, l'indirizzo IP corretto per il binding del server OSC, l'affidabilità della conversione in Library Override (§3.6), la temporizzazione del pass di chroma key (§4.6), la mappatura effettiva degli assi sui pin della camera virtuale (§4.5.6). Applicato sistematicamente, l'approccio costituisce esso stesso un risultato trasferibile: in un'integrazione

tra software commerciali non progettati per comunicare tra loro, il comportamento reale dell'interfaccia va misurato, non dedotto.

6.3 Sviluppi futuri

Le direzioni di lavoro che restano aperte discendono in gran parte dai limiti dichiarati in §5.6, e sono qui ordinate dalla più immediata alla più ampia.

Validazione con hardware di tracking reale. È lo sviluppo più diretto, e completerebbe il contributo tecnico del capitolo 4. La catena presenta due anelli non ancora verificati, di complessità crescente: il collegamento di un dispositivo di controllo movimento — anche soltanto virtuale — che renda operativo il canale nativo *Output posizioni motori* di DragonFrame, oggi inutilizzabile in sua assenza; e il collegamento di un rig motorizzato fisico (DMC-16/DMC+) che generi i pacchetti OSC a partire dal proprio movimento reale. Essendo la logica software già validata in simulazione su tutti e cinque gli assi, il lavoro residuo riguarda la sorgente dei messaggi, non la loro interpretazione. Il completamento di questo anello porterebbe la sincronizzazione a coprire, oltre a fotogramma, shot ed esposizione, anche il movimento della camera fisica — il traguardo che la letteratura di settore individua come ancora da colmare.

Espansione a ulteriori assi motorizzati. L'architettura a `Switch on String` adottata per lo smistamento dei messaggi (§4.5.2) è stata limitata ai cinque assi rilevanti per l'allestimento disponibile, ma il tutorial da cui il meccanismo è stato adattato [19] ne dimostra l'applicazione a un insieme più ampio, incluso un piatto girevole motorizzato. L'aggiunta di un asse richiederebbe un nuovo ramo dedicato, senza modifiche architetturali di fondo.

Verifica ergonomica con strumentazione dedicata. Un test condotto con strumenti di misura adeguati e su un pupazzo dotato di rig permetterebbe di determinare una distanza schermo-set ottimale motivata da dati anziché da una singola osservazione qualitativa, e di misurare lo spazio operativo effettivamente disponibile per la manipolazione con gli strumenti di scena.

Confronto quantitativo tra pipeline tradizionale e digitalizzata. L'analisi del carico manuale eliminato (§5.3) stabilisce *quante* operazioni la pipeline sostituisce, non quanto tempo esse richiedano. Cronometrare l'esecuzione manuale di quelle stesse operazioni su un campione di shot, contrapponendola alla sincronizzazione automatica, fornirebbe la misura mancante con un onere sperimentale contenuto; una raccolta più estesa, su più episodi e più unità di ripresa, permetterebbe di valutare il beneficio alla scala tipica della produzione seriale.

Validazione su più episodi e produzioni. La pipeline è stata validata su un singolo caso di produzione (§3.11). Estenderne la verifica a episodi con shotlist e asset diversi, e idealmente a una produzione differente, permetterebbe di stabilire

quanto i risultati ottenuti dipendano dalle specificità del caso di studio adottato e quanto siano invece trasferibili.

Bibliografia

[1] Tom Livingstone. «Virtual production at model scale: A case-study of virtual production R&D in Aardman Animations». In: *Convergence: The International Journal of Research into New Media Technologies* (2025). DOI: 10.1177/13548565251394186 (cit. alle pp. 1, 7, 9, 71).

[2] MyWorld. *MyWorld – Creative Technology Programme*. UKRI Strength in Places Fund programme, University of Bristol. URL: https://www.myworld-creates.com/ (cit. a p. 1).

[3] DataIntelo. *Stop Motion Animation Market Research Report 2033*. Rapporto di ricerca di mercato a fonte commerciale. 2025. URL: https://dataintelo.com/report/stop-motion-animation-market (cit. a p. 1).

[4] Growth Market Reports. *Stop Motion Animation Market Research Report 2033*. Rapporto di ricerca di mercato a fonte commerciale. 2025. URL: https://growthmarketreports.com/report/stop-motion-animation-market (cit. a p. 1).

[5] MarketIntelo. *Stop Motion Animation Market Research Report 2034*. Rapporto di ricerca di mercato a fonte commerciale. 2026. URL: https://marketintelo.com/report/stop-motion-animation-market (cit. a p. 1).

[6] Christopher Perry, Sica von Medicus, Abhishek Kumar e Ushma Pandya. «Exploring the Novel Real-Time Pipeline of Nickelodeon's Max & the Midknights». In: *ACM SIGGRAPH 2025 Production Sessions*. ACM, 2025. DOI: 10.1145/3698897.3718367 (cit. a p. 2).

[7] DZED Systems LLC. *DragonFrame Documentation*. Version 2026.07. 2026. URL: https://www.dragonframe.com/manual/ (cit. a p. 6).

[8] Susannah Shaw. *Stop Motion: Craft Skills for Model Animation*. 3rd. Boca Raton: Routledge / CRC Press, 2017. ISBN: 978-1-138-77931-0 (cit. a p. 7).

[9] Blender Foundation. *Blender Python API Documentation*. URL: https://docs.blender.org/api/current/ (cit. alle pp. 8, 15, 25).

[10] T. Chlubna, M. Vlnas, T. Milet e P. Zemčík. «Survey of FOSS 3D/2D graphics software Blender usage in science, academia, and industry». In: *The Visual Computer* 42.1 (2025). DOI: 10.1007/s00371-025-04281-1 (cit. a p. 9).

[11] Oleksandra Sobchyshak, Santiago Berrezueta-Guzman e Stefan Wagner. *Pushing the Boundaries of Immersion and Storytelling: A Technical Review of Unreal Engine*. ScienceDirect Article. 2025. arXiv: 2507.08142 (cit. a p. 9).

[12] Epic Games. *Unreal Engine 5.7 Documentation*. URL: https://dev.epicgames.com/documentation/en-us/unreal-engine/ (cit. alle pp. 9, 41).

[13] David Gray, Eve Bolotova, Jonah Ling e Leon Cresdee. «Digital Meets Handmade: Can VP enhance stop-motion?» In: *SIGGRAPH Talks '25: Proceedings of the Special Interest Group on Computer Graphics and Interactive Techniques Conference Talks*. ACM, 2025, pp. 1–3. DOI: 10.1145/3721239.3734092 (cit. alle pp. 9, 71).

[14] Jon Swords e Nina Willment. «The emergence of virtual production – a research agenda». In: *Convergence: The International Journal of Research into New Media Technologies* 30.5 (2024), pp. 1557–1574. DOI: 10.1177/13548565241253903 (cit. a p. 10).

[15] Autodesk. *Universal Scene Description | What is OpenUSD?* URL: https://www.autodesk.com/solutions/media-entertainment/openusd (cit. a p. 10).

[16] Julien Blervaque e Ubisoft Animation Studio. *Shot Manager*. Blender add-on, sviluppato per la serie animata Rabbids Invasion (2020). URL: https://github.com/ubisoft/shotmanager (cit. a p. 10).

[17] Xavier Loux. *Blender For UnrealEngine (BFU)*. Add-on open-source. URL: https://github.com/xavier150/Blender-For-UnrealEngine-Addons (cit. a p. 30).

[18] Pitchfork Academy. *How To Make A Stylized Skybox - Unreal Engine 5 Tutorial*. Video YouTube - 5 set 2025. URL: https://youtu.be/MnriY-z72EI (cit. a p. 38).

[19] Biolapse Studios. *Dragonframe Unreal Integration*. Video YouTube - 27 lug 2025. URL: https://youtu.be/XeDS2Ac7luY (cit. alle pp. 52, 72).